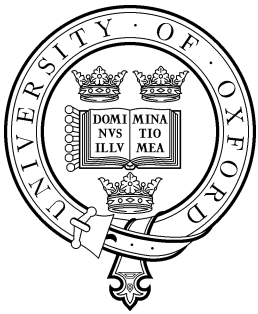
Stream Fusion: Practical shortcut fusion for coinductive sequence types



Duncan Coutts

Worcester College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Michaelmas 2010

Abstract

In functional programming it is common practice to build modular programs by composing functions where the intermediate values are data structures such as lists or arrays. A desirable optimisation for programs written in this style is to fuse the composed functions and thereby eliminate the intermediate data structures and their associated runtime costs.

Stream fusion is one such fusion optimisation that can eliminate intermediate data structures, including lists, arrays and other abstract data types that can be viewed as coinductive sequences. The fusion transformation can be applied fully automatically by a general purpose optimising compiler. The stream fusion technique itself has been presented previously and many practical implementations exist. The primary contributions of this thesis address the issues of correctness and optimisation: whether the transformation is correct and whether the transformation is an optimisation.

Proofs of shortcut fusion laws have typically relied on parametricity by making use of free theorems. Unfortunately, most functional programming languages have semantics for which classical free theorems do not hold unconditionally; additional side conditions are required. In this thesis we take an approach based not on parametricity but on data abstraction. Using this approach we prove the correctness of stream fusion for lists – encompassing the fusion system as a whole, not merely the central fusion law. We generalise this proof to give a framework for proving the correctness of stream fusion for any abstract data type that can be viewed as a coinductive sequence and give as an instance of the framework, a simple model of arrays. The framework requires that each fusible function satisfies a simple data abstraction property. We give proofs of this property for several standard list functions.

Previous empirical work has demonstrated that stream fusion can be an optimisation in many cases. In this thesis we take a more universal view and consider the issue of optimisation independently

of any particular implementation or compiler. We make a semi-formal argument that, subject to certain syntactic conditions on fusible functions, stream fusion on lists is strictly an improvement, as measured by the number of allocations of data constructors. This detailed analysis of how stream fusion works may be of use in writing fusible functions or in developing new implementations of stream fusion.

Contents

Acknowledgements

Foremost I must thank my D.Phil supervisor, Professor Oege de Moor, for being patient, for encouraging me to publish and especially for helping me through a change in thesis topic.

I would also like to thank my co-conspirators Don Stewart and Roman Leshchinskiy for their personal and intellectual enthusiasm which made collaboration such a joy. I must also thank Manuel Chakravarty for inviting me to visit UNSW for six weeks to work with Don and Roman on our second stream fusion paper.

Don, Roman and I are most grateful to Simon Peyton Jones and Simon Marlow for their help on numerous occasions and for many productive discussions. In particular we are indebted to Simon Peyton Jones for clarifying and extending GHC's optimiser, which greatly assisted our research project.

Thanks to my examiners Ralf Hinze and Andy Gill for ploughing through all the details of what has turned out to be a rather long thesis and for the interesting discussion and feedback during the viva. Thanks to Ralf Hinze and Jeremy Gibbons for their detailed comments and feedback on a draft of this thesis during my confirmation of status. Thanks also to Ian Lynagh, Tom Harper, Peter Jonsson and Lennart Kolmodin for feedback on drafts of various chapters.

My research was supported for five years by the Oxford University Computing Laboratory[1] via a teaching assistant studentship. I am thankful that the Comlab offered me this comparatively long period. It has proved to be very valuable, not just in allowing time for a successful research conclusion after a change in thesis topic but in allowing time to develop practical skills which now enable me to apply functional programming in the commercial realm.

I was honoured to have Tom Harper choose to have me advise him on his MSc project into an application of stream fusion.

[1] Recently renamed to the Department of Computer Science, University of Oxford.

Thanks to Ralf Hinze and Andres Löh for creating the excellent `lhs2TeX` system which makes typesetting Haskell code so easy. Thanks particularly to Andres for advice and code snippets to customise `lhs2TeX`.

Thanks to my business partner Ian Lynagh for being patient during the many months while I was writing up.

Last but not least, thanks to my partner Elizabeth Baldwin for her support and for putting up with the many late nights I spent writing.

Duncan Coutts, Trinity term 2011

Statement of Originality

I am the sole author of the text of this thesis. Much of the research on which this thesis is based was done in collaboration with others. In particular much of the practical work has previously been reported in two published papers (Coutts et al., 2007a,b) which were jointly authored by Don Stewart, Roman Leshchinskiy and myself.

Duncan Coutts, Michaelmas term 2010

Chapter 1

Introduction

1.1 Aims

This thesis is about a technique for making beautiful programs run fast.

Programmers and computer scientists like beautiful programs. We also like fast programs. These qualities are often in conflict in the design of programs and we are often forced to sacrifice one to achieve the other.

For decades computer scientists have looked for techniques to make real progress in this tradeoff. One popular approach has been to *derive* fast programs from beautiful programs. This has sometimes taken the form of calculation methods performed by hand. Wherever possible people have tried to design automated methods, often with a view to include them as optimisation passes in a compiler.

This thesis is a contribution to the work on deriving fast programs from beautiful programs. It is, of course, not a contribution of universal applicability to all programs in all programming languages. It is about a particular automated technique – stream fusion – that works for a particular class of programs in a particular class of programming languages.

Specifically, the technique applies in the context of functional programming languages. It applies to programs where a sequence data structure is produced as the result of one function and consumed immediately by another function. Its application is restricted by the condition that the producing and consuming functions be written in terms of special named combinators. Where the technique applies, it transforms the composition of the producing and consuming functions into a single *fused* function. This fused function performs fewer memory allocations at runtime and as a consequence usually runs faster.

By way of example, consider the following small program in the functional programming language Haskell (Peyton Jones et al., 2003), which calculates the sum of the squares of a list of numbers

$$sumSq\ xs = sum\ (map\ sq\ xs)$$

It uses the following auxiliary definitions[1]

$$\begin{aligned} sum\ [\,] &= 0 \\ sum\ (x : xs) &= x + sum\ xs \\ map\ f\ [\,] &= [\,] \\ map\ f\ (x : xs) &= f\ x : map\ f\ xs \\ sq\ x &= x \times x \end{aligned}$$

This style of programming, using lazy lists as the glue to connect reusable functions, is widely regarded as powerful and elegant[2]. It is undeniable however that using many intermediate lists slows down the execution of a program. During the course of evaluating, say $sumSq\ [2]$, we will construct the intermediate list $[sq\ 2]$ before summing the result. This is unfortunate since we can write a direct definition of *sumSq* that avoids constructing any intermediate list and thus executes faster and uses less memory

$$\begin{aligned} sumSq\ [\,] &= 0 \\ sumSq\ (x : xs) &= x \times x + sumSq\ xs \end{aligned}$$

On the other hand, this version sacrifices the clarity and modularity of the original. What we really want is to write the first version and have the compiler automatically derive the second version.

Using the stream fusion technique we can achieve exactly that. If we redefine the standard list functions *map*, *sum* et cetera in terms of the special stream fusion combinators then we can automatically transform the original beautiful program into the equivalent fast program.

[1]The astute reader will notice that the given definition of *sum* does not make use of an accumulating parameter. We use this definition because it simplifies the initial presentation. A definition using an accumulating parameter poses no problem for stream fusion, though it does for one of the other techniques that we will consider in this chapter.

[2]Hughes (1989) presents the arguments for this style.

1.2 Synopsis

This thesis makes two main claims.

1. Stream fusion is a correct program transformation.
2. Stream fusion is an optimisation. Specifically the claim is that in programs where stream fusion applies, the transformed program performs strictly fewer data constructor allocations during evaluation than does the original program.

Chapters 3 and 4 are dedicated to substantiating these claims and they form the nub of the thesis.

The rest of this thesis is organised as follows:

- The remainder of this chapter provides some context by introducing existing related fusion techniques. It goes on to introduce stream fusion and to show where it fits relative to the existing techniques.
- Chapter 2 covers some technical preliminaries which are assumed in later chapters. Readers familiar with Haskell and the semantics of functional languages (in particular the semantics of recursively defined data types) may prefer to skip this chapter.
- Chapter 3 looks at why we should believe stream fusion to be correct.
- Chapter 4 looks at why we expect stream fusion to be an optimisation.
- Chapter 5 compares stream fusion to other related work, though the most closely related work is covered in the current chapter.
- Chapter 6 concludes and considers further work.

Parts of the research presented here have been reported on in previous papers (Coutts et al., 2007a,b). These papers need not be considered as prerequisites however. While we do make occasional reference to related material in these papers, the directly relevant material is covered in this chapter.

Much of the early research on stream fusion was conducted using the *ByteString* library. The details of the *ByteString* implementation are available as a separate technical report (Coutts, 2010).

1.3 Research context

Stream fusion is not unique in its ability to perform the optimisation we have seen on the *sumSq* example. There are other techniques, including ones we label as 'fusion' or 'shortcut fusion', that can achieve the same result. The differences between the various techniques lie in the range of programs to which they apply, how effective they are in the cases where they do apply and whether or not they can be automated.

To understand why stream fusion is an interesting fusion technique it helps to put it in the context of previous research. The following sections are not a comprehensive look at related work, but an introduction to a selection of previous work that has directly or indirectly motivated our own. It should help to explain how we got to the starting point for this research and why we headed in the direction we did.

In looking back at the previous research we will notice that the trend has been of increasing automation and decreasing generality. The early techniques are applicable to a wide range of programs but are primarily manual and have some tricky side-conditions to verify. The later techniques have simple side-conditions and can be automated; however, they apply only to programs in particular special forms. The research described in this thesis is no exception to the trend.

1.3.1 The unfold / fold framework

A significant early paper by Burstall and Darlington (1977) describes a calculation method to transform "lucid" programs into more efficient ones. It works on programs written as recursive equations – what we would now describe as a functional style. An interesting point made in this paper is that the functional style is better suited to transformation than the "usual Algol" or imperative style. The rest of the works we will consider (including this thesis) are concerned with programs written in a functional style.

The Burstall and Darlington paper describes a system of transformation rules, each of which preserves the meaning of a program. By cunning application of the rules, a programmer may transform a lucid program into a faster one. Indeed it can improve the asymptotic complexity and not just the constant factors. It is a calculation method and not an automatic algorithm. The decisions about which rules to apply (and when to apply them) are up to the programmer, though there are strategies that produce good results in many cases.

It is commonly referred to as the 'fold-unfold' transformation technique. In this context 'unfolding' means substituting named parametrised terms for their definitions while 'folding' is the reverse transformation of substituting an instance of a definition for the named term[3]. The essence of the 'fold-unfold' method is as follows. It starts by unfolding and simplifying the original recursive definitions. The crucial step is to look in the resulting terms for instances of the original definitions and to fold those instances back into new recursive definitions.

The Fibonacci example

They use the Fibonacci function as an example. We will look at this example because it illustrates the distinguishing features of their technique. In particular, with this example, the technique is able to achieve an improvement in the asymptotic complexity. However, because the subsequent techniques that we will consider cannot change the asymptotic complexity, we will also look at a second example that is somewhat simpler.

We start with the original recursive definition, written as a set of equations. Under a standard evaluation model, $f\ n$ will take $\Omega(2^n)$ time to evaluate.

$$\begin{aligned} f\ 0 &= 0 \\ f\ 1 &= 1 \\ f\ (n+2) &= f\ (n+1) + f\ n \end{aligned}$$

We then have a "eureka" step, where we invent an auxiliary definition based on our original definition.

$$g\ n = (f\ (n+1), f\ n)$$

We instantiate this auxiliary definition for the 0 and $(n+1)$ cases, each time unfolding calls to f wherever possible

$$\begin{aligned} g\ 0 &= (f\ 1, f\ 0) \\ &= (1, \ \ 0) \\ g\ (n+1) &= (f\ (n+2), \quad f\ (n+1)) \\ &= (f\ (n+1) + f\ n, f\ (n+1)) \end{aligned}$$

[3]The use of the terms fold and unfold in this context should not be confused with the common functions *fold* and *unfold*.

Then we have another tricky step. We abstract over $f\ (n+1)$ and $f\ n$, replacing them with variables u and v. We notice we have an instance of a call to g and we fold it back into an actual application of g.

$$\begin{aligned} g\ (n+1) &= (u+v,u)\ \textbf{where}\ (u,v) = (f\ (n+1),f\ n) \\ &= (u+v,u)\ \textbf{where}\ (u,v) = g\ n \end{aligned}$$

We do something similar for $f\ (n+2)$, abstracting over $f\ (n+1)$ and $f\ n$, replacing them with variables u and v and then folding into an application of g.

$$\begin{aligned} f\ (n+2) &= u+v\ \textbf{where}\ (u,v) = (f\ (n+1),f\ n) \\ &= u+v\ \textbf{where}\ (u,v) = g\ n \end{aligned}$$

Our final equations are

$$\begin{aligned} f\ 0 &= 0 \\ f\ 1 &= 1 \\ f\ (n+2) &= u+v \qquad \textbf{where}\ (u,v) = g\ n \\ g\ 0 &= (1,0) \\ g\ (n+1) &= (u+v,u)\ \textbf{where}\ (u,v) = g\ n \end{aligned}$$

We now have a different pattern of recursion and have improved the running time from exponential to linear – a dramatic improvement. All the later techniques we will consider can only make constant factor improvements to the running time or space use. This example also illustrates that there are several points in the derivation that are not just mechanical unfolding and substitution. We have to invent suitable auxiliary definitions and must also choose what to abstract over.

This kind of exercise is great for exam questions but is less useful for real programming. Burstall and Darlington provide a strategy for applying the rules and they implemented a system that performs many of the steps automatically, including the unfolding, abstraction and folding steps. The algorithm still requires any auxiliary definitions, such as g in the Fibonacci example, to be supplied as input.

A further issue is that, applied blindly, the technique does not necessarily preserve termination. For example as a final step in the Fibonacci example we could make use of the fact that $g\ n = (f\ (n+1),f\ n)$ and thus $f\ n = snd\ (g\ n)$ to get a somewhat shorter result

$$\begin{aligned} f\ n &= snd\ (g\ n) \\ g\ 0 &= (1,0) \\ g\ (n+1) &= (u+v,u)\ \textbf{where}\ (u,v) = g\ n \end{aligned}$$

However if we were to have applied the fact $f\ n = snd\ (g\ n)$ at an earlier stage of the derivation we could have ended up with

$$\begin{aligned} f\ n &= snd\ (g\ n) \\ g\ n &= (f\ (n+1), f\ n) \end{aligned}$$

This is tantamount to stating that $f\ n = f\ n$ which is a true statement but is less than helpful. It is only helpful to redefine f once we have manipulated g to the point where it is defined in terms of itself rather than in terms of f.

The *sumSq* example

To make fair comparisons with the techniques we will consider next, we must turn to a simpler example that does not involve asymptotic improvements. We return to the introductory example of optimising a function that calculates the sum of the squares of a list of numbers. We start with the "lucid" definitions:

$$\begin{aligned} sumSq\ xs &= sum\ (map\ sq\ xs) \\ sum\ [\,] &= 0 \\ sum\ (x : xs) &= x + sum\ xs \\ map\ f\ [\,] &= [\,] \\ map\ f\ (x : xs) &= f\ x : map\ f\ xs \\ sq\ x &= x \times x \end{aligned}$$

To apply the method of Burstall and Darlington in the *sumSq* example we do not need to invent any auxiliary definitions. We can instantiate *sumSq* in the $[\,]$ and $(x : xs)$ cases and then get straight on with unfolding definitions.

$$\begin{aligned} sumSq\ [\,] &= sum\ (map\ sq\ [\,]) \\ &= sum\ [\,] \\ &= 0 \\ sumSq\ (x : xs) &= sum\ (map\ sq\ (x : xs)) \\ &= sum\ (sq\ x : map\ sq\ xs) \\ &= sq\ x + sum\ (map\ sq\ xs) \\ &= x \times x + sum\ (map\ sq\ xs) \end{aligned}$$

In the $(x : xs)$ case we spot a simple instance of *sumSq* which we can fold back into a call.

$$= x \times x + sumSq\ xs$$

So we arrive at our final definition:

$$\begin{aligned} sumSq\ [\,] &= 0 \\ sumSq\ (x : xs) &= x \times x + sumSq\ xs \end{aligned}$$

With this simpler example the improvements are much less dramatic than in the Fibonacci example but we did not need any imagination, the whole thing was essentially mechanical. The question arising out of these examples is: can we characterise precisely the subset of definitions that we can optimise mechanically; that is, without needing a "eureka" step.

Much research subsequent to that of Burstall and Darlington went into finding restrictions or improvements to enable a fully automatic algorithm. As in the *sumSq* example, an important special case is that of removing intermediate data structures in compositions of functions. There is a line of research taking this approach. The goal remains to generate fast programs from elegant programs, but for the common special case of expressions of the form $f\ (g\ x)$ where the intermediate type is a complicated data structure. In a whole program there may be many instances of this form. This is true especially in programming styles that emphasise building programs by composing simpler re-usable functions.

1.3.2 Wadler's deforestation algorithm

Wadler (1990b) proposed an algorithm to deal automatically with a further special case: compositions of functions that produce and consume trees that are themselves defined in a special form called "treeless form". The algorithm guarantees that if both functions are in treeless form then the resulting function will also be in treeless form. Furthermore it guarantees that the number of allocations cannot increase. The intention of course is that the number of allocations actually decrease and indeed in many examples they do decrease.

The main drawback of Wadler's deforestation algorithm is the limited class of programs to which it applies. The treeless form is first order, it requires that variables be used linearly and requires that there be no other intermediate data structures. Naïve generalisations of the algorithm lead to non-termination in certain cases by performing infinite sequences of unfoldings. Wadler suggests several generalisations. One lifts the linearity restriction at least for atomic non-tree types by introducing **let** bindings. He also suggests handling higher order functions by "macro expansion" into first order programs. This idea was extended by Marlow and Wadler (1993).

1.3.3 Shortcut fusion

The deforestation algorithm did not appear to be particularly successful in practice. It has not been included in a release of any compiler, though a prototype was implemented (Davis, 1987) in the LML compiler. One approach to obtaining a more practical algorithm was to take yet another special case. Instead of considering general recursive definitions, the idea is to only attempt to improve definitions written in terms of particular designated functions – fusion combinators. This approach is named *shortcut* deforestation or shortcut fusion. Shortcut fusion has primarily been applied to eliminate lists as intermediate data structures, though approaches exist to tackle trees and arrays.

The key idea of shortcut fusion is that we use equations involving the fusion combinators as local rewrite rules. When we compose functions that are written in terms of the fusion combinators it becomes possible to rewrite the composition to a fused function which does not use any intermediate data structure.

Different shortcut fusion systems use different fusion combinators and associated equations as rewrite rules. We can compare shortcut fusion systems by the range of functions that may be expressed in terms of their fusion combinators and also the quality of the optimisation when the fusion rules apply.

As an example, let us invent a trivial and inexpressive fusion system that we will call '*map*/*map*'. It uses a single fusion combinator *map* and employs the following equation as a local rewrite rule

Fusion rule (*map*/*map*)**.**

$$\forall f\ g.\ map\ f \circ map\ g = map\ (f \circ g)$$

We can apply this system by looking in a source program and wherever we find instances of the rule's left hand side we replace them with the right hand side. This works as an algorithm because the equation is correct and provided that we only use the equation as a rewrite rule in a left to right direction then as a rewrite system it is terminating. What is more, it is an optimisation: we can see that when the rule does apply it does remove intermediate allocations.

Generalising from the '*map*/*map*' example, our obligations as designers of shortcut fusion systems are these:

- we must prove that the equations we use are correct;
- we must show that when the equations are used as rewrite rules, that the rewrite system is terminating;

- we should show that the system is an optimisation.

Showing that the rewrite system is terminating is usually straightforward because in most systems the number of occurrences of some combinator decreases with each rule application.

The fact that the shortcut fusion approach only uses local transformations is a key advantage in comparison to the more general deforestation algorithm. Assuming we can prove the equations we use, then total correctness of the shortcut fusion algorithm is fairly straightforward and does not involve any restrictive side conditions on the form of definitions.

Using local transformations also makes it relatively easy to integrate into the optimisation phase of a compiler. The first shortcut fusion system by Gill et al. (1993) was implemented directly in the Glasgow Haskell Compiler (GHC Team, 2010). A later development by Peyton Jones et al. (2001) was the introduction of a rules language to allow such rewrite rules to be written in a source module and then applied automatically during compilation. This innovation greatly helped later research into similar fusion approaches by making it easy to try experiments. In the concrete syntax of the GHC rules language, the *map/map* rule is written as

```
{-# RULES "map/map"  forall f g.  map f . map g
                                = map (f . g)  #-}
```

Since each equation that is used as a local transformation rule must be proved, we wish to minimise the number of such rules while at the same time maximising the number of functions that we can express in terms of the fusion combinators.

Naïve generalisations of the *map/map* system, e.g. taking in *filter* et cetera, lead to a quadratic explosion in the number of equations required. It is crucial to the design of a shortcut fusion system to find a small set of combinators which are able to express a large class of functions. Intuitively, these combinators should capture the patterns used to construct, consume or transform the intermediate data structures of interest.

1.3.4 Ad hoc shortcut fusion

A somewhat scalable approach is to take a collection of functions of interest and find a smaller set of more general functions that can express them. Continuing the trivial *map/map* example, if we just added *filter* as a combinator then we

would need equations for all four pairings[4] of *map* and *filter*. If on the other hand we define a function *mapFilter* that is capable of expressing both then we would need only one equation of the form

Fusion rule (*mapFilter*/*mapFilter*)**.**

$$\begin{aligned}\forall f\ g\ p\ q.\quad & mapFilter\ f\ p \circ mapFilter\ g\ q \\ & = mapFilter\ (f \circ g)\ (\lambda x \to q\ x \wedge p\ (g\ x))\end{aligned}$$

There is quite a wide scope for fusion systems of this sort. The choice of fusion combinators is inevitably rather ad hoc, chosen to cover a particular range of functions. There is also the danger of excessive complexity caused by trying to find unnatural generalisations of several basis functions. While there is no great theoretical elegance to such systems it is quite possible to build practical implementations that cover a useful range of functions.

1.3.5 Functional array fusion

An enlightening example system of this sort is *functional array fusion* developed by Chakravarty and Keller (2001, 2003). As the name suggests this system was designed specifically for array code. The choice of fusion combinators is driven by the typical range of array operations and array access patterns. The primary fusion combinator *loop* is essentially the generalisation of *filter*, *scanl* and *foldl* on arrays. The *loop* combinator captures left-to-right array traversals:

$$loop :: (a \to s \to (Maybe\ a, s)) \to s \to Array\ a \to (Array\ a, s)$$

The result of a traversal depends on the supplied 'stepper' function which, given a state and an array element, produces a new state and, optionally, a new element. Note that *loop* transforms an array into another array and thus a second combinator is required to construct any initial array. The *replicate* combinator constructs an array consisting of n copies of a given element value

$$replicate :: Int \to a \to Array\ a$$

The system uses two equations as fusion rules, a main one that relates *loop* with itself and one that relates *loop* with *replicate*.

[4]We would also need an extra function like *mapFilter* to express the result since the composition of a *map* and a *filter* is neither a *map* nor a *filter*.

Fusion rule (*loop*/*loop*)**.** The main fusion rule combines adjacent loops by suitably composing the stepper functions

$$\forall f\ s\ g\ t.\ loop\ g\ t \circ fst \circ loop\ f\ s = loopSndAcc \circ loop\ (fuseStep\ f\ g)\ (s,t)$$

The auxiliary function *loopSndAcc* just selects out the appropriate components of the result

$$loopSndAcc\ (xs,(s,t)) = (xs,t)$$

while *fuseStep* lifts a pair of stepper functions into a stepper function on pairs

$$\begin{array}{l} fuseStep\ f\ g\ x\ (a,b) = \\ \quad \textbf{case}\ f\ x\ a\ \textbf{of} \\ \qquad (Nothing, a') \to (Nothing, (a', b)) \\ \qquad (Just\ x',\ \ a') \to \textbf{case}\ g\ x'\ b\ \textbf{of} \\ \qquad\qquad\qquad\qquad (Nothing, b') \to (Nothing, (a', b')) \\ \qquad\qquad\qquad\qquad (Just\ x'',\ \ b') \to (Just\ x'',\ \ (a', b')) \end{array}$$

The *sumSq* **example**

Let us look at the *sumSq* example in this system. Of course this time we have to interpret it over arrays rather than over lists

$$\begin{array}{l} sumSq :: Array\ Int \to Int \\ sumSq\ xs = sum\ (map\ sq\ xs) \end{array}$$

We also must define *map* and *sum* in terms of the fusion combinator *loop*

$$\begin{array}{l} map\ f = fst\ \ \circ loop\ (stepMap\ f)\ () \\ sum\ \ \ = snd \circ loop\ stepSum\ \ \ \ 0 \\ \\ stepMap\ f\ x\ () = (Just\ (f\ x), ()) \\ stepSum\ \ \ x\ a\ \ = (Nothing,\ \ x + a) \end{array}$$

Now we can unfold and apply the fusion rule

$$\begin{array}{ll} & sum\ (map\ sq\ xs) \\ = & \quad \{\ \text{unfold the definitions of } sum \text{ and } map\ \} \\ & snd\ (loop\ stepSum\ 0\ (fst\ (loop\ (stepMap\ sq)\ ()\ xs))) \\ = & \quad \{\ \text{apply } loop/loop \text{ rule}\ \} \\ & snd\ (loopSndAcc\ (loop\ (fuseStep\ (stepMap\ sq)\ stepSum)\ ((),0)\ xs)) \end{array}$$

We are left with a single loop, albeit one with a somewhat complex state and stepper function.

The combined stepper function can be simplified:

$$
\begin{aligned}
& \mathit{fuseStep}\ (\mathit{stepMap}\ sq)\ \mathit{stepSum}\ x\ (a, b) \\
= \quad & \{\ \text{unfold the definition of } \mathit{fuseStep}\ \} \\
& \textbf{case}\ \mathit{stepMap}\ sq\ x\ a\ \textbf{of} \\
& \quad (\mathit{Nothing}, a') \to (\mathit{Nothing}, (a', b)) \\
& \quad (\mathit{Just}\ x',\ \ a') \to \textbf{case}\ \mathit{stepSum}\ b\ x'\ \textbf{of} \\
& \qquad\qquad\qquad\qquad (\mathit{Nothing}, b') \to (\mathit{Nothing}, (a', b')) \\
& \qquad\qquad\qquad\qquad (\mathit{Just}\ x'',\ \ b') \to (\mathit{Just}\ x'',\ \ (a', b')) \\
= \quad & \{\ \text{unfold definitions of } \mathit{stepMap} \text{ and } \mathit{stepSum}\ \} \\
& \textbf{case}\ (\mathit{Just}\ (sq\ x), ())\ \textbf{of} \\
& \quad (\mathit{Nothing}, a') \to (\mathit{Nothing}, (a', b)) \\
& \quad (\mathit{Just}\ x',\ \ a') \to \textbf{case}\ (\mathit{Nothing}, x' + b)\ \textbf{of} \\
& \qquad\qquad\qquad\qquad (\mathit{Nothing}, b') \to (\mathit{Nothing}, (a', b')) \\
& \qquad\qquad\qquad\qquad (\mathit{Just}\ x'',\ \ b') \to (\mathit{Just}\ x'',\ \ (a', b')) \\
= \quad & \{\ \text{case reduction}\ \} \\
& (\mathit{Nothing}, ((), sq\ x + b))
\end{aligned}
$$

So while we have certainly eliminated the allocation of an intermediate array, the final code is not simple. In particular the fused stepper function has quite a bit of redundant code. All it is doing is accumulating a value over the array yet it has to produce a *Nothing* and a $()$ at each step. It is a tall order to expect the compiler to completely eliminate all of this. Indeed, it is clear that current compilers cannot do so (See Coutts et al., 2007a, Section 5).

This system works well enough for standard array algorithms such as *map*, *filter*, *foldl* and *scanl*. It can be extended to support operations that process arrays from right to left. The system can be further extended to handle operations that consume multiple arrays by adding an additional fusion combinator *zip* and rules relating it to *loop* and *replicate*. A real limitation however is that the array transformer *loop* can never produce an array that is longer than its input array; this makes it impossible to express functions such as *concatMap*.

1.3.6 Natural shortcut fusion

The next two systems we will consider are *foldr/build* fusion and *unbuild/unfoldr* fusion. These systems are the primary reference points for the work on stream fusion and we will make frequent comparisons to them in the remaining chapters.

Unlike functional array fusion they both occupy natural points in the design space for shortcut fusion. In their simplest formulation both system use just two fusion combinators and a single equation as a fusion rule. What makes them natural is that instead of an ad hoc choice of fusion combinators they use combinators which are based on the recursive structure of data itself. There are two standard ways of constructing recursive data types, called *data* and the *co-data*, and correspondingly there are two natural shortcut fusion systems.

1.3.7 The *foldr/build* fusion system

The first shortcut fusion system, indeed the one for which the term was coined, is the *foldr/build* system (Gill et al., 1993; Gill, 1996). As the name suggests it is based on the fusion combinators *foldr* and *build*:

$$
\begin{aligned}
&\mathit{foldr} :: (a \to b \to b) \to b \to [a] \to b \\
&\mathit{build} :: (\forall b.(a \to b \to b) \to b \to b) \to [a]
\end{aligned}
$$

The *foldr* function is familiar to all functional programmers as

$$
\begin{aligned}
&\mathit{foldr}\ f\ z\ [\,] &&= z \\
&\mathit{foldr}\ f\ z\ (x : xs) &&= f\ x\ (\mathit{foldr}\ f\ z\ xs)
\end{aligned}
$$

It embodies a general pattern of recursion on lists. It can express a huge range of common functions on lists including

$$
\begin{aligned}
&\mathit{sum} && = \mathit{foldr}\ (+)\ 0 \\
&xs +\!\!+ ys && = \mathit{foldr}\ (:)\ ys\ xs \\
&\mathit{map}\ f && = \mathit{foldr}\ (\lambda x\ xs \to f\ x : xs)\ [\,] \\
&\mathit{filter}\ p && = \mathit{foldr}\ (\lambda x\ xs \to \mathbf{if}\ p\ x\ \mathbf{then}\ x : xs\ \mathbf{else}\ xs)\ [\,] \\
&\mathit{foldl}\ f\ v\ xs && = \mathit{foldr}\ (\lambda x\ g \to (\lambda a \to g\ (f\ a\ x)))\ \mathit{id}\ xs\ v \\
&\mathit{dropWhile}\ p && = \mathit{fst} \circ \mathit{foldr}\ f\ ([\,], [\,]) \\
&\quad \mathbf{where} \\
&\qquad f\ x\ (ys, xs) = (\mathbf{if}\ p\ x\ \mathbf{then}\ ys\ \mathbf{else}\ x : xs, x : xs)
\end{aligned}
$$

Hutton (1999) covers these and other examples. The last two are particularly interesting and somewhat surprising. They show how using higher order or pair types can extend the expressiveness beyond the simple recursive patterns that people typically associate with *foldr*.

The function *build* is not common and its purpose is less immediately obvious

$$
\begin{aligned}
&\mathit{build} :: (\forall b.(a \to b \to b) \to b \to b) \to [a] \\
&\mathit{build}\ g = g\ (:)\ [\,]
\end{aligned}
$$

While *foldr* is for consuming lists, *build* can be used to construct them. For example the list $[1,2,3]$ can be written as

$$
\begin{array}{l}
\mathit{build}\ l \\
\quad \textbf{where} \\
\quad\quad l :: \forall b.\ (\mathit{Int} \to b \to b) \to b \to b \\
\quad\quad l\ \mathit{cons}\ \mathit{nil} = 1\ `\mathit{cons}`\ (2\ `\mathit{cons}`\ (3\ `\mathit{cons}`\ \mathit{nil}))
\end{array}
$$

The application of *build* to the function l gives us the list we first thought of by substituting $(:)$ in place of *cons* and $[\,]$ in place of *nil*. While we may be familiar with using *foldr* to write list consumers, this trick with *build* is certainly not a common way to write lists producers.

Fusion rule (*foldr*/*build*)**.** The key equation of the *foldr*/*build* system is

$$\forall f\ g\ z.\ \mathit{foldr}\ f\ z\ (\mathit{build}\ g) = g\ f\ z$$

On the left hand side, we have *build* producing a list and *foldr* immediately consuming it. The right hand side does not mention lists at all. Thus if we can express our list functions in terms of *foldr* and *build* then we can use this equation as a local rewrite rule to eliminate intermediate lists.

The *sumSq* example

Let us see the system in action with the *sumSq* example:

$$
\begin{array}{l}
\mathit{sumSq}\ xs = \mathit{sum}\ (\mathit{map}\ \mathit{sq}\ xs) \\
\mathit{sq}\ x = x \times x
\end{array}
$$

We require that the functions *map* and *sum* are rewritten in terms of *foldr* and *build*.

$$
\begin{array}{l}
\mathit{sum}\ \ = \mathit{foldr}\ (+)\ 0 \\
\mathit{map}\ f = \mathit{foldr}\ (\lambda x\ xs \to f\ x : xs)\ [\,]
\end{array}
$$

Note that with this definition of *map*, the result list is built explicitly using $(:)$. However, for us to apply the *foldr*/*build* rule to $\mathit{sum}\ (\mathit{map}\ \mathit{sq}\ xs)$, the *map* function must be defined in terms of *build*. The derivation is straightforward.

$$
\begin{array}{ll}
 & \mathit{map}\ f\ xs \\
= & \quad \{\ \text{instance of } \mathit{foldr}\ \} \\
 & \quad\quad \mathit{foldr}\ (\lambda x\ ys \to f\ x :\ \ \ \ ys)\ [\,]\ \ xs \\
= & \quad \{\ \text{abstract over } (:) \text{ and } [\,]\ \} \\
 & \quad\quad (\lambda \mathit{cons}\ \mathit{nil} \to \mathit{foldr}\ (\lambda x\ ys \to f\ x\ `\mathit{cons}`\ ys)\ \mathit{nil}\ xs)\ (:)\ [\,] \\
= & \quad \{\ \text{instance of } \mathit{build}\ \} \\
 & \quad \mathit{build}\ (\lambda \mathit{cons}\ \mathit{nil} \to \mathit{foldr}\ (\lambda x\ ys \to f\ x\ `\mathit{cons}`\ ys)\ \mathit{nil}\ xs)
\end{array}
$$

We can now try to optimise our *sumSq* definition. The technique is to use standard transformations guided by heuristics and whenever the fusion rule can be applied we do so.

$$
\begin{aligned}
& sumSq\ xs \\
= & \quad \{ \text{ unfold the definition of } sumSq \ \} \\
& sum\ (map\ sq\ xs) \\
= & \quad \{ \text{ unfold the definition of } sum \ \} \\
& foldr\ (+)\ 0\ (map\ sq\ xs) \\
= & \quad \{ \text{ unfold the definition of } map \ \} \\
& foldr\ (+)\ 0\ (build\ (\lambda c\ n \rightarrow foldr\ (\lambda x\ ys \rightarrow sq\ x\ `c`\ ys)\ n\ xs)) \\
= & \quad \{ \ foldr/build \text{ fusion rule } \} \\
& (\lambda c\ n \rightarrow foldr\ (\lambda x\ ys \rightarrow sq\ x\ `c`\ ys)\ n\ xs)\ (+)\ 0 \\
= & \quad \{ \ \beta\text{-reduce } \} \\
& foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ xs
\end{aligned}
$$

If we now split on the two possible cases of xs we get the final definition. In the $[\,]$ case

$$
\begin{aligned}
& sumSq\ [\,] \\
= & \quad \{ \text{ above derivation } \} \\
& foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ [\,] \\
= & \quad \{ \text{ unfold the definition of } foldr \text{ for case } [\,] \ \} \\
& 0
\end{aligned}
$$

And in the $(x : xs)$ case

$$
\begin{aligned}
& sumSq\ (x : xs) \\
= & \quad \{ \text{ above derivation } \} \\
& foldr\ (\lambda x\ ys \rightarrow sq\ x + ys)\ 0\ (x : xs) \\
= & \quad \{ \text{ unfold the definition of } foldr \text{ for case } (:) \ \} \\
& (\lambda x\ ys \rightarrow sq\ x + ys)\ x\ (sumSq\ xs) \\
= & \quad \{ \ \beta\text{-reduce } \} \\
& sq\ x \ \ + sumSq\ xs \\
= & \quad \{ \text{ unfold the definition of } sq \ \} \\
& x \times x + sumSq\ xs
\end{aligned}
$$

Thus the final definition is

$$
\begin{aligned}
sumSq\ [\,] \quad\ &= 0 \\
sumSq\ (x : xs) &= x \times x + sumSq\ xs
\end{aligned}
$$

Note that there are no lists constructed in this final version. Note also that we have arrived at the same final code as we obtained using the Burstall and Darlington method.

Automation

We have done this sequence of transformations by hand, but it can also be performed automatically by the optimisation phase of a compiler (albeit in more ponderous detail and not necessarily in the same order). The following is the full source module for the *sumSq* example ready to feed to the GHC optimiser.

$$
\begin{array}{l}
\textbf{module}\ \mathit{SumSq}\ (\mathit{sumSq})\ \textbf{where} \\
\textbf{import}\ \mathit{Prelude\ hiding}\ (\mathit{map}, \mathit{sum}) \\
\textbf{import}\ \mathit{GHC.Exts}\ (\mathit{build}) \\
\mathit{map}\ f\ \mathit{xs} \;\; = \mathit{build}\ (\lambda \mathit{cons}\ \mathit{nil} \rightarrow \mathit{foldr}\ (\lambda x\ \mathit{ys} \rightarrow f\ x\ \text{`}\mathit{cons}\text{`}\ \mathit{ys})\ \mathit{nil}\ \mathit{xs}) \\
\mathit{sum}\ \mathit{xs} \;\; = \mathit{foldr}\ (+)\ 0\ \mathit{xs} \\
\mathit{sq}\ x \;\; = x \times x \\
\mathit{sumSq} \;\; :: [\mathit{Int}] \rightarrow \mathit{Int} \\
\mathit{sumSq}\ \mathit{xs} = \mathit{sum}\ (\mathit{map}\ \mathit{sq}\ \mathit{xs})
\end{array}
$$

In the version of GHC which we are using[5], the functions *foldr* and *build* and the *foldr/build* rule are defined in one of the core modules. If we were to give the rule ourselves we would write it as follows

```
{-# RULES "foldr/build"
      forall f z g.  foldr f z (build g) = g f z  #-}
```

The following shows the transformed core code (edited for clarity).

$$
\begin{array}{l}
\mathit{go} :: [\mathit{Int}] \rightarrow \mathit{Int\#} \\
\mathit{go} = \lambda \mathit{xs} \rightarrow \textbf{case}\ \mathit{xs}\ \textbf{of} \\
\qquad\qquad [\,] \quad\;\; \rightarrow 0 \\
\qquad\qquad x : \mathit{xs}' \rightarrow \textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad I\#\ x\# \rightarrow \textbf{case}\ \mathit{go}\ \mathit{xs}'\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad s\# \rightarrow (x\# \;\times\#\; x\#)\; +\#\; s\# \\
\mathit{sumSq} :: [\mathit{Int}] \rightarrow \mathit{Int} \\
\mathit{sumSq} = \mathit{\_\_inline\_me}\ (\lambda \mathit{xs} \rightarrow \textbf{case}\ \mathit{go}\ \mathit{xs}\ \textbf{of}\ s \rightarrow I\#\ s)
\end{array}
$$

[5]GHC version 6.12

Looking at the output code we see that, modulo the use of unboxed primitives[6], the code is essentially the same as the result we obtained manually. In particular the intermediate list between *sum* and *map* has been eliminated. The optimiser has applied the *foldr/build* rule once, as we did manually, along with unfolding, β-reduction and numerous other simplifications according to its standard heuristics.

Range of fusible functions

The *foldr/build* fusion system is remarkably effective. In particular it works well for list comprehensions. By translating list comprehensions into uses of *build* and *foldr*, the *foldr/build* fusion system can eliminate all the lists that are internal to the comprehension itself. Furthermore each of the list generators are consumed with a *foldr* which gives the potential to fuse with them. There is also the possibility to fuse with an overall consumer because the overall list is built using *build*.

Unfortunately, not all list consumers can be written effectively in terms of *foldr*. In particular *foldl* and *zip* are problematic. While we saw earlier that *foldl* can be written in terms of *foldr* and can thereby be fused, the way it is written uses higher order functions and so does the resulting fused code. With standard compilation schemes this fused code is extremely inefficient. Gill (1996, Section 3.2.3, 4.4) proposes an arity-raising transformation that would turn the higher order *foldl* into the standard recursion with an accumulating parameter. At the time of writing however, the optimisation has not been implemented in any major compiler.

The *sumSq* example demonstrates the problem with accumulating parameters. We had to define *sum* as a *foldr* when it is almost always more efficient to define it as a *foldl*. Indeed, looking again at the core code above we see that the worker function is not tail recursive – it uses linear stack space. If however we do define *sum* in the standard way as a *foldl* then, while it fuses, it runs slower.

We must watch out for this issue when evaluating fusion systems. While the fusion transformation itself may be an improvement, if we have to write functions in a highly non-standard way then we may simply be shuffling allocations from one place to another. In the worst case it is even possible that the overall result may be worse than the original simple unfused version.

The *zip* function is a problem in the *foldr/build* system because it cannot be written so as to consume both input lists with a *foldr*. It can produce the result

[6] The # -suffix naming convention is used to indicate unboxed primitive machine types.

list using *build* and can be written to consume one input list or the other using *foldr*, but not both simultaneously.

1.3.8 The *unbuild*/*unfoldr* fusion system

One approach to addressing the challenge of left folds and zips was proposed by Svenningsson (2002). It is another shortcut fusion system, using the fusion combinators *unfoldr* and *unbuild*[7]

$$unfoldr :: (s \to Maybe\ (a, s)) \to s \to [a]$$
$$unbuild :: (\forall s.\ (s \to Maybe\ (a, s)) \to s \to b) \to [a] \to b$$

Most functional programmers are at least vaguely aware of *unfoldr*

```
unfoldr :: (s -> Maybe (a, s)) -> s -> [a]
unfoldr next s = case next s of
                   Nothing    -> []
                   Just (x, s') -> x : unfoldr next s'
```

It captures a general pattern for constructing lists. One can see it as an iterator style, where an iterator consists of a state and a stepper function from that state to a sequence element along with a new state, or to a terminal value. We can express a wide range of functions that produce lists in this style.

```
iterate f          = unfoldr (\x -> Just (x, f x))
enumFromTo n m     = let next i | i > m      = Nothing
                                | otherwise = Just (i, i + 1)
                     in unfoldr next n
map f              = let next []       = Nothing
                         next (x : xs) = Just (f x, xs))
                     in unfoldr next
filter p           = let next []                  = Nothing
                         next (x : xs) | p x      = Just (x, xs))
                                       | otherwise = next xs
                     in unfoldr next
lines              = let getLine ""   = Nothing
                         getLine str  = Just (line, drop 1 str')
                           where
                             (line, str') = break (== '\n') str
                     in unfoldr getLine
```

[7] Svenningsson calls this function *destroy* but for symmetry we follow Gill and call it *unbuild*.

Note that there is some overlap with the functions that can be expressed with *foldr*; in particular functions like *map* and *filter* that both consume and produce lists.

So while *unfoldr* is for producing lists, *unbuild* is for consuming them. It takes a function which consumes sequences produced in the iterator style of *unfoldr* and applies the function to a suitable argument so as to obtain an equivalent function that consumes lists.

$$
\begin{aligned}
&\mathit{unbuild} :: (\forall s.\ (s \to \mathit{Maybe}\ (a, s)) \to s \to b) \to [a] \to b \\
&\mathit{unbuild}\ g\ \mathit{xs} = g\ \mathit{uncons}\ \mathit{xs} \\
&\quad \textbf{where} \\
&\qquad \mathit{uncons} :: [a] \to \mathit{Maybe}\ (a, [a]) \\
&\qquad \mathit{uncons}\ [\,] \quad\ = \mathit{Nothing} \\
&\qquad \mathit{uncons}\ (x : \mathit{xs}) = \mathit{Just}\ (x, \mathit{xs})
\end{aligned}
$$

Using this function we can write various common list consumers and can rewrite transformers like *map* and *filter* to consume their input using it.

$$
\begin{aligned}
&\mathit{foldr}\ f\ z = \mathit{unbuild}\ (\lambda \mathit{next}\ s_0 \to \\
&\qquad\qquad \textbf{let}\ \mathit{go}\ s = \textbf{case}\ \mathit{next}_0\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad \mathit{Nothing} \quad \to z \\
&\qquad\qquad\qquad\qquad \mathit{Just}\ (x, s') \to f\ x\ (\mathit{go}\ s') \\
&\qquad\qquad \textbf{in}\ \mathit{go}\ s_0) \\
&\mathit{foldl}\ f\ a\ = \mathit{unbuild}\ (\lambda \mathit{next}\ s_0 \to \\
&\qquad\qquad \textbf{let}\ \mathit{go}\ a\ s = \ \textbf{case}\ \mathit{next}_0\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\ \ \mathit{Nothing} \quad \to a \\
&\qquad\qquad\qquad\qquad\ \ \mathit{Just}\ (x, s') \to \mathit{go}\ (f\ a\ x)\ s' \\
&\qquad\qquad \textbf{in}\ \mathit{go}\ a\ s_0) \\
&\mathit{map}\ f \quad = \mathit{unbuild}\ (\lambda \mathit{next}_0\ s_0 \to \\
&\qquad\qquad \textbf{let}\ \mathit{next}\ s = \ \textbf{case}\ \mathit{next}_0\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\ \ \mathit{Nothing} \quad \to \mathit{Nothing} \\
&\qquad\qquad\qquad\qquad\ \ \mathit{Just}\ (x, s') \to \mathit{Just}\ (f\ x, s') \\
&\qquad\qquad \textbf{in}\ \mathit{unfoldr}\ \mathit{next}\ s_0) \\
&\mathit{filter}\ p \quad = \mathit{unbuild}\ (\lambda \mathit{next}_0\ s_0 \to \\
&\qquad\qquad \textbf{let}\ \mathit{next}\ s = \ \textbf{case}\ \mathit{next}_0\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\ \ \mathit{Nothing} \qquad\qquad\ \ \to \mathit{Nothing} \\
&\qquad\qquad\qquad\qquad\ \ \mathit{Just}\ (x, s') \mid p\ x \qquad\ \ \to \mathit{Just}\ (x, s') \\
&\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} \to \mathit{next}\ s' \\
&\qquad\qquad \textbf{in}\ \mathit{unfoldr}\ \mathit{next}\ s_0)
\end{aligned}
$$

Fusion rule (*unbuild*/*unfoldr*)**.** The main equation of the *unbuild*/*unfoldr* system is

$$\forall k\ g\ s.\ unbuild\ g\ (unfoldr\ k\ s) = g\ k\ s$$

On the left hand side, the list produced by *unfoldr* is immediately consumed by *unbuild*. The arguments to *unfoldr* are the stepper function and initial state – exactly what the function passed to *unbuild* is expecting. On the right hand side the iterator consumer function is applied directly to the stepper function and initial state, with no mention of lists anywhere.

The *sumSq* example

As usual, let us illustrate the fusion technique with the *sumSq* example.

$$sumSq\ xs = sum\ (map\ sq\ xs)$$

This time we need to define *sum* and *map* in terms of *unbuild* and *unfoldr*. We already defined *map* above. We could define *sum* as an instance of *foldr* which in turn we defined above in terms of *unbuild*. However, since one of the primary claimed advantages of this technique is that it can handle functions that use accumulating parameters, let us define *sum* in terms of *foldl* which is also defined directly in terms of *unbuild*.

$$sum = foldl\ (+)\ 0$$

As we defined them above, the body of *map* and *foldl* are rather large. To make the derivation more presentable let us adjust the definitions of *map* and *sum* using named helper functions:

$$\begin{aligned}
&sum \quad = unbuild\ sumIter\\
&map\ f = unbuild\ (mapIter\ f)\\
&sumIter \quad next_0\ s_0 = sumGo\ next_0\ 0\ s_0\\
&mapIter\ f\ next_0\ s_0 = unfoldr\ (next_{map}\ f\ next_0)\ s_0\\
&sumGo\ next_0 = \mathbf{let}\ go\ a\ s = \mathbf{case}\ next_0\ s\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\qquad Nothing \quad \to a\\
&\qquad\qquad\qquad\qquad\qquad\qquad Just\ (x, s') \to go\ (a + x)\ s'\\
&\qquad\qquad\quad \mathbf{in}\ go\\
&next_{map}\ f\ next_0\ s = \mathbf{case}\ next_0\ s\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad Nothing \quad \to Nothing\\
&\qquad\qquad\qquad\qquad Just\ (x, s') \to Just\ (f\ x, s')
\end{aligned}$$

Now we can start with unfolding definitions

$$
\begin{aligned}
& \mathit{sum}\ (\mathit{map}\ \mathit{sq}\ \mathit{xs}) \\
= \quad & \{\ \text{unfold the definition of } \mathit{sum} \text{ and } \mathit{map}\ \} \\
& \mathit{unbuild}\ \mathit{sumIter}\ (\mathit{unbuild}\ (\mathit{mapIter}\ \mathit{sq})\ \mathit{xs})
\end{aligned}
$$

When we get to this point however it looks like we are stuck. We are trying to find an instance of $\mathit{unbuild}\ g\ (\mathit{unfoldr}\ f\ e)$ but what we have here is *unbuild* applied to *unbuild* with the *unfoldr* hidden within the parameter to the inner *unbuild*. We want to somehow move the outer *unbuild* inside the inner one so that it will be applied directly to the *unfoldr*.

Fortunately we can calculate a suitable equation

$$
\begin{aligned}
& \mathit{unbuild}\ g\ (\mathit{unbuild}\ g'\ \mathit{xs}) \\
= \quad & \{\ \text{unfold the definition of } \mathit{unbuild}\ \} \\
& g\ \mathit{uncons}\ (g'\ \mathit{uncons}\ \mathit{xs}) \\
= \quad & \{\ \text{abstract over } \mathit{xs} \text{ and the second occurrence of } \mathit{uncons}\ \} \\
& (\lambda \mathit{next}\ s_0 \to g\ \mathit{uncons}\ (g'\ \mathit{next}\ s_0))\ \mathit{uncons}\ \mathit{xs} \\
= \quad & \{\ \text{fold instance of } \mathit{unbuild} \text{ into a use of } \mathit{unbuild}\ \} \\
& \mathit{unbuild}\ (\lambda \mathit{next}\ s_0 \to \mathit{unbuild}\ g\ (g'\ \mathit{next}\ s_0))\ \mathit{xs}
\end{aligned}
$$

We can use this equation as a rewrite rule just as we do with fusion rules

Fusion rule (*unbuild*/*unbuild*)**.**

$$
\begin{aligned}
\forall g\ g'\ \mathit{xs}.\quad & \mathit{unbuild}\ g\ (\mathit{unbuild}\ g'\ \mathit{xs}) \\
= \ & \mathit{unbuild}\ (\lambda \mathit{next}\ s_0 \to \mathit{unbuild}\ g\ (g'\ \mathit{next}\ s_0))\ \mathit{xs}
\end{aligned}
$$

We may now carry on from where we left off above and apply the *unbuild*/*unbuild* rule

$$
\begin{aligned}
& \mathit{unbuild}\ \mathit{sumIter}\ (\mathit{unbuild}\ (\mathit{mapIter}\ \mathit{sq})\ \mathit{xs}) \\
= \quad & \{\ \mathit{unbuild}/\mathit{unbuild} \text{ rule}\ \} \\
& \mathit{unbuild}\ (\lambda \mathit{next}\ s_0 \to \mathit{unbuild}\ \mathit{sumIter}\ (\mathit{mapIter}\ \mathit{sq}\ \mathit{next}\ s_0))\ \mathit{xs}
\end{aligned}
$$

It will be convenient for the moment to focus on the inner *unbuild* sub-expression

$$
\begin{aligned}
& \mathit{unbuild}\ \mathit{sumIter}\ (\mathit{mapIter}\ \mathit{sq}\ \mathit{next}\ s_0) \\
= \quad & \{\ \text{unfold the definition of } \mathit{mapIter}\ \} \\
& \mathit{unbuild}\ \mathit{sumIter}\ (\mathit{unfoldr}\ (\mathit{next}_{\mathit{map}}\ \mathit{sq}\ \mathit{next})\ s_0) \\
= \quad & \{\ \mathit{unbuild}/\mathit{unfoldr} \text{ fusion}\ \} \\
& \mathit{sumIter}\ (\mathit{next}_{\mathit{map}}\ \mathit{sq}\ \mathit{next})\ s_0
\end{aligned}
$$

So we have managed to get the *unbuild* and *unfoldr* together so that we could apply the fusion rule, however we are only half way to getting good final code.

Note that if we peek inside the composition of *sumIter* with $next_{map}$ we will see that $next_{map}$ allocates values of type *Maybe* which are immediately consumed by the worker function of *sumIter*. So in fact, while we have applied the fusion rule we have not reduced the number of runtime allocations in comparison to the original list version of *sumSq*, we have just shuffled them from one place to another. It is certainly the case that applying the fusion rule has reduced allocations in comparison to the version of *sumSq* that used *sum* and *map* defined in terms of *unbuild* and *unfoldr* but that is only because they were worse than the original list versions!

Svenningsson (2002, Section 3.3 footnote 2) assures us that the *Maybe* type is only transient and will be eliminated by further transformation. It is of course crucial that it is eliminated otherwise the fusion system is not an optimisation. The transformation that finally eliminates the intermediate *Maybe* allocations is the 'case-of-case' transformation (Peyton Jones and Santos, 1998, Section 5). We can expose the opportunity for this by unfolding more definitions

$$
\begin{array}{ll}
 & \mathit{sumIter}\ (\mathit{next}_{map}\ \mathit{sq}\ \mathit{next})\ s_0 \\
= & \quad \{\ \text{unfold the definition of } \mathit{sumIter}\ \} \\
 & \mathit{sumGo}\ (\mathit{next}_{map}\ \mathit{sq}\ \mathit{next})\ 0\ s_0 \\
= & \quad \{\ \text{unfold the definition of } \mathit{sumGo}\ \} \\
 & \textbf{let}\ \mathit{go}\ a\ s = \textbf{case}\ \mathit{next}_{map}\ \mathit{sq}\ \mathit{next}\ s\ \textbf{of} \\
 & \qquad\qquad \mathit{Nothing} \quad \rightarrow a \\
 & \qquad\qquad \mathit{Just}\ (x, s') \rightarrow \mathit{go}\ (a + x)\ s' \\
 & \textbf{in}\ \mathit{go}\ 0\ s_0 \\
= & \quad \{\ \text{unfold the definition of } \mathit{next}_{map}\ \} \\
 & \textbf{let}\ \mathit{go}\ a\ s = \textbf{case}\ (\textbf{case}\ \mathit{next}\ s\ \textbf{of} \\
 & \qquad\qquad\qquad\qquad \mathit{Nothing} \quad \rightarrow \mathit{Nothing} \\
 & \qquad\qquad\qquad\qquad \mathit{Just}\ (x, s') \rightarrow \mathit{Just}\ (\mathit{sq}\ x, s'))\ \textbf{of} \\
 & \qquad\qquad \mathit{Nothing} \quad \rightarrow a \\
 & \qquad\qquad \mathit{Just}\ (x, s') \rightarrow \mathit{go}\ (a + x)\ s' \\
 & \textbf{in}\ \mathit{go}\ 0\ s_0 \\
= &
\end{array}
$$

$$
\begin{aligned}
&= \quad \{ \text{ case-of-case transformation } \} \\
&\quad \textbf{let } go\ a\ s = \textbf{case } next\ s \textbf{ of} \\
&\qquad\qquad\qquad\quad Nothing \quad\ \to a \\
&\qquad\qquad\qquad\quad Just\ (x, s') \to go\ (a + sq\ x)\ s' \\
&\quad \textbf{in } go\ 0\ s_0
\end{aligned}
$$

So we have eliminated the intermediate allocations between *sumIter* and $next_{map}$ but we still have the input to $next_{map}$ to worry about.

Recall that we were looking at a sub-expression and the outer context was

$$
\begin{aligned}
&\ unbuild\ (\lambda next\ s \to (\ldots))\ xs \\
&= \\
&\ (\lambda next\ s_0 \to (\ldots))\ uncons\ xs
\end{aligned}
$$

So let us return to the outer context and β-reduce

$$
\begin{aligned}
&= \quad \{ \text{ substitute } next := uncons, s_0 := xs \ \} \\
&\quad \textbf{let } go\ a\ s = \textbf{case } uncons\ s \textbf{ of} \\
&\qquad\qquad\qquad\quad Nothing \quad\ \to a \\
&\qquad\qquad\qquad\quad Just\ (x, s') \to go\ (a + sq\ x)\ s' \\
&\quad \textbf{in } go\ 0\ xs \\
&= \quad \{ \text{ unfold the definition of } uncons \ \} \\
&\quad \textbf{let } go\ a\ s = \textbf{case } (\textbf{case } s \textbf{ of} \\
&\qquad\qquad\qquad\quad [\,] \quad\ \to Nothing \\
&\qquad\qquad\qquad\quad (x : s') \to Just\ (x, s')) \\
&\qquad\qquad\quad \textbf{of} \\
&\qquad\qquad\qquad\quad Nothing \quad\ \to a \\
&\qquad\qquad\qquad\quad Just\ (x, s') \to go\ (a + sq\ x)\ s' \\
&\quad \textbf{in } go\ 0\ xs \\
&= \quad \{ \text{ case-of-case transformation } \} \\
&\quad \textbf{let } go\ a\ s = \textbf{case } s \textbf{ of} \\
&\qquad\qquad\qquad\quad [\,] \quad\ \to a \\
&\qquad\qquad\qquad\quad (x : s') \to go\ (a + sq\ x)\ s' \\
&\quad \textbf{in } go\ 0\ xs
\end{aligned}
$$

Again it is the case-of-case transformation that reduces the allocations. We are finally down to doing strictly fewer allocations than the original list version of *sumSq*.

Our final fused definition is

$$
\begin{array}{l}
sumSq\ xs = go\ 0\ xs \\
\quad \textbf{where} \\
\quad\quad go\ a\ [\,] \quad\quad = a \\
\quad\quad go\ a\ (x : xs) = go\ (a + x \times x)\ xs
\end{array}
$$

This is an optimal result as far as fusion goes since all the list allocations have been eliminated.

Although the derivation was somewhat long winded it did not need any terribly sophisticated transformations. Primarily it involved unfolding, β-reduction, rewrite rule application and the case-of-case transformation. All of these are local and have purely syntactic criteria.

Note that the derivation would not have been much different if we had tried the version of *sum* that does not use an accumulating parameter, as we used with *foldr/build*. In comparison, while the derivation was straightforward with *foldr/build*, if we had tried the accumulating parameter version then we would have needed to do an arity analysis and arity raising transformation to turn the fused code into fast code that does not allocate any closures.

The *zip* function

The other claimed advantage of the *unbuild/unfoldr* system over the *foldr/build* system is that it can express *zip* and effectively fuse it on all inputs and outputs. The definition is somewhat verbose:

$$
\begin{array}{l}
zip\ as\ bs = \\
\quad unbuild\ (\lambda next_a\ s_{a\,0} \rightarrow \\
\quad\quad unbuild\ (\lambda next_b\ s_{b\,0} \rightarrow \\
\quad\quad\quad\quad\quad \textbf{let}\ next\ (s_a, s_b) = \\
\quad\quad\quad\quad\quad\quad\quad \textbf{case}\ next_a\ s_a\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad Nothing \quad\quad \rightarrow Nothing \\
\quad\quad\quad\quad\quad\quad\quad\quad Just\ (a, s'_a) \rightarrow \textbf{case}\ next_b\ s_b\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Nothing \quad\quad \rightarrow Nothing \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Just\ (b, s'_b) \rightarrow Just\ ((a, b), (s'_a, s'_b)) \\
\quad\quad\quad\quad\quad \textbf{in}\ unfoldr\ next\ (s_{a\,0}, s_{b\,0}) \\
\quad\quad)\ bs \\
\quad)\ as
\end{array}
$$

The key part is the definition of the new *next* iterator function which combines the $next_a$ and $next_b$ of the corresponding inputs.

It is interesting to compare this definition to the standard definition of *zip*:

$$\begin{array}{lll} zip\ (a : as)\ (b : bs) & = & (a, b) : zip\ as\ bs \\ zip\ \_\ \ \ \_\ \ \ & = & [\,] \end{array}$$

The standard definition is nice and declarative and appears to be symmetric in the two input lists. By contrast the version above using *unbuild* and *unfoldr* is verbose, somewhat operational in character and asymmetric in its treatment of the two inputs, in that the *next* function has to pull from one input sequence before the other. These characteristics are typical of definitions in the *unbuild*/*unfoldr* framework. If we desugar the pattern matching in the standard definition of zip however then we see that the two versions are more recognisably similar.

$$\begin{array}{lll} zip\ as\ bs = \textbf{case}\ as\ \textbf{of} & & \\ \qquad\qquad [\,] & \to [\,] & \\ \qquad\qquad (a : as') & \to \textbf{case}\ bs\ \textbf{of} & \\ \qquad\qquad\qquad\qquad [\,] & \to [\,] & \\ \qquad\qquad\qquad\qquad (b : bs') & \to (a, b) : zip\ as'\ bs' & \end{array}$$

For a full example of fusing *zip* see (Svenningsson, 2002, Section 4). The derivation goes through in much the same way as with *sum* and *map* above, using two instances of *unbuild*/*unfoldr* fusion to expose opportunities for the case-of-case transformation to finally eliminate the intermediate allocations.

Limitations

The *unbuild*/*unfoldr* system is not a panacea. Just as there are functions that cannot be effectively fused in in the *foldr*/*build* system, such as *foldl*, there are functions that cannot be effectively fused under the *unbuild*/*unfoldr* system. The canonical example is *filter*. Let us see what goes wrong.

Let us use a modification of the *sumSq* example:

$$sumEven\ xs = sum\ (filter\ even\ xs)$$

The definitions are the same as before except for the $next_{filter}$ function.

$$
\begin{aligned}
&\mathit{sum} \quad = \mathit{unbuild}\ \mathit{sumIter}\\
&\mathit{filter}\ p = \mathit{unbuild}\ (\mathit{next}_{\mathit{filter}}\ p)\\
&\mathit{sumIter} \quad \mathit{next}_0\ s_0 = \mathit{sumGo}\ \mathit{next}_0\ 0\ s_0\\
&\mathit{filterIter}\ p\ \mathit{next}_0\ s_0 = \mathit{unfoldr}\ (\mathit{next}_{\mathit{filter}}\ p\ \mathit{next}_0)\ s_0\\
&\mathit{sumGo}\ \mathit{next}_0 = \mathbf{let}\ \mathit{go}\ a\ s = \mathbf{case}\ \mathit{next}_0\ s\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Nothing} \quad \rightarrow a\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Just}\ (x, s') \rightarrow \mathit{go}\ (a + x)\ s'\\
&\qquad\qquad\qquad \mathbf{in}\ \mathit{go}\\
&\mathit{next}_{\mathit{filter}}\ p\ \mathit{next}_0 =\\
&\quad \mathbf{let}\ \mathit{fnext}\ s = \mathbf{case}\ \mathit{next}_0\ s\ \mathbf{of}\\
&\qquad\qquad\qquad\quad \mathit{Nothing} \qquad\qquad\qquad\quad \rightarrow \mathit{Nothing}\\
&\qquad\qquad\qquad\quad \mathit{Just}\ (x, s') \quad \mid p\ x \qquad\quad \rightarrow \mathit{Just}\ (x, s')\\
&\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} \rightarrow \mathit{fnext}\ s'\\
&\quad\ \ \mathbf{in}\ \mathit{next}
\end{aligned}
$$

The derivation initially proceeds exactly as in the *sum*/*map* example; indeed we are able to apply the *unbuild*/*unfoldr* fusion rule. This gives us the following sub-expression, from which point we can start unfolding definitions.

$$
\begin{aligned}
&\quad \mathit{sumIter}\ (\mathit{next}_{\mathit{filter}}\ \mathit{even}\ \mathit{next})\ s_0\\
&= \quad \{\ \text{unfold the definition of } \mathit{sumIter}\ \}\\
&\quad \mathit{sumGo}\ (\mathit{next}_{\mathit{filter}}\ \mathit{even}\ \mathit{next})\ 0\ s_0\\
&= \quad \{\ \text{unfold the definition of } \mathit{sumGo}\ \}\\
&\quad \mathbf{let}\ \mathit{go}\ a\ s = \mathbf{case}\ \mathit{next}_{\mathit{filter}}\ \mathit{even}\ \mathit{next}\ s\ \mathbf{of}\\
&\qquad\qquad\qquad\quad \mathit{Nothing} \quad \rightarrow a\\
&\qquad\qquad\qquad\quad \mathit{Just}\ (x, s') \rightarrow \mathit{go}\ (a + x)\ s'\\
&\quad \mathbf{in}\ \mathit{go}\ 0\ s_0\\
&= \quad \{\ \text{unfold the definition of } \mathit{next}_{\mathit{filter}}\ \}\\
&\quad \mathbf{let}\ \mathit{go}\ a\ s = \mathbf{case}\ (\mathbf{let}\ \mathit{fnext}\ s = \mathbf{case}\ \mathit{next}\ s\ \mathbf{of}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Nothing} \qquad\qquad\qquad \rightarrow \mathit{Nothing}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{Just}\ (x, s') \mid \mathit{even}\ x \quad\ \rightarrow \mathit{Just}\ (x, s')\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} \rightarrow \mathit{fnext}\ s'\\
&\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \mathit{fnext}\ s)\\
&\qquad\qquad\quad \mathbf{of}\\
&\qquad\qquad\qquad\quad \mathit{Nothing} \quad \rightarrow a\\
&\qquad\qquad\qquad\quad \mathit{Just}\ (x, s') \rightarrow \mathit{go}\ (a + x)\ s'\\
&\quad \mathbf{in}\ \mathit{go}\ 0\ s_0
\end{aligned}
$$

But now we are stuck. Where previously we could do a simple case-of-case transformation we are now faced with a situation we might call "case of fixed

point of case". We can try floating the **let** out of the way and unfolding the definition of *fnext* by one step to give us something closer to what we want:

$$
\begin{array}{l}
\textbf{let } \mathit{fnext}\ s = \textbf{case } \mathit{next}\ s \textbf{ of} \\
\qquad\qquad \begin{array}{lll}
\mathit{Nothing} & & \to \mathit{Nothing} \\
\mathit{Just}\ (x, s') & \mid \mathit{even}\ x & \to \mathit{Just}\ (x, s') \\
& \mid \mathit{otherwise} & \to \mathit{fnext}\ s'
\end{array} \\
\quad \mathit{go}\ a\ s = \textbf{case } (\textbf{case } \mathit{next}\ s \textbf{ of} \\
\qquad\qquad\qquad \begin{array}{lll}
\mathit{Nothing} & & \to \mathit{Nothing} \\
\mathit{Just}\ (x, s') & \mid \mathit{even}\ x & \to \mathit{Just}\ (x, s') \\
& \mid \mathit{otherwise} & \to \mathit{fnext}\ s')
\end{array} \\
\qquad\quad \textbf{of} \\
\qquad\qquad \begin{array}{ll}
\mathit{Nothing} & \to a \\
\mathit{Just}\ (x, s') & \to \mathit{go}\ (a + x)\ s'
\end{array} \\
\textbf{in } \mathit{go}\ 0\ s_0
\end{array}
$$

However we still cannot apply case-of-case because one out of the three branches of the inner case is not manifestly a *Nothing* or *Just* constructor.

As things stand, this is where the compiler will stop[8] and the final code will allocate and immediately consume *Just* constructors. That is, we have succeeded in exchanging one set of allocations for another but we have failed to actually eliminate any. This example underlines the need to analyse fusion systems carefully. It is not enough to check that the fusion rules themselves reduce allocations.

This example is a key motivation for the approach we take with stream fusion. The problem is with the local recursion getting in the way of the case-of-case transformation. The local recursion arises from the *filter* function's recursive *next* function. This suggests that a possible solution would be to make the *next* non-recursive so that the ordinary case-of-case transformation would work. This is indeed the approach we take with stream fusion.

1.3.9 Expressiveness in shortcut fusion systems

It is in the nature of shortcut fusion systems that they restrict the way we write fusible functions, simply because we are forced to write functions in terms of the given fusion combinators.

[8] In fact GHC will stop at the previous step because it refuses to unfold recursive definitions at all.

With *foldr*/*build* fusion we have a good deal of flexibility in how we write list producers while we are rather restricted in how we write list consumers. For list producers we can use ordinary recursive functions; we get to determine the control flow using locally and mutually recursive functions. The only constraint is that the collection of functions be parametrised by *cons* and *nil*. On the other hand, list consumers have to fit the *foldr* recursion pattern. While it is certainly true that *foldr* is highly expressive, many list consumers are most naturally expressed as collections of mutually recursive functions and forcing them into the *foldr* mould can feel like an exercise in obfuscation.

As usual, the situation with *unbuild*/*unfoldr* is reversed: we must write list producers in a restricted way while we have much more freedom in how we write list consumers. List consumers using *unbuild* have to use the supplied initial state and stepper function but otherwise they are free to use whatever pattern of recursive functions is most natural. List producers on the other hand have to be defined using a single stepper function that gets passed a single state type. Again, it is true that *unfoldr* is quite expressive but definitions using it can seem quite unnatural when compared to equivalent free-form recursive definitions.

There are limits to what data structures we can hope to eliminate using deforestation techniques. In informal terms, fusion systems only attempt to eliminate intermediate data structures that are used for communication, not data structures that are used for data storage. Functions that produce lists using *build* or *unfoldr* do so in a way that is 'write only'; that is, once each list element is produced it cannot subsequently be inspected.

For example we could not hope to eliminate all the intermediate allocations in a comparison-based *sort* function. Although we could eliminate lists used as the input and output, we would necessarily have to replace them with some other intermediate data structure (e.g. a heap). Consider, for example, a simple insertion sort on lists

$$
\begin{aligned}
&\mathit{sort} = \mathit{foldr}\ \mathit{insert}\ [\,] \\
&\mathit{insert}\ x\ [\,] &&= x : [\,] \\
&\mathit{insert}\ x\ (y : \mathit{ys}) \mid x \geqslant y &&= y : \mathit{insert}\ x\ \mathit{ys} \\
&\qquad\qquad\quad \mid \mathit{otherwise} &&= x : y : \mathit{ys}
\end{aligned}
$$

We cannot change this definition to construct the result list using *build* because the *insert* function inspects the list so it can add the new element in the right position; it uses the list as a read/write data store.

1.4 Stream fusion

A note on terminology. *The term 'stream' is unfortunately somewhat overloaded. Some authors take it to mean infinite sequences while others take it to mean possibly terminating sequences. That is, some use the functor $S\ a\ x = (a, x)$ while other use $S\ a\ x = \mathbf{1} + (a, x)$. The streams in stream fusion are possibly terminating sequences.*

Stream fusion is an alternative formulation of, and an evolutionary improvement on, *unbuild*/*unfoldr* fusion. We will look at two formulations of streams: with and without 'skip'. Streams without skip are simply a reformulation of the *unbuild*/*unfoldr* system. The addition of skip is the essential extra ingredient that enables us to solve the *filter* problem.

1.4.1 Streams without skip

Recall the types of *unfoldr* and *unbuild*

$$\begin{aligned}&\mathit{unfoldr} :: (s \to \mathit{Maybe}\ (a, s)) \to s \to [a]\\ &\mathit{unbuild} :: (\forall s.\ (s \to \mathit{Maybe}\ (a, s)) \to s \to b) \to [a] \to b\end{aligned}$$

These functions capture how we construct and consume lists using the iterator view. In particular, the *unfoldr* function takes an initial state and a stepper function; it construct a list by repeatedly applying the stepper function to successive states, starting with the initial state.

A stream captures the iterator view by packing the initial state and stepper function into a data structure. We can define a non-skipping stream as

$$\mathbf{data}\ \mathit{Stream}\ a = \exists s.\ \mathit{Stream}\ (s \to \mathit{Maybe}\ (a, s))\ s$$

So while a stream is not a list, it is intended to be equivalent to a list. We can convert from a *Stream* back to a list simply by unpacking the data structure and applying the normal *unfoldr* function

$$\begin{aligned}&\mathit{unstream} :: \mathit{Stream}\ a \to [a]\\ &\mathit{unstream}\ (\mathit{Stream}\ \mathit{next}\ s) = \mathit{unfoldr}\ \mathit{next}\ s\end{aligned}$$

The opposite conversion is from a list to a *Stream*

$$\begin{aligned}&\mathit{stream} :: [a] \to \mathit{Stream}\ a\\ &\mathit{stream}\ xs = \mathit{Stream}\ \mathit{uncons}\ xs\\ &\quad \mathbf{where}\\ &\qquad \mathit{uncons} :: [a] \to \mathit{Maybe}\ (a, [a])\\ &\qquad \mathit{uncons}\ [\,] \quad\ = \mathit{Nothing}\\ &\qquad \mathit{uncons}\ (x : xs) = \mathit{Just}\ (x, xs)\end{aligned}$$

Equivalently we could define *stream* in terms of *unbuild*

$$\begin{aligned}&stream :: [a] \to Stream\ a\\&stream\ xs = unbuild\ Stream\ xs\end{aligned}$$

To convince oneself that this is the case it helps to stare for a moment at the type of the *Stream* data constructor and to recall the definition of *unbuild*

$$\begin{aligned}&Stream :: \quad \forall s.\ (s \to Maybe\ (a, s)) \to s \to Stream\ a\\&unbuild :: (\forall s.\ (s \to Maybe\ (a, s)) \to s \to b) \to [a] \to b\\&unbuild\ g\ xs = g\ uncons\ xs\end{aligned}$$

Of course for there to be a genuine equivalence between streams and lists it will have to be the case that *stream* and *unstream* are mutual inverses:

$$\begin{aligned}&stream \circ unstream = id :: Stream\ a \to Stream\ a\\&unstream \circ stream = id :: [a] \qquad\ \to [a]\end{aligned}$$

We will return to this issue in Chapter 3.

Fusion rule (*stream*/*unstream*)**.** The stream fusion system uses *stream* and *unstream* as the fusion combinators and the key equation of the system is

$$\forall s :: Stream\ a.\ \ stream\ (unstream\ s) = s$$

That is, converting from a stream to a list and back to a stream is an identity operation. We use this as a rewrite rule to eliminate redundant conversions.

Having first looked at the *foldr*/*build* and *unbuild*/*unfoldr* systems, it is perhaps somewhat surprising that the stream fusion combinators are merely conversion functions and that the fusion rule merely eliminates redundant conversions. Many standard list functions are direct instances of *foldr* and *unfoldr* whereas there are few interesting functions one can define just using functions that convert between lists and streams.

With stream fusion, operations are defined directly on the *Stream* type. For example the *map* function on streams, which we will call map_s, is defined as

$$\begin{aligned}&map_s :: (a \to b) \to Stream\ a \to Stream\ b\\&map_s\ f\ (Stream\ next_0\ s_0) = Stream\ next\ s_0\\&\quad \textbf{where}\\&\qquad next\ s = \textbf{case}\ next_0\ s\ \textbf{of}\\&\qquad\qquad\qquad Nothing \quad \to Nothing\\&\qquad\qquad\qquad Just\ (x, s') \to Just\ (f\ x, s')\end{aligned}$$

To lift this operation on streams to an operation on lists we use *stream* and *unstream* to convert the input and output. The corresponding *map* on lists is defined as

$$
\begin{aligned}
&map :: (a \to b) \to [a] \to [b] \\
&map\ f = unstream \circ map_s\ f \circ stream
\end{aligned}
$$

Notice that the stream version of map_s closely resembles the definition of *map* under the *unbuild*/*unfoldr* system, in particular the definition of a new *next* function in terms of the old $next_0$ function. Of course this is not surprising given the correspondence between streams and the *unbuild*/*unfoldr* functions.

Another example function on streams is *foldl* which consumes a stream

$$
\begin{aligned}
&foldl :: (b \to a \to b) \to b \to [a] \to b \\
&foldl\ f\ a = foldl_s\ f\ a \circ stream \\
&foldl_s :: (b \to a \to b) \to b \to Stream\ a \to b \\
&foldl_s\ f\ a\ (Stream\ next\ s_0) = go\ a\ s_0 \\
&\quad \textbf{where} \\
&\qquad go\ a\ s = \textbf{case}\ next\ s\ \textbf{of} \\
&\qquad\qquad\qquad Nothing \quad \to a \\
&\qquad\qquad\qquad Just\ (x, s') \to go\ (f\ a\ x)\ s'
\end{aligned}
$$

It is interesting to compare the form of the body of this function with that of map_s. The body of $foldl_s$ unfolds the entire stream whereas map_s does not, it merely creates a new stepper function. Functions on streams are characterised by an operational style. Unlike in a list data structure there is no lazy evaluation in a stream; delayed evaluation and control flow are represented explicitly.

The opportunity for fusion in this system arises when a function that produces an output list with *unstream* is composed with a function that consumes a list with *stream*. We then have the opportunity to eliminate redundant conversions. We will see this in action with our standard *sumSq* example.

As was the case for the *unbuild*/*unfoldr* system, we must be careful to check what extra allocations are introduced and when they are eliminated. For example with *map* as defined above in terms of *unstream*, map_s and *stream*, there are three data constructor allocations per sequence element. This contrasts with just one for the normal list version of *map*. These allocations must all be eliminated for stream fusion to be an improvement.

The *sumSq* example

$$sumSq\ xs = sum\ (map\ sq\ xs)$$

We have already defined *map* and we will define *sum* in terms of *foldl*

$$sum = foldl\ (+)\ 0$$

We start with unfolding definitions

$$\begin{aligned} & sum\ (map\ sq\ xs) \\ = \quad & \{\ \text{unfold the definition of } sum, foldl \text{ and } map\ \} \\ & foldl_s\ (+)\ 0\ (stream\ (unstream\ (map_s\ sq\ (stream\ xs)))) \end{aligned}$$

We are now at the point where we can see a redundant conversion from stream to list and back again. We now apply the stream fusion rule

$$\begin{aligned} & foldl_s\ (+)\ 0\ (stream\ (unstream\ (map_s\ sq\ (stream\ xs)))) \\ = \quad & \{\ stream/unstream \text{ fusion rule}\ \} \\ & foldl_s\ (+)\ 0\ (map_s\ sq\ (stream\ xs)) \end{aligned}$$

As was the case with the *unbuild*/*unfoldr* system, applying the fusion rule only gets us half way towards good final code. Were we to stop here we would get a worse result than with the original list code. While applying the *stream*/*unstream* fusion rule has eliminated two allocations per sequence element we still have two remaining allocations of *Maybe* constructors. This is one more allocation per element than in the straightforward list version.

The allocation points are at the composition boundary between functions on streams. The producer allocates *Maybe* constructors and the consumer takes them apart. In the *sumSq* example the allocation points are at the composition between $foldl_s$ and map_s and at the composition between map_s and *stream*. The strategy to eliminate the allocations at each composition boundary is to unfold definitions and to apply the case-of-case transformation.

We proceed first with the inner composition $map_s\ sq\ (stream\ xs)$ and secondly with the outer composition $foldl_s\ (+)\ 0\ (\ldots)$.

$$\begin{aligned} & foldl_s\ (+)\ 0\ (map_s\ sq\ (stream\ xs)) \\ = \quad & \{\ \text{unfold definition of } stream\ xs\ \} \\ & map_s\ sq\ (Stream\ uncons\ xs) \\ & \quad \textbf{where} \\ & \quad\quad uncons\ [\,] \quad\quad = Nothing \\ & \quad\quad uncons\ (x : xs) = Just\ (x, xs) \\ = \quad & \end{aligned}$$

$$
\begin{aligned}
&= \quad \{ \text{ unfold definition of } map_s\ sq \ \} \\
&\quad foldl_s\ (+)\ 0\ (Stream\ next\ xs) \\
&\qquad \textbf{where} \\
&\qquad\quad next\ s = \textbf{case}\ uncons\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad Nothing \quad \to Nothing \\
&\qquad\qquad\qquad\qquad Just\ (x, s') \to Just\ (sq\ x, s') \\
&\qquad\quad uncons\ [\,] \qquad = Nothing \\
&\qquad\quad uncons\ (x : xs) = Just\ (x, xs) \\
&= \quad \{ \text{ inline } uncons \text{ and case-of-case transformation } \} \\
&\quad foldl_s\ (+)\ 0\ (Stream\ next\ xs) \\
&\qquad \textbf{where} \\
&\qquad\quad next\ s = \textbf{case}\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad [\,] \qquad \to Nothing \\
&\qquad\qquad\qquad\qquad (x : xs) \to Just\ (sq\ x, xs)
\end{aligned}
$$

The case-of-case transformation has eliminated the allocation for the inner composition between map_s and *stream*. We continue with the outer composition with $foldl_s$.

$$
\begin{aligned}
&= \quad \{ \text{ unfold definition of } foldl_s\ (+)\ 0 \ \} \\
&\quad go\ 0\ xs \\
&\qquad \textbf{where} \\
&\qquad\quad go\ a\ s = \textbf{case}\ next\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad Nothing \quad \to a \\
&\qquad\qquad\qquad\qquad Just\ (x, s') \to go\ (a + x)\ s' \\
&\qquad\quad next\ s = \textbf{case}\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad [\,] \qquad \to Nothing \\
&\qquad\qquad\qquad\qquad (x : xs) \to Just\ (sq\ x, xs) \\
&= \quad \{ \text{ inline } next \text{ and case-of-case transformation } \} \\
&\quad go\ 0\ xs \\
&\qquad \textbf{where} \\
&\qquad\quad go\ a\ s = \textbf{case}\ next\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad [\,] \qquad \to a \\
&\qquad\qquad\qquad\qquad (x : xs) \to go\ (a + sq\ x)\ xs
\end{aligned}
$$

So our final fused definition of *sumSq* is

$$
\begin{aligned}
&sumSq\ xs = go\ 0\ xs \\
&\quad \textbf{where} \\
&\qquad go\ a\ [\,] \qquad = a \\
&\qquad go\ a\ (x : xs) = go\ (a + x \times x)\ xs
\end{aligned}
$$

This is exactly the same optimal outcome as with the *unbuild*/*unfoldr* system.

Whether we should prefer the *unbuild*/*unfoldr* view or the stream view is somewhat a matter of taste. A slight advantage of the stream presentation is that many of the definitions of standard functions are syntactically superficially simpler. A pedagogical observation is that many people seem to find it easier to understand and write their own functions in this iterator style when the focus is on an intermediate object – the stream – rather than when the focus is on the construction (*unfoldr*) and consumption (*unbuild*) of lists. An aesthetic advantage is that only a single fusion rule is needed; there is no equivalent in the stream fusion system to the auxiliary *unbuild*/*unbuild* rule.

1.4.2 Streams with skip

Of course, merely reformulating the *unbuild*/*unfoldr* system to focus on the intermediate representation as streams does not solve the shortcomings of the system for functions like *filter*.

Let us revisit the core of the problem that we encountered when trying to fuse *sum* (*filter even xs*). The inner loop looked like so

$$
\begin{array}{l}
\textbf{let}\ go\ a\ s = \textbf{case}\ (\textbf{let}\ fnext\ s = \textbf{case}\ next\ s\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad Nothing \qquad\qquad\qquad \rightarrow Nothing \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad Just\ (x, s') \mid even\ x \qquad \rightarrow Just\ (x, s') \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid otherwise \rightarrow fnext\ s' \\
\qquad\qquad\qquad\quad \textbf{in}\ fnext\ s) \\
\qquad\qquad\quad \textbf{of} \\
\qquad\qquad\quad\ Nothing \quad\ \rightarrow a \\
\qquad\qquad\quad\ Just\ (x, s') \rightarrow go\ (a + x)\ s'
\end{array}
$$

The key feature is that while we had hoped for a case-of-case situation, we have a recursive **let** wrapped around the inner case expression. This was because the body of the *filter* function used a recursive stepper function

$$
\begin{array}{l}
next\ s = \textbf{case}\ next_0\ s\ \textbf{of} \\
\qquad\qquad Nothing \qquad\qquad\qquad \rightarrow Nothing \\
\qquad\qquad Just\ (x, s') \mid p\ x \qquad\quad\ \rightarrow Just\ (x, s') \\
\qquad\qquad\qquad\qquad\quad\ \mid otherwise \rightarrow next\ s'
\end{array}
$$

Recall the definition of the non-skipping stream data type

$$
\textbf{data}\ Stream\ a = \exists s.\ Stream\ (s \rightarrow Maybe\ (a, s))\ s
$$

Focus for a moment on the type of the stepper function

$$s \to \mathit{Maybe}\ (a, s)$$

The stepper function for *filter* had to be recursive because the type of the stepper function dictates that it *must* yield an element or terminate the sequence. Yet in the case that an element is filtered out we have no element to yield, we are forced to recurse until we do get an element that satisfies the filter predicate.

This suggests a possible solution: give the stepper function a third option, let it not yield an element. Let it return a new stream state, but no corresponding element of the sequence. We can describe the options with a new data type

```
data Step a s = Done
              | Skip    s
              | Yield a s
```

We then redefine *Stream* using this new *Step* type

```
data Stream a = ∃s. Stream (s → Step a s) s
```

The intended meaning of these 'skipping streams' is that skips are ignored. The abstract sequence is the elements produced via *Yield* with any intermediate *Skip* steps ignored.

Having redefined *Stream* we must adjust the *stream* and *unstream* functions to take account of *Skip*.

```
stream :: [a] → Stream a
stream xs = Stream uncons xs
  where
    uncons []       = Done
    uncons (x : xs) = Yield x xs

unstream :: Stream a → [a]
unstream (Stream next s0) = unfold next s0
  where
    unfold next s = case next s of
                      Done      → []
                      Skip    s' →    unfold next s'
                      Yield x s' → x : unfold next s'
```

Note the recursion in the *Skip* case in *unstream*[9].

[9] This recursion is not guaranteed to be productive. We will return to this issue in Section 3.6

We must also adjust the other operations to take account of the fact that input streams may skip. This is a knock-on cost for every function on streams, even those that do not introduce additional skips themselves. For example we must redefine map_s and $foldl_s$

$$
\begin{aligned}
&map_s :: (a \to b) \to Stream\ a \to Stream\ b \\
&map_s\ f\ (Stream\ next_0\ s_0) = Stream\ next\ s_0 \\
&\quad \textbf{where} \\
&\quad\quad next\ s = \textbf{case}\ next_0\ s\ \textbf{of} \\
&\quad\quad\quad\quad\quad\quad Done \to Done \\
&\quad\quad\quad\quad\quad\quad Skip\ s' \to Skip\ s' \\
&\quad\quad\quad\quad\quad\quad Yield\ x\ s' \to Yield\ (f\ x)\ s' \\
&foldl_s :: (b \to a \to b) \to b \to Stream\ a \to b \\
&foldl_s\ f\ a\ (Stream\ next\ s_0) = go\ a\ s_0 \\
&\quad \textbf{where} \\
&\quad\quad go\ a\ s = \textbf{case}\ next\ s\ \textbf{of} \\
&\quad\quad\quad\quad\quad\quad Done \to a \\
&\quad\quad\quad\quad\quad\quad Skip\ s' \to go\ a\ s' \\
&\quad\quad\quad\quad\quad\quad Yield\ x\ s' \to go\ (f\ a\ x)\ s'
\end{aligned}
$$

The *map* and *foldl* list wrappers are unchanged, though obviously they use the redefined *stream* and *unstream*.

Now with the option to skip, we can write the $filter_s$ function on *Stream* so that the stepper function does not need to recurse

$$
\begin{aligned}
&filter :: (a \to Bool) \to Stream\ a \to Stream\ a \\
&filter\ p = unstream \circ filter_s\ p \circ stream \\
&filter_s :: (a \to Bool) \to Stream\ a \to Stream\ a \\
&filter_s\ p\ (Stream\ next_0\ s_0) = Stream\ next\ s_0 \\
&\quad \textbf{where} \\
&\quad\quad next\ s = \textbf{case}\ next_0\ s\ \textbf{of} \\
&\quad\quad\quad\quad\quad\quad Done \to Done \\
&\quad\quad\quad\quad\quad\quad Skip\ s' \to Skip\ s' \\
&\quad\quad\quad\quad\quad\quad Yield\ x\ s' \mid p\ x \to Yield\ x\ s' \\
&\quad\quad\quad\quad\quad\quad\quad\quad \mid otherwise \to Skip\ s'
\end{aligned}
$$

With this definition we should revisit the *sumEven* example that failed to fuse effectively under the *unbuild*/*unfoldr* system.

The *sumEven* example

$$sumEven\ xs = sum\ (filter\ even\ xs)$$

The strategy is now completely routine. We unfold definitions, apply the stream fusion rule and then reduce the result by unfolding definitions and applying the case-of-case transformation.

$$
\begin{array}{ll}
 & sum\ (filter\ even\ xs) \\
= & \quad \{ \text{ unfold definition of } filter \text{ and } sum \ \} \\
 & foldl_s\ (+)\ 0\ (stream\ (unstream\ (filter_s\ even\ (stream\ xs)))) \\
= & \quad \{ \ stream/unstream \text{ fusion } \} \\
 & foldl_s\ (+)\ 0\ (filter_s\ even\ (stream\ xs)) \\
= & \quad \{ \text{ unfold definition of } stream \ \} \\
 & foldl_s\ (+)\ 0\ (filter_s\ even\ (Stream\ uncons\ xs)) \\
 & \quad \textbf{where} \\
 & \quad\quad uncons\ [\,] \quad\quad = Done \\
 & \quad\quad uncons\ (x : xs) = Yield\ x\ xs \\
= & \quad \{ \text{ unfold definition of } filter_s \ \} \\
 & foldl_s\ (+)\ 0\ (Stream\ next\ xs) \\
 & \quad \textbf{where} \\
 & \quad\quad next\ s = \textbf{case}\ uncons\ s\ \textbf{of} \\
 & \quad\quad\quad\quad\quad\quad Done \qquad\qquad\quad\quad\quad \to Done \\
 & \quad\quad\quad\quad\quad\quad Skip \quad s' \qquad\quad\quad\quad \to Skip \quad s' \\
 & \quad\quad\quad\quad\quad\quad Yield\ x\ s' \mid even\ x \quad\quad \to Yield\ x\ s' \\
 & \quad\quad\quad\quad\quad\quad\quad\quad\quad \mid otherwise \to Skip \quad s' \\
 & \quad\quad uncons\ [\,] \quad\quad = Done \\
 & \quad\quad uncons\ (x : xs) = Yield\ x\ xs \\
= & \quad \{ \text{ inline } uncons \text{ and case-of-case transformation } \} \\
 & foldl_s\ (+)\ 0\ (Stream\ next\ xs) \\
 & \quad \textbf{where} \\
 & \quad\quad next\ s = \textbf{case}\ s\ \textbf{of} \\
 & \quad\quad\quad\quad\quad\quad [\,] \qquad\qquad\quad\quad\quad \to Done \\
 & \quad\quad\quad\quad\quad\quad (x : s') \mid even\ x \quad\quad \to Yield\ x\ s' \\
 & \quad\quad\quad\quad\quad\quad\quad\quad\quad \mid otherwise \to Skip \quad s' \\
= &
\end{array}
$$

$$
\begin{array}{ll}
= & \{ \text{ unfold definition of } \mathit{foldl}_s \ (+) \ 0 \ \} \\
& \quad \mathit{go} \ 0 \ \mathit{xs} \\
& \qquad \textbf{where} \\
& \qquad\quad \mathit{go} \ a \ s = \textbf{case} \ \mathit{next} \ s \ \textbf{of} \\
& \qquad\qquad\qquad\quad \mathit{Done} \qquad\quad \rightarrow a \\
& \qquad\qquad\qquad\quad \mathit{Skip} \quad s' \rightarrow \mathit{go} \ a \qquad\quad s' \\
& \qquad\qquad\qquad\quad \mathit{Yield} \ x \ s' \rightarrow \mathit{go} \ (a + x) \ s' \\
& \qquad\quad \mathit{next} \ s = \textbf{case} \ s \ \textbf{of} \\
& \qquad\qquad\qquad\quad [\,] \qquad\qquad\quad\ \rightarrow \mathit{Done} \\
& \qquad\qquad\qquad\quad (x : s') \mid \mathit{even} \ x \quad\ \rightarrow \mathit{Yield} \ x \ s' \\
& \qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} \rightarrow \mathit{Skip} \quad s' \\
= & \quad \{ \text{ inline } \mathit{next} \text{ and case-of-case transformation } \} \\
& \quad \mathit{go} \ 0 \ \mathit{xs} \\
& \qquad \textbf{where} \\
& \qquad\quad \mathit{go} \ a \ s = \textbf{case} \ \mathit{next} \ s \ \textbf{of} \\
& \qquad\qquad\qquad\quad [\,] \qquad\qquad\quad\ \rightarrow a \\
& \qquad\qquad\qquad\quad (x : s') \mid \mathit{even} \ x \quad\ \rightarrow \mathit{go} \ (a + x) \ s' \\
& \qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} \rightarrow \mathit{go} \ a \qquad\quad s'
\end{array}
$$

So the overall *sumEven* definition is

$$
\begin{array}{l}
\mathit{sumEven} \ \mathit{xs} = \mathit{go} \ 0 \ \mathit{xs} \\
\quad \textbf{where} \\
\quad\quad \mathit{go} \ a \ [\,] \qquad\qquad\quad\ = a \\
\quad\quad \mathit{go} \ a \ (x : \mathit{xs}) \mid \mathit{even} \ x \quad\ = \mathit{go} \ (a + x) \ \mathit{xs} \\
\quad\quad\qquad\qquad\quad\ \mid \mathit{otherwise} = \mathit{go} \ a \qquad\quad \mathit{xs}
\end{array}
$$

This is another optimal result: all the list allocations have been eliminated.

This example is one where stream fusion fares better than either of the *foldr/build* or *unbuild/unfoldr* fusion systems. Recall that *foldr/build* was unable to fuse *foldl*, while *unbuild/unfoldr* could not effectively fuse *filter*.

Note that the reformulation as streams is independent from the addition of skip. The *unbuild/unfoldr* system could also be extended with skip, in which case we would expect it to handle *filter* successfully.

Worries about skip

There are several legitimate worries we might have about skip.

1. It is just an ad hoc extension to handle *filter*? Will we need other extensions for other functions or can all functions on streams now be written with a non-recursive stepper function?

2. Skip adds extra complexity to each function that works with streams. Is it worth it?

3. Skip adds extra complexity to the theory. It is no longer obvious that skipping streams are equivalent to lists.

The long answer to these questions are given in Chapters 3 and 4. The short answer is that it turns out that skip is a general extension, that it helps with functions other than *filter* and further that it enables very many stream functions to be written with non-recursive stepper functions. It is true that skip makes the theory more difficult but we will show in Chapter 3 that there is a suitable theoretical explanation. It is also true that skip complicates the definition of functions on streams however the treatment of skip in input streams is entirely uniform so the extra complexity is not especially great. Overall, adding skip does seem to be worth it. It appears to be essential to effectively fuse a number of common functions.

1.4.3 Writing stream functions

We have already seen a few examples of stream versions of list functions: map_s, $filter_s$ and $foldl_s$. These examples are relatively simple and they process sequences in a very regular way. It is reasonable to wonder how we might write more complicated stream functions that process sequences in less regular ways. Indeed, given that we have banished some uses of recursion we may wonder whether streams are expressive enough to write many interesting functions at all.

The first observation we can make is that most list functions defined in terms of *unfoldr* can be easily translated into stream versions. The only limitation is that the *unfoldr* stepper function must not be recursive. This easy translation helps us only so far because despite encouragement to use *unfoldr* (Gibbons and Jones, 1998), it remains a rather infrequently used tool in the functional programmer's toolbox.

When writing functions that produce or consume ordinary lists we are able to use an expressive and flexible style. We are able to make use of mutually recursive functions to handle switching between modes. The usual variable scoping rules allow variables defined in an outer scope to be used in inner recursive functions.

By contrast, writing functions for the *unbuild*/*unfoldr* and stream fusion systems is harder because it constrains us to use a single stepper function. We have what is sometimes called an 'inversion of control'. With recursive list functions we simply pass parameters to functions. With a stepper function on the other hand, any information needed for future steps has to be stashed away in the state and then retrieved in subsequent steps. Stream fusion makes things harder still by requiring that the stepper function be non-recursive.

Modes and state shapes

The trick to writing stepper functions within the restrictions is to make liberal use of *structure* in the stream state and liberal use of *Skip*. This helps particularly with functions that need to construct sequences in an irregular way, switching between a number of 'modes'. Traditionally one might use mutually recursive functions to handle multiple modes, and straightforward function calls to switch between modes.

$$
\begin{aligned}
&\mathit{mode1} :: \mathit{Int} \to \mathit{Int} \to [\mathit{Int}] \\
&\mathit{mode1}\ a\ b \mid p\ a = \mathit{mode2}\ (f\ b) \\
&\mathit{mode2} :: \mathit{Int} \to [\mathit{Int}]
\end{aligned}
$$

With a stepper function, multiple modes can be represented by using a data type for the stream state that has multiple alternatives, one for each mode. Each mode can have a separate set of variables.

$$
\begin{aligned}
&\textbf{data}\ \mathit{State} = \mathit{Mode1}\ \mathit{Int}\ \mathit{Int} \\
&\qquad\qquad \mid\ \mathit{Mode2}\ \mathit{Int}
\end{aligned}
$$

The stepper function can be defined separately for each mode.

$$
\begin{aligned}
&\mathit{next}\ (\mathit{Mode1}\ a\ b) = \ldots \\
&\mathit{next}\ (\mathit{Mode2}\ c) \quad = \ldots
\end{aligned}
$$

Instead of passing parameters to functions, the stepper function can return *Skip* specifying a mode and the parameters for that mode.

$$\mathit{next}\ (\mathit{Mode1}\ a\ b) \mid p\ a = \mathit{Skip}\ (\mathit{Mode2}\ (f\ b))$$

In this style, local recursive functions have to be lifted to the top level and given their own mode. The changes involved are rather akin to defunctionalisation (Reynolds, 1972; Danvy and Nielsen, 2001).

In Chapter 3 we will see that stream modes, and transitions between modes, turn out to be relevant for the structure of inductive proofs about individual stream functions.

In Chapter 4 we will refer the data constructors representing the different modes as 'state shapes'. This is because the constructors provide the top level structure, or shape, of the stream state. We will also find that it is useful to have state shapes that use nested applications of constructors.

Example

Let us look at an example of a stream function with a non-trivial stream state, and correspondingly with multiple state shapes. This also serves as an informal example of the derivation of a stream version of a list function.

The *init* function is a standard list function that returns the initial elements of a list, or more precisely all elements except the last element. The list version can be written simply as

$$\begin{aligned} &\mathit{init}\ [x] &&= [\,] \\ &\mathit{init}\ (x : xs) &&= x : \mathit{init}\ xs \end{aligned}$$

Note that the $[\,]$ case is an error, which is sometimes written explicitly

$$\mathit{init}\ [\,] = \mathit{error}\ \texttt{"init: empty list"}$$

This simple definition belies an inefficiency: the function looks one list element beyond what it consumes in each iteration. This fact is clearer if we desugar the pattern matching of the three clauses above

$$\begin{aligned} \mathit{init}\ xs = \mathbf{case}\ xs\ \mathbf{of}& \\ &[\,] &&\to \mathit{error}\ \texttt{"init: empty list"} \\ &(x : xs') &&\to \mathbf{case}\ xs'\ \mathbf{of} \\ &&&\quad [\,] &&\to [\,] \\ &&&\quad (\_ : \_) &&\to x : \mathit{init}\ xs' \end{aligned}$$

Notice in the last case, that the recursive call uses xs' which is known to be of the pattern $(\_ : \_)$.

This observation motivates the following optimised[10] definition

$$
\begin{array}{lll}
\mathit{init} & [\,] & = \mathit{error}\ \texttt{"init: empty list"} \\
\mathit{init} & (x : xs) & = \mathit{init}'\ x\ xs \\
\mathit{init}'\ x & [\,] & = [\,] \\
\mathit{init}'\ x & (x' : xs) & = x : \mathit{init}'\ x'\ xs
\end{array}
$$

This version has two modes, one mode for the very first list element and a second mode for the remaining elements. Note that in the second mode we hold on to the preceding list element rather than having to look ahead. Crucially, this version only deconstructs one list cell per iteration rather than two in the original. This optimised definition can be directly translated into a stream version

$$
\begin{array}{l}
\mathit{init}_s :: \mathit{Stream}\ a \to \mathit{Stream}\ a \\
\mathit{init}_s\ (\mathit{Stream}\ \mathit{next}_0\ s_0) = \mathit{Stream}\ \mathit{next}\ (\mathit{Nothing}, s_0) \\
\quad \textbf{where} \\
\quad\quad \mathit{next}\ (\mathit{Nothing}, s) = \textbf{case}\ \mathit{next}_0\ s\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{Done} \to \mathit{error}\ \texttt{"init: empty stream"} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{Skip}\ s' \to \mathit{Skip}\ (\mathit{Nothing}, s') \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{Yield}\ x\ s' \to \mathit{Skip}\ (\mathit{Just}\ x, s') \\
\quad\quad \mathit{next}\ (\mathit{Just}\ x, s) = \textbf{case}\ \mathit{next}_0\ s\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{Done} \to \mathit{Done} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{Skip}\ s' \to \mathit{Skip}\ (\mathit{Just}\ x, s') \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{Yield}\ x'\ s' \to \mathit{Yield}\ x\ (\mathit{Just}\ x', s')
\end{array}
$$

We have two state shapes: $(\mathit{Nothing}, s)$ and $(\mathit{Just}\ x, s)$.

Worrying about allocations

While in the above example we have been able to derive a stream version of a list function, we do not provide a general translation of list-producing functions into equivalent stream-producing functions[11]. Of course in a trivial sense there is always a translation: simply wrap *stream* around an existing list producer. Since the purpose of stream fusion is to reduce allocations however then what we are really interested in is a translation that uses the same number of allocations. A stream version that uses additional internal data structures is not acceptable.

[10] This improved definition can be derived automatically; Peyton Jones (2007) uses the similar example of *last* as the introductory example to motivate call-pattern specialisation.

[11] In Section 4.8.4 we do describe a partial translation that works for many cases.

There are other situations where we must worry about allocations: we must not traverse a stream multiple times, including limited 'lookahead'. In the *init* example above we were able to transform the function so that instead of looking ahead it stores extra information in the stream state. The reason is that while lists are memoised, streams are not: calling a stepper function twice with the same input state will repeat the work and any allocations for that step. We consider this and similar issues in Chapter 4.

Given that we must worry about the allocations of stream versions in comparison to their list counterparts, we must be concerned about using extra allocations to represent stream state shapes. It would certainly be unfortunate if our main technique to gain expressiveness in stream functions leads to using more allocations thus neutralising any saving from stream fusion.

The solution to this conundrum is to promise to eventually eliminate all the allocations for the data constructors used to represent the state shapes. A significant portion of the optimisation argument in Chapter 4 is dedicated to substantiating this promise.

Chapter 2

Technical preliminaries

The purpose of this chapter is to present the major concepts that we make use of in the subsequent chapters. In particular this chapter introduces the syntax and semantics of System F and the representation of data structures within System F. Haskell and CPOs are also introduced briefly.

This chapter is not a standalone introduction to these topics, rather the purpose is to make clear which aspects we need for the subsequent chapter and to exhibit the notation we will use.

2.1 System F

For the initial exploration of proofs for shortcut fusion systems, including stream fusion, it helps to use a theoretical model that is simpler than the real programming language in which we want to apply the results. Using a simpler theoretical model helps to distinguish the essential aspects from the incidental aspects and helps to make any results more portable to other similar real programming languages.

We are working in the context of functional programming, which means some extension of the lambda calculus. Specifically, we wish to apply fusion results in the functional programming language Haskell. Since Haskell is typed we must use a typed lambda calculus for our theoretical model. System F (Girard et al., 1989, Chapter 11), also known as the polymorphic lambda calculus, is the natural choice because it is relatively simple, well understood and expressive enough to handle the Hindley-Milner type system on which Haskell's own type system is based. System F, like simply-typed lambda calculus, is an explicitly typed language. It is also strongly normalising: evaluation always terminates and evaluation order does not matter.

System F is a good initial setting to explore shortcut fusion. Despite being relatively simple, System F allows us to encode data types and there are semantic models for System F which allow us to prove strong properties about these encodings. Using encodings we can express familiar types like sums and products, and more interestingly, recursive types like lists and trees.

The encodings of recursive data types in System F clarifies the distinction between the view of data used by *foldr/build* fusion and the view used by *unbuild/unfoldr* fusion. In System F the types of data used by each system is actually completely distinct whereas in standard functional programming languages the two fusion systems merely take different views on the same types of data.

2.1.1 Syntax

System F extends simply-typed lambda calculus with a universal type quantifier ($\forall a.\ \ldots$) and two additional term-level constructs: universal abstraction ($\Lambda a \rightarrow \ldots$) and universal application. This directly expresses parametric polymorphism and terms that have universally quantified types.

A good example is the definition and use of the polymorphic identity function. That is, an identity function that works for all types[1]. Consider the monomorphic identity function in simply-typed lambda calculus, for a specific type A:

$$\lambda(x :: A) \rightarrow x$$

In simply-typed lambda calculus we must annotate lambda-bound variables with their type. Some presentations also annotate variables at their use sites. This improves clarity in some circumstances but is not strictly necessary. The term above has the type $A \rightarrow A$ and we write it as

$$\lambda(x :: A) \rightarrow x \ \ :: A \rightarrow A$$

The polymorphic lambda calculus allows us to express the polymorphic identity function. We use the universal type quantifier and type variables to describe the type, namely $\forall a.a \rightarrow a$. In the term, the parametrisation by a type is made explicit by passing the type as an extra argument. This is directly analogous to lambda abstraction for parametrising terms by values.

[1] Another view is to say that we have a family of identity functions, one for each type.

Universal abstraction parametrises terms by types:

$$\Lambda a \to \lambda(x :: a) \to x \;\; :: \; \forall a.\, a \to a$$

When we use this polymorphic identity function we have to supply both a type parameter and a value parameter.

For value parameters we use ordinary term application and for type parameters we use universal application. To distinguish universal application from ordinary term application, some presentations distinguish type variables from value variables by the use of upper or lower case characters. We follow the Haskell convention of using lower case for type variables and upper case for type constants. Since we use lower case for both term variables and type variables, there is the potential that the distinction between universal application and ordinary term application is somewhat unclear. It can be distinguished however since type variables are bound by a universal abstraction while term variables are bound by lambda abstraction. In addition, we will typically use $A, B, C, \ldots$ and $a, b, c, \ldots$ for type constants and variables respectively. For term variables we will use f, g, h, k for functions and x, y, z for other parameters. Alternatively we will use longer more descriptive names such as *map*, *fold* etc.

We will sometimes use subscripts on names. This is to be understood not as some kind of parameter but as a tag that is part of the name. In particular we use subscript s to distinguish stream versions of functions such as map_s and *fold_s* from their ordinary counterparts.

Given the polymorphic identity function, we can obtain a monomorphic version at type $A \to A$ by applying the polymorphic version to the type A:

$$(\Lambda a \to \lambda(x :: a) \to x)\; A \;\; :: \;\; (a \to a)\; [a := A]$$

The usual β-reduction and substitution applies to give us the monomorphic version

$$\lambda(x :: A) \to x \;\; :: \; A \to A$$

In the following sections where we use System F we will take a few liberties for the sake of presentation. We will sometimes omit explicit typing where it is not ambiguous. We will use an equational style where we give explicit names for terms. For example we can name the polymorphic identity function *id* and define it using the syntax

$$id = \Lambda a \to \lambda(x :: a) \to x$$

Uses of such names are to be understood as their corresponding terms, using normal capture-avoiding substitution. Note that since these names are part

of the meta-language and not the concrete syntax of System F, these names cannot cannot appear in the right hand side of their own definitions. Similarly, mutually recursive definitions are not allowed. It must always be possible to expand all the 'syntactic sugar' to get raw System F syntax.

We will sometimes specify the type for a named term separately from the definition of the term itself, such as

$$id :: \forall a.\, a \to a$$
$$id = \Lambda a \to \lambda(x :: a) \to x$$

Indeed we will occasionally give a partial specification of a named term by giving just the name and its type.

We will also use the syntactic convention that definitions such as

$$f\ x\ y = \ldots$$

with variables on the left hand side are understood as lambda abstractions (or universal abstractions as appropriate) on the right hand side

$$f = \lambda x \to \lambda y \to \ldots$$

We will make frequent use of pairs, using the notation $(x, y) :: (A, B)$, where $x :: A$ and $y :: B$. This notation does not add anything new since pairs can be encoded in System F. We will cover the details of the encoding shortly.

2.1.2 Parametricity

In lambda calculus and its many extensions, there are many proofs we can do using just syntax. For example proving a particular term has a particular type, or that two terms are equivalent by reduction rules are things we can do using just the syntactic rules of the language.

As a concrete example, we can prove that we can encode pairs in System F. We do this by inventing a suitable encoding and then proving that the encoding has the usual properties that we expect of pairs, namely that we can put two things in and then get them out again. In particular, in System F the encoding for a pair type (A, B) uses terms of type

$$(A, B) = \forall c.\, (A \to B \to c) \to c$$

We can show syntactically that there are terms of this type and that we can legitimately call them pairs. Going further, we might hope that *every* term of this type is a pair. We cannot prove it however if we only stick to syntactic rules.

A statement about *all* members of some type is a semantic property, not a syntactic one. It is the semantic model that describes the full range of possible values, and in a typed language the range of values in each type. In a sense, the syntax of terms gives a lower bound on what is in each type and the semantics gives the upper bound. It is quite possible to construct different semantics for the same syntax that admit different ranges of possible values in the same type.

As a simple example, consider the polymorphic identity function in System F

$$\Lambda a \to \lambda(x :: a) \to x \;\; :: \; \forall a.\, a \to a$$

There is a straightforward syntactic proof that this term has the described type. We could construct other terms with the same type, though all the ones we could construct would be equivalent by reduction rules and thus any sensible denotational semantics must assign them the same value. So although we can use syntax to show the is at least one value in the type $\forall a.\, a \to a$, the full range of values depends on the semantic model that we pick. It is quite possible to pick a semantics that says that there is a greater range of values in a type than we can actually construct using terms in the syntax. For instance, is it possible to have an $f :: \forall a.\, a \to a$ such that $f\ 3 = 4$? This depends on the semantic model we pick. We could imagine a model which interprets elements of $\forall a.\, a \to a$ as being a family of functions $a \to a$ for each type a, with no particular restrictions. In this interpretation there is no problem with each member of the family doing something different on each type. This is known as ad hoc polymorphism. Languages that have a 'type case' construct allow ad hoc polymorphism.

The intention and intuition for System F types like $\forall a.\, a \to a$ is that there is a degree of uniformity in how values of different types are treated. This is the intuitive notion of parametricity, that parametric polymorphic functions behave the same at each type. Conversely, polymorphic functions cannot make ad hoc distinctions based on specific types. This notion would outlaw $f\ 3 = 4$ because it is doing something special for integers that cannot be done at all other types. Our intention with the type $\forall a.\, a \to a$ is that the *only* value of this type is the identity function (or the family of identity functions, one for each type). A parametric semantic model would give us this guarantee.

There are several parametric models of System F, including PERs (Bainbridge et al., 1990) and frame models (Bruce et al., 1990). Fortunately the details of each model are not crucial; the specific properties in which we are interested can be derived from the definition of parametricity without reference to the details of the underlying model.

2.1.3 Free theorems

An important observation is that parametricity tells us particular properties about the values of particular types. This observation was made by Wadler (1989) who coined the term *free theorems* for properties about specific types obtained via parametricity.

In the next chapter we rely in several places on free theorems obtained from parametric polymorphic types so we now review the details of the derivation technique.

There is more than one way of formalising parametricity (Plotkin and Abadi, 1993, Section 2.2). The free theorems technique uses the standard relational formalisation of parametricity. The approach is to consider types as relations. The relations are built up following the structure of the type. The parametricity property is then stated as a membership property on these relations.

The free theorem for a type is derived by starting with the relation corresponding to the type. The definition of parametricity on that relation is then unfolded step by step according to the structure of the relation. The final step is to specialise the property on relations to a corresponding property on functions[2].

To illustrate the technique we will derive the free theorem for the type of the polymorphic identity function. We start with some arbitrary value g of the type in question

$$g :: \forall a.a \to a$$

We must first give the relation corresponding to this type. It is built using the $\to$ and $\forall$ connectives. Wadler (1989, Section 2) gives the rules for interpreting these operations on relations. We write $\mathcal{A} :: A \leftrightarrow A'$ which should be read as a relation $\mathcal{A}$ between the type A and the type A'.

Definition 2.1.1. The relation corresponding to a base type such as *Int* is the identity relation $\mathcal{I}_{Int} :: Int \leftrightarrow Int$

$$(x, x') \in \mathcal{I}_{Int} \iff x = x'$$

Definition 2.1.2. For a relation $\mathcal{A} :: A \leftrightarrow A'$ and relation $\mathcal{B} :: B \leftrightarrow B'$ the relation $\mathcal{A} \to \mathcal{B} :: (A \to B) \leftrightarrow (A' \to B')$ is defined by

$$\begin{array}{l} (f, f') \in \mathcal{A} \to \mathcal{B} \\ \iff \\ \textit{for all } (x, x') \in \mathcal{A}, (f\ x, f'\ x') \in \mathcal{B} \end{array}$$

[2]This is because we are only interested in functions. The original property on relations is useful in other contexts.

Definition 2.1.3. For a parametrised relation $\mathcal{F}\ \mathcal{A} :: F\ A \leftrightarrow F'\ A'$, the relation $\forall\mathcal{X}.\mathcal{F}\ \mathcal{X} :: \forall X.F\ X \leftrightarrow \forall X'.F'\ X'$ is defined by

$$
\begin{array}{l}
(f, f') \in \forall\mathcal{X}.\mathcal{F}\ \mathcal{X} \\
\iff \\
\textit{for all } \mathcal{A} :: A \leftrightarrow A', (f\ A, f'\ A') \in \mathcal{F}\ \mathcal{A}
\end{array}
$$

The parametricity property itself is stated as:

Definition 2.1.4 (Parametricity)**.** If $t :: T$ and $\mathcal{T}$ is the relation corresponding to the type T then $(t, t) \in \mathcal{T}$.

We start by writing down the parametricity property for g and then unfolding the membership definition for the relation corresponding to the type of g

$$
\begin{array}{ll}
 & (g, g) \in \forall\mathcal{X}.\mathcal{X} \to \mathcal{X} \\
\iff & \quad \{ \text{ unfold membership definition for a parametrised relation } \} \\
 & \textit{for all } \mathcal{A} :: A \leftrightarrow A', \\
 & \quad (g\ A, g\ A') \in \mathcal{A} \to \mathcal{A} \\
\iff & \quad \{ \text{ unfold membership definition for a } (\to) \text{ relation } \} \\
 & \textit{for all } \mathcal{A} :: A \leftrightarrow A', \\
 & \quad \textit{for all } (x, x') \in \mathcal{A}, \\
 & \quad\quad (g\ A\ x, g\ A'\ x') \in \mathcal{A}
\end{array}
$$

Now instead of a relation $\mathcal{A} :: A \leftrightarrow A'$ we want to specialise to a function $f :: A \to A'$. Where we had $(x, x') \in \mathcal{A}$, we now get $f\ x = x'$.

$$
\begin{array}{l}
\textit{for all } f :: A \to A', \\
\quad \textit{for all } x :: A, \\
\quad\quad f\ x = x' \\
\quad\quad \implies \\
\quad\quad f\ (g\ A\ x) = g\ A'\ x'
\end{array}
$$

We can simplify this by substituting x' and using function composition

$$
\begin{array}{l}
f\ x = x' \implies f\ (g\ A\ x) = g\ A'\ x' \\
\implies \\
f\ (g\ A\ x) = g\ A'\ (f\ x) \\
\implies \\
f \circ g\ A = g\ A' \circ f
\end{array}
$$

So the overall statement of the free theorem for g is

$$\begin{array}{l} \textit{for all } f :: A \to A', \\ \quad f \circ g\ A = g\ A' \circ f \end{array}$$

and note that this holds for arbitrary types A and A'.

Given the free theorem we can now see that $g = id$. We pick f as a constant function $const\ a'$ for some $a' :: A'$

$$\begin{array}{ll} & const\ a' \circ g\ A = g\ A' \circ const\ a' \\ \Longrightarrow & \quad \{ \text{ apply both sides to some } a :: A \ \} \\ & (const\ a' \circ g\ A)\ a = (g\ A' \circ const\ a')\ a \\ \Longrightarrow & \quad \{ \text{ unfold } const \text{ and } \beta\text{-reduce } \} \\ & a' = g\ A'\ a' \end{array}$$

Hence g is a polymorphic identity function, which is unique up to isomorphism.

2.1.4 Existential types

We will make use of the fact that System F lets us use existentially quantified types (Girard et al., 1989, Section 11.3.5). Existential quantification can be defined in System F in terms of universal quantification by the following encoding

$$\exists a.T\ a = \forall b.(\forall a.T\ a \to b) \to b$$

We can construct terms of this type using the following form

$$\Lambda b \to \lambda(f :: \forall a.T\ a \to b) \to f\ A\ x$$

Where x is some term of type $T\ A$. We use a pair notation (T A, x) for these terms. That is, given $x :: T\ A$ we define

$$\begin{array}{l} (T\ A, x) :: \exists a.T\ a \\ (T\ A, x) = \Lambda b \to \lambda(f :: \forall a.T\ a \to b) \to f\ A\ x \end{array}$$

This pair notation can be distinguished from ordinary pair terms since for an existential type, the first component of the pair is a type.

A way to understand this encoding is to consider how to use a value of an existentially quantified type. For the sake of being concrete, imagine we have a value e of type $\exists a.a$. We cannot do anything useful with this type of course, but it is instructive to see exactly why.

We are interested in consuming our value $e :: \exists a.a$ to produce a value of some concrete type B. Expanding the encoding we get $e :: \forall b.(\forall a.a \to b) \to b$. So e is a function expecting a type parameter and a 'consumer' function. We will supply our concrete type B and a consumer function of type $\forall a.a \to B$. Due to parametricity, all functions with type $\forall a.a \to B$ must be constant functions – which explains why we cannot do anything useful with $e :: \exists a.a$. Let us pick a constant function that returns a particular $y :: B$. Thus our consumer term is

$$(\Lambda a \to \lambda(\_ :: a) \to y) \;\; :: \;\; \forall a.a \to B$$

We apply e to the result type B and the consumer function:

$$e\ B\ (\Lambda a \to \lambda(\_ :: a) \to y) \;\; :: \;\; B$$

A key thing to notice is that the 'real' internal type of the value is only available within the scope of the consuming lambda function (as the type parameter a). Of course the lambda function is required to be polymorphic in that type, so it must be prepared to consume the value irrespective of the concrete type parameter. Note also that the result of the consuming function cannot use the 'real' internal type because it must be polymorphic in it and the result type cannot be parametrised by it because the scope of the type variables does not make that possible. Thus the 'real' internal type of the value is fully encapsulated.

A somewhat more interesting example is $p :: \exists a.(a, a \to Bool)$ since then at least we can do something with the existentially typed value; we can apply the function to the value.

$$\begin{aligned}
&p &&:: \exists a.(a, a \to Bool)\\
&consume &&:: \forall a.(a, a \to Bool) \to Bool\\
&consume &&= \Lambda a \to \lambda((x,f) :: (a, a \to Bool)) \to f\ x\\
&p\ Bool\ consume &&:: Bool
\end{aligned}$$

In addition to consuming, we need to be able to construct terms with an existential type. Let us take the second example, namely $p :: \exists a.(a, a \to Bool)$. When we invent the term of this type we get to pick the concrete 'real' type. So we want a term p with the type

$$p :: \forall b.(\forall a.(a, a \to Bool) \to b) \to b$$

Following the structure of the type, we need to start with a type lambda

$$p = \Lambda b \to \ldots$$

Next is the function argument of type $\forall a.(a, a \to Bool) \to b$

$$p = \Lambda b \to \lambda(f :: \forall a.(a, a \to Bool) \to b) \to \ldots$$

Since the result must be of type b we must call f with suitable parameters. The function f expects a type argument and a pair of value and function. Here is where we get to choose the type argument and also the pair of values using whatever type we choose. For example let us pick *Int* and the pair $(3, \lambda n \to n \geqslant 0)$:

$$p = \Lambda b \to \lambda(f :: \forall a.(a, a \to Bool) \to b) \to f\ Int\ (3, \lambda n \to n \geqslant 0)$$

This gives us an instance of the general form mentioned at the beginning of this section. Using the pair notation we can rewrite p as just

$$p = (Int, (3, \lambda n \to n \geqslant 0))$$

To take such values apart we use a lambda pattern notation:

$$h :: \exists a.(a, a \to Bool) \to \ldots$$
$$h = \lambda(a, (x, f) :: (a, a \to Bool)) \to \ldots$$

This translates as

$$h :: \exists a.(a, a \to Bool) \to \ldots$$
$$h = \lambda p \to p\ (\Lambda a \to \lambda((x, f) :: (a, a \to Bool)) \to \ldots)$$

2.1.5 Categorical view

Notation and concepts from category theory are used to describe various constructions in System F, in particular the semantics of data types.

Wraith (1989) argues that a good semantic model for System F should implement Hagino's notion of a categorical programming language. A categorical programming language (Hagino, 1987) is required to provide various type constructors with appropriate corresponding categorical properties. For example, data types for sums and products are required to be proper categorical sums and products. For System F these various types are implemented by encodings in terms of the basic System F syntax. We will cover the most significant constructions in the next section.

The category for System F has types as the objects and functions as the morphisms. In this context a functor F is a type parametrised by a type variable.

Given a function $f :: A \to B$ we can obtain a corresponding 'lifted' function $f' :: F\ A \to F\ B$. As is required for F to be a categorical functor, the functions we can obtain by lifting preserve identities and function composition.

The traditional notation for a lifted function is

$$F\ f :: F\ A \to F\ B$$

This notation uses the name of the functor at the term level as well as at the type level. When using Haskell notation we will instead use the functor name only at the type level and use the special term *fmap* to lift functions.

$$fmap\ f :: F\ A \to F\ B$$

Which functor is being used for the lifting is determined by the type[3].

2.2 Representation of data

The notion of data is of fundamental importance in programming. A syntax and semantics for data is directly supported in standard functional programming languages. Lambda calculus does not have data as primitive. Despite this, the notion of data originates with lambda calculus. The notion crops up naturally via encodings in terms of lambda functions. The encoding gives rise to a semantics and that semantics can be captured abstractly. It is this abstract semantics that is implemented in standard functional programming languages in terms of efficient machine primitives.

Though the meaning of data can be precisely described in abstract terms without reference to the lambda calculus encoding, many of the patterns and properties of functions that manipulate data are closely related to the lambda calculus encoding. It thus behoves us to study the encoding.

The standard encoding is called the *Church* encoding, named after Alonzo Church who discovered it. In the Church encoding, data values are represented by higher-order functions for consuming the data. Where standard functional languages provide a special **case** syntax for consuming data, such as

case e **of**
  $C1\ x\ y \to \ldots$
  $C2\ z \quad \to \ldots$

[3]Note therefore that when using this Haskell notation we are limited to one functor per type.

In the Church encoding, the equivalent is achieved by applying the 'representation function' (the value e in this example) to suitable functions that are prepared to consume the sub-components of the data structure

$$\begin{array}{l} e\ (\lambda x\ y \to \ldots) \\ \quad (\lambda z \quad \to \ldots) \end{array}$$

The Church encoding also works in System F. Indeed it works particularly nicely in System F because the encodings for various structures have precisely corresponding types. In the context of System F, this form of data is called a free structure (Girard et al., 1989, Section 11.4).

The kinds of data types that we can define using the Church encoding correspond to the algebraic data types that standard functional programming languages provide as the primary mechanism for user-defined data types. These types include products (records/tuples), sums (alternatives) and recursive types like lists and trees. It does not include types that are added to languages as primitives such as machine integers or arrays.

Girard et al. (1989, Section 11.3–11.5) gives a detailed yet accessible introduction to the encoding of data within System F. In the remainder of this section we give a brief presentation.

2.2.1 Products and sums

A pair type, consisting of types A and B, can be encoded using terms of type $\forall c.\ (A \to B \to c) \to c$. Note that we use the notation (A, B) for pair types rather than the traditional $A \times B$. We thus define

$$(A, B) = \forall c.\ (A \to B \to c) \to c$$

At the term level, a pair $(x, y) :: (A, B)$ is encoded as the term

$$(\Lambda c \to \lambda (f :: A \to B \to c) \to f\ x\ y) \ :: \ \forall c.\ (A \to B \to c) \to c$$

We can define a general pair constructor and first and second projections

$$\begin{array}{ll} \mathit{pair} & :: \forall a\ b.\ a \to b \to (\forall c.(a \to b \to c) \to c) \\ \mathit{pair}\ a\ b\ x\ y & = \Lambda c \to \lambda f \to f\ c\ x\ y \\ \mathit{fst} & :: \forall a\ b.\ (\forall c.(a \to b \to c) \to c) \to a \\ \mathit{fst}\ a\ b\ p & = p\ a\ (\lambda x\ y \to x) \\ \mathit{snd} & :: \forall a\ b.\ (\forall c.(a \to b \to c) \to c) \to b \\ \mathit{snd}\ a\ b\ p & = p\ b\ (\lambda x\ y \to y) \end{array}$$

To be sure that these encodings behave like pairs we would wish to show that $fst\ a\ b\ (pair\ a\ b\ x\ y) = x$ and that $snd\ a\ b\ (pair\ a\ b\ x\ y) = y$. This is trivial by β-reduction.

We would also like to know that *all* values of type $\forall c.\ (A \to B \to c) \to c$ correspond to the encoding of some pair. That is we would like to show that $pair\ (fst\ p)\ (snd\ p) = p$ for all $p :: \forall c.\ (A \to B \to c) \to c$. This can be proved using parametricity. For example, Plotkin and Abadi (1993, Section 3.1) show that a formulation of parametricity in terms of dinatural transformations can be derived from the standard relational formulation and use dinaturality to prove the above property for pairs.

The encoding for alternatives, or sums, is similar in style. The sum type $A + B$ is encoded using terms of type

$$A + B = \forall c.\ (A \to c) \to (B \to c) \to c$$

The constructors *left* and *right*, and the $case_{sum}$ deconstructor are defined as

$$\begin{aligned}
&left \quad :: \forall a\ b.\ a \to (\forall c.(a \to c) \to (b \to c) \to c)\\
&left \quad x = \lambda l\ r \to l\ x\\
&right \quad :: \forall a\ b.\ b \to (\forall c.(a \to c) \to (b \to c) \to c)\\
&right\ y = \lambda l\ r \to r\ y\\
&case_{sum} \quad :: \forall a\ b\ c.\ (a \to c) \to (b \to c)\\
&\qquad\qquad \to (\forall c.(a \to c) \to (b \to c) \to c) \to c\\
&case_{sum}\ s\ l\ r = s\ l\ r
\end{aligned}$$

The proofs that $case_{sum}\ (left\ x)\ l\ r = l\ x$ and $case_{sum}\ (right\ x)\ l\ r = r\ x$ are again syntactic and straightforward. The proof that all values in the sum type correspond to encodings of sums requires parametricity.

2.2.2 Free structures

The appropriate generalisation of simple binary sums and products is a free structure (Girard et al., 1989, Section 11.4). A free structure is an abstract notion of a sum-of-products data structure, which in System F we can implement using a Church encoding.

A free structure D is described by zero or more *data constructors* which are functions $f_0, f_1, \ldots, f_n$ with corresponding types $S_0, S_1, \ldots, S_n$. Each type S_i is of the form

$$S_i = T_{i,0} \to T_{i,1} \to \ldots \to T_{i,k_i} \to D$$

That is, we can have any finite number of constructors and each constructor can have any finite number of arguments. Each data value of the structure D must be represented uniquely by application of the data constructor functions. The uniqueness property is necessary for the possibility of taking a data value apart without losing anything.

As an example, consider a (parametrised) free structure *Maybe b* with two data constructors *Nothing* and *Just*

$$\begin{aligned} &\mathit{Nothing} :: \mathit{Maybe} \\ &\mathit{Just} \quad\quad :: b \to \mathit{Maybe} \end{aligned}$$

In System F, a free structure D is encoded using terms of type

$$D = \forall a.\, S_0\ [D := a] \to S_1\ [D := a] \to \ldots \to S_n\ [D := a] \to a$$

where each S_i is as defined above and all occurrences of D are replaced by the type variable a. All terms of this type correspond to applications of the data constructors $f_0 \ldots f_n$ (Girard et al., 1989, Section 11.4.3, 15.1.1).

For example, for the free structure *Maybe b* we have two constructors with types S_0 and S_1 which are

$$\begin{aligned} &S_0\ a = a \\ &S_1\ a = b \to a \end{aligned}$$

so that the overall type *Maybe b* is encoded as

$$\mathit{Maybe}\ b = \forall a.\, a \to (b \to a) \to a$$

There are various special cases of free structures that have special notation. Arbitrary sized products and sums are special cases of free structures, the types of which we write as $(A, B, C, \ldots)$ and $A + B + C \ldots$ respectively. There are also a couple interesting degenerate cases. A free structure with one data constructor with no parameters is encoded using type $\forall c.c \to c$ and we sometimes write it as type $\mathbf{1}$. Similarly, a free structure with no data constructors is encoded using type $\forall c.c$ and we write it as type $\mathbf{0}$. There are no terms of type $\mathbf{0}$ and there is exactly one term of type $\mathbf{1}$ (namely the polymorphic identity function). The types $\mathbf{0}$ and $\mathbf{1}$ are the initial and final types in the category of types and functions[4]. Where we need to write the term of type $\mathbf{1}$ we use 1, that is we write $1 :: \mathbf{1}$.

The really interesting encodings are for inductive data structures such as natural numbers, lists and trees. These are also cases of free structures.

[4]For this reason some authors write the types $\mathbf{0}$ and $\mathbf{1}$ as $\bot$ and $\top$ respectively.

Recall that a free structure D has constructor functions f_i with types S_i of the form

$$S_i = T_{i,0} \to T_{i,1} \to \ldots \to T_{i,k_i} \to D$$

The definition of a free structure allows D to occur in the types $T_{i,j}$. This is subject to the restriction that D only appears in *positive* positions, meaning to the left of an even number of function arrows.

There is no special change in the System F encoding for a free structure with positive occurrences of D in the types $T_{i,j}$ – *all* occurrences of D in the types S_i are replaced by the type variable a.

$$D = \forall a.\ S_0\ [D := a] \to S_1\ [D := a] \to \ldots \to S_n\ [D := a] \to a$$

The most famous example of this kind of free structure is the Peano natural numbers. We can define the free structure *Nat* with two data constructors

$$\mathit{Zero} :: \mathit{Nat}$$
$$\mathit{Succ} :: \mathit{Nat} \to \mathit{Nat}$$

The corresponding encoding is exactly the standard Church encoding of the natural numbers.

$$\mathit{Nat} = \forall a.a \to (a \to a) \to a$$

We can also give lambda terms for the data constructors

$$\mathit{Zero} :: \mathit{Nat}$$
$$\mathit{Zero} = \Lambda a \to \lambda z \to \lambda s \to z$$
$$\mathit{Succ} :: \mathit{Nat} \to \mathit{Nat}$$
$$\mathit{Succ}\ n = \Lambda a \to \lambda z \to \lambda s \to s\ (n\ a\ z\ s)$$

The notion and notation of free structures is very useful in programming. The form of data definitions in languages like ML and Haskell strongly resembles free structures. A downside of the ML, Haskell and free structure presentation of data is that it is hard to write generic functions that can work for any choice of data structure. For example, the type of the natural fold function the is rather different for each free structure.

$$\mathit{fold}_{\mathit{Maybe}} :: \forall a.a \to (b \to a) \to \mathit{Maybe}\ b \to a$$
$$\mathit{fold}_{\mathit{Nat}} \quad :: \forall a.a \to (a \to a) \to \mathit{Nat} \qquad \to a$$

To describe it generally we have to use an unsatisfying imprecise "..." notation

$$\mathit{fold}_D :: \forall a.\ S_0\ [D := a] \to S_1\ [D := a] \to \ldots \to S_n\ [D := a] \to D \to a$$

and remember that there is another "..." within each S_i as the number of arguments for each data constructor is different for different free structures.

2.2.3 Describing data via functors

An alternative presentation of free structures is using functors[5].

A functor describes a single 'level' of a recursive data structure. The type parameter is used in the position(s) where there are recursive occurrences of the same data type. For example, one level of the naturals is described by the functor

$$NatF\ a = \mathbf{1} + a$$

So although each functor is different, there are many definitions we can write that work for any suitable functor.

The notation $\mu a.F\ a$ is used for the type of the encoding of the overall recursive data structure. This is a variable binding construct, like $\forall a.T\ a$ or $\exists a.T\ a$. It is defined as

$$\mu a.F\ a = \forall a.(F\ a \to a) \to a$$

It is not immediately obvious that this gives us the same encoding as that for a free structure. It requires expanding out the definition of the functor and applying some type isomorphisms. For example, for the functor $NatF$ we define the overall type Nat as

$$Nat = \mu a.NatF\ a$$

We can expand this out to get the standard type for the Church encoding of natural numbers

$$\begin{array}{ll} & \mu a.NatF\ a \\ = & \quad \{\ \text{definition of } \mu a.F\ a\ \} \\ & \forall a.(NatF\ a \to a) \to a \\ = & \quad \{\ \text{definition of } NatF\ \} \\ & \forall a.((\mathbf{1} + a) \to a) \to a \\ \equiv & \quad \{\ \text{distribute, } (A + B) \to C \equiv (A \to C, B \to C)\ \} \\ & \forall a.(\mathbf{1} \to a, a \to a) \to a \\ \equiv & \quad \{\ \text{nullary function, } \mathbf{1} \to A \equiv A\ \} \\ & \forall a.(a, a \to a) \to a \\ \equiv & \quad \{\ \text{curry, } (A, B) \to C \equiv A \to (B \to C)\ \} \\ & \forall a.a \to (a \to a) \to a \end{array}$$

[5] More specifically, endo-functors.

This presentation using a functor is often preferable to the free structure presentation because we can describe generic functions without having to commit to a specific functor, i.e. a specific data structure. It is also more concise and does not need to resort to an imprecise "..." notation. In particular, this presentation lets us define a general fold function

Definition 2.2.1 (*fold* for the Church encoding)**.**

$$fold :: \forall a.\ (F\ a \to a) \to \mu b.F\ b \to a$$
$$fold = \Lambda a \to \lambda(k :: F\ a \to a) \to \lambda(x :: \mu b.F\ b) \to x\ a\ k$$

Or in an equational style and omitting the explicit typing

$$fold :: \forall a.\ (F\ a \to a) \to \mu b.F\ b \to a$$
$$fold\ a\ k\ x = x\ a\ k$$

In addition we can define the *build* function from the first chapter but now generalised to any data type that is described by a functor

Definition 2.2.2 (*build* for the Church encoding)**.**

$$build :: (\forall a.(F\ a \to a) \to a) \to \mu b.F\ b$$
$$build = \lambda(g :: \forall a.(F\ a \to a) \to a) \to \Lambda b \to \lambda(k :: F\ b \to b) \to g\ b\ k$$

and in the equational style without explicit typing

$$build :: (\forall a.(F\ a \to a) \to a) \to \mu b.F\ b$$
$$build\ g\ b\ k = g\ b\ k$$

2.2.4 The co-Church encoding

The Church encoding looks at recursive data in terms of how we take it apart to produce some other value. The co-Church encoding takes the dual view, that we look at *co-data* in terms of how we can construct it.

The co-Church encoding is also based on a functor F that describes one layer of a data type we would like to define. The notation $\nu c.F\ c$ is used for the type of the encoding of the overall recursive data structure. It is defined as

$$\nu a.F\ a = \exists a.(a \to F\ a, a)$$

The intuition is that a data structure represented in this way consists of a 'seed' value and a 'next' function. By applying the next function to the seed value we

get a single layer unfolding of the data structure. Occurrences of the functor parameter in this single layer unfolding correspond to new seed values. We can then apply the same next function to these new seed values to get the next stage of the unfolding. By repeating this process we can unfold the data structure indefinitely. Note that due to the existential quantification the 'actual' type of the seed values is hidden, the only information available is given by the structure of the unfolding.

While the natural operation on data in the Church encoding is a *fold*, the natural operation on data in the co-Church encoding is an *unfold*

Definition 2.2.3 (*unfold* for the co-Church encoding)**.**

$$\begin{aligned}&\mathit{unfold} :: \forall a.\ (a \to F\ a) \to a \to \nu b.F\ b\\ &\mathit{unfold} = \Lambda a \to \lambda(k :: a \to F\ a) \to \lambda(s :: a) \to (a, (k, s))\end{aligned}$$

and again with less explicit typing

$$\begin{aligned}&\mathit{unfold} :: \forall a.\ (a \to F\ a) \to a \to \nu b.F\ b\\ &\mathit{unfold}\ a\ k\ s = (a, (k, s))\end{aligned}$$

In addition we can define the *unbuild* function for any functor

Definition 2.2.4 (*unfold* for the co-Church encoding)**.**

$$\begin{aligned}&\mathit{unbuild} :: \forall c.\ (\forall a.(a \to F\ a) \to a \to c) \to (\nu b.F\ b \to c)\\ &\mathit{unbuild} = \Lambda c \to \lambda(g :: \forall a.(a \to F\ a) \to a \to c) \to\\ &\qquad\qquad \lambda(b, (k, s) :: (b \to F\ b, b)) \to g\ b\ k\ s\end{aligned}$$

and again with less explicit typing

$$\begin{aligned}&\mathit{unbuild} :: \forall c.\ (\forall a.(a \to F\ a) \to a \to c) \to (\nu b.F\ b \to c)\\ &\mathit{unbuild}\ c\ g\ (b, (k, s)) = g\ b\ k\ s\end{aligned}$$

We can use the same functor to give us both the Church and the co-Church encoding. For example we can use the same *NatF* functors as before to define the *CoNat* type

$$\begin{aligned}\mathit{CoNat} &= \nu a.\mathit{NatF}\ a\\ &= \exists a.(a \to \mathit{NatF}\ a, a)\\ &= \exists a.(a \to \mathbf{1} + a, a)\end{aligned}$$

There is a reasonable claim that the *CoNat* type can represent natural numbers. A value of this type is of the form $(A, (k, s))$ for some (unknown) type A. When

we apply k to s, if the result is in the left side of the sum $\mathbf{1} + a$ then we have an encoding of the natural number zero. If the result is in the right side of the sum then we have an encoding of a number that is the successor to the number encoded as $(A, (k, s'))$, where s' is the new seed.

While at first sight it seems that we can represent natural numbers by either the Church or co-Church encodings, the two are not the same. There is a value of type *CoNat* that does not represent any natural number, in particular the value

$$inf = (1, (\lambda\_ \to right\ 1, 1))$$

The next function here always returns in the right side of the sum, hence no matter how often we chase successors we never get to zero. This value is equivalent to the limit of the naturals, the simple countable infinity.

There is relationship between the types *Nat* and *CoNat*. We can write an injective function mapping *Nat* and *CoNat*, so all values of *Nat* have a corresponding value of type *CoNat*. On the other hand we know that *CoNat* has at least one more value in it than *Nat* hence we cannot write a injective function in the other direction.

2.2.5 Abstract semantics

For binary products and sums the abstract semantics are relatively easy to state and it is not too tricky to prove that the encodings for binary products and sums in Section 2.2.1 do indeed implement the abstract semantics. So far, for data $\mu a.F\ a$ and co-data $\nu a.F\ a$ types we have given the encoding but not yet specified the abstract semantics.

It turns out that the appropriate semantics for data and co-data defined by a functor F uses initial F-algebras and final F-co-algebras respectively. F-algebras and F-co-algebras are yet more concepts from category theory. We will cover this in greater detail in the next chapter since it is vital to the proofs for shortcut fusion.

This semantics of data and co-data using initial algebras and final co-algebras is exactly the semantics that Hagino (1987) requires of a categorical programming language. Wraith (1989) gives the System F encodings for $\mu a.F\ a$ and $\nu a.F\ a$ and shows that they are weakly initial F-algebras and weakly final F-co-algebras. Hasegawa (1991) proves the same encodings are strongly initial and final if the semantic model is parametric.

This combination of results serves to emphasise the point that these two forms of recursive data structure are canonical. It also goes some way to explaining

why these two views of data are natural choices for the basis of shortcut fusion systems – in contrast with the many choices of ad hoc shortcut fusion system (as discussed in Section 1.3.4).

2.3 CPOs

If functional programmers are Platonists, then when writing a function the platonic object the programmer has in mind is probably a set-theoretic function. The meaning of programs in simply-typed lambda calculus can be described using ordinary set theory. However most interesting extensions of lambda calculus cannot be fully described by set-theoretic functions[6]. Domain theory was invented to explain the semantics of functional programming languages that use partial functions and general recursion.

In domain theory, complete partial orders (CPOs) describe the range of values that can be computed. These values are related to each other by a partial ordering ($\sqsubseteq$). The ordering describes how 'defined' a value is, or to put it another way, how much information we know about a value. CPOs have a bottom element, written $\bot$, that is less defined than all other values of the same type. This 'undefined' value is used to describe values about which we know nothing, such as the 'result' of non-terminating evaluation.

One can use CPOs as a semantic model for System F, however one typically does not, at least not for unadulterated System F, because the range of values in the semantic model is greater than the ones that can be described by terms in the language. In particular since System F is strongly normalising there are no terms that correspond to the $\bot$ element in the CPO semantic model.

The main purpose in using CPOs as a semantic model is to change the System F syntax by adding a fixpoint combinator as an additional constant

$$\mathit{fix} :: \forall a.\ (a \to a) \to a$$

with the reduction rule

$$\mathit{fix}\, f = f\ (\mathit{fix}\, f)$$

Of course this means the strong normalisation property is lost as the fixpoint combinator makes it possible to write terms with no normal form such as $\mathit{fix}\ \mathit{id}$.

[6] At least not ordinary ZF set theory. There are some alternative approaches based on intuitionistic set theory. Rosolini and Simpson (2004) give a recent and reasonably accessible presentation of this approach.

When using the CPO model we will write recursive definitions of the form

$$f = \ldots f \ldots$$

which are to be interpreted as uses of the *fix* combinator

$$f = \mathit{fix}\ (\lambda f' \to \ldots f' \ldots)$$

The basic proof method for programs that use the *fix* combinator is fixpoint induction.

Property 2.3.1 (Fixpoint induction proof scheme)**.** For a continuous function h and a chain-complete property P

$$\begin{array}{l} P\ \bot \quad \wedge \quad \forall g.\ P\ g \Rightarrow P\ (h\ g) \\ \Longrightarrow \\ P\ (\mathit{fix}\ h) \end{array}$$

All functions definable in the language are continuous. Similarly, checking that a predicate is chain complete is usually easy, as all equations between functions definable in the language are chain complete.

Applying this proof scheme requires first that we arrange for the property we wish to prove to be an instance of the conclusion of the proof scheme $P\ (\mathit{fix}\ h)$. In particular this means identifying a function within the property that is defined in terms of *fix*. Having set up the property P, the next step is to prove $P\ \bot$ and $P\ g \Rightarrow P\ (h\ g)$. The first is usually straightforward and the latter we usually approach by starting with $P\ (h\ g)$ then unfolding definitions, splitting into cases as necessary and at some point using the induction hypothesis $P\ g$.

The existence of a $\bot$ element weakens many properties that hold in System F. While the encodings of sums and products still work, they are no longer categorical sums or products. This is because it is no longer true that all values in the sum/product type correspond to actual sum/product terms, in particular $\bot$ does not.

There are also some slightly unexpected consequences of adding $\bot$ and a fixpoint combinator. For example, not only does the *Nat_bottom* type have a $\bot$ value, it also has all the partial naturals and an upper limit. The partial naturals are the following terms, related by the partial ordering

$$\bot \sqsubseteq \mathit{Succ}\ \bot \sqsubseteq \mathit{Succ}\ (\mathit{Succ}\ \bot) \sqsubseteq \ldots$$

Since these form an ascending chain then the CPO properties mean that we must also have a value that is the upper limit of this chain.

Indeed using *fix* we can define a term that is this limit

$$infinity :: Nat_\perp$$
$$infinity = fix\ (\lambda inf\ z\ s \to s\ (inf\ z\ s))$$

There are similar partial naturals in the $CoNat_\perp$ encoding and of course it already had an upper limit. The previous natural injections from *Nat* to *CoNat* still work with the new partial naturals. Furthermore, using *fix* it is now possible to write an injective function from $CoNat_\perp$ to $Nat_\perp$ [7].

As mentioned in the previous section, a parametric semantics of System F has initial F-algebras and final F-co-algebras. Domains also have initial F-algebras and final F-co-algebras. For initial F-algebras however this is restricted to domains with strict functions. Final F-co-algebras are not similarly restricted; they exist in domains with strict and non-strict functions (Abramsky and Jung, 1995, Section 5.3.2). As is suggested by our ability to convert $CoNat_\perp$ to $Nat_\perp$, the initial F-algebras and final F-co-algebras are isomorphic, though only for domains with strict functions.

The fact that the Church and co-Church encodings of data are isomorphic in CPOs sometimes leads people to believe that the distinction is unimportant. The fact that they are isomorphic says very little about performance, which is after all the goal of fusion. Standard functional languages such as ML and Haskell provide data that implements the semantics of both encodings in a single representation. Thus there is no conversion cost to constructing a data structure using an unfold and then consuming it using a fold. However this does not mean that we can magically fuse a fold with an unfold; the Church or co-Church view of the data is still important.

2.4 Haskell

At the time of writing, Haskell (Peyton Jones et al., 2003) is the standard pure non-strict functional programming language. It is a practical programming language and while its semantics are not completely precisely specified, they are based on CPOs.

Haskell is statically typed. The type system of Haskell 98 is based on the Hindley-Milner type system, with extensions for type classes and polymorphic

[7] by mapping the $CoNat_\perp$ infinity $(1, (\lambda\_ \to right\ 1, 1))$ to the $Nat_\perp$ infinity $fix\ (\lambda inf\ z\ s \to s\ (inf\ z\ s))$

recursion. Terms and types in Haskell can be translated[8] into System F extended with a fixpoint combinator and using a CPO semantics.

In Haskell 98, polymorphic functions are given types where the free variables are implicitly universally quantified over. For example the identity function is written as

$$id :: a \to a$$
$$id\ x = x$$

A common extension which we will make use of is to make the universal quantifier explicit

$$id :: \forall a.\ a \to a$$

Haskell makes almost full use of the fact that its semantics is based on CPOs by providing the function *seq* which has the property:

$$seq\ \bot\ x = \bot$$
$$seq\ y\ \ x = x \qquad \{ \text{ where } y \text{ is not } \bot \}$$

Although this function cannot be defined using the term language, its meaning is perfectly reasonable within the CPO semantics. In particular *seq* makes clear that the function type in Haskell is lifted.

As mentioned, Haskell is a relatively large and complex language and does not have a completely formally specified semantics. In the theory chapter where we are using Haskell we will try wherever possible to stick to the simple subset that is easily translated into the extended System F. In particular we will make use of algebraic data types, their data constructors and **case** deconstructors.

Where it helps the presentation however we will make use of some of Haskell's syntactic sugar. In particular we will make use of patterns and guards in the left hand side of function definitions; these desugar into case expressions. We will also use **where** clauses for local definitions which desugar into **let** expressions.

We will make use of an language extension that lets us use existential types. Existential types are not supported as first class, instead only data constructors can contain existentially typed values. We use the following form of syntax[9]

$$\textbf{data}\ P = \exists a.MkP\ a\ (a \to Bool)$$

[8] More specifically, System F_ω is required since Haskell has higher kinds.

[9] The common language extension actually uses the syntax 'forall' rather than 'exists'. The 'forall' syntax emphasises the type of the data constructor itself rather than the type of the contained data.

Here we have a type P with a single data constructor MkP. This data type is equivalent to $\exists a.(a, a \rightarrow Bool)$. The type of the MkP constructor function is universally quantified but the type variable does not appear in the result type P

$$MkP :: \forall a.a \rightarrow (a \rightarrow Bool) \rightarrow P$$

It should be noted that in future it may become standard practice in Haskell to write existential types using GADT syntax. The above example may be written using GADT syntax as

$$\textbf{data } P \textbf{ where}$$
$$MkP :: \forall a.a \rightarrow (a \rightarrow Bool) \rightarrow P$$

When describing syntactic transformations it is occasionally helpful to explicitly distinguish variables in the concrete syntax from metavariables that stand for whole terms, particularly where those terms may contain free variables. In Chapter 4 we will use the syntax $\langle x \rangle$ to denote a metavariable that stands for some term.

Chapter 3

Stream fusion is correct

In this chapter we will look at the theoretical basis for the correctness of shortcut fusion in general and stream fusion in particular.

3.1 Introduction and strategy

We want to prove that stream fusion is valid. That means we must justify transformations of the form

$$\begin{array}{ll} & \mathit{unstream} \circ f_s \circ \mathit{stream} \circ \mathit{unstream} \circ g_s \circ \mathit{stream} \\ = & \quad \{ \text{“stream fusion”} \} \\ & \mathit{unstream} \circ f_s \circ g_s \circ \mathit{stream} \end{array}$$

We must be precise both about the kinds of streams we are working with, skipping or non-skipping, and the language and semantic domain we are working in. There are three cases that we are interested in:

- non-skipping streams in System F, with a parametric semantic model;
- non-skipping streams in Haskell, using CPOs;
- skipping streams in Haskell, using CPOs.

These cases are in order of increasing realism but the results we can hope to obtain are decreasingly powerful. We can expect to obtain a much stronger result with non-skipping streams in System F than we can with skipping streams in Haskell. On the other hand, ultimately we wish to apply this transformation in practice in the context of Haskell. It is worth asking which cases are really worth considering separately and which as part of the next, more general but weaker, case.

It is instructive to start with non-skipping streams in the simpler setting of System F. The semantic domain for System F is more favourable for our purposes than Haskell's CPO-based semantics. Indeed for System F the result we want is already present in the literature. In System F we will take an approach based on parametricity. The initial approach will be to prove that $stream \circ unstream = id$ as this gives a straightforward justification of the stream fusion transformation.

When we move on to CPOs we will demonstrate a weaker, yet sufficient, result that does not rely on parametricity. In our final setting we allow streams to skip and this has the consequence that $stream \circ unstream = id$ is no longer true. We can still justify the stream fusion transformation however. To do so we have to use a slightly weaker property and add side conditions on f_s and g_s.

Our strategy for the two Haskell cases is to tackle the more general case of skipping streams since any proof technique for the skipping should be easily adaptable to cover the non-skipping case. We will not attempt to find a stronger result in the non-skipping case. That is, we will not attempt to prove $stream \circ unstream = id$ for non-skipping streams in the CPO setting. While it may be possible to obtain such a result using techniques demonstrated by Johann (2003) (using an operational model of Haskell without polymorphic *seq*), it would be of little practical use and probably would not provide any more theoretical insights than the stronger result in System F.

We will show that a sufficient condition for the fusion transformation, even in the skipping case, is that each stream function f_s and its ordinary list counterpart f satisfy

$$unstream \circ f_s = f \circ unstream$$

We must in any case prove a property like this for each stream function in our library. We need this simply to be sure that the stream version really does correspond to the list version. Strictly speaking, merely to show the correspondence between the stream and list versions it would be sufficient to show $f = unstream \circ f_s \circ stream$; however, the formulation above is slightly stronger and it is sufficient to justify the stream fusion transformation.

While in practice we have only implemented the stream fusion system for sequences, in principle it should work for any co-data[1]. For most of the proofs in this chapter we will use the greater generality obtained by working with data and co-data in terms of an arbitrary functor F. The primary motivation for generalising to an arbitrary functor is to simplify the proofs. The results for lists can be obtained by substituting in the list functor.

[1]It is not clear however that there would be any performance benefits for non-sequence types.

The remainder of this chapter is organised as follows:

- We will start in Section 3.2 by considering shortcut fusion in the setting of System F. We will see that when using the Church and co-Church encodings directly, correctness proofs of fusion are almost embarrassingly trivial.
- In Sections 3.3 and 3.4 we use the abstract definition of data and co-data rather than specific encodings. In this context can prove the two shortcut fusion rules by taking advantage of parametricity.
- In Section 3.5 we build on the proof of the *unfold*/*unbuild* fusion rule in System F and find that the stream fusion rule is a simple extension – at least for streams without skip.
- In Section 3.6 we look at the more complicated case of streams with skip but find that System F is not the appropriate model for describing skipping streams.
- In Section 3.7 we move from System F to a semantics based on CPOs and look at streams in this new context.
- In Section 3.8 we look at the fusion rule for skipping streams in the CPO context. While we can still prove the transformation is correct we find that we need side conditions on the library functions involved.
- In Section 3.9 we look at how to prove that library functions satisfy the side conditions identified in the previous section. We go through the details of fixpoint induction proofs for a number of standard functions and give some heuristics for a general method.
- While the earlier sections look at lists or more generally data and co-data, in Section 3.10 we look at stream fusion for abstract data types. In particular we look at what properties an ADT must have for us to be able to use stream fusion correctly. We sketch proofs for the specific example of arrays.
- Finally in Section 3.11 we look at a testing technique suitable for stream fusion implementations. This can give us a limited degree of confidence in our implementations with a substantially lower effort compared to the formal proofs of Section 3.9.

3.2 Fusion with the Church and co-Church encodings

We have seen (in Sections 2.2.3 and 2.2.4) that with the Church and co-Church encodings of data and co-data that we can write very simple definitions of the functions *fold*, *unfold*, *build* and *unbuild*. It should not surprise us therefore that in this setting the fusion rules *fold*/*build* and *unbuild*/*unfold* are similarly trivial. Let us see just how trivial.

Recall (from Section 2.2.3) the definition of the Church encoding of $\mu a.F\ a$ and the general *fold* and *build* for a data type functor F

$$\mu a.F\ a = \forall a.(F\ a \to a) \to a$$
$$fold :: \forall a.\ (F\ a \to a) \to \mu b.F\ b \to a$$
$$fold\ a\ k\ x = x\ a\ k$$
$$build :: (\forall a.(F\ a \to a) \to a) \to \mu b.F\ b$$
$$build\ g\ b\ k = g\ b\ k$$

Theorem 3.2.1 (*fold*/*build* fusion rule with Church encoding)**.**

$$fold\ a\ k\ (build\ g) = g\ a\ k$$

Proof. It follows directly from the definitions

$$\begin{aligned} & fold\ a\ k\ (build\ g) \\ = & \quad \{ \text{ by definition of } fold \ \} \\ & (build\ g)\ a\ k \\ = & \quad \{ \text{ by definition of } build \ \} \\ & g\ a\ k \end{aligned}$$

□

Similarly for $\nu a.F\ a$, *unbuild* and *unfold* (from Section 2.2.4)

$$\nu a.F\ a = \exists a.(a \to F\ a, a)$$
$$unfold :: \forall a.\ (a \to F\ a, a) \to \nu b.F\ b$$
$$unfold\ a\ k\ s = (a, (k, s))$$
$$unbuild :: \forall c.\ (\forall a.(a \to F\ a, a) \to c) \to (\nu b.F\ b \to c)$$
$$unbuild\ c\ g\ (b, (k, s)) = g\ b\ k\ s$$

Theorem 3.2.2 (*unbuild*/*unfold* fusion rule with Church encoding)**.**

$$unbuild\ c\ g\ (unfold\ a\ k\ s) = g\ a\ k\ s$$

Proof. Again, it follows directly from the definitions

$$\begin{aligned} & unbuild\ c\ g\ (unfold\ a\ k\ s) \\ = & \quad \{ \text{ by definition of } unfold \ \} \\ & unbuild\ c\ g\ (a, (k, s)) \\ = & \quad \{ \text{ by definition of } unbuild \ \} \\ & g\ a\ k\ s \end{aligned}$$

□

3.3 Fusion with initial data

We see that with the Church and co-Church encodings the standard shortcut fusion rules are trivially true. A more interesting result is to show that the same equations hold not just for some specific encoding but for any correct implementation of data or co-data, purely from the required semantics of data and co-data. Such a result is of practical significance because in a programming language based on System F we do not use the Church encoding of data and co-data but instead use an equivalent, more efficient machine representation. We would like any fusion results to hold for the efficient representation too.

To be able to decide if an implementation of data or co-data is correct requires that we have an abstract semantics of data and co-data. As mentioned in Section 2.2.5, the appropriate semantics for data and co-data is initial algebras and final co-algebras respectively. The details are given by Wadler (1990a) but we will give a short presentation here.

3.3.1 Initial data

For both data and co-data we start with a functor F. A standard example is the functor for lists (with element type e):

$$ListF\ e\ a = \mathbf{1} + (e, a)$$

For data, we call the abstract type $\mu a.F\ a$ though we will sometimes abbreviate it to simply T. The abstract type has functions

$$in :: F\ T \to T$$
$$out :: T \to F\ T$$

These essentially wrap and unwrap a single layer of the data structure. We have not stated all the required properties yet, but it is worth noting that a consequence of the properties is that *in* and *out* are mutual inverses and that therefore $F\ T$ and T are isomorphic. This is why we say that T is a fixpoint of F.

The other required operation is *fold*:

$$fold :: \forall a.\ (F\ a \to a) \to T \to a$$

The property of being an initial algebra is stated in terms of concepts from category theory. In our context of types and functions, an algebra for the functor F is a pair (A, k) consisting of a type A and a function $k :: F\ A \to A$. An initial algebra is an algebra that is initial in the category of F-algebras, meaning there is a unique mapping from the initial algebra to all the other F-algebras. A mapping in the category of F-algebras, say from (A, k) to (A', k') is a function $h :: A \to A'$ that satisfies the property

$$h \circ k = k' \circ F\ h$$

In the case of data T for a functor F we require that (T, in) is the initial algebra. This means there must be a mapping h from (T, in) to any other F-algebra (A, k) and that the mapping must be unique. The mapping in question is *fold A k*. The statement of this fact is called the universal property of *fold*.

Property 3.3.1 (Universal property of *fold*)**.**

$$h \circ in = k \circ F\ h$$
$$\iff$$
$$h = fold\ A\ k$$

Now that we are armed with the abstract semantics of data there are two things to do. We must check that the Church encoding satisfies the property. Then, using just the abstract semantics, we wish to prove that the *fold*/*build* property holds. First however we take a brief explanatory diversion.

3.3.2 Universal property of *fold* for lists

Stated in the above form, the universal property is rather abstract. It is instructive to derive the familiar universal property for *fold* on lists by specialising the above property for the list functor. Some readers may choose to skip this diversion.

Since the purpose is to derive the familiar we will use Haskell's data type syntax instead of using the Church encoding. We will represent the list functor $ListF\ e\ a = \mathbf{1} + (e, a)$ in Haskell types as

$$\textbf{type}\ ListF\ e\ a = Maybe\ (e, a)$$

Thus for F and T we substitute $ListF\ e$ (for some e) and $[e]$ respectively.

The function $in :: F\ T \to T$ will be

$$\begin{aligned} &in :: ListF\ e\ [e] \to [e] \\ &in\ Nothing \qquad\ = [\,] \\ &in\ (Just\ (x, xs)) = x : xs \end{aligned}$$

Next we specialise $h \circ in = k \circ F\ h$. Since we are using Haskell, we write $F\ h$ as $fmap\ h$. Note that the argument type of each side is $ListF\ e\ [e]$ so without loss of generality we can apply both sides to values $Nothing$ and $Just\ (x, xs)$:

$$\begin{aligned} &h \circ in = k \circ fmap\ h \\ \iff\ &\\ &h\ (in\ Nothing) \qquad\ = k\ (fmap\ h\ Nothing) \\ &h\ (in\ (Just\ (x, xs))) = k\ (fmap\ h\ (Just\ (x, xs))) \end{aligned}$$

We can simplify both cases since we have a definition for in and the definition of $fmap$ on $ListF\ a$ is straightforward. For the two cases on the left hand side we have

$$\begin{aligned} &h\ (in\ Nothing) \\ =\ &\quad \{\ \text{since}\ in\ Nothing = [\,]\ \} \\ &h\ [\,] \end{aligned}$$

and

$$\begin{aligned} &h\ (in\ (Just\ (x, xs))) \\ =\ &\quad \{\ \text{since}\ in\ (Just\ (x, xs)) = x : xs\ \} \\ &h\ (x : xs) \end{aligned}$$

And similarly on the right hand side

$$\begin{array}{ll} & k\ (fmap\ h\ Nothing) \\ = & \quad \{\ \text{since}\ fmap\ h\ Nothing = Nothing\ \} \\ & k\ Nothing \end{array}$$

and

$$\begin{array}{ll} & k\ (fmap\ h\ (Just\ (x, xs))) \\ = & \quad \{\ \text{since}\ fmap\ h\ (Just\ (x, xs)) = Just\ (x, h\ xs)\ \} \\ & k\ (Just\ (x, h\ xs)) \end{array}$$

If we put these together we get

$$\begin{array}{ll} h\ [\,] & = k\ Nothing \\ h\ (x : xs) & = k\ (Just\ (x, h\ xs)) \end{array}$$

We can obtain a more standard presentation by defining:

$$\begin{array}{ll} z & = k\ Nothing \\ f\ x\ xs & = k\ (Just\ (x, xs)) \end{array}$$

and thus we get

$$\begin{array}{ll} h\ [\,] & = z \\ h\ (x : xs) & = f\ x\ (h\ xs) \end{array}$$

This gives us the standard form of the universal property on lists, that if h satisfies the above equations then it is equal to *fold k*, or equivalently, to *foldr f z*.

3.3.3 Church encoding is initial in parametric models

We should check that the Church encoding of data satisfies the abstract semantics of data, i.e. the universal property of *fold*

$$\begin{array}{l} h \circ in = k \circ F\ h \\ \Longleftrightarrow \\ h = fold\ A\ k \end{array}$$

Again we follow the presentation of Wadler (1990a). There are two parts to prove. The easier part is to check that $fold\ A\ k$ is a function h that satisfies $h \circ in = k \circ F\ h$. The second part is to show that $fold\ A\ k$ is the only such function.

Recall that the Church encoding represents data defined by a functor F using terms of the type $\forall a.(F\ a \to a) \to a$. Thus in this context we use

$$T\ =\ \mu a.F\ a\ =\ \forall a.(F\ a \to a) \to a$$

Lemma 3.3.2. $h = \mathit{fold}\ A\ k \implies h \circ \mathit{in} = k \circ F\ h$

Proof. Performing the substitution for h and applying both sides to an arbitrary y gives us the equivalent goal of showing

$$\mathit{fold}\ A\ k\ (\mathit{in}\ y) = k\ (F\ (\mathit{fold}\ A\ k)\ y)$$

We will define suitable lambda terms for *in* and *fold* and then use the ordinary reduction rules to show the terms on each side are equal.

For the Church encoding we define *fold* and *in* as

$$\begin{aligned}
&\mathit{fold} :: \forall a.\ (F\ a \to a) \to T \to a\\
&\mathit{fold}\ a\ k\ x = x\ a\ k\\
&\mathit{in} :: F\ T \to T\\
&\mathit{in}\ y\ b\ k = k\ (F\ (\mathit{fold}\ b\ k)\ y)
\end{aligned}$$

We now simplify the left hand side until it matches the right

$$\begin{aligned}
&\quad \mathit{fold}\ A\ k\ (\mathit{in}\ y)\\
&= \quad \{\ \text{unfold definition of } \mathit{in}\ \}\\
&\quad \mathit{fold}\ A\ k\ (\lambda a\ k \to k\ (F\ (\mathit{fold}\ a\ k)\ y))\\
&= \quad \{\ \text{unfold definition of } \mathit{fold}\ \}\\
&\quad (\lambda a\ k \to k\ (F\ (\mathit{fold}\ a\ k)\ y))\ A\ k\\
&= \quad \{\ \beta\text{-reduce}\ \}\\
&\quad k\ (F\ (\mathit{fold}\ A\ k)\ y)
\end{aligned}$$

□

To show that $\mathit{fold}\ \ A\ \ k$ is the unique function h satisfying $h \circ \mathit{in} = k \circ F\ \ h$, we cannot use a simple syntactic proof. The proof relies on properties of the semantic model that we are using. In particular it is sufficient that the semantic model has the parametricity property.

Lemma 3.3.3. $h \circ \mathit{in} = k \circ F\ h \implies h = \mathit{fold}\ A\ k$

Proof. We start with the free theorem for the type of *fold*

$$h \circ k' = k \circ F\ h \implies h \circ \mathit{fold}\ A'\ k' = \mathit{fold}\ A\ k$$

We are free here to pick h, k and k' as we choose.

Substituting $k' := in, A' := T$ gives us

$$h \circ in = k \circ F\ h \implies h \circ fold\ T\ in = fold\ A\ k$$

If we can show that $fold\ T\ in = id\ T$ then we are left with

$$h \circ in = k \circ F\ h \implies h = fold\ A\ k$$

which would complete the proof. We can prove $fold\ T\ in = id\ T$ as follows. We again take the free theorem for the type of *fold* and substitute $h := fold\ A\ k$ and $k' = in, A' := T$

$$fold\ A\ k \circ in = k \circ F\ (fold\ A\ k) \implies fold\ A\ k \circ fold\ T\ in = fold\ A\ k$$

We can discharge the antecedent because it is a statement of the previous lemma, leaving us with just

$$fold\ A\ k \circ fold\ T\ in = fold\ A\ k$$

We now calculate

$$\begin{array}{ll} & fold\ A\ k \circ fold\ T\ in = fold\ A\ k \\ \implies & \{\ \eta\text{-expand}\ \} \\ & fold\ A\ k\ (fold\ T\ in\ x) = fold\ A\ k\ x \\ \implies & \{\ \text{by definition}\ fold\ a\ k\ x = x\ a\ k\ \} \\ & fold\ T\ in\ x\ A\ k = x\ A\ k \\ \implies & \\ & fold\ T\ in = id\ T \end{array}$$

□

3.3.4 *fold/build* holds for initial data in parametric models

So we have seen that the *fold/build* property holds for the Church encoding and that the Church encoding does give us initial algebras in parametric models. We want to be sure that the *fold/build* property holds for any implementation of data, which means proving the property from the abstract semantics of data, without assuming any specific encoding.

Recall the *fold/build* fusion property:

$$fold\ a\ k\ (build\ g) = g\ a\ k$$

Let us state our assumptions precisely. We are working from the abstract semantics of initial data $\mu a.F\ a$ for a covariant functor F. Again we use the alias $T = \mu a.F\ a$. It being initial data gives us the abstract terms

$$\begin{aligned} &fold :: \forall a.\ (F\ a \to a) \to T \to a \\ &in \quad :: F\ T \to T \end{aligned}$$

and the universal property of *fold*

$$\begin{aligned} &h \circ in = k \circ F\ h \\ \iff \\ &h = fold\ A\ k \end{aligned}$$

We also have the assumption that we are working in a parametric model.

The properties of initial data give us *fold* directly but we must define *build* in terms of the functions we have to hand, namely *in*, *out* and *fold*. We use the following definition

$$\begin{aligned} &build :: (\forall a.(F\ a \to a) \to a) \to T \\ &build\ g = g\ T\ in \end{aligned}$$

One may reasonably wonder how we conjure this definition of *build* and whether it is a sensible definition. The technique is to 'guess then check'. We guess based on the structure of the type. We will then check that the *fold*/*build* property using this definition. If the property holds then we know that this is a suitable definition of *build*.

Theorem 3.3.4 (*fold*/*build* fusion for initial data)**.**

$$fold\ a\ k\ (build\ g) = g\ a\ k$$

Proof. We naturally start off by unfolding definitions

$$\begin{aligned} &fold\ a\ k\ (build\ g) \\ = \quad &\{\ \text{definition of } build\ \} \\ &fold\ a\ k\ (g\ T\ in) \end{aligned}$$

The difficulty at this step is that we do not know much about g. In fact the only thing we know about g is its type. This is where we must rely on the assumption that we are using a semantic model with the parametricity property. We can apply parametricity to give us a property about g purely from its type; the free theorem for g (see Section 2.1.3 for an introduction to free theorems).

We will state the free theorem for g and derive it later as a separate lemma.

$$
\begin{aligned}
&g :: \forall a.(F\ a \to a) \to a \\
&h :: A' \quad \to A \\
&k :: F\ A \ \to A \\
&k' :: F\ A' \to A' \\
&h \circ k' = k \circ F\ h \implies h\ (g\ A'\ k') = g\ A\ k
\end{aligned}
$$

So this tells us what we know about g, given only its type. Note that for convenience in the next step we have alpha-renamed the free variables A / A' and k / k' compared to the natural naming order.

We return to where we had got stuck with unfolding definitions. It remains to show that

$$fold\ a\ k\ (g\ T\ in) = g\ a\ k$$

Fortunately, if we do a little pattern matching we can see that, with the right substitution, this matches the consequent of the free theorem for g, namely $h\ (g\ A'\ k') = g\ A\ k$. The appropriate substitutions are

$$
\begin{aligned}
h &:= fold\ a\ k \\
A &:= a \\
k' &:= in \\
A' &:= T
\end{aligned}
$$

Let us perform the substitution

$$
\begin{aligned}
&(h \circ k' = k \circ F\ h \implies h\ (g\ A'\ k') = g\ A\ k)\ [h := fold\ a\ k, A := a, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad k' := in, A' := T] \\
\equiv\ &\\
&fold\ a\ k \circ in = k \circ F\ (fold\ a\ k) \implies fold\ a\ k\ (g\ T\ in) = g\ a\ k
\end{aligned}
$$

Since the consequent matches our goal, we now just need to show that

$$fold\ a\ k \circ in = k \circ F\ (fold\ a\ k)$$

which is of course a simple consequence of the universal property of *fold*. □

Note that if we are using the Church encoding specifically then we have a simple corollary

$$
\begin{aligned}
&fold\ a\ k\ (g\ T\ in) = g\ a\ k \\
\iff \quad &\{\ \text{by definition of } fold\ a\ k\ x = x\ a\ k\ \} \\
&(g\ T\ in)\ a\ k = g\ a\ k
\end{aligned}
$$

This justifies our original definition of $build\ g = g$ for the special case of the Church encoding rather than the general $build\ g = g\ in$.

Lemma 3.3.5. Free theorem for $g :: \forall a.(F\ a \to a) \to a$

Proof. We must first give the relation corresponding to this type. It is built using the $\to$ and $\forall$ connectives.

We start by writing down the parametricity property for g and then unfolding the membership definition for the relation corresponding to the type of g

$$
\begin{array}{ll}
 & (g, g) \in \forall \mathcal{X}.(\mathcal{F}\ \mathcal{X} \to \mathcal{X}) \to \mathcal{X} \\
\iff & \quad \{ \text{ by membership definition for a } \forall \text{ relation } \} \\
 & \textit{for all } \mathcal{A} :: A \leftrightarrow A', \\
 & \quad (g\ A, g\ A') \in (\mathcal{F}\ \mathcal{A} \to \mathcal{A}) \to \mathcal{A} \\
\iff & \quad \{ \text{ by membership definition for a } (\to) \text{ relation } \} \\
 & \textit{for all } \mathcal{A} :: A \leftrightarrow A', \\
 & \quad \textit{for all } (k, k') \in \mathcal{F}\ \mathcal{A} \to \mathcal{A}, \\
 & \quad\quad (g\ A\ k, g\ A'\ k') \in \mathcal{A}
\end{array}
$$

We can go one step further for the part concerning (k, k'):

$$
\begin{array}{ll}
 & (k, k') \in \mathcal{F}\ \mathcal{A} \to \mathcal{A}, \\
\iff & \quad \{ \text{ by membership definition for a } (\to) \text{ relation } \} \\
 & \textit{for all } (x, x') \in \mathcal{F}\ \mathcal{A}, \\
 & \quad (k\ x, k'\ x') \in \mathcal{A}
\end{array}
$$

So the property as a whole is

$$
\begin{array}{l}
\textit{for all } \mathcal{A} :: A \leftrightarrow A', \\
\quad \textit{for all } k :: F\ A \to A, k' :: F\ A' \to A', \\
\quad\quad (\textit{for all } (x, x') \in \mathcal{F}\ \mathcal{A}, \\
\quad\quad\quad (k\ x, k'\ x') \in \mathcal{A}) \\
\quad\quad \implies \\
\quad\quad\quad (g\ A\ k, g\ A'\ k') \in \mathcal{A}
\end{array}
$$

Now instead of a relation $\mathcal{A} :: A \leftrightarrow A'$ we want to specialise to a function $h :: A \to A'$. Where we had $(x, x') \in \mathcal{A}$ we now get $h\ x = x'$ and for $(x, x') \in \mathcal{F}\ \mathcal{A}$ we now get $F\ h\ x = x'$.

$$
\begin{array}{l}
\textit{for all } h :: A \to A', \\
\quad \textit{for all } k :: F\ A \to A, k' :: F\ A' \to A', \\
\quad\quad (\textit{for all } x :: A, \\
\quad\quad\quad F\ h\ x = x' \implies h\ (k\ x) = k'\ x') \\
\quad\quad \implies \\
\quad\quad\quad h\ (g\ A\ k) = g\ A'\ k'
\end{array}
$$

We can simplify the inner part slightly by substituting x' and using function composition

$$
\begin{aligned}
& F\ h\ x = x' \implies h\ (k\ x) = k'\ x') \\
\iff & \\
& h\ (k\ x) = k'\ (F\ h\ x) \\
\iff & \\
& h \circ k = k' \circ F\ h
\end{aligned}
$$

We can now present the free theorem for g in a slightly more comprehensible fashion:

$$
\begin{aligned}
& g :: \forall a.(F\ a \to a) \to a \\
& h :: A \quad \to A' \\
& k :: F\ A \to A \\
& k' :: F\ A' \to A' \\
& h \circ k = k' \circ F\ h \implies h\ (g\ A\ k) = g\ A'\ k'
\end{aligned}
$$

□

3.4 Fusion with final co-data

The dual of data is co-data. While the abstract semantics of data uses initial algebras, the abstract semantics of co-data uses final co-algebras. In this section we give a short presentation of co-data, following that of Wadler (1990a); we check that the co-Church encoding satisfies the abstract semantics for co-data; finally we prove the *unfold*/*unbuild* property holds for final co-data in a parametric model.

3.4.1 Final co-data

For co-data, as with data, we start with a functor F. We continue to use the list functor as an example.

$$ListF\ e\ a = \mathbf{1} + (e, a)$$

While with data this functor gave us finite lists, with co-data we will get potentially-infinite lists.

For co-data, given a functor F we call the abstract type $\nu a.F\ a$. As before, we will abbreviate this as T. As with data, the abstract semantics for co-data is specified

in terms of the existence of certain operations and properties the operations satisfy. We get *in* and *out* functions

$$in :: F\ T \to T$$
$$out :: T \to F\ T$$

We also get an operation *unfold*

$$unfold :: \forall a.\ (a \to F\ a) \to a \to T$$

The *unfold* and *out* functions are required to satisfy the final co-algebra property, which we will now state. A co-algebra for the functor F is a pair (A, k) consisting of a type A and a function $k :: A \to F\ A$. A final co-algebra is a co-algebra that is final in the category of F-co-algebras, meaning that there is a unique mapping from all F-co-algebras to the final co-algebra. A mapping in the category of F-co-algebras, say from $(A, k :: A \to F\ A)$ to $(A', k' :: A' \to F\ A')$ is a function $h :: A \to A'$ that satisfies the property

$$F\ h \circ k = k' \circ h$$

For co-data T for a functor F we require that (T, out) is the final co-algebra. This mean there must be a unique mapping from any F-co-algebra, say (A, k) to the final co-algebra (T, out). This mapping is given by *unfold k*. Plugging this into the above property gives us the statement of the universal property of *unfold*

Property 3.4.1 (Universal property of *unfold*)**.**

$$F\ h \circ k = out \circ h$$
$$\iff$$
$$h = unfold\ A\ k$$

3.4.2 Universal property of *unfold* for lists

Again, it helps ones understanding to see what the universal property of *unfold* looks like in the concrete case of lists. Readers familiar with co-data may choose to skip this section.

As before we will represent the list functor $ListF\ e\ a = \mathbf{1} + (e, a)$ in Haskell types as

type $ListF\ e\ a = Maybe\ (e, a)$

So F is replaced by $ListF\ e$ and T by $[e]$.

The function $out :: T \to F\ T$ will be

$$
\begin{aligned}
&out :: [e] \to ListF\ e\ [e] \\
&out\ [\,] \quad\ = Nothing \\
&out\ (x : xs) = Just\ (x, xs)
\end{aligned}
$$

We now specialise $F\ h \circ k = out \circ h$, again writing $F\ h$ as $fmap\ h$. Previously when we looked at the universal property of *fold* on lists we were able to split on the *Nothing* and *Just* cases of the argument type. Here we must split on the cases of the result. Both functions $F\ h \circ k$ and $out \circ h$ have the result type $ListF\ e\ [e]$ so (without loss of generality) we can split on the case that the result is of the form *Nothing* or $Just\ (x, xs)$:

$$
\begin{aligned}
&\quad F\ h \circ k = out \circ h \\
\iff\ &\\
&\quad Nothing \quad\ = fmap\ h\ (k\ s) = out\ (h\ s) \\
&\vee\ Just\ (x, xs) = fmap\ h\ (k\ s) = out\ (h\ s)
\end{aligned}
$$

We can simplify both cases since we know the definition of *out* and *fmap* for the *ListF e* functor. In the *Nothing* case

$$
\begin{aligned}
&Nothing = fmap\ h\ (k\ s) \\
\iff\ &\\
&Nothing = k\ s
\end{aligned}
$$

and

$$
\begin{aligned}
&Nothing = out\ (h\ s) \\
\iff\ &\\
&h\ s = [\,]
\end{aligned}
$$

And in the $Just\ (x, xs)$ case

$$
\begin{aligned}
&Just\ (x, xs) = fmap\ h\ (k\ s) \\
\iff\ &\\
&k\ s = Just\ (y, ys) \wedge y = x \wedge xs = h\ ys
\end{aligned}
$$

and

$$
\begin{aligned}
&Just\ (x, xs) = out\ (h\ s) \\
\iff\ &\\
&(x : xs) \quad\ = h\ s
\end{aligned}
$$

If we combine and simplify the resulting conjunction of conditions we get

$$
\begin{array}{ll}
 & k\ s = \mathit{Just}\ (y, ys) \wedge y = x \wedge xs = h\ ys \wedge h\ s = (x : xs) \\
\iff & \quad \{\ \text{substitute } y := x\ \} \\
 & k\ s = \mathit{Just}\ (x, ys) \wedge xs = h\ ys \wedge h\ s = (x : xs) \\
\iff & \quad \{\ \text{substitute } xs := h\ ys\ \} \\
 & k\ s = \mathit{Just}\ (x, ys) \wedge h\ s = (x : h\ y) \\
\iff & \quad \{\ \alpha\text{-rename } ys \text{ to } s'\ \} \\
 & k\ s = \mathit{Just}\ (x, s') \wedge h\ s = (x : h\ y)
\end{array}
$$

If we put this all together we get

$$
\begin{array}{l}
h\ s = [\,] \iff k\ s = \mathit{Nothing} \\
\wedge \\
h\ s = x : h\ s' \iff k\ s = \mathit{Just}\ (x, s')
\end{array}
$$

Of course the standard way to express these conditions in Haskell syntax is using a case expression:

$$
\begin{array}{l}
h\ s = \mathbf{case}\ k\ s\ \mathbf{of} \\
\qquad \begin{array}{ll}
\mathit{Nothing} & \to [\,] \\
\mathit{Just}\ (x, s') & \to x : h\ s'
\end{array}
\end{array}
$$

The 'if' and 'only if' directions hold because the patterns and result have disjoint structure (corresponding to the fact that *in* and *out* are bijective and mutual inverses).

Thus overall, the universal property for *unfold* for lists is

$$
\begin{array}{l}
h\ s = \mathbf{case}\ k\ s\ \mathbf{of} \\
\qquad \begin{array}{ll}
\mathit{Nothing} & \to [\,] \\
\mathit{Just}\ (x, s') & \to x : h\ s'
\end{array} \\
\iff \\
h = \mathit{unfold}\ k
\end{array}
$$

which, fortunately, is the standard[2] way the property is presented.

[2]Some alternative but equivalent presentations in the literature (e.g. Gibbons and Jones, 1998) split the *unfold* argument function $a \to \mathit{Maybe}\ (b, a)$ into separate functions $a \to \mathit{Bool}$, $a \to b$ and $a \to a$.

3.4.3 co-Church encoding is final in parametric models

We must check that the co-Church encoding of co-data satisfies the abstract semantics we specified above, namely the universal property of *unfold*:

$$\begin{array}{l} F\ h \circ k = out \circ h \\ \iff \\ h = unfold\ A\ k \end{array}$$

We will consider each direction as separate lemmas. The easier direction is to check that $unfold\ A\ k$ is indeed a function h satisfying $F\ h \circ k = out \circ h$. We can do this purely syntactically by plugging in the definitions of *out* and *unfold*. The harder direction is to show that $unfold\ k$ is the unique function h satisfying the equation. As with the proof in the previous section for data and *fold*, this proof relies on us using a parametric model for System F.

Recall that the co-Church encoding represents co-data defined by a functor F using terms of the type $\exists a.(a \to T\ a, a)$. Thus in this context we use

$$T\ =\ \nu a.F\ a\ =\ \exists a.(a \to F\ a, a)$$

For terms with existentially quantified types we use the syntactic sugar from Section 2.1.4. In particular if $t :: \exists a.(a \to F\ a, a)$ then we may match t against $(a, (k, s))$ with types $k :: a \to F\ a$ and $s :: a$.

Lemma 3.4.2. $h = unfold\ A\ k \implies F\ h \circ k = out \circ h$

Proof. We start with lambda terms for *unfold* and *out*

$$\begin{array}{l} unfold :: \forall a.(a \to F\ a) \to a \to T \\ unfold = \Lambda a \to \lambda(k :: a \to F\ a) \to \lambda(s :: a) \to (a, (k, s)) \\ out :: T \to F\ T \\ out = \lambda(a, (k :: a \to F\ a, s :: a)) \to F\ (unfold\ a\ k)\ (k\ s) \end{array}$$

For the sake of presentation we will omit the explicit typing and use

$$\begin{array}{l} unfold\ a\ k\ s = (a, (k, s)) \\ out\ (a, (k, s)) = F\ (unfold\ a\ k)\ (k\ s) \end{array}$$

We substitute these into $F\ h \circ k = out \circ h$ with $h := unfold\ A\ k$ and η-expand, applying both sides to an argument s

$$F\ (unfold\ A\ k)\ (k\ s) = out\ (unfold\ A\ k\ s)$$

Starting with the right hand side we unfold definitions

$$\begin{aligned}
& out\ (unfold\ A\ k\ s) \\
= & \\
& out\ (A, (k, s)) \\
= & \\
& F\ (unfold\ A\ k)\ (k\ s)
\end{aligned}$$

Until we arrive at the left hand side. □

Lemma 3.4.3. $F\ h \circ k = out \circ h \implies h = unfold\ A\ k$

Proof. To show $unfold\ A\ k$ is the unique h satisfying $F\ h \circ k = out \circ h$ we must rely on parametricity. Specifically we use the free theorem for the type of *unfold*:

$$F\ h \circ k = k' \circ h \implies unfold\ A'\ k' \circ h = unfold\ A\ k$$

If we substitute $k' := out$ then this very nearly matches our goal

$$F\ h \circ k = out \circ h \implies unfold\ T\ out \circ h = unfold\ A\ k$$

If we can show that $unfold\ T\ out = id\ T$ then we are done. Fortunately we can do just that. We take the free theorem for *unfold* again and substitute $h := unfold\ A\ k$ and $k' := out$

$$\begin{aligned}
& F\ (unfold\ A\ k) \circ k = out \circ unfold\ A\ k \implies \\
& \qquad unfold\ T\ out \circ unfold\ A\ k = unfold\ A\ k
\end{aligned}$$

Of course the antecedent here is just what we showed in the previous lemma so we can reduce this to just

$$unfold\ T\ out \circ unfold\ A\ k = unfold\ A\ k$$

Now we just reduce this syntactically by η-expanding, unfolding the definition of *unfold* and η-reducing again

$$\begin{aligned}
& unfold\ T\ out \circ unfold\ A\ k = unfold\ A\ k \\
\implies & \quad \{\ \eta\text{-expand, apply to } s\ \} \\
& unfold\ T\ out\ (unfold\ A\ k\ s) = unfold\ A\ k\ s \\
\implies & \quad \{\ \text{by definition } unfold\ A\ k\ s = (A, (k, s))\ \} \\
& unfold\ T\ out\ (A, (k, s)) = (A, (k, s)) \\
\implies & \\
& unfold\ T\ out = id\ T
\end{aligned}$$

□

3.4.4 *unfold*/*unbuild* holds for final co-data in parametric models

Recall the *unfold*/*unbuild* property:

$$unbuild\ c\ g\ (unfold\ a\ k\ s) = g\ a\ k\ s$$

Our challenge is to prove this from the universal property of *unfold* and the assumption that we are working in a parametric model of System F. In particular we do not assume any concrete structure for the type $T = \forall a.F\ a$ or concrete lambda terms for *unfold* or *out*.

So our assumptions are that we have a covariant functor F, the type alias $T = \forall a.F\ a$, the abstract terms

$$unfold :: \forall a.(a \to F\ a) \to a \to T$$
$$out :: T \to F\ T$$

the universal property of *unfold*

$$F\ h \circ k = out \circ h$$
$$\iff$$
$$h = unfold\ A\ k$$

and the assumption that we are working in a parametric model.

While we start with an abstract definition of *unfold*, we must invent a suitable definition of *unbuild*. We will use

$$unbuild :: \forall b.(\forall a.(a \to F\ a) \to a \to b) \to T \to b$$
$$unbuild\ b\ g\ t = g\ T\ out\ t$$

Theorem 3.4.4 (*unfold*/*unbuild* fusion for final co-data)**.**

$$unbuild\ b\ g\ (unfold\ a\ k\ s) = g\ a\ k\ s$$

Proof. We start by unfolding definitions

$$unbuild\ b\ g\ (unfold\ a\ k\ s)$$
$$=$$
$$g\ T\ out\ (unfold\ a\ k\ s)$$

We cannot get far with unfolding definitions of course as *unbuild* is the only definition we know here; all the other terms are free variables or have abstract

definitions. To try and make some progress we turn to the free theorem for the type of g, namely

$$g :: \forall a.(a \to F\ a) \to a \to B$$

Note that at this point we can treat b as some fixed B, not a polymorphic b. This is important since the simpler type has a considerably simpler free theorem. The free theorem for g is

$$\begin{aligned}
&g :: \forall a.(a \to F\ a) \to a \to B \\
&h :: A \to A' \\
&k :: A \to F\ A \\
&k' :: A' \to F\ A' \\
&F\ h \circ k = k' \circ h \implies g\ A\ k = g\ A'\ k' \circ h
\end{aligned}$$

It turns out to be more convenient to η-expand (and flip) this as

$$F\ h \circ k = k' \circ h \implies g\ k'\ (h\ s) = g\ k\ s$$

Returning to the point where we got stuck unfolding definitions, it remains to show that

$$g\ T\ out\ (unfold\ a\ k\ s) = g\ a\ k\ s$$

This matches the consequent of the free theorem for g if we use the following substitutions

$$\begin{aligned}
h &:= unfold\ a\ k \\
A &:= a \\
k' &:= out \\
A' &:= T
\end{aligned}$$

Performing the substitution, we obtain

$$\begin{aligned}
&(F\ h \circ k = k' \circ h \implies g\ A'\ k'\ (h\ s) = g\ A\ k\ s)\ [h := unfold\ a\ k, \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad A := a, k' := out, A' := T] \\
\equiv\ &F\ (unfold\ a\ k) \circ k = out \circ unfold\ a\ k \implies \\
&\qquad\qquad\qquad\qquad\qquad\qquad g\ T\ out\ (unfold\ a\ k\ s) = g\ a\ k\ s
\end{aligned}$$

The antecedent $F\ (unfold\ a\ k) \circ k = out \circ unfold\ a\ k$ is of course a simple consequence of the universal property of *unfold*. Thus we have $g\ T\ out\ (unfold\ a\ k\ s) = g\ a\ k\ s$ which completes the proof. □

3.5 Streams without skip

We have seen abstract initial data with its natural accompanying functions *fold* and *build* and we have seen abstract final co-data with its functions *unfold* and *unbuild*. So what of streams?

A note on terminology. *Of course the term 'stream' refers only to sequences, however the trick of reformulating unfold/unbuild fusion as stream fusion applies not just for the list functor but for any functor. In lieu of a better name we will abuse terminology and refer to the 'stream form' of a functor F as Stream F. Readers incensed by this abuse may simply read this as the ordinary stream, i.e. using the list functor. We do not especially rely on the extra generality.*

3.5.1 Streams without skip are the co-Church encoding

If we start by looking at streams without skip then the stream form of a functor F is simply the co-Church encoding.

$$Stream\ F = \exists a.(a \to F\ a, a)$$

For example, the ordinary stream using the list functor $ListF\ e\ a = \mathbf{1} + (e, a)$ is

$$Stream\ (ListF\ e) = \exists a.(a \to \mathbf{1} + (e, a), a)$$

This corresponds to the definition of non-skipping streams that we gave in the first chapter, using Haskell syntax

$$\mathbf{data}\ Stream\ e = \exists a.\ Stream\ (a \to Maybe\ (e, a))\ a$$

The *stream* and *unstream* functions are supposed to convert between the 'ordinary' co-data type $\nu a.F\ a$ and the stream representation. Of course if we are using the co-Church encoding to implement $\nu a.F\ a$ then *stream* and *unstream* are just identity functions. If however we consider $\nu a.F\ a$ as abstract final co-data then we can still implement *stream* and *unstream* trivially in terms of *unbuild* and *unfold*. Again, the practical importance of working from the abstract definition of co-data is that we expect a real programming language to implement co-data not as the co-Church encoding but using some efficient machine representation which nevertheless implements the abstract semantics.

Recall the types of *unbuild* and *unfold*

$$unfold\ \ :: \forall a.\ (a \to F\ a) \to a \to \nu b.F\ b$$
$$unbuild :: \forall c.\ (\forall a.(a \to F\ a) \to a \to c) \to \nu b.F\ b \to c$$

We will define *stream* and *unstream* in terms of *unbuild* and *unfold* as follows

Definition 3.5.1 (*stream* and *unstream* for non-skipping streams)**.**

$$
\begin{aligned}
&\mathit{stream} :: \nu a.F\ a \to \mathit{Stream}\ F \\
&\mathit{stream} = \mathit{unbuild}\ (\mathit{Stream}\ F)\ (\Lambda a \to \lambda k \to \lambda s \to (a, (k, s)))\ )
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unstream} :: (\mathit{Stream}\ F) \to \nu a.F\ a \\
&\mathit{unstream}\ (a, (k, s)) = \mathit{unfold}\ a\ k\ s
\end{aligned}
$$

3.5.2 *stream*/*unstream* holds for final co-data in parametric models

With these definitions of *stream* and *unstream* the stream fusion rule is a simple consequence of the *unbuild*/*unfold* fusion rule.

Theorem 3.5.2 (Stream fusion for non-skipping streams in System F)**.**

$$\mathit{stream} \circ \mathit{unstream} = \mathit{id}\ (\mathit{Stream}\ F)$$

Proof. It is sufficient to show that

$$\mathit{stream}\ (\mathit{unstream}\ (a, (k, s))) = (a, (k, s))$$

That is, in addition to η-expanding, we can use $(a, (k, s))$ as the term of the stream type $\exists a.(a \to F\ a, a)$. This is because, in a parametric model, all values of existential type correspond to some term (A, x) (See e.g. Plotkin and Abadi, 1993, Section 3, Theorem 7).

$$
\begin{aligned}
&\mathit{stream}\ (\mathit{unstream}\ (a, (k, s)))\ ) \\
=\quad &\{\ \text{by definition of } \mathit{unstream}\ \} \\
&\mathit{stream}\ (\mathit{unfold}\ a\ k\ s) \\
=\quad &\{\ \text{by definition of } \mathit{stream}\ \} \\
&\mathit{unbuild}\ (\mathit{Stream}\ F)\ (\Lambda a \to \lambda k \to \lambda s \to (a, (k, s)))\ (\mathit{unfold}\ a\ k\ s) \\
=\quad &\{\ \text{by } \mathit{unbuild}\ c\ g\ (\mathit{unfold}\ a\ k\ s) = g\ a\ k\ s \text{ fusion rule}\ \} \\
&(\Lambda a \to \lambda k \to \lambda s \to (a, (k, s)))\ )\ a\ k\ s \\
=\quad &\{\ \text{by } \beta\text{-reduction}\ \} \\
&(a, (k, s))
\end{aligned}
$$

□

Note that we have not relied on any concrete definitions of *unbuild* and *unfold*, we have only relied on the *unbuild*/*unfold* fusion rule and the definitions of *stream* and *unstream*. Thus since the *unbuild*/*unfold* fusion rule holds for final co-data in a parametric model then the *stream*/*unstream* rule does too.

Having shown that $stream \circ unstream$ is the identity we are half way to confirming our expectation that *Stream F* is isomorphic to $\nu a.F\ a$. It just remains to show that $unstream \circ stream$ is also the identity.

Lemma 3.5.3. $unstream \circ stream = id\ (\nu a.F\ a)$

Proof. The only interesting part of the proof is the use of the fact that since $(\nu a.F\ a, out_F)$ is the final F-co-algebra then $unfold\ (\nu a.F\ a)\ out_F = id\ (\nu a.F\ a)$.

$$
\begin{aligned}
& unstream\ (stream\ x) \\
= \quad & \{ \text{ unfold definition of } stream \ \} \\
& unstream\ (unbuild\ (Stream\ F)\ (\Lambda a \to \lambda k \to \lambda s \to (a, (k, s)))\ )\ x) \\
= \quad & \{ \text{ unfold definition of } unbuild \ \} \\
& unstream\ ((\Lambda a \to \lambda k \to \lambda s \to (a, (k, s)))\ )\ (\nu a.F\ a)\ out_F\ x) \\
= \quad & \{ \ \beta\text{-reduce} \ \} \\
& unstream\ (\nu a.F\ a, (out_F, x)) \\
= \quad & \{ \text{ unfold definition of } unstream \ \} \\
& unfold\ (\nu a.F\ a)\ out_F\ x \\
= \quad & \{ \ unfold\ (\nu a.F\ a)\ out_F = id\ (\nu a.F\ a) \ \} \\
& x
\end{aligned}
$$

□

3.6 Streams with skip

We now consider streams with skip. As we saw in Section 1.3.8, using the *unbuild*/*unfoldr* fusion system, we cannot effectively fuse functions like *filter* with standard local transformations. As we saw in Section 1.4.2, adding skip makes the straightforward local transformation approach work for *filter* as well.

Thus the reason we wish to add skip is purely practical, not theoretical. Indeed it adds tiresome complexity to the theory. Adding skip does appear however to be essential so we must also deal with skip in the theory.

Recall that without skip, the stream form for a functor F is

$$Stream\ F = \exists a.(a \to F\ a, a)$$

To add skip we use an extended functor $Skip\ F$ which we define as

$$Skip\ F\ a = F\ a + a$$

The extra '$+\ a$' corresponds to the additional possibility to skip. Then we define a skipping stream as the stream form of the extended functor

$$\begin{aligned} Stream\ (Skip\ F) &= \exists a.(a \to \ \ Skip\ F\ a, a) \\ &= \exists a.(a \to (F\ a + a), a) \end{aligned}$$

For example, with the list functor

$$ListF\ e\ a = \mathbf{1} + (e, a)$$

the skip form is

$$Skip\ (ListF\ e)\ a = (\mathbf{1} + (e, a)) + a$$

This corresponds to the Haskell *Step* data type that we introduced in Section 1.4.2.

$$\begin{aligned} \mathbf{data}\ Step\ a\ s = &\ Done \\ \mid &\ Skip\quad s \\ \mid &\ Yield\ a\ s \end{aligned}$$

The intended meaning of the skipping stream type is to be 'the same' as the non-skipping stream type. The skips should somehow be ignored; values of $Stream\ (Skip\ F)$ that differ only in the skips are supposed to represent the same value of $\nu a.F\ a$.

The difficulty of course with skipping streams is that they are not isomorphic to the ordinary co-data $\nu a.F\ a$. Instead we have many values in $Stream\ (Skip\ F)$ that all represent the same value of $\nu a.F\ a$.

A helpful notion is that of data abstraction: the $Stream\ (Skip\ F)$ type is intended to be a representation of the abstract type $\nu a.F\ a$. Of course $Stream\ F$ is also a representation of $\nu a.F\ a$ but a rather simpler one since the types are isomorphic.

To formalise the relationship between skipping streams and the corresponding ordinary co-data we need an abstraction function and data type invariant. We will postpone consideration of whether or not we need a non-trivial invariant. The abstraction function maps each value of $Stream\ (Skip\ F)$ to the value of

$\nu a.F\ a$ which it represents. We will expect the abstraction function to coincide with the *unstream* function for skipping streams so we will call the abstraction function *unstream*

$$unstream :: Stream\ (Skip\ F) \to \nu a.F\ a$$

The abstraction function will have to 'erase' all of the skips. The abstraction function must be surjective and this gives us an equivalence relation $Stream\ (Skip\ F)$. The equivalence relation relates values of $Stream\ (Skip\ F)$ that map to the same value of $\nu a.F\ a$, i.e. those that differ only in the skips.

$$s \approx s' \iff unstream\ s = unstream\ s'$$

An additional burden is that all functions that take a $Stream\ (Skip\ F)$ as input will have to respect the data abstraction. That is, a function applied to equivalent input streams should give us the same results. We cannot allow a function operating on $Stream\ (Skip\ F)$ – that is meant to represent a function on $\nu a.F\ a$ – to distinguish values of $Stream\ (Skip\ F)$ that are indistinguishable in $\nu a.F\ a$.

So the first task is to define the abstraction function. Unfortunately we fall at this first hurdle; we cannot define the abstraction function in System F. A function that erases the skips cannot be written. The problem is that, with the way we have formulated $Stream\ (Skip\ F)$ there can be unbounded sequences of skips. Needless to say, no function to erase an unbounded sequences of skips can be written in a strongly normalising language such as System F.

We might consider reformulating the skipping stream type so that it only has finite sequences of skips. This is possible, by using data for the sequences of skips rather than co-data

$$FiniteSkip\ F\ a = F\ (\mu b.b + a)$$
$$Stream\ (FiniteSkip\ F) = \exists a.(a \to F\ (\mu b.b + a), a)$$

Though this is possible, it is not useful. Modelling skips as data rather than co-data would be a move away from what we ultimately want to implement in the real system. Furthermore, skip is primarily to deal with functions like *filter* and these cannot be implemented for co-data in strongly normalising languages. We must conclude that ordinary System F is not the right framework to use to study skipping streams and that it is time to move to our next semantic model: CPOs.

3.7 Streams with skip in CPOs

In this section we will use the language we introduced in Section 2.3. It uses the syntax of System F but with a CPO semantics and a fixpoint combinator as an additional constant term.

Of course in a CPO the semantics of a stream is changed somewhat. In particular the CPO semantics allows for partial streams. We can construct a partial stream by having the stepper function produce $\perp$ after some number of unfolding steps.

In a CPO we can capture the relationship between skipping streams $Stream\ (Skip\ F)$ and ordinary co-data $va.F\ a$. In particular, the previously problematic case of an infinite sequence of skips can now be mapped to $\perp$, giving us partial co-data. We can implement this mapping using the fixpoint combinator. As mentioned above, the abstraction function is just *unstream* on skipping streams

$$
\begin{aligned}
&unstream :: Stream\ (Skip\ F) \to va.F\ a \\
&unstream\ (a, (k, s)) = unfold\ a\ (force\ a\ k)\ s \\
&force :: \forall a.(a \to Skip\ F\ a) \to (a \to F\ a) \\
&force\ a = fix\ (force'\ a) \\
&force'\ a\ f\ next = \lambda(s :: a) \to case_{sum}\ (next\ s) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\lambda(y :: F\ a) \to y) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\lambda(s' :: a) \quad \to f\ next\ s')
\end{aligned}
$$

We make explicit use of *fix* because this is the form we need for subsequent syntactic proofs. It may be helpful however to see an alternative presentation to clarify the meaning

$$
\begin{aligned}
&force :: \forall a.(a \to Skip\ F\ a) \to (a \to F\ a) \\
&force\ next = \lambda s \to \textbf{case}\ next\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad Left \quad y \to y \\
&\qquad\qquad\qquad\qquad Right\ s' \to force\ next\ s'
\end{aligned}
$$

The *force* function converts a stepper function for a skipping stream into a stepper function for a non-skipping stream. It recurses in the case of skip, hence an unbounded sequence of skips gives $\perp$. The *unstream* abstraction function is simply the composition of the *unstream* function for non-skipping streams with the *force* conversion from skipping to non-skipping streams.

The *stream* function maps $va.F\ a$ into $Stream\ (Skip\ F)$. It is straightforward in that it maps to a stream without any uses of skip.

We can define it as

$$\begin{aligned}&\mathit{stream} :: \mathit{va}.F\ a \to \mathit{Stream}\ (\mathit{Skip}\ F)\\&\mathit{stream} = \mathit{unbuild}\ (\mathit{Stream}\ F)\ (\Lambda a \to \lambda k \to \lambda s \to (a, (\mathit{left} \circ k, s)))\end{aligned}$$

The only difference with *stream* for non-skipping streams is the extra composition of f with *left*. This lifts the stepper function from $a \to F\ a$ to $a \to \mathit{Skip}\ F\ a$, since $\mathit{Skip}\ F\ a = F\ a + a$.

As mentioned above, the abstraction function gives us an equivalence relation on $\mathit{Stream}\ (\mathit{Skip}\ F)$ that describes 'equivalence modulo skip'. It will be useful to extend equivalence modulo skip to all types, including those that contain skipping streams as sub-components. That is, we want an equivalence relation that is ordinary value equality but that respects the data abstraction for skipping streams. The reason for this extra generality is that later it will let us write some properties rather more concisely and with greater generality.

The appropriate formalisation that gives us an equivalence relation with this property is a *logical relation*. A logical relation is a relation on values that is made up of a family of relations, indexed by the type. Our types are function types, functorial types like sums and pairs, streams and base types such as *Int*.

$$\star ::= \star \to \star \mid F \star \mid \mathit{Stream}\ (\mathit{Skip}\ F) \mid \mathit{Int} \mid \ldots$$

In our case for equivalence we only need a binary relation. We write $a\ \mathcal{R}_A\ b$ for values $a, b :: A$ that are related by $\mathcal{R}_A$ which is the member of the family of relations $\mathcal{R}$ for the type A. The relations that make up the family must respect the following property

$$f\ \mathcal{R}_{A\to B}\ g \iff \forall x\ x'.\ x\ \mathcal{R}_A\ x' \implies f\ x\ \mathcal{R}_B\ g\ x'$$

That is, the relation for type $A \to B$ has to be consistent with the relation for type A and the relation for type B. For a family of relations that represents equivalence, this property is just extensional equivalence for functions. We can construct a family of relations that has this property by specifying the relations for base types and then using the property above as a definition for all function types.

It is worth noting that we have seen this kind of logical relation before (Section 2.1.3): the parametricity property is stated as $t\ \mathcal{T}\ t$ where the binary relation $\mathcal{T}$ is constructed as a logical relation.

Specifically for our $\approx$ equivalence relation we have the property for function types

$$f \approx g \iff \forall x\ x'.\ x \approx x' \implies f\ x \approx g\ x'$$

At skipping stream types we define $\approx$ to be equivalence modulo skip

$$s \approx s' \iff \mathit{unstream}\ s = \mathit{unstream}\ s'$$

If our language has any extra atomic types (e.g. machine integers) then $\approx$ is ordinary equality at that type. At functorial types such as pairs and sums, $\approx$ relates values with the same structure and equivalent components. For example, for pairs we have

$$(x, y) \approx (x', y') \iff x \approx x' \wedge y \approx y'$$

This behaviour at functorial types is prescribed since it has to match the behaviour of the Church encoding which is itself prescribed by the behaviour at function types.

It is worth noting that for types that do not contain the stream type then $\approx$ is simply $=$. We can see this by induction on the structure of a type. It is true by definition for base types and is preserved for function and other functorial type constructors.

3.8 Fusion for streams with skip

3.8.1 Sufficient conditions for stream fusion

Recall that overall we are trying to prove transformations such as

$$\begin{array}{ll} & \mathit{unstream} \circ f_s \circ \mathit{stream} \circ \mathit{unstream} \circ g_s \circ \mathit{stream} \\ = & \quad \{\ \text{“stream fusion”}\ \} \\ & \mathit{unstream} \circ f_s \circ g_s \circ \mathit{stream} \end{array}$$

The property $\mathit{stream} \circ \mathit{unstream} = \mathit{id}$ is certainly sufficient to prove the transformation. Unfortunately we know that this property is not true for skipping streams so we are interested in finding a weaker yet sufficient property. The intuition is that we are working in a context where we do not need exact equality on streams, only an equivalence.

Let us derive a weaker condition and then attempt to prove any lemmas we find that we need. We start from our original example

$$\mathit{unstream} \circ f_s \circ \mathit{stream} \circ \mathit{unstream} \circ g_s \circ \mathit{stream}$$

This example is rather too specific. A simpler and more general case would be this transformation

$$\begin{array}{ll} & \mathit{unstream}\ (f_s\ (\mathit{stream}\ (\mathit{unstream}\ s))) \\ = & \quad \{\ \text{“stream fusion”}\ \} \\ & \mathit{unstream}\ (f_s\ s) \end{array}$$

This still assumes however that f_s both consumes and produces a stream. It would be better to allow f_s to produce any type at all, including types containing a stream as a sub-component (e.g. a pair of streams). We can generalise the above property by making use of our $\approx$ equivalence relation.

$$f_s\ (\mathit{stream}\ (\mathit{unstream}\ s)) \approx f_s\ s$$

For a function f_s that does produce a stream then this property is the same as the previous one above. For a f_s that returns some other non-stream type then we get an appropriate form of equality. In particular, if the result type contains no stream types as sub-components then $\approx$ is straightforward equality. The question now is what lemmas and assumptions would be sufficient to prove this fusion property.

The function $(\mathit{stream} \circ \mathit{unstream}) :: \mathit{Stream}\ a \to \mathit{Stream}\ a$ has the effect of eliminating all the skips, though the resulting stream should still be equivalent to the original. Looking at the fusion property, in the right hand side the f_s function is presented with a stream possibly containing skips while on the left side $f_s\ (\mathit{stream}\ (\mathit{unstream}\ s))$ all the skips have been eliminated. In both cases we require f_s to produce equivalent results. That is, we present f_s with different, albeit equivalent, input streams and expect the results also to be equivalent. It is clear that we will need some assumption about f_s. The f_s function has unrestricted access to the skips and it could distinguish between equivalent streams on the basis of differences in the skips. What we want is for f_s to preserve equivalence on streams, that is it should have the property

Property 3.8.1 (preservation of equivalence)**.**

$$s \approx s' \implies f_s\ s \approx f_s\ s'$$

Or equivalently, by the definition of $\approx$ at function types

$$f_s \approx f_s$$

As we noted, $stream \circ unstream$ eliminates skips but the resulting stream should still be equivalent. That is, we would hope that the following lemma holds.

Lemma 3.8.2. $stream\ (unstream\ s) \approx s$

We defer the proof of this lemma for a moment.

This lemma and the assumption about f_s are sufficient to prove the fusion rule for skipping streams.

Theorem 3.8.3 (Stream fusion for skipping streams in CPOs)**.** When f_s preserves equivalence on streams (i.e. $f_s \approx f_s$) then

$$f_s\ (stream\ (unstream\ s)) \approx f_s\ s$$

Proof. Take the definition of $f_s \approx f_s$

$$\forall x\ x'.\ x \approx x' \implies f_s\ x \approx f_s\ x'$$

We are free to pick x and x'. Take $x := stream\ (unstream\ s), x' := s$

$$stream\ (unstream\ s) \approx s \implies f_s\ (stream\ (unstream\ s)) \approx f_s\ s$$

The antecedent here is our lemma $stream\ (unstream\ s) \approx s$ and the consequent is the statement of the theorem. □

Proof of Lemma 3.8.2. Since s is a stream we can expand out the definition of $\approx$ at this type.

$$\begin{array}{l} stream\ (unstream\ s) \approx s \\ \iff \\ unstream\ (stream\ (unstream\ s)) = unstream\ s \end{array}$$

This will hold if it is the case that

$$unstream\ (stream\ x) = x$$

which we will verify as a separate lemma □

So if we can show that $unstream \circ stream$ is still an identity then it follows easily that $stream\ (unstream\ s) \approx s$. There is good reason to believe that $unstream \circ stream$ is still an identity on skipping streams despite the fact that $stream \circ unstream$ is not. This is because *stream* gives us a stream with no skips which *unstream* should then map back to the same original stream.

Lemma 3.8.4. $unstream\ (stream\ x) = x$

Proof. The proof is primarily syntactic, with the final step relying on a semantic property of domains. In particular we rely on the fact final co-algebras exist (without needing any restriction to domains of strict functions). The specific property we make use of is that $unfold\ T\ out = id\ T$. By finality of the co-algebra (T, out), the function $unfold\ T\ out$ is the unique function from T to T but so is $id\ T :: T \to T$ and thus $unfold\ T\ out = id\ T$.

We start by unfolding definitions for *stream*, *unstream* and *unbuild*.

$$
\begin{aligned}
& unstream\ (stream\ x) \\
= \quad & \{\ \text{unfold definition of } stream\ \} \\
& unstream\ (unbuild\ (Stream\ F)\ (\Lambda a \to \lambda k \to \lambda s \to (a, (left \circ k, s))\ )\ x) \\
= \quad & \{\ \text{unfold definition of } unbuild\ \} \\
& unstream\ ((\Lambda a \to \lambda k \to \lambda s \to (a, (left \circ k, s))\ )\ T\ out\ x) \\
= \quad & \{\ \beta\text{-reduce}\ \} \\
& unstream\ (T, (left \circ out, x)) \\
= \quad & \{\ \text{unfold definition of } unstream\ \} \\
& unfold\ T\ (force\ T\ (left \circ out))\ x
\end{aligned}
$$

Had we been working with non-skipping streams then at this stage we would have had simply $unfold\ T\ out\ x$ but instead we have *left* to lift into the skipping stream functor and *force* to map back into non-skipping streams. Our hope of course is that these two cancel each other out

$$
\begin{aligned}
& force\ T\ (left \circ out) \\
= \quad & \{\ \text{unfold definition of } force\ \} \\
& (fix\ (force'\ T))\ (left \circ out) \\
= \quad & \{\ \text{fixpoint rule } fix\ f = f\ (fix\ f)\ \} \\
& force'\ T\ (fix\ (force'\ t))\ (left \circ out) \\
= \quad & \{\ \text{unfold definition of } force', (left \circ out)\ s = left\ (out\ s)\ \} \\
& \lambda(s :: T) \to case_{sum}\ (left\ (out\ s)) \\
& \qquad\qquad (\lambda(y :: F\ T) \to y) \\
& \qquad\qquad (\lambda(s' :: T) \to fix\ (force'\ t)\ (left \circ out)\ s') \\
= \quad & \{\ \text{use the rule } case_{sum}\ (left\ x)\ l\ r = l\ x\ \} \\
& \lambda(s :: T) \to (\lambda(y :: F\ T) \to y)\ (out\ s) \\
=
\end{aligned}
$$

$$
\begin{aligned}
&= \quad \{\ \beta\text{-reduce}\ \} \\
&\quad \lambda(s :: T) \to out\ s \\
&= \quad \{\ \eta\text{-reduce (valid since } out \text{ is not } \bot)\ \} \\
&\quad out
\end{aligned}
$$

We can return to where we left off and apply $force\ T\ (left \circ out) = out$

$$
\begin{aligned}
&\quad unfold\ T\ (force\ T\ (left \circ out))\ x \\
&= \quad \{\ force\ T\ (left \circ out) = out\ \} \\
&\quad unfold\ T\ out\ x \\
&= \quad \{\ unfold\ T\ out = id\ T\ \} \\
&\quad x
\end{aligned}
$$

□

3.8.2 Use as an automatic transformation system

To summarise, we know that

$$
\begin{aligned}
&unstream\ (stream\ x) = x \\
&stream\ (unstream\ s) \approx s
\end{aligned}
$$

and that the stream fusion rule holds provided that f_s preserves equivalence on streams

$$f_s \approx f_s \Longrightarrow f_s\ (stream\ (unstream\ s)) \approx f_s\ s$$

Before going any further it is worth considering if this fusion rule is useful. One of the touted advantages of short-cut fusion over previous more general fusion systems was that it uses a syntactic transformation with no side conditions[3]. This fact makes short-cut fusion relatively easy to integrate into a compiler.

On the face of it we are in danger of designing a fusion system with a fusion rule that is not actually an equation, merely an equivalence, and worse it is a rule that has a tricky side condition. One may well wonder how can we possibly justify the simple rewrite rule $stream\ (unstream\ s) = s$ that we introduced in Chapter 1.

The intuition is that since skipping streams are a data abstraction, that by hiding the representation we can hope to draw a boundary that gives us a

[3]Ignoring for the moment that *foldr/build* does have a strictness side condition when used in CPOs.

context where, by construction, the fusion rule is an actual equality and the side conditions are automatically satisfied.

The approach is to make the stream type itself a non-observable type. That is the representation is hidden by some mechanism (such as a module) and the only way to view the stream type is via a limited library of stream functions. These functions are all required to respect the data abstraction, meaning they must preserve equivalence on streams. The syntactic context of an expression $stream\ (unstream\ s)$ is then some combination of stream functions from the library.

$$C\ [stream\ (unstream\ s)]$$

It is worth observing again that the equivalence relation $\approx$ becomes simple equality when used at types that do not contain the stream type anywhere. Of course only types that do not contain the stream type are observable. By assumption the overall context is at an observable type. We can turn the context C into a function by lambda abstracting over the position of the occurrence of $stream\ (unstream\ s)$. This gives us an expression that is an instance of the left hand side of the fusion rule. Since it is at an observable type then the $\approx$ is equality. So if we can be sure that the context preserves equivalence on streams then we can apply the fusion rule as an equality.

The argument that the context does preserves equivalence is by induction on the syntactic structure of the context. It is trivially true that $f \approx f$ holds for all external values that do not involve the stream type. By assumption $f_s \approx f_s$ for all the stream functions in the library. As $\approx$ is a logical relation it is preserved by abstraction and application. Thus all combinations of external values and stream functions from the library give contexts that preserve equivalence.

So we may apply the fusion transformation unconditionally since the context guarantees the side condition

$$C\ [stream\ (unstream\ s)] = C\ [s]$$

So an implementation may use $stream\ (unstream\ s) = s$ as a rewrite rule provided that it uses a mechanism such as a module to ensure that the stream type is not externally observable and that each function f_s that does have direct access to the representation respects the data abstraction, i.e. $f_s \approx f_s$.

3.8.3 Properties that library functions must satisfy

Recall that the first stage in the stream fusion process is a transformation such as

$$\begin{array}{l} \quad map\ f \circ filter\ p \\ = \\ \quad unstream \circ map_s\ f \circ stream \circ unstream \circ filter_s\ p \circ stream \end{array}$$

We need to know that the stream versions of these functions are faithful to the original versions. In this example we are relying on these equations

$$\begin{array}{l} map\ f\ = unstream \circ map_s\ f\ \circ stream \\ filter\ p = unstream \circ filter_s\ p \circ stream \end{array}$$

We cannot dodge our obligations by using these equations as definitions since we still need to know that these re-definitions are equal to the original list versions.

So for each function in our library of stream functions we are obliged to show both that it preserves equivalence on streams and that it is faithful to the corresponding original function. We can reduce this effort somewhat by using a single data abstraction property which covers both of these obligations.

The abstraction property relates functions on streams to the corresponding functions on ordinary data via the abstraction function *unstream*. For example, if f_s consumes a stream to produce some other type (e.g. a function like sum_s) and the corresponding list version is f then the combined obligation is the standard data abstraction property

$$f_s = f \circ unstream$$

It is straightforward to show that this justifies the form $f = f_s \circ stream$ which we use as a rewrite rule.

Lemma 3.8.5.

$$\begin{array}{l} \quad f_s = f \circ unstream \\ \Longrightarrow \\ \quad f_s \circ stream = f \end{array}$$

Proof.

$$
\begin{array}{ll}
& f_s = f \circ \mathit{unstream} \\
\Longrightarrow & \quad \{ \text{ compose both sides with } \mathit{stream} \ \} \\
& f_s \circ \mathit{stream} = f \circ \mathit{unstream} \circ \mathit{stream} \\
\Longleftrightarrow & \quad \{ \text{ by lemma } \mathit{unstream} \circ \mathit{stream} = \mathit{id} \ \} \\
& f_s \circ \mathit{stream} = f
\end{array}
$$

□

The more important point is that if f_s and f satisfy the data abstraction property then f_s preserves equivalence on streams.

Lemma 3.8.6.

$$
\begin{array}{l}
f_s = f \circ \mathit{unstream} \\
\Longrightarrow \\
s \approx s' \implies f_s\ s \approx f_s\ s'
\end{array}
$$

Proof. First an equality that we will need

$$
\begin{array}{ll}
& f_s = f \circ \mathit{unstream} \\
\Longrightarrow & \quad \{ f = f_s \circ \mathit{stream} \ \} \\
& f_s = f_s \circ \mathit{stream} \circ \mathit{unstream}
\end{array}
$$

Now we start with $s \approx s'$ and work towards $f_s\ s \approx f_s\ s'$

$$
\begin{array}{ll}
& s \approx s' \\
\Longrightarrow & \quad \{ \text{ definition of } \approx \text{ at stream types } \} \\
& \mathit{unstream}\ s = \mathit{unstream}\ s' \\
\Longrightarrow & \quad \{ \text{ apply } f_s \circ \mathit{stream} \text{ to both sides } \} \\
& f_s\ (\mathit{stream}\ (\mathit{unstream}\ s)) = f_s\ (\mathit{stream}\ (\mathit{unstream}\ s')) \\
\Longrightarrow & \quad \{ f_s \circ \mathit{stream} \circ \mathit{unstream} = f_s \ \} \\
& f_s\ s = f_s\ s' \\
\Longrightarrow & \quad \{ \approx \text{ at some other non-stream type } \} \\
& f_s\ s \approx f_s\ s'
\end{array}
$$

□

Of course the data abstraction property $f_s = f \circ \mathit{unstream}$ is only for the type of functions that consume a stream to produce some other non-stream type. We also have to worry about functions that transform streams or just produce streams. Indeed, more generally we want a data abstraction property for functions between any types and where those types may contain streams as sub-components.

Consider for example *concatMap* on lists and the corresponding version on streams

$$\begin{aligned}&\mathit{concatMap} &&:: (a \to [b]) \to [a] \to [b]\\ &\mathit{concatMap}_s &&:: (a \to \mathit{Stream}\ b) \to \mathit{Stream}\ a \to \mathit{Stream}\ b\end{aligned}$$

With some thought we may conclude that the appropriate abstraction property between these should be

$$\mathit{unstream} \circ \mathit{concatMap}_s\ f_s = \mathit{concatMap}\ (\mathit{unstream} \circ f_s) \circ \mathit{unstream}$$

Clearly the appropriate abstraction property depends on the type.

The general property for f_s and f is given by an abstraction relation $\mathcal{A}$ which is another binary logical relation

$$f_s\ \mathcal{A}\ f$$

This time it is not an equivalence relation because we relate different types. We relate concrete types with abstract types. As before if we have any atomic base types (e.g. machine integers) then we use simple equality. The interesting case is that we relate the concrete stream types to the corresponding abstract data type using the abstraction function *unstream*

$$s\ \mathcal{A}\ x \iff \mathit{unstream}\ s = x$$

For function types we have the standard logical relation property

$$f\ \mathcal{A}\ g \iff x\ \mathcal{A}\ x' \Rightarrow f\ x\ \mathcal{A}\ g\ x$$

Note that the $\mathcal{A}$ relation is the identity for all types that do not contain any stream types.

The definition of the $\mathcal{A}$ relation means that for each specific type we can get the abstraction property in an equational form. For example we can derive the abstraction property for $\mathit{concatMap}_s$ and *concatMap*. We simply expand the definition of the relation based on the type.

$$\begin{aligned}&\mathit{concatMap}_s\ \mathcal{A}\ \mathit{concatMap}\\ \Longrightarrow\quad &\{\ \text{definition of } \mathcal{A} \text{ at function types}\ \}\\ &f_s\ \mathcal{A}\ f \implies \mathit{concatMap}_s\ f_s\ \mathcal{A}\ \mathit{concatMap}\ f\end{aligned}$$

Take $f_s \; \mathcal{A} \; f$ separately

$$
\begin{array}{ll}
 & f_s \; \mathcal{A} \; f \\
\Longrightarrow & \quad \{ \text{ definition of } \mathcal{A} \text{ at function types } \} \\
 & x \; \mathcal{A} \; x' \Longrightarrow f_s \; x \; \mathcal{A} \; f \; x' \\
\Longrightarrow & \quad \{ \; x \; \mathcal{A} \; x' \text{ type does not contain stream types } \} \\
 & x = x' \Longrightarrow f_s \; x \; \mathcal{A} \; f \; x' \\
\Longrightarrow & \quad \{ \text{ true for all } x, x' \text{ so can pick } x' := x \; \} \\
 & f_s \; x \; \mathcal{A} \; f \; x \\
\Longrightarrow & \quad \{ \text{ definition of } \mathcal{A} \text{ for stream type } \} \\
 & \mathit{unstream} \; (f_s \; x) = f \; x \\
\Longrightarrow & \quad \{ \text{ function composition and } \eta\text{-reduce } \} \\
 & \mathit{unstream} \circ f_s = f
\end{array}
$$

Returning to the outer derivation, we substitute for f

$$
\begin{array}{ll}
 & \mathit{concatMap}_s \; f_s \; \mathcal{A} \; \mathit{concatMap} \; (\mathit{unstream} \circ f_s) \\
\Longrightarrow & \quad \{ \text{ definition of } \mathcal{A} \text{ at function types } \} \\
 & s \; R \; x \Rightarrow \mathit{concatMap}_s \; f_s \; s \; \mathcal{A} \; \mathit{concatMap} \; (\mathit{unstream} \circ f_s) \; x \\
\Longrightarrow & \quad \{ \text{ definition of } \mathcal{A} \text{ for stream type } \} \\
 & \mathit{unstream} \; s = x \Rightarrow \mathit{concatMap}_s \; f_s \; s \; \mathcal{A} \; \mathit{concatMap} \; (\mathit{unstream} \circ f_s) \; x \\
\Longrightarrow & \\
 & \mathit{concatMap}_s \; f_s \; s \; \mathcal{A} \; \mathit{concatMap} \; (\mathit{unstream} \circ f_s) \; (\mathit{unstream} \; s) \\
\Longrightarrow & \quad \{ \text{ definition of } \mathcal{A} \text{ for stream type } \} \\
 & \mathit{unstream} \; (\mathit{concatMap}_s \; f_s \; s) = \mathit{concatMap} \; (\mathit{unstream} \circ f_s) \; (\mathit{unstream} \; s) \\
\Longrightarrow & \\
 & \mathit{unstream} \circ \mathit{concatMap}_s \; f_s = \mathit{concatMap} \; (\mathit{unstream} \circ f_s) \circ \mathit{unstream}
\end{array}
$$

Which is the abstraction property we stated previously for *concatMap*.

3.8.4 Notes on strictness

Historically, shortcut fusion results have been justified using free theorems. Johann and Voigtländer (2006) demonstrated that the classical free theorems do not hold unconditionally once we add fixpoints and polymorphic *seq*, but require certain side conditions. They give examples for the *foldr/build* rule and the *unbuild/unfoldr* rule where one side is less defined than the other. The

challenge Voigtländer sets to designers of new shortcut fusion systems is to prove total correctness (or at least to precisely state side conditions) for a realistic semantic model and not to rely on the naïve use of free theorems.

While we have used free theorems in the earlier sections of this chapter, this was in the context of System F where free theorems hold unconditionally. For the results about skipping streams in CPOs we make much weaker claims and do not rely on parametricity or free theorems.

If we were to try to prove a stronger result like $stream \circ unstream = id$ in CPOs, even for non-skipping streams then we would immediately run into problems. Firstly there is the issue that $stream\ (unstream\ \bot) \not\equiv \bot$ because the left hand side constructs a non-$\bot$ stream (that unfolds to $\bot$). Secondly, the 'proof' of $stream\ (unstream\ (Stream\ next\ s)) = Stream\ next\ s$ would have a step such as

$$\begin{aligned} & unbuild\ Stream\ (unfoldr\ next\ s) \\ = \quad & \{\ unbuild/unfoldr \text{ fusion}\ \} \\ & Stream\ next\ s \end{aligned}$$

While this is valid in System F, with fixpoints and polymorphic *seq* the side conditions are that *next* must be non-$\bot$, strict and total (Johann and Voigtländer, 2006, Laws 16 and 17).

These side conditions would have to be respected by all functions constructing streams. While these side conditions are likely rather easier to check than the abstraction property, they are more restrictive[4]. Voigtländer gives the example of *enumFromTo* which, recast as a stream function, is

$$\begin{aligned} & enumFromTo\ n\ m = Stream\ next\ n \\ & \quad \textbf{where} \\ & \qquad next\ i \mid i > m \quad\ = Nothing \\ & \qquad\qquad\ \mid otherwise = Just\ (i, i+1) \end{aligned}$$

The problem here is that $enumFromTo\ n\ \bot$ will construct a stream where $next\ n = \bot$, violating the requirement that *next* be total.

In a later paper Voigtländer (2008b) suggests variations on the definitions of the *foldr/build* and *unbuild/unfoldr* rules which require no side conditions – even in the presence of polymorphic *seq*. He suggests (Voigtländer, 2008b, Section 6) that this trick may be applicable to stream fusion since stream fusion is derived from *unbuild/unfoldr* fusion. Indeed it seems likely that the trick would be applicable to the case of non-skipping streams. That said, the primary innovation

[4]The astute reader will notice that we do not prove that *enumFromTo* works in our system, however Huffman (2009) does provide a proof.

of stream fusion over *unbuild*/*unfoldr* fusion is skipping streams. We cannot hope to preserve the property $stream \circ unstream = id$ in the case of skipping streams. The data abstraction approach does not appear to have problems with subtle strictness conditions; the side condition that stream functions preserve equivalence on streams is sufficient.

3.9 Proving stream library functions

Having established what we must prove about each library function, in this section we consider what proof techniques are appropriate and give proofs for a few key functions. Recall that for each stream function f_s and its supposed equivalent on lists f we must show that they are related by the abstraction relation

$$f_s \; \mathcal{A} \; f$$

which expands to an equational property per type. For example for homogeneous functions like *map* f or *filter* p we must show

$$unstream \circ f_s = f \circ unstream$$

Unfortunately we have rather a lot to prove. A full scale implementation of a stream fusion library contains many functions and we need a proof for each one. We are therefore interested in techniques that can reduce the overall proof effort. In particular we are interested in proof techniques that work reliably for whole classes of functions, even if this means that in particular cases we do not obtain the most insightful or elegant proofs. To put it another way, we are looking for a reliable handle-turning method.

The proofs developed in this section are not the first proofs of the corresponding theorems. Huffman (2009) has developed machine-checked proofs of the abstraction property for a number of functions. He uses the Isabelle/HOLCF framework which supports proofs in domain theory using fixpoint induction. The contribution of this section is to explore and explain the proof techniques. We use a 'by-hand' proof style rather than a style suited for an automated proof assistant.

We will concentrate solely on the case of streams and lists, rather than the general case for a arbitrary functor. We will use Haskell notation for data definitions, data constructors and case expressions.

Definition 3.9.1 (Skipping stream)**.**

$$\begin{aligned}&\textbf{data } \mathit{Stream}\ a = \exists s.\ \mathit{Stream}\ (s \to \mathit{Step}\ a\ s)\ s\\&\textbf{data } \mathit{Step}\ a\ s = \mathit{Done}\\&\qquad\qquad\quad \mid \mathit{Skip}\quad s\\&\qquad\qquad\quad \mid \mathit{Yield}\ a\ s\end{aligned}$$

3.9.1 Consideration of proof techniques

Before we plunge into attempting proofs for specific functions we should consider what techniques are available to us and what is likely to work. Starting with the example of *map*, we aim to show

$$\mathit{unstream} \circ \mathit{map}_s\, f = \mathit{map}\, f \circ \mathit{unstream}$$

That is we must prove equality between two functions. Each side of the equation is a function from streams to lists. Following the terminology of Gibbons and Hutton (2005), these functions are both *corecursive programs*. They define corecursive programs to be functions whose range is a type defined recursively as the greatest solution of some equation – or more simply types that are final co-algebras. We thus have available to us all the proof methods for corecursive programs (Gibbons and Hutton, 2005, Sections 3–6), including fixpoint induction, the approximation lemma, coinduction and unfold fusion.

In addition to functions like map_s that transform streams, the other common classes of functions are those that only consume streams or that only produce streams. The eponymous representatives of these three classes are as follows

$$\begin{aligned}\mathit{consume}_s &:: \mathit{Stream}\ A \to B\\ \mathit{consume} &:: [A] \to B\\ \mathit{transform}_s &:: \mathit{Stream}\ A \to \mathit{Stream}\ B\\ \mathit{transform} &:: [A] \to [B]\\ \mathit{produce}_s &:: A \to \mathit{Stream}\ B\\ \mathit{produce} &:: A \to [B]\end{aligned}$$

Assuming the types A and B do not contain streams then the abstraction property for each are

Property 3.9.2 (Abstraction property for stream consumers, transformers and producers)**.**

$$\begin{aligned} \mathit{consume}_s &= \mathit{consume} \circ \mathit{unstream} \\ \mathit{unstream} \circ \mathit{transform}_s &= \mathit{transform} \circ \mathit{unstream} \\ \mathit{unstream} \circ \mathit{produce}_s &= \mathit{produce} \end{aligned}$$

We can use corecursive proof techniques for the transformers and producers because the range type is a list. The consumers are somewhat harder because they are functions from a stream to some other type that is different for each function. We will consider transformers and producers first and return to producers later in this section.

We will evaluate our selected proof approaches using $\mathit{map}_s / \mathit{map}$ as an initial simple example. After *map* we will look at examples of increasing complexity:

- *filter* has essentially the same structure as *map* but makes non-trivial use of skip (Section 3.9.4);
- *append* and *zip* use non-trivial stream states and two input streams (Sections 3.9.5 and 3.9.6);
- *concatMap* has a complex stream state using a nested stream (Section 3.9.7);

Finally, having seen these various examples, in Section 3.9.8 we consider a general "handle turning" method for these kinds of proofs.

Note that we have not yet met the stream versions of *append*, *zip* or *concatMap*. This section is primarily about proofs and not about the design of stream functions. It may help to refer to Chapter 4 where the design of stream functions is considered in more detail, before returning to the later examples in this section.

One challenge we must deal with is the fact that *unstream* uses a recursion that is not well-founded. The standard approach for properties about non-well-founded recursion is fixpoint induction. We can reasonably guess that we will need to use fixpoint induction somewhere to deal with the aspect of *unstream* that uses non-well-founded recursion to consume unbounded sequences of skips. It is not obvious a priori that the entire proof need be by fixpoint induction.

How we choose to structure *unstream* is related to how we most naturally structure proofs about *unstream*. In the current context of the list functor, our previous general definition of *unstream* for skipping streams is the following

Definition 3.9.3 (Structured version of *unstream*)**.**

$$
\begin{aligned}
&\mathit{unstream} :: \mathit{Stream}\ a \to [a] \\
&\mathit{unstream}\ (\mathit{Stream}\ \mathit{next}\ s) = \mathit{unfoldr}\ (\mathit{force}\ f)\ s \\
&\mathit{unfoldr} :: (s \to \mathit{Maybe}\ (a, s)) \to s \to [a] \\
&\mathit{unfoldr}\ \mathit{next}\ s = \mathbf{case}\ \mathit{next}\ s\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad \mathit{Nothing} \quad \to [\,] \\
&\qquad\qquad\qquad\qquad \mathit{Just}\ (x, s') \to x : \mathit{unfoldr}\ \mathit{next}\ s' \\
&\mathit{force} :: (s \to \mathit{Step}\ a\ s) \to (s \to \mathit{Maybe}\ (a, s)) \\
&\mathit{force} = \mathit{fix}\ h \\
&\quad \mathbf{where} \\
&\qquad h\ g\ \mathit{next}\ s = \mathbf{case}\ \mathit{next}\ s\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad \mathit{Done} \quad\quad \to \mathit{Nothing} \\
&\qquad\qquad\qquad\qquad \mathit{Skip} \quad s' \to g\ \mathit{next}\ s' \\
&\qquad\qquad\qquad\qquad \mathit{Yield}\ x\ s' \to \mathit{Just}\ (x, s')
\end{aligned}
$$

This definition is structured with an *unfold* and a separate non-well-founded *force*. As an alternative definition of *unstream* we could combine the two recursions into a single non-well-founded recursion

Definition 3.9.4 (Single fixpoint version of *unstream*)**.**

$$
\begin{aligned}
&\mathit{unstream} :: \mathit{Stream}\ a \to [a] \\
&\mathit{unstream}\ (\mathit{Stream}\ \mathit{next}\ s) = \mathit{unfold}_{\mathit{Step}}\ \mathit{next}\ s \\
&\mathit{unfold}_{\mathit{Step}} :: (s \to \mathit{Step}\ a\ s) \to s \to [a] \\
&\mathit{unfold}_{\mathit{Step}} = \mathit{fix}\ h \\
&\quad \mathbf{where} \\
&\qquad h\ g\ \mathit{next}\ s = \mathbf{case}\ \mathit{next}\ s\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad \mathit{Done} \quad\quad \to [\,] \\
&\qquad\qquad\qquad\qquad \mathit{Skip} \quad s' \to \quad g\ \mathit{next}\ s' \\
&\qquad\qquad\qquad\qquad \mathit{Yield}\ x\ s' \to x : g\ \mathit{next}\ s'
\end{aligned}
$$

The former definition would likely be preferable if we believe we can structure the proof in two parts, one part using the more pleasant properties of *unfold* and a second part to deal with the non-well-founded *force*. On the other hand, if we do the entire proof using fixpoint induction then the latter definition is likely to be preferable since it uses *fix* once rather than twice. We will evaluate both approaches.

3.9.2 map_s/map by a single fixpoint induction

We will start with the second definition of *unstream* and use fixpoint induction (see Section 2.3 for a brief introduction).

We will use the standard definition of *map* on lists and the following as our definition of map_s

Definition 3.9.5 (map_s function)**.**

$$
\begin{aligned}
&map_s :: (a \to b) \to Stream\ a \to Stream\ b \\
&map_s\ f\ (Stream\ next\ s) = Stream\ (next_{map}\ f\ next)\ s \\
&next_{map}\ f\ next\ s = \textbf{case}\ next\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad Done \to Done \\
&\qquad\qquad\qquad\qquad Skip\ s' \to Skip\ s' \\
&\qquad\qquad\qquad\qquad Yield\ x\ s' \to Yield\ (f\ x)\ s'
\end{aligned}
$$

Theorem 3.9.6 (map_s/map abstraction property)**.**

$$unstream \circ map_s\ f = map\ f \circ unstream$$

Proof. We apply both sides to an arbitrary stream. We must account for the case of a $\perp$ stream. Fortunately this is a matter of observing that *unstream*, *map* and map_s are all strict. So we can move on to the primary case of an arbitrary non-$\perp$ stream *Stream next s*

$$unstream\ (map_s\ f\ (Stream\ next\ s)) = map\ f\ (unstream\ (Stream\ next\ s))$$

We can unfold the definition of *unstream* and map_s to get

$$unfold_{Step}\ (next_{map}\ f\ next)\ s = map\ f\ (unfold_{Step}\ next\ s)$$

This exposes $unfold_{Step}$ on both sides which is the key function defined by a fixpoint. Thus we can use this equation as the fixpoint induction goal.

We now need to define a property P and a function h such that $P\ (fix\ h)$ is the above induction goal

$$P\ (fix\ h) \iff unfold_{Step}\ (next_{map}\ f\ next)\ s = map\ f\ (unfold_{Step}\ next\ s)$$

We can define $P\ g$ by abstracting over $unfold_{Step}$

Definition 3.9.7.

$$P\ g \iff g\ (next_{map}\ f\ next)\ s = map\ f\ (g\ next\ s)$$

Now we can restate our goal as $P\ \mathit{unfold}_{\mathit{Step}}$. Since $\mathit{unfold}_{\mathit{Step}} = \mathit{fix}\ h$, $P\ \mathit{unfold}_{\mathit{Step}} = P\ (\mathit{fix}\ h)$ which is the required form for the conclusion of the fixpoint induction proof scheme.

Having set up the property P what remains is to actually do the fixpoint induction. The $P\ \bot$ case is straightforward, relying only on the fact that *map* is strict.

$$
\begin{array}{l}
P\ \bot \\
\Longleftrightarrow \\
\bot\ (\mathit{next}_{\mathit{map}}\ f\ \mathit{next})\ s = \mathit{map}\ f\ (\bot\ \mathit{next}\ s) \\
\Longleftrightarrow \\
\bot \qquad\qquad\qquad\qquad = \bot
\end{array}
$$

For the inductive case we start with $P\ (h\ g)$

$$
\begin{array}{ll}
 & P\ (h\ g) \\
\Longleftrightarrow & \\
 & h\ g\ (\mathit{next}_{\mathit{map}}\ f\ \mathit{next})\ s = \mathit{map}\ f\ (h\ g\ \mathit{next}\ s) \\
\Longleftrightarrow & \quad \{\ \text{unfold definition of } h \text{ and } \mathit{next}_{\mathit{map}}\ \} \\
 & \mathbf{case}\ (\mathbf{case}\ \mathit{next}\ s\ \mathbf{of} \\
 & \qquad \begin{array}{lll}
\mathit{Done} & \to & \mathit{Done} \\
\mathit{Skip}\ \ s' & \to & \mathit{Skip}\ \ s' \\
\mathit{Yield}\ x\ s' & \to & \mathit{Yield}\ (f\ x)\ s'
\end{array} \\
 & \mathbf{of} \\
 & \quad \begin{array}{lll}
\mathit{Done} & \to & [\,] \\
\mathit{Skip}\ \ s' & \to & g\ (\mathit{next}_{\mathit{map}}\ f\ \mathit{next})\ s' \\
\mathit{Yield}\ x\ s' & \to & x : g\ (\mathit{next}_{\mathit{map}}\ f\ \mathit{next})\ s'
\end{array} \\
 & = \\
 & \mathit{map}\ f\ (\mathbf{case}\ \mathit{next}\ s\ \mathbf{of} \\
 & \qquad \begin{array}{lll}
\mathit{Done} & \to & [\,] \\
\mathit{Skip}\ \ s' & \to & g\ \mathit{next}\ s' \\
\mathit{Yield}\ x\ s' & \to & x : g\ \mathit{next}\ s'
\end{array}
\end{array}
$$

We now split into the four possible cases of $\mathit{next}\ s$ ($\bot$, Done, $\mathit{Skip}\ s'$ or $\mathit{Yield}\ x\ s'$) and evaluate each side of the equation. The evaluation steps are straightforward but verbose so we omit the details and just tabulate the results:

$\mathit{next}\ s$	$h\ g\ (\mathit{next}_{\mathit{map}}\ f\ \mathit{next})\ s$	$\mathit{map}\ f\ (h\ g\ \mathit{next}\ s)$
$\bot$	$\bot$	$\bot$
Done	$[\,]$	$[\,]$
$\mathit{Skip}\ s'$	$g\ (\mathit{next}_{\mathit{map}}\ f\ \mathit{next})\ s'$	$\mathit{map}\ f\ (g\ \mathit{next}\ s')$
$\mathit{Yield}\ x\ s'$	$f\ x : g\ (\mathit{next}_{\mathit{map}}\ f\ \mathit{next})\ s'$	$f\ x : \mathit{map}\ f\ (g\ \mathit{next}\ s')$

The first two cases are simply equal and the latter two cases are equal with an application of the induction hypothesis $P\ g$. So we have that $P\ g \Rightarrow P\ (h\ g)$. $\square$

A very slight variation on the same proof is instead of splitting on cases and evaluating, to keep and transform the definitions as a whole. Starting again from

$$h\ g\ (next_{map}\ f\ next)\ s = map\ f\ (h\ g\ next\ s)$$

We unfold the left hand side to

$$\begin{array}{ll}
 & h\ g\ (next_{map}\ f\ next)\ s \\
= & \quad \{ \text{ unfold definition of } h \text{ and } next_{map} \ \} \\
 & \quad \textbf{case}\ (\ \textbf{case}\ next\ s\ \textbf{of} \\
 & \qquad\qquad Done \rightarrow Done \\
 & \qquad\qquad Skip\ \ s' \rightarrow Skip\ \ s' \\
 & \qquad\qquad Yield\ x\ s' \rightarrow Yield\ (f\ x)\ s') \\
 & \quad \textbf{of} \\
 & \qquad Done \rightarrow [\,] \\
 & \qquad Skip\ \ s' \rightarrow \ \ g\ (next_{map}\ f\ next)\ s' \\
 & \qquad Yield\ x\ s' \rightarrow x : \ g\ (next_{map}\ f\ next)\ s' \\
= & \quad \{ \text{ case-of-case transformation } \} \\
 & \quad \textbf{case}\ next\ s\ \textbf{of} \\
 & \qquad Done \rightarrow [\,] \\
 & \qquad Skip\ \ s' \rightarrow \ \ g\ (next_{map}\ f\ next)\ s' \\
 & \qquad Yield\ x\ s' \rightarrow f\ x : g\ (next_{map}\ f\ next)\ s'
\end{array}$$

And on the right hand side

$$\begin{array}{ll}
 & map\ f\ (h\ g\ next\ s) \\
= & \quad \{ \text{ unfold } h \ \} \\
 & \quad map\ f\ (\textbf{case}\ next\ s\ \textbf{of} \\
 & \qquad\qquad Done \rightarrow [\,] \\
 & \qquad\qquad Skip\ \ s' \rightarrow \ \ g\ next\ s' \\
 & \qquad\qquad Yield\ x\ s' \rightarrow x : g\ next\ s') \\
= & \quad \{ \text{ push function application through case } \} \\
 & \quad \textbf{case}\ next\ s\ \textbf{of} \\
 & \qquad Done \rightarrow map\ f\ [\,] \\
 & \qquad Skip\ \ s' \rightarrow map\ f\ (\ \ g\ next\ s') \\
 & \qquad Yield\ x\ s' \rightarrow map\ f\ (x : g\ next\ s') \\
= &
\end{array}$$

$$
\begin{array}{ll}
= & \{ \text{ unfold } map \} \\
& \textbf{case } next\ s \textbf{ of} \\
& \quad Done \quad \to [\,] \\
& \quad Skip \quad s' \to \quad map\ f\ (g\ next\ s') \\
& \quad Yield\ x\ s' \to f\ x : map\ f\ (g\ next\ s')
\end{array}
$$

By applying the induction hypothesis these two expressions are equal.

3.9.3 map_s/map by a structured approach

We will now try a proof using the more structured definition of *unstream*. The approach here is to 'pull' the map through the *force* and the *unfold*. Let us sketch the outline and then return to consider the two key steps.

Theorem 3.9.8 (map_s / map abstraction property)**.**

$$unstream \circ map_s\ f = map\ f \circ unstream$$

Proof. As before we can easily verify the case of a $\perp$ stream. The main case is

$$unstream\ (map_s\ f\ (Stream\ next\ s)) = map\ f\ (unstream\ (Stream\ next\ s))$$

We unfold definitions on both sides to get

$$unfold\ (force\ (next_{map}\ f\ next))\ s = map\ f\ (unfold\ (force\ next)\ s)$$

We now want to 'pull' the map through both *force* and *unfoldr*

$$
\begin{array}{ll}
& unfold\ (force\ (next_{map}\ f\ next))\ s \\
= & \quad \{ \text{ lemma } force/next_{map} \} \\
& unfold\ (next'_{map}\ f\ (force\ next))\ s \\
= & \quad \{ \text{ lemma } map/unfold \} \\
& map\ f\ (unfold\ (force\ next)\ s)
\end{array}
$$

□

For the $force/next_{map}$ lemma we need

$$force\ (next_{map}\ f\ next) = next'_{map}\ f\ (force\ next)$$

for some suitable definition of $next'_{map}$.

It is instructive to consider the types of $next_{map}$ and $next'_{map}$

$$\begin{aligned} &next_{map} :: (a \to b) \to (s \to Step\ a\ s) \qquad \to (s \to Step\ b\ s) \\ &next'_{map} :: (a \to b) \to (s \to Maybe\ (a, s)) \to (s \to Maybe\ (b, s)) \end{aligned}$$

The $next_{map}$ transforms stepper functions on skipping streams while $next'_{map}$ does the equivalent transformation for stepper functions on non-skipping streams. Its definition is straightforward

$$\begin{aligned} next'_{map}\ f\ next\ s = \mathbf{case}\ next\ s\ \mathbf{of} \\ \qquad Nothing \quad &\to Nothing \\ \qquad Just\ (x, s') &\to Just\ (f\ x, s') \end{aligned}$$

The obvious approach to proving this first lemma is by fixpoint induction. We have a single instance of *fix* in the *force* function.

Lemma 3.9.9 (*force*/$next_{map}$)**.**

$$force\ (next_{map}\ f\ next)\ s = next'_{map}\ f\ (force\ next)\ s$$

Proof. Recall the definition of *force*

$$\begin{aligned} &force = fix\ h \\ &\quad \mathbf{where} \\ &\qquad h\ g\ next\ s = \mathbf{case}\ next\ s\ \mathbf{of} \\ &\qquad\qquad Done \quad &\to Nothing \\ &\qquad\qquad Skip \quad s' &\to g\ next\ s' \\ &\qquad\qquad Yield\ x\ s' &\to Just\ (x, s') \end{aligned}$$

Define fixpoint induction property P by

$$P\ g \iff g\ (next_{map}\ f\ next)\ s = next'_{map}\ f\ (g\ next)\ s$$

Note that $P\ (fix\ h) = P\ force$ which is the statement of the lemma.

The $P\ \bot$ case is straightforward. For the $P\ (h\ g)$ case we have

$$\begin{aligned} &P\ (h\ g) \\ \iff &\\ &h\ g\ (next_{map}\ f\ next)\ s = next'_{map}\ f\ (h\ g\ next)\ s \\ \iff &\end{aligned}$$

$$
\begin{aligned}
&\iff \\
&\quad \textbf{case}\ (\textbf{case}\ \mathit{next}\ s\ \textbf{of} \\
&\qquad\qquad \mathit{Done} \to \mathit{Done} \\
&\qquad\qquad \mathit{Skip}\ s' \to \mathit{Skip}\ s' \\
&\qquad\qquad \mathit{Yield}\ x\ s' \to \mathit{Yield}\ (f\ x)\ s') \\
&\quad \textbf{of} \\
&\qquad \mathit{Done} \to \mathit{Nothing} \\
&\qquad \mathit{Skip}\ s' \to g\ (\mathit{next}_{map}\ f\ \mathit{next})\ s' \\
&\qquad \mathit{Yield}\ x\ s' \to \mathit{Just}\ (x, s') \\
&= \\
&\quad \textbf{case}\ (\textbf{case}\ \mathit{next}\ s\ \textbf{of} \\
&\qquad\qquad \mathit{Done} \to \mathit{Nothing} \\
&\qquad\qquad \mathit{Skip}\ s' \to g\ \mathit{next}\ s' \\
&\qquad\qquad \mathit{Yield}\ x\ s' \to \mathit{Just}\ (x, s')) \\
&\quad \textbf{of} \\
&\qquad \mathit{Nothing} \to \mathit{Nothing} \\
&\qquad \mathit{Just}\ (x, s') \to \mathit{Just}\ (f\ x, s')
\end{aligned}
$$

We split on the four possibilities for $\mathit{next}\ s$ ($\bot$, Done, $\mathit{Skip}\ s'$ or $\mathit{Yield}\ x\ s'$) and evaluate each side of the equation.

$\mathit{next}\ s$	$\mathit{force}\ (\mathit{next}_{map}\ f\ \mathit{next})\ s$	$\mathit{next}'_{map}\ f\ (\mathit{force}\ \mathit{next})\ s$
$\bot$	$\bot$	$\bot$
Done	$\mathit{Nothing}$	$\mathit{Nothing}$
$\mathit{Skip}\ s'$	$g\ (\mathit{next}_{map}\ f\ \mathit{next})\ s'$	{see below}
$\mathit{Yield}\ x\ s'$	$\mathit{Just}\ (f\ x, s')$	$\mathit{Just}\ (f\ x, s')$

For the right hand side of the $\mathit{Skip}\ s'$ case we get

$$
\begin{aligned}
&\quad \mathit{next}'_{map}\ f\ (\mathit{force}\ \mathit{next})\ s \\
&= \\
&\quad \textbf{case}\ g\ \mathit{next}\ s'\ \textbf{of} \\
&\qquad \mathit{Nothing} \to \mathit{Nothing} \\
&\qquad \mathit{Just}\ (x, s') \to \mathit{Just}\ (f\ x, s') \\
&= \quad \{\ \text{fold definition of } \mathit{next}'_{map}\ \} \\
&\quad \mathit{next}'_{map}\ f\ (g\ \mathit{next})\ s'
\end{aligned}
$$

which gives us an instance of the induction hypothesis. □

For the *map*/*unfold* lemma we need

$$
\begin{array}{l}
\quad \mathit{unfold}\ (\mathit{next}'_{\mathit{map}}\ f\ (\mathit{force}\ \mathit{next}))\ s \\
= \\
\quad \mathit{map}\ f\ (\mathit{unfold}\ (\mathit{force}\ \mathit{next})\ s)
\end{array}
$$

For this lemma we can use fixpoint induction, the approximation lemma or coinduction. The choice does not matter as each technique amounts to the same thing in the context of this lemma. We split on the three[5] possible cases for $\mathit{force}\ \mathit{next}\ s_0$: $\bot$, *Nothing* or $\mathit{Just}\ (x, s')$. In each case we unfold definitions on both sides of the equation.

$$\mathit{unfold}\ (\mathit{next}'_{\mathit{map}}\ f\ (\mathit{force}\ \mathit{next}))\ s = \mathit{map}\ f\ (\mathit{unfold}\ (\mathit{force}\ \mathit{next})\ s)$$

On the left hand side

$$
\begin{array}{l}
\quad \mathit{unfold}\ (\mathit{next}'_{\mathit{map}}\ f\ (\mathit{force}\ \mathit{next}))\ s \\
= \\
\quad \textbf{case}\ \mathit{next}'_{\mathit{map}}\ f\ (\mathit{force}\ \mathit{next})\ s\ \textbf{of} \\
\qquad \mathit{Nothing} \quad \to [\,] \\
\qquad \mathit{Just}\ (x, s') \to x : \mathit{unfold}\ (\mathit{next}'_{\mathit{map}}\ f\ (\mathit{force}\ \mathit{next}))\ s' \\
= \\
\quad \textbf{case}\ (\textbf{case}\ \mathit{force}\ \mathit{next}\ s\ \textbf{of} \\
\qquad\qquad \mathit{Nothing} \quad \to \mathit{Nothing} \\
\qquad\qquad \mathit{Just}\ (x, s') \to \mathit{Just}\ (f\ x, s')) \\
\quad \textbf{of} \\
\qquad \mathit{Nothing} \quad \to [\,] \\
\qquad \mathit{Just}\ (x, s') \to x : \mathit{unfold}\ (\mathit{next}'_{\mathit{map}}\ f\ (\mathit{force}\ \mathit{next}))\ s'
\end{array}
$$

On the right hand side

$$
\begin{array}{l}
\quad \mathit{map}\ f\ (\textbf{case}\ \mathit{force}\ \mathit{next}\ s\ \textbf{of} \\
\qquad\qquad \mathit{Nothing} \quad \to [\,] \\
\qquad\qquad \mathit{Just}\ (x, s') \to x : \mathit{unfold}\ (\mathit{force}\ \mathit{next})\ s') \\
= \\
\quad \textbf{case}\ \mathit{force}\ \mathit{next}\ s\ \textbf{of} \\
\qquad \mathit{Nothing} \quad \to [\,] \\
\qquad \mathit{Just}\ (x, s') \to f\ x : \mathit{map}\ f\ (\mathit{unfold}\ (\mathit{force}\ \mathit{next})\ s')
\end{array}
$$

[5]Technically we should also consider $\mathit{Just}\ \bot$ however we could easily eliminate this case by defining *unfold* using a specialised data type that does not include this extra unwanted value.

Evaluating both sides for the three cases of $\mathit{force\ next\ s}$ gives

$\mathit{force\ next\ s}$	$\mathit{unfold}\ (\mathit{next}'_{\mathit{map}}\ f\ (\mathit{force\ next}))\ s$
$\bot$	$\bot$
$\mathit{Nothing}$	$[\,]$
$\mathit{Just}\ (x, s')$	$f\ x : \mathit{unfold}\ (\mathit{next}'_{\mathit{map}}\ f\ (\mathit{force\ next}))\ s'$

$\mathit{force\ next\ s}$	$\mathit{map}\ f\ (\mathit{unfold}\ (\mathit{force\ next})\ s)$
$\bot$	$\bot$
$\mathit{Nothing}$	$[\,]$
$\mathit{Just}\ (x, s')$	$f\ x : \mathit{map}\ f\ (\mathit{unfold}\ (\mathit{force\ next})\ s')$

These observations amount to a 'casual' fixpoint induction proof where we do go to the effort of defining the predicate or the recursive function in terms of *fix*. The danger in this style is that we may accidentally attempt to apply the induction hypothesis before having unrolled the *fix* by one step.

We can turn these observations into a fixpoint induction proof by using the following P and an appropriate definition of *unfold* as an instance of *fix*.

$$P\ g \iff g\ (\mathit{next}'_{\mathit{map}}\ f\ (\mathit{force\ next}))\ s = \mathit{map}\ f\ (g\ (\mathit{force\ next})\ s)$$

In the $\mathit{Just}\ (x, s')$ case we get an instance of the induction hypothesis $P\ g$.

A proof using the approximation lemma would also be straightforward. In the $\mathit{Just}\ (x, s')$ case we would have

$$\begin{aligned} & \mathit{approx}\ (m+1)\ (f\ x : \mathit{unfold}\ (\mathit{next}'_{\mathit{map}}\ f\ (\mathit{force\ next}))\ s') \\ = \\ & \mathit{approx}\ (m+1)\ (f\ x : \mathit{map}\ f\ (\mathit{unfold}\ (\mathit{force\ next})\ s')) \end{aligned}$$

Unfolding $\mathit{approx}\ (m+1)$ gives us

$$\begin{aligned} & f\ x : \mathit{approx}\ m\ (\mathit{unfold}\ (\mathit{next}'_{\mathit{map}}\ f\ (\mathit{force\ next}))\ s') \\ = \\ & f\ x : \mathit{approx}\ m\ (\mathit{map}\ f\ (\mathit{unfold}\ (\mathit{force\ next})\ s')) \end{aligned}$$

Which is then true by the induction hypothesis that for all n

$$\begin{aligned} & \mathit{approx}\ n\ (\mathit{unfold}\ (\mathit{next}'_{\mathit{map}}\ f\ (\mathit{force\ next}))\ s') \\ = \\ & \mathit{approx}\ n\ (\mathit{map}\ f\ (\mathit{unfold}\ (\mathit{force\ next})\ s')) \end{aligned}$$

A proof using coinduction is perhaps the most direct. We would use the standard bisimulation property on lists and the relation

$$R = \{(unfold\ (next'_{map}\ f\ (force\ next))\ s, map\ f\ (unfold\ (force\ next)\ s)\}$$

Each of the three cases for *force next s* lets us exhibit that the lists are either both $\bot$ or $[\,]$ or start with equal elements. In the $Just\ (x, s')$ case we get directly that the tails are related by R.

In evaluating the two proof approaches we have considered it is fair to say that for this example the single fixpoint approach is somewhat less effort. It is disappointing that the approach that separates the well-founded and non-well founded recursions does not give rise to a noticeably shorter proof. Indeed the fixpoint proof about *force* is comparable in size to the whole proof in the other case. The more structured approach also requires we invent or derive a suitable definition of the stepper function on non-skipping lists. The only advantage of the more structured approach is that it provides us with an intuition about pulling the operation through the *force* fixpoint and the *unfold*.

3.9.4 $filter_s$/*filter*

For *filter* we will use fixpoint induction again but also give the highlights of the alternative more structured approach.

We will use the standard definition of *filter* and use the following as our definition of $filter_s$.

Definition 3.9.10 ($filter_s$ function)**.**

$$\begin{aligned}
&filter_s :: (a \to Bool) \to Stream\ a \to Stream\ a \\
&filter_s\ p\ (Stream\ next\ s) = Stream\ (next_{filter}\ p\ next)\ s \\
&next_{filter}\ p\ next\ s = \mathbf{case}\ next\ s\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\quad Done \to Done \\
&\qquad\qquad\qquad\qquad\qquad\quad Skip\ s' \to Skip\ s' \\
&\qquad\qquad\qquad\qquad Yield\ x\ s' \to \mathbf{if}\ p\ x\ \mathbf{then}\ Yield\ x\ s' \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ Skip\ s'
\end{aligned}$$

Theorem 3.9.11 ($filter_s$/*filter* abstraction property)**.**

$$unstream \circ filter_s\ p = filter\ p \circ unstream$$

Proof. The $\bot$ stream case is straightforward since $filter_s$ and $filter\ p$ are strict. The main case is

$$
\begin{aligned}
& unstream\ (filter_s\ p\ (Stream\ next\ s)) = filter\ p\ (unstream\ (Stream\ next\ s)) \\
= \quad & \{ \text{ unfold definition of } unstream \text{ and } filter_s \ \} \\
& unfold_{Step}\ (next_{filter}\ p\ next)\ s = filter\ p\ (unfold_{Step}\ next\ s)
\end{aligned}
$$

Our fixpoint induction property abstracts over $unfold_{Step}$

$$P\ g \iff g\ (next_{filter}\ p\ next)\ s = filter\ p\ (g\ next\ s)$$

The $P\ \bot$ case is straightforward as $filter\ p$ is strict. The $P\ (h\ g)$ case is

$$
\begin{aligned}
& P\ (h\ g) \\
\iff & \\
& h\ g\ (next_{filter}\ p\ next)\ s = filter\ p\ (h\ g\ next\ s) \\
\iff & \quad \{ \text{ unfold definition of } h \text{ and } next_{filter} \ \} \\
& \textbf{case}\ (\textbf{case}\ next\ s\ \textbf{of} \\
& \qquad \begin{array}{lll} Done & \to & Done \\ Skip\ \ s' & \to & Skip\ \ s' \\ Yield\ x\ s' & \to & \textbf{if}\ p\ x\ \textbf{then}\ Yield\ x\ s' \\ & & \textbf{else}\ Skip\ \ s' \end{array} \\
& \textbf{of} \\
& \quad \begin{array}{ll} Done & \to [\,] \\ Skip\ \ s' & \to g\ (next_{filter}\ p\ next)\ s' \\ Yield\ x\ s' & \to x : g\ (next_{filter}\ p\ next)\ s' \end{array} \\
= & \\
& filter\ p\ (\textbf{case}\ next\ s\ \textbf{of} \\
& \qquad \begin{array}{ll} Done & \to [\,] \\ Skip\ \ s' & \to g\ next\ s' \\ Yield\ x\ s' & \to x : g\ next\ s' \end{array})
\end{aligned}
$$

We split on the cases of $next\ s$ and additionally we split the $Yield\ x\ s'$ case into two, $p\ x$ and $not\ (p\ x)$

$next\ s$	$h\ g\ (next_{filter}\ p\ next)\ s$	$filter\ p\ (h\ g\ next\ s)$
$\bot$	$\bot$	$\bot$
$Done$	$[\,]$	$[\,]$
$Skip\ s'$	$g\ (next_{filter}\ p\ next)\ s'$	$filter\ p\ (g\ next\ s')$
$Yield\ x\ s' \wedge p\ x$	$x : g\ (next_{filter}\ p\ next)\ s'$	$x : filter\ p\ (g\ next\ s')$
$Yield\ x\ s' \wedge not\ (p\ x)$	$g\ (next_{filter}\ p\ next)\ s'$	$filter\ p\ (g\ next\ s')$

These are all equal, either directly or using the induction hypothesis $P\ g$

$$g\ (next_{filter}\ p\ next)\ s' = filter\ p\ (g\ next\ s')$$

□

For the more structured approach, the outline is as follows with the two important steps being those in which we pull the filter through the *force* and *unfold* functions

$$\begin{aligned}
& unstream\ (filter_s\ p\ (Stream\ next\ s)) \\
= \quad & \{\ \text{unfold definition of } filter_s\ \} \\
& unstream\ (Stream\ (mapFilter\ p\ next)\ s) \\
= \quad & \{\ \text{unfold definition of } unstream\ \} \\
& unfold\ (force\ (mapFilter\ p\ next))\ s \\
= \quad & \{\ \text{lemma } force/mapFilter\ \} \\
& unfold\ (mapFilter'\ f\ (force\ next))\ s \\
= \quad & \{\ \text{lemma } filter/unfold\ \} \\
& filter\ p\ (unfold\ (force\ next)\ s) \\
= \quad & \{\ \text{fold definition of } unstream\ \} \\
& filter\ p\ (unstream\ (Stream\ next\ s))
\end{aligned}$$

The main difference with the previous *map* example is that while $next'_{map}$ is not recursive, the $next'_{filter}$ is not only recursive but a non-well-founded recursion.

$$\begin{aligned}
next'_{filter}\ p\ next\ s = \mathbf{case}\ next\ s\ \mathbf{of} \\
\qquad Nothing \quad &\rightarrow \quad Nothing \\
\qquad Just\ (x, s') &\rightarrow \mathbf{if}\ p\ x\ \mathbf{then}\ Just\ (x, s') \\
&\qquad\qquad \mathbf{else}\ next'_{filter}\ p\ next\ s'
\end{aligned}$$

In turn this complicates the proof of both lemmas. The first lemma is

$$force\ (next_{filter}\ p\ next)\ s = next'_{filter}\ p\ (force\ next)\ s$$

The proof is complicated by the fact that there are two instances of *fix*, one in *force* and one in $next'_{filter}$. The second lemma is

$$unfold\ (next'_{filter}\ p\ (force\ next))\ s = filter\ p\ (unfold\ (force\ next)\ s)$$

Unlike in the *map* example, this second lemma also has to use fixpoint induction. The approximation lemma and coinduction methods are not available since we cannot demonstrate that both sides produce a list element. Again the presence of two instances of *fix* complicates the fixpoint induction proof.

For this *filter* example, the single fixpoint induction is clearly less effort.

3.9.5 $append_s$/$append$

The standard list *append* (or $+\!\!+$) function is

Definition 3.9.12 (*append* function)**.**

$$
\begin{aligned}
&append :: [a] \to [a] \to [a] \\
&append\ [\,]\ ys = ys \\
&append\ (x : xs)\ ys = x : append\ xs\ ys
\end{aligned}
$$

The corresponding stream version is as follows.

Definition 3.9.13 ($append_s$ function)**.**

$$
\begin{aligned}
&append_s :: Stream\ a \to Stream\ a \to Stream\ a \\
&append_s\ (Stream\ next_a\ s_a)\ (Stream\ next_b\ s_b) = \\
&\quad Stream\ (next_{append}\ next_a\ next_b)\ (Left\ (s_a, s_b)) \\
&next_{append}\ next_a\ next_b\ (Left\ (s_a, s_b)) = \\
&\quad \textbf{case}\ next_a\ s_a\ \textbf{of} \\
&\qquad Done \to Skip\ (Right\ s_b) \\
&\qquad Skip\ s'_a \to Skip\ (Left\ (s'_a, s_b)) \\
&\qquad Yield\ x\ s'_a \to Yield\ x\ (Left\ (s'_a, s_b)) \\
&next_{append}\ next_a\ next_b\ (Right\ s_b) = \\
&\quad \textbf{case}\ next_b\ s_b\ \textbf{of} \\
&\qquad Done \to Done \\
&\qquad Skip\ s'_b \to Skip\ (Right\ s'_b) \\
&\qquad Yield\ x\ s'_b \to Yield\ x\ (Right\ s'_b)
\end{aligned}
$$

There are a couple points worth noting. Firstly, $append_s$ takes two input streams and more interestingly it uses a non-trivial stream state. For $append_s$ the stream state is $Either\ (s_a, s_b)\ s_b$, where s_a and s_b are the types of the states of the two input streams. By contrast, map_s and $filter_s$ have the same type for the state of the output stream as for the input stream.

These points will have an impact on the structure of the proof. Instead of trying to guess the full structure of the proof from the outset we will explore using the approach from the previous examples and adjust as necessary. The hope is that the reasons for the failure of the initial approach will be as illuminating as any final proof.

For $append_s/append$, the abstraction property is

$$unstream\ (append_s\ a\ b) = append\ (unstream\ a)\ (unstream\ b)$$

Assuming streams a and b are not $\bot$ we can use use $a = Stream\ next_a\ s_a$ and similarly for b. We can then start unfolding definitions

$$\begin{aligned}
& unstream\ (append_s\ (Stream\ next_a\ s_a)\ (Stream\ next_b\ s_b)) \\
&= append\ (unstream\ sa)\ (unstream\ sb) \\
\iff & \quad \{ \text{ unfold definition of } unstream \text{ and } append_s \text{ on both sides } \} \\
& unfold_{Step}\ (next_{append}\ next_a\ next_b)\ (Left\ (s_a, s_b)) \\
&= append\ (unfold_{Step}\ next_a\ s_a)\ (unfold_{Step}\ next_b\ s_b)
\end{aligned}$$

There are two thing of note. Firstly we note the property mentions only the $Left\ (s_a, s_b)$ mode and not the $Right\ s_b$ mode. Inspection of the *Done* case in $next_{append}$ tells us that we will need to know something about the $Right\ s_b$ mode. Secondly, while it is clear that we will need to use $unfold_{Step}$ as the fixpoint, there are multiple occurrences and it is not immediately clear which ones we should abstract over when we pick our fixpoint induction hypothesis.

Initially we will define the fixpoint induction property by abstracting over all occurrences of $unfold_{Step}$ and see where that gets us

$$\begin{aligned}
P\ g \iff & g\ (next_{append}\ next_a\ next_b)\ (Left\ (s_a, s_b)) \\
= & \ append\ (g\ next_a\ s_a)\ (g\ next_b\ s_b)
\end{aligned}$$

The $P\ \bot$ case is straightforward as *append* is strict in its first argument. The $P\ (h\ g)$ case starts with

$$\begin{aligned}
P\ (h\ g) \iff & h\ g\ (next_{append}\ next_a\ next_b)\ (Left\ (s_a, s_b)) \\
= & \ append\ (h\ g\ next_a\ s_a)\ (h\ g\ next_b\ s_b)
\end{aligned}$$

Unfolding the left hand side gives

$$\begin{aligned}
& h\ g\ (next_{append}\ next_a\ next_b)\ (Left\ (s_a, s_b)) \\
= & \\
& \textbf{case}\ next_{append}\ next_a\ next_b\ (Left\ (s_a, s_b))\ \textbf{of} \\
& \quad Done \to [\,] \\
& \quad Skip\ s' \to g\ (next_{append}\ next_a\ next_b)\ s' \\
& \quad Yield\ x\ s' \to x : g\ (next_{append}\ next_a\ next_b)\ s' \\
= &
\end{aligned}$$

$$
\begin{array}{l}
= \\
\quad \textbf{case}\ (\textbf{case}\ next_a\ s_a\ \textbf{of} \\
\qquad\qquad Done \quad\quad \rightarrow Skip \quad\quad (Right\ s_b) \\
\qquad\qquad Skip \quad s'_a \rightarrow Skip \quad\quad (Left\ (s'_a, s_b)) \\
\qquad\qquad Yield\ x\ s'_a \rightarrow Yield\ x\ (Left\ (s'_a, s_b))) \\
\quad \textbf{of} \\
\qquad Done \quad\quad \rightarrow [\,] \\
\qquad Skip \quad s' \rightarrow \quad\quad g\ (next_{append}\ next_a\ next_b)\ s' \\
\qquad Yield\ x\ s' \rightarrow x : g\ (next_{append}\ next_a\ next_b)\ s'
\end{array}
$$

And unfolding the right hand side gives

$$
\begin{array}{l}
\quad append\ (h\ g\ next_a\ s_a)\ (h\ g\ next_b\ s_b) \\
= \\
\quad append\ (\textbf{case}\ next_a\ s_a\ \textbf{of} \\
\qquad\qquad\quad Done \quad\quad \rightarrow [\,] \\
\qquad\qquad\quad Skip \quad s'_a \rightarrow \quad\quad g\ next_a\ s'_a \\
\qquad\qquad\quad Yield\ x\ s'_a \rightarrow x : g\ next_a\ s'_a) \\
\qquad\qquad (h\ g\ next_b\ s_b) \\
= \\
\quad \textbf{case}\ next_a\ s_a\ \textbf{of} \\
\qquad Done \quad\quad \rightarrow \qquad\qquad\qquad\qquad\qquad h\ g\ next_b\ s_b \\
\qquad Skip \quad s'_a \rightarrow \quad\quad append\ (g\ next_a\ s'_a)\ (h\ g\ next_b\ s_b) \\
\qquad Yield\ x\ s'_a \rightarrow x : append\ (g\ next_a\ s'_a)\ (h\ g\ next_b\ s_b)
\end{array}
$$

We now evaluate both sides for the four $next_a\ s_a$ possibilities and tabulate the results for the left and right hand sides

$next_a\ s_a$	$h\ g\ (next_{append}\ next_a\ next_b)\ (Left\ (s_a, s_b))$
$\bot$	$\bot$
$Done$	$g\ (next_{append}\ next_a\ next_b)\ (Right\ s_b)$
$Skip\ s'$	$g\ (next_{append}\ next_a\ next_b)\ (Left\ (s'_a, s_b))$
$Yield\ a\ s'$	$x : g\ (next_{append}\ next_a\ next_b)\ (Left\ (s'_a, s_b))$

$next_a\ s_a$	$append\ (h\ g\ next_a\ s_a)\ (h\ g\ next_b\ s_b)$
$\bot$	$\bot$
$Done$	$unfold_{Step}\ next_b\ s_b$
$Skip\ s'$	$append\ (g\ next_a\ s'_a)\ (h\ g\ next_b\ s_b)$
$Yield\ a\ s'$	$x : append\ (g\ next_a\ s'_a)\ (h\ g\ next_b\ s_b)$

We want to show that the corresponding entries between the two tables are equal, possibly with the use of the induction hypothesis. Reading off the tables

however we can see two issues. The first is that in the *Done* case we are left with the sub-goal of

$$g\ (next_{append}\ next_a\ next_b)\ (Right\ s_b) = h\ g\ next_b\ s_b$$

We were expecting to have to prove something about the $Right\ s_b$ mode and this sub-goal gives shows us what the general structure of that should be. It is straightforward to prove a similar property, namely

$$unfold_{Step}\ (next_{append}\ next_a\ next_b)\ (Right\ s_b) = unfold_{Step}\ next_b\ s_b$$

which we can do using fixpoint induction with the induction hypothesis

$$P\ g \iff g\ (next_{append}\ next_a\ next_b)\ (Right\ s_b) = g\ next_b\ s_b$$

It is not immediately obvious however how to connect this lemma with our sub-goal, though we may expect that we may either be able to use a secondary fixpoint induction like the above or to strengthen the main induction hypothesis with a clause similar to the above property.

The second problem is that in all the non-$\perp$ cases we cannot actually apply our induction hypothesis because on the left hand side we have g but on the right we have $h\ g$.

Intuitively, the problem is that we are simultaneously unrolling several fixpoints at once but we actually want to unroll some and not others. The *append* function has two phases: the first where it steps through the first stream and a second phase where it steps through the second stream. With our initial choice of induction hypothesis we are unfolding the second stream by one step even though in the first phase we only want to unfold the first stream.

We could try reformulating our induction hypothesis so that we leave the second stream alone. That is we do not abstract over $unfold_{Step}$ on the right hand side in the second argument to *append*

$$\begin{aligned} P\ g &\iff g\ (next_{append}\ next_a\ next_b)\ (Left\ (s_a, s_b)) \\ &= append\ (g\ next_a\ s_a)\ (unfold_{Step}\ next_b\ s_b) \end{aligned}$$

While this will let us successfully apply the induction hypothesis in the *Skip* and *Yield* cases, for the *Done* case we are left with

$$g\ (next_{append}\ next_a\ next_b)\ (Right\ s_b) = unfold_{Step}\ next_b\ s_b$$

We cannot prove this as a general lemma since we know nothing about g. We cannot simply add this property into the induction hypothesis because it is not true for the $P\ \perp$ case; the left hand side is $\perp$ but the right is not.

Again, the intuition is that we are still trying to unroll two fixpoints simultaneously, but that the definition does not have enough symmetry for this to work out. Instead we should take an asymmetric approach and verify each fixpoint independently.

The key trick is that we can use anti-symmetry to allow an asymmetric approach. We can split an equality property $g = f$ into $g \sqsubseteq f \ \wedge \ f \sqsupseteq g$, where $\sqsubseteq$ is the normal domain ordering. In the context of fixpoint induction we formulate similar properties P and Q for the two direction and we prove each by a separate fixpoint induction. To take actually advantage of the asymmetry we define the properties P and Q such that we only abstract over a fixpoint on one side of the partial order relation.

With the above example sub-goal we can use

$$\begin{array}{lll} P\ g \iff g & & (next_{append}\ next_a\ next_b)\ (Right\ s_b) \sqsubseteq unfold_{Step}\ next_b\ s_b \\ Q\ g \iff unfold_{Step}\ (next_{append}\ next_a\ next_b)\ (Right\ s_b) \sqsupseteq g & & next_b\ s_b \end{array}$$

Then if we can complete both fixpoint inductions then the conclusions $P\ (fix\ h)$ (equivalently $P\ unfold_{Step}$) and $Q\ (fix\ h)$ (equivalently $Q\ unfold_{Step}$) give us

$$\begin{array}{rl} & unfold_{Step}\ (next_{append}\ next_a\ next_b)\ (Right\ s_b) \sqsubseteq unfold_{Step}\ next_b\ s_b \\ \wedge & unfold_{Step}\ (next_{append}\ next_a\ next_b)\ (Right\ s_b) \sqsupseteq unfold_{Step}\ next_b\ s_b \\ \iff & \{\ \text{anti-symmetry}\ \} \\ & unfold_{Step}\ (next_{append}\ next_a\ next_b)\ (Right\ s_b) = unfold_{Step}\ next_b\ s_b \end{array}$$

We can now tackle the proof by applying the anti-symmetry technique and by strengthening the induction hypothesis to cover both *Left* and *Right* modes.

Theorem 3.9.14 ($append_s$/*append* abstraction property)**.**

$$unstream\ (append_s\ a\ b) = append\ (unstream\ a)\ (unstream\ b)$$

Proof. For non-$\bot$ streams a and b the statement is equivalent to

$$\begin{array}{l} \quad unfold_{Step}\ (next_{append}\ next_a\ next_b)\ (Left\ (s_a, s_b)) \\ = append\ (unfold_{Step}\ next_a\ s_a)\ (unfold_{Step}\ next_b\ s_b) \end{array}$$

We will prove a stronger result, also covering the *Right* mode

$$\begin{array}{l} \quad unfold_{Step}\ (next_{append}\ next_a\ next_b)\ (Left\ (s_a, s_b)) \\ = append\ (unfold_{Step}\ next_a\ s_a)\ (unfold_{Step}\ next_b\ s_b) \\ \wedge \\ \quad unfold_{Step}\ (next_{append}\ next_a\ next_b)\ (Right\ s_b) \\ = unfold_{Step}\ next_b\ s_b \end{array}$$

We split each equation using anti-symmetry and we will prove the $\sqsubseteq$ and $\sqsupseteq$ directions independently by fixpoint induction. To form the induction properties we abstract over the $unfold_{Step}$ on the lesser side of the $\sqsubseteq$ partial order. Although logically the $\sqsubseteq$ and $\sqsupseteq$ directions are independent we can save space by considering both directions at once. To enable us to consider both directions with a single set of unfoldings we will define parametrised versions of the left and right hand sides of the two equations. Each is abstracted over the occurrence of the $unfold_{Step}$ we will do the fixpoint induction on.

$$
\begin{aligned}
LHS_a\ g &= g\ (next_{append}\ next_a\ next_b)\ (Left\ (s_a, s_b)) \\
RHS_a\ g &= append\ (g\ next_a\ s_a)\ (unfold_{Step}\ next_b\ s_b) \\
LHS_b\ g &= g\ (next_{append}\ next_a\ next_b)\ (Right\ s_b) \\
RHS_b\ g &= g\ next_b\ s_b
\end{aligned}
$$

We now need to construct the two induction hypotheses P and Q. Our first guess might be to do something fairly symmetric such as

$$
\begin{aligned}
P\ g &\iff P_a\ g \wedge P_b\ g \\
P_a\ g &\iff LHS_a\ g &&\sqsubseteq RHS_a\ unfold_{Step} \\
P_b\ g &\iff LHS_b\ g &&\sqsubseteq RHS_b\ unfold_{Step} \\
Q\ g\ g &\iff Q_a\ g \wedge Q_b\ g \\
Q_a\ g &\iff LHS_a\ unfold_{Step} \sqsupseteq RHS_a\ g \\
Q_b\ g &\iff LHS_b\ unfold_{Step} \sqsupseteq RHS_b\ g
\end{aligned}
$$

With these properties the corresponding induction steps would be

$$
\begin{aligned}
P_a\ (h\ g) &\Longleftarrow P_a\ g \wedge P_b\ g \\
P_b\ (h\ g) &\Longleftarrow P_a\ g \wedge P_b\ g \\
Q_a\ (h\ g) &\Longleftarrow Q_a\ g \wedge Q_b\ g \\
Q_b\ (h\ g) &\Longleftarrow Q_a\ g \wedge Q_b\ g
\end{aligned}
$$

However if we do this we will find that we get stuck in the *Done* case for $Q_a\ (h\ g)$. The *Done* case for $P_a\ (h\ g)$ is straightforward as it reduces to the obligation

$$g\ (next_{append}\ next_a\ next_b)\ (Right\ s_b) \sqsubseteq unfold_{Step}\ next_b\ s_b$$

which we can discharge as it is exactly the induction hypothesis $P_b\ g$. On the other hand, the *Done* case for $Q_a\ (h\ g)$ reduces to the obligation

$$unfold_{Step}\ (next_{append}\ next_a\ next_b)\ (Right\ s_b) \sqsupseteq unfold_{Step}\ next_b\ s_b$$

This is not an instance of $Q_b\ g$ because it is not g on the right hand side, rather it is $unfold_{Step}$. So while this property is not an instance of the induction hypothesis

we do still expect it to be true. Proving it however requires a separate fixpoint induction.

So the thing we should notice here is that we never needed $Q_b\ g$ to be in the induction hypothesis but that we do need $Q_b\ \mathit{unfold}_{Step}$ which itself can be proved using fixpoint induction with $Q_b\ g$ as the hypothesis. We should therefore adjust the structure of the proof and the induction hypotheses. We eliminate Q_b from the property Q. We will use Q_b as the property for an independent fixpoint induction

$$\begin{array}{lcl} Q\ g\ g & \iff & Q_a\ g \\ Q_a\ g & \iff & \mathit{LHS}_a\ \mathit{unfold}_{Step} \sqsupseteq \mathit{RHS}_a\ g \\ Q_b\ g & \iff & \mathit{LHS}_b\ \mathit{unfold}_{Step} \sqsupseteq \mathit{RHS}_b\ g \end{array}$$

The induction steps that we will aim to prove are now

$$\begin{array}{lcl} Q_a\ (h\ g) & \Longleftarrow & Q_a\ g \wedge Q_b\ \mathit{unfold}_{Step} \\ Q_b\ (h\ g) & \Longleftarrow & Q_b\ g \end{array}$$

So we use Q_b in a simple independent induction with a conclusion of $Q_b\ (\mathit{fix}\ h) = Q_b\ \mathit{unfold}_{Step}$. This is then used as an assumption in the induction step of Q_a. The P_a and P_b are as before. The overall theorem is stated as $P_a\ \mathit{unfold}_{Step} \wedge Q_a\ \mathit{unfold}_{Step}$.

The $\bot$ cases are straightforward. $P_a\ \bot$, $P_b\ \bot$ and $Q_b\ \bot$ are trivial while $Q_a\ \bot$ relies on the fact that *append* is strict in its first argument.

The structure of the remainder of the proof is to unfold the left and right hand sides of each clause and to do the usual evaluation for the cases of $\mathit{next}_a\ s_a$ and $\mathit{next}_b\ s_b$. Unfortunately a straightforward handle-turning approach generates many similar cases to check. We will take advantage of our factorised definition to give a somewhat less verbose proof. We will tabulate the results and use the tables to check all four cases of the four induction steps. We try to discharge each case either by simple syntactic equality or by applying an appropriate clause of the induction hypothesis.

We now unfold $\mathit{LHS}_a\ (h\ g)$ and $\mathit{RHS}_a\ (h\ g)$

$$\begin{array}{l} \quad \mathit{LHS}_a\ (h\ g) \\ = \\ \quad h\ g\ (\mathit{next}_{append}\ \mathit{next}_a\ \mathit{next}_b)\ (\mathit{Left}\ (s_a, s_b)) \\ = \end{array}$$

$$
\begin{aligned}
&= \\
&\quad \textbf{case } next_{append}\ next_a\ next_b\ (Left\ (s_a, s_b))\ \textbf{of} \\
&\qquad Done \quad\ \ \to [\,] \\
&\qquad Skip \quad s' \to \quad\ g\ (next_{append}\ next_a\ next_b)\ s' \\
&\qquad Yield\ x\ s' \to x : g\ (next_{append}\ next_a\ next_b)\ s' \\
&= \\
&\quad \textbf{case } (\textbf{case } next_a\ s_a\ \textbf{of} \\
&\qquad\qquad Done \quad\ \ \to Skip \quad (Right\ s_b) \\
&\qquad\qquad Skip \quad s'_a \to Skip \quad (Left\ (s'_a, s_b)) \\
&\qquad\qquad Yield\ x\ s'_a \to Yield\ x\ (Left\ (s'_a, s_b))) \\
&\quad \textbf{of} \\
&\qquad Done \quad\ \ \to [\,] \\
&\qquad Skip \quad s' \to \quad\ g\ (next_{append}\ next_a\ next_b)\ s' \\
&\qquad Yield\ x\ s' \to x : g\ (next_{append}\ next_a\ next_b)\ s' \\
&= \\
&\quad \textbf{case } next_a\ s_a\ \textbf{of} \\
&\qquad Done \quad\ \ \to \quad\ g\ (next_{append}\ next_a\ next_b)\ (Right\ s_b) \\
&\qquad Skip \quad s'_a \to \quad\ g\ (next_{append}\ next_a\ next_b)\ (Left\ (s'_a, s_b)) \\
&\qquad Yield\ x\ s'_a \to x : g\ (next_{append}\ next_a\ next_b)\ (Left\ (s'_a, s_b))
\end{aligned}
$$

Unfolding the right hand side gives

$$
\begin{aligned}
&\quad RHS_a\ (h\ g) \\
&= \\
&\quad append\ (h\ g\ next_a\ s_a)\ (unfold_{Step}\ next_b\ s_b) \\
&= \\
&\quad append\ (\textbf{case } next_a\ s_a\ \textbf{of} \\
&\qquad\qquad Done \quad\ \ \to [\,] \\
&\qquad\qquad Skip \quad s'_a \to \quad\ g\ next_a\ s'_a \\
&\qquad\qquad Yield\ x\ s'_a \to x : g\ next_a\ s'_a) \\
&\qquad (h\ g\ next_b\ s_b) \\
&= \\
&\quad \textbf{case } next_a\ s_a\ \textbf{of} \\
&\qquad Done \quad\ \ \to \qquad\qquad\qquad\qquad\qquad unfold_{Step}\ next_b\ s_b \\
&\qquad Skip \quad s'_a \to \quad\ append\ (g\ next_a\ s'_a)\ (unfold_{Step}\ next_b\ s_b) \\
&\qquad Yield\ x\ s'_a \to x : append\ (g\ next_a\ s'_a)\ (unfold_{Step}\ next_b\ s_b)
\end{aligned}
$$

Similarly we unfold $LHS_b\ (h\ g)$ and $RHS_b\ (h\ g)$

$$
\begin{array}{l}
\quad LHS_b\ (h\ g) \\
= \\
\quad h\ g\ (next_{append}\ next_a\ next_b)\ (Right\ s_b) \\
= \\
\quad \textbf{case}\ next_{append}\ next_a\ next_b\ (Right\ s_b)\ \textbf{of} \\
\qquad Done \quad \rightarrow [\,] \\
\qquad Skip \quad s' \rightarrow \quad g\ (next_{append}\ next_a\ next_b)\ s' \\
\qquad Yield\ x\ s' \rightarrow x : g\ (next_{append}\ next_a\ next_b)\ s' \\
= \\
\quad \textbf{case}\ (\textbf{case}\ next_b\ s_b\ \textbf{of} \\
\qquad\qquad Done \quad \rightarrow Done \\
\qquad\qquad Skip \quad s'_b \rightarrow Skip \quad (Right\ s'_b) \\
\qquad\qquad Yield\ x\ s'_b \rightarrow Yield\ x\ (Right\ s'_b)) \\
\quad \textbf{of} \\
\qquad Done \quad \rightarrow [\,] \\
\qquad Skip \quad s' \rightarrow \quad g\ (next_{append}\ next_a\ next_b)\ s' \\
\qquad Yield\ x\ s' \rightarrow x : g\ (next_{append}\ next_a\ next_b)\ s' \\
= \\
\quad \textbf{case}\ next_b\ s_b\ \textbf{of} \\
\qquad Done \quad \rightarrow [\,] \\
\qquad Skip \quad s'_b \rightarrow \quad g\ (next_{append}\ next_a\ next_b)\ (Right\ s'_b) \\
\qquad Yield\ x\ s'_b \rightarrow x : g\ (next_{append}\ next_a\ next_b)\ (Right\ s'_b)
\end{array}
$$

and right hand side

$$
\begin{array}{l}
\quad RHS_b\ (h\ g) \\
= \\
\quad h\ g\ next_b\ s_b \\
= \\
\quad \textbf{case}\ next_b\ s_b\ \textbf{of} \\
\qquad Done \quad \rightarrow [\,] \\
\qquad Skip \quad s'_b \rightarrow \quad g\ next_b\ s'_b \\
\qquad Yield\ x\ s'_b \rightarrow x : g\ next_b\ s'_b
\end{array}
$$

We evaluate both sides for the cases of $next_a\ s_a / next_b\ s_b$ and tabulate the results

$next_a\ s_a$	$LHS_a\ (h\ g)$
$\bot$	$\bot$
$Done$	$g\ (next_{append}\ next_a\ next_b)\ (Right\ s_b)$
$Skip\ s'_b$	$g\ (next_{append}\ next_a\ next_b)\ (Left\ (s'_a, s_b))$
$Yield\ x\ s'_b$	$x : g\ (next_{append}\ next_a\ next_b)\ (Left\ (s'_a, s_b))$

$next_a\ s_a$	$RHS_a\ (h\ g)$
$\bot$	$\bot$
$Done$	$unfold_{Step}\ next_b\ s_b$
$Skip\ s'_b$	$append\ (g\ next_a\ s'_a)\ (unfold_{Step}\ next_b\ s_b)$
$Yield\ x\ s'_b$	$x : append\ (g\ next_a\ s'_a)\ (unfold_{Step}\ next_b\ s_b)$

$next_b\ s_b$	$LHS_b\ (h\ g)$	$RHS_b\ (h\ g)$
$\bot$	$\bot$	$\bot$
$Done$	$[\,]$	$[\,]$
$Skip\ s'_b$	$g\ (next_{append}\ next_a\ next_b)\ (Right\ s'_b)$	$g\ next_b\ s'_b$
$Yield\ x\ s'_b$	$x : g\ (next_{append}\ next_a\ next_b)\ (Right\ s'_b)$	$x : g\ next_b\ s'_b$

Recall the definitions

$$
\begin{aligned}
P_a\ g &\iff LHS_a\ g &&\sqsubseteq RHS_a\ unfold_{Step}\\
P_b\ g &\iff LHS_b\ g &&\sqsubseteq RHS_b\ unfold_{Step}\\
Q_a\ g &\iff LHS_a\ unfold_{Step} &&\sqsupseteq RHS_a\ g\\
Q_b\ g &\iff LHS_b\ unfold_{Step} &&\sqsupseteq RHS_b\ g
\end{aligned}
$$

We now need to check the four induction steps

$$
\begin{aligned}
P_a\ (h\ g) &\Longleftarrow P_a\ g \wedge P_b\ g\\
P_b\ (h\ g) &\Longleftarrow P_b\ g\\
Q_a\ (h\ g) &\Longleftarrow Q_a\ g \wedge Q_b\ unfold_{Step}\\
Q_b\ (h\ g) &\Longleftarrow Q_b\ g
\end{aligned}
$$

We do this by using the tables to read off the terms of for each case of $next_a\ s_a / next_b\ s_b$ and applying the induction hypothesis. For the $LHS_x\ (h\ g)/RHS_x\ (h\ g)$ terms, the tables give them directly. For$LHS_x\ unfold_{Step}/RHS_x\ unfold_{Step}$ we note that $unfold_{Step} = fix\ h = h\ (fix\ h) = h\ unfold_{Step}$ and so we can obtain the desired terms by substituting $g := unfold_{Step}$ in $LHS_x\ (h\ g)$ and $RHS_x\ (h\ g)$.

All of the 16 cases are simple enough that they can be checked by inspection. The $\bot$ cases are all trivial, as is the *Done* case for P_b and Q_b. The *Skip* and *Yield* cases are straightforward as they are all instances of the corresponding induction hypothesis. For example in the *Skip* s'_b case of $P_b\ (h\ g)$ we have to check

$$LHS_b\ (h\ g) \sqsubseteq RHS_b\ unfold_{Step} \Longleftarrow LHS_b\ g \sqsubseteq RHS_b\ unfold_{Step}$$

We do so by inspecting the *Skip* s'_b row with $g := unfold_{Step}$ on the right hand side and noting that

$$g\ (next_{append}\ next_a\ next_b)\ (Right\ s'_b) \sqsubseteq unfold_{Step}\ next_b\ s'_b$$

is the induction hypothesis $P_b\ g$.

The interesting cases are $P_a\ (h\ g)$ and $Q_a\ (h\ g)$ in the *Done* case. For $P_a\ (h\ g)$ we have

$$g\ (next_{append}\ next_a\ next_b)\ (Right\ s_b) \sqsubseteq unfold_{Step}\ next_b\ s_b$$

which is the induction hypothesis for the *Right* mode, $P_b\ g$. For $Q_a\ (h\ g)$ we have

$$unfold_{Step}\ (next_{append}\ next_a\ next_b)\ (Right\ s_b) \sqsupseteq unfold_{Step}\ next_b\ s_b$$

This is where we need the conclusion of the separate Q_b fixpoint induction. This property is exactly $Q_b\ (fix\ h) = Q_b\ unfold_{Step}$. This completes the proof. □

What is interesting is that out of all the cases (all $4 \times 2 \times 2$ of them) the only ones that have any asymmetry are the *Done* cases for the *Left* mode. This is the stage in evaluation where the *append* function transitions from the first list to the second.

While this proof is rather tiresome due to the large number of cases, the structure and principles are fairly simple. Compared to the proofs for *map* and *filter* there are two additions to the technique. Firstly we must strengthen the induction property to cover all the stream modes (in this case *Left* and *Right*). Secondly to handle asymmetry in the induction property we can use anti-symmetry to partition the induction property and treat different fixpoints independently.

We see that the form of the function mirrors the form of the proof. Note in particular how the dependencies between induction properties in the induction steps mirrors the dependencies between stages of evaluation in the function.

3.9.6 zip_s/zip

The proof for the zip_s/zip example is substantially similar to that for $append_s$/$append$. In particular it requires strengthening the induction hypothesis and using anti-symmetry. We will find that one part of the proof needs an additional technique; rather than the standard proof scheme for fixpoint induction we have used so far, we will need to use fixpoint induction in two variables. We will see that the need for the additional technique is yet another instance of the form of the proof reflecting the form of the recursion in the function.

The standard list function zip is

Definition 3.9.15 (zip function)**.**

$$
\begin{array}{l}
zip :: [a] \to [b] \to [(a,b)] \\
zip\ [\,]\ \quad ys\ \quad = [\,] \\
zip\ xs\ \quad [\,]\ \quad = [\,] \\
zip\ (x : xs)\ (y : ys) = (x,y) : zip\ xs\ ys
\end{array}
$$

The corresponding stream version is as follows.

Definition 3.9.16 (zip_s function)**.**

$$
\begin{array}{l}
zip :: Stream\ a \to Stream\ b \to Stream\ (a,b) \\
zip\ (Stream\ next_a\ s_a)\ (Stream\ next_b\ s_b) = \\
\quad Stream\ (next_{zip}\ next_a\ next_b)\ (s_a, s_b, Nothing) \\
next_{zip}\ next_a\ next_b\ (s_a, s_b, Nothing) = \\
\quad \textbf{case}\ next_a\ s_a\ \textbf{of} \\
\qquad Done\ \quad \to Done \\
\qquad Skip\ \quad s'_a \to Skip\ (s'_a, s_b, Nothing) \\
\qquad Yield\ a\ s'_a \to Skip\ (s'_a, s_b, Just\ a) \\
next_{zip}\ next_a\ next_b\ (s'_a, s_b, Just\ a) = \\
\quad \textbf{case}\ next_b\ s_b\ \textbf{of} \\
\qquad Done\ \quad \to Done \\
\qquad Skip\ \quad s'_b \to Skip\ \quad\quad (s'_a, s'_b, Just\ a) \\
\qquad Yield\ b\ s'_b \to Yield\ (a,b)\ (s'_a, s'_b, Nothing)
\end{array}
$$

As was the case for $append_s$ we have two modes for $next_{zip}$. Unlike with $append_s$, the two modes here are mutually dependent. The initial mode is *Nothing* and when it has obtained an element from the first stream it moves into the *Just a* mode. After yielding a pair it moves back into the *Nothing* mode. The induction hypothesis will have to cover both modes.

Theorem 3.9.17 (zip_s/*zip* abstraction property)**.**

$$unstream\ (zip_s\ a\ b) = zip\ (unstream\ a)\ (unstream\ b)$$

Proof. We start by unfolding definitions. For non-$\bot$ streams a and b we have

$$\begin{aligned} &unstream\ (zip_s\ (Stream\ next_a\ s_a)\ (Stream\ next_b\ s_b)) \\ =\ &zip\ (unstream\ (Stream\ next_a\ s_a))\ (unstream\ (Stream\ next_b\ s_b)) \end{aligned}$$

and unfolding *unstream* and zip_s gives us

Property 3.9.18 (Mode *Nothing*)**.**

$$\begin{aligned} &unfold_{Step}\ (next_{zip}\ next_a\ next_b)\ (s_a, s_b, Nothing) \\ =\ &zip\ (unfold_{Step}\ next_a\ s_a)\ (unfold_{Step}\ next_b\ s_b) \end{aligned}$$

Note that the property mentions only the *Nothing* mode and not the *Just a* mode. In the *Yield* case we will need a property about the *Just a* mode. We can either guess what it should be or we do a trial derivation. It turns out to be

Property 3.9.19 (Mode *Just a*)**.**

$$\begin{aligned} &unfold_{Step}\ (next_{zip}\ next_a\ next_b)\ (s_a, s_b, Just\ a) \\ =\ &zip\ (a : unfold_{Step}\ next_a\ s_a)\ (unfold_{Step}\ next_b\ s_b) \end{aligned}$$

This is perhaps not surprising as it expresses precisely the purpose of the *Just a* mode; that it represents the stage of evaluation of *zip* where only an element from the first stream/list is known.

It is not immediately obvious how to formulate our fixpoint induction property, however it is clear that it must have clauses covering both properties above and that it will abstract over occurrences of $unfold_{Step}$. Following our experience with $append_s$ it is clear that we cannot have a single equational induction property that abstracts simultaneously over all occurrences of $unfold_{Step}$. This is because we do not want to unroll all the $unfold_{Step}$ occurrences simultaneously. The structure of the recursion in zip_s is that we alternate between modes. In the proof we must also alternate between unrolling the $unfold_{Step}$ occurrences corresponding to the first and second input streams.

Since we will treat different occurrences of $unfold_{Step}$ separately we will again have to use anti-symmetry. To help keep concise the statements of induction goals and induction properties we will define parametrised versions of the

left and right hand side expressions, parametrising independently over each occurrence of $unfold_{Step}$.

$$
\begin{aligned}
LHS_a\ g_1 &= g_1\ (next_{zip}\ next_a\ next_b)\ (s_a, s_b, Nothing) \\
RHS_a\ g_2\ g_3 &= zip\ (g_2\ next_a\ s_a)\ (g_3\ next_b\ s_b) \\
LHS_b\ g_1 &= g_1\ (next_{zip}\ next_a\ next_b)\ (s_a, s_b, Just\ a) \\
RHS_b\ g_2\ g_3 &= zip\ (a : g_2\ next_a\ s_a)\ (g_3\ next_b\ s_b)
\end{aligned}
$$

The LHS_a and RHS_a expressions correspond to the two sides of the property of the *Nothing* mode while the LHS_b and RHS_b expressions correspond to the *Just a* mode.

We can now state the goal that we seek to prove by fixpoint induction as

$$
\begin{aligned}
&LHS_a\ unfold_{Step} \sqsubseteq RHS_a\ unfold_{Step}\ unfold_{Step} \\
&LHS_a\ unfold_{Step} \sqsupseteq RHS_a\ unfold_{Step}\ unfold_{Step} \\
&LHS_b\ unfold_{Step} \sqsubseteq RHS_b\ unfold_{Step}\ unfold_{Step} \\
&LHS_b\ unfold_{Step} \sqsupseteq RHS_b\ unfold_{Step}\ unfold_{Step}
\end{aligned}
$$

We now seek to formulate the induction property or properties. For the $\sqsubseteq$ direction we can abstract over the $unfold_{Step}$ on the left giving us

$$
\begin{aligned}
P_a\ g &\iff LHS_a\ g \sqsubseteq RHS_a\ unfold_{Step}\ unfold_{Step} \\
P_b\ g &\iff LHS_b\ g \sqsubseteq RHS_b\ unfold_{Step}\ unfold_{Step} \\
P\ \ g &\iff P_a\ g \wedge P_b\ g
\end{aligned}
$$

For the $\sqsupseteq$ direction however it is less clear how to formulate an induction property since we have two occurrences of $unfold_{Step}$ and we know that we will need to alternate between unrolling the two occurrences rather than unrolling both simultaneously. We will do the induction for the $\sqsubseteq$ direction using the induction property P and return to the $\sqsupseteq$ direction later.

As usual, the $P\ \bot$ case is easy, relying on the fact that *zip* is strict in its first argument.

As before, the approach for the $P\ (h\ g)$ step is to unfold $LHS_a\ (h\ g)$, $RHS_a\ (h\ g)\ g'$, $LHS_b\ (h\ g)$, $RHS_b\ g'\ (h\ g)$ and to tabulate the resulting expressions for each case of $next_a\ s_a / next_b\ s_b$. We then compare table entries to check the two refinements hold, with suitable application of the induction hypothesis $P\ g$.

For the induction step we must verify $P_a\ (h\ g) \wedge P_b\ (h\ g) \Longleftarrow P_a\ g \wedge P_b\ g$. We will start with the P_a part $P_a\ (h\ g) \Longleftarrow P_a\ g \wedge P_b\ g$, so we unfold definitions for both $LHS_a\ (h\ g)$

$$
\begin{aligned}
& LHS_a\ (h\ g) \\
= \\
& h\ g\ (next_{zip}\ next_a\ next_b)\ (s_a, s_b, Nothing) \\
= \\
& \textbf{case}\ next_{zip}\ next_a\ next_b\ (s_a, s_b, Nothing)\ \textbf{of} \\
& \quad Done \rightarrow [\,] \\
& \quad Skip\ s' \rightarrow g\ (next_{zip}\ next_a\ next_b)\ s' \\
& \quad Yield\ x\ s' \rightarrow x : g\ (next_{zip}\ next_a\ next_b)\ s' \\
= \\
& \textbf{case}\ (\textbf{case}\ next_a\ s_a\ \textbf{of} \\
& \qquad Done \rightarrow Done \\
& \qquad Skip\ s'_a \rightarrow Skip\ (s'_a, s_b, Nothing) \\
& \qquad Yield\ a\ s'_a \rightarrow Skip\ (s'_a, s_b, Just\ a)) \\
& \textbf{of} \\
& \quad Done \rightarrow [\,] \\
& \quad Skip\ s' \rightarrow g\ (next_{zip}\ next_a\ next_b)\ s' \\
& \quad Yield\ x\ s' \rightarrow x : g\ (next_{zip}\ next_a\ next_b)\ s' \\
= \\
& \textbf{case}\ next_a\ s_a\ \textbf{of} \\
& \quad Done \rightarrow [\,] \\
& \quad Skip\ s'_a \rightarrow g\ (next_{zip}\ next_a\ next_b)\ (s'_a, s_b, Nothing) \\
& \quad Yield\ a\ s'_a \rightarrow g\ (next_{zip}\ next_a\ next_b)\ (s'_a, s_b, Just\ a)
\end{aligned}
$$

and $RHS_a\ (h\ g)\ g'$

$$
\begin{aligned}
& RHS_a\ (h\ g)\ g' \\
= \\
& zip\ (h\ g\ next_a\ s_a)\ (g'\ next_b\ s_b) \\
= \\
& zip\ (\textbf{case}\ next_a\ s_a\ \textbf{of} \\
& \qquad Done \rightarrow [\,] \\
& \qquad Skip\ s'_a \rightarrow g\ next_a\ s'_a \\
& \qquad Yield\ a\ s'_a \rightarrow a : g\ next_a\ s'_a) \\
& \quad (g'\ next_b\ s_b) \\
= \\
& \textbf{case}\ next_a\ s_a\ \textbf{of} \\
& \quad Done \rightarrow [\,] \\
& \quad Skip\ s'_a \rightarrow zip\ (g\ next_a\ s'_a)\ (g'\ next_b\ s_b) \\
& \quad Yield\ a\ s'_a \rightarrow zip\ (a : g\ next_a\ s'_a)\ (g'\ next_b\ s_b)
\end{aligned}
$$

Tabulating the results we have

$next_a\ s_a$	$LHS_a\ (h\ g)$	$RHS_a\ (h\ g)\ g'$
$\bot$	$\bot$	$\bot$
$Done$	$[\,]$	$[\,]$
$Skip\ s'_a$	$g\ (next_{zip}\ next_a\ next_b)\ (s'_a, s_b, Nothing)$	$zip\ (g\ next_a\ s'_a)\ (g'\ next_b\ s_b)$
$Yield\ a\ s'_a$	$g\ (next_{zip}\ next_a\ next_b)\ (s'_a, s_b, Just\ a)$	$zip\ (a : g\ next_a\ s'_a)\ (g'\ next_b\ s_b)$

Proving $P_a\ (h\ g) \Longleftarrow P_a\ g \wedge P_b\ g$ means checking that the following property holds, given the induction hypothesis

$$LHS_a\ (h\ g) \sqsubseteq RHS_a\ unfold_{Step}\ unfold_{Step}$$

We can check this property by comparing table entries in each of the four cases of $next_a\ s_a$. Recall that $unfold_{Step} = h\ unfold_{Step}$ and so we can use the tables to look up $RHS_a\ unfold_{Step}\ unfold_{Step}$.

The $\bot$ and $Done$ cases are trivial since they are equal. For $Skip$ it is a simple application of the $P_a\ g$ part of induction hypothesis. For $Yield$ we apply the $P_b\ g$ part of the induction hypothesis.

We now move on to the P_b part of the induction step: $P_b\ (h\ g) \Longleftarrow P_a\ g \wedge P_b\ g$. We unfold definitions for $LHS_b\ (h\ g)$

$$
\begin{array}{ll}
 & LHS_b\ (h\ g) \\
= & \\
 & h\ g\ (next_{zip}\ next_a\ next_b)\ (s_a, s_b, Just\ a) \\
= & \\
 & \textbf{case}\ next_{zip}\ next_a\ next_b\ (s_a, s_b, Just\ a)\ \textbf{of} \\
 & \quad Done \rightarrow [\,] \\
 & \quad Skip\ s' \rightarrow g\ (next_{zip}\ next_a\ next_b)\ s' \\
 & \quad Yield\ x\ s' \rightarrow x : g\ (next_{zip}\ next_a\ next_b)\ s' \\
= & \\
 & \textbf{case}\ (\textbf{case}\ next_b\ s_b\ \textbf{of} \\
 & \qquad Done \rightarrow Done \\
 & \qquad Skip\ s'_b \rightarrow Skip\ (s'_a, s'_b, Just\ a) \\
 & \qquad Yield\ b\ s'_b \rightarrow Yield\ (a, b)\ (s'_a, s'_b, Nothing))\ \textbf{of} \\
 & \quad Done \rightarrow [\,] \\
 & \quad Skip\ s' \rightarrow g\ (next_{zip}\ next_a\ next_b)\ s' \\
 & \quad Yield\ x\ s' \rightarrow x : g\ (next_{zip}\ next_a\ next_b)\ s' \\
= &
\end{array}
$$

$$
\begin{array}{l}
\quad = \\
\quad \textbf{case}\ next_b\ s_b\ \textbf{of} \\
\qquad Done \quad\ \ \rightarrow [\,] \\
\qquad Skip \quad s'_b \rightarrow \quad\ \ g\ (next_{zip}\ next_a\ next_b)\ (s'_a, s'_b, Just\ a) \\
\qquad Yield\ b\ s'_b \rightarrow (a, b) : g\ (next_{zip}\ next_a\ next_b)\ (s'_a, s'_b, Nothing)
\end{array}
$$

and $RHS_b\ g\ (h\ g')$

$$
\begin{array}{l}
\quad RHS_b\ g\ (h\ g') \\
= \\
\quad zip\ (a : g\ next_a\ s_a)\ (g'\ next_b\ s_b) \\
= \\
\quad zip\ (a : g\ next_a\ s_a) \\
\qquad (\textbf{case}\ next_b\ s_b\ \textbf{of} \\
\qquad\quad Done \quad\ \ \rightarrow [\,] \\
\qquad\quad Skip \quad s'_b \rightarrow \quad g'\ next_b\ s'_b \\
\qquad\quad Yield\ b\ s'_b \rightarrow b : g'\ next_b\ s'_b) \\
= \\
\quad \textbf{case}\ next_b\ s_b\ \textbf{of} \\
\qquad Done \quad\ \ \rightarrow zip\ (a : g\ next_a\ s_a)\ [\,] \\
\qquad Skip \quad s'_b \rightarrow zip\ (a : g\ next_a\ s_a)\ (\quad g'\ next_b\ s'_b) \\
\qquad Yield\ b\ s'_b \rightarrow zip\ (a : g\ next_a\ s_a)\ (b : g'\ next_b\ s'_b) \\
= \\
\quad \textbf{case}\ next_b\ s_b\ \textbf{of} \\
\qquad Done \quad\ \ \rightarrow [\,] \\
\qquad Skip \quad s'_b \rightarrow zip\ (a : g\ next_a\ s_a)\ (g'\ next_b\ s'_b) \\
\qquad Yield\ b\ s'_b \rightarrow (a, b) : zip\ (g\ next_a\ s_a)\ (g'\ next_b\ s'_b)
\end{array}
$$

Tabulating the results we have

$next_b\ s_b$	$LHS_b\ (h\ g)$
$\bot$	$\bot$
$Done$	$[\,]$
$Skip\ s'_b$	$g\ (next_{zip}\ next_a\ next_b)\ (s'_a, s'_b, Just\ a)$
$Yield\ a\ s'_b$	$(a, b) : g\ (next_{zip}\ next_a\ next_b)\ (s'_a, s'_b, Nothing)$

$next_b\ s_b$	$RHS_b\ g\ (h\ g')$
$\bot$	$\bot$
$Done$	$[\,]$
$Skip\ s'_b$	$zip\ (a : g\ next_a\ s_a)\ (g'\ next_b\ s'_b)$
$Yield\ a\ s'_b$	$(a, b) : zip\ (g\ next_a\ s_a)\ (g'\ next_b\ s'_b)$

We now use the table to check that the following property holds given the induction hypothesis

$$LHS_b\ (h\ g) \sqsubseteq RHS_b\ unfold_{Step}\ unfold_{Step}$$

The $\bot$ and *Done* cases are again trivial. For *Skip* we apply $P_b\ g$ and for *Yield* we apply $P_a\ g$.

So we have checked both $P_a\ (h\ g) \Longleftarrow P_a\ g \wedge P_b\ g$ and $P_b\ (h\ g) \Longleftarrow P_a\ g \wedge P_b\ g$. So we have $P\ (h\ g) \Longleftarrow P\ g$ and by fixpoint induction we have $P\ (fix\ h) = P\ unfold_{Step}$. Note that each part $P_a\ (h\ g)$ and $P_b\ (h\ g)$ required both $P_a\ g$ and $P_b\ g$. This is due to the fact that the *Nothing* and *Just a* modes depend on each other.

We now return to the $\sqsupseteq$ direction. Recall that our goal here is to show

$$LHS_a\ unfold_{Step} \sqsupseteq RHS_a\ unfold_{Step}\ unfold_{Step}$$
$$LHS_b\ unfold_{Step} \sqsupseteq RHS_b\ unfold_{Step}\ unfold_{Step}$$

We need to do this by fixpoint induction over the $unfold_{Step}$ fixpoints on the right hand side. The problem we face in formulating an induction property is that we will need to unroll the two $unfold_{Step}$ occurrences alternately rather than simultaneously. Instead of guessing up front we will look at the unfoldings of both sides and see what we can prove and work backwards to a supportable induction property.

We can reuse the parametrised tables of unfoldings we produced previously. Firstly for the LHS_a and RHS_a

$next_a\ s_a$	$LHS_a\ (h\ g)$
$\bot$	$\bot$
Done	$[\,]$
Skip s'_a	$g\ (next_{zip}\ next_a\ next_b)\ (s'_a, s_b, Nothing)$
Yield $a\ s'_a$	$g\ (next_{zip}\ next_a\ next_b)\ (s'_a, s_b, Just\ a)$

$next_a\ s_a$	$RHS_a\ (h\ g)\ g'$
$\bot$	$\bot$
Done	$[\,]$
Skip s'_a	$zip\ (g\ next_a\ s'_a)\ (g'\ next_b\ s_b)$
Yield $a\ s'_a$	$zip\ (a : g\ next_a\ s'_a)\ (g'\ next_b\ s_b)$

We instantiate the left hand side with $g := unfold_{Step}$ and on the right hand side use g and g' as is and then we consider the two interesting cases of *Skip* and *Yield*.

For *Skip* we have

$$\begin{aligned} & \mathit{unfold}_{\mathit{Step}}\ (\mathit{next}_{\mathit{zip}}\ \mathit{next}_a\ \mathit{next}_b)\ (s'_a, s_b, \mathit{Nothing}) \\ \sqsupseteq\ & \mathit{zip}\ (g\ \mathit{next}_a\ s'_a)\ (g'\ \mathit{next}_b\ s_b) \end{aligned}$$

The left hand side here is exactly $\mathit{LHS}_a\ \mathit{unfold}_{\mathit{Step}}$ while the right is exactly $\mathit{RHS}_a\ g\ g'$. For *Yield* we have

$$\begin{aligned} & \mathit{unfold}_{\mathit{Step}}\ (\mathit{next}_{\mathit{zip}}\ \mathit{next}_a\ \mathit{next}_b)\ (s'_a, s_b, \mathit{Just}\ a) \\ \sqsupseteq\ & \mathit{zip}\ (a : g\ \mathit{next}_a\ s'_a)\ (g'\ \mathit{next}_b\ s_b) \end{aligned}$$

The left hand side here is exactly $\mathit{LHS}_b\ \mathit{unfold}_{\mathit{Step}}$ while the right is exactly $\mathit{RHS}_b\ g\ g'$. Collecting these two we see that we can prove

$$\begin{aligned} \mathit{LHS}_a\ \mathit{unfold}_{\mathit{Step}} \sqsupseteq \mathit{RHS}_a\ (h\ g)\ g' \Longleftarrow\ & \mathit{LHS}_a\ \mathit{unfold}_{\mathit{Step}} \sqsupseteq \mathit{RHS}_a\ g\ g' \\ & \wedge\ \mathit{LHS}_b\ \mathit{unfold}_{\mathit{Step}} \sqsupseteq \mathit{RHS}_b\ g\ g' \end{aligned}$$

We can restate this more concisely if we define Q_a and Q_b by

$$\begin{aligned} Q_a\ g\ g' &\iff \mathit{LHS}_a\ \mathit{unfold}_{\mathit{Step}} \sqsupseteq \mathit{RHS}_a\ g\ g' \\ Q_b\ g\ g' &\iff \mathit{LHS}_b\ \mathit{unfold}_{\mathit{Step}} \sqsupseteq \mathit{RHS}_b\ g\ g' \end{aligned}$$

It then becomes

$$Q_a\ (h\ g)\ g' \Longleftarrow Q_a\ g\ g' \wedge Q_b\ g\ g'$$

We now turn to the second part and the tables for LHS_b and RHS_b

$\mathit{next}_b\ s_b$	$\mathit{LHS}_b\ (h\ g)$
$\bot$	$\bot$
Done	$[\,]$
Skip s'_b	$g\ (\mathit{next}_{\mathit{zip}}\ \mathit{next}_a\ \mathit{next}_b)\ (s'_a, s'_b, \mathit{Just}\ a)$
Yield $a\ s'_b$	$(a, b) : g\ (\mathit{next}_{\mathit{zip}}\ \mathit{next}_a\ \mathit{next}_b)\ (s'_a, s'_b, \mathit{Nothing})$

$\mathit{next}_b\ s_b$	$\mathit{RHS}_b\ g\ (h\ g')$
$\bot$	$\bot$
Done	$[\,]$
Skip s'_b	$\mathit{zip}\ (a : g\ \mathit{next}_a\ s_a)\ (g'\ \mathit{next}_b\ s'_b)$
Yield $a\ s'_b$	$(a, b) : \mathit{zip}\ (g\ \mathit{next}_a\ s_a)\ (g'\ \mathit{next}_b\ s'_b)$

Again we instantiate the left hand side with $g := \mathit{unfold}_{\mathit{Step}}$ and on the right hand side we use g and g' unchanged.

The *Skip* and *Yield* cases are then

$$\begin{aligned} & \mathit{unfold}_{\mathit{Step}}\ (\mathit{next}_{\mathit{zip}}\ \mathit{next}_a\ \mathit{next}_b)\ (s'_a, s'_b, \mathit{Just}\ a) \\ \sqsupseteq\ & \mathit{zip}\ (a : g\ \mathit{next}_a\ s_a)\ (g'\ \mathit{next}_b\ s'_b) \end{aligned}$$

and

$$\begin{aligned} & (a, b) : \mathit{unfold}_{\mathit{Step}}\ (\mathit{next}_{\mathit{zip}}\ \mathit{next}_a\ \mathit{next}_b)\ (s'_a, s'_b, \mathit{Nothing}) \\ \sqsupseteq\ & (a, b) : \mathit{zip}\ (g\ \mathit{next}_a\ s_a)\ (g'\ \mathit{next}_b\ s'_b) \end{aligned}$$

The first is an instance of $\mathit{LHS}_b\ \mathit{unfold}_{\mathit{Step}} \sqsupseteq \mathit{RHS}_b\ g\ g'$ while the second reduces to an instance of $\mathit{LHS}_a\ \mathit{unfold}_{\mathit{Step}} \sqsupseteq \mathit{RHS}_a\ g\ g'$. So we can summarise what we can show here as

$$Q_b\ (h\ g)\ g' \Longleftarrow Q_a\ g\ g' \wedge Q_b\ g\ g'$$

Putting the two parts together we are saying that we can show

$$\begin{aligned} Q_a\ (h\ g)\ g' \quad & \Longleftarrow Q_a\ g\ g' \wedge Q_b\ g\ g' \\ Q_b\ g \quad (h\ g') & \Longleftarrow Q_a\ g\ g' \wedge Q_b\ g\ g' \end{aligned}$$

That is, we are looking for an induction principle in two variables that lets us prove two related properties. In the induction step each property depends both on itself and on the other property. Fortunately there is a suitable proof scheme for fixpoint induction in two variables.

Property 3.9.20 (Fixpoint induction proof scheme in two variables)**.**

$$\begin{aligned} & A\ \bot\ y \ \wedge\ \ \forall x\ y.\ A\ x\ y \wedge B\ x\ y \Rightarrow A\ (f\ x)\ y \\ & B\ x\ \bot \ \wedge\ \ \forall x\ y.\ A\ x\ y \wedge B\ x\ y \Rightarrow B\ x\ (g\ y) \\ \Longrightarrow\ & \\ & A\ (\mathit{fix}\ f)\ (\mathit{fix}\ g)\ \wedge\ B\ (\mathit{fix}\ f)\ (\mathit{fix}\ g) \end{aligned}$$

We can instantiate this proof scheme with both f and g as h and using Q_a and Q_b. We have already checked the induction step above so we need only check that

$$Q_a\ \bot\ g' \ \wedge\ Q_b\ g\ \bot$$

The first reduces to checking that $\mathit{zip}\ \bot\ \mathit{ys} = \bot$ and the second reduces to checking $\mathit{zip}\ (x : \mathit{xs})\ \bot = \bot$. Both are true.

Thus we can conclude $Q_a\ \mathit{unfold}_{\mathit{Step}}\ \mathit{unfold}_{\mathit{Step}} \wedge Q_b\ \mathit{unfold}_{\mathit{Step}}\ \mathit{unfold}_{\mathit{Step}}$ which was our goal for the $\sqsupseteq$ direction. □

3.9.7 $concatMap_s$/$concatMap$

Definition 3.9.21 ($concatMap$ function)**.**

$$
\begin{aligned}
&concatMap\ f\ [\,] &&= [\,]\\
&concatMap\ f\ (a : as) &&= f\ a +\!\!+ concatMap\ f\ as
\end{aligned}
$$

Definition 3.9.22 ($concatMap_s$ function)**.**

$$
\begin{aligned}
&concatMap_s :: (a \to Stream\ b) \to Stream\ a \to Stream\ b\\
&concatMap_s\ f\ (Stream\ next_a\ s_a) =\\
&\quad Stream\ (next_{concatMap}\ next_a)\ (s_a, Nothing)\\
&next_{concatMap}\ next_a\ (s_a, Nothing) =\\
&\quad \textbf{case}\ next_a\ s_a\ \textbf{of}\\
&\qquad Done \to Done\\
&\qquad Skip\ s'_a \to Skip\ (s'_a, Nothing)\\
&\qquad Yield\ a\ s'_a \to Skip\ (s'_a, Just\ (f\ a))\\
&next_{concatMap}\ next_a\ (s_a, Just\ (Stream\ next_b\ s_b)) =\\
&\quad \textbf{case}\ next_b\ s_b\ \textbf{of}\\
&\qquad Done \to Skip\ (s_a, Nothing)\\
&\qquad Skip\ s'_b \to Skip\ (s_a, Just\ (Stream\ next_b\ s'_b))\\
&\qquad Yield\ b\ s'_b \to Yield\ b\ (s_a, Just\ (Stream\ next_b\ s'_b))
\end{aligned}
$$

The key feature is that we have an outer stream and an inner stream. There are two modes. In the *Nothing* mode it tries to obtain an element from the outer stream, giving it a new inner stream. In the *Just* mode it yields elements one by one from the inner stream. When the inner stream is exhausted it switches back to the first mode.

The proof for the *concatMap* example is similar to that for *append*, though there are superficial similarities with *zip*. Like zip_s, $concatMap_s$ has two modes which are mutually dependent. We will again have an induction hypothesis in two parts, one for each mode. We use anti-symmetry and prove each direction as a separate induction. Although we alternate between unrolling the inner and outer streams, we will not need (and not be able) to use the technique from the *zip* example of doing fixpoint induction in two variables. Instead, like $append_s$ we will use a secondary induction, though in this case it will be a nested induction, reflecting the nesting of the two streams in $concatMap_s$.

Theorem 3.9.23 ($concatMap_s$/$concatMap$ abstraction property)**.**

$$unstream \circ concatMap_s\ f_s = concatMap\ (unstream \circ f_s) \circ unstream$$

Proof. We start by applying each side to a non-$\bot$ stream $Stream\ next_a\ s_a$

$$\begin{aligned}
&\ unstream\ (concatMap_s\ f_s\ (Stream\ next_a\ s_a)) \\
=&\ concatMap\ (unstream \circ f_s)\ (unstream\ (Stream\ next_a\ s_a))
\end{aligned}$$

Unfolding definitions gives us

$$\begin{aligned}
&\ unfold_{Step}\ (next_{concatMap}\ f_s\ next_a)\ (s_a, Nothing) \\
=&\ concatMap\ (unstream \circ f_s)\ (unfold_{Step}\ next_a\ s_a)
\end{aligned}$$

This gives us our goal for the *Nothing* mode. We will need to find a similar property for the *Just* mode.

We define parametrised left and right hand sides for this first equation

$$\begin{aligned}
LHS_a\ g &= g\ (next_{concatMap}\ f_s\ next_a)\ (s_a, Nothing) \\
RHS_a\ g &= concatMap\ (unstream \circ f_s)\ (g\ next_a\ s_a)
\end{aligned}$$

The first part of the induction property will be

$$\begin{aligned}
P_a\ g &\iff LHS_a\ g &&\sqsubseteq RHS_a\ unfold_{Step} \\
Q_a\ g &\iff LHS_a\ unfold_{Step} &&\sqsupseteq RHS_a\ g
\end{aligned}$$

To find the second part of the induction property we will explore by unfolding $LHS_a\ (h\ g)$ and $RHS_a\ (h\ g)$

$$\begin{aligned}
&\ LHS_a\ (h\ g) \\
=& \\
&\ h\ g\ (next_{concatMap}\ f_s\ next_a)\ (s_a, Nothing) \\
=& \\
&\ \textbf{case}\ next_{concatMap}\ f_s\ next_a\ (s_a, Nothing)\ \textbf{of} \\
&\quad Done \to [\,] \\
&\quad Skip\quad s' \to\quad g\ (next_{concatMap}\ f_s\ next_a)\ s' \\
&\quad Yield\ x\ s' \to x : g\ (next_{concatMap}\ f_s\ next_a)\ s' \\
=& \\
&\ \textbf{case}\ (\textbf{case}\ next_a\ s_a\ \textbf{of} \\
&\qquad\quad Done \to Done \\
&\qquad\quad Skip\quad s'_a \to Skip\ (s'_a, Nothing) \\
&\qquad\quad Yield\ a\ s'_a \to Skip\ (s'_a, Just\ (f\ a)))\ \textbf{of} \\
&\quad Done \to [\,] \\
&\quad Skip\quad s' \to\quad g\ (next_{concatMap}\ f_s\ next_a)\ s' \\
&\quad Yield\ x\ s' \to x : g\ (next_{concatMap}\ f_s\ next_a)\ s' \\
=&
\end{aligned}$$

$$
\begin{aligned}
&= \\
&\quad \textbf{case } next_a\ s_a \textbf{ of} \\
&\qquad Done \quad\ \ \to [\,] \\
&\qquad Skip \quad s'_a \to g\ (next_{concatMap}\ f_s\ next_a)\ (s'_a, Nothing) \\
&\qquad Yield\ a\ s'_a \to g\ (next_{concatMap}\ f_s\ next_a)\ (s'_a, Just\ (f\ a))
\end{aligned}
$$

And for the right hand side

$$
\begin{aligned}
&\quad RHS_a\ (h\ g)\ g' \\
&= \\
&\quad concatMap\ (unstream \circ f_s)\ (h\ g\ next_a\ s_a) \\
&= \\
&\quad concatMap\ (unstream \circ f_s)\ (\textbf{case } next_a\ s_a \textbf{ of} \\
&\qquad\qquad\qquad\qquad Done \quad\ \ \to [\,] \\
&\qquad\qquad\qquad\qquad Skip \quad s'_a \to \quad g\ next_a\ s'_a \\
&\qquad\qquad\qquad\qquad Yield\ a\ s'_a \to a : g\ next_a\ s'_a) \\
&= \\
&\quad \textbf{case } next_a\ s_a \textbf{ of} \\
&\qquad Done \quad\ \ \to concatMap\ (unstream \circ f_s)\ [\,] \\
&\qquad Skip \quad s'_a \to concatMap\ (unstream \circ f_s)\ (\quad g\ next\ s'_a) \\
&\qquad Yield\ a\ s'_a \to concatMap\ (unstream \circ f_s)\ (a : g\ next\ s'_a) \\
&= \\
&\quad \textbf{case } next_a\ s_a \textbf{ of} \\
&\qquad Done \quad\ \ \to [\,] \\
&\qquad Skip \quad s'_a \to \qquad\qquad\qquad concatMap\ (unstream \circ f_s)\ (g\ next\ s'_a) \\
&\qquad Yield\ a\ s'_a \to unstream\ (f_s\ a) +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next\ s'_a)
\end{aligned}
$$

The *Yield* case is the interesting one. In the right hand side we have

$$unstream\ (f_s\ a) +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next\ s'_a)$$

If we use the assumption that $f\ a \not\equiv \bot$ then we can substitute $f\ a$ for an arbitrary stream $Stream\ next_b\ s_b$ and then unfold *unstream* to get

$$unfold_{Step}\ next_b\ s_b +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next\ s')$$

So in the *Yield* case showing $P_a\ (h\ g)$ and $Q_a\ (h\ g)$ amounts to showing both

$$
\begin{aligned}
&\quad g\ (next_{concatMap}\ f_s\ next_a)\ (s'_a, Just\ (Stream\ next_b\ s_b)) \\
&\sqsubseteq unfold_{Step}\ next_b\ s_b +\!\!+ concatMap\ (unstream \circ f_s)\ (unfold_{Step}\ next\ s'_a) \\
&\quad unfold_{Step}\ (next_{concatMap}\ f_s\ next_a)\ (s'_a, Just\ (Stream\ next_b\ s_b)) \\
&\sqsupseteq unfold_{Step}\ next_b\ s_b +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next\ s'_a)
\end{aligned}
$$

For the $\sqsubseteq$ direction we can reasonably expect that we can simply add this to the induction hypothesis as the property for the *Just* mode. For the $\sqsupseteq$ direction we might expect that we can reuse the technique from the proof for *zip* and parametrise over the $unfold_{Step}$ in the right hand side and then do an induction in two variables. This will not work however. We might imagine a property like

$$\begin{aligned} & unfold_{Step}\ (next_{concatMap}\ f_s\ next_a)\ (s'_a, Just\ (Stream\ next_b\ s_b)) \\ \sqsupseteq\ & g'\ next_b\ s_b +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next\ s'_a) \end{aligned}$$

However we would not be able to 'connect up' to such an induction property because we would need a g' in the first part of our induction property for the *Nothing* mode. The reason we cannot do that is because it would have to abstract over the $unfold_{Step}$ implicit in the $unstream \circ f_s$ part. We would then have a right hand side like

$$g'\ next_b\ s_b +\!\!+ concatMap\ (case\_Stream\ g' \circ f_s)\ (g\ next\ s'_a)$$

but in this mode we only want to unroll the $unfold_{Step}$ to the left of the $+\!\!+$, leaving the other instance alone.

Instead we simply prove the $\sqsupseteq$ direction by another nested fixpoint induction. It really is nested because it turns out that the *Done* case of the inner induction relies on the induction hypothesis of the outer induction. This mirrors the nested nature of the streams in the $concatMap_s$ function.

Let us now define parametrised left and right hand sides for the property for the *Just* mode

$$\begin{aligned} LHS_b\ g &= g\ (next_{concatMap}\ f_s\ next_a)\ (s'_a, Just\ (Stream\ next_b\ s_b)) \\ RHS_b\ g\ g' &= g'\ next_b\ s_b +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next\ s') \end{aligned}$$

The second part of the induction property is just

$$P_b\ g \iff LHS_b\ g \sqsubseteq RHS_b\ unfold_{Step}\ unfold_{Step}$$

We do not include the $\sqsupseteq$ direction in the induction hypothesis because we do not need it.

We now review the overall structure of the induction proof. We will have three separate inductions, one for the $\sqsubseteq$ direction, one for the $\sqsupseteq$ direction and a third nested induction used in the induction step of the induction for the $\sqsupseteq$ direction.

For the $\sqsubseteq$ direction we use the induction property

$$\begin{aligned} P\ g &\iff P_a\ g \wedge P_b\ g \\ P_a\ g &\iff LHS_a\ g \sqsubseteq RHS_a\ unfold_{Step} \\ P_b\ g &\iff LHS_b\ g \sqsubseteq RHS_b\ unfold_{Step}\ unfold_{Step} \end{aligned}$$

For the $\sqsupseteq$ direction we use the induction property

$$Q\ g \iff Q_a\ g$$
$$Q_a\ g \iff LHS_a\ unfold_{Step} \sqsupseteq RHS_a\ g$$

These are both simple inductions in one variable. The induction steps will require us to show

$$P_a\ (h\ g) \Longleftarrow P_a\ g \wedge P_b\ g$$
$$P_b\ (h\ g) \Longleftarrow P_a\ g \wedge P_b\ g$$
$$Q_a\ (h\ g) \Longleftarrow Q_a\ g$$

For the nested induction we will use

$$Q_b\ g\ g' \iff LHS_b\ unfold_{Step} \sqsupseteq RHS_b\ g\ g'$$

We do induction in the g' parameter, with g kept constant. The induction step will be

$$Q_b\ g\ (h\ g') \Longleftarrow Q_b\ g\ g' \wedge Q_a\ g$$

Note that it uses Q_a the induction hypothesis of the outer induction. The conclusion of the nested induction therefore also needs Q_a as an assumption

$$Q_b\ g\ unfold_{Step} \Longleftarrow Q_a\ g$$

We should first check the $\bot$ cases of each induction property. As usual they are all straightforward. $P_a\ \bot$ and $P_b\ \bot$ are trivial while $Q_a\ \bot$ relies on the strictness of *concatMap* in its list argument and and Q_b relies on $+\!\!+$ being strict in its first argument.

Before we can check all the induction steps, we need to unfold $LHS_b\ (h\ g)$ and $RHS_b\ g\ (h\ g')$

$$\begin{aligned}
&LHS_b\ (h\ g)\\
=\ &h\ g\ (next_{concatMap}\ f_s\ next)\ (s, Just\ (Stream\ next_b\ s_b))\\
=\ &\mathbf{case}\ next_{concatMap}\ f_s\ next\ (s, Just\ (Stream\ next_b\ s_b))\ \mathbf{of}\\
&\quad Done \to [\,]\\
&\quad Skip\ s' \to g\ (next_{concatMap}\ f_s\ next)\ s'\\
&\quad Yield\ x\ s' \to x : g\ (next_{concatMap}\ f_s\ next)\ s'\\
=\ &
\end{aligned}$$

$$
\begin{aligned}
&= \\
&\quad \textbf{case}\ (\textbf{case}\ next_b\ s_b\ \textbf{of} \\
&\qquad\qquad Done \to Skip\ (s_a, Nothing) \\
&\qquad\qquad Skip\ s'_b \to Skip\ (s_a, Just\ (Stream\ next_b\ s'_b)) \\
&\qquad\qquad Yield\ b\ s'_b \to Yield\ b\ (s_a, Just\ (Stream\ next_b\ s'_b))) \\
&\quad \textbf{of} \\
&\qquad Done \to [\,] \\
&\qquad Skip\ s' \to g\ (next_{concatMap}\ f_s\ next_a)\ s' \\
&\qquad Yield\ x\ s' \to x : g\ (next_{concatMap}\ f_s\ next_a)\ s' \\
&= \\
&\quad \textbf{case}\ next_b\ s_b\ \textbf{of} \\
&\qquad Done \to g\ (next_{concatMap}\ f_s\ next)\ (s, Nothing) \\
&\qquad Skip\ s'_b \to g\ (next_{concatMap}\ f_s\ next)\ (s, Just\ (Stream\ next_b\ s'_b)) \\
&\qquad Yield\ b\ s'_b \to b : g\ (next_{concatMap}\ f_s\ next)\ (s, Just\ (Stream\ next_b\ s'_b))
\end{aligned}
$$

And for the right hand side

$$
\begin{aligned}
&\quad RHS_b\ g\ (h\ g') \\
&= \\
&\quad h\ g'\ next_b\ s_b +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next_a\ s_a) \\
&= \\
&\quad (\textbf{case}\ next_b\ s_b \\
&\qquad Done \to [\,] \\
&\qquad Skip\ s'_b \to g'\ next_b\ s'_b \\
&\qquad Yield\ b\ s'_b \to b : g'\ next_b\ s'_b \\
&\quad) \\
&\qquad +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next_a\ s_a) \\
&= \\
&\quad \textbf{case}\ next_b\ s_b \\
&\qquad Done \to [\,] +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next_a\ s_a) \\
&\qquad Skip\ s'_b \to g'\ next_b\ s'_b +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next_a\ s_a) \\
&\qquad Yield\ b\ s'_b \to b : g'\ next_b\ s'_b +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next_a\ s_a) \\
&= \\
&\quad \textbf{case}\ next_b\ s_b \\
&\qquad Done \to concatMap\ (unstream \circ f_s)\ (g\ next_a\ s_a) \\
&\qquad Skip\ s'_b \to g'\ next_b\ s'_b +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next_a\ s_a) \\
&\qquad Yield\ b\ s'_b \to b : g'\ next_b\ s'_b +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next_a\ s_a)
\end{aligned}
$$

We now tabulate $LHS_a\ (h\ g)$ with $RHS_a\ (h\ g)$ and also $LHS_b\ (h\ g)$ with $RHS_b\ (h\ g)$

$next_a\ s_a$	$LHS_a\ (h\ g)$
$\bot$	$\bot$
$Done$	$[\,]$
$Skip\ s'_a$	$g\ (next_{concatMap}\ f_s\ next_a)\ (s'_a, Nothing)$
$Yield\ a\ s'_a$	$g\ (next_{concatMap}\ f_s\ next_a)\ (s'_a, Just\ (Stream\ next_b\ s_b))$

$next_a\ s_a$	$RHS_a\ (h\ g)$
$\bot$	$\bot$
$Done$	$[\,]$
$Skip\ s'_a$	$concatMap\ (unstream \circ f_s)\ (g\ next\ s'_a)$
$Yield\ a\ s'_a$	$unfold_{Step}\ next_b\ s_b +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next\ s'_a)$

$next_b\ s_b$	$LHS_b\ (h\ g)$
$\bot$	$\bot$
$Done$	$g\ (next_{concatMap}\ f_s\ next)\ (s, Nothing)$
$Skip\ s'_a$	$g\ (next_{concatMap}\ f_s\ next)\ (s, Just\ (Stream\ next_b\ s'_b))$
$Yield\ a\ s'_a$	$b : g\ (next_{concatMap}\ f_s\ next)\ (s, Just\ (Stream\ next_b\ s'_b))$

$next_b\ s_b$	$RHS_b\ g\ (h\ g')$
$\bot$	$\bot$
$Done$	$concatMap\ (unstream \circ f_s)\ (g\ next_a\ s_a)$
$Skip\ s'_a$	$g'\ next_b\ s'_b +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next_a\ s_a)$
$Yield\ a\ s'_a$	$b : g'\ next_b\ s'_b +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next_a\ s_a)$

Recall the definitions

$$
\begin{array}{lll}
P_a\ g & \iff LHS_a\ g & \sqsubseteq RHS_a\ unfold_{Step} \\
P_b\ g & \iff LHS_b\ g & \sqsubseteq RHS_b\ unfold_{Step}\ unfold_{Step} \\
Q_a\ g & \iff LHS_a\ unfold_{Step} & \sqsupseteq RHS_a\ g \\
Q_b\ g\ g' & \iff LHS_b\ unfold_{Step} & \sqsupseteq RHS_b\ g \qquad g'
\end{array}
$$

We now need to check the induction steps for each of the four cases by reading off the above tables

$$
\begin{array}{ll}
P_a\ (h\ g) & \Longleftarrow P_a\ g \wedge P_b\ g \\
P_b\ (h\ g) & \Longleftarrow P_a\ g \wedge P_b\ g \\
Q_a\ (h\ g) & \Longleftarrow Q_a\ g \wedge Q_b\ g\ unfold_{Step} \\
Q_b\ g\ (h\ g') & \Longleftarrow Q_a\ g \wedge Q_b\ g\ g'
\end{array}
$$

Most of the 16 cases are easy and uninteresting. The $\bot$ cases are trivial, as are the *Done* cases for P_a and Q_a. All the *Skip* cases are straightforward applications of the corresponding part of the induction hypothesis, as is the *Yield* case for P_b and Q_b. The interesting cases are the transitions between the modes, in particular the *Yield* case for $P_a\ (h\ g)\ /\ Q_a\ (h\ g)$ and the *Done* case for $P_b\ (h\ g)\ /\ Q_b\ g\ (h\ g')$.

In the *Yield* case for $P_a\ (h\ g)$ we have

$$\begin{aligned} & g\ (next_{concatMap}\ f_s\ next_a)\ (s'_a, Just\ (Stream\ next_b\ s_b)) \\ \sqsubseteq\ & unfold_{Step}\ next_b\ s_b +\!\!+ concatMap\ (unstream \circ f_s)\ (unfold_{Step}\ next\ s'_a) \end{aligned}$$

which is just $P_b\ g$. For $Q_a\ (h\ g)$ we have

$$\begin{aligned} & unfold_{Step}\ (next_{concatMap}\ f_s\ next_a)\ (s'_a, Just\ (Stream\ next_b\ s_b)) \\ \sqsupseteq\ & unfold_{Step}\ next_b\ s_b +\!\!+ concatMap\ (unstream \circ f_s)\ (g\ next\ s'_a) \end{aligned}$$

which is $Q_b\ g\ unfold_{Step}$. That is, it is the conclusion of the nested fixpoint induction with the Q_b property. As we stated previously, using this conclusion requires that we have the assumption $Q_a\ g$, which is just the induction hypothesis in this case.

In the *Done* case for $P_b\ (h\ g)$ we have

$$\begin{aligned} & g\ (next_{concatMap}\ f_s\ next)\ (s, Nothing) \\ \sqsubseteq\ & concatMap\ (unstream \circ f_s)\ (unfold_{Step}\ next_a\ s_a) \end{aligned}$$

which is $Q_a\ g$ which is the induction hypothesis of the outer induction. In the *Done* case for $Q_b\ g\ (h\ g')$ we have

$$\begin{aligned} & unfold_{Step}\ (next_{concatMap}\ f_s\ next)\ (s, Nothing) \\ \sqsupseteq\ & concatMap\ (unstream \circ f_s)\ (g\ next_a\ s_a) \end{aligned}$$

which is $Q_a\ g$. □

3.9.8 A handle-turning method

Having looked at three non-trivial examples we are interested in whether there is a general method or general heuristics for finding proofs for these kinds of problems. These proofs all end up with a few inductions using a few induction properties. Once we have arrived at the right induction properties and induction schemes then proving all the induction steps is fairly straightforward. The question is whether there is a reliable method to find what the appropriate

induction properties and induction schemes are, without too much guesswork or trial and error experimentation.

Comparing the three examples, *append*, *zip* and *concatMap*, they have many superficial similarities. They all take two input streams and the output stream has a stepper function with two modes. The proofs are also superficially similar: the induction properties have a clause for each of the two modes and we do independent inductions for the $\sqsubseteq$ and $\sqsupseteq$ directions. There are also significant differences. Given our aim of finding a handle-turning method it is initially somewhat worrying that each proof uses different induction patterns. The proof for *append* uses two separate inductions with one relying on the other. The *zip* proof uses a fixpoint induction in two variables and the *concatMap* proof uses two inductions, one nested in the other.

One way to see a greater degree of commonality is to look at the induction steps in each proof. The induction steps can be viewed as dependencies between the induction properties. Below are the induction steps for the three examples. While the *LHS* and *RHS* terms are of course different for each example, the P_x and Q_x properties are more or less the same.

So we have *LHS* and *RHS* for each clause (*a* and *b* here) and each term has one or more parameters.

$$
\begin{aligned}
LHS_a\ g &= \ldots \\
RHS_a\ g\ g' &= \ldots \\
LHS_b\ g &= \ldots \\
RHS_b\ g\ g' &= \ldots
\end{aligned}
$$

This example happens to be for *zip*. Then we have properties for each clause, a P_x property in the $\sqsubseteq$ direction and a similar Q_x property in the $\sqsupseteq$ direction.

$$
\begin{aligned}
P_a\ g &\iff LHS_a\ g \sqsubseteq RHS_a\ unfold_{Step}\ unfold_{Step} \\
P_b\ g &\iff LHS_b\ g \sqsubseteq RHS_b\ unfold_{Step}\ unfold_{Step} \\
Q_a\ g\ g' &\iff LHS_a\ unfold_{Step} \sqsupseteq RHS_a\ g\ g' \\
Q_b\ g\ g' &\iff LHS_b\ unfold_{Step} \sqsupseteq RHS_b\ g\ g'
\end{aligned}
$$

Again this example is for *zip* but the others differed only in the number of parameters for the *LHS*/*RHS* terms. In particular the parameters for P_x and Q_x are always on the lesser side of the $\sqsubseteq/\sqsupseteq$ ordering.

Then we have the induction steps expressed in terms of the P_x and Q_x properties. For $append_s/append$ we had

$$\begin{aligned}
P_a\ (h\ g) &\Longleftarrow P_a\ g\ \wedge P_b\ g\\
P_b\ (h\ g) &\Longleftarrow \quad\quad P_b\ g\\
Q_a\ (h\ g) &\Longleftarrow Q_a\ g \wedge Q_b\ unfold_{Step}\\
Q_b\ (h\ g) &\Longleftarrow \quad\quad Q_b\ g
\end{aligned}$$

For zip_s/zip we had

$$\begin{aligned}
P_a\ (h\ g) \quad\quad &\Longleftarrow P_a\ g \quad\ \wedge P_b\ g\\
P_b\ (h\ g) \quad\quad &\Longleftarrow P_a\ g \quad\ \wedge P_b\ g\\
Q_a\ (h\ g)\ g' \quad &\Longleftarrow Q_a\ g\ g' \wedge Q_b\ g\ g'\\
Q_b\ g \quad\ (h\ g') &\Longleftarrow Q_a\ g\ g' \wedge Q_b\ g\ g'
\end{aligned}$$

For $concatMap_s/concatMap$ we had

$$\begin{aligned}
P_a\ (h\ g) \quad &\Longleftarrow P_a\ g\ \wedge P_b\ g\\
P_b\ (h\ g) \quad &\Longleftarrow P_a\ g\ \wedge P_b\ g\\
Q_a\ (h\ g) \quad &\Longleftarrow Q_a\ g \wedge Q_b\ g\ unfold_{Step}\\
Q_b\ g\ (h\ g') &\Longleftarrow Q_a\ g \wedge Q_b\ g\ g'
\end{aligned}$$

This is where we see the interesting differences, especially in the Q_x properties.

The key idea is that we should derive the induction scheme from the pattern of dependencies between the properties. In the *zip* example the pattern of dependencies between Q_a and Q_b tells us that we should use fixpoint induction in two variables. In the *concatMap* example the dependencies between Q_a and Q_b points us in the direction of using two nested fixpoint inductions.

So the idea is that it is rather easier to discover the dependencies between the P_x and Q_x properties than it is to guess upfront what the appropriate fixpoint induction scheme should be (which would in turn determine what induction step properties we would need to show).

Looking at it this way should lift our worries about the fact that we end up with such a diversity of patterns of fixpoint induction. The pattern is a consequence of the pattern of dependencies between the various induction properties which in turn is a consequence of the pattern of state transitions in the stream function. For example if we tried to prove the abstraction property for $zip5_s/zip5$ then we would expect to en up with five P_x and Q_x properties and we would expect the Q_x properties to depend on each other in a cyclic pattern. We would then use fixpoint induction in five variables.

Of course there is also the matter of how to discover the right induction properties. The method is as follows. We start from the original statement of the theorem. We unfold definitions to exposes the key fixpoints and this gives us our first equation $LHS_a = RHS_a$. Unrolling the key fixpoints may lead to a number of other equations, one for each mode of the output stream.

$$LHS_a = RHS_a, LHS_b = RHS_b, \ldots$$

From these equations we extract the left and right hand side terms and we parametrise them over the fixpoint functions that we had to unroll at any stage

$$LHS_a\ g_1 \ldots g_n = RHS_a\ g_1 \ldots g_m, \ldots$$

Next we define P_x and Q_x properties by splitting each equation using anti-symmetry. The P_x properties are parametrised by the same parameters as the LHS_a and respectively for the Q_x.

$$\begin{aligned} P_a\ g_1 \ldots g_n &\iff LHS_a\ g_1 \ldots g_n \sqsubseteq RHS_a\ (fix\ h) \ldots (fix\ h) \\ Q_a\ g_1 \ldots g_m &\iff LHS_a\ (fix\ h) \ldots (fix\ h) \sqsupseteq RHS_a\ g_1 \ldots g_m \end{aligned}$$

We next want to look at P_x/Q_x with some parameters set to $h\ g_i$ and others remaining as g_j. The ones where we use $h\ g_i$ are those that stand for fixpoints that we unrolled at the earlier stage when finding all the equations. For example with *concatMap* for Q_b we gave two parameters but only one of them gets unrolled when unfolding definitions in $LHS_b = RHS_b$.

Then we unfold each P_x/Q_x with the appropriate parameters set to $h\ g_i$ and do the appropriate case analysis. Now we look at which other P_x/Q_x properties are needed to prove each case. This gives us a pattern of induction steps like in the three examples above.

As a hypothetical example, consider $zip5_s/zip5$. The theorem is

$$unstream\ (zip5_s\ a\ b\ c\ d\ e) = zip5\ (unstream\ a) \ldots (unstream\ e)$$

We unfold this to expose the $unfold_{Step}$ fixpoint, giving us our first equation.

$$\begin{aligned} &unfold_{Step}\ (next_{zip5}\ next_a \ldots next_e)\ (Mode1\ s_a \ldots s_e) \\ &= zip5\ (unfold_{Step}\ next_a\ s_a) \ldots (unfold_{Step}\ next_e\ s_e) \end{aligned}$$

Now we unroll the first $unfold_{Step}$ and look at the four cases of $next_a\ s_a$. The $\bot$ and $[\,]$ cases do not give us any new non-trivial equations. The *Skip* case will of course give us another instance of the same equation.

Only the *Yield* case will give us a new equation

$$\begin{aligned}&\mathit{unfold}_{\mathit{Step}}\ (\mathit{next}_{\mathit{zip5}}\ \mathit{next}_a \ldots \mathit{next}_e)\ (\mathit{Mode2}\ s'_a \ldots s_e\ a)\\ =\ &\mathit{zip5}\ (\mathit{unfold}_{\mathit{Step}}\ \mathit{next}_a\ s'_a) \ldots (\mathit{unfold}_{\mathit{Step}}\ \mathit{next}_e\ s_e)\end{aligned}$$

We would continue this process of unrolling until we get no new equations. In our *zip5* example the final equation would be

$$\begin{aligned}&\mathit{unfold}_{\mathit{Step}}\ (\mathit{next}_{\mathit{zip5}}\ \mathit{next}_a \ldots \mathit{next}_e)\ (\mathit{Mode5}\ s'_a \ldots s_e\ a\ b\ c\ d)\\ =\ &\mathit{zip5}\ (\mathit{unfold}_{\mathit{Step}}\ \mathit{next}_a\ s'_a) \ldots (\mathit{unfold}_{\mathit{Step}}\ \mathit{next}_e\ s_e)\end{aligned}$$

which in the *Yield* case would go back to *Mode1* and be an instance of the first equation. So we end up with five equations.

The next step is to define LHS_x/RHS_x terms parametrised over the fixpoint functions that we had to unroll. Note that in the five right hand side terms we only unrolled one $\mathit{unfold}_{\mathit{Step}}$ per equation but overall we ended up unrolling them all. So we parametrise RHS_x with $g_1 \ldots g_5$ over all five occurrences.

Next we define the P_x and Q_x properties

$$\begin{aligned}&P_a\ g = LHS_a\ g \sqsubseteq RHS_a\ (\mathit{fix}\ h) \ldots (\mathit{fix}\ h)\\ &P_b\ g = LHS_b\ g \sqsubseteq RHS_b\ (\mathit{fix}\ h) \ldots (\mathit{fix}\ h)\\ &\ldots\\ &Q_a\ g_1 \ldots g_5 = LHS_a\ (\mathit{fix}\ h) \sqsupseteq RHS_a\ g_1 \ldots g_5\\ &Q_b\ g_1 \ldots g_5 = LHS_b\ (\mathit{fix}\ h) \sqsupseteq RHS_b\ g_1 \ldots g_5\\ &\ldots\end{aligned}$$

Next we need to look at $P_a\ (h\ g), P_b\ (h\ g), \ldots$ and similarly for $Q_a, Q_b, \ldots$. For the Q_x properties we use $h\ g_i$ for the parameters corresponding to the fixpoints we unrolled to get the original five equations. Recall that we only unrolled one occurrence of $\mathit{unfold}_{\mathit{Step}}$ in each equation to get the next. So we want to look at

$$\begin{aligned}&Q_a\ (h\ g_1)\ g_2\ g_3\ g_4\ g_5\\ &Q_b\ g_1\ (h\ g_2)\ g_3\ g_4\ g_5\\ &\ldots\\ &Q_e\ g_1\ g_2\ g_3\ g_4\ (h\ g_5)\end{aligned}$$

For each P_x/Q_x above we unfold definitions and look at the cases (possibly using a table). We need to prove the property in each case of $\mathit{next}_x\ s_x$ and we look at what other properties P_x/Q_x we need.

In the *zip5* example we will find that we can prove

$$
\begin{aligned}
&P_a\ (h\ g) \Longleftarrow P_a\ g \quad \wedge \quad P_b\ g \\
&P_b\ (h\ g) \Longleftarrow P_b\ g \quad \wedge \quad P_c\ g \\
&\ldots \\
&P_e\ (h\ g) \Longleftarrow P_e\ g \quad \wedge \quad P_a\ g \\
&Q_a\ (h\ g_1)\ g_2 \ldots g_5 \Longleftarrow Q_a\ g_1 \ldots g_5 \wedge Q_b\ g_1 \ldots g_5 \\
&Q_b\ g_1\ (h\ g_2) \ldots g_5 \Longleftarrow Q_b\ g_1 \ldots g_5 \wedge Q_c\ g_1 \ldots g_5 \\
&\ldots \\
&Q_e\ g_1\ g_2 \ldots (h\ g_5) \Longleftarrow Q_e\ g_1 \ldots g_5 \wedge Q_a\ g_1 \ldots g_5
\end{aligned}
$$

Looking at this dependency pattern we decide that the appropriate choice for the P_x properties is to add them all into a single five-clause induction hypothesis and use ordinary fixpoint induction in a single variable. For the Q_x properties on the other hand we need to use fixpoint induction in five variables for all five properties.

3.10 Stream fusion for abstract types

The initial motivation for stream fusion was to fuse functions on sequences represented as arrays (Coutts et al., 2007a). In our formal treatment thus far we have only considered stream fusion for (co)data types defined as the fixpoint of a functor. Arrays are not such a type.

Another touted promise of stream fusion is that it is potentially a generic framework that could operate over any sequence type for which one can define suitable *stream* and *unstream* functions. In particular there is the hope that it should be possible for functions on streams to be defined once and reused multiple times with different sequence types.

At the time of writing there have been at least five full-scale implementations of sequence libraries using stream fusion. Of these, only one is for lists while the others are for specialised representations of sequences of bytes or characters.

More generally than arrays, we would like a theoretical framework that we can apply for any abstract type that represents a sequence. Such a framework should set out the proof obligations for an author of a sequence ADT who wishes to use stream fusion for their library.

The approach we have taken in System F relies heavily on the properties of final co-data and does not obviously generalise to abstract types. On the other hand, the data abstraction approach we have use for skipping streams in CPOs

seems like a natural approach to take for proving correctness of stream fusion for abstract sequence types.

In our data abstraction approach with streams and lists, we use a skipping stream type as the concrete type and lists as the abstract type. The intuition is that we should be able to apply stream fusion whenever skipping streams are a valid data abstraction of a target type. That is, the skipping stream remains the concrete type and the sequence ADT is the abstract type.

To develop a theory for stream fusion for ADTs we should re-analyse the proofs from Section 3.8 and see what list properties we rely on that we might lift out as requirements for an ADT.

For starters, assume we have some abstract type A with functions *stream* and *unstream*. The A type could be polymorphic in an element type

$$\begin{aligned} &\mathit{stream} &&:: \mathit{Stream}\ x \to A\ x \\ &\mathit{unstream} &&:: A\ x \to \mathit{Stream}\ x \end{aligned}$$

Or it could be monomorphic and use some fixed element type

$$\begin{aligned} &\mathit{stream} &&:: \mathit{Stream}\ E \to A \\ &\mathit{unstream} &&:: A \to \mathit{Stream}\ E \end{aligned}$$

The definitions and proofs below will not depend on this choice.

We will need to redefine the $\approx$ equivalence relation. Recall that it is a logical relation so the behaviour at function and other functorial types is prescribed and we only get to choose the behaviour at base types. At stream type we defined $\approx$ by

$$s \approx s' \iff \mathit{unstream}\ s = \mathit{unstream}\ s'$$

We can keep this definition and use the new *unstream* for our abstract type.

Recall Theorem 3.8.3, the fusion rule for skipping streams

$$f_s \approx f_s \Longrightarrow f_s\ (\mathit{stream}\ (\mathit{unstream}\ s)) \approx f_s\ s$$

The proof for this theorem was

$$\begin{aligned} & f_s \approx f_s \\ \iff & \quad \{ \text{ definition of } \approx \text{ at function types } \} \\ & s \approx s' \implies f_s\ s \approx f_s\ s' \\ \Longrightarrow & \quad \{ \text{ substitute } s' := s, s := \mathit{stream}\ (\mathit{unstream}\ s) \ \} \\ & \mathit{stream}\ (\mathit{unstream}\ s) \approx s \implies f_s\ (\mathit{stream}\ (\mathit{unstream}\ s)) \approx f_s\ s \\ \Longrightarrow & \quad \{ \text{ lemma } \mathit{stream}\ (\mathit{unstream}\ s) \approx s \ \} \\ & f_s\ (\mathit{stream}\ (\mathit{unstream}\ s)) \approx f_s\ s \end{aligned}$$

Beyond the updated $\approx$, the only other property this proof relies on is the lemma $stream\ (unstream\ s) \approx s$. The proof for this lemma was

$$
\begin{array}{ll}
 & stream\ (unstream\ s) \approx s \\
\iff & \quad \{\ \text{definition of } \approx \text{ at stream type}\ \} \\
 & unstream\ (stream\ (unstream\ s)) = unstream\ s \\
\impliedby & \quad \{\ \text{follows from lemma with } x := unstream\ s\ \} \\
 & unstream\ (stream\ x) = x
\end{array}
$$

The $\approx$ property still holds so the only property we rely on here is the lemma $unstream\ (stream\ x) = x$.

So we can conclude that the stream fusion transformation holds for any ADT for which $unstream\ (stream\ x) = x$. Of course the fusion transformation still has the side condition $f_s \approx f_s$ and this will need to be proved afresh for each stream function for each ADT with its associated definitions of *stream* and *unstream*.

3.10.1 Simple array example

We should look at some example beyond lists if only to check that the requirements on sequence ADTs given above are general and are not only satisfied by lists. Arrays are a relevant example since most stream fusion implementations to date have been for some kind of array. We will look at a very simple model of arrays. We are not interested in every last detail of yet-another induction proof, we will simply sketch out the structure.

We will use the following simple signature of a one-dimensional array indexed by natural numbers that is polymorphic in its element type.

$$
\begin{array}{l}
\textbf{data}\ Array\ a \\
(!) :: Array\ a \to Int \to a \\
len :: Array\ a \to Int \\
arr :: Builder\ a \to Array\ a
\end{array}
$$

The *len* operation gives the length which is the exclusive upper bound on the range of the array. Arrays are constructed using a builder monoid with the following signature

$$
\begin{array}{l}
\textbf{data}\ Builder\ a \\
empty :: Builder\ a \\
(\diamond) \quad :: Builder\ a \to Builder\ a \to Builder\ a \\
single :: a \to Builder\ a
\end{array}
$$

While these two signatures are simple and nothing requires that they be implemented as an array – the simplest implementation would be a binary tree – it is nevertheless a realistic model because it can be implemented as an array with the usual asymptotic complexity for the array operations. In particular it is possible to implement the *Builder* abstraction such that constructing an array takes only linear time and with low constant factors[6].

Of course we need to know something about how arrays behave. We will assume a number of properties that arrays should obviously support. Firstly, we need an equality principle.

$$\begin{aligned} a = a' \iff & \mathit{len}\ a = \mathit{len}\ a' \\ \wedge \quad & \forall i.\ i < \mathit{len}\ a \Longrightarrow a\,!\,i = a'\,!\,i \end{aligned}$$

Next we need to know the behaviour of observations on arrays constructed in the various possible ways. For empty and single-place arrays we assume

$$\begin{aligned} \mathit{len}\ (\mathit{arr}\ \mathit{empty}) &= 0 \\ \mathit{len}\ (\mathit{arr}\ (\mathit{single}\ e)) &= 1 \\ \mathit{arr}\ (\mathit{single}\ e)\,!\,0 &= e \end{aligned}$$

For arrays constructed via the builder monoid using the monoid operation we assume

$$\begin{aligned} \mathit{len}\ (\mathit{arr}\ (b \diamond b')) &= \mathit{len}\ (\mathit{arr}\ b) + \mathit{len}\ (\mathit{arr}\ b') \\ \mathit{arr}\ (b \diamond b')\,!\,i &= \textbf{if}\ i < lb\ \textbf{then}\ \mathit{arr}\ b\ !\,i \\ &\qquad\qquad\quad \textbf{else}\ \ \mathit{arr}\ b'\,!\,(i - lb) \\ &\quad\ \textbf{where} \\ &\qquad lb = \mathit{len}\ (\mathit{arr}\ b) \end{aligned}$$

Note that we have not specified the strictness of the *single* function or $\diamond$ operator. We will return to this point.

We can now define the *stream* and *unstream* functions

$$\begin{aligned} &\mathit{stream} :: \mathit{Array}\ a \to \mathit{Stream}\ a \\ &\mathit{stream}\ a = \mathit{Stream}\ \mathit{next}\ 0 \\ &\quad \textbf{where} \\ &\qquad \mathit{next}\ i \mid i < \mathit{len}\ a \ \ = \mathit{Yield}\ (a\,!\,i)\ (i+1) \\ &\qquad\qquad\ \mid \mathit{otherwise} = \mathit{Done} \end{aligned}$$

[6] For example using a monoid to compose actions on mutable arrays in the *ST* monad.

$$
\begin{aligned}
&\mathit{unstream} :: \mathit{Stream}\ a \to \mathit{Array}\ a \\
&\mathit{unstream}\ (\mathit{Stream}\ \mathit{next}\ s) = \mathit{arr}\ (\mathit{go}\ s) \\
&\quad \textbf{where} \\
&\qquad \mathit{go}\ s = \textbf{case}\ \mathit{next}\ s\ \textbf{of} \\
&\qquad\qquad\qquad \mathit{Done} \to \mathit{empty} \\
&\qquad\qquad\qquad \mathit{Skip}\quad s' \to \mathit{go}\ s' \\
&\qquad\qquad\qquad \mathit{Yield}\ x\ s' \to \mathit{single}\ x \diamond \mathit{go}\ s'
\end{aligned}
$$

With these definitions and properties in place we can look at how we might prove the basic stream fusion lemma.

Lemma 3.10.1. $\mathit{unstream}\ (\mathit{stream}\ a) = a$

Proof. Working on the left hand side gives us a straightforward recursive definition

$$
\begin{aligned}
&\quad \mathit{unstream}\ (\mathit{stream}\ a) \\
&= \\
&\quad \mathit{unstream}\ (\mathit{Stream}\ \mathit{next}\ 0) \\
&\qquad \textbf{where} \\
&\qquad\quad \mathit{next}\ i \quad \mid i < \mathit{len}\ a \ = \mathit{Yield}\ (a\,!\,i)\ (i+1) \\
&\qquad\qquad\qquad\ \mid \mathit{otherwise} = \mathit{Done} \\
&= \\
&\quad \mathit{arr}\ (\mathit{go}\ 0) \\
&\qquad \textbf{where} \\
&\qquad\quad \mathit{go}\ s = \textbf{case}\ \mathit{next}\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad \mathit{Done} \to \mathit{empty} \\
&\qquad\qquad\qquad\qquad \mathit{Skip}\quad s' \to \mathit{go}\ s' \\
&\qquad\qquad\qquad\qquad \mathit{Yield}\ x\ s' \to \mathit{single}\ x \diamond \mathit{go}\ s' \\
&\qquad\quad \mathit{next}\ i \quad \mid i < \mathit{len}\ a \ = \mathit{Yield}\ (a\,!\,i)\ (i+1) \\
&\qquad\qquad\qquad\ \mid \mathit{otherwise} = \mathit{Done} \\
&= \\
&\quad \mathit{arr}\ (\mathit{go}\ 0) \\
&\qquad \textbf{where} \\
&\qquad\quad \mathit{go}\ i \mid i < \mathit{len}\ a \ = \mathit{single}\ (a\,!\,i) \diamond \mathit{go}\ (i+1) \\
&\qquad\qquad\ \mid \mathit{otherwise} = \mathit{empty}
\end{aligned}
$$

To prove this is equal to simply a we need to use our array equality principle

$$
\begin{aligned}
a = a' \iff & \mathit{len}\ a = \mathit{len}\ a' \\
\wedge \quad & \forall i.\ i < \mathit{len}\ a \implies a\,!\,i = a'\,!\,i
\end{aligned}
$$

This gives us two parts to prove. For the two parts we can use the properties of $len\ (arr\ (b \diamond b'))$ and $arr\ (b \diamond b')\,!\,i$ to break down the recursive case $single\ (a\,!\,i) \diamond go\ (i+1)$.

We can prove $len\ (arr\ (go\ 0)) = len\ a$ by natural number induction using the hypothesis

$$i < len\ a \Longrightarrow len\ (arr\ (go\ (len\ a - i))) = i$$

The zero case is a matter of evaluation and applying the array rule $len\ (arr\ empty) = 0$. For the $i+1$ case we calculate

$$\begin{array}{ll}
& len\ (arr\ (go\ (len\ a - (i+1)))) \\
= & \{ guard\ is\ true\ since \mid len\ a - (i+1) < len\ a \mid \} \\
& len\ (arr\ (single\ (a\,!\,i) \diamond go\ (len\ a - (i-1) + 1))) \\
= & \quad \{ \text{ array rule } len\ (arr\ (b \diamond b')) = len\ (arr\ b) + len\ (arr\ b') \ \} \\
& len\ (arr\ (single\ (a\,!\,i))) + len\ (arr\ (go\ (len\ a - (i-1) + 1))) \\
= & \quad \{ \text{ array rule } len\ (arr\ (single\ e)) = 1 \ \} \\
& 1 + len\ (arr\ (go\ (len\ a - i))) \\
= & \quad \{ \text{ induction hypothesis } len\ (arr\ (go\ (len\ a - i))) = i \ \} \\
& 1 + i
\end{array}$$

We can do a similar natural number induction for the indexing part. We can prove the following by induction on i

$$i \leqslant j < len\ a \Longrightarrow arr\ (go\ (j - i))\,!\,i = a\,!\,j$$

Instantiating this with $i := j$ gives us the result

$$j < len\ a \Longrightarrow arr\ (go\ 0)\,!\,j = a\,!\,j$$

The zero case relies on the array rule to decompose $arr\ (b \diamond b')\,!\,j$

$$\begin{array}{l}
\quad arr\ (go\ j)\,!\,0 \\
= \\
\quad arr\ (single\ (a\,!\,j) \diamond go\ (j+1))\,!\,0 \\
= \\
\quad \textbf{if}\ 0 < 1 \\
\qquad \textbf{then}\ arr\ (single\ (a\,!\,j))\,!\,0 \\
\qquad \textbf{else}\ \ \ldots \\
= \\
\quad a\,!\,j
\end{array}$$

The inductive case is similar

$$
\begin{aligned}
& arr\ (go\ (j-(i+1)))\,!\,(i+1) \\
= & \\
& arr\ (single\ (a\,!\,(j-i-1)) \diamond go\ (j-i))\,!\,(i+1) \\
= & \\
& \textbf{if}\ (i+1)<1 \\
& \quad \textbf{then}\ldots \\
& \quad \textbf{else}\ arr\ (go\ (j-i))\,!\,i \\
= & \\
& arr\ (go\ (j-i))\,!\,i \\
= & \\
& a\,!\,j
\end{aligned}
$$

□

Of course the other part of applying stream fusion to arrays is the need to prove the abstraction property that relates each stream and equivalent array function. Consider a very simple example, the *head* function. On arrays this is simply

$$
\begin{aligned}
& head_a :: Array\ a \to a \\
& head_a\ a = a\,!\,0
\end{aligned}
$$

The stream version is

$$
\begin{aligned}
& head_s :: Stream\ a \to a \\
& head_s\ (Stream\ next\ s) = loop\ s \\
& \quad \textbf{where} \\
& \quad\quad loop\ s = \textbf{case}\ next\ s\ \textbf{of} \\
& \quad\quad\quad\quad\quad\quad Done\ \ \ \ \ \ \to error\ \texttt{"head: empty"} \\
& \quad\quad\quad\quad\quad\quad Skip\ \ \ \ s' \to loop\ s' \\
& \quad\quad\quad\quad\quad\quad Yield\ x\ \_ \to x
\end{aligned}
$$

The abstraction property relating the two is

$$head_s = head_a \circ unstream$$

In the non-$\perp$ case we have to show

$$head_s\ (Stream\ next\ s) = head_a\ (unstream\ (Stream\ next\ s))$$

Unfolding the left hand side gives

$$
\begin{array}{l}
\quad \mathit{head}_a\ (\mathit{unstream}\ (\mathit{Stream}\ \mathit{next}\ s)) \\
= \\
\quad \mathit{arr}\ (\mathit{go}\ s)\ !\ 0 \\
\qquad \textbf{where} \\
\qquad\quad \mathit{go}\ s = \textbf{case}\ \mathit{next}\ s\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Done} \rightarrow \mathit{empty} \\
\qquad\qquad\qquad\qquad \mathit{Skip}\quad s' \rightarrow \mathit{go}\ s' \\
\qquad\qquad\qquad\qquad \mathit{Yield}\ x\ s' \rightarrow \mathit{single}\ x \diamond \mathit{go}\ s'
\end{array}
$$

Similarly with the right hand side

$$
\begin{array}{l}
\quad \mathit{head}_s\ (\mathit{Stream}\ \mathit{next}\ s) \\
= \\
\quad \mathit{loop}\ s \\
\qquad \textbf{where} \\
\qquad\quad \mathit{loop}\ s = \textbf{case}\ \mathit{next}\ s\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{Done} \rightarrow \mathit{error}\ \texttt{"head: empty"} \\
\qquad\qquad\qquad\qquad \mathit{Skip}\quad s' \rightarrow \mathit{loop}\ s' \\
\qquad\qquad\qquad\qquad \mathit{Yield}\ x\ \_ \rightarrow x
\end{array}
$$

Of course here we are back to non-well-founded recursion due to the *Skip* case so we would have to return to using fixpoint induction. In the *Done* case we index the empty array on the left and we have error on the right hand side. The *Skip* case would use the induction hypothesis.

The *Yield* case is the interesting one. On the left hand side we have $\mathit{single}\ x \diamond \mathit{go}\ s'$ so we would obviously want to apply the array rule matching $\mathit{arr}\ (b \diamond b')\ !\ i$ so that we get $\mathit{arr}\ (\mathit{single}\ x)\ !\ 0 = x$ on the left hand side. Whether or not we can apply this rule however depends on whether our model of arrays says that $\diamond$ is strict or not. If $\diamond$ is non-strict then we can use the array rule to decompose $\mathit{single}\ x \diamond \mathit{go}\ s'$ even if $\mathit{go}\ s'$ turns out to be $\bot$. However most types that we would recognise as arrays (as opposed to trees) will require that $\diamond$ is strict in both arguments. In this case then we must accept that the theorem is not true because we cannot guarantee the side condition that $\mathit{go}\ s'$ is not $\bot$.

Stepping back, what this tells us is that operations on streams are not totally independent of the choice of concrete data structure we use them with. While we could use head_s as defined above to model the list function *head*, we cannot use it to model *head* on arrays. The key difference is that lists are non-strict in their tails while arrays are strict. So a stream function that only consumes a prefix of a stream is perfectly valid to model a list function but most array

implementations require the full array spine be constructed. Similarly, some arrays are strict in their elements. In our simple array model this corresponds to *single* being strict.

It is still possible to correctly model the array functions using stream functions, it just requires some adaptation of the stream functions. For example a version of $last_s$ intended to model the list function *last* would work unchanged for arrays that are not strict in their elements because $last_s$ is already strict in the spine of the sequence. On the other hand it would require modification to accurately model fully-strict arrays; it would have to force each element, e.g. using *seq*.

More generally, we can observe that arrays are simply a smaller type than lists or streams. While we can use streams as a concrete representation for both lists and arrays, every stream corresponds to some list while the same is not true for arrays – some stream values represent no array. More formally, the abstraction function *unstream* for lists is total while for arrays it is partial. It is not a problem to pick a concrete representation with unused values but it does mean not all stream functions correspond to array functions, such as $head_s$ above.

3.10.2 Fusing conversions between fusible types

If we have several fusible types, e.g. lists and arrays, than we can convert between them using the appropriate combinations of *stream* and *unstream*, for example

$$listToArray :: [a] \to Array\ a$$
$$listToArray = unstream_{array} \circ stream_{list}$$
$$arrayToList :: Array\ a \to [a]$$
$$arrayToList = unstream_{list} \circ stream_{array}$$

We can also fuse such conversions. Take for example the term

$$map_{array}\ f \circ listToArray$$

which we can unfold to

$$map_s\ f \circ stream_{array} \circ unstream_{array} \circ stream_{list}$$

We can apply the *stream*/*unstream* rule for the array type leaving us with just

$$map_s\ f \circ stream_{list}$$

There is no problem with the two sequence types having different strictness properties, as is the case between lists and arrays. Correctness is ensured by the

producers and consumers satisfying the abstraction property for the appropriate data type. As illustrated at the end of the previous section with the *head* example, it should not be assumed that a stream function that satisfies the abstraction property for one type will satisfy it for another.

3.11 Testing stream fusion

In the previous sections we concentrated on formal proofs that stream functions and the corresponding list functions satisfy the abstraction property. While the results are satisfactory, it is clear that the effort per function is relatively high. In practice, for people implementing a library using stream fusion, there may be insufficient time available to develop formal proofs for each function, even with the help of a proof assistant. It is desirable therefore to have additional lightweight semi-formal methods that provide some degree of confidence but with considerably reduced effort. This section presents a testing method which has been employed in a full-scale implementation of a list library using stream fusion.

The method uses two kinds of property-based testing. Property-based testing is a common technique in the functional programming community. It was first popularised by Claessen and Hughes (2000) with the *QuickCheck* testing framework which takes the approach of testing executable properties with randomly generated test cases. For example, we wish to check the abstraction property for the $lines_s$/*lines* functions

$$\forall s.\ unstream\ (lines_s\ s) = lines\ (unstream\ s)$$

A key idea in property based testing is to convert this mathematical property into an executable property. We define an executable property by changing mathematical equality $(=)$ for value equality within the language $(\equiv)$, and by making the universally quantified variable a function parameter

$$prop\_lines :: Stream\ Char \to Bool$$
$$prop\_lines\ s = unstream\ (lines_s\ s) \equiv lines\ (unstream\ s)$$

The approach *QuickCheck* takes is to randomly generate input values and to evaluate *prop_lines* on each test case. The generation of arbitrary inputs is done in a type-directed and modular way using type classes. Since testing the *prop_lines* property relies on generating arbitrary values of type *Stream Char* we must provide an appropriate type class instance for the *Stream* type.

With that in place we can test the property

```
ghci> quickCheck prop_lines
OK, passed 100 tests.
```

In practice the property-based method works well for finding bugs. Runciman et al. (2008) proposed another approach to property-based testing, embodied in the the *SmallCheck* framework. The key difference is in how test cases are constructed. Instead of using random values, test cases by generated by enumerating all possible values of the parameter type, up to some size bound. For example, for integers we simply enumerate them up to some limit. Structured types can also be enumerated in a regular way.

Both techniques can generate test cases that use function values as inputs. This is important for properties such as

$$prop\_map\ f :: (Int \to Int) \to Stream\ Int \to Bool$$
$$prop\_map\ f\ s = unstream\ (map_s\ f\ s) \equiv map\ f\ (unstream\ s)$$

We have successfully used this method in practice to test an implementation of a list library using stream fusion (Coutts et al., 2007b, Section 6.3). The library implements the list functions from the Haskell 98 specification, providing fusible versions where possible. There is an an executable property for each function, relating it to the definition given in the Haskell 98 specification[7]. In a few cases we must add side conditions to satisfy preconditions such as lists being non-empty. For testing functions such as *iterate* and *cycle* that generate infinite lists, we use an equality approximation that considers only a finite prefix.

3.11.1 Strictness properties

There is, however, a big blind spot with the testing system as described thus far and that is strictness properties. The remainder of this section describes a new member of the family of property-based testing methods and its application to testing stream library functions.

Strictness is a crucial part of the semantics of functions in a non-strict language. Getting the strictness wrong has often proved to be the source of subtle bugs

[7] At the time this list library was developed, the significance of the abstraction property was not appreciated. The test properties used were weaker versions that cannot determine if skips are handled incorrectly.

in real programs and yet strictness properties are frequently overlooked, especially in testing. Indeed even the Haskell 98 report does not specify the strictness properties of the functions in the *List* library, except by giving informal sample implementations. As a consequence, and as we discovered, some of the functions from the Haskell 98 *List* module arguably have the wrong strictness properties. We will look at these infelicities once we have introduced the strictness testing technique.

Strictness is both a semantic issue and an operational issue. Operationally, strictness affects how much must be evaluated to get an answer and in what order the evaluation may be done. Semantically, strictness determines the result of functions on partial inputs.

So the way to test strictness properties is to test with partial values. Both *QuickCheck* and *SmallCheck* generate test cases using only total values. Thus we cannot use either framework directly, but we can use a similar approach.

Consider the first example again

$$\forall s.\ unstream\ (lines_s\ s) = lines\ (unstream\ s)$$

Previously we tested this property with s ranging over only total streams. We would now like to test it with s ranging over all partial streams. There is a problem, however, when it comes to making this property into an executable test. For total values we were able to change mathematical equality $(=)$ for value equality within the language $(\equiv)$. The problem is that with mathematical equality $\bot = \bot$, while with value equality $\bot \equiv \bot = \bot$. That is, evaluating $(\equiv)$ where both inputs are $\bot$ itself evaluates to $\bot$ rather than to *True*.

We use the techniques of Danielsson and Jansson (2004) to define an operator $(\mathrel{\mathring{=}})$ that is ordinary equality $(\equiv)$ on total values but with different behaviour on $\bot$ values

$$\begin{aligned}
\bot &\mathrel{\mathring{=}} \bot = True \\
_ &\mathrel{\mathring{=}} \bot = False \\
\bot &\mathrel{\mathring{=}} _ = False \\
a &\mathrel{\mathring{=}} b = a \equiv b
\end{aligned}$$

Defining this operator requires a modicum of 'cheating', relying as it does on a function with the properties

$$\begin{aligned}
isBottom\ \bot &= True \\
isBottom\ _ &= False
\end{aligned}$$

Such a function cannot be defined within the language: if it existed it would solve the halting problem. We can approximate *isBottom* for the subset of $\bot$

values that represent computations that terminate with an error, but not $\perp$ values representing non-termination, that is

$$\begin{aligned} &\mathit{isBottom}\ \perp_{\mathit{error}} = \mathit{True} \\ &\mathit{isBottom}\ \perp_{\mathit{loop}} = \perp_{\mathit{loop}} \\ &\mathit{isBottom}\ \_ = \mathit{False} \end{aligned}$$

This approximation is sufficient in practice for writing properties because we typically do not write programs or properties that involve non-termination.

Even this approximation cannot be defined within in the language because it is not continuous in the CPO sense and it distinguishes different kinds of $\perp$ values that are semantically equal. It can be implemented by going outside of the normal language semantics and making use of a backdoor in common Haskell implementations. The trick is described in more detail by Danielsson and Jansson (2004, Section 5). Due to the non-standard semantics however, we must be careful with how we use *isBottom* and any operators defined in terms of it. We should use it only as if we were writing meta-level properties and not use it in ordinary value-level computations.

Armed with the $\stackrel{\circ}{=}$ operator we can write the executable property

$$\mathit{prop\_lines}\ s = \mathit{unstream}\ (\mathit{lines}_s\ s) \stackrel{\circ}{=} \mathit{lines}\ (\mathit{unstream}\ s)$$

An interesting feature of testing with partial values rather than just total values is that specifying properties of partial functions is simpler. When testing partial functions (e.g. *head*) with total values we must only generate values within the domain of the function (i.e. non-empty lists) so as to avoid the whole test evaluating to $\perp$. When testing with partial inputs we are already prepared to handle $\perp$ as a result. We can thus simplify the test properties for partial functions by discarding preconditions on the test data. Indeed we must remove any preconditions implemented by filtering test data as they themselves would immediately evaluate to $\perp$ and we would not evaluate the function under test.

3.11.2 Generating partial values

Now that we can write executable properties over partial values, the next step is to generate test cases. We have seen the approaches taken for *QuickCheck* and *SmallCheck*. One observation is that almost all the interesting variations in partial structures are exhibited in 'small' values. We want to be sure that we test all these small partial values. It is thus natural to adopt the *SmallCheck* model of generating all values up to some size limit. Another reason is that

since functions cannot observe $\bot$ and list functions usually work in a regular fashion over the structure of lists, we do not need very complex partial values to cover all control flow paths. That said, there are some functions where we need enough variation in the total parts of the value to get sufficient coverage. For example, to test the rather subtle strictness properties of the *lines* function we need to test with strings that contain newline characters.

The way *SmallCheck*'s generators work is to produce a series of all values in order of size, up to some given bound. We can adapt *SmallCheck* to generate all partial values, in an order consistent with the domain theoretic ordering. For example for lists of integers we get

$$\bot,\ [\,],\ \bot : \bot,\ 0 : \bot,\ \bot : [\,],\ 0 : [\,],\ \ldots$$

This adaptation is relatively straightforward. For example, *SmallCheck* defines the series operators $(|||)$ and $(\bowtie)$

$$\begin{aligned}
&\textbf{type}\ \mathit{Series}\ a = \mathit{Int} \to [a] \\
&(|||) :: \mathit{Series}\ a \to \mathit{Series}\ a \to \mathit{Series}\ a \\
&(\bowtie) :: \mathit{Series}\ a \to \mathit{Series}\ b \to \mathit{Series}\ (a, b)
\end{aligned}$$

These operators are used to help build *Series* of values for regular algebraic data types. For example they are used in the *Serial* instances for pairs and lists

$$\begin{aligned}
&\textbf{instance}\ (\mathit{Serial}\ a, \mathit{Serial}\ b) \Rightarrow \mathit{Serial}\ (a, b)\ \textbf{where} \\
&\quad \mathit{series} = \mathit{series} \bowtie \mathit{series} \\
&\textbf{instance}\ \mathit{Serial}\ a \Rightarrow \mathit{Serial}\ [a]\ \textbf{where} \\
&\quad \mathit{series} = \mathit{cons0}\ [\,] \ |||\ \mathit{cons2}\ (:)
\end{aligned}$$

The original definitions of $(|||)$ and $(\bowtie)$ for total values are

$$\begin{aligned}
&(|||) :: \mathit{Series}\ a \to \mathit{Series}\ a \to \mathit{Series}\ a \\
&s1 \ ||| \ s2 = \lambda d \to s1\ d +\!\!+ s2\ d \\
&(\bowtie) :: \mathit{Series}\ a \to \mathit{Series}\ b \to \mathit{Series}\ (a, b) \\
&s1 \bowtie s2 = \lambda d \to [(x, y) \mid x \leftarrow s1\ d, y \leftarrow s2\ d]
\end{aligned}$$

To modify these to generate *Series* of partial values, we merely need to add an extra $\bot$ at the beginning of the series and to shuffle the rest of the series down by one

$$\begin{aligned}
&(s1 \ ||| \ s2)\ 0 = \bot : [\,] \\
&(s1 \ ||| \ s2)\ d = \bot : s1\ (d-1) +\!\!+ s2\ (d-1) \\
&(s1 \bowtie s2)\ 0 = \bot : [\,] \\
&(s1 \bowtie s2)\ d = \bot : [(x, y) \mid x \leftarrow s1\ (d-1), y \leftarrow s2\ (d-1)]
\end{aligned}$$

With these modified definitions the *Serial* instances for pairs and lists work unaltered. The instances for primitive types like *Int* do have to be changed.

Another difference compared to *SmallCheck* is that the pretty printer displays $\perp$ values as text rather than failing with an exception. This enables the display of failing test cases. Again, this trick is described by Danielsson and Jansson (2004).

It is not directly possible to generate values with polymorphic types. When testing properties using polymorphic functions such as *map* we must write properties that instantiate them monomorphically.

$$prop\_map :: (A \to B) \to Stream\ A \to Bool$$
$$prop\_map\ f\ s = map\ f\ (unstream\ s) \stackrel{\circ}{=} unstream\ (map_s\ f\ s)$$

The types A and B stand in for polymorphic type variables but are in fact monomorphic representative types. For testing at total values we would typically pick A and B to be a type with many possible values like *Int*. For strictness testing however we need only $\perp$ and non-$\perp$ values[8]. We therefore define

$$\textbf{newtype}\ A = A\ ()\ \textbf{deriving}\ (Eq, Show, Serial)$$
$$\textbf{newtype}\ B = B\ ()\ \textbf{deriving}\ (Eq, Show, Serial)$$

This means A and B have series consisting of just $\perp$ and $()$.

3.11.3 Evaluation

To evaluate the utility of the testing method we compared the Haskell 98 list specification with the implementation shared by the major Haskell implementations. We wrote properties to equate corresponding functions. To our surprise, the tests uncovered several differences between the Haskell 98 specification and the common `base` implementation[9] of the *List* module. In three cases the common implementation is stricter. In each case however the differences can be attributed to mistakes in the Haskell 98 specification or to legitimate disagreement.

[8] Ironically this depends on a parametricity argument.

[9] Three Haskell implementations, GHC, NHC98 and Hugs98, share the same `base` package which contains an implementation of much of the Haskell standard library.

Two differences are for *splitAt* and *partition* which the Haskell 98 report specifies by

$$
\begin{aligned}
&\mathit{splitAt} && :: \mathit{Int} \to [a] \to ([a],[a]) \\
&\mathit{splitAt}\ n\ xs && = (\mathit{take}\ n\ xs, \mathit{drop}\ n\ xs) \\
&\mathit{partition} && :: (a \to \mathit{Bool}) \to [a] \to ([a],[a]) \\
&\mathit{partition}\ p\ xs && = (\mathit{filter}\ p\ xs, \mathit{filter}\ (\mathit{not} \circ p)\ xs)
\end{aligned}
$$

The implementations in the *List* module from the common `base` package differ in at least these cases

$$
\begin{aligned}
&\mathit{splitAt}_{h98}\ \bot\ xs = (\bot, \bot) \\
&\mathit{splitAt}_{h98}\ 0\ \bot = ([\,], \bot) \\
&\mathit{splitAt}_{h98}\ n\ \bot = (\bot, \bot) \\
&\mathit{splitAt}_{base}\ \bot\ xs = \bot \\
&\mathit{splitAt}_{base}\ n\ \bot = \bot \\
&\mathit{partition}_{h98}\ p\ \bot = (\bot, \bot) \\
&\mathit{partition}_{base}\ p\ \bot = \bot
\end{aligned}
$$

The general style of the Haskell 98 *List* functions is to be as lazy as possible, except where there are good reasons for being stricter. The reason to make *splitAt* and *partition* stricter than the above definitions is to allow more efficient implementations that share intermediate results. For example in *partition* we could call the predicate p only once for each element in the input list. Indeed implementations using sharing were given in the Haskell 1.4 library report. These two functions got 'simplified' in the transition to the Haskell 98 report and in the process their strictness properties were changed. For these reasons it seems clear that the specification is wrong and that the common implementation is correct.

The other case is *genericTake*. The common implementation uses a different order of pattern matching compared to the version given in the specification. The version from the specification is lazy in the numeric argument when the list is empty. On the other hand, the common implementation is always strict in the numeric argument. There is a good argument for preferring the common implementation rather than the specification: with the version of *genericTake* from the Haskell 98 specification it is not the case that $\mathit{take} = \mathit{genericTake}$ where as with the common implementation we do have that property.

Based on this experiment, of comparing the common list module implementation with the Haskell 98 specification, we may reasonably conclude that the strictness property testing method is useful. It lets programmers reuse their experience with other property-based testing methods and can uncover subtle

strictness mistakes that appear to be hard to find by other means. To our knowledge, the infelicities between the Haskell 98 list specification and the common implementation were not previously well known.

During the development of the the stream fusion version of the list library, several mistakes were found using strictness property tests. It is worth noting that these mistakes were not caught by the corresponding tests on total values – the mistakes were in the strictness of the functions.

For example the standard *lines* function is less strict than one might initially imagine. Our initial implementation using streams had the property that it would yield a line only once it encountered the end of the line. However the standard *lines* function is less strict; for example

$$lines\ (\texttt{'a'} : \texttt{'b'} : \texttt{'c'} : \bot) = (\texttt{'a'} : \texttt{'b'} : \texttt{'c'} : \bot) : \bot$$

That is, it returns a list where the first element is a list containing the initial input. It does not appear to be possible to implement this semantics using a stream version; at least not without using additional intermediate data structures.

In addition to detecting mistakes in the stream implementation of the library, the strictness testing highlighted two cases where our stream implementation is actually *less* strict than the specification. This is somewhat surprising since the Haskell 98 specification usually picks the least strict version.

The functions in question are given in the report as

$$\begin{aligned}
&intersperse_{h98} :: a \to [a] \to [a] \\
&intersperse_{h98}\ sep\ [\,] \quad\ = [\,] \\
&intersperse_{h98}\ sep\ [x] \quad\ = [x] \\
&intersperse_{h98}\ sep\ (x : xs) = x : sep : intersperse_{h98}\ sep\ xs \\
&unwords_{h98} :: [String] \to String \\
&unwords_{h98}\ [\,] \ = \texttt{""} \\
&unwords_{h98}\ ws = foldr1\ (\lambda w\ s \to w +\!\!+ \texttt{' '} : s)\ ws
\end{aligned}$$

These are straightforward and reasonable definitions. It is not immediately obvious how one could modify them to be less strict. An observation which may explain why the stream versions ended up less strict is that the stream versions are naturally rather low level. Instead of pattern matching more than one step ahead, as we can do with lists, stream versions must be completely explicit about demanding input and yielding output. It is perhaps not so surprising therefore that such an approach would tend to produce the least strict implementation, compared to a high level approach aimed at simplicity and elegance.

For easier comparison we can translate the stream versions back into functions directly on lists[10]

$$
\begin{aligned}
&\mathit{intersperse'} :: a \to [a] \to [a] \\
&\mathit{intersperse'}\ \_\ \ [\,] \qquad\quad = [\,] \\
&\mathit{intersperse'}\ \mathit{sep}\ (\mathit{x0} : \mathit{xs0}) = \mathit{x0} : \mathit{go}\ \mathit{xs0} \\
&\quad \textbf{where} \\
&\qquad \mathit{go}\ [\,] \qquad = [\,] \\
&\qquad \mathit{go}\ (x : \mathit{xs}) = \mathit{sep} : x : \mathit{go}\ \mathit{xs} \\
&\mathit{unwords'} :: [\mathit{String}] \to \mathit{String} \\
&\mathit{unwords'}\ [\,] \qquad\quad = [\,] \\
&\mathit{unwords'}\ (\mathit{cs0} : \mathit{css0}) = \mathit{go}\ \mathit{cs0}\ \mathit{css0} \\
&\quad \textbf{where} \\
&\qquad \mathit{go}\ [\,] \qquad \mathit{css} = \mathit{to}\ \mathit{css} \\
&\qquad \mathit{go}\ (c : \mathit{cs})\ \mathit{css} = c : \mathit{go}\ \mathit{cs}\ \mathit{css} \\
&\qquad \mathit{to}\ [\,] \qquad\quad = [\,] \\
&\qquad \mathit{to}\ (\mathit{cs} : \mathit{ccs}) \quad = \text{' '} : \mathit{go}\ \mathit{cs}\ \mathit{ccs}
\end{aligned}
$$

Their low level nature is apparent. These definitions differ from the versions in the Haskell 98 specification in the cases

$$
\begin{aligned}
&\mathit{intersperse}_{h98}\ (x : \bot) = \bot \\
&\mathit{intersperse'} \quad (x : \bot) = x : \bot \\
&\mathit{unwords}_{h98}\ (\mathit{cs} : \bot) = \bot \\
&\mathit{unwords'} \quad (\mathit{cs} : \bot) = \mathit{cs} +\!\!+ \bot
\end{aligned}
$$

To verify that our implementation is merely less strict and not simply wrong, we need a different comparison than $\doteq$. We use an operator $\mathbin{\mathring{\sqsubseteq}}$ that is an executable approximation of the domain-theoretic partial ordering $\sqsubseteq$. This and related operators are described by Danielsson and Jansson (2004). With the $\mathbin{\mathring{\sqsubseteq}}$ operator we are able to write properties to confirm that our implementations of *unwords* and *intersperse* do indeed refine the Haskell 98 specification.

[10] It has recently been proposed that the common implementation of *intersperse* be changed to be less strict by using this definition. See `http://hackage.haskell.org/trac/ghc/ticket/4282`

Chapter 4

Stream fusion is an optimisation

In this chapter we look at the details of the transformations involved in stream fusion and in what circumstances we can be sure that it leads to an optimisation.

4.1 Introduction

The general context, as set out in Chapter 1, is that we are working with pure functional programs that manipulate sequences and we will perform transformations on the programs with the aim of optimising them. Though we use Haskell (see Section 2.4), everything presented in this chapter should be transferable to any other pure functional programming language.

Some of our design choices, particularly the choice to use general purpose optimisations, are motivated by the desire to implement stream fusion in the context of a general purpose optimising compiler. Similarly, although it is not essential that stream fusion be applied at compile time, we want to be able to use stream fusion in the optimisation phase of a traditional static compiler. Thus it is important that it be possible to apply stream fusion at compile time, before any program inputs are available.

4.1.1 Justifying optimisation claims

As stated in the first chapter, there are two main criteria by which we should evaluate fusion systems: they should be correct and they should be an optimisation. When we claim that a transformation is correct, it is fairly clear what constitutes a valid justification. There is more room for interpretation however when it comes to what constitutes a valid justification for an optimisation claim.

In a previous paper (Coutts et al., 2007b) we gave an empirical justification that stream fusion could be an optimisation. We demonstrated that running times and memory allocations were improved for some benchmark programs[1]. There is some value in this kind of empirical result. It demonstrates that there are no overlooked details that would always prevent the optimisation in practice. For example, had such an experiment been performed for *unbuild/unfoldr* fusion then the discovery that it cannot optimise compositions with *filter* would have come much sooner.

On the other hand, such empirical evidence is not universally applicable and it does not provide any guarantees. It is tied to a specific implementation – GHC in the case of our own previous publications. It can be hard to tell precisely what transformations are being performed by the compiler to achieve the observed results. For negative results where benchmarks are made worse, it can be hard to distinguish between fundamental problems and problems due to the quality of implementation. Perhaps most importantly, it is not immediately clear which transformations it would be sufficient to implement in another compiler to be able to use the fusion system effectively.

As a complement to our earlier empirical work, in this chapter we will argue that the stream fusion system is an optimisation, and do so independently of any particular compiler. More precisely we will argue that there exists a sequence of semantic-preserving, syntactic, local transformations that leads to an improvement. We will only give heuristics for what transformations should be applied at which stage; we do not provide any algorithm to guarantee that the sequence of local transformations will always be found.

The choice of program representation and transformation is based on the goal of allowing implementations in a general purpose compiler. In particular by taking the approach of using a syntactic representation of programs and using a sequence of local transformations we match the traditional approach of optimising compilers. In a more special-purpose implementation, a hardware compiler perhaps, it may make sense to use a special representation of stream functions[2] and to implement the optimisation and compilation in a direct way.

As is common practice we will use heap allocations as a cost measure since this is much easier to reason about than time measurements. In practice allocations are a reasonable proxy for time. We will not use a full formal cost model; rather we will note the change in allocations for each local transformation. We will

[1] It was actually a comparison to *foldr/build* rather than to a baseline of no fusion.

[2] The state machine form described in Section 4.4 is a plausible choice for a special purpose representation.

declare a sequence of transformations to be an improvement if it decreases the number of allocations.

As a caveat, it should be noted that we will account only for allocations of data constructors, not of closures. In the *foldr/build* system, for example, it is important to account for closures since the implementation of *foldl* in terms of *foldr* introduces the allocation of additional closures[3]. There does not appear to be a similar issue for stream fusion.

4.1.2 Terminology

We should be precise about what we mean by the terms 'fusion', 'fusion system' and 'optimisation'.

- By *fusion* we mean any kind of transformation that is intended to be an optimisation, though we typically use the term only for transformations that combine objects.
- A *fusion system* is more than just the application of a fusion transformation; a fusion system is all the transformations needed to eliminate the intermediate data structures.
- An *optimisation* is an improvement in some cost measure, usually time or allocations.

In particular, we use the term fusion to cover more than just the application of a fusion rule. For example, in this chapter we will describe fusion between *stream producers* and *stream consumers* which is rather more complicated than the simple application of a rewrite rule.

4.1.3 Good producers and good consumers

Recall from Section 1.3.8 that with the *unbuild/unfoldr* system, the *filter* example does not result in a reduced number of allocations. This was despite the fusion rule being applied successfully. The problem was that a subsequent and essential transformation could not be applied. The lesson is that the successful application of a fusion rule must not be confused with a fusion system giving an overall optimisation – even when the fusion rule is locally an optimisation.

[3] Gill (1996, Section 4.4) describes the analysis and transformation required to eliminate these additional closure allocations.

We must analyse the fusion system as a whole to be sure that it is an overall optimisation.

With stream fusion it was the addition of skips that enabled the definition of *filter* to be changed, which in turn lead to an improvement. Note that the switch to stream fusion only made the better definition possible, the change was not automatic. It is straightforward to translate the original poorly performing definition of *filter* into a poorly performing stream definition; simply make the stepper function recurse rather than use *Skip*:

$$
\begin{aligned}
next_{filter}\ p\ next\ s = \mathbf{case}\ next\ s\ \mathbf{of}& \\
\quad Done\ \ \ \ &\rightarrow Done \\
\quad Skip\ \ \ s' &\rightarrow Skip\ s' \\
\quad Yield\ x\ s' &\rightarrow \mathbf{if}\ p\ x\ \mathbf{then}\ Yield\ x\ s' \\
&\qquad\qquad\ \mathbf{else}\ \ next_{filter}\ p\ next\ s'
\end{aligned}
$$

The question therefore is whether there are any rules we can follow when writing stream functions that will ensure we end up with an optimisation.

Following the terminology of Gill (1996, Sections 3.5.2–3.5.4) we talk about *good producers* and *good consumers*. The purpose of this nomenclature is to provide a simple way to explain to programmers when to expect fusion to take place, without programmers having to know the details of the transformation. We can explain that particular list-producing functions are good producers and that particular list-consuming functions are good consumers in a particular argument (or arguments). We then promise programmers that the fusion optimisation occurs when a list produced by a good producer is directly consumed by a good consumer. This terminology and explanation can be used for lists, arrays and other sequence types using stream fusion.

This promise is relatively simple to explain and simple for programmers to remember and to apply. It is not quite so easy to substantiate. Our task is to define good producers and consumers, and to show how these definitions lead to an overall optimisation. As mentioned previously, we do not provide an algorithm to find a suitable sequence of transformations. While that task is necessary to be able to provide the simple promise, we must leave it to those implementing the fusion system in a particular compiler.

4.1.4 Overview

The argument that stream fusion for some sequence type is an overall optimisation has the following structure:

- we describe a sequence of transformations;
- we define sufficient conditions on good producers and consumers to ensure that we can perform the transformations;
- we account for the change in allocations over the whole sequence of transformations.

In the case of stream fusion for lists, the overall accounting argument is that for each good producer/consumer pair, stream fusion saves exactly one allocation per list element.

The transformations in the stream fusion system can be broken down into two major phases:

- the first involves applying the *stream*/*unstream* fusion rule to bring *stream producers* and *stream consumers* together;
- the second involves optimising compositions of stream producers and stream consumers to eliminate intermediate data constructors.

The first phase is relatively straightforward while the second is rather more involved.

The argument for stream fusion being an overall optimisation can be broken down into arguments for each phase which can be tackled separately. The structure of the argument in each phase follows the outline above.

The remainder of this chapter is organised as follows:

- We cover the first phase in Section 4.2. We have to account for the difference between simple functions and their fusible equivalents. We have to place conditions on good consumers and producers to ensure that it is always possible to unfold them to the point where we can apply the *stream*/*unstream* fusion rule.
- In Section 4.3 we outline the transformations and arguments for the second phase. In particular we identify *stream transformers* as an important class of stream functions in addition to stream producers and stream consumers.

- In Section 4.4 we describe a correspondence between stream producers and a kind of state machine. The state machine view helps to provide an intuition about the dynamic behaviour of streams. It also gives some intuition for how – and under what conditions – stream consumers, transformers and producers can be successfully fused.

- In Section 4.5 we give the transformation and arguments covering the fusion of stream transformers with stream producers. In particular we set out the constraints on stream producers.

- In Section 4.6 we give the transformation and arguments covering the fusion of stream consumers with stream producers. We set out the constraints on stream consumers.

- In Section 4.7 we account for the change in allocations for the stream fusion system as a whole.

- In Section 4.8 we look at the issue of expressiveness. Since we have to place constraints on good producers and good consumers to ensure that we can fuse them effectively, then we must consider the extent to which this restricts the range of fusible functions we can express.

4.1.5 Sufficient compiler optimisations

Sections 4.2, 4.5 and 4.6 give a detailed account of the sequence of transformations that make up stream fusion. While implementations do not have to follow this description it does serve as a checklist of compiler optimisations that are sufficient to support stream fusion.

We summarise that checklist here:

- Fusion rewrite rule

  The fusion law itself is a simple syntactic transformation. This could be implemented directly in the compiler, or via a general purpose rewrite rule system (e.g. Peyton Jones et al., 2001).

- Inlining and beta-reduction

  These are very basic local transformations. Santos (1995, Section 3.1 and 3.2) and Peyton Jones and Santos (1998, Section 4) give an account.

- Let floating

  This is used to get **let**-bindings "out of the way" to enable other transformations involving **case** expressions. Santos (1995, Section 3.4) and Peyton Jones and Santos (1998, Section 7) give a description.

- Case-of-known-constructor and case-of-case transformations

  These are key transformations that eliminate constructor allocations. Santos (1995, Sections 3.3 and 3.4) and Peyton Jones and Santos (1998, Section 5) describe these transformations. We also give a presentation in Sections 4.5.2 and 4.5.7.

- Call pattern specialisation

  This is a somewhat more sophisticated transformation introduced by (Peyton Jones, 2007). We also give a presentation in Section 4.6.2.

4.1.6 Lists and other sequence types

In this chapter we make the optimisation argument for stream fusion as applied to the standard list data type. As noted in the previous chapter, stream fusion can be applied to sequences types other than lists. Much of what we cover in this chapter is easily transferable to other sequence types such as arrays. In particular the transformation process and the argument that the transformations are possible is essentially independent of the sequence type. The details of the allocation accounting argument are however specific to each sequence type, though the general strategy should be transferable.

Throughout this chapter we will assume the following definition for the *Stream* data type.

$$\begin{aligned}
&\textbf{data}\ \mathit{Stream}\ a = \exists s.\ \mathit{Stream}\ (s \to \mathit{Step}\ a\ s)\ s \\
&\textbf{data}\ \mathit{Step}\ a\ s = \mathit{Done} \\
&\qquad\qquad\quad \mid \mathit{Skip}\quad s \\
&\qquad\qquad\quad \mid \mathit{Yield}\ a\ s
\end{aligned}$$

For lists, the definition of *stream* and *unstream* are as follows.

$$\begin{aligned}
&\mathit{stream} :: [a] \to \mathit{Stream}\ a \\
&\mathit{stream}\ xs = \mathit{Stream}\ \mathit{uncons}\ xs \\
&\quad \textbf{where} \\
&\qquad \mathit{uncons}\ [\,] \qquad = \mathit{Done} \\
&\qquad \mathit{uncons}\ (x : xs) = \mathit{Yield}\ x\ xs
\end{aligned}$$

```
unstream :: Stream a → [a]
unstream (Stream next s) = unfold next s
  where
    unfold next s = case next s of
                      Done      → []
                      Skip    s' →    unfold next s'
                      Yield x s' → x : unfold next s'
```

As it turns out, the details of the *stream* and *unstream* definitions are not especially important. We are able to treat them as ordinary stream producers and stream consumers. What matters is that they satisfy the conditions for stream producers and stream consumers that we set out in Sections 4.5.4 and 4.6.3 respectively. This gains us some independence from the sequence type: if we were to make an optimisation argument for a different sequence type then we would need to show that the *stream* and *unstream* definitions for that type satisfy the stream producer and consumer constraints.

4.1.7 Correctness requirements

We will of course have to start with the correctness requirements from the previous chapter. In particular, recall from Section 3.8.2 that to be able to use the simple rewrite rule $stream\ (unstream\ s) = s$, we need a context in which all functions that can observe streams have the property that they preserve equivalence on streams ($f_s \approx f_s$).

We follow the suggestion from Section 3.8.2 and take the approach of using a library. The library exports functions that operate on lists but internally are implemented in terms of streams. For example, instead of the ordinary definition of *map* on lists, our library defines *map* on lists in terms of map_s on streams:

$$map\ f = unstream \circ map_s\ f \circ stream$$

We require that all the functions in the library preserve equivalence on streams and that the *Stream* type is not exported from the library. Not exporting the *Stream* type means we cannot export any functions with types that mention *Stream*.

4.2 Applying *stream*/*unstream* fusion

Our starting situation is a list produced by a good producer that is immediately consumed by a good consumer.

$$consume\ (produce\ \ldots)$$

In this phase we can treat all such instances in a program independently, even for a function that is a good consumer in multiple arguments. In the next phase we will have to consider larger units.

The aim in this phase is to statically transform the above into the form

$$consume_s\ (stream\ (unstream\ (produce_s\ \ldots)))$$

at which point we will be able to apply *stream*/*unstream* fusion. A good producer must use *unstream* to construct its result and likewise a function that is a good consumer in a particular argument must use *stream* to consume that argument. This is not quite a sufficient condition however. The need to perform the transformation statically is significant. We must be able to perform the transformation without having to evaluate values only available at runtime. Consider for example the following producer

$$\begin{array}{l} consume\ (\textbf{if}\ even\ n\ \textbf{then}\ unstream\ (\ldots) \\ \qquad\qquad\qquad\qquad\ \ \textbf{else}\ \ unstream\ (\ldots)) \end{array}$$

It constructs a list using *unstream* but in two different ways depending on a runtime test. While in this simple example we could push the consumer inside each branch of the dynamic test, it is easy to construct more complex examples, e.g. using fixpoints, where we can no longer do a static transformation.

4.2.1 Good producer conditions

In deciding what conditions to require of good producers and good consumers we must be guided by the needs of the transformations we wish to perform. Our aim is to define relatively simple conditions, not to find the most general conditions or the most general form of transformation.

In this phase, we want to end up with an applicative form of stream consumers and stream producers, ready for the second phase. This will require that we bring *stream* and *unstream* combinators together and apply the *stream*/*unstream* fusion rule. In addition it will require that what we are left with after applying the fusion rule is an applicative form of stream consumers and producers. A

straightforward approach is to constrain the syntactic form of good consumers and good producers so that they can easily be unfolded into the form we need.

We stipulate that good producers be terms with the following form. At the top level we allow any mixture of lambda abstraction and **let** binding. The first body term must be either *unstream* applied to a *stream producer* term or it may be another good producer.

That is, syntactically, good producers will look like

$$f\ x = unstream\ \langle produce_s \rangle$$
$$\quad \textbf{where}\ \ldots$$

or similarly

$$f\ x = h\ (\ldots)$$
$$\quad \textbf{where}\ \ldots$$

Where h is some existing good producer. The first form is used for direct definitions in terms of stream functions such as

$$unfoldr\ f\ s = unstream\ (unfoldr_s\ f\ s)$$

while the second form allows derived definitions such as

$$iterate\ f = unfoldr\ (\lambda x \rightarrow Just\ (x, f\ x))$$

We will obviously also require that where good producers are defined in terms of other good producers that these definitions are not cyclic. We also require there to be a finite number of good producers in any particular program and that their definitions be available (which rules out dynamic code loading).

We will require that the stream producer term satisfies certain constraints, the details of which we defer to Section 4.5.4.

4.2.2 Good consumer conditions

We have a similar set of conditions for good consumers.

A function that is a good consumer in some argument must consume that argument exactly once and do so directly with *stream* and a *stream consumer* term or with another good consumer. Thus, syntactically, good consumers will look like

$$g\ x = \langle consume_s \rangle\ (stream\ x)$$

or similarly

$$g\ x = \ldots (h\ x) \ldots$$

where h is some existing good consumer. These two forms allow definitions such as

$$foldl\ f\ a\ xs = foldl_s\ f\ a\ (stream\ xs)$$
$$sum\ xs = foldl\ (+)\ 0\ xs$$

As with the producers, we require that the collection of good consumers is finite, acyclic and that all definitions are available. We will require that the stream consumer term $\langle consume_s \rangle$ satisfies a set of constraints which we will set out in Section 4.6.3.

It is important for our allocation accounting argument that good consumers consume their argument at most once. A consumer that has multiple occurrences where an argument is consumed would break our accounting argument due to the possibility of duplication. With a list argument it is possible to share the input list data structure at runtime and make multiple passes over it. However to apply the *stream*/*unstream* rule would require that we duplicate the producer into each occurrence where the argument is consumed. Duplicating the producer may also duplicate its runtime allocations which would be a problem for our allocation argument[4].

4.2.3 Bringing *stream* and *unstream* together

Taken together, the forms of good consumers and good producers ensure that through a sequence of transformations we can bring the *stream* and *unstream* combinators together.

Let us take a hypothetical example. Because we require the consumer to be applied directly to the producer then the producer function is necessarily fully applied.

$$consumer\ (producer\ x\ y)\ z$$

We can unfold definitions to expose the *stream* and *unstream* combinators. If the consumer or producer are defined in terms of other good producers or consumers then we will need to unfold these.

[4] In some special cases it may be profitable to duplicate a simple producer that does little allocation, such as *enumFromTo*, to enable additional fusion.

In general this will give us a number of nested lambda abstractions and **let** bindings.

$$\ldots (\mathit{stream}\ ((\lambda x\ y \rightarrow \textbf{let}\ \ldots \\ \textbf{in}\ \mathit{unstream}\ (\ldots) \\ )\ x\ y))\ldots$$

We now need to push the *stream* inwards. We can push the application of *stream* through **let** bindings. Since we know the producer is fully applied, we can β-reduce to eliminate each lambda abstraction. To preserve sharing this may introduce additional **let** bindings.

$$\ldots (\textbf{let}\ \ldots \\ \textbf{in}\ \mathit{stream}\ (\mathit{unstream}\ (\ldots)))\ldots$$

At this point we have a direct application of *stream* to *unstream* applied to some expression. We can now apply the *stream*/*unstream* fusion rule.

As explained in Section 3.8.2, we are able to use the simple version of the the *stream*/*unstream* fusion rule because by construction of the library, we have a context where all functions that manipulate streams are part of the library and those functions follow the rules.

Having applied the *stream*/*unstream* fusion rule, we are left with the application of a stream consumer term to a stream producer term. More generally, if we start with an applicative term of good consumers and good producers then we will end up with a compound applicative term of stream consumers and producers.

Note that there are no 'naked' stream inputs or results, that is, stream values only exist as intermediate types in the applicative term of stream consumers and producers. To see that this is the case, recall that the exported library functions do not have stream inputs or results, and thus all combinations have the same property. The property is also preserved by the fusion rule.

4.2.4 Streams embedded in higher-order or compound types

The definition of good producers and good consumers that we have given so far exclude the case of streams embedded in higher-order or compound types. Notably *concatMap* is not a good consumer in its first argument under our definition of good consumer. Similarly, functions like *unzip* and *partition* that produce pairs of lists cannot be good producers under our definition of good producer.

It is no coincidence that our definition of good consumers excludes *concatMap*. While the first phase of applying *stream*/*unstream* fusion with *concatMap* is straightforward, the second phase of optimising the resulting stream function is problematic. There are certainly cases where the intermediate allocations can be eliminated, however no general method is currently known. We return to this issue in Section 4.8.3.

In the simple case of functions that produce or consume pairs of lists it may be possible to transform the definitions such that they do satisfy the good producer/consumer conditions. In the case of consumers, simple currying would usually suffice. An example that could not be so easily transformed would be a function that performs some dynamic computation on the pair before extracting the stream components. Similarly, a function producing a pair of lists, where each list is produced by *stream*, might possibly be transformed into a pair of good producers.

Consider the function *partition* which produces a pair of lists

$$partition :: (a \to Bool) \to [a] \to ([a], [a])$$

Denotationally, this function is equal to

$$partition\ p\ xs = (filter\ p\ xs, filter\ (not \circ p)\ xs)$$

However the standard definition is

$$\begin{array}{l} partition\ p\ xs = foldr\ select\ ([\,], [\,])\ xs \\ \quad \textbf{where} \\ \quad\quad select\ x\ tfs = \textbf{let}\ (ts, fs) = tfs \\ \quad\quad\quad\quad\quad\quad\quad \textbf{in if}\ p\ x\ \textbf{then}\ (x : ts,\quad fs) \\ \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \textbf{else}\ (\quad ts, x : fs) \end{array}$$

This definition relies on lazy evaluation and sharing so that it only has to use the predicate p once per element of the input list. While the first definition can be converted into a pair of good producers, the second definition cannot and therefore cannot be made into a good producer.

4.3 Optimising stream functions

The next few sections (4.3–4.6) are concerned with how we consume streams. The challenge is to optimise the consumption of streams. Specifically, the goal is to eliminate all of the step constructors (*Yield*, *Skip* and *Done*), and all of the constructors used to represent the stream state.

A useful analogy is to see a stream as a static description of a sequence, as a little program. We can straightforwardly interpret this description, incurring the costs of representing the program steps and dynamic program states. Alternatively we can compile the description and not pay any interpretation overhead. Of course we consume a stream so as to produce some other result, so when compiling a stream we intertwine it with the code of the consumer.

Later in this section we will make the 'little program' analogy semi-formal by defining a restricted form of streams that are equivalent to a kind of state machine. We will incorporate these conditions into the requirements on good producers. We will use this state machine form to help make the argument that we can always optimise the consumption of streams so that the various constructors are eliminated.

Example

Let us start by illustrating the kind of transformations we are concerned with in this section. Consider this simple example of a good consumer applied to a good producer

$$sum\ [0\,..\,9]$$

The first phase (Section 4.2) transforms this so that we have a stream consumer applied directly to a stream producer.

$$sum_s\ (enumFromTo_s\ 0\ 9)$$

The definition of sum_s is

$$\begin{array}{l} sum_s\ (Stream\ next\ s) = go\ 0\ s \\ \quad \textbf{where} \\ \quad\quad go\ a\ s = \textbf{case}\ next\ s\ \textbf{of} \\ \quad\quad\quad\quad\quad\quad Done \quad\quad\quad\quad \to a \\ \quad\quad\quad\quad\quad\quad Skip \quad\ s' \to go\ a \quad\quad\quad\ s' \\ \quad\quad\quad\quad\quad\quad Yield\ x\ s' \to go\ (a + x)\ s' \end{array}$$

while $enumFromTo_s$ is defined as

$$
\begin{array}{l}
enumFromTo_s\ n\ m = Stream\ next\ n \\
\quad \textbf{where} \\
\quad\quad next\ n \mid n \leqslant m \quad\ = Yield\ n\ (n+1) \\
\quad\quad\quad\quad\ \mid otherwise = Done
\end{array}
$$

A straightforward evaluation of $sum_s\ (enumFromTo_s\ 0\ 9)$ will allocate a *Yield* constructor and a constructor for the *Int* stream state for each iteration of the stream. An optimised version of this composition looks like

$$
\begin{array}{l}
\quad sum_s\ (enumFromTo_s\ 0\ 9) \\
= \\
\quad go\ 0\ 0 \\
\quad\quad \textbf{where} \\
\quad\quad\quad go\ a\ n \mid n \leqslant 9 \quad\ = go\ (a+n)\ (n+1) \\
\quad\quad\quad\quad\quad\ \mid otherwise = a
\end{array}
$$

Here we have fused the stream description into a single recursive function that intertwines the code of the producer and consumer. The *Yield* constructors have been eliminated. What was the stream state is now passed directly as a parameter of the recursive function. We call this *stream consumer/producer fusion*.

4.3.1 Scope of the problem

Before looking at the details of how we perform transformations such as the one above, we should consider what is the most general case that we must deal with. In the first phase we were able to consider each application of a good consumer to a good producer independently. In this phase we must consider a slightly larger unit at a time, consisting of stream producers, transformers and consumers.

We distinguish *stream transformers* as a special class. A stream transformer is not merely the conjunction of being a stream producer and a stream consumer. Stream transformers are special in that they construct a new stream directly in terms of existing streams, rather than consuming the input streams. More precisely, a stream transformer defines the new stepper function in terms of the old stepper functions and the new initial state in terms of the old initial states.

To illustrate the difference, consider these two functions

$$
\begin{aligned}
&plus1_s :: Stream\ Int \to Stream\ Int \\
&plus1_s = map_s\ (+1) \\
&sort_s :: Stream\ Int \to Stream\ Int \\
&sort_s = unHeap_s \circ mkHeap_s \\
&\quad \textbf{where} \\
&\qquad mkHeap_s :: Stream\ Int \to Heap\ Int \\
&\qquad unHeap_s :: Heap\ Int \quad \to Stream\ Int
\end{aligned}
$$

While $sort_s$ is obviously a stream consumer and a stream producer, we do not classify it as a stream transformer. It consumes an input stream to produce an intermediate heap data structure and then constructs a stream producer which yields the elements of the intermediate heap. On the other hand, map_s directly constructs a new stream – initial state and stepper function – out of an existing stream's initial state and stepper function. It has no recursion to dynamically consume its input stream.

The reason stream transformers are important is that because they let the user build bigger and more complex streams, they make the task of optimising the consumption of streams more difficult. Instead of being able to consider each stream consumer/producer application in isolation, we must consider the consumption of streams that are be built up through a combination of transformers and producers.

In the general case, the first phase transforms applicative terms consisting of good consumers and producers into applicative terms consisting of stream producers, transformers and consumers. Typically this gives a term that produces and consumes a single compound stream. For example this term that manipulates lists

$$sum\ (zipWith\ (\times)\ [0\,..\,9]\ [10\,..\,19])$$

gets transformed into an equivalent term that manipulates streams

$$sum_s\ (zipWith_s\ (\times)\ (enumFromTo_s\ 0\ 9)\ (enumFromTo_s\ 10\ 19))$$

Note that since $zipWith_s$ is a stream transformer, we have a single stream term

$$zipWith_s\ (\times)\ (enumFromTo\ 0\ 9)\ (enumFromTo\ 10\ 19)$$

We must consider this stream term as one unit when explaining how to optimise its consumption by sum_s.

On the other hand, some combinations of good consumers and producers give rise to multiple stream terms that are best considered independently. Consider this term that uses lists

$$take\ 10\ (sort\ (take\ 100\ xs))$$

If we have list *sort* defined as

$$sort = unstream \circ unHeap_s \circ mkHeap_s \circ stream$$

then the above term gets transformed into

$$unstream\ (take_s\ 10\ (unHeap_s\ (mkHeap_s\ (take_s\ 100\ (stream\ xs)))))$$

Because $unHeap_s$ and $mkHeap_s$ are not stream transformers, we have two independent stream-producing terms. There is $take_s\ 100\ (stream\ xs)$ which gets consumed by $mkHeap_s$. There is also $take_s\ 10\ (unHeap_s\ (\ldots))$ which gets consumed by *unstream*.

So the general form that we must deal with is an applicative term consisting of a stream consumer applied to a tree of stream transformers, with stream producers at the leaves. As mentioned previously in Section 4.2.3, the way we defined good consumers and producers guarantees that there are no 'naked' streams as inputs or as the result; there is always a stream consumer at the top and stream producers at the bottom, even if it is merely *unstream* or *stream*.

4.3.2 Overview of the transformations

Consider again the example term

$$sum_s\ (zipWith_s\ (\times)\ (enumFromTo_s\ 0\ 9)\ (enumFromTo_s\ 10\ 19))$$

and its depiction as an expression tree in frame (a) in Figure 4.1. As stated before, the general form is a tree-shaped term of stream consumers, transformers and producers. Within such terms, each application of a consumer or transformer is a point where we must eliminate intermediate *Step* constructors. The top level application of a stream consumer has the additional challenge of eliminating the constructors used for the stream state. Our overall optimisation argument relies on eliminating all these constructors.

Our approach to optimising such terms is first to fuse the stream producers and transformers into a single (potentially rather complicated) stream producer (frame (b) of Figure 4.1). The second and final step is to fuse the top level stream consumer with the remaining stream producer (frame (c) of Figure 4.1).

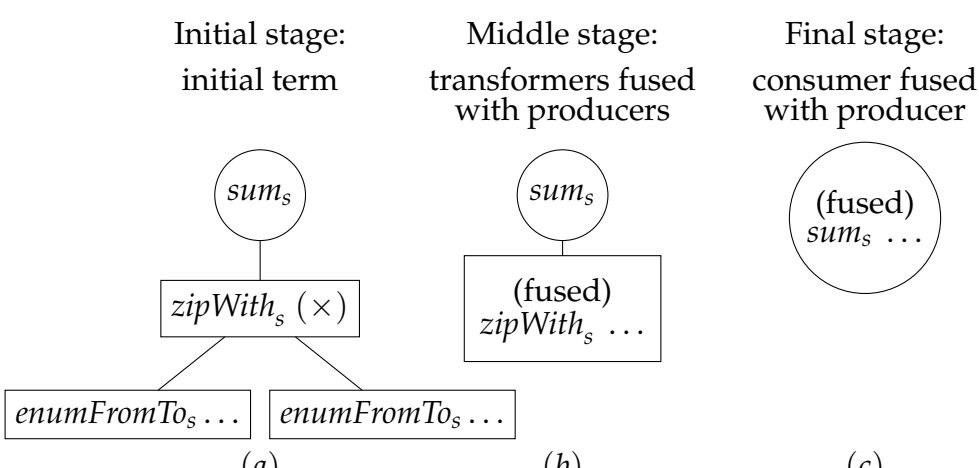


Figure 4.1: Optimisation stages

These two steps achieve similar results in terms of eliminating allocations; we distinguish them because the code transformations involved are different.

Note that an alternative approach would be to fuse the stream consumers with the stream transformers to give a more complicated stream consumer, before fusing this consumer with the remaining stream producers. Will not consider this alternative approach any further as our approach appears to be easier to explain.

With our example term, the first step of fusing $zipWith_s$ with the two producers gives us the following single stream producer.

$$
\begin{array}{l}
\quad zipWith_s\ (\times)\ (enumFromTo_s\ 0\ 9)\ (enumFromTo_s\ 10\ 19) \\
= \\
\quad Stream\ next\ (0, 10, Nothing) \\
\quad\quad \textbf{where} \\
\quad\quad\quad next\ (n, m, Nothing) \mid n \leqslant 9 \quad = Skip \qquad\qquad\quad (n+1, m,\ Just\ n) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid otherwise = Done \\
\quad\quad\quad next\ (n', m, Just\ n) \quad \mid m \leqslant 19 \quad = Yield\ (n \times m)\ (n', m+1, Nothing) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid otherwise = Done
\end{array}
$$

We will cover the details of this transformation step, which we call *stream transformer/producer fusion*, in Section 4.5. Note that we have eliminated the intermediate *Step* constructors between $zipWith_s$ and the two $enumFromTo_s$ stream producers. We are left with a single stream producer. It is not uncommon for these single stream producers to have quite complicated stepper functions and complicated stream states with several modes.

The last step is to fuse the above stream producer with the top level stream consumer sum_s

$$
\begin{array}{l}
\quad sum_s\ (Stream\ next\ (0, 10, Nothing)) \\
\quad\quad \textbf{where} \\
\quad\quad\quad next \ldots \\
= \\
\quad go\_Nothing\ 0\ 0\ 10 \\
\quad\quad \textbf{where} \\
\quad\quad\quad go\_Nothing\ a\ n\ m \quad \mid n \leqslant 9 \quad = go\_Just\ a\ (n+1)\ m\ n \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\ \mid otherwise = a \\
\quad\quad\quad go\_Just \quad\quad a\ n'\ m\ n \mid m \leqslant 19 \quad = go\_Nothing\ (a + n \times m)\ n'\ (m+1) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\ \mid otherwise = a
\end{array}
$$

This transformation step eliminates both the step constructors and also the stream state constructors, which in this example, were $(\_, \_, Nothing)$ and $(\_, \_, Just\ \_)$. As mentioned previously, we call this transformation *stream consumer/producer fusion*. We will cover the details of stream consumer/producer fusion in Section 4.6.

The stream transformer/producer fusion step relies at its core on the case-of-case transformation. The stream consumer/producer fusion step relies both on the case-of-case transformation and on a transformation known as *call pattern specialisation* (Peyton Jones, 2007).

While it is reasonably clear in the example above that we can concoct and apply suitable transformations to eliminate the allocations, for the general case we need to explain exactly what transformations should be applied and give an argument for why we can always apply these transformations.

4.3.3 Overview of the argument

The approach we take is to impose constraints on the way we write functions that produce or consume streams. We will argue that these constraints ensure that the various transformations are applicable. We will also argue that the sequence of transformations reduces allocations (Section 4.7).

The argument that our transformations are applicable is as follows:

- We require that all the stream producers and stream transformers satisfy the stream producer constraints (Section 4.5.4).

- We argue that the application of a stream transformer to a stream producer, where both satisfy the stream producer constraints, can be fused to give a single stream producer that also satisfies the stream producer constraints (Section 4.5.5).

- Since the producer constraints are preserved by fusing transformers with producers then by induction we can fuse all the transformers and producers into a single stream that satisfies the producer constraints.

- We require that all the stream consumers satisfy the stream consumer constraints (Section 4.6.3).

- Finally we argue that a stream producer and a stream consumer, where both satisfy the respective constraints, can be fused so that the stream's state is encoded as control-flow rather than as dynamically allocated data (Section 4.6.4–4.6.5).

As mentioned, we use an inductive argument to explain why we can fuse all the transformers and producers into a single producer. The induction hypothesis is simply that the producer satisfies the producer constraints. As is typical with inductive arguments, some parts of the induction hypothesis are there because we need them to be there at the end, i.e. for the stream consumer/producer fusion, while others are there just to make the induction hypothesis strong enough for the induction step go through, i.e. for the stream transformer/producer fusion.

4.3.4 State shapes in the allocation argument

The specific steps above that remove allocations are the transformer/producer fusion and consumer/producer fusion steps. Each removes exactly one allocation per sequence element. These are the only points where the genuine savings due to stream fusion accrue.

There is another class of allocations that are eliminated as part of the stream consumer/producer fusion step. Eliminating this class of allocations does not represent an overall saving however because these are allocations that we introduced ourselves by using stream versions of functions rather than ordinary list versions. Recall from Section 1.4.3 that to increase the expressiveness of stream functions we use multiple state shapes represented by patterns of data constructors. The allocations for these extra data constructors pose a problem because we cannot afford for stream versions of functions to use any more allocations than their list counterparts. The solution we propose in Section 1.4.3

is for stream functions to be able to use these additional allocations 'for free' by us promising somehow to eventually eliminate all the data constructors used to represent the state shapes.

The call pattern specialisation transformation can eliminate the data constructors we are interested in, subject to a number of preconditions on the stream stepper functions. We can thus construct an overall argument as follows:

- We assign a set of state shapes to each stream producer.
- Some of the stream producer constraints are specified in terms of these state shapes.
- For the inductive step where we fuse stream transformers with stream producers:
  - we assign the state shapes of the fused producer to be the composition of the state shapes of the constituent transformer and producers;
  - the composition is such that eliminating the allocations for the composed state shapes is equivalent to separately eliminating the allocations for the state shapes of the original transformer and producers.
- Finally in the consumer/producer fusion step we argue that the combination of the producer and consumer constraints guarantees that we can apply call pattern specialisation and eliminate all the constructors used to represent the state shapes.

Thus overall, if stream producers, transformers and consumers satisfy their respective constraints then we can eliminate the allocations used by the representation of stream state shapes.

When writing stream functions we think of the state shapes as arising from the syntax of the definitions we write. For the purposes of the allocation argument however we consider the state shapes to be assigned to the producer and that the producer follows the rules. This view is helpful because in the inductive step we appear to have a choice about what the state shapes of the fused producer ought to be. We choose the option that makes the allocation accounting work out. This choice of state shapes then entails some further optimisation of the fused stream's stepper function to bring it into compliance with the producer constraints for the new set of state shapes (see Section 4.5.5).

4.4 State machine view

The stream producers that satisfy the producer constraints have the property that they can be viewed as a kind of state machine. This state machine view is a useful abstraction. The state machines can be composed. The state machine of a fused stream is the composition of the state machines of its components. It is also possible to view the way a stream is fused with a consumer as 'compiling' the state machine corresponding to the stream.

The state machine view is also useful as a pedagogical tool to help explain the behaviour of streams and how to write them. In particular it lets us visualise streams as simple state machine diagrams.

4.4.1 The basic correspondence

We can view a stream as a state machine by considering the stream's stepper function as a state transition function. The internal stream states correspond to the state machine states. The initial stream state denotes the initial state in the state machine. The transitions of the state machine are labelled with *Step* actions. The final state in the state machine is reached via a *Done* transition.

Streams are of course not restricted to a finite number of internal states. The corresponding state machines therefore cannot be finite state machines. Nevertheless we can present these state machines as graphs with finite numbers of nodes and edges. Instead of graph nodes representing individual states, they represent *state shapes* which are parametrised collections of states. The transitions between state shapes are similarly parametrised.

As we described in Section 1.4.3, many stream's internal states are divided into a number of different 'modes'. In Chapter 3 we took advantage of these modes (and transitions between modes) to help us structure fixpoint induction properties, using different properties for different modes. Recall for example (Section 3.9.6) that zip_s has two modes and that it alternates between them each time it consumes an element from one of the two input streams. In the state machine view, state machine graph *nodes* correspond exactly to stream *modes*.

To illustrate the state machine perspective, consider a very simple stream that enumerates the integers from 0 to 9. Like all streams, it consists of a stepper function and an initial state

Stream next 0

The state is an integer, with 0 as the initial state.

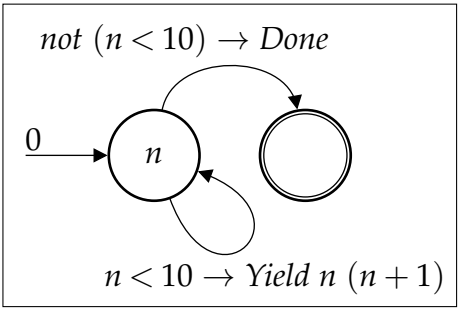


Figure 4.2: State machine for a stream which enumerates $0 \ldots 9$

The *next* function is

$$\begin{aligned} \mathit{next}\ n \mid n < 10 \quad &= \mathit{Yield}\ n\ (n+1) \\ \mid \mathit{otherwise} &= \mathit{Done} \end{aligned}$$

As we consume this stream step by step, the internal stream state will proceed through the values $0 \ldots 9$ until finally we get to *Done*. This simple stream has just a single mode. The corresponding state machine diagram is given in Figure 4.2. The state machine has states for each value $0 \ldots 9$ and also a terminal state. The key feature of this state machine description is that we group the states $0 \ldots 9$ together into a single node parametrised by a variable n.

The *Done* transition leads to the terminal state which is a separate node in the state machine diagram. The primary node has two outgoing transitions

$$\begin{aligned} n < 10 \quad &\to \mathit{Yield}\ n\ (n+1) \\ \mathit{not}\ (n < 10) &\to \mathit{Done} \end{aligned}$$

These transitions correspond to a stream's stepper function returning *Done* or returning *Yield* with a new stream state. The transitions are labelled with their *Step* constructor, which in the case of *Skip* and *Yield* give the new state variable(s) for the target node. The *Step* labels are the actions of the state machine; they are the observations that external code makes as it unfolds the stream. The transitions are also guarded with predicates. Since streams are deterministic, in the corresponding state machines, the predicates on the transitions do not overlap.

4.4.2 Stream transformers as state machines

In addition to simple stream producers, we can also view stream transformers as state machines.

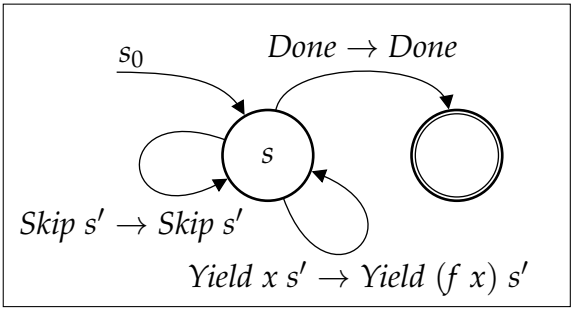


Figure 4.3: State machine view of map_s stream function

Consider, for example, map_s

$$
\begin{aligned}
&map_s :: (a \rightarrow b) \rightarrow Stream\ a \rightarrow Stream\ b \\
&map_s\ f\ (Stream\ next_0\ s_0) = Stream\ next\ s_0 \\
&\quad \textbf{where} \\
&\quad\quad next\ s = \textbf{case}\ next_0\ s\ \textbf{of} \\
&\quad\quad\quad\quad\quad\quad Done \rightarrow Done \\
&\quad\quad\quad\quad\quad\quad Skip\ s' \rightarrow Skip\ s' \\
&\quad\quad\quad\quad\quad\quad Yield\ x\ s' \rightarrow Yield\ (f\ x)\ s'
\end{aligned}
$$

The corresponding state machine diagram is given in Figure 4.3. Like the previous example, it has two nodes: the primary node and a terminal node. The primary node has three outgoing transitions. The predicates in these transitions are *Step* data constructor patterns. The three transitions correspond to demanding a single step from the input stream and then depending on whether it was *Done*, *Skip* or *Yield*, making a transition into a new state.

Note that whenever we have a state that can transition by accepting a *Step* from an input stream then we must have transitions for each of *Done*, *Skip* and *Yield*. We do not get to choose the input *Step*; we must be prepared to accept any possible *Step* (if necessary using a transition to an error state).

The $filter_s$ function is structurally very similar

$$
\begin{aligned}
&filter_s :: (a \rightarrow Bool) \rightarrow Stream\ a \rightarrow Stream\ a \\
&filter_s\ p\ (Stream\ next_0\ s_0) = Stream\ next\ s_0 \\
&\quad \textbf{where} \\
&\quad\quad next\ s = \textbf{case}\ next_0\ s\ \textbf{of} \\
&\quad\quad\quad\quad\quad\quad Done \rightarrow Done \\
&\quad\quad\quad\quad\quad\quad Skip\ s' \rightarrow Skip\ s' \\
&\quad\quad\quad\quad\quad\quad Yield\ x\ s' \mid p\ x \rightarrow Yield\ x\ s' \\
&\quad\quad\quad\quad\quad\quad\quad\quad\ \mid otherwise \rightarrow Skip\ s'
\end{aligned}
$$

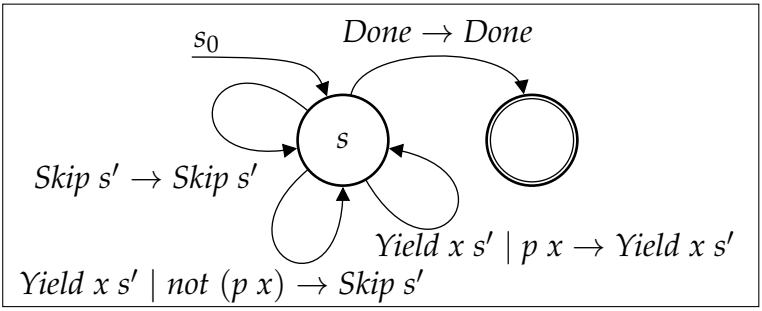


Figure 4.4: State machine view of $filter_s$ stream function

The corresponding state machine (in Figure 4.4) again has one primary state node. It has one input stream and the transitions for *Yield* are predicated on the yielded values

$$\begin{array}{ll} Yield\ x\ s' \mid p\ x & \rightarrow Yield\ x\ s' \\ Yield\ x\ s' \mid not\ (p\ x) & \rightarrow Skip \quad s' \end{array}$$

Alternative general syntax for transition predicates

The above syntax for transition predicates is a convenient shorthand that leaves the choice of the input stream implicit. It is suitable when there is just one input stream and even with multiple input streams there is usually no problem of ambiguity. We will use this shorthand syntax throughout this chapter since all the transition predicates we need to use are of the simple variety.

In general however the state machine transition predicates can include multiple pattern bindings and predicate expressions. While not a problem for the examples in this chapter, it would be useful to have a syntax capable of expressing the general case. A suitable syntax is the Haskell language extension known as *pattern guards* (Erwig and Peyton Jones, 2001). This allows a sequence of expressions matched against patterns and Boolean predicates. Variables bound in patterns can be used in subsequent predicates, expressions and the result.

$$\langle pat_1 \rangle \leftarrow \langle exp_1 \rangle, \langle pred_2 \rangle, \langle pat_3 \rangle \leftarrow \langle exp3 \rangle, \ldots \quad \rightarrow \langle result \rangle$$

The above transitions for $filter_s$ would be written as

$$\begin{array}{ll} Yield\ x\ s' \leftarrow next_0\ s, p\ x & \rightarrow Yield\ x\ s' \\ Yield\ x\ s' \leftarrow next_0\ s, not\ (p\ x) & \rightarrow Skip \quad s' \end{array}$$

In the general case, for each transition, at most one pattern may match one of the *Step* constructors from one input stream. As mentioned previously, the state machines are deterministic so the transition predicates may not overlap.

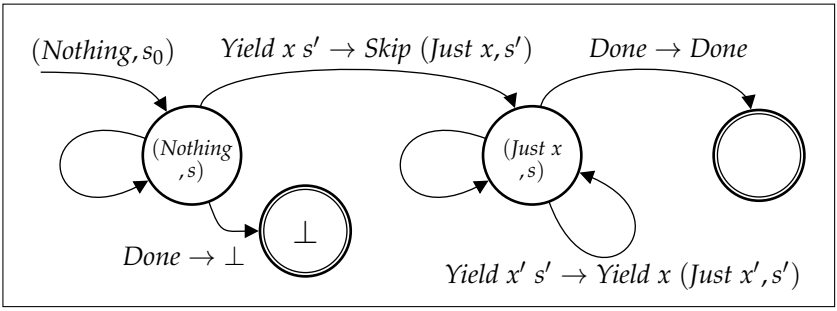


Figure 4.5: State machine view of $init_s$ stream function

4.4.3 State machines with multiple state shapes

Let us look at an example with a non-trivial stream state, and correspondingly with multiple state shapes. Recall from Section 1.4.3 that we derived the following definition for the stream version of the *init* function.

$$
\begin{aligned}
&init_s :: Stream\ a \to Stream\ a \\
&init_s\ (Stream\ next_0\ s_0) = Stream\ next\ (Nothing, s_0) \\
&\quad \textbf{where} \\
&\qquad next\ (Nothing, s) = \textbf{case}\ next_0\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad Done\ \ \ \ \ \ \to error\ \texttt{"init: empty stream"} \\
&\qquad\qquad\qquad\qquad Skip\ \ \ s' \to Skip\ (Nothing, s') \\
&\qquad\qquad\qquad\qquad Yield\ x\ s' \to Skip\ (Just\ x, \ \ s') \\
&\qquad next\ (Just\ x, s)\ \ \ = \textbf{case}\ next_0\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad Done\ \ \ \ \ \ \to Done \\
&\qquad\qquad\qquad\qquad Skip\ \ \ s' \to Skip\ \ \ (Just\ x, \ s') \\
&\qquad\qquad\qquad\qquad Yield\ x'\ s' \to Yield\ x\ (Just\ x', s')
\end{aligned}
$$

The corresponding state machine in Figure 4.5 has two primary nodes (in addition to a terminal error node and the usual terminal node). The two state shapes are $(Nothing, s)$ and $(Just\ x, s)$. These state shapes statically partition the reachable state space of $(Maybe\ a, s)$.

4.4.4 Multiple input streams

The state machine view works for stream transformers that consume multiple streams. The transitions simply have to indicate which input stream they refer to.

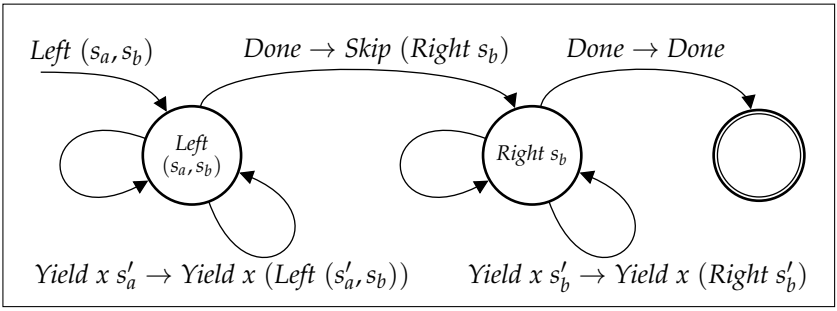


Figure 4.6: State machine view of $append_s$ stream function

Consider the $append_s$ function

$$
\begin{aligned}
&append_s :: Stream\ a \to Stream\ a \to Stream\ a \\
&append_s\ (Stream\ next_a\ s_a)\ (Stream\ next_b\ s_b) = Stream\ next\ (Left\ (s_a, s_b)) \\
&\quad \textbf{where} \\
&\qquad next\ (Left\ (s_a, s_b)) = \textbf{case}\ next_a\ s_a\ \textbf{of} \\
&\qquad\qquad\qquad\qquad Done \to Skip\ (Right\ s_b) \\
&\qquad\qquad\qquad\qquad Skip\ s'_a \to Skip\ (Left\ (s'_a, s_b)) \\
&\qquad\qquad\qquad\qquad Yield\ x\ s'_a \to Yield\ x\ (Left\ (s'_a, s_b)) \\
&\qquad next\ (Right\ s_b) = \textbf{case}\ next_b\ s_b\ \textbf{of} \\
&\qquad\qquad\qquad\qquad Done \to Done \\
&\qquad\qquad\qquad\qquad Skip\ s'_b \to Skip\ (Right\ s'_b) \\
&\qquad\qquad\qquad\qquad Yield\ x\ s'_b \to Yield\ x\ (Right\ s'_b)
\end{aligned}
$$

The corresponding state machine (in Figure 4.6) has two main nodes: the state shapes $Left\ (s_a, s_b)$ and $Right\ s_b$.

4.4.5 An approach to formalising the state machine view

While in this chapter we present only semi-formal arguments, it is useful and interesting to draw connections with theory that one might use to formalise the arguments.

A stream defines a deterministic *labelled state transition system* (Arnold, 1994). The state space is the type of the stream's internal state. The grouping of states into parametrised nodes can be formalised as *state values* which is a state vector associated with each state. The transitions are given by the stream's stepper function. The transitions are labelled with the step action: done, skip or yield with a value.

The semantics of the stream is given by the *trace* of the transition system. Since the transition systems are deterministic there is only a single trace. If the skip action is considered as the hidden τ action then stream equivalence can be described by *weak trace equivalence*.

If we take a stream that is the composition of a stream transformer with a stream producer, its transition systems can be described by the *synchronous product* of the transition systems of the two component streams. Specifically, we can write a *synchronisation constraint* that ensures that the stream producer makes a step only when the stream transformer is ready to accept the same step. This description also explains why producers that yield in multiple places leads to the duplication of states (see Section 4.5.8). In the *free product* of transition systems we get the binary product of states; the synchronous product restricts the states to a subset of the full product.

4.5 Stream transformer/producer fusion

Stream transformer/producer fusion is the transformation whereby the application of a single stream transformer to one or more stream producers is fused to give a single optimised stream producer. This section contributes a component of the overall argument outlined previously in Section 4.3.3: that this transformation always gives a stream producer that satisfies the constraints. That the transformation reduces allocations is covered in Section 4.7.9.

In the overview in Section 4.3.2 we gave the following example of stream transformer/producer fusion

$$
\begin{array}{l}
\quad \mathit{zipWith}_s\ (\times)\ (\mathit{enumFromTo}_s\ 0\ 9)\ (\mathit{enumFromTo}_s\ 10\ 19) \\
= \\
\quad \mathit{Stream}\ \mathit{next}\ (0, 10, \mathit{Nothing}) \\
\qquad \textbf{where} \\
\qquad\quad \mathit{next}\ (n, m, \mathit{Nothing}) \mid n \leqslant 9 \quad = \mathit{Skip} \qquad\qquad (n+1, m,\ \mathit{Just}\ n) \\
\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} = \mathit{Done} \\
\qquad\quad \mathit{next}\ (n', m, \mathit{Just}\ n) \mid m \leqslant 19 \quad = \mathit{Yield}\ (n \times m)\ (n', m+1, \mathit{Nothing}) \\
\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} = \mathit{Done}
\end{array}
$$

In this example we have the stream transformer $\mathit{zipWith}_s$ applied to two instances of the stream producer $\mathit{enumFromTo}_s$. The fused result uses exactly two fewer allocations per sequence element. The *Step* constructors used by each of the two instances of $\mathit{enumFromTo}_s$ have been eliminated.

More generally, the transformation takes a stream transformer and stream producers that satisfy the stream producer constraints and gives an equivalent stream that also satisfies the stream producer constraints. Crucially, the fused stream uses fewer allocation per sequence element, in particular it eliminates the *Step* constructors used by each stream producer.

At the core of stream transformer/producer fusion is the case-of-case transformation. As we saw in Chapter 1, the case-of-case transformation is the core of both *unbuild*/*unfoldr* fusion (Section 1.3.8) and of stream fusion (Section 1.4.2). It is the core transformation in the sense that it is the one that actually removes allocations. While applying the *stream*/*unstream* fusion rule also removes allocations, it only removes allocations that were previously introduced by using fusible definitions of the list functions. It is not illegitimate to take the view the whole stream fusion system is a complex method of rearranging code so as to be able to apply the simple case-of-case optimisation.

4.5.1 A simple example

For the $zipWith_s$ example above, two instances of case-of-case are required: one for each of the two producers. Since the general case allows for stream transformers applied to multiple producers, it is useful to keep the $zipWith_s$ example in mind. To illustrate the details of the transformation however we will use a smaller and simpler example.

$$
\begin{aligned}
&enum10 :: Stream\ Int \\
&enum10 = snoc_s\ (enumFromTo_s\ 0\ 9)\ 10
\end{aligned}
$$

This stream is constructed from a stream that enumerates $0\,..\,9$ and the stream transformer $snoc_s$ that appends a final element[5].

The $enumFromTo_s$ function is defined as before, except that for clarity we rename the local *next* function to indicate its origin and we desugar the Boolean guards into a case expression.

$$
\begin{aligned}
&enumFromTo_s :: Int \to Int \to Stream\ Int \\
&enumFromTo_s\ n\ m = Stream\ next_{enum}\ n \\
&\quad \textbf{where} \\
&\qquad next_{enum}\ n = \textbf{case}\ n \leqslant m\ \textbf{of} \\
&\qquad\qquad\qquad\qquad True\ \to Yield\ n\ (n+1) \\
&\qquad\qquad\qquad\qquad False \to Done
\end{aligned}
$$

[5]The function *snoc* is so named, as a pun on the name *cons*, because it is the reverse of *cons*.

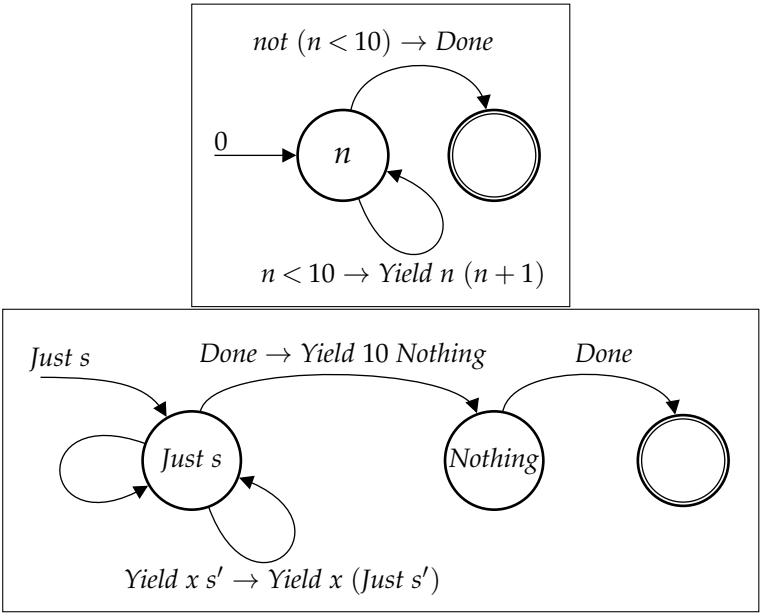


Figure 4.7: State machine view of $enumFromTo_s$ 0 9 and $snoc_s$

The $snoc_s$ function is defined as

$$
\begin{aligned}
&snoc_s :: Stream\ a \to a \to Stream\ a \\
&snoc_s\ (Stream\ next_0\ s)\ y = Stream\ next_{snoc}\ (Just\ s) \\
&\quad \textbf{where} \\
&\qquad next_{snoc}\ (Just\ s) = \textbf{case}\ next_0\ s\ \textbf{of} \\
&\qquad\qquad\qquad\qquad Done \to Yield\ y\ Nothing \\
&\qquad\qquad\qquad\qquad Skip\ s' \to Skip\ (Just\ s') \\
&\qquad\qquad\qquad\qquad Yield\ x\ s' \to Yield\ x\ (Just\ s') \\
&\qquad next_{snoc}\ Nothing = Done
\end{aligned}
$$

The corresponding state machines are given in Figure 4.7.

The behaviour of *snoc* is first to yield each element of its input stream. It yields the additional final element when the input stream finally produces *Done*. Of course after this there is nothing to do but to produce *Done*. A stream cannot produce both *Yield* and *Done* in a single step however, so it must move into another state in which it produces *Done*. Thus this stream has two modes *Just s* and *Nothing*.

Let us start with the application of the stream transformer $snoc_s$ to the stream producer $enumFromTo_s$ 0 9

$$snoc_s\ (enumFromTo_s\ 0\ 9)\ 10$$

We can unfold the definition of $enumFromTo_s$ to expose the *Stream* constructor. We also get the stream's stepper function as an associated local definition.

$$
\begin{aligned}
& snoc_s\ (enumFromTo_s\ 0\ 9)\ 10 \\
= \\
& snoc_s\ (Stream\ next_{enum}\ 0)\ 10 \\
& \quad \textbf{where} \\
& \quad\quad next_{enum} = \ldots
\end{aligned}
$$

The next step is to unfold $snoc_s$. The $snoc_s$ function matches on the input stream and exposes a *Stream* result.

$$
\begin{aligned}
& snoc_s\ (Stream\ next_{enum}\ 0)\ 10 \\
& \quad \textbf{where} \\
& \quad\quad next_{enum} = \ldots \\
= \\
& Stream\ next_{snoc}\ (Just\ 0) \\
& \quad \textbf{where} \\
& \quad\quad next_{snoc}\ = \ldots \\
& \quad\quad next_{enum} = \ldots
\end{aligned}
$$

Note that the top level structure of the term consists of a *Stream* term with associated local definitions and that the initial stream state identifies one of the stream state shapes.

We can now optimise this stream's stepper function so that it uses fewer allocations. The first step is simply to unfold the definition of $next_{enum}$ in the 'scrutinee' position of the case expression

$$
\begin{aligned}
next_{snoc}\ (Just\ s)\ = & \textbf{case}\ (\textbf{case}\ s \leqslant 9\ \textbf{of} \\
& \qquad\qquad\qquad True\ \rightarrow Yield\ s\ (s+1) \\
& \qquad\qquad\qquad False \rightarrow Done) \\
& \qquad\qquad \textbf{of} \\
& \qquad\qquad\quad Done\qquad\ \rightarrow Yield\ 10\ Nothing \\
& \qquad\qquad\quad Skip\quad s' \rightarrow Skip\qquad (Just\ s') \\
& \qquad\qquad\quad Yield\ x\ s' \rightarrow Yield\ x\quad (Just\ s') \\
next_{snoc}\ Nothing = & Done
\end{aligned}
$$

We are now in a position to apply the case-of-case transformation, which leaves us with

$$
\begin{aligned}
next_{snoc}\ (Just\ s)\ = & \textbf{case}\ s \leqslant 9\ \textbf{of} \\
& \qquad\qquad\quad True\ \rightarrow Yield\ s\quad (Just\ (s+1)) \\
& \qquad\qquad\quad False \rightarrow Yield\ 10\ Nothing \\
next_{snoc}\ Nothing = & Done
\end{aligned}
$$

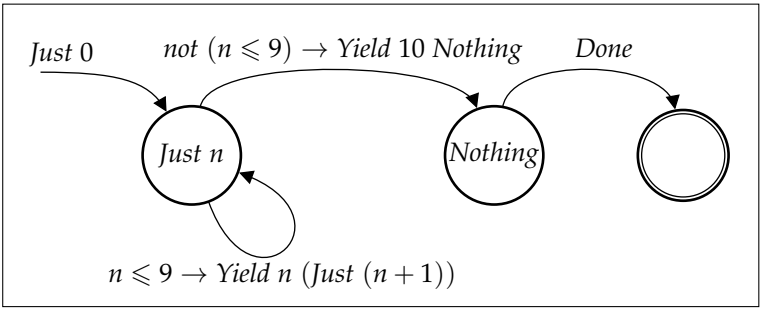


Figure 4.8: State machine view of fused $snoc_s\ (enumFromTo_s\ 0\ 9)\ 10$

This eliminates the allocation of the *Step* constructors in each branch of the inner case expression. The corresponding state machine for this final fused version is given in Figure 4.8. It is instructive to compare this state machine with those from Figure 4.7.

4.5.2 The basic case-of-case transformation

Note that strictly speaking, 'case-of-case' refers only to the first of two steps that make up the transformation. If we were to more closely follow the terminology of Santos (1995) then we should more correctly refer to the overall transformation as the combination of the case-of-case transformation followed by *case reduction*. Since the two are so often used together and the main purpose of case-of-case is to expose opportunities for case reduction, then we and other authors tend to refer only to case-of-case, leaving implicit the fact that case reduction is also involved.

We typically present the combination of the case-of-case transformation followed by case reduction as a single step. To understand and to generalise the transformation, it is useful to look at the two component transformations separately. We start with the following situation

$$
\begin{array}{l}
\textbf{case}\ (\textbf{case}\ s \leqslant 9\ \textbf{of} \\
\qquad\quad \mathit{True}\ \to \mathit{Yield}\ s\ (s+1) \\
\qquad\quad \mathit{False} \to \mathit{Done}) \\
\textbf{of} \\
\quad \mathit{Done} \qquad\quad \to \mathit{Yield}\ 10\ \mathit{Nothing} \\
\quad \mathit{Skip}\quad s' \to \mathit{Skip} \qquad (\mathit{Just}\ s') \\
\quad \mathit{Yield}\ x\ s' \to \mathit{Yield}\ x \quad (\mathit{Just}\ s')
\end{array}
$$

The first step is to push the outer case expression through the inner one, duplicating it into each branch. This is the step known properly as the case-of-case

transformation (Santos, 1995, Section 3.5.2). Consider, for a moment, just one of the branches of the inner case expression.

$$
\begin{array}{l}
\textbf{case}\ (\textbf{case}\ s \leqslant 9\ \textbf{of} \\
\qquad \mathit{True} \to \mathit{Yield}\ s\ (s+1) \\
\qquad \ldots\ ) \\
\textbf{of} \\
\quad \ldots \\
\quad \mathit{Yield}\ x\ s' \to \mathit{Yield}\ x\ (\mathit{Just}\ s')
\end{array}
$$

Pushing the outer case through and into this branch gives us

$$
\begin{array}{l}
\textbf{case}\ s \leqslant 9\ \textbf{of} \\
\quad \mathit{True} \to \textbf{case}\ \mathit{Yield}\ s\ (s+1)\ \textbf{of} \\
\qquad\qquad \ldots \\
\qquad\qquad \mathit{Yield}\ x\ s' \to \mathit{Yield}\ x\ (\mathit{Just}\ s') \\
\quad \ldots
\end{array}
$$

Of course now we have a situation where we have a "case expression scrutinising a known constructor" (Santos, 1995, Section 3.3.1). The second step is to reduce the case-of-known-constructor. In this example we have the term $\mathit{Yield}\ s\ (s+1)$ matched against the pattern $\mathit{Yield}\ x\ s'$ which gives us the substitution $x := s,\ s' := s+1$. The result then, is the term in the *Yield* branch of outer case, but with the substitution applied

$$
\begin{array}{l}
\quad (\mathit{Yield}\ x\ (\mathit{Just}\ s'))\ [x := s,\ s' := s+1] \\
= \\
\quad \mathit{Yield}\ s\ (\mathit{Just}\ (s+1))
\end{array}
$$

And this is the final result term in the *True* branch. Taking the same approach for all the branches give the following intermediate result

$$
\begin{array}{l}
\textbf{case}\ s \leqslant 9\ \textbf{of} \\
\quad \mathit{True}\ \to (\mathit{Yield}\ x\ \ (\mathit{Just}\ s'))\ [x := s,\ s' := s+1] \\
\quad \mathit{False} \to (\mathit{Yield}\ 10\ \mathit{Nothing})\ [\{\mathit{empty\ substitution}\}]
\end{array}
$$

Applying the substitutions gives us the final result

$$
\begin{array}{l}
\textbf{case}\ s \leqslant 9\ \textbf{of} \\
\quad \mathit{True}\ \to \mathit{Yield}\ s\ \ (\mathit{Just}\ (s+1)) \\
\quad \mathit{False} \to \mathit{Yield}\ 10\ \mathit{Nothing}
\end{array}
$$

The case-of-case transformation is described in more detail by by Santos (1995, Section 3.5.2) and again by Peyton Jones and Santos (1998, Section 5).

4.5.3 State shape matching

Having considered simple examples, it is tempting to presume that we can always use the case-of-case transformation to fuse multiple stepper functions to give a single stepper function that fits the state machine form. The reality, unfortunately, is not quite so simple. We will illustrate the problem with an example and consider the options to make the system work in the general case.

Consider the stream transformer zip_s (presented previously in Section 3.9.6).

$$
\begin{array}{l}
zip :: Stream\ a \to Stream\ b \to Stream\ (a, b) \\
zip\ (Stream\ next_a\ s_a)\ (Stream\ next_b\ s_b) = Stream\ next\ (s_a, s_b, Nothing) \\
\quad \textbf{where} \\
\quad\quad next\ (s_a, s_b, Nothing) = \textbf{case}\ next_a\ s_a\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Done \to Done \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Skip\ \ s'_a \to Skip\ (s'_a, s_b, Nothing) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Yield\ a\ s'_a \to Skip\ (s'_a, s_b, Just\ a) \\
\quad\quad next\ (s'_a, s_b, Just\ a) = \quad \textbf{case}\ next_b\ s_b\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Done \to Done \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Skip\ \ s'_b \to Skip\ \ \ \ \ \ \ \ (s'_a, s'_b, Just\ a) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad Yield\ b\ s'_b \to Yield\ (a, b)\ (s'_a, s'_b, Nothing)
\end{array}
$$

This transformer has an internal state type of $(s_a, s_b, Maybe\ a)$, where s_a and s_b are the internal state types of the two input streams. The stepper function *next* has two modes, using the state shapes $(\_, \_, Nothing)$ and $(\_, \_, Just\ \_)$. The important point to note about this stepper function is that s_a is only scrutinised in the first mode, and s_b only in the second. The first mode passes s_b around without inspecting it, while the second mode does the same with s'_a.

We will now consider what happens if we fuse zip_s with producers that use multiple state shapes. For the sake of argument we will use the following contrived stream producer.

$$
\begin{array}{l}
toggle_s :: Int \to Stream\ Int \\
toggle_s\ n = Stream\ next\ n \\
\quad \textbf{where} \\
\quad\quad next\ (Left\ \ n) = Yield\ n\ (Right\ (n + 1)) \\
\quad\quad next\ (Right\ n) = Yield\ n\ (Left\ \ (n - 1))
\end{array}
$$

This producer uses an internal state type of $Either\ Int\ Int$ and has two state shapes $Left\ \_$ and $Right\ \_$. If we apply zip_s to two copies of $toggle_s$ and fuse them together, then the resulting stream will have the following internal state type

$$(Either\ Int\ Int,\ Either\ Int\ Int,\ Maybe\ Int)$$

We will expect the stepper function to use the following eight state shapes, which are simply the product of the shapes of the transformer with those of the two producers.

$$
\begin{array}{l}
(\mathit{Left}\ \ \_, \mathit{Left}\ \ \_, \mathit{Just}\ \_) \\
(\mathit{Left}\ \ \_, \mathit{Right}\ \_, \mathit{Just}\ \_) \\
(\mathit{Right}\ \_, \mathit{Left}\ \ \_, \mathit{Just}\ \_) \\
(\mathit{Right}\ \_, \mathit{Right}\ \_, \mathit{Just}\ \_)
\end{array}
$$

$$
\begin{array}{l}
(\mathit{Left}\ \ \_, \mathit{Left}\ \ \_, \mathit{Nothing}) \\
(\mathit{Left}\ \ \_, \mathit{Right}\ \_, \mathit{Nothing}) \\
(\mathit{Right}\ \_, \mathit{Left}\ \ \_, \mathit{Nothing}) \\
(\mathit{Right}\ \_, \mathit{Right}\ \_, \mathit{Nothing})
\end{array}
$$

Let us now check the actual result. We start by combining the stepper function of zip_s with two copies of the stepper function for $toggle_s$.

$$
\begin{array}{lll}
\mathit{next}\ (s_a, s_b, \mathit{Nothing}) = & \textbf{case}\ (\textbf{case}\ s_a\ \textbf{of} & \\
 & \quad \mathit{Left}\ \ n \to \mathit{Yield}\ n\ (\mathit{Right}\ \ (n+1)) & \\
 & \quad \mathit{Right}\ n \to \mathit{Yield}\ n\ (\mathit{Left}\ \ \ (n-1)))\ \textbf{of} & \\
 & \quad \mathit{Done} \to \mathit{Done} & \\
 & \quad \mathit{Skip}\ \ s'_a \to \mathit{Skip}\ (s'_a, s_b, \mathit{Nothing}) & \\
 & \quad \mathit{Yield}\ a\ s'_a \to \mathit{Skip}\ (s'_a, s_b, \mathit{Just}\ a) & \\
\mathit{next}\ (s'_a, s_b, \mathit{Just}\ a) = & \textbf{case}\ (\textbf{case}\ s_b\ \textbf{of} & \\
 & \quad \mathit{Left}\ \ m \to \mathit{Yield}\ m\ (\mathit{Right}\ (m+1)) & \\
 & \quad \mathit{Right}\ m \to \mathit{Yield}\ m\ (\mathit{Left}\ \ \ (m-1)))\ \textbf{of} & \\
 & \quad \mathit{Done} \to \mathit{Done} & \\
 & \quad \mathit{Skip}\ \ s'_b \to \mathit{Skip}\ \ \ (s'_a, s'_b, \mathit{Just}\ a) & \\
 & \quad \mathit{Yield}\ b\ s'_b \to \mathit{Yield}\ (a,b)\ (s'_a, s'_b, \mathit{Nothing}) &
\end{array}
$$

If we now apply the case-of-case transform we obtain the following rather simpler stepper function

$$
\begin{array}{lll}
\mathit{next}\ (s_a, s_b, \mathit{Nothing}) = & \textbf{case}\ s_a\ \textbf{of} & \\
 & \quad \mathit{Left}\ \ n \to \mathit{Skip}\ (\mathit{Right}\ (n+1), s_b, \mathit{Just}\ n) & \\
 & \quad \mathit{Right}\ n \to \mathit{Skip}\ (\mathit{Left}\ \ \ (n-1), s_b, \mathit{Just}\ n) & \\
\mathit{next}\ (s'_a, s_b, \mathit{Just}\ n) = & \textbf{case}\ s_b\ \textbf{of} & \\
 & \quad \mathit{Left}\ \ m \to \mathit{Yield}\ (n,m)\ (s'_a, \mathit{Right}\ (m+1), \mathit{Nothing}) & \\
 & \quad \mathit{Right}\ m \to \mathit{Yield}\ (n,m)\ (s'_a, \mathit{Left}\ \ \ (m-1), \mathit{Nothing}) &
\end{array}
$$

This stepper function does not use state shapes in the expected standard way. The set of shapes it matches on is not the same as the set of shapes used as the target of *Skip*/*Yield* transitions. It matches on these four state shapes

$$
\begin{array}{l}
(\mathit{Left}\ \_, \_\ \ \ \ , \mathit{Nothing}) \\
(\mathit{Right}\ \_, \_\ \ \ \ , \mathit{Nothing}) \\
(\_\ \ \ \ , \mathit{Left}\ \ \_, \mathit{Just}\ \_) \\
(\_\ \ \ \ , \mathit{Right}\ \_, \mathit{Just}\ \_)
\end{array}
$$

While it targets these four

$$
\begin{array}{l}
(\_\ \ \ \ , \mathit{Left}\ \ \_, \mathit{Nothing}) \\
(\_\ \ \ \ , \mathit{Right}\ \_, \mathit{Nothing}) \\
(\mathit{Left}\ \ \_, \_\ \ \ \ , \mathit{Just}\ \_) \\
(\mathit{Right}\ \_, \_\ \ \ \ , \mathit{Just}\ \_)
\end{array}
$$

This poses a problem. For a simple approach to fusing transformers with producers, we would like all combinations of stepper functions to be able to fuse such that they have the state machine form – where the set of source and target nodes are the same.

The options

We have two options. We can do an additional transformation at this stage to preserve the simpler conditions. Alternatively we can weaken the conditions we place on stepper functions – to allow examples such as the above – and then deal with the more complex situation in the final stage when we fuse consumers with producers.

For the first option, we could at this stage specialise the stepper function on all of the state shapes, thereby restoring the property that the stepper function corresponds to a simple state machine. The advantage of this option is that the transformation is relatively simple to explain and to perform. The conditions we place on stream producers also remain relatively simple. The primary disadvantage of this option is that it is not what real implementations of stream fusion do in practice. If we wish to maintain that stream fusion can be implemented using only general purpose optimisations then we should not use this method because it is unreasonable to expect a general purpose compiler to perform the needed specialisation at this stage: it does not appear to have a general benefit at this point.

The alternative option corresponds more closely to what real implementations of stream fusion do in practice. It involves defining weaker conditions on stream

producers and, in the final stream producer/consumer fusion stage, relying on call pattern specialisation (and extensions thereof) to resolve everything appropriately. The disadvantage is that both the transformation and the explanation are more complex. Worse, it is not entirely clear that the method is universal. For these reasons, for the purpose of the optimisation argument, we will take the simpler approach. We will however explain how the other approach works – at least in the cases where it clearly does work. We defer further consideration of the more complex approach to Section 4.6.6.

Strong state shape matching and specialisation

For the simple approach it is useful to define a notion that we will call *strong state shape matching*. Strong state shape matching is simple: every part of each state shape is scrutinised. This is easy to see syntactically if the stepper function is written in an equational style; the function will have exactly one clause for each state shape which will pattern-match exactly on that shape, e.g.

$$\begin{aligned} &next\ (Left\ n,\ \ Left\ \ \ m, Nothing) = \ldots \\ &next\ (Left\ n,\ \ Right\ m, Nothing) = \ldots \\ &\ldots \\ &next\ (Right\ n, Right\ m, Just\ n) \ \ \ = \ldots \end{aligned}$$

This is the form we want for our example – the fused zip_s stepper function. We can get it into this form by specialising.

The specialising transformation is straightforward. We build a new stepper function by parts using pattern matching. The function is defined for each state shape to be the original stepper function but specialised for that state shape. The new stepper function is equal to the original because it is equal for each state shape and the state shapes partition the state type. Of course multiple clauses with these simple patterns are just syntactic sugar, via the standard pattern matching algorithm, for a single definition using a tree of case expressions.

We start with the original stepper function

$$next\ s = \langle body \rangle$$

Then for each state shape we generate a clause: the state shape is used as a pattern which may binds variables, the body of the clause is the body of the original stepper function but the state parameter substituted for the state shape pattern.

$$next'\ \langle shape_x \rangle = \langle body \rangle [s := \langle shape_x \rangle]$$

We then simplify the body in each clause by reducing cases of known constructors.

With the example fused zip_s stepper function we start with the following

$$
\begin{array}{l}
next\ s = \textbf{case}\ s\ \textbf{of} \\
\qquad (s_a, s_b, Nothing) \rightarrow \\
\qquad\quad \textbf{case}\ s_a\ \textbf{of} \\
\qquad\qquad Left\quad n \rightarrow Skip\ (Right\ (n+1), s_b, Just\ n) \\
\qquad\qquad Right\ n \rightarrow Skip\ (Left\quad (n-1), s_b, Just\ n) \\
\qquad (s_a, s_b, Nothing) \rightarrow \\
\qquad\quad \textbf{case}\ s_b\ \textbf{of} \\
\qquad\qquad Left\quad m \rightarrow Yield\ (n, m)\ (s'_a, Right\ (m+1), Nothing) \\
\qquad\qquad Right\ m \rightarrow Yield\ (n, m)\ (s'_a, Left\quad (m-1), Nothing)
\end{array}
$$

For the first state shape $(Left\ n, Left\ m, Nothing)$, we have

$$
\begin{array}{l}
\quad next'\ (Left\ n, Left\ m, Nothing) \\
= \\
\quad \langle body \rangle [s := (Left\ n, Left\ m, Nothing)] \\
= \\
\quad \textbf{case}\ (Left\quad n, Left\ m, Nothing)\ \textbf{of} \\
\qquad (s_a, s_b, Nothing) \rightarrow \\
\qquad\quad \textbf{case}\ s_a\ \textbf{of} \\
\qquad\qquad Left\quad n \rightarrow Skip\ (Right\ (n+1), s_b, Just\ n) \\
\qquad\qquad Right\ n \rightarrow Skip\ (Left\quad (n-1), s_b, Just\ n) \\
\qquad \ldots
\end{array}
$$

If we now reduce the cases of known constructors we are left with

$$next'\ (Left\ n, Left\ m, Nothing) = Skip\ (Right\ (n+1), Left\ m, Just\ n)$$

The clauses for the other seven state shapes follow similarly.

4.5.4 Stream producer constraints

We impose a number of constraints on stream producers and transformers. In this section we will describe the constraints. In the next section we will argue that these constraints are sufficient to ensure that we can always successfully fuse stream transformers with stream producers. We will also rely on these constraints in Section 4.6 to make a similar argument about fusing stream consumers with stream producers. In addition, some constraints relate to the correct handling of *Skip* and to the allocation accounting argument.

We describe the constraints both in terms of a number of syntactic conditions and in terms of some higher level properties. We are primarily interested in the properties, the syntactic conditions are the route to achieving the desired properties. Again, we do not attempt to specify the most general syntactic form, our aim is to specify something simple and just sufficiently expressive.

Syntactic conditions

Recall from Section 4.2.1 that we defined the top level structure of good list producers so that we could always unfold them to expose an application of *unstream*. We impose similar conditions on stream producers to ensure that we can always unfold them to expose a *Stream* constructor applied to a stepper function and initial state.

Producer condition 1 (top level syntactic form)**.** We stipulate that, syntactically, stream producers look like

$$f\ x = \mathit{Stream}\ (\ldots)\ (\ldots)$$
$$\quad \textbf{where}\ \ldots$$

or

$$f\ x = h\ (\ldots)$$
$$\quad \textbf{where}\ \ldots$$

Where h is some existing stream producer. That is, at the top level we allow any mixture of lambda abstraction and **let** binding (or equivalently, **where** clauses). The first body term must either be *Stream* or another stream producer. We have the usual requirement that the collection of stream producers is finite; that all definitions are available; and that where producers are defined in terms of other producers, that the definitions do not form cycles. This guarantees that we can unfold any fully applied stream producer function to expose a *Stream* constructor along with a stepper function and initial state.

Producer condition 2 (top level of transformers)**.** For stream transformers there is an additional constraint: that all input streams are matched on at the top level.

$$t\ x\ a\ b = \textbf{case}\ a\ \textbf{of}\ \mathit{Stream}\ \mathit{next}_a\ s_a \rightarrow$$
$$\qquad \textbf{case}\ b\ \textbf{of}\ \mathit{Stream}\ \mathit{next}_b\ s_b \rightarrow \mathit{Stream}\ (\ldots)\ (\ldots)$$
$$\quad \textbf{where}\ \ldots$$

This guarantees that saturated applications of transformers can be reduced to expose the *Stream* constructor result.

Producer condition 3 (stepper functions are case expression trees)**.** The general syntactic form for a stream stepper function is a tree of case expressions; that is, case expressions possibly with more case expressions in the branches and finally *Step* constructors in the leaves.

Producer condition 4 (strong state shape matching)**.** If the stream uses multiple state shapes then the top level of the case expression tree must match on all of these shapes.

$$
\begin{array}{l}
next\ s = \textbf{case}\ s\ \textbf{of} \\
\qquad\qquad Mode1\ \_\ \_ \rightarrow \ldots \\
\qquad\qquad Mode2\ \_\ \_ \rightarrow \ldots \\
\qquad\qquad \ldots
\end{array}
$$

If the state shapes are represented with compound data constructors then nested cases will be required. Strong state shape matching is required: every part of each state shape must be scrutinised (see Section 4.5.3).

State shapes that make use of nested constructors are very common; in earlier sections we used examples that use combinations of pair, *Maybe* and *Either* data constructors. The $init_s$ function, for example, used shapes $(Nothing, s)$ and $(Just\ x, s)$. It is essential that compound constructors can be used for state shapes because such cases occur when fusing stream functions.

Producer condition 5 (scrutinee terms in case expression trees)**.** Within the tree of case expressions, below the matching on the state shapes, we may have further case expressions.

$$
\begin{array}{l}
next\ s = \textbf{case}\ s\ \textbf{of} \\
\qquad\qquad Mode1\ s_a\ \_ \rightarrow \textbf{case}\ next_a\ s_a\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad Mode2\ x\ \_ \rightarrow\ \textbf{case}\ h\ x\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad \textbf{where} \\
\qquad\qquad\qquad\qquad\quad h = fix\ (\ldots)
\end{array}
$$

The scrutinee of these case expressions may be either:

- an application of an input stream's stepper function, which may only appear in the tree directly below the matching on the state shapes;
- some other arbitrary term that does not use any stream stepper function (neither those of input streams nor recursive use of the stepper function being defined).

In particular, in the latter case it is acceptable to use recursive functions, just not to make a recursive call to the stepper function. Local definitions may also be used, provided they do not use any stream stepper function.

In previous examples we have used Haskell's guard syntax with Boolean predicates. This is acceptable because it desugars into case expressions on terms of Boolean type. Similarly, function definitions using simple patterns also desugar into case expressions.

Producer condition 6 (result state terms are state shapes)**.** The leaves of this tree of case expressions must be the application of a *Step* constructor: *Done*, *Skip* or *Yield*. In the case of *Skip* and *Yield* the new state must manifestly be one of the state shapes.

$$
\begin{array}{l}
next\ s = \textbf{case}\ s\ \textbf{of} \\
\qquad Mode1\ s_a\ \_ \rightarrow \textbf{case}\ next_a\ s_a\ \textbf{of} \\
\qquad\qquad\qquad Done \qquad\quad \rightarrow \ldots \\
\qquad\qquad\qquad Skip \quad s'_a \rightarrow Skip \qquad\quad (Mode1\ s'_a\ \ldots) \\
\qquad\qquad\qquad Yield\ x\ s'_a \rightarrow Yield\ (\ldots)\ (Mode2\ \ldots)
\end{array}
$$

The *Yield* element and the variables within each state shape result can be arbitrary terms, with the proviso that they do not use any stream stepper function.

Producer condition 7 (skip handling)**.** For case expressions that scrutinise the result of an input stream's stepper function, the result in the *Skip* branch must be *Skip*. Furthermore, the result state shape must be the same as the state shape matched in the branch of the top level case expression. In the above example, all *Skip* results under the *Mode1* branch of the top level case expression are also *Mode1*.

Producer condition 8 (single use of stepper functions)**.** Tracing down the paths of the tree of case expressions, in each path, there there may be at most one use of an input stream stepper function.

Producer condition 9 (initial state is a state shape)**.** The initial stream state must be manifestly one of the state shapes.

Summary grammar

We can approximate the syntactic conditions with the following grammar. To capture some of the conditions we specify a side condition that some of the non-terminals must be used maximally. These cases are noted below.

$$
\begin{array}{lll}
\langle good\_producer \rangle & ::= & f = \langle top \rangle \\
\langle top \rangle & ::= & \lambda a \to \langle top \rangle \\
 & | & \textbf{let} \ldots \textbf{in} \ \langle top \rangle \\
 & | & h \ \{ \langle expr \rangle \} \\
 & | & \langle top\_stream \rangle \\
\langle top\_stream \rangle & ::= & \textbf{case} \ a \ \textbf{of} \ Stream \ next_x \ s_x \to \langle top\_stream \rangle \\
 & | & \textbf{let} \ next \ s = \langle case\_tree\_top \rangle \\
 & & \ \textbf{in} \ Stream \ next \ \langle state\_shape\_expr \rangle \\
\langle case\_tree\_top \rangle & ::= & \textbf{case} \ s_x \ \textbf{of} \\
 & & \quad Shape_1 \ s_{11} \ s_{12} \ldots \to \langle case\_tree\_top \rangle \\
 & & \quad Shape_2 \ s_{21} \ s_{22} \ldots \to \langle case\_tree\_top \rangle \\
 & & \quad \ldots \\
 & | & \langle case\_tree\_middle \rangle \\
\langle case\_tree\_middle \rangle & ::= & \textbf{case} \ next_x \ s_x \ \textbf{of} \\
 & & \quad Done \to \langle case\_tree\_leaf \rangle \\
 & & \quad Skip \ s'_x \to Skip \ \langle state\_shape\_expr \rangle \\
 & & \quad Yield \ a \ s'_x \to \langle case\_tree\_leaf \rangle \\
 & | & \langle case\_tree\_bottom \rangle \\
\langle case\_tree\_bottom \rangle & ::= & \textbf{case} \ \langle expr \rangle \ \textbf{of} \\
 & & \quad \ldots \to \langle case\_tree\_bottom \rangle \\
 & & \quad \ldots \\
 & | & \langle case\_tree\_leaf \rangle \\
\langle case\_tree\_leaf \rangle & ::= & Done \\
 & | & Skip \quad \langle state\_shape\_expr \rangle \\
 & | & Yield \ \langle expr \rangle \ \langle state\_shape\_expr \rangle \\
\langle state\_shape\_expr \rangle & ::= & Shape_x \ \{ \langle state\_shape\_expr \rangle \} \\
 & | & \langle expr \rangle
\end{array}
$$

In $\langle top \rangle$, h must refer to another good stream producer.

The following non-terminals must be used maximally

- $\langle top\_stream \rangle$: stream transformers must match all their stream inputs.
- $\langle case\_tree\_top \rangle$: all state shapes must be matched.
- $\langle case\_tree\_middle \rangle$: $\langle case\_tree\_bottom \rangle$ is not allowed to match an expression **case** $next_x \ s_x$ **of**.
- $\langle state\_shape\_expr \rangle$: the expression has to correspond to a state shape.

Properties arising from the syntactic conditions

Producer property 1 (static nodes and transitions)**.** The stream stepper function must manifestly identify the input state shape and result state shape. This constraint is implicit in the state machine viewpoint since we require that the stepper function correspond to a set of transitions between nodes and the transition arrows identify a source and target node. This property constraint corresponds to syntactic condition 4 that the top level case expression must match on all the state shapes and condition 6 that the *Skip* and *Yield* results must use one these same state shapes.

This constraint is crucial for the optimisation arguments we will make in Section 4.6. The key is that the state shapes and the state transitions between them are statically known.

As an example of what this constraint excludes, consider a stream similar to $init_s$ that uses nodes with the shapes $(Nothing, s)$ and $(Just\ x, s)$ but which has the following transition label

$$Yield\ x\ s' \rightarrow Yield\ x\ (f\ x, s')$$

This is not an acceptable transition because it does not manifestly identify the target node. One cannot draw this transition in a state machine diagram. The state it transitions into depends on the result of $f\ x$. To be acceptable, this bad transition must be refactored into two guarded transitions.

$$\begin{array}{ll} Yield\ x\ s' \mid p\ x & \rightarrow Yield\ x\ (Just\ (g\ x), s') \\ Yield\ x\ s' \mid not\ (p\ x) & \rightarrow Yield\ x\ (Nothing, \quad s') \end{array}$$

Here we still do a dynamic test on x but now we identify the target nodes explicitly.

Producer property 2 (input skips)**.** For stream transformers, when a step is demanded from an input stream and that input stream step turns out to be a skip then the overall transition must also be a skip. Furthermore it is a skip transition back to the same node, though with appropriate updated state variable(s). This constraint guarantees that the stream transformer is oblivious to skips, which, as we recall from Chapter 3, is necessary for the correctness of fusion on skipping streams.

Producer property 3 (one input step per transition)**.** Another constraint that affects stream transformers is that each transition may demand at most one step from the input streams. That is, a single transition may not demand multiple steps from a single input stream, or a step from more than one input stream.

This property follows from syntactic condition 8 about the occurrences of stream stepper functions in the paths of the tree of case expressions. The syntactic condition is slightly stronger than is necessary for the property we want, however there is little to no benefit in trying to demand multiple steps from input streams in a single transition of the overall stream. The stronger syntactic constraint can be thought of as merely a simplification.

To see why there is no benefit, consider the zip_s function. It has two input streams and its stepper function has to obtain an element from each input stream before it can yield the pair of elements. We might think about trying to write the step function in the following way

$$
\begin{aligned}
next\ (s_a, s_b) = \textbf{case}\ next_a\ s_a\ \textbf{of}& \\
&\mathit{Done} \to \ldots \\
&\mathit{Skip}\ \ s'_a \to \ldots \\
&\mathit{Yield}\ a\ s'_a \to \textbf{case}\ next_b\ s_b\ \textbf{of} \\
&\qquad\qquad \mathit{Done} \to \ldots \\
&\qquad\qquad \mathit{Skip}\ \ s'_b \to \ldots \\
&\qquad\qquad \mathit{Yield}\ b\ s'_b \to \mathit{Yield}\ (a, b)\ (s'_a, s'_b)
\end{aligned}
$$

But what about the case where $next_b\ s_b$ gives us *Skip*? There must be some mode to transition into. We might consider discarding the element and new state obtained from the first stream and transition back to the main node with an updated state from the second stream:

$$
\begin{aligned}
next\ (s_a, s_b) = \textbf{case}\ next_a\ s_a\ \textbf{of}& \\
&\mathit{Done} \to \ldots \\
&\mathit{Skip}\ \ s'_a \to \ldots \\
&\mathit{Yield}\ a\ s'_a \to \textbf{case}\ next_b\ s_b\ \textbf{of} \\
&\qquad\qquad \mathit{Done} \to \ldots \\
&\qquad\qquad \mathit{Skip}\ \ s'_b \to \mathit{Skip}\ (s_a, s'_b) \\
&\qquad\qquad \mathit{Yield}\ b\ s'_b \to \ldots
\end{aligned}
$$

Of course this is not acceptable because of the danger of repeating the work (and allocations) done to obtain the element from the first stream.

We might think about recursing until the second stream gives us either *Done* or *Yield*. This is not an option, firstly because such recursion is not allowed (see property 4) and secondly because it would violate property 1: the transition would not statically identify the target state node.

There must be a specific stream mode representing the situation where we have obtained an element from the first stream but not yet obtained one from the

second. In this mode we must be prepared to handle the second stream skipping multiple times.

Even given that we need a separate mode we might optimistically consider trying to pull from both input streams in one step and only if the second skips would we move into the separate state.

$$
\begin{aligned}
next\ (s_a, s_b, Nothing) = \ & \textbf{case}\ next_a\ s_a\ \textbf{of} \\
& \quad Done \rightarrow \ldots \\
& \quad Skip\ \ s'_a \rightarrow \ldots \\
& \quad Yield\ a\ s'_a \rightarrow \textbf{case}\ next_b\ s_b\ \textbf{of} \\
& \qquad\qquad Done \rightarrow \ldots \\
& \qquad\qquad Skip\ \ s'_b \rightarrow Skip\ (s'_a, s'_b, Just\ a) \\
& \qquad\qquad Yield\ b\ s'_b \rightarrow \ldots \\
next\ (s'_a, s_b, Just\ a) = \ & \textbf{case}\ next_b\ s_b\ \textbf{of} \\
& \quad \ldots
\end{aligned}
$$

There is however no benefit in trying to do this. It is considerably simpler to skip to the $(s'_a, s_b, Just\ a)$ mode after obtaining an element from the first stream.

Producer property 4 (no recursion)**.** Recall from the first chapter (specifically Section 1.4.2) that eliminating recursion in the stepper functions is what enabled stream fusion to be an optimisation in the *filter* example where it was not an optimisation under the *unbuild*/*unfoldr* system. It should come as no surprise therefore that we impose a constraint on the use of recursion in stepper functions.

It is interesting to note that this constraint is almost redundant given the first constraint about statically identifying source and target nodes. There is very little interesting that we could do with a recursive stepper function while still statically identifying the target state mode. Nevertheless we must ban stepper functions such as the following for although there is only one target mode the fixpoint will still get in the way of the case-of-case transformation.

$$
\begin{aligned}
& next\ s = f\ s \\
& \quad \textbf{where} \\
& \qquad f\ s \mid p\ s \qquad = f\ (g\ s) \\
& \qquad\quad \mid otherwise = Yield\ (h\ s)\ s
\end{aligned}
$$

This example can be trivially refactored to use a local fixpoint instead

$$
\begin{aligned}
& next\ s = \textbf{case}\ f\ s\ \textbf{of} \\
& \qquad\qquad (x, s') \rightarrow Yield\ x\ s' \\
& \quad \textbf{where} \\
& \qquad f\ s \mid p\ s \qquad = f\ (g\ s) \\
& \qquad\quad \mid otherwise = (h\ s, s)
\end{aligned}
$$

Specifically, the no recursion constraint is that the stepper function result must not be wrapped in a fixpoint, and for transformers, uses of the input stream's stepper function also must not be wrapped in a fixpoint. On the other hand it is acceptable to use recursion in auxiliary definitions, in the scrutinee term in case expressions or to calculate the values of variables in new states.

Producer property 5 (no allocation on skip)**.** Parts of condition 4 and 5 about where things occur in the tree of case expressions are related to the allocation accounting argument (see Section 4.7.8). In particular we have the requirement that the matching on state shapes should appear at the top of the tree and that uses of input stream's stepper functions should appear immediately beneath. Together these ensure that no allocations are incurred by stream transformers when an input stream skips.

It would be possible to have a more relaxed condition so long as we could ensure that there is no allocation on the path from the root of the expression tree to any use of an input stream stepper function. The simpler and more stringent condition does not restrict expressiveness however because it is possible to split a long chain of case expressions by introducing a *Skip* into a new top level state shape.

4.5.5 Combining transformers with producers

The transformations for stream transformer/producer fusion follow in essentially the same way in the general case as in the simple example of Section 4.5.1. We can justify that the various steps are possible based on the stream producer conditions. We must also verify that the resulting stream producer satisfies all the stream producer conditions.

As mentioned previously, the general case is a stream transformer fully applied to the appropriate number of stream producers. The transformer and producers are assumed to satisfy the stream producer conditions. Note that we never have to deal with a transformer that is partially applied, or applied to a stream term that is an abstracted variable (see Sections 4.2.3 and 4.3.1).

Top level structure

The first step is unfolding the stream producers and then the transformer to expose a *Stream* constructor. This step relies on the conditions on the top-level syntactic structure of stream producers and transformers. In particular,

producer condition 1 ensures that stream producers can be unfolded to give a *Stream* constructor applied to a stepper function and initial state.

$$
\begin{array}{ll}
 & \mathit{producer}\ x\ y\ z \\
= & \quad \{\ \text{unfold the producer}\ \} \\
 & \mathbf{let}\quad \mathit{next}\ s = \ldots \\
 & \qquad \ldots \\
 & \quad \mathbf{in}\ \mathit{Stream}\ \mathit{next}\ (\ldots)
\end{array}
$$

The *Stream* constructor may be wrapped in **let** expressions that supply local function definitions such as a stepper function. For presentational convenience we have usually used **where** clauses, however these desugar to **let** expressions.

A requirement for stream transformers (condition 2) is that input streams are deconstructed at the top level, but within these top level case expressions, the first body term is a *Stream*.

$$
\begin{array}{ll}
 & \mathit{transformer}\ a\ b\ x\ y \\
= & \quad \{\ \text{unfold the transformer}\ \} \\
 & \mathbf{case}\ a\ \mathbf{of}\ \mathit{Stream}\ \mathit{next}_a\ s_a \rightarrow \\
 & \mathbf{case}\ b\ \mathbf{of}\ \mathit{Stream}\ \mathit{next}_b\ s_b \rightarrow \\
 & \quad \mathbf{let}\quad \mathit{next}\ s = \ldots \\
 & \qquad \ldots \\
 & \quad \mathbf{in}\ \mathit{Stream}\ \mathit{next}\ (\ldots)
\end{array}
$$

When we unfold the transformer we get each stream producer term in the scrutinee position of the corresponding top level case expression.

$$
\begin{array}{l}
\mathbf{case}\ (\mathbf{let}\ \ \mathit{next}\ s = \ldots \\
\qquad\qquad \ldots \\
\qquad\quad \mathbf{in}\ \mathit{Stream}\ \mathit{next}\ (\ldots)) \\
\mathbf{of}\ \mathit{Stream}\ \mathit{next}_a\ s_a \rightarrow \\
\qquad\quad \ldots
\end{array}
$$

We can immediately float the definitions in the **let** expression outwards and reduce since we have a case of a known constructor. Since the transformer is fully applied to its stream arguments then we can do this for all the input streams. Note that because we substitute for all the input streams, we no longer have a stream transformer but a stream producer. We are left with the stream transformer's *Stream* constructor term and the associated local definitions. This syntactic form satisfies the conditions on the the top level structure (producer condition 1).

New state shapes

The next stage will be to optimise the stepper function and check that we can satisfy all the producer conditions. Before doing any optimisation however, let us first consider the various producer conditions as they apply to the stepper function that we have at this stage of the transformation.

At this stage we have essentially the same stepper function as that of the original stream transformer. The original stepper function from the transformer refers to the stepper functions from the transformer's input stream parameters. The difference at this stage is simply that the top-level unfolding step has given us local definitions for these stepper functions.

While the stepper function is essentially unchanged, the conditions that we judge it against have changed. As noted above, we now have a stream producer, rather than a transformer, but more interestingly the new producer has a new set of state shapes, which is important because many of the stream producer conditions refer to the state shapes.

Recall from the overview in Section 4.3.4 that, for the purposes of the argument, we assign a set of state shapes to each stream producer. The promise we are trying to keep is that for producers that satisfy the producer constraints, we will eventually eliminate the allocations associated with their state shapes. At this inductive step of the argument we can assume that we can keep this promise for the original transformer and all its input producers. If we fuse the transformer and producers into a single producer then we need to preserve the same promise. Furthermore we must preserve the total number of allocations associated with the state shapes, otherwise after the fusion step we might not be able to eliminate as many allocations as before. Thus we need to assign the fused producer a set of state shapes such that the total number of allocations is preserved. Then with whatever set of state shapes we select, we will have to ensure that the fused producer satisfies the producer conditions for the chosen state shapes.

One option would be to select the same set of state shapes as the transformer. That would be easy since the stream stepper function at this stage already satisfies the producer constraints for the state shapes of the transformer. We cannot actually select this option of course because we would not eliminate the allocations associated with the state shapes of the producers: they would end up as dynamic allocations rather than being compiled into control-flow state.

We choose as the new set of state shapes to assign, the composition of the state shapes of the transformer with those of the producers. This set is obtained

by substituting all combinations of the state shapes of the producers into the appropriate locations in the state shapes of the transformer.

Example

To see more clearly what is going on, let us revisit the earlier simple example

$$enum10 = snoc_s\ (enumFromTo_s\ 0\ 9)\ 10$$

The internal stream state type for the $snoc_s$ transformer is *Maybe s*, where *s* is the stream state type of the input stream. As we have seen previously, the state shapes for $snoc_s$ are *Just* _ and *Nothing*. The internal state type for $enumFromTo_s\ 0\ 9$ is *Int* with a single trivial state shape. When we apply $snoc_s$ to $enumFromTo_s\ 0\ 9$ then the new internal stream state type becomes *Maybe Int*. In this case, because the producer $enumFromTo_s$ uses only a single trivial state shape, then the new state shapes are still *Just* _ and *Nothing*.

As an example where both the transformer and the producer use multiple state shapes, consider

$$enum11 = snoc_s\ enum10\ 11$$

That is, we take the previous example and apply $snoc_s$ again. In this case the producer's internal state type is *Maybe Int* with shapes *Just* _ and *Nothing* and the transformer's internal state type is *Maybe s*, also with shapes *Just* _ and *Nothing*. With the transformer applied to the producer, the new internal state type and state shapes are simply the composition of the components; that is the state type is *Maybe* (*Maybe Int*) and the state shapes are *Just* (*Just* _), *Just Nothing* and *Nothing*. The general case follows the same pattern; the stream state types are composed giving rise to all the possible combinations of state shapes.

Applying case-of-case

Having chosen the state shapes for the new fused producer, we can again consider the producer conditions. The most important producer condition that is not met at this stage is condition 4 which requires that the top level of the stepper function matches against all the state shapes. To restore this constraint we must proceed to the next stage and apply the case-of-case optimisation for each of the transformer's stream inputs.

The stepper functions for both the transformer and producers are trees of case expressions (condition 3). The aim with the case-of-case transformation is to

fuse them into a single tree of case expressions. We rely on condition 5 that the transformer's use of its input streams is no more complicated than a simple application $next_a\ s_a$ in the scrutinee position of a case expression.

A stepper function $next_a$ can be unfolded in the scrutinee position. This gives us a case expression with the scrutinee term being a tree of case expressions with *Step* constructors in the leaves. We rely on condition 6 that each producer has only *Step* terms in the leaf positions of its tree of case expressions. This gives us a situation that is a straightforward generalisation of the simple case-of-case described previously in Section 4.5.2. We will cover the details of the generalised case-of-case shortly in Section 4.5.7.

Matching on state shapes

In the *enum11* example, the final stepper function after applying case-of-case is the following tree of case expressions

$$
\begin{array}{l}
\textbf{case}\ s\ \textbf{of} \\
\quad \mathit{Just}\ s' \quad \to \textbf{case}\ s'\ \textbf{of} \\
\qquad\qquad\qquad \mathit{Just}\ n \quad \to \textbf{case}\ n \leqslant 9\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{True} \quad \to \mathit{Yield}\ n\ \ (\mathit{Just}\ (\mathit{Just}\ (n+1))) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{False} \quad \to \mathit{Yield}\ 10\ (\mathit{Just}\ \mathit{Nothing}) \\
\qquad\qquad\qquad \mathit{Nothing} \to \mathit{Yield}\ 11\ \mathit{Nothing} \\
\quad \mathit{Nothing} \to \mathit{Done}
\end{array}
$$

Note how the new stepper function matches explicitly on all the new compound state shapes $\mathit{Just}\ (\mathit{Just}\ \_)$, $\mathit{Just}\ \mathit{Nothing}$ and $\mathit{Nothing}$. A bit of syntactic sugar makes this clearer

$$
\begin{array}{lll}
\mathit{next}\ (\mathit{Just}\ (\mathit{Just}\ n)) & \mid n \leqslant 9 & = \mathit{Yield}\ n\ \ (\mathit{Just}\ (\mathit{Just}\ (n+1))) \\
& \mid \mathit{otherwise} & = \mathit{Yield}\ 10\ (\mathit{Just}\ \mathit{Nothing}) \\
\mathit{next}\ (\mathit{Just}\ \mathit{Nothing}) & & = \mathit{Yield}\ 11\ \mathit{Nothing} \\
\mathit{next}\ \mathit{Nothing} & & = \mathit{Done}
\end{array}
$$

The corresponding state machine is given in Figure 4.9.

As discussed previously in Section 4.5.3, in the general case we cannot guarantee that, after the case-of-case transformation, the new stepper function matches on all the compound state shapes. In the new combined tree of case expressions, we have the matches on the original transformer's state shapes. Where the transformer scrutinised $next_x\ s_x$ we now have a tree of case expressions which matches on the state shapes of the corresponding producer. We would be able to guarantee that the new combined tree of case expressions matches all

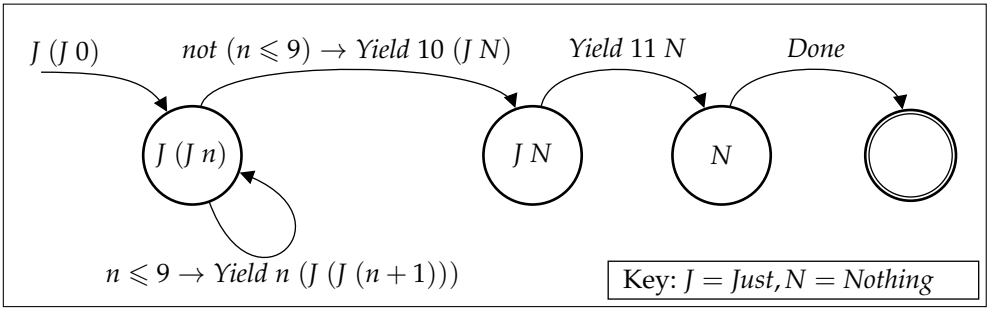


Figure 4.9: State machine view of fused $snoc_s\ (snoc_s\ (enumFromTo_s\ 0\ 9)\ 10)\ 11$

the compound state shapes if the original transformer scrutinises $next_x\ s_x$ for each stream state variable s_x in each mode. We do not however place such a requirement on transformers – zip_s could not be expressed if we did so.

Applying state shape specialisation

The solution we adopt is to apply a specialisation transformation to the stepper function, the details of which are covered in Section 4.5.3. We note that the new stepper function is semantically equal to the old function and we rely on the fact that, by its construction, it matches all the state shapes (producer condition 4). We also rely on two further properties:

1. that in the *Skip* and *Yield* results, the state term exactly matches a state shape (producer condition 6);
2. and that the transformation does not increase allocations.

These two points need further justification.

For the first property we are interested in the possible forms of the new state term. To illustrates the possible cases, consider the following fragment of a transformer's stepper function.

$$
\begin{aligned}
next\ (Mode1\ s_a\ s_b) = \textbf{case}\ next_a\ s_a\ \textbf{of}\\
\qquad\qquad\qquad\qquad\quad & Done \rightarrow Skip\ Mode2\\
& Skip\ s'_a \rightarrow Skip\ (Mode1\ s'_a\ s_b)\\
& \ldots
\end{aligned}
$$

Each of the new state terms is manifestly one of the transformer's state shapes. Now consider the situation after case-of-case and specialisation transformations.

Terms that do not use an input stream's state at all, such as *Mode2* above, are unproblematic; their shape is complete, they are not affected by the subsequent transformations. For terms that do use input stream's state variables we need to check that they get substituted for a constructor term that is one of the producer's state shapes. There are two possibilities, both illustrated by the *Mode1* result above: a new state variable (e.g. s'_a) obtained from using an input stream's stepper function, or alternatively an unmodified state variable (e.g. s_b). The first case is covered by the case-of-case transformation: the new state variable will be substituted for a state term obtained as a result of the input stream's stepper function which necessarily uses one of that stream's state shapes. The second case is covered by the specialisation transformation: the state variable is bound by the pattern for the transformer's state shape; after substitution and case reduction this state variable will be bound to the appropriate component of the overall state shape.

For the second property, consider that when generating each specialised clause, we substitute the state shape pattern – considered now as an applicative term of constructors and bound variables – into the body of the original stepper function. Despite the fact that we are inserting extra constructor terms, this does not increase allocations. To see that this is the case, consider where such constructor terms can appear: either in the scrutinee position of a case expression or in the new state term for *Skip* or *Yield* results. Constructors appearing in a scrutinee position can be reduced giving rise to subterms appearing in another scrutinee positions or the new state term of the result. After all the case reduction is performed then the extra constructors appear only in the new state term of the result. Constructors representing state shapes that appear in state term of the result are exactly those constructors that we promise to eliminate (see Sections 1.4.3 and 4.3.4). Thus adding such constructors does not increase allocations because they will eventually be eliminated.

Reestablishing the producer conditions

We must now check that we can satisfy all the producer conditions.

Condition 1 (about the the top level syntactic form): We dealt with this condition in the early part of this section.

Condition 2 (about the the top level form of transformers): This applies only to transformers. Having combined a transformer with one or more producers we are left with a producer.

Condition 3 (that stepper functions are trees of case expressions): The specialisation transformation gives a new stepper with simple top-level pattern matching which in turn desugars into a tree of case expressions.

Note that this condition holds even if we do not use the specialisation transformation. The original transformer's stepper function is a tree of case expressions. Substituting the stepper function of input producer streams does not affect this. Applying the generalised case-of-case transform preserves the tree form, indeed it gives us a bigger case tree. See Section 4.5.7 for details.

Condition 4 (about strong state shape matching): As described previously, by construction, the stepper function obtained from the specialisation transformation matches all the state shapes.

Condition 5 (about what scrutinee terms are allowed in stepper function case expression trees): For this condition we note there are no remaining uses of input stream stepper functions – all occurrences of $next_x\ s_x$ have been substituted for – thus all scrutinee terms fall into the second permissible category: arbitrary terms that does not use any stream stepper function.

Condition 6 (that result state terms are state shapes): This condition requires that the leaves of the case tree be *Step* terms and that for *Skip* and *Yield* the new state be a term matching one of the stream's state shapes. It is clear that the leaf terms of the case expression tree are *Step* terms because it is true of the original transformer and the the case-of-case and specialisation transformations only apply substitutions to the leaf terms – the structure of the leaf terms is not affected.

For the second part, as described in the previous section, after applying state shape specialisation, the state new terms for *Skip* and *Yield* match one of the state shapes of the new stream.

Condition 7 (about the correct handling of *Skip* by transformers): This applies only to transformers.

Condition 8 (about the use of stepper functions in transformers): This applies only to transformers.

Condition 9 (that the initial stream state is a state shape): The term for the initial state of the transformer is defined using the initial state variables of the input stream producers. The new initial state term is that of the transformer but with the terms from the producers substituted in. Since the initial state term of the transformer and of the producers match one of

their respective state shapes then their composition matches one of state shapes of the new stream.

4.5.6 Revisiting the fixpoint problem

Recall the problem we encountered in Chapter 1 with fixpoints getting in the way of the case-of-case transformation. It is interesting to see how the producer conditions that we have defined exclude the problematic cases. In particular for the $filter_s$ function, in Chapter 1 we initially tried writing the stepper function in the following way using recursion

$$
\begin{array}{l}
next_{filter}\ s = \textbf{case}\ next_0\ s\ \textbf{of} \\
\qquad\qquad\begin{array}{lll}
Done & & \rightarrow Done \\
Yield\ x\ s' & \mid p\ x & \rightarrow Yield\ x\ s' \\
 & \mid otherwise & \rightarrow next_{filter}\ s'
\end{array}
\end{array}
$$

This wraps a fixpoint around the whole case expression. This use of recursion is excluded by condition 6 that the leaves of the case expression tree be *Step* constructors and nothing else.

Another alternative way of writing the $filter_s$ stepper function would be to use a local fixpoint rather that wrapping the fixpoint around the whole case expression tree.

$$
\begin{array}{l}
next_{filter}\ s = \textbf{case}\ force\ next_0\ s\ \textbf{of} \\
\qquad\qquad\begin{array}{ll}
Nothing & \rightarrow Done \\
Just\ (x, s') & \rightarrow Yield\ x\ s'
\end{array} \\
\quad \textbf{where} \\
\qquad force\ f\ s = \textbf{case}\ f\ s\ \textbf{of} \\
\qquad\qquad\qquad\begin{array}{lll}
Done & & \rightarrow Nothing \\
Yield\ x\ s' & \mid p\ x & \rightarrow Just\ (x, s') \\
 & \mid otherwise & \rightarrow force\ f\ s'
\end{array}
\end{array}
$$

This use of recursion is excluded by condition 5 that in transformers' stepper functions, the case expression scrutinee terms be either an application of an input stream's stepper function or another term that does not involve any stream stepper function.

These recursion constraints are motivated by the fact that we unfold stepper functions into the scrutinee positions of other stepper functions.

$$
\begin{array}{l}
next\_outer\ s = \textbf{case}\ next\_inner\ s\ \textbf{of} \\
\qquad\qquad \ldots \\
next\_inner\ s = \textbf{case}\ \ldots
\end{array}
$$

It is essential for the case-of-case transformation that in this situation, the inner case expression is not wrapped in any fixpoint.

$$next\_outer\ s = \textbf{case}\ (\textbf{case}\ \ldots\ )\ \textbf{of}$$
$$\ldots$$

That requires both that the stepper function we unfold does not come wrapped in a fixpoint

$$next\_inner\ s = fix\ (\lambda f \rightarrow \textbf{case}\ \ldots\ )$$

and also that the scrutinee term does not wrap a fixpoint around the stepper function

$$next\_outer\ s = \textbf{case}\ fix\ (\lambda f \rightarrow \ldots next\_inner\ s \ldots)\ \textbf{of}$$

4.5.7 The general case-of-case transformation

It should again be noted that properly speaking, what we typically refer to as the case-of-case transformation actually consists of the combination of the case-of-case transformation followed by case reduction. The general case-of-case transformation is then simply repeated application of the case-of-case transformation followed eventually by case reduction.

The general case we must deal with is a case expression where the scrutinee term is not simply a single case expression with *Step* terms immediately in each branch, but where the inner term is a whole tree of case expressions with *Step* terms in the leaves.

Consider, as an example, a stream producer constructed using $filter_s$. This gives us a stepper function that consists of multiple case expressions in a tree

$$\begin{aligned}
&filter_s\ even\ (enumFromTo_s\ 0\ 9)\\
=\ &Stream\ next_{filter}\ n\\
&\quad \textbf{where}\\
&\qquad next_{filter}\ n = \textbf{case}\ n \leqslant m\ \textbf{of}\\
&\qquad\qquad True\ \rightarrow \textbf{case}\ even\ n\ \textbf{of}\\
&\qquad\qquad\qquad\qquad True\ \rightarrow Yield\ n\ (n+1)\\
&\qquad\qquad\qquad\qquad False \rightarrow Skip\quad\ (n+1)\\
&\qquad\qquad False \rightarrow Done
\end{aligned}$$

If we now apply an additional stream transformer (e.g. $map_s\ f$) then it gives us a more complicated case-of-case example where the scrutinee term is a tree of case expressions

$$
\begin{array}{l}
\textbf{case}\ (\textbf{case}\ n \leqslant m\ \textbf{of} \\
\qquad\qquad \mathit{True}\ \rightarrow \textbf{case}\ \mathit{even}\ n\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{True}\ \rightarrow \mathit{Yield}\ n\ (n+1) \\
\qquad\qquad\qquad\qquad \mathit{False} \rightarrow \mathit{Skip}\quad (n+1) \\
\qquad\qquad \mathit{False} \rightarrow \mathit{Done}) \\
\textbf{of} \\
\quad \mathit{Done}\quad\ \ \rightarrow \ldots \\
\quad \mathit{Skip}\quad s' \rightarrow \ldots \\
\quad \mathit{Yield}\ x\ s' \rightarrow \ldots
\end{array}
$$

This general case is actually no harder than the simple case outlined previously. It consists of the same two basic transformations:

1. pushing one case expression through an inner one (Santos, 1995, Section 3.5.2);

2. reducing a case of a known constructor (Santos, 1995, Section 3.3.1).

Where the inner term is a tree we simply have to apply the first transformation multiple times. We push the outer case down each path of the inner tree of case expressions and use case reduction when we reach the leaves. For example, the intermediate step in this example is

$$
\begin{array}{l}
\textbf{case}\ n \leqslant m\ \textbf{of} \\
\quad \mathit{True}\ \rightarrow \textbf{case}\ (\textbf{case}\ \mathit{even}\ n\ \textbf{of} \\
\qquad\qquad\qquad\qquad \mathit{True}\ \rightarrow \mathit{Yield}\ n\ (n+1) \\
\qquad\qquad\qquad\qquad \mathit{False} \rightarrow \mathit{Skip}\quad (n+1)) \\
\qquad\quad \textbf{of} \\
\qquad\qquad \mathit{Done}\quad\ \ \rightarrow \ldots \\
\qquad\qquad \mathit{Skip}\quad s' \rightarrow \ldots \\
\qquad\qquad \mathit{Yield}\ x\ s' \rightarrow \ldots \\
\quad \mathit{False} \rightarrow \ldots
\end{array}
$$

Here we have pushed the outer case down one level. Repeating this one more time and using case reduction will complete the process.

Though we usually gloss over the presence of local definitions in **let** expressions, it is clear we can do so because we can simply float a **let** from a **case** scrutinee (Santos, 1995, Section 3.4.3).

4.5.8 The possibility of duplication

In general, the case-of-case transformation can duplicate code. This happens when the inner case expression produces the same constructor in multiple branches.

$$
\begin{array}{l}
\quad \textbf{case}\ (\textbf{case}\ e\ \textbf{of} \\
\qquad\qquad A \to C\ a \\
\qquad\qquad B \to C\ b \\
\qquad\qquad \ldots) \\
\quad \textbf{of} \\
\qquad C\ x \to \langle exp \rangle \\
= \\
\quad \textbf{case}\ e\ \textbf{of} \\
\qquad A \to \langle exp \rangle [x := a] \\
\qquad B \to \langle exp \rangle [x := b] \\
\qquad \ldots
\end{array}
$$

The term in the corresponding branch of the outer case expression is duplicated for each occurrence of the constructor as a result in the inner case expression.

In stepper functions this duplication can happen if *Done* or *Yield* is used multiple times as a result. The constraints on how *Skip* is handled makes it benign for producers to skip in multiple places. Multiple uses of *Yield* will in general lead to duplication of code; the code in the consumer that handles *Yield* must be duplicated for each occurrence of *Yield* in the producer. In principle the same applies for *Done*, though the way consumers typically handle *Done* makes it less problematic in practice.

The usual solution (Santos, 1995, Section 3.5.2) is to make a join point so that there is no duplication.

$$
\begin{array}{l}
\textbf{case}\ e\ \textbf{of} \\
\quad A \to join\ a \\
\quad B \to join\ b \\
\textbf{where} \\
\quad join\ x = \langle exp \rangle
\end{array}
$$

Unfortunately we cannot use this solution for stream stepper functions as we have defined them. Doing so would destroy the property that the stepper functions are a tree of case expressions.

The canonical example of a stream producer that uses *Yield* in two places is $append_s$. Consider for example its use in this stream producer

$$append_s\ (enumFromTo_s\ 0\ 9)\ (enumFromTo_s\ 10\ 19)$$

After transformer/producer fusion the stepper function is as follows

$$\begin{aligned}
next_{append}\ (Left\ (n, n')) &= \textbf{case}\ n \leqslant 9\ \textbf{of} \\
&\qquad True\ \rightarrow Yield\ n\ (Left\ (n+1, n')) \\
&\qquad False \rightarrow Skip\quad (Right\ n') \\
next_{append}\ (Right\ n') &= \textbf{case}\ n' \leqslant 19\ \textbf{of} \\
&\qquad True\ \rightarrow Yield\ n'\ (Right\ (n'+1)) \\
&\qquad False \rightarrow Done
\end{aligned}$$

To illustrate the duplication let us apply a further stream transformer, $filter_s\ even$. The fused stepper function becomes

$$\begin{aligned}
next_{append}\ (Left\ (n, n')) &= \textbf{case}\ n \leqslant 9\ \textbf{of} \\
&\qquad True\ \rightarrow \textbf{case}\ even\ n\ \textbf{of} \\
&\qquad\qquad True\ \rightarrow Yield\ n\ (Left\ (n+1, n')) \\
&\qquad\qquad False \rightarrow Skip\quad (Left\ (n+1, n')) \\
&\qquad False \rightarrow Skip\quad (Right\ n') \\
next_{append}\ (Right\ n') &= \textbf{case}\ n' \leqslant 19\ \textbf{of} \\
&\qquad True\ \rightarrow \textbf{case}\ even\ n'\ \textbf{of} \\
&\qquad\qquad True\ \rightarrow Yield\ n'\ (Right\ (n'+1)) \\
&\qquad\qquad False \rightarrow Skip\quad (Right\ (n'+1)) \\
&\qquad False \rightarrow Done
\end{aligned}$$

Note that the code for the in the *Yield* case of $filter_s$ stepper function has had to be duplicated into both modes of the $append_s$ stepper function.

We might imagine relaxing our constraints to allow join points. Instead of a tree of case expressions we would instead effectively have a directed acyclic graph, or to put it another way, a tree with shared – parametrised – subtrees. In the above $filter_s/append_s$ example it would look like

$$\begin{aligned}
next_{append}\ (Left\ (n, n')) &= \textbf{case}\ n \leqslant 9\ \textbf{of} \\
&\qquad True\ \rightarrow join\ n\ (Left\ (n+1, n')) \\
&\qquad False \rightarrow Skip\quad (Right\ n') \\
next_{append}\ (Right\ n') &= \textbf{case}\ n' \leqslant 19\ \textbf{of} \\
&\qquad True\ \rightarrow join\ n'\ (Right\ (n'+1)) \\
&\qquad False \rightarrow Done
\end{aligned}$$

$$\begin{aligned}
join\ n\ s' = \textbf{case}\ even\ n\ \textbf{of} \\
\qquad True\ \rightarrow Yield\ n\ s' \\
\qquad False \rightarrow Skip\quad s'
\end{aligned}$$

Note that the target mode is not manifestly identified, instead the mode is passed as a parameter to the join point. While this generalisation may work for transformer/producer fusion it is not at all clear that it would allow effective consumer/producer fusion since, as we will see in Section 4.6.5, this relies on statically identifying transitions between stream modes.

Not only can we get duplication due to case-of-case but we will also duplicate code if a transformer pulls from the same input stream in multiple places. In this situation the producer's step function must be unfolded for each occurrence.

These two kinds of duplication lead us to the guideline that stream producers should aim to yield in only one place and that transformers should pull from each input stream in one place only. It should be emphasised that both kinds of duplication relate only to code, not to runtime allocations. Thus it really is a guideline and not a constraint, as the allocation accounting is not affected.

4.6 Stream consumer/producer fusion

Stream consumer/producer fusion is the transformation whereby the application of a single stream consumer to a single stream producer is fused to give a collection of mutually recursive functions. The transformation consists of two phases. The first phase is very similar to stream transformer/producer fusion in that it eliminates *Step* constructors by using the case-of-case transformation. The second phase uses call pattern specialisation (Peyton Jones, 2007) to eliminate those data constructors in the stream state that are used to represent the static state shapes.

This section contributes another part of the argument of Section 4.3.3: we will argue that, subject to certain constraints on the consumer, the transformation is always applicable. Section 4.7.10 covers the argument that the transformation reduces allocations. We will start with an example to illustrate the transformation.

4.6.1 A simple example

In the overview in Section 4.3.2 we used the following example

$$sum_s\ (zipWith_s\ (\times)\ (enumFromTo_s\ 0\ 9)\ (enumFromTo_s\ 10\ 19))$$

After the stream transformer/producer fusion, this example had been reduced to sum_s applied to a single, somewhat complex, stream producer

$$
\begin{array}{l}
\quad sum_s\ (zipWith_s\ (\times)\ (enumFromTo_s\ 0\ 9)\ (enumFromTo_s\ 10\ 19)) \\
= \\
\quad sum_s\ (Stream\ next\ (0, 10, Nothing)) \\
\quad\quad \textbf{where} \\
\quad\quad\quad next\ (n, m, Nothing) \mid n \leqslant 9 \quad = Skip \qquad\qquad (n+1, m,\ Just\ n) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid otherwise = Done \\
\quad\quad\quad next\ (n', m, Just\ n) \quad \mid m \leqslant 19 \quad = Yield\ (n \times m)\ (n', m+1, Nothing) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid otherwise = Done
\end{array}
$$

We are now in a position to apply stream consumer/producer fusion between the stream consumer sum_s and the producer $Stream\ next\ (0, 10, Nothing)$.

First phase: combining consumers with producers

A preliminary step is to unfold the definition of sum_s. Recall the definition of sum_s

$$
\begin{array}{l}
sum_s\ (Stream\ next\ s) = go\ 0\ s \\
\quad \textbf{where} \\
\quad\quad go\ a\ s = \textbf{case}\ next\ s\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad Done \quad\quad\quad\quad \to a \\
\quad\quad\quad\quad\quad\quad\quad Skip \quad\ s' \to go\ a \quad\quad\quad s' \\
\quad\quad\quad\quad\quad\quad\quad Yield\ x\ s' \to go\ (a+x)\ s'
\end{array}
$$

Since the top level of sum_s matches directly on a *Stream* constructor and the top level of a stream producer is a *Stream* constructor then we can unfold sum_s to give just an application of the worker function go

$$
\begin{array}{l}
\quad sum_s\ (Stream\ next\ (0, 10, Nothing)) \\
= \\
\quad go\ 0\ (0, 10, Nothing) \\
\quad\quad \textbf{where} \\
\quad\quad\quad go\ a\ s = \textbf{case}\ next\ s\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad Done \quad\quad\quad\quad \to a \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad Skip \quad\ s' \to go\ a \quad\quad\quad s' \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad Yield\ x\ s' \to go\ (a+x)\ s' \\
\quad\quad\quad next\ (n, m, Nothing) \mid n \leqslant 9 \quad = Skip \qquad\qquad (n+1, m,\ Just\ n) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid otherwise = Done \\
\quad\quad\quad next\ (n', m, Just\ n) \quad \mid m \leqslant 19 \quad = Yield\ (n \times m)\ (n', m+1, Nothing) \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mid otherwise = Done
\end{array}
$$

This gets us to the key part of the first phase: the part where we apply the case-of-case transformation. The *go* worker function scrutinises an application of the stepper function. The stepper function of course satisfies the various producer constraints. We can therefore unfold *next* in the scrutinee position and apply the general case-of-case transformation. This leaves us with

$$
\begin{array}{l}
go\ 0\ (0, 10, Nothing) \\
\quad \textbf{where} \\
\quad\quad \begin{array}{lll}
go\ a\ (n, m, Nothing) & \mid n \leqslant 9 & = go\ a\ (n+1, m, Just\ n) \\
 & \mid otherwise & = a \\
go\ a\ (n', m, Just\ n) & \mid m \leqslant 19 & = go\ (a + n \times m)\ (n', m+1, Nothing) \\
 & \mid otherwise & = a
\end{array}
\end{array}
$$

At this point we have eliminated all the *Step* constructors, however we are still left with the various data constructors used to represent the stream state shapes. In this example each recursive call of *go* allocates two data constructors: the constructor for the 3-tuple and a *Just* or *Nothing* constructor.

Second phase: optimising stream consumption

The second phase of the transformation eliminates these remaining data constructors. We make versions of the function *go* that are specialised to the two patterns of the stream state argument, that is the patterns $(\_, \_, Nothing)$ and $(\_, \_, Just\ \_)$. We will call these specialised versions *go_Nothing* and *go_Just* respectively. The top level call of *go* and all calls in the body are replaced by calls to the appropriate specialised version. This leaves us with

$$
\begin{array}{l}
go\_Nothing\ 0\ 0\ 10 \\
\quad \textbf{where} \\
\quad\quad \begin{array}{lll}
go\_Nothing\ a\ n\ m & \mid n \leqslant 9 & = go\_Just\ a\ (n+1)\ m\ n \\
 & \mid otherwise & = a \\
go\_Just\ a\ n'\ m\ n & \mid m \leqslant 19 & = go\_Nothing\ (a + n \times m)\ n'\ (m+1) \\
 & \mid otherwise & = a
\end{array}
\end{array}
$$

We now have a pair of mutually recursive functions and there are no remaining allocations of constructors used to represent the static state shapes. The dynamic parts of the stream state remain as data parameters passed between the recursive functions. The same pattern is repeated in the general case: the transitions between the static state shapes are compiled into a set of mutually recursive functions, while the variable parts of the state become parameters passed between the functions.

Note that while the allocations for the dynamic parts of the stream state remain, all these allocations are also present in the list version. In comparing with the list version we only have to eliminate the list data constructors while keeping other allocations the same.

4.6.2 Call pattern specialisation

Call pattern specialisation (Peyton Jones, 2007) is a transformation that can optimise programs by removing redundant pattern matching and/or allocations. It is a form of function specialisation and so involves making specialised copies of functions. In general, function specialisation involves restricting a function to a particular value or range of values for some argument and then taking advantage of the extra static information that is available based on the restricted range of inputs. For call pattern specialisation, the way the range of values is restricted is that the values always match a particular pattern of data constructors, a so-called *call pattern*.

For example if we have a function f with an argument of type *Maybe a*

$$
\begin{aligned}
&f :: \mathit{Maybe}\ a \to \ldots \\
&f\ x = \langle \mathit{body} \rangle
\end{aligned}
$$

we could choose to make a version of this function specialised to the call pattern *Just y*.

$$
\begin{aligned}
&f\_\mathit{Just} :: a \to \ldots \\
&f\_\mathit{Just}\ y = \langle \mathit{body} \rangle [x := \mathit{Just}\ y]
\end{aligned}
$$

The relationship between the original general function f and the specialised version f_Just is simply

$$f\ (\mathit{Just}\ y) = f\_\mathit{Just}\ y$$

The opportunity for optimisation is when the body of the function does case analysis on the specialised argument and matches on the special call pattern. The case analysis can be anywhere in the body of the function, it does not have to be at the top level. For the sake of a simple example however, suppose the body of f is

$$
\begin{aligned}
&f :: \mathit{Maybe}\ a \to \ldots \\
&f\ x = \textbf{case}\ x\ \textbf{of} \\
&\qquad\quad \mathit{Nothing} \to \langle \mathit{body\_nothing} \rangle \\
&\qquad\quad \mathit{Just}\ a \quad \to \langle \mathit{body\_just} \rangle
\end{aligned}
$$

With this example the specialised version *f_Just* is

$$
\begin{array}{l}
f\_Just :: a \to \ldots \\
f\_Just\ y = (\textbf{case}\ x\ \textbf{of} \\
\qquad\qquad\qquad Nothing \to \langle body\_nothing \rangle \\
\qquad\qquad\qquad Just\ a \quad\ \to \langle body\_just \rangle\ )[x := Just\ y]
\end{array}
$$

The result of the case analysis becomes known statically and we can reduce the case of the known constructor.

$$f\_Just\ y = \langle body\_just \rangle [a := y]$$

Of course the specialisation is only useful if there are any calls of f where it is known statically that the argument matches the special call pattern and thus where the call to the general version can be replaced with a call of the special version.

Much of the detail of call pattern specialisation is concerned with deciding which specialisations are likely to be profitable. We will cover the detailed conditions shortly. The basic idea is to identify that we make calls with statically known patterns and that those patterns are also matched upon somewhere in the body of the function. We need both elements for specialisation to be profitable. Call pattern specialisation limits the scope in which we look for call instances with static patterns to recursive calls, either directly recursive or mutually recursive. The rationale for limiting the scope is that the analysis becomes local which is cheaper and simpler and secondly the greatest benefit is typically for recursive calls because they correspond to loops.

The general notation for a call pattern is $[v_1, \ldots, v_m] \triangleright [p_1, \ldots, p_n]$. For a function with n arguments it consists of list of n patterns $p_1, \ldots, p_n$ and a set $v_1, \ldots, v_m$ of *pattern variables* that occur in the patterns. In the previous example we had the call pattern *Just y* which we would write in this notation as $[y] \triangleright [Just\ y]$.

Having identified the call patterns, specialisation proceeded by defining specialised versions of the recursive function. For each call pattern, a specialised function is obtained by abstracting over the pattern variables and taking the original function body with the list of patterns substituted in place of the original function arguments.

$$
\begin{array}{l}
g\ \ x_1 \ldots x_n\ = \langle e \rangle \\
g'\ v_1 \ldots v_m = \langle e \rangle [x_1 := p_1, \ldots, x_n := p_n]
\end{array}
$$

In the above example, suppose we identify $[y] \triangleright [Just\ y]$ and $[\,] \triangleright [Nothing]$ as call patterns for which it is worth specialising. The first call pattern has one pattern

variable y while the second has none. We obtain the specialised definitions by abstracting over the pattern variables and substituting the appropriate pattern into the body

$$f\_Just\ y \quad = \langle body \rangle [x := Just\ y]$$
$$f\_Nothing = \langle body \rangle [x := Nothing]$$

Specialisation continues by applying standard optimisations such as case reduction in the bodies of these new functions.

The last phase is to make use of the specialised functions. We have an equational relationship between each specialised version and the original.

$$f\ (Just\ y) = f\_Just\ y$$
$$f\ Nothing = f\_Nothing$$

These equations are then used as rewrite rules, in a left to right direction. They should be used everywhere in the program where the function is in scope. Thus all call instances that use the identified static call patterns will be rewritten to be a call to the appropriate specialised function.

Call pattern specialisation is a general purpose optimisation. That is, it is useful in a fairly wide range of programs. For stream fusion we use it in a rather special-purpose way and the degree of improvement is rather dramatic. In particular when we apply call pattern specialisation to the worker function of a stream consumer, we aim to rewrite *all* calls of the worker function to one of the specialised versions so that the original general version becomes dead code and can be discarded.

Call pattern specialisation preconditions

The correctness of call pattern specialisation does not depend on the choice of call patterns. We have a choice for the precise set of conditions we use to decide which call patterns to specialise on. Peyton Jones (2007, Section 3.3) gives a set of "heuristics" which are tuned to work well in a wide range of programs, balancing the benefits of specialisation with the costs of code duplication. Note that if we commit to using the call pattern specialisation system as described then we must treat these heuristics as conditions.

The conditions Peyton Jones sets out are as follows; for a function f and a call instance $f\ e_1 \ldots e_n$ to be specialisable under the call pattern specialisation system:

H1 *The function f is bound by a definition of the form*

$$f = \lambda x_1 \ldots x_a \to e$$

That is, the lambdas are explicit, and the function has arity $a > 0$.

H2 *The right hand side e is "sufficiently small".*

We rely on a modification to omit this condition, see below for details.

H3 *The function f is recursive, and the specialisable call appears in its right-hand side.*

This is the basic version of the condition. Peyton Jones (2007, Section 4.4) extends this to allow a recursive group of functions and to allow specialisable calls in the body of any function in the group.

H4 *All f's arguments are supplied in the call; that is $n \geqslant a$.*

H5 *At least one of the arguments e_i is a constructor application.*

This is the basic version of the condition. Peyton Jones (2007, Section 4.2) describes an extension which allows specialising on nested structure in the call pattern. With this extension, the argument e_i may be nested constructor applications. In addition there is an extension which allows e_i to be a variable bound to a constructor application (Peyton Jones, 2007, Section 4.1).

H6 *That argument is case-analysed somewhere in the body of f.*

With the nested structure extension, then correspondingly f must **case**-analyse the full nested structure somewhere in the body.

Condition H2 is appropriate when specialisation is used as a general purpose optimisation, however it is problematic for stream fusion. Any upper limit will break the universality of our optimisation argument. In particular it would prevent the elimination of the stream state shape constructors in cases where the producer stepper function is large, e.g. due to composing together several stream transformers.

Our optimisation argument therefore relies on a modification of call pattern specialisation to omit this condition. One practical solution may be to annotate the consumer worker functions with a compiler pragma to remove the limit for just those functions.

Additional notes

Peyton Jones (2007, Section 6.2, 6.3) mentions two areas of further work that are relevant to stream fusion. Firstly that specialising on function arguments (rather than constructor arguments) is possible but tricky and secondly that the algorithm cannot specialise functions where recursive call patterns are via a join point.

Specialising on function arguments is one possible approach to optimising *concatMap*. Recall that our optimisation arguments in this chapter do not extend to higher order stream inputs such as in *concatMap*.

Recall from Section 4.5.8 that we do not allow stream stepper functions to use join points to share construction of *Step* results between multiple branches. The reason for this limitation is now clear: call pattern specialisation cannot handle this form.

4.6.3 Stream consumer constraints

We impose some constraints on stream consumers. As with the constraints on stream producers and transformers, the purpose of most of the constraints is to ensure that the various transformations can be performed successfully. There are additional constraints relating to the allocation accounting argument and to the correct handling of *Skip*.

As before, we do not attempt to find constraints that are both necessary and sufficient. The aim is to specify relatively simple, sufficient, constraints that let us write most common stream consumers. The guide for establishing sufficient constraints is the details of the transformation; by looking at what is needed at each stage of the transformation we aim to find simple constraints that cover all the needs. It is likely that spending more effort on this analysis process would lead us to find more general constraints or more general versions of the transformations.

We could have defined stream consumer/producer fusion so as to allow consumers that consume multiple streams, however for the sake of simplicity, for the present argument we restrict our attention to functions that consume a single stream. Consuming only a single stream does not appear to be a significant practical limitation; most examples where we wish to consume multiple streams can be handled by using a stream transformer to combine multiple streams into a single stream. Should this prove to be a significant limitation, it seems likely that the argument could be generalised to cover consuming multiple streams.

Before describing the constraints we impose, we will first review what is needed for each stage of the transformation.

- At the top level we need to be able to unfold a stream consumer to obtain a case expression that matches a *Stream* constructor.
- We need to be able to apply the case-of-case transformation everywhere the stream stepper function is applied.
- We need worker functions to satisfy the conditions for call pattern specialisation. In particular we need to be able to specialise worker functions on the stream state shapes and all calls to worker functions need to be able to be resolved to one of the specialised versions.
- We need the consumer to handle *Skip* correctly.
- We need the consumer to not duplicate work and allocations performed by the stream producer.

Top level unfolding

Consumer condition 1. At the top level we need to be able to unfold a stream consumer function to get a case expression that matches a *Stream* constructor. This is easy to guarantee if we stipulate that syntactically, the top level of a stream consumer takes one of the following forms

$$f\ x\ a = \textbf{case}\ a\ \textbf{of}\ \mathit{Stream}\ \mathit{next}\ s \rightarrow \ldots$$
$$\quad \textbf{where}\ \ldots$$

or

$$f\ x = h\ (\ldots)$$
$$\quad \textbf{where}\ \ldots$$

Where h is some existing stream consumer. That is, at the top level we allow any mixture of lambda abstraction and **let** binding (or equivalently, **where** clauses). The function body must either match on the input stream to extract the stepper function and initial state, or alternatively it may directly call another stream consumer. The latter alternative allows a very limited degree of abstraction, for example it allows us to define sum_s in terms of $foldl_s$. We have the usual requirement that the collection of stream consumers is finite, that all definitions are available and that where consumers are defined in terms of other consumers, that the definitions do not form cycles.

Case-of-case

Consumer condition 2. Everywhere that the stream stepper function is called, we need to be able to apply the case-of-case transformation so as to eliminate the *Step* constructors. The preconditions for the case-of-case transformation are simple and syntactic (see Section 4.5.7). We can guarantee them if we stipulate that the only permitted use of the stream stepper function is its application to the stream state, as the scrutinee term of a case expression.

case *next s* **of** ...

The case expression must match (and must only match) on all three *Step* constructors. Note therefore that the result of the stream stepper function may not be memoised or passed as a parameter to any function.

Static call patterns

The aim is to specialise worker functions on a stream state parameter, for each of the static state shapes of the stream producer. To be able to apply call pattern specialisation, we need the relevant parts of the stream consumer to satisfy the call pattern conditions set out in Section 4.6.2.

Recall that the basic version of condition H3 states that candidate functions for specialisation be directly recursive, while the extended version allows recursive groups of functions. We take advantage of this extension to allow stream consumers to use several mutually recursive worker functions. Some consumers such as *foldr1* are most naturally expressed using multiple worker functions.

Consumer condition 3. Worker functions may have any number of parameters, of which one must be the stream state. The stream state parameter must be accepted via a top-level lambda. For simplicity we stipulate that the stream state must be an individual parameter to a worker function, rather than being embedded in any other data structure passed to the worker function.

Consumer condition 4. Somewhere within the body, a worker function must scrutinise the result of the stepper function applied to the stream state parameter.

Consumer condition 5. At all worker function call sites, the function must be applied to a stream state variable: either the initial stream state, the input stream state parameter or a new state obtained from the stepper function. This rules out applying any functions to the stream state before calling a worker function. Worker functions do not otherwise need to be fully applied, it is only the the stream state parameter that is essential.

Input skips

Independent of the need to optimise the composition of stream consumers with stream producers, we need to ensure that consumers treat *Skip* correctly. Recall from Chapter 3 that stream consumers (and transformers) must be oblivious to skips. It is straightforward to find a syntactic constraint that ensures that consumers are oblivious to skips.

Consumer condition 6. As stated above, the result of the stream stepper function is scrutinised within the body of each worker function. The intention is that when the result is *Skip* that we 'try again' but with the updated stream state. The most obvious constraint to achieve this is to require that the result in the *Skip* branch be a self-recursive call of the worker function with the same input parameters but with the new stream state parameter. This constraint is only effective however when the case expression itself is in a tail call position, otherwise the result could still be affected by skips. The simplest solution therefore is simply to require that case expressions that scrutinise the result of the stepper function should appear in a tail call position within the body of the worker function.

Note that this constraint does not require that all recursive calls of the worker function be tail recursive. Consider for example $foldr_s$

$$
\begin{array}{l}
foldr_s\ f\ z\ (Stream\ next\ s) = go\ s \\
\quad \textbf{where} \\
\quad\quad go\ s = \textbf{case}\ next\ s\ \textbf{of} \\
\quad\quad\quad\quad\quad Done \quad\quad\ \ \rightarrow z \\
\quad\quad\quad\quad\quad Skip \quad\ s' \rightarrow go\ s' \\
\quad\quad\quad\quad\quad Yield\ x\ s' \rightarrow f\ x\ (go\ s')
\end{array}
$$

The case expression is in a tail call position. In the *Skip* branch the recursive call is tail recursive but the one in the *Yield* branch is not.

Note that this constraint has the consequence that all worker functions are necessarily directly self-recursive, even those that consume no more than the first element from a stream.

Consumer condition 7. Since input streams can skip any number of times we cannot afford to spend any allocations in the case that an input stream skips, otherwise those allocations could be repeated an unbounded number of times. We require therefore that case expressions that scrutinise the result of the stepper function do so in a context that does no allocation.

Effectively this means the top level of worker functions can only be case expressions that pattern-match input parameters. Scrutinising the result of the stepper function in an applicative term would not be tail recursive while a **let** expression may involve allocation. If nested case expressions are used then the scrutinee terms themselves must not involve allocations.

No stream state duplication

For the allocation accounting argument we need to make sure we do not write consumers that lose sharing. Loss of sharing may lead to unbounded duplication of work and of allocations. We will revisit this issue in Section 4.7.11.

When consuming an ordinary list data structure is it possible to inspect the same list tail multiple times. Whatever work was required to calculate the list is done only once irrespective of how many passes are made over the same list. The same is not true for streams. The unfolding of a stream is not memoised or materialised into an in-memory data structure.

Consumer condition 8. To prevent a stream consumer repeating work done by the producer we require that each stream state is passed to the stepper function at most once during any evaluation of a stream consumer. This means the stream state must be used linearly, though stream states may be discarded. For example it is possible to use **case** *next s* **of** multiple times in different branches of a worker function but not in two terms that both contribute to the result.

Consumer condition 9. This leads us to further simplifications, given that the stream state cannot be reused then there is little point in storing or manipulating the stream state. Thus to simplify later arguments we stipulate that the stream state is never stored in any data structure or passed to any function other than a worker function. Similarly, we require each worker function take exactly one stream state parameter.

Note that there is no limitation on storing and sharing stream elements, only on the stream state itself.

Summary grammar

We can approximate the syntactic conditions with the following grammar. There are some additional side condition noted below.

$$
\begin{array}{lll}
\langle good\_consumer \rangle & ::= & f = \langle top \rangle \\
\langle top \rangle & ::= & \lambda a \to \langle top \rangle \\
& | & \textbf{let} \ldots \textbf{in} \ \langle top \rangle \\
& | & h \ \{ \langle expr \rangle \} \\
& | & \textbf{case} \ a \ \textbf{of} \ Stream \ next \ s \to \\
& & \quad \textbf{let} \quad go_0 \ s \ [a \ b \ldots] = \langle worker\_top \rangle \\
& & \qquad\quad go_1 \ s \ [a \ b \ldots] = \langle worker\_top \rangle \\
& & \qquad\quad \ldots \\
& & \quad \textbf{in} \ go_0 \ s \\
\langle worker\_top \rangle & ::= & \textbf{case} \ next \ s \ \textbf{of} \\
& & \quad Done \ \to \langle worker\_result \rangle \\
& & \quad Skip \ s' \ \to go_x \ s' \\
& & \quad Yield \ a \ s' \to \langle worker\_result \rangle \\
& | & \textbf{case} \ a \ \textbf{of} \\
& & \quad \ldots \to \langle worker\_top \rangle \\
& & \quad \ldots \\
& | & \langle worker\_result \rangle \\
\langle worker\_result \rangle & ::= & a \\
& | & \lambda a \to \langle worker\_result \rangle \\
& | & \langle worker\_result \rangle \ \langle worker\_result \rangle \\
& | & \textbf{let} \ldots \textbf{in} \ \langle worker\_result \rangle \\
& | & \textbf{case} \ \langle worker\_result \rangle \ \textbf{of} \\
& & \quad \ldots \to \langle worker\_result \rangle \\
& & \quad \ldots \\
& | & \langle worker\_call \rangle \\
\langle worker\_call \rangle & ::= & go_x \ s \ \ \{ \langle expr \rangle \} \\
& | & go_x \ s' \ \{ \langle expr \rangle \}
\end{array}
$$

There are a number of side conditions:

- In $\langle top \rangle$, h must refer to another good stream consumer.
- In $\langle worker\_top \rangle$, the $go_x \ s'$ must be the same go_x function that the expression appears in. That is, it is a self-recursive call.
- In $\langle worker\_result \rangle$ expressions, only one $\langle worker\_call \rangle$ is allowed anywhere within the expression.

4.6.4 Combining consumers with producers

The transformation to combine stream consumers with stream producers is almost identical to that described in Section 4.5.5 for combining stream transformers with stream producers. The argument for why the transformation is always possible is also near identical, relying on the syntactic constraints on consumers and producers.

The general case we consider is a stream consumer applied to a stream producer.

$$\mathit{consumer}\ (\mathit{producer}\ldots)$$

We can assume that the stream consumer and producer satisfy the respective constraints. Note that we never have to deal with a stream consumer that is partially applied in its stream argument, or applied to a stream term that is an abstracted variable (see Sections 4.2.3 and 4.3.1).

Top level unfolding

The first step is to unfold the stream producer to expose a *Stream* constructor and to unfold the stream consumer to expose a case expression that matches on a *Stream* constructor. This step relies on the constraints on the top-level syntactic structure of stream producers and consumers. Previously, in Section 4.5.5 we argued that the constraints on producers ensure that a finite sequence of unfolding producer definitions always exposes a *Stream* constructor.

$$\begin{array}{ll} & \mathit{producer}\ldots \\ = & \quad \{\ \text{unfold the producer}\ \} \\ & \mathbf{let}\ \mathit{next}_a\ s = \ldots \\ & \qquad \ldots \\ & \mathbf{in}\ \mathit{Stream}\ \mathit{next}_a\ \langle \mathit{initial} \rangle \end{array}$$

The *Stream* constructor may be wrapped in several layers of **let** expressions that supply local function definitions, including the stepper function.

The argument for consumers is analogous to the previous argument about transformers. In particular the body of the consumer is either is directly in the form we need – a case expression that matches a *Stream* constructor – or it is a call to another consumer (consumer condition 1). Thus a finite sequence of unfoldings gives us the case expression, possibly wrapped in layers of **let** expressions supplying local definitions.

$$
\begin{array}{ll}
 & \mathit{consumer}\ a \\
= & \quad \{\ \text{unfold the consumer}\ \} \\
 & \mathbf{let}\ \mathit{worker}\ s = \langle \mathit{body} \rangle \\
 & \mathbf{in\ case}\ a\ \mathbf{of}\ \mathit{Stream}\ \mathit{next}\ s \to \langle \mathit{top} \rangle
\end{array}
$$

Note that the consumer will always be fully applied to its stream argument, but it is not necessary for it to be fully applied to all other arguments.

When we unfold the consumer we get the stream producer term in the scrutinee position of the top level case expression.

$$
\begin{array}{l}
\mathbf{let}\ \mathit{worker}\ s = \langle \mathit{body} \rangle \\
\mathbf{in\ case}\ (\mathbf{let}\ \mathit{next}_a\ s = \ldots \\
\qquad\qquad\qquad \ldots \\
\qquad\qquad \mathbf{in}\ \mathit{Stream}\ \mathit{next}_a\ \langle \mathit{initial} \rangle) \\
\quad \mathbf{of}\ \mathit{Stream}\ \mathit{next}\ s \to \langle \mathit{top} \rangle
\end{array}
$$

The local **let** definitions for the producer can be floated outwards which allows us to reduce the top level case expression of the consumer since we have a case of a known constructor.

$$
\begin{array}{l}
\mathbf{let}\ \mathit{worker}\ s = \langle \mathit{body} \rangle [\mathit{next} := \mathit{next}_a] \\
\quad \mathit{next}_a \quad s = \ldots \\
\quad \ldots \\
\mathbf{in}\ \langle \mathit{top} \rangle [s := \langle \mathit{initial} \rangle]
\end{array}
$$

We are left with the body term of the stream consumer's case expression but with the stream producer's terms and/or local definitions for the initial state and stepper function. In particular, the local definition for the stepper function is now available in the consumer's worker function.

Applying case-of-case

The next step is to apply the case-of-case optimisation in the body of the consumer's worker functions so as to eliminate all the *Step* constructors. We rely on consumer constraint 2 that the only occurrences of the input stepper function are applications to a stream state in the scrutinee position of a case expression. Additionally, the previous step of unfolding and reducing the consumer and producer has given us a local definition for the input stepper function. Finally, stream producer conditions 3 and 6 guarantee that the stepper function is a tree of case expressions that return *Step* constructors. These conditions guarantee

that we can unfold each occurrence of the stepper function and apply the general case-of-case transformation (Section 4.5.7) to eliminate the *Step* constructors used in the result of the stepper function.

This much is analogous to the transformation and argument in the stream transformer/producer situation. One difference is that we do not need to re-establish conditions on a new stepper function, since unlike transformers, consumer do not produce a new stepper function. On the other hand there are some properties of the new worker functions that we will rely upon in the next stage when we come to apply call pattern specialisation. We will describe the details in the next section.

4.6.5 Optimising stream consumption

Recall the example from Section 4.6.1.

$$
\begin{array}{l}
go\ 0\ (0, 10, Nothing) \\
\quad \textbf{where} \\
\quad\quad \begin{array}{lll}
go\ a\ (n, m, Nothing) & \mid n \leqslant 9 & = go\ a\ (n+1, m, Just\ n) \\
 & \mid otherwise & = a \\
go\ a\ (n', m, Just\ n) & \mid m \leqslant 19 & = go\ (a + n \times m)\ (n', m+1, Nothing) \\
 & \mid otherwise & = a
\end{array}
\end{array}
$$

This example is typical in that it uses a single worker function. At this stage of the transformation we have one or more worker functions, each of which takes a stream state parameter. Each worker function does at least one case analysis on the input stream state that matches it against all the state shapes; in the above example the shapes $(\_, \_, Nothing)$ and $(\_, \_, Just\ \_)$. In the recursive calls to the worker functions, the term passed as the stream state parameter is an application of data constructors that matches one of the stream state shapes. The aim is to transform each worker function and all the calls to the worker functions so that the data constructors used for the state shapes are eliminated.

$$
\begin{array}{l}
go\_Nothing\ 0\ 0\ 10 \\
\quad \textbf{where} \\
\quad\quad \begin{array}{lll}
go\_Nothing\ a\ n\ m & \mid n \leqslant 9 & = go\_Just\ a\ (n+1)\ m\ n \\
 & \mid otherwise & = a \\
go\_Just\ a\ n'\ m\ n & \mid m \leqslant 19 & = go\_Nothing\ (a + n \times m)\ n'\ (m+1) \\
 & \mid otherwise & = a
\end{array}
\end{array}
$$

One can think of the original as encoding control flow as data, and that the transformation converts this back to ordinary control flow in the form of mutually recursive functions.

To achieve this transformation by call pattern specialisation would require that we:

1. specialise each worker function on its stream state argument;
2. specialise using all the stream state shapes as call patterns;
3. rewrite all calls to worker functions into calls to an appropriate specialised version.

As we will describe in the remainder of this section, the details are almost but not quite as simple as this.

Why call pattern specialisation

Call pattern specialisation is not the only transformation that would be effective here. More general techniques such as partial evaluation, or more special-purpose transformations would also likely be effective. There are good reasons to prefer call pattern specialisation:

- it is an existing, relatively simple and well documented transformation;
- it is a general purpose optimisation that can benefit a range of programs, which helps to justify the effort of implementing it in a general purpose compiler;
- our previous work (Coutts et al., 2007b) demonstrates that it works in practice;
- it is preferable that the argument we present here should use the same, or a very similar, transformation as that used in previous empirical work.

On the other hand, there are downsides to using a general purpose optimisation. The standard heuristics for deciding which call patterns to specialise upon are tuned to work well in a wide range of programs. In our application there are corner cases where the standard heuristics miss some of the call patterns that we would wish to specialise on.

In the remainder of this section we will describe how the standard heuristics cover the usual case and what corner cases they do not cover. We will also present two solutions to cover the corner cases: one using different specialisation heuristics and one using an additional subsequent transformation.

Note that when the stream producer uses a single trivial state shape then there is nothing to do: there are no allocations to eliminate. It is possible to use stream fusion while forgoing the transformation that eliminates the state shapes, but only if we have a restriction that all stream producers and transformers use only a single state shape. We would be restricted to simple stream functions like map_s and $filter_s$ that do not make use of multiple state shapes. This restriction was implicit in our earliest work on stream fusion for arrays (Coutts et al., 2007a).

Properties of worker functions

Given a function that we wish to specialise, the argument to specialise it on and the set of call patterns to specialise against, the key conditions for call pattern specialisation involve:

1. looking at what patterns the function argument is matched against in the body of the function;
2. looking at what call patterns are used in recursive call instances.

For a pattern to be a candidate for specialisation it must both be matched against in the body and it must be used in a recursive call instance. With this in mind, it is worth considering the form of the worker functions at this stage of the transformation; that is, the stage after combining consumers with producers and applying case-of-case. In particular we are interested in the relationship between 1) the state shapes of the stream producer and 2) the call patterns matched in the body, and patterns used in call instances of the worker functions.

Let us consider what happens when we combine a worker function with a stepper function and apply the case-of-case transformation. The part of the the worker function we are particularly interested in is the part where it scrutinises *next s*. Recall that consumer condition 4 guarantees that *next s* is scrutinised somewhere in the body of the worker function. Each such occurrence has the following general form

$$
\begin{array}{l}
\textbf{case } \mathit{next}\ s \textbf{ of} \\
\quad \begin{array}{lll}
\mathit{Done} & & \rightarrow \langle \mathit{exp}_{\mathit{done}} \rangle \\
\mathit{Skip} & s' & \rightarrow \langle \mathit{exp}_{\mathit{skip}} \rangle \\
\mathit{Yield}\ x & s' & \rightarrow \langle \mathit{exp}_{\mathit{yield}} \rangle
\end{array}
\end{array}
$$

The other part we are interested in is the stepper function from the stream producer. Recall that stepper functions take the form of a tree of case expressions

with *Done*, *Skip*, or *Yield* terms in the leaves. For the most part we are interested only in the leaf terms of the tree of case expressions, not in the interior nodes. Imagine a typical stepper function of the following form. The interior nodes are elided.

$$
\begin{array}{l}
\textbf{case} \ldots \textbf{of} \\
\quad \ldots \to \mathit{Skip} \quad\quad \langle \mathit{state}_1 \rangle \\
\quad \ldots \to \mathit{Done} \\
\quad \ldots \to \mathit{Skip} \quad\quad \langle \mathit{state}_2 \rangle \\
\quad \ldots \to \mathit{Skip} \quad\quad \langle \mathit{state}_3 \rangle \\
\quad \ldots \to \mathit{Yield}\ \langle \mathit{val}_4 \rangle\ \langle \mathit{state}_4 \rangle
\end{array}
$$

In general there can be any number of *Done*, *Skip* or *Yield* leaf terms. Recall that stream producer condition 6 states that in the *Skip* and *Yield* terms, the new state must manifestly be one of the state shapes. That is, each $\langle \mathit{state}_x \rangle$ term matches one of the producer's state shapes.

When we combine the above two parts together and we apply the case-of-case transformation then we obtain an expression of the form

$$
\begin{array}{l}
\textbf{case} \ldots \textbf{of} \\
\quad \ldots \to \langle \mathit{exp}_{\mathit{skip}} \rangle \ \ [s' := \langle \mathit{state}_1 \rangle\,] \\
\quad \ldots \to \langle \mathit{exp}_{\mathit{done}} \rangle \ [\ ] \\
\quad \ldots \to \langle \mathit{exp}_{\mathit{skip}} \rangle \ \ [s' := \langle \mathit{state}_2 \rangle\,] \\
\quad \ldots \to \langle \mathit{exp}_{\mathit{skip}} \rangle \ \ [s' := \langle \mathit{state}_3 \rangle\,] \\
\quad \ldots \to \langle \mathit{exp}_{\mathit{yield}} \rangle \ [s' := \langle \mathit{state}_4 \rangle, x := \langle \mathit{val}_4 \rangle\,]
\end{array}
$$

That is we have the body of the stepper function but with new leaf terms. The new leaf terms are the $\langle \mathit{exp}_{\mathit{done}} \rangle$, $\langle \mathit{exp}_{\mathit{skip}} \rangle$, $\langle \mathit{exp}_{\mathit{yield}} \rangle$ expressions from the worker function with the appropriate substitutions for the stream state and yielded value.

We can now make a couple observations about the form of the worker function at this stage of the transformation. Consider the outline of a typical worker function

$$
\begin{array}{l}
\mathit{go}\ s = \textbf{case}\ \mathit{next}\ s\ \textbf{of} \\
\qquad\qquad \ldots \\
\qquad\qquad \mathit{Skip} \quad s' \to \mathit{go}\ s' \\
\qquad\qquad \mathit{Yield}\ x\ s' \to \ldots (\mathit{go}\ s') \ldots
\end{array}
$$

After unfolding the stepper function and applying the case-of-case transformation, these recursive call instances will all be of the form $(\mathit{go}\ s')[s' := \langle \mathit{state}_x \rangle\,]$, or simply $\mathit{go}\ \langle \mathit{state}_x \rangle$. More generally, and more precisely, we can describe the set

of these recursive call instances that we obtain after applying case-of-case. The set is described in terms of the Cartesian product of the set of such recursive call instances in the original worker function and the set of $\langle state_x \rangle$ terms produced by the stepper function.

Property 4.6.1. Let $calls_{yield}$ and $calls_{skip}$ be the set of all recursive call instances in the original worker function that use the new state s' as obtained from *Yield* and *Skip* respectively. Let $states_{yield}$ and $states_{skip}$ be the set of $\langle state_x \rangle$ state terms that appear in *Yield* and *Skip* results of the stepper function. For each $call_{yield}$ in $calls_{yield}$ and $state_{yield}$ in $states_{yield}$, the new worker has a recursive call $\langle call_{yield} \rangle [s' := \langle state_{yield} \rangle]$. Similarly for $calls_{skip}$ and $states_{skip}$.

Notice that the use of the Cartesian product in the description takes into account the corner case where there are no such recursive calls or no such state terms. For example, the stepper function may not produce a *Skip* term at all. In this situation there will be no $\langle exp_{skip} \rangle$ leaf terms in the result. Similarly if the original worker does not contain a recursive call that uses a new stream state obtained from *Yield*, then there will be no corresponding recursive calls after the case-of-case transformation. Notice in particular that while the original worker is required to scrutinise *next s* and handle the case of *Skip* s' by recursing with the new state s', there is no guarantee that the stepper function actually produces any *Skip* terms.

If instead of looking at the set of all state terms $\langle state_x \rangle$, we look just at the set of state shapes used by the terms $\langle state_x \rangle$, then taking the Cartesian product gives us the set of calls and call patterns in the new worker.

Let us now consider what patterns are matched in the body of each worker function.

Property 4.6.2. The body of each worker function matches its stream state argument against all the state shapes used by the stream producer.

We note that that all original workers functions scrutinise *next s* somewhere in their body. We note that the transformed expression contains the same tree of case expressions as the the stepper function, albeit with different leaf terms. Finally, recall that producer condition 4 states that the stepper function matches its input stream state against all the stream's state shapes. Thus the transformed expression matches *s* against all the state shapes.

Applying call pattern specialisation

As stated previously, the aim is to transform the worker functions to eliminate the data constructors used to represent the stream state shapes. Achieving this by specialisation would require that we 1) specialise each worker function on its stream state parameter, 2) using all the stream state shapes as call patterns and 3) that we rewrite all calls to worker functions to be calls to an appropriate specialised version.

For 1) and 2) we would need each worker function to satisfy the conditions for call pattern specialisation on all the stream state shapes. For 3) we need all calls to worker functions to use a call pattern that matches one of stream state shapes.

Let us assume for the time being that we can select all the stream state shapes as the call patterns on which to specialise each worker function. Let us see how the transformation proceeds under this assumption. We will return to the issue of whether and to what extent we can justify the assumption.

Specialisation proceeds by making copies of the worker functions, one copy for each worker function for each state shape. In place of the stream parameter, each one takes the variables in the state shape as extra parameters. The body of each copy has the state parameter substituted for the state shape pattern.

The next stage is to rewrite calls of the original worker functions into calls to the appropriate specialised version. There are three classes of call instance to consider. We know from condition 5 that there were three classes at the previous stage of the transformation and although the call instances are affected by the case-of-case transform, the classes of call remain distinct. Condition 5 tells us that the possible kinds of calls to a worker function are:

1. *top-level calls*: calls to a worker function in the top level of the consumer;
2. *recursive calls, same state*: recursive calls in the body of a worker function that use the stream state passed into the worker function;
3. *recursive calls, new state*: recursive calls in the body of a worker function that use the new state obtained from *next*, from either a *Skip* or a *Yield*.

All three classes are illustrated in the following example stream consumer.

$$
\begin{array}{l}
f\ (\mathit{Stream}\ \mathit{next}\ s) = \mathit{go}\ s\ a_0 \\
\quad \textbf{where} \\
\quad\quad \mathit{go}\ s\ a \mid p\ a \qquad\quad = \ldots (\mathit{go}\ s\ a') \ldots \\
\quad\quad\quad\quad \mid \mathit{otherwise} = \textbf{case}\ \mathit{next}\ s\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{Done} \quad\quad \rightarrow \ldots \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{Skip} \quad s' \rightarrow \mathit{go}\ s'\ a \\
\quad\quad\quad\quad\quad\quad\quad\quad \mathit{Yield}\ x\ s' \rightarrow \ldots (\mathit{go}\ s'\ a') \ldots
\end{array}
$$

For top-level calls we note that producer condition 9 requires that the initial stream state term must manifestly be one of the state shapes. Consumer condition 5 requires that top level worker function calls use the initial stream state variable directly. Once the stream consumer is combined with the stream producer then the consumer's initial stream state variable is bound to the producer's initial stream state term – which itself matches one of the state shapes. So the top level call sites do not necessarily directly use a term that matches a state shape, instead they may use a variable which in turn is bound to a term matching a state shape. In this situation we rely on the extension of call pattern specialisation for variables that have known structure (Peyton Jones, 2007, Section 4.1). In particular this extension ensures that the top-level worker function calls using the initial stream state variable can still be rewritten into a call to the appropriate specialised worker function.

For recursive calls that use a new stream state we can rely on consumer condition 5. This guarantees all such recursive calls in the worker function use a state term produced by the stepper function – which must therefore match a stream state shape. Thus all these calls in the original and specialised versions of the worker functions can be rewritten to calls to specialised versions of the worker functions.

The last case is recursive calls that use the stream state passed into the worker function. In the specialised versions of the worker functions the state parameter used in the recursive call has been substituted for the call pattern the worker function was specialised on, so it trivially uses a call pattern matching a state shape. For reasons that will shortly become clear, occurrences in the non-specialised worker functions do not need to be considered.

Having performed the rewrites of the above calls, the non-specialised worker functions are now dead code and can be eliminated. This is because they are no longer called from the top level and they are not called from any of the specialised worker functions.

Overall, if we can select all state shapes as call patterns then we can rewrite all worker function calls to calls to specialised versions. Thus all the constructors allocated for the state shapes at all worker call sites can be eliminated.

Satisfying the conditions for call pattern specialisation

To apply specialisation in the way we desire, we would need each worker function to satisfy all the conditions for call pattern specialisation on all the stream state shapes.

The call pattern specialisation conditions H1–H6 are set out in Section 4.6.2. The easy conditions are H1, H2 and H4. The interesting conditions are H3, H5 and H6.

Condition H1 requires the worker functions to have lambdas at the top level, while condition H4 requires recursive calls to the worker function to supply at least as many arguments as there are top-level lambdas. Since we are only interested in specialising the worker function on its stream state parameter then at a minimum, at the top level, we require a single lambda for the state, while at call sites we require a state argument to be supplied. We can guarantee both since by condition 3 we require worker functions to take at least the stream state as a parameter via a top-level lambda and by condition 5 we require call sites to at least supply an argument for the state parameter. As described previously, we omit condition H2.

For condition H5 we rely on the extension to call pattern specialisation that allows call patterns to be nested structures. We need the extensions because stream state shapes are compound. In particular we can end up with compound state shapes from the composition of stream transformers.

H6 requires that the function scrutinise the argument being specialised on and match the argument against the call patterns of interest; that is, the worker function must match the stream state parameter against all the stream state shapes, including the full nested structure. We established this previously with property 4.6.2.

H3 and H5 require that specialisable calls, using the call patterns of interest, must appear in the body of the function, or another function in the same recursive binding group. Property 4.6.1 describes the set of call instances that appear in the body of each worker function. Recall that it defines:

- $calls_{yield}$ and $calls_{skip}$ as the set of all recursive calls in the original worker function that use the new state s' as obtained from *Yield* and *Skip* respectively.
- $states_{yield}$ and $states_{skip}$ as the set of state terms that appear in *Yield* and *Skip* results of the stepper function.

- The sets of call instances that appear in the body of a worker function are defined in terms of the products $calls_{yield} \times states_{yield}$ and $calls_{skip} \times states_{skip}$.

We want these product sets to cover all the state terms, because we cannot otherwise hope to cover all the state shapes. Thus we need the $calls_{yield}$ and $calls_{skip}$ sets to be non-empty. The $calls_{skip}$ set is guaranteed to be non-empty because each worker function must scrutinise *next* s and it must handle *Skip* s' by recursing with the new state s'.

Corner cases

The above description in terms of product sets highlights why the preconditions can usually be met and the cases where they will not be met. There are two cases in which the above product sets may fail to cover all the stream's state shapes:

1. The terms in the $states_{yield}$ and $states_{skip}$ may not cover all of the stream's state shapes.

2. The $calls_{yield}$ set may be empty.

In addition, if both $states_{yield}$ and $states_{skip}$ are empty – that is the stepper function only produces *Done* – then although we may trivially cover all the state shapes, it will also usually lead to a non-recursive worker function, but condition H3 requires candidate functions to be recursive.

For the first corner case, consider the state machine view of a stream stepper function. Each transition arrow originating at one node and pointing to another node corresponds to the stepper function producing a *Skip* or *Yield* with a new state matching one of the state shapes. In this view we see that the state shapes covered by the $states_{yield}$ and $states_{skip}$ sets are those that have a transition arrow pointing to them. Note however that the initial stream state is not such a transition. In the usual case, the state machine transition graph is such that all nodes have an incoming transition from another node. However it is possible that the initial node in the state machine is pointed to only from the initial state and not by a node-to-node transition. This is the case for example with $cons_s$. The state machine for $cons_s$ has two nodes, an initial node and a main node. The only transition from the initial node is to the main node. The initial stream state identifies the initial node, however the corresponding worker function call in the consumer appears at the top level rather than within the body of one of the

worker functions. It is thus not considered when looking for call patterns to specialise upon.

In the second corner case there are no recursive calls using an updated stream state obtained from calling *next*. For example a stream consumer may look only at the first element in a stream. It is therefore possible to miss state shapes in the transition graph that were only reachable via a *Yield* transition.

Solutions

The most direct solution is simply to change the conditions we use for deciding which call patterns to specialise upon. Recall that the choice of call patterns is not important for correctness (see Section 4.6.2). We can simply declare that we choose all the stream state shapes as call patterns to specialise on. We would of course be giving up on the stated aim of using general purpose optimisations.

The second solution is to use call pattern specialisation with the standard conditions but to handle the remaining corner cases by the combination of a little analysis, a refinement of the goal and some additional transformations.

Firstly, consider the second corner case where we may not be able to specialise the worker function on state shapes that were only reachable via a *Yield* transition. This actually is not a problem. We do not actually need to specialise on all the stream state shapes, more precisely the set of patterns we specialise on needs to cover all the calls in the body of the worker function. For example if we have $head_s\ (cons_s\ x\ \langle remainder \rangle)$ then the worker function does not recurse in the *Yield* case, so there are no calls that use any of the state shapes of the remainder of the stream.

A little further analysis should convince us that all the remaining corner cases involve worker functions that are non-recursive. This means that although we cannot specialise, we can use simple unfolding and reduction.

In the first corner case the situation is that the initial node in the state machine is pointed to only from the initial state and not from a node-to-node transition. This means the state shape of the initial node is not used as a call pattern to specialise on. Thus the top-level worker function call that uses the initial call pattern will not be rewritten to a call to a specialised version. All other calls to the worker function will be rewritten however, both in the generic version and the specialised versions. Thus the generic version will no longer be recursive and is only called from the top level.

Similarly, we already noted above that when both $states_{yield}$ and $states_{skip}$ are empty then the worker function does not recurse with a new state. Depending

on whether there are other recursive calls we may or may not be able to eliminate the constructors used for the initial stream state. This is all benign however because it is constant and does not affect the number of constructors allocated per stream element.

4.6.6 Weaker state shape matching

Recall from the overview in Sections 4.3.2 and 4.3.3 that overall we have an inductive argument involving repeatedly fusing transformers with producers to give a new producer and finally taking the remaining producer and fusing it with a consumer. The argument makes use of a set of properties about the stream producers that we have at each stage. We have some potential for flexibility in the choice of properties and in the choice of transformations we use to maintain the properties; the only constraint is that the properties are strong enough at the end to ensure that the consumer/producer fusion will work.

As we discussed in Section 4.5.3, based on exploring simple examples we are naturally led to try picking properties that make stream producers equivalent to the state machines of Section 4.4, and using simple unfolding and case-of-case as the transformation to preserve the state machine form in the transformer/producer fusion step. More complicated examples (involving zip_s) exposed the problem that the case-of-case transformation on its own is not quite enough to preserve the state machine form. We concluded from this failure that either the properties are too strong or that the transformation is not sufficient.

In Section 4.5.3 we pursued the simpler option of using an additional transformation to preserve the original choice of properties. In this section we will explore the other option: that of weakening the properties and sticking to using just the case-of-case transformation. We are interested in this approach because it corresponds more closely to what real implementations of stream fusion do. It seems intuitive that the case-of-case transformation should preserve some kind of weaker property. The question is quite what such a property should be and whether any such property is strong enough to ensure that consumer/producer fusion will still work.

We will start our exploration by returning to the zip_s example and see whether the form we are left with after the simple case-of-case transformation can still be successfully fused with a consumer.

We start with the fused stepper function after applying case-of-case.

$$
\begin{aligned}
next\ (s_a, s_b, Nothing) &= \textbf{case}\ s_a\ \textbf{of} \\
&\qquad Left\quad n \rightarrow Skip\ (Right\ (n+1), s_b, Just\ n) \\
&\qquad Right\ n \rightarrow Skip\ (Left\quad (n-1), s_b, Just\ n) \\
next\ (s'_a, s_b, Just\ n) &= \textbf{case}\ s_b\ \textbf{of} \\
&\qquad Left\quad m \rightarrow Yield\ (n, m)\ (s'_a, Right\ (m+1), Nothing) \\
&\qquad Right\ m \rightarrow Yield\ (n, m)\ (s'_a, Left\quad (m-1), Nothing)
\end{aligned}
$$

Note that s_a is only scrutinised in the first mode, and s_b only in the second. So it is not the case that the full structure of each state shape is matched in each of the two modes. It is however the case that each *part* of the shape is matched by *some* mode. To put it another way, the full shape is covered by the combination of the two modes.

Let us blindly plough ahead with consumer/producer fusion. For simplicity we will use *unstream* as the consumer. Having applied *unstream* then after the case-of-case phase we obtain the following worker function.

$$
\begin{aligned}
go\ (s_a, s_b, Nothing) &= \textbf{case}\ s_a\ \textbf{of} \\
&\qquad Left\quad n \rightarrow \qquad go\ (Right\ (n+1), s_b,\ Just\ n) \\
&\qquad Right\ n \rightarrow \qquad go\ (Left\quad (n-1), s_b,\ Just\ n) \\
go\ (s'_a, s_b, Just\ n) &= \textbf{case}\ s_b\ \textbf{of} \\
&\qquad Left\quad m \rightarrow (n, m) : go\ (s'_a, Right\ (m+1), Nothing) \\
&\qquad Right\ m \rightarrow (n, m) : go\ (s'_a, Left\quad (m-1), Nothing)
\end{aligned}
$$

It is now time to apply call pattern specialisation. There are four recursive call sites, giving rise to four call patterns.

$$
\begin{aligned}
&n', s_b, n \triangleright (Right\ n', s_b,\ Just\ n) \\
&n', s_b, n \triangleright (Left\quad n', s_b,\ Just\ n) \\
&s'_a, m' \quad \triangleright (s'_a, Right\ m', Nothing) \\
&s'_a, m' \quad \triangleright (s'_a, Left\quad m', Nothing)
\end{aligned}
$$

Recall that call pattern specialisation filters the call patterns with the aim of keeping only those where specialisation is likely to be profitable. Peyton Jones (2007, Section 4.2) describes the exact test used when dealing with patterns with nested structure. All four patterns above do pass the test. It is worth seeing why they pass the test as it is a somewhat subtle and fragile property.

The test involves collecting argument usage information. For an argument that is scrutinised, the usage information records, for each possible alternative data constructor, which of its fields are scrutinised and their usage information

recursively. The usage information is collected for the function as a whole. Peyton Jones uses the following notation for the usage information of a function parameter.

$$\left[\begin{array}{lcl} Constructor_a & \mapsto & [\langle field_1\rangle, \langle field_2\rangle, \ldots] \\ Constructor_b & \mapsto & [\langle field_1\rangle, \langle field_2\rangle, \ldots] \\ \vdots & & \end{array}\right]$$

In this notation there is an entry for each of the possible data constructors for the type. For each constructor there is a list for the fields of the constructor. Each $\langle field_1\rangle$ is either $\diamond$ if that field is not used, or a further nested usage notation.

In our example it is clear that the $(,,)$-constructor itself is used. The third component of the tuple is also clearly scrutinised. The first clause of *next* scrutinises s_a which is the first component of the tuple. The second clause of *next* scrutinises s_b, the second component of the tuple. Merging this information together into usage for the function as a whole tells us that all three components of the tuple are scrutinised. Using the above notation we write it as follows.

$$\left[\begin{array}{lcl} (,,) \mapsto & \left[\begin{array}{lcl} Left & \mapsto & [\diamond] \\ Right & \mapsto & [\diamond] \end{array}\right], & \\ & \left[\begin{array}{lcl} Left & \mapsto & [\diamond] \\ Right & \mapsto & [\diamond] \end{array}\right], & \\ & \left[\begin{array}{lcl} Just & \mapsto & [\diamond] \\ Nothing & \mapsto & [] \end{array}\right] & \end{array}\right]$$

If we now use this usage information to filter the set of call patterns we identified previously then we find that all the patterns are retained because each component of the $(,,)$-tuple are scrutinised somewhere in the function body.

We can now move on to generating specialised versions of the function. We omit the details of the substitution. The specialisations for the four call patterns are

$$\begin{array}{l} \quad \{ \text{ specialisation for } (Right\ n', s_b, Just\ n) \ \} \\ go_1\ n'\ s_b\ n = \textbf{case}\ s_b\ \textbf{of} \\ \qquad\qquad Left\quad m \to (n,m) : go\ (Right\ n', Right\ (m+1), Nothing) \\ \qquad\qquad Right\ m \to (n,m) : go\ (Right\ n', Left\quad (m-1), Nothing) \end{array}$$

$$\begin{array}{l} \quad \{ \text{ specialisation for } (Left\ n', s_b, Just\ n) \ \} \\ go_2\ n'\ s_b\ n = \textbf{case}\ s_b\ \textbf{of} \\ \qquad\qquad Left\quad m \to (n,m) : go\ (Left\ n', Right\ (m+1), Nothing) \\ \qquad\qquad Right\ m \to (n,m) : go\ (Left\ n', Left\quad (m-1), Nothing) \end{array}$$

$$
\begin{aligned}
&\{ \text{ specialisation for } (s'_a, \mathit{Right}\ m', \mathit{Nothing}) \ \} \\
&go_3\ s'_a\ m' = \mathbf{case}\ s'_a\ \mathbf{of} \\
&\qquad\qquad \mathit{Left}\quad n \to go\ (\mathit{Right}\ (n+1), \mathit{Right}\ m', \mathit{Just}\ n) \\
&\qquad\qquad \mathit{Right}\ n \to go\ (\mathit{Left}\quad (n-1), \mathit{Right}\ m', \mathit{Just}\ n)
\end{aligned}
$$

$$
\begin{aligned}
&\{ \text{ specialisation for } (s'_a, \mathit{Left}\ m', \mathit{Nothing}) \ \} \\
&go_4\ s'_a\ m' = \mathbf{case}\ s'_a\ \mathbf{of} \\
&\qquad\qquad \mathit{Left}\quad n \to go\ (\mathit{Right}\ (n+1), \mathit{Left}\ m', \mathit{Just}\ n) \\
&\qquad\qquad \mathit{Right}\ n \to go\ (\mathit{Left}\quad (n-1), \mathit{Left}\ m', \mathit{Just}\ n)
\end{aligned}
$$

The interesting thing here is that the bodies of the specialised versions contain new call patterns. In fact these new call patterns are exactly the eight state shapes that we assigned for the fused stream producer.

We can make use of these new call patterns by using yet another extension to call pattern specialisation. Peyton Jones (2007, Section 4.2) describes an extension where specialisation is iterated to a fixpoint. New call patterns are collected from the bodies of specialised functions, then the original function is specialised again with these new call patterns.

An important property is that amongst the specialised copies, the call patterns appearing in their bodies are always at least as specialised. This means that when we generate the eight new specialisations, their bodies contain call patterns that refer to each other but not to the four previous partially specialised versions. When we replace all calls to the original version with calls to appropriate specialised versions, then the eight fully specialised versions only call each other. The four partially specialised versions call the fully specialised versions, but not the other way around. Since the initial call is to a fully specialised version then this means that the original and partially specialised versions are dead code and can be eliminated. The final result is optimal:

$$
\begin{aligned}
&\{ \text{ specialisation for } (\mathit{Right}\ n, \mathit{Right}\ m, \mathit{Nothing}) \ \} \\
&go_5\ n\ m \qquad = go_{10}\ (n-1)\ m\ n \\
&\vdots \\
&\{ \text{ specialisation for } (\mathit{Left}\ n', \mathit{Left}\ m, \mathit{Just}\ n) \ \} \\
&go_{12}\ n'\ m\ n = (n, m) : go_6\ n'\ (m+1)
\end{aligned}
$$

This is a very promising result. By using the specialisation fixpoint extension we have been able to successfully fuse this producer even though it was not quite in the state machine form.

Sadly, in general, the situation is more complex yet. Our original zip_s uses the internal state type $(s_a, s_b, Maybe\ a)$. A more extensible style would be to use a type with one constructor per mode.

$$\begin{aligned}&\textbf{data}\ ZipState\ s_a\ s_b\ a = ModeA\ s_a\ s_b\\ &\qquad\qquad\qquad\qquad\ \mid\ ModeB\ s_a\ s_b\ a\end{aligned}$$

When using this stream state type, our fused stepper function would look very similar

$$\begin{aligned}next\ (ModeA\ s_a\ s_b) &= \textbf{case}\ s_a\ \textbf{of}\\ &\qquad\qquad Left\quad n\ \to Skip\ (ModeB\ (Right\ (n+1))\ s_b\ n)\\ &\qquad\qquad Right\ n\ \to Skip\ (ModeB\ (Left\quad (n-1))\ s_b\ n)\\ next\ (ModeB\ s'_a\ s_b\ n) &= \textbf{case}\ s_b\ \textbf{of}\\ &\qquad\qquad Left\quad m\ \to Yield\ (n,m)\ (ModeA\ s'_a\ (Right\ (m+1)))\\ &\qquad\qquad Right\ m\ \to Yield\ (n,m)\ (ModeA\ s'_a\ (Left\quad (m-1)))\end{aligned}$$

While this may appear to be merely a cosmetic change, the argument usage information changes sufficiently to cause problems. The argument usage is now

$$\left[\begin{array}{l} ModeA \mapsto \left[\begin{array}{lll} Left & \mapsto & [\diamond] \\ Right & \mapsto & [\diamond] \end{array}\right], \diamond \\ \\ ModeB \mapsto \left[\diamond, \begin{array}{lll} Left & \mapsto & [\diamond] \\ Right & \mapsto & [\diamond] \end{array}\right], \diamond \end{array}\right]$$

Because the modes now have separate constructors, the fact that s_a and s_b are only scrutinised in one mode is now recorded precisely in the usage information. Previously the information from the two clauses was merged because it was the same field in each mode. The call patterns are as follows.

$$\begin{aligned}&n', s_b, n \triangleright (ModeB\ (Right\ n')\ s_b\ n)\\ &n', s_b, n \triangleright (ModeB\ (Left\quad n')\ s_b\ n)\\ &s'_a, m' \quad \triangleright (ModeA\ s'_a\ (Right\ m')\ Nothing)\\ &s'_a, m' \quad \triangleright (ModeA\ s'_a\ (Left\quad m')\ Nothing)\end{aligned}$$

Note that the parts of the call pattern that have nested substructure have corresponding usage information indicating that the structure is not scrutinised. The effect is that we do not get any specialisations at all.

This is a serious problem. It will affect many non-trivial stream transformers, particularly transformers that have multiple input streams or that make liberal use of extra modes. It is important for the expressiveness of stream functions that extra modes can be introduced freely.

One possible solution might be to allow the programmer to annotate data types that should always be considered for call pattern specialisation. It would involve a simple modification to the step of the algorithm where argument usage information is collected: data constructors of an annotated data type would always be considered to be used. With this modification to the definition of argument usage, if the *ZipState* type has the specialisation annotation added then the call patterns we want would be retained and the specialisation will work as desired. More generally the approach would be for state shapes to only use new specially defined and suitably annotated data types – rather than using general purpose types such as *Maybe* where any specialisation annotation would affect other programs. This annotation solution may provide an adequate compromise between the desire to use reusable general purpose optimisations and also to allow reliable optimisation in this special case.

4.7 Accounting for allocations

This section contributes the final part of the argument of Section 4.3.3, that overall, the sequence of transformations is an optimisation.

As mentioned in Sections 4.1.1–4.1.2, we declare a sequence of transformations to be an optimisation if it decreases our cost measure. Our chosen cost measure is the number of heap allocations for data constructors that are incurred in the course of program evaluation. The approximation of considering only the number of data constructors and not their size is largely justified; in practice the costs of allocating, using and deallocating constructors is dominated by the constant per-constructor overheads rather than the per-field factors.

It will be helpful to customise our cost measure to the context of functions on lists where the most natural measure is the number of data constructors allocated per list element. The allocations per element is independent of the list length and is often relatively easy to read-off from the code.

We ignore allocations relating to the end of the sequence. We will also ignore the allocation of the *Stream* constructor itself: it is only a single allocation for a whole sequence. It is in any case eliminated during stream transformer/producer fusion or stream consumer/producer fusion.

The choice of cost measure serves to highlight the result of this section: that by this cost measure, the transformations decrease the cost by exactly one. That is, one fewer data constructors are allocated per list element. This is as much as can be expected as lists only use one data constructor per list element.

Note that a similar cost measure that may be useful in some circumstances is to consider the sum of allocations involved in evaluating the first n sequence elements and define one measure to be less than another if for all n the sum is less.

4.7.1 Relating allocations in the ordinary and fusible functions

The change in allocations involved in the application of the *stream*/*unstream* fusion rule is simple. Tracing the change in allocations through the two later phases is also relatively straightforward. So while it is possible to show that the transformations described lead to a reduction in allocations, this is not on its own a useful result. Taking specially crafted inefficient list functions and making them less inefficient is not itself useful. Recall that our starting point for the transformation process is good list producers and good list consumers which are list functions that internally are implemented in terms of stream functions. We know that, prior to any transformations, these good producers and consumers perform more allocations than their counterpart ordinary list functions. To make a useful optimisation claim we must take ordinary list functions as our baseline.

The difficulty then is that we need some allocation constraint that relates the fusible list functions to ordinary their counterparts. Typically the fusible versions have a strong resemblance to the ordinary list versions but in principle – provided that they are semantically equivalent – they can be arbitrarily different. We need to ensure however that they are not arbitrarily worse; stream fusion cannot save us if we write poor fusible definitions. We know that initially the fusible versions are worse in terms of allocations, but we must be able to put some bound on how much worse. Indeed it must be a tight bound because we only expect to save a single allocation.

The intuition is that the only difference in allocations between the ordinary and fusible definitions is due the representation of the sequences. That is, if we take an ordinary and an equivalent fusible list function and we ignore the $(:)$, *Skip*, *Yield* and the constructors used for the static state shapes, then the remaining allocations should be identical.

Guided by this intuition, our aim is to find constraints relating ordinary and fusible definitions which, if satisfied, ensures that stream fusion is an optimisation. Since this is a rather backwards approach – starting with the hope that stream fusion is an optimisation and working back to constraints that make it so – it is essential that we check the resulting constraints against reality. In Sec-

tion 4.7.5 we will use a number of common fusible list producers and consumers to check that the allocation constraints are satisfiable.

Our strategy to find a suitable relation involves breaking down and classifying the various allocations of the ordinary and fusible list functions. This breakdown must be sufficiently precise to isolate the classes of allocations that are eliminated by the stream fusion transformations.

Our method for classifying the allocations involves analysing the allocations performed during evaluation of a term to weak head normal form (WHNF). While we do not formally define an evaluation machine, we do assume lazy evaluation, i.e. non-strict evaluation where shared subterms are evaluated at most once. Consider the evaluation of a term e to a head constructor C with further terms $e_1, e_2, \ldots e_n$ in the fields of the constructor

$$e \rightsquigarrow (C\ e_1\ e_2 \ldots e_n)$$

As a specific instance, think of $xs \rightsquigarrow (x : xs')$, that is evaluating a list xs to weak head normal form with a head term x and a tail term xs'.

Performing the evaluation $e \rightsquigarrow (C\ e_1\ e_2 \ldots e_n)$ has the effect of allocating some number of data constructors. We can classify these allocations by whether or not they are retained in the result $C\ e_1\ e_2 \ldots e_n$, and if so, where in the result they are retained. We can obviously identify the head constructor C itself, which, depending on the original term e, may have been allocated during the evaluation step. We will classify the other retained allocations by the field in the constructor that retains them. Where an allocation is retained by multiple fields we will account it to the leftmost field. This is an arbitrary choice but as we shall see the reverse choice will not affect the analysis in any essential way.

The setup for the argument is as follows: we have an arbitrary ordinary list producer/consumer pair such that the application of the consumer to the producer is well typed.

$$consumer_{ordinary}\ (producer_{ordinary}\ \ldots)$$

We have another producer/consumer pair that are equal to the ordinary producer and consumer but that satisfy the good list producer and good list consumer conditions.

$$consumer_{fusible}\ (producer_{fusible}\ \ldots)$$

We will then be interested in the number of allocations involved in the following cases:

1. the ordinary list consumer/producer application;
2. the equivalent fusible consumer/producer application prior to fusion;
3. and the fusible consumer/producer application after fusion.

4.7.2 Allocations in ordinary list producers and consumers

Take the list xs that is generated by our list producer function $producer_{ordinary}$. We are accounting for allocations on a per sequence-element basis so we are interested in the allocations involved in evaluating this list to weak head normal form which gives us a single element

$$xs \rightsquigarrow (x : xs')$$

By carefully analysing the list producer function we may write down an expression for the number of allocations in this evaluation step; that is, the number of allocations per sequence element. Consider, for example, the following contrived list producer

$$\begin{array}{l} evensFrom\ n \mid even\ n \quad = n : evensFrom\ (n+1) \\ \qquad\qquad\quad \mid otherwise = \quad evensFrom\ (n+1) \end{array}$$

$$\begin{array}{l} \quad evensFrom\ 0 \\ = \\ \quad 0 : 2 : 4 : \ldots \end{array}$$

In this example the number of allocations per sequence element is four: one for the $(:)$ constructor, one for the *Int* element itself, one for the intermediate *Int* value that is constructed and discarded and one for the *Int* value constructed for the recursive call in the list tail. It is clear that in the general case for a list producer, the term describing the number of allocations per element may be rather complicated. For the most part we will not be concerned with the detailed form of the term.

Focusing again on the allocations involved in the evaluation to weak head normal form $xs \rightsquigarrow (x : xs')$, we break down the allocations involved into different classes following the method outlined above. For the classes of known size we give the appropriate term and for the others, that depend on the details of the list producer, we identify them with a named term.

We have four classes of allocations:

- ($whnf_{cons}$) those involved in the evaluation $\rightsquigarrow$ but not retained;
- (1) the single (:) constructor;
- ($head_{cons}$) those retained by the head x;
- ($tail_{cons}$) and those retained by the tail xs'.

We will require that the (:) constructor is a fresh allocation. This turns out to be an important constraint. We will consider the effect of violating it in Section 4.7.11.

For the ordinary list consumer we do not need to do any breakdown. We simply have a term $consume_{cons}$ for the number of allocations used per sequence element. Thus for the ordinary list consumer/producer application, the total number of allocations per sequence element is

$$total_{ordinary} = whnf_{cons} + 1 + head_{cons} + tail_{cons} + consume_{cons}$$

4.7.3 Allocations in fusible list producers and consumers

For the fusible producer and consumer we can split the allocations in a similar, though more detailed, way.

The good producer conditions ensure that the producer can be unfolded to give *unstream* applied to a stream producer, which itself can be unfolded to give us a stream term $Stream\ next\ s_0$. We are then interested in the allocations involved in the evaluation to weak head normal form

$$unstream\ (Stream\ next\ s_0) \rightsquigarrow (x : xs')$$

This degree of evaluation places demand on the result of *unstream* which in turn demands steps from the stream.

$$unstream\ (Stream\ next\ s_0) \rightsquigarrow unfold\ next\ s_0$$

The general case for the stream is a finite number of *Skip* steps followed by a *Yield*.

$$\begin{array}{ll} next\ s_0 \rightsquigarrow Skip & s_1 \\ next\ s_1 \rightsquigarrow Skip & s_2 \\ \ldots & \\ next\ s_n \rightsquigarrow Yield\ x\ s_{n+1} & \end{array}$$

This gives us a corresponding sequence of evaluation steps of the overall list term

$$
\begin{aligned}
&\mathit{unfold\ next\ s_0} \rightsquigarrow \quad \mathit{unfold\ next\ s_1}\\
&\mathit{unfold\ next\ s_1} \rightsquigarrow \quad \mathit{unfold\ next\ s_2}\\
&\ldots\\
&\mathit{unfold\ next\ s_n} \rightsquigarrow \mathit{x} : \mathit{unfold\ next\ s_{n+1}}
\end{aligned}
$$

The number of skips is of course determined by the definition of the stream producer and need not be a constant.

Within the various stream states s_x we need to distinguish the constructors used to represent the state shapes from those used to represent the other variable parts of the state. This is of course because we expect to eliminate the allocations associated with the state shapes but not the other variable parts.

We can now classify the allocations involved. We have:

- ($\mathit{whnf}_{\mathit{skip}}$) the allocations involved in but not retained by the evaluation of the skip steps $\mathit{next}\ s_x \rightsquigarrow \mathit{Skip}\ s_{x+1}$;
- (skips) the number of *Skip* constructors;
- ($\mathit{shapes}_{\mathit{skip}}$) the constructors for the state shapes in the skip states s_x
- ($\mathit{vars}_{\mathit{skip}}$) the constructors for the variables in the shapes in the skip states s_x
- ($\mathit{whnf}_{\mathit{yield}}$) the allocations involved in but not retained by the evaluation of the yield step $\mathit{next}\ s_n \rightsquigarrow \mathit{Yield}\ x\ s_{n+1}$;
- (1) the single *Yield* constructor;
- ($\mathit{elem}_{\mathit{yield}}$) the allocations retained by the element x;
- ($\mathit{shapes}_{\mathit{yield}}$) the constructors for the state shapes in the yield state s_n;
- ($\mathit{vars}_{\mathit{yield}}$) the constructors for the variables in the shape in the yield state s_n;
- (1) the single (:) constructor;

Note that the first four classes cover all the skip steps – the sum of the allocations for the individual steps.

For the fusible consumer, the good consumer conditions require that it consumes its input list using the function *stream* and a stream consumer. The only breakdown of allocations we need is to identify the single *Yield* constructor allocated by *stream* for each sequence element. The remaining allocations come from the stream consumer and we label the term $consume_{yield}$.

Thus for the fusible list consumer/producer application, prior to fusion, the total number of allocations per sequence element is

$$\begin{aligned} total_{prefuse} = {} & whnf_{skip} + skips + shapes_{skip} + vars_{skip} \\ & + whnf_{yield} + 1 + elem_{yield} + shapes_{yield} + vars_{yield} \\ & + 1 \\ & + 1 + consume_{yield} \end{aligned}$$

4.7.4 Allocation constraints for fusible list functions

Our choice of classification of the allocations identifies the classes that we expect to be able to eliminate as part of the stream fusion transformations. In particular:

- $(1 + 1)$ by applying the *stream*/*unstream* fusion rule we expect to eliminate the $(:)$ and *Yield* allocations originating from the *stream* and *unstream* functions;
- $(skips + 1)$ by applying the case-of-case transformation we expect to eliminate the *Skip* and *Yield* constructors of the stream producer;
- $(shapes_{skip} + shapes_{yield})$ by applying call pattern specialisation we expect to eliminate the state shapes in the *Skip* and *Yield* steps.

In Sections 4.7.6–4.7.10 we will check that the various transformations really do eliminate these classes of allocations. For the moment if we assume that we can eliminate them, then after stream fusion the total number of allocations per sequence element is

$$\begin{aligned} total_{postfuse} = {} & whnf_{skip} + vars_{skip} \\ & + whnf_{yield} + elem_{yield} + vars_{yield} \\ & + consume_{yield} \end{aligned}$$

Recall that the allocations for the ordinary list producer/consumer application is

$$\begin{aligned} total_{ordinary} = {} & whnf_{cons} + 1 + head_{cons} + tail_{cons} \\ & + consume_{cons} \end{aligned}$$

Our target is for the fused application to use exactly one fewer allocation than the ordinary list producer/consumer application, i.e. $total_{postfuse} = total_{ordinary} - 1$, which is equivalent to

$$\begin{aligned} & whnf_{skip} + vars_{skip} = whnf_{cons} + head_{cons} + tail_{cons} \\ & + whnf_{yield} + elem_{yield} + vars_{yield} \\ & + consume_{yield} + consume_{cons} \end{aligned}$$

This gives us the relationship between the ordinary and fusible list functions that we have been looking for. We can now interpret what it means and what constraints it places on the definitions of fusible list producers and consumers. We shall see that it precisely expresses our initial intuition that the only difference in allocations between the ordinary and fusible definitions should be due the representation of the sequences.

As stated, this relationship is a tight bound. We could state it more generally by taking $total_{postfuse} < total_{ordinary}$. Informally this would mean we allow for the fusible versions to be simply better than the ordinary equivalent, e.g. by using a better algorithm. It is not especially useful to consider this greater generality however because the situation does not occur in practice; any improved stream function can trivially be translated into a list version.

Since the allocations of the producer and consumer are independent we can use a simpler special case of the relationship

$$\begin{aligned} & whnf_{skip} + vars_{skip} = whnf_{cons} + head_{cons} + tail_{cons} \\ & + whnf_{yield} + elem_{yield} + vars_{yield} \end{aligned}$$

$$consume_{yield} = consume_{cons}$$

Allocation constraint for list consumers

The constraint for consumers is simply

$$consume_{yield} = consume_{cons}$$

The interpretation is also simple.

Recall that the total number of allocations per sequence element for the ordinary consumer is $consume_{cons}$. For the fusible consumer the total is $consume_{yield} + 1$. The $+1$ is for the single *Yield* constructor allocated by *stream*, and this is the only allocation due to the *stream* function – all the remaining $consume_{yield}$ allocations are due to the stream consumer itself.

So this relation simply says that the stream consumer and the ordinary list consumer must use the same number of allocations per sequence element. The allocation due to the *stream* function can be ignored as it will be eliminated.

Allocation constraint for list producers

The constraint for producers is

$$\begin{aligned} &whnf_{skip} &&+ vars_{skip} \\ &+ whnf_{yield} + elem_{yield} + vars_{yield} \end{aligned} \quad = \quad whnf_{cons} + head_{cons} + tail_{cons}$$

For ordinary list producers the total allocations per sequence element is

$$whnf_{cons} + head_{cons} + tail_{cons} + 1$$

The $+1$ is for the list $(:)$ constructor itself.

For the fusible list producers, the total allocations per sequence element is

$$\begin{aligned} &whnf_{skip} + skips &&+ shapes_{skip} + vars_{skip} \\ &+ whnf_{yield} + 1 + elem_{yield} + shapes_{yield} + vars_{yield} + 1 \end{aligned}$$

If we look at the total allocations, less those mentioned in the constraint above then we have

$$\begin{aligned} &skips + shapes_{skip} \\ &+ 1 \quad + shapes_{yield} + 1 \end{aligned}$$

The final $+1$ is for the single (:) constructor allocated by *unstream*. The $skips + shapes_{skip}$ allocations are for the *Skip* constructors and the constructors used to represent the *Skip* state shapes. Similarly, the $1 + shapes_{yield}$ allocations are for the *Yield* constructor and the constructors used to represent the *Yield* state shape.

So the interpretation of this relation is that the stream producer and the ordinary list producer must use the same number of allocations per sequence element, except that we discount the $(:)$ for the ordinary list producer and for the stream consumer we discount the *Skip*, *Yield* and the constructors used for the static state shapes.

4.7.5 Feasibility of the allocation constraints

Having seen that the allocation constraints appear reasonable, we will now look at a few examples to see if they satisfy the allocation constraint. The purpose is primarily to check that the constraints we derived are not always impossible and secondarily to demonstrate how the check can be done. For a library of fusible functions, each function must be checked to make sure it satisfies the allocation constraint.

A simple example

We will start with the same contrived list producer as before, along with an equivalent stream version

$$
\begin{array}{l}
\mathit{evensFrom}\ n \mid \mathit{even}\ n \quad\ = n : \mathit{evensFrom}\ (n+1) \\
\qquad\qquad\quad \mid \mathit{otherwise} = \quad \mathit{evensFrom}\ (n+1)
\end{array}
$$

$$
\begin{array}{l}
\mathit{evensFrom}_s\ n = \mathit{Stream}\ \mathit{next}\ n \\
\quad \textbf{where} \\
\quad\ \mathit{next}\ n \mid \mathit{even}\ n \quad\ = \mathit{Yield}\ n\ (n+1) \\
\qquad\qquad\ \mid \mathit{otherwise} = \mathit{Skip} \quad\ (n+1)
\end{array}
$$

For the ordinary list version we have the following typical evaluation

$$
\begin{array}{l}
\quad \mathit{evensFrom}\ 1 \\
\rightsquigarrow \\
\quad \mathit{evensFrom}\ (1+1) \\
\rightsquigarrow \\
\quad 2 : \mathit{evensFrom}\ (2+1)
\end{array}
$$

Apart from the $(:)$, there are three allocations: the 1 in the $1+1$ intermediate term, the reduced value 2 and the 1 in the $2+1$ term. For simplicity we assume a naïve evaluator that does no strictness analysis or primitive redex analysis. An optimising compiler would be able to avoid allocating the 1 in the $1+1$ intermediate term. This issue does not affect our analysis as long as we use the same evaluator consistently.

We are interested in the sum $\mathit{whnf}_{\mathit{cons}} + \mathit{head}_{\mathit{cons}} + \mathit{tail}_{\mathit{cons}}$. As we noted previously, this sum is all of the allocations of the list producer apart from the $(:)$ itself. We do not need to assign the allocations to the classes $\mathit{whnf}_{\mathit{cons}}$, $\mathit{head}_{\mathit{cons}}$ and $\mathit{tail}_{\mathit{cons}}$ because the allocation constraint involves their sum. Nevertheless, for the first simple example it may be illuminating to do so. We will not repeat the exercise for the later examples.

In this example then, $head_{cons} = 1$ because it is a newly allocated *Int*, not one that was passed in. We also have a distinct *Int* allocated and retained in the tail, thus $tail_{cons} = 1$. Finally, there is the intermediate *Int*, distinct from the other two, that is not retained in the result, hence $whnf_{cons} = 1$. Note that the allocated 2 is shared between the head and tail terms. We arbitrarily assigned it to the head but it is clear that if we made the reverse decision then, though the breakdown is different, the sum would not be affected. It is a benefit of not performing the more detailed breakdown that we are not faced with such arbitrary choices.

For the fusible version we have a similar sequence of evaluation steps

$$\begin{aligned} &next\ 1 && \rightsquigarrow Skip && (1+1) \\ &next\ (1+1) && \rightsquigarrow Yield\ 2\ (2+1) \end{aligned}$$

We are interested in the sum $whnf_{skip} + vars_{skip} + whnf_{yield} + elem_{yield} + vars_{yield}$ which we noted previously is all of the stream producer allocations except for the *Skip*, *Yield* and any constructors used to represent the state shapes. This example uses no constructors for state shapes because it uses a single trivial state shape so for this example we are interested in all the allocations except for the *Skip* and *Yield*. Again it is not essential that we assign the allocations to the various classes, we do so only for this first example.

In the first evaluation step we allocate a 1 for the *Skip* result state. In the second evaluation step we reduce $1 + 1$ and allocate the result 2 which becomes the *Yield* element. We also allocate a 1 for the *Yield* result state. Thus we have $vars_{skip} = 1$, $elem_{yield} = 1$ and $vars_{yield} = 1$. We have $whnf_{skip} = 0$ and $whnf_{yield} = 0$ because there are no other allocations that are not retained by an intermediate *Skip* state or by the *Yield* result. The allocated 2 is shared between the element and new state in the *Yield*. Again, we arbitrarily assign it to the $elem_{yield}$ class and again it is clear that this choice does not affect the sum.

In totality then, the sum is three allocations for the stream producer which is the same as for the ordinary list version and thus this pair of ordinary and fusible list producers satisfy the allocation constraint.

Apart from the minor issue of how shared allocations are accounted, it will typically be the case that the classes of allocations match up as follows

$$\begin{aligned} whnf_{cons} &= whnf_{skip} + vars_{skip} + whnf_{yield} \\ head_{cons} &= elem_{yield} \\ tail_{cons} &= vars_{yield} \end{aligned}$$

The *unstream* function maps streams to lists, but it also gives rise to a mapping from the stream allocations classes into the list allocation classes. If the stream

and list version are very similar then this mapping into the list allocation classes gives us the above matches.

Allocations for *filter* and $filter_s$

For a standard example of a list producer we look at *filter*. It is a transformer and also has non-trivial allocation behaviour.

$$
\begin{array}{l}
filter :: (a \to Bool) \to [a] \to [a] \\
filter\ p\ xs = \textbf{case}\ xs\ \textbf{of} \\
\qquad\qquad [\,] \qquad\qquad\qquad\quad \to [\,] \\
\qquad\qquad (x : xs') \mid p\ x \qquad\quad \to x : filter\ p\ xs \\
\qquad\qquad\qquad\quad \mid otherwise \to \quad filter\ p\ xs
\end{array}
$$

We must decide how to account for allocations by the transformer's input sequence. The obvious choice is to assign them to the list producer function that generated the input list. In this example the top level case scrutinises xs and, in the branch we are interested in, it performs the evaluation $xs \rightsquigarrow (x : xs')$. We assign the allocations involved in this evaluation to the producer of the input list. Overall, to evaluate $filter\ p\ xs \rightsquigarrow (x : xs')$ we must evaluate p on some finite sequence of elements where $p\ x_i \rightsquigarrow False$ followed by evaluating $p\ x_n \rightsquigarrow True$.

$$
\begin{array}{l}
filter_s :: (a \to Bool) \to Stream\ a \to Stream\ a \\
filter_s\ p\ (Stream\ next_0\ s_0) = Stream\ next\ s_0 \\
\quad \textbf{where} \\
\qquad next\ s = \textbf{case}\ next_0\ s\ \textbf{of} \\
\qquad\qquad\qquad Done \qquad\qquad\quad \to Done \\
\qquad\qquad\qquad Skip \quad s' \qquad\quad \to Skip \quad s' \\
\qquad\qquad\qquad Yield\ x\ s' \mid p\ x \qquad \to Yield\ x\ s' \\
\qquad\qquad\qquad\qquad\quad \mid otherwise \to Skip \quad s'
\end{array}
$$

For the stream version we can ignore the allocations involved in evaluating $next_0\ s$ because they are assigned to the producer of the input stream. There are no allocations in the *Skip* case. So we have just the allocations involved in evaluating $p\ x_i \rightsquigarrow False$ some number of times, followed by evaluating $p\ x_n \rightsquigarrow True$. Since the sequence elements are the same between the list and stream versions then the evaluations and allocations performed are identical.

Allocations for zip **and** zip_s

In many ways zip is even simpler than *filter*. The only allocation of interest is the $(,)$ in the (a, b) element result.

$$
\begin{aligned}
&zip :: [a] \to [b] \to [(a,b)] \\
&zip\ [\,]\ \ \ \ ys\ \ \ \ = [\,] \\
&zip\ xs\ \ \ \ [\,]\ \ \ \ = [\,] \\
&zip\ (x : xs)\ (y : ys) = (x, y) : zip\ xs\ ys
\end{aligned}
$$

$$
\begin{aligned}
&zip :: Stream\ a \to Stream\ b \to Stream\ (a, b) \\
&zip\ (Stream\ next_a\ s_a)\ (Stream\ next_b\ s_b) = Stream\ next\ (s_a, s_b, Nothing) \\
&\quad \textbf{where} \\
&\qquad next\ (s_a, s_b, Nothing) = \textbf{case}\ next_a\ s_a\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\qquad Done\ \ \ \ \ \ \to Done \\
&\qquad\qquad\qquad\qquad\qquad\qquad Skip\ \ \ s'_a \to Skip\ (s'_a, s_b, Nothing) \\
&\qquad\qquad\qquad\qquad\qquad\qquad Yield\ a\ s'_a \to Skip\ (s'_a, s_b, Just\ a) \\
&\qquad next\ (s'_a, s_b, Just\ a)\ \ \ \ = \textbf{case}\ next_b\ s_b\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\qquad Done\ \ \ \ \ \ \to Done \\
&\qquad\qquad\qquad\qquad\qquad\qquad Skip\ \ \ s'_b \to Skip\ \ \ \ \ \ \ (s'_a, s'_b, Just\ a) \\
&\qquad\qquad\qquad\qquad\qquad\qquad Yield\ b\ s'_b \to Yield\ (a, b)\ (s'_a, s'_b, Nothing)
\end{aligned}
$$

Again, the input streams may skip so we have a finite sequence of *Skip* evaluations followed by a *Yield* step which allocates a $(,)$ constructor. Since *next* performs no allocations in the *Skip* two cases then there is only the single allocation.

Allocations for $foldr$ **and** $foldr_s$

Recall that for a consumer we need the allocations per sequence element consumed to be identical between the list and stream version.

$$
\begin{aligned}
&foldr\ f\ z\ xs = go\ f\ z\ xs \\
&\quad \textbf{where} \\
&\qquad go\ f\ z\ xs = \textbf{case}\ xs\ \textbf{of} \\
&\qquad\qquad\qquad\qquad [\,]\ \ \ \ \ \ \to z \\
&\qquad\qquad\qquad\qquad (x : xs') \to f\ x\ (go\ f\ z\ xs')
\end{aligned}
$$

Again, the allocations involved in evaluating the input list to WHNF is assigned to the producer so we need not consider it here. We need only be concerned with the case branch $(x : xs') \to f\ x\ (go\ f\ z\ xs')$. There are no immediate constructor

allocations here, all the parameters passed to f and go are preexisting terms. All the allocations here are in the evaluation of $f\ x\ (go\ f\ z\ xs')$ to WHNF. Clearly if f demands $go\ f\ z\ xs'$ then allocations involved in its evaluation should be assigned to the next iteration, since we are looking at allocations per sequence element.

The stream version is almost identical.

$$
\begin{array}{l}
foldr_s\ f\ z\ (Stream\ next\ s) = go\ f\ z\ s \\
\quad \textbf{where} \\
\quad\quad go\ f\ z\ s = \textbf{case}\ next\ s\ \textbf{of} \\
\qquad\qquad\qquad\qquad Done \quad\quad \to z \\
\qquad\qquad\qquad\qquad Skip \quad s' \to go\ f\ z\ s' \\
\qquad\qquad\qquad\qquad Yield\ x\ s' \to f\ x\ (go\ f\ z\ s')
\end{array}
$$

As required, the *Skip* case performs no allocation, so a finite sequence of *Skip*s followed by a *Yield* only involves the allocations involved in evaluating $f\ x\ (go\ f\ z\ xs')$ to WHNF. Again, we can assign the cost of $go\ f\ z\ xs'$ to the next iteration. Thus the list and stream versions have the same allocation cost per sequence element.

4.7.6 Overall allocation change due to stream fusion

Recall that the allocation argument starts with a single good consumer applied to a single good producer.

$$consumer_{fusible}\ (producer_{fusible}\ \ldots)$$

In Sections 4.7.3 and 4.7.4 we identified various classes of allocations in this good consumer/producer application term that must be eliminated for stream fusion to be an overall optimisation:

- $(1 + 1)$ the $(:)$ and *Yield* constructors in the *stream* and *unstream* functions;
- $(skips + 1)$ the *Skip* and *Yield* constructors in the stream producer;
- $(shapes_{skip} + shapes_{yield})$ the constructors used to represent the state shapes in the *Skip* and *Yield* steps in the stream producer.

There is a relationship between these allocation classes and the transformations that eliminate them. For the three major transformations in stream fusion we expect that:

- the initial phase, including the use of the *stream*/*unstream* fusion rule (see Section 4.2), will eliminate:
  - $(1 + 1)$ the $(:)$ and *Yield* constructors in the *stream* and *unstream* functions;
- stream transformer/producer fusion (see Section 4.5) will eliminate:
  - $(skips + 1)$ the *Skip* and *Yield* constructors in the stream producer;
  - but that it may increase allocations in the $shapes_{skip}$ and $shapes_{yield}$ classes;
- stream consumer/producer fusion (see Section 4.6) will eliminate:
  - $(skips + 1)$ the *Skip* and *Yield* constructors in the stream producer;
  - $(shapes_{skip} + shapes_{yield})$ the constructors used to represent the state shapes in the *Skip* and *Yield* steps in the stream producer.

For the first transformation, the relationship is simple: given a good consumer applied to a good producer, there is a single opportunity to apply the *stream*/*unstream* fusion rule which will eliminate the $1 + 1$ allocations.

For the latter two transformations we need to relate our single good producer/consumer application to uses of the two fusion steps so that we can account for all the other $skips + 1 + shapes_{skip} + shapes_{yield}$ allocations that we need to eliminate.

Recall from Section 4.2.1 and 4.2.2 that a good producer is defined using a stream producer and that a good consumer is defined using a stream consumer, however either or both of the stream producer and stream consumer may in fact be a stream transformer. Recall from Section 4.3.1 that we distinguish stream transformers as a special case, and in particular that we have to treat as a single unit applicative combinations of stream consumer, stream transformers and stream producers. Optimising these larger units may involve multiple applications of stream transformer/producer fusion followed by a single application of stream consumer/producer fusion.

In Sections 4.2.3 and 4.3.1 we discussed how the construction of good producers and consumers ensures that there are no 'naked' streams as inputs or as results,

and hence that these applicative terms of consumer, transformers and producers always take the form of trees with a single consumer at the root and producers at the leaves. Recall from Section 4.3.2 that we proceed bottom-up, repeatedly applying stream transformer/producer fusion at the leaves until we are left with a consumer applied to a single stream producer, at which point we apply stream consumer/producer fusion.

Each good producer/consumer application pair gives rise to a single edge in such a tree. We apply stream transformer/producer fusion as many times as there are interior nodes in the tree. Since stream transformer/producer fusion fuses the transformer with all of its input producers then this covers all non-root edges in the tree. We apply stream consumer/producer fusion only once, which covers the root edge. Hence all edges of the tree are covered by one of the two fusion steps. For all edges we must eliminate both $skips + 1$ and $shapes_{skip} + shapes_{yield}$. Since both stream transformer/producer fusion and stream consumer/producer fusion eliminate the $skips + 1$ allocations for all their input producers then we have covered these allocations for all edges. Stream transformer/producer fusion preserves and may even increase the $shapes_{skip} + shapes_{yield}$ allocations both of the transformer and of the producer so that the resulting stream producer contains at least their sum. Thus the $shapes_{skip} + shapes_{yield}$ allocations for each non-root node are preserved or increased; they are accumulated up the tree and finally they are eliminated by the single use of stream consumer/producer fusion.

Overall then, the combined effect of these transformation steps is to eliminate the classes of allocations required for stream fusion to be an optimisation. It remains to show that the major transformation steps do eliminate the allocations that we expect and that they do not increase other allocations.

4.7.7 Lack of fusion is not a pessimisation

Our argument in this chapter is primarily concerned with allocation change in the case that we can fuse producers and consumers. It is also important however to consider what happens when fusion does not occur. While we guarantee fusion to occur when a good consumer is directly applied to a good producer, there are other cases to consider:

- a good producer or a good consumer that is used in isolation;
- a function that is a good consumer in multiple arguments, but a use site where it is only applied to good producers for some of those arguments;

- a function that is both a good consumer and a good producer but a use site where it is only used with a good producer or consumer on one side.

One can characterise stream fusion as a process involving taking one step back before taking two steps forward, so it is reasonable to worry that we may end up making things worse in cases where we are unable to fuse a function in one or more of its inputs or outputs. Fortunately there is no such problem: in all such cases stream fusion will be exactly neutral in terms of our allocation cost measure.

There are two possibilities: the rest of this chapter is concerned with the case where a good consumer is directly applied to a good producer; the alternative possibility is a good producer or a good consumer that does not appear in a good producer/consumer application term. Since we do not have a good producer/consumer application then we cannot apply the *stream*/*unstream* fusion rule. The consequence is that we are left with a good producer or a good consumer. A good consumer however unfolds to give a stream consumer applied to a *stream* term, while a good producer unfolds to give *unstream* applied to a stream producer. Since *stream* is a stream producer and *unstream* is a stream consumer, then in both cases we have a stream producer applied to a stream consumer and we can apply stream consumer/producer fusion. In both cases, applying stream consumer/producer fusion leaves us with exactly as many allocations as we would have had with the ordinary list version[6].

Let us briefly do the accounting, first for the producer case, then for the consumer case. Recall from Sections 4.7.2 and 4.7.3 that the allocation cost of an ordinary producer is

$$whnf_{cons} + 1 + head_{cons} + tail_{cons}$$

while the allocation cost of its fusible equivalent is

$$\begin{array}{l} \quad whnf_{skip} \; + skips \qquad\quad + shapes_{skip} \; + vars_{skip} \\ + whnf_{yield} + 1 + elem_{yield} + shapes_{yield} + vars_{yield} \\ + 1 \end{array}$$

We may assume that the ordinary and fusible versions satisfy the allocation constraint.

$$\begin{array}{l} \quad whnf_{skip} \qquad\quad + vars_{skip} \;\; = \;\; whnf_{cons} + head_{cons} + tail_{cons} \\ + whnf_{yield} + elem_{yield} + vars_{yield} \end{array}$$

[6]There is at most some code duplication – something not captured by our cost measure.

We are interested in the difference between the allocations of the ordinary producer and the fusible producer. If we take the difference and simplify it by using the allocation constraint as a substitution, followed by cancelling out terms, then we are left with

$$skips + 1 + shapes_{yield} + shapes_{skip}$$

That is, prior to any fusion, the fusible producer uses this many more allocations than the ordinary producer. Recall from Section 4.7.6 that this is exactly the allocations that we expect to be eliminated by stream consumer/producer fusion.

For consumers the accounting is simpler. The original consumer allocates $consume_{cons}$ while the fusible equivalent allocates $1 + consume_{yield}$. Given the allocation constraint $consume_{yield} = consume_{cons}$ then the difference is just 1. We next consider the allocations that are eliminated when we apply stream consumer/producer fusion on the application of the stream consumer to the stream producer function *stream*. For an arbitrary stream producer, we would eliminate $skips + 1 + shapes_{yield} + shapes_{skip}$ but in the case of *stream* specifically, the *stream* stepper function does not use any state shapes and it does not use *Skip* so we eliminate only the the *Yield*, exactly 1 allocation.

Hence, for both cases – unfused good producers and unfused good consumers – stream consumer/producer fusion eliminates exactly the number of allocations such that the allocation change is neutral compared to the ordinary list producers or consumers.

4.7.8 Allocation change in the *stream*/*unstream* fusion phase

The only transformations in this phase that affect the number of data constructor allocations is the application of the *stream*/*unstream* fusion rule itself. The unfolding, β-reduction and **let**-floating transformations do not affect the number of allocations.

It should be noted that, in this context, by unfolding we mean just replacing a name by the expression that it stands for. Unfolding may of course give rise to opportunities for reducing cases of known constructors – which does reduce allocations – but case reduction must be considered distinctly from unfolding.

Given a good consumer applied to a single good producer, there is a single opportunity to apply the *stream*/*unstream* fusion rule. There is one allocation per sequence element in each of the *stream* and *unstream* functions: the $(:)$ and *Yield*

constructors respectively. Applying the *stream*/*unstream* fusion rule eliminates both functions and thereby the corresponding two allocations.

While it is clear that the fusion rule eliminates two allocations we must also check that it does not increase allocations elsewhere. Consider a consumer/producer term before applying the fusion rule.

$$consumer_s\ (stream\ (unstream\ (producer_s \ldots)))$$

The *unstream* function filters out the *Skip* steps so that the consumer never sees any.

$$consumer_s\ (producer_s \ldots)$$

After applying the *stream*/*unstream* fusion rule the stream consumer is directly applied to the stream producer and will receive any *Skip* steps the producer emits.

To guarantee that the *stream*/*unstream* fusion rule reduces allocations it is therefore essential that the consumer does not perform any allocations while processing a *Skip* step. It is for this reason that we have imposed such a condition on stream consumers and transformers: producer condition 5 (Section 4.5.4) and consumer condition 7 (Section 4.6.3).

4.7.9 Allocation change in stream transformer/producer fusion

As described in Section 4.5, stream transformer/producer fusion is the transformation whereby the application of a single stream transformer to one or more stream producers is fused to give a single stream producer. The remaining obligation for this transformation is to show that it eliminates the allocation of the *Skip* and *Yield* constructors in each of the input stream producers and that while the transformation may increase the number of allocations, the increase is restricted to the $shapes_{skip}$ and $shapes_{yield}$ allocation classes.

Stream transformer/producer fusion consists of a number of simpler transformations which are described in detail in Section 4.5.5. In summary, the steps are:

- unfolding the definitions of the transformer and producers;
- **let**-floating followed by case of a known constructor reduction to eliminate the *Stream* constructors

- unfolding the definition of the producers' stepper functions in scrutinee positions in the transformer's stepper function;
- the general case-of-case transformation;
- the specialisation transformation used to preserve the strong state shape matching property.

Unfolding and **let**-floating do not change the number of constructor allocations. The *Stream* constructor for each of the stream producers is eliminated, though as mentioned in the introduction to Section 4.7, we have ignored this allocation since it is per sequence and not per sequence element.

The important step is the use of the general case-of-case transformation to eliminate the *Done*, *Skip* and *Yield* constructors produced as the result of each stream producer's stepper function. The general case-of-case transformation is described in Section 4.5.7, and the specific use of it is described in Section 4.5.5.

The stream producers' stepper functions only occur in case scrutinee positions in the transformer's stepper function. The general case-of-case transformation is used once for each such occurrence. In each use it eliminates the *Step* constructors in the leaf positions of the stepper function. Thus, for each stream producer that has its stepper function used by the stream transformer, the *Step* constructors used in the stepper function's result are eliminated. The null case, where an input stream producer's stepper function is not used, contributes zero allocations so can be ignored.

To see that it does not change any other classes of allocations, consider that the general case-of-case transformation consists of: 1) pushing one case expression through an inner one, which does not change allocations; and 2) reducing a case of a known constructor, which eliminates one allocation but is used only for the leaf positions. The leaf positions contain only *Step* constructors (producer condition 6).

The specialisation transformation is described in Section 4.5.3. The argument that it increases allocations only in the classes $shapes_{skip}$ and $shapes_{yield}$ is given in Section 4.5.5.

4.7.10 Allocation change in stream producer/consumer fusion

As described in Section 4.6, stream consumer/producer fusion is the transformation whereby the application of a single stream consumer to a single stream producer is fused to give a collection of mutually recursive functions. Our

obligation for this transformation is to show that it eliminates certain allocations in the stream producer and that it otherwise leaves the number of allocations unchanged. We expect it to eliminate the *Skip* and *Yield* constructors and the constructors used to represent the state shapes in the *Skip* and *Yield* steps.

Section 4.6.4 describes the various transformations that make up consumer/producer fusion. In summary the steps are:

- unfolding the definition of the consumer and producer;
- **let**-floating followed by case of a known constructor reduction to eliminate the *Stream* constructor
- unfolding the definition of the producer's stepper function in scrutinee positions in the consumer's worker function;
- the general case-of-case transformation;
- the call pattern specialisation transformation.

The important steps are the final two.

The unfolding and **let**-floating do not change the number of allocations. The *Stream* constructor in the producer is eliminated by case reduction following unfolding the definition of the producer.

The general case-of-case transformation is used – in almost exactly the same way as in stream transformer/producer fusion – to eliminate the *Step* constructors in the producer. The stepper function of the stream producer occurs only in case scrutinee positions in the consumer's worker function. The general case-of-case transformation is used once for each such occurrence which eliminates the *Step* constructors in the leaf positions of the stepper function. Thus if the stream producer's stepper function is used at all by the stream consumer then the *Step* constructors used in the stepper function's result are eliminated. In the null case the stream producer's stepper function is not used and hence contributes zero allocations. The general case-of-case only eliminates the constructors in the leaf positions of the stepper function and hence the transformation does not affect any of the other allocation classes.

The final step is the call pattern specialisation transformation. The argument that it eliminates the constructors used to represent the producer's state shapes is given in Section 4.6.5. To see that it does not affect other allocation classes consider: 1) the only step that adds constructors is the step where the specialised worker functions are defined; for each state shape they are defined using the

body of the original worker function but with the state parameter substituted for the state shape pattern; 2) the state parameter occurs only in recursive calls to the worker function or case expression trees that match all state shapes; 3) in both cases the constructors are eliminated by replacing a call to a specialised worker or by case of known constructor reduction respectively.

4.7.11 Sharing

We will look at two examples where violating one of the good producer or consumer conditions leads to the fusion transformations not being an optimisation. In both cases this is due to a loss of sharing.

Violating the linearity condition

Svenningsson (2002, Appendix A) states that shortcut fusion is not an improvement. This is based on a space-usage metric similar to the one that we have employed in this section. Svenningsson gives an example where *foldr/build* increases sharing and an example where *unbuild/unfoldr* loses sharing. Either an increase or a decrease in sharing is problematic: decreased sharing can lead to duplication of allocations while increased sharing can lead to increased peak heap residency.

Although the *unbuild/unfoldr* example can easily be translated into a stream version, Svenningsson's observation does not contradict the result of this section, that stream fusion – as we have defined it – is strictly an improvement. The key of course is the conditions we place on good consumers and producers. Let us consider Svenningsson's example and see which of our conditions it violates.

We start with two slightly peculiar definitions

$$
\begin{array}{l}
f :: [a] \to Int \\
f\ xs = unbuild\ f'\ xs \\
\quad \textbf{where} \\
\qquad f'\ psi\ xs = \textbf{case}\ psi\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad Just\ (a, ys) \to 1 \\
\qquad\qquad\qquad\qquad Nothing \quad \to \textbf{case}\ psi\ xs\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Nothing \quad \to 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad Just\ (b, zs) \to 1 \\
traverse\ [\,] \qquad = Nothing \\
traverse\ (x : xs) \ = traverse\ xs
\end{array}
$$

Note that *traverse* does as its name suggests and traverses its input list but always returns *Nothing* in the end. Now take the expression

$$f\ (unfoldr\ traverse\ xs)$$

Since $traverse\ xs = Nothing$ then $unfoldr\ traverse\ xs = [\,]$. In evaluating this expression the list *xs* is traversed only once: when *traverse* is called by *unfoldr*.

If we now unfold the definition of f we get an opportunity to apply the *unbuild*/*unfoldr* fusion rule

$$\begin{array}{l} unbuild\ f'\ (unfoldr\ traverse\ xs) \\ \quad \textbf{where} \\ \quad\quad f' = \ldots \end{array}$$

Applying the fusion rule gives us

$$\begin{array}{l} \textbf{case}\ traverse\ xs\ \textbf{of} \\ \quad Just\ (a, ys) \rightarrow 1 \\ \quad Nothing \quad \rightarrow \textbf{case}\ traverse\ xs\ \textbf{of} \\ \qquad\qquad\qquad\quad Nothing \quad \rightarrow 1 \\ \qquad\qquad\qquad\quad Just\ (b, zs) \rightarrow 1 \end{array}$$

We now see that *traverse xs* is evaluated twice – a loss of sharing. Prior to using the fusion rule, the result of *traverse* was memoised in the list structure. Afterwards, there is no memoisation and if the stepper function *psi* is called multiple times with the same arguments then the evaluation is duplicated.

The example can easily be translated into a stream version

$$\begin{array}{l} f :: [a] \rightarrow Int \\ f = f_s \circ stream \\ f_s :: Stream\ a \rightarrow Int \\ f_s\ (Stream\ next\ s) = go\ s \\ \quad \textbf{where} \\ \quad\quad go\ s = \textbf{case}\ next\ s\ \textbf{of} \\ \qquad\qquad\quad Yield\ a\ s' \rightarrow 1 \\ \qquad\qquad\quad Done \quad \rightarrow \textbf{case}\ next\ s\ \textbf{of} \\ \qquad\qquad\qquad\qquad\qquad\quad Done \quad \rightarrow 1 \\ \qquad\qquad\qquad\qquad\qquad\quad Yield\ b\ s'' \rightarrow 1 \end{array}$$

The key feature is that the stream worker function *go* uses *next s* twice. This is prohibited by consumer condition 8 which requires that the stream state be used linearly. If the stream state is used linearly then each step in the stream can only be evaluated once and hence there can be no loss of sharing, despite the lack of memoisation.

Violating the allocation condition

It is interesting to see an example that does not satisfy the allocation constraint of Section 4.7.4. Consider *tail*

$$
\begin{aligned}
&\mathit{tail} :: [a] \to [a] \\
&\mathit{tail}\ \mathit{xs} = \mathbf{case}\ \mathit{xs}\ \mathbf{of} \\
&\qquad\qquad [\,] \quad\ \to \mathit{error}\ \texttt{"tail []"} \\
&\qquad\qquad (x : \mathit{xs}) \to \mathit{xs}
\end{aligned}
$$

We do not even need to compare it to the stream version because the list version does not satisfy a crucial condition. Suppose we have $ys = \mathit{tail}\ xs$ and we evaluate $ys \rightsquigarrow (y : ys')$. Recall from Section 4.7.2 that we require the top level constructor $(:)$ to be allocated during the evaluation step, but this is not the case for *tail*. What would go wrong is that since there are no list $(:)$ constructor allocations that can be saved, then the stream version cannot be an optimisation.

If we wrote a non-standard and inefficient version of *tail* that reconstructed the tail, then we could easily establish the allocation constraint. The other important list function in this class is $(+\!\!+)$/*append* because it returns its second argument rather than allocating new $(:)$ constructors.

This problem does not mean that it is never profitable to fuse functions such as *tail* and *append*. Consider

$$\mathit{tail}\ (\mathit{enumFromTo}\ n\ m)$$

The combined producer does allocate fresh $(:)$ constructors. We could take equivalent stream versions and profitably fuse them with a consumer which would eliminate the intermediate $(:)$ allocations. Note however that it would only eliminate one set of $(:)$ allocations, not two as with other transformer/producer combinations that do satisfy the allocation constraint. So whether or not it is beneficial to fuse functions such as *tail* depends on the context in which it appears, however taking advantage of this opportunity in a real implementation would unfortunately add complexity.

4.8 Expressiveness

Having imposed numerous conditions on good producers and good consumers, it is reasonable to worry that we may be not just restricting the way in which fusible functions are defined, but severely restricting which fusible functions can be defined at all. In this section we assess evidence to see whether or not we have excessively restricted expressiveness.

The most obvious evidence that we have not excessively restricted expressiveness is that we can in fact define many of the standard list functions in such a way that they satisfy the good producer and good consumer conditions.

In addition, we can look for functions that we cannot define within our system. By surveying the Haskell 98 List module we can be reasonably sure that we cover the major obvious examples. When looking at functions that we cannot define, we must distinguish between inherent limitations of *unfoldr*-based fusion systems and extra limitations imposed by our particular choice of good producer and good consumer conditions.

The only major class of functions we identify, where it is our extra conditions that are to blame, is the class represented by *concatMap*. We look briefly at the difficulty with *concatMap*.

Finally we can show how to translate a significant subset of functions defined using *unfoldr* into stream versions that satisfy the good producer conditions.

4.8.1 Standard functions

We start by looking at some standard list functions where we can define versions that satisfy the good producer and good consumer conditions. In particular we can define

Producers

unfoldr,
iterate, *replicate*,
enumFrom, *enumFromTo*

Transformers

map, *filter*,
zip, *zipWith*, *zipWith3*,
scanl, *scanl1*,
init, *take*, *drop*, *takeWhile*, *dropWhile*,
nub

Consumers

foldr, *foldr1*, *foldl*, *foldl1*, *foldl'*,
head, *last*,
elem, *lookup*, *find*, *index*, *findIndex*

This list is not comprehensive but is covers many classes of list functions.

The *unfoldr* function is an important example of course but it is perhaps slightly misleading: fusing *unfoldr* only involves eliminating the list it generates; it does not cover eliminating the *Nothing* and *Just* (a, s') constructors that are produced by the *unfoldr* stepper function. It is possible to automatically eliminate these constructors using the transformations described in this chapter, but only if the *unfoldr* stepper function satisfies conditions equivalent to those imposed on stream producer's stepper functions. In particular, the stepper should be non-recursive and should be structured as a tree of case expressions with *Just*/*Nothing* terms in the leaves. Of course, without the ability to use *Skip*, the expressiveness of such stepper functions is severely limited. In Section 4.8.4 we describe a somewhat more general translation from instances of *unfoldr* to streams.

The functions *iterate*, *repeat*, *replicate* and the enumeration functions are entirely straightforward. It is interesting to note that these functions are so cheap that it may make sense to duplicate them if it enables more fusion. Consider for example

$$\begin{array}{ll}\textbf{let} & ns = enumFromTo\ n\ m \\ & xs' = zipWith\ f\ xs\ ns \\ & ys' = zipWith\ g\ ys\ ns \\ \textbf{in} & \ldots\end{array}$$

Here it would be profitable to duplicate *enumFromTo n m* into both use-sites, allowing stream fusion, rather than building and sharing the list. Compilers typically avoid such duplication since, in general, it leads to duplicated work. Some compilers, have mechanisms to allow certain function terms to be duplicated[7].

The list transformer functions *map*, *filter* and the various *zip* functions are important representative functions and we have covered them in detail in previous sections.

The *nub* function is somewhat interesting: the standard definition uses a list as an internal data structure (representing a set) but it is otherwise an unremarkable instance of both *foldr* and *unfoldr*. The stream version of *nub* works in

[7]GHC allows programmers to annotate functions as being 'constructor-like' which declares to the optimiser that the function should be considered cheap enough to duplicate if it enables a rewrite rule to be applied. For more details see the GHC users guide on the 'CONLIKE' pragma.

exactly the same way, using a list as an internal data structure; the input and output lists can however be eliminated.

The list consumers *foldr* and *foldl* are of course important patterns for consumers and we can express both in a direct fashion. Interestingly we can also express the variants *foldr1* and *foldl1* easily whereas these are hard to express in terms of *foldr* and thus hard to fuse under the *foldr/build* fusion system.

4.8.2 Functions that cannot be defined within the system

There are of course a number of functions that we cannot define at all or that we cannot define in such a way that they satisfy the good producer and good consumer conditions. We will look at a number of examples grouped by the reason for the problem.

Stream fusion is an *unfoldr*-based system and we expect to inherent many of its expressive limitations. Thus when looking at the various problems, in addition to considering the immediate reason – described in terms of the good producer and good consumer conditions – it is useful to consider whether the function in question could be defined equivalently in terms of *unfoldr*. There are a series of reasons why a list producer may not be expressible as an instance of *unfoldr*:

- not expressible as an *unfoldr* at all
- not expressible as an *unfoldr* with the same results at partial values
- not expressible as an *unfoldr* with the same sharing properties
- not expressible as an *unfoldr* with the same number of constructor allocations

Of course, looking at definability in terms of *unfoldr* only helps us with analysing limitations on stream producers; it is less helpful for stream transformers and not useful for stream consumers.

Single stream output limitation

$$
\begin{array}{ll}
\mathit{unzip} & :: [(a,b)] \rightarrow ([a],[b]) \\
\mathit{splitAt} & :: \mathit{Int} \rightarrow [a] \rightarrow ([a],[a]) \\
\mathit{partition} & :: (a \rightarrow \mathit{Bool}) \rightarrow [a] \rightarrow ([a],[a]) \\
\mathit{span}, \mathit{break} & :: (a \rightarrow \mathit{Bool}) \rightarrow [a] \rightarrow ([a],[a])
\end{array}
$$

Recall from Section 4.2.4 that we only aim to improve functions that have lists as simple inputs or outputs, not lists embedded in other types such as pairs or functions. The above group of functions return pairs of lists and they cannot be decomposed into pairs of list producers without a loss of sharing.

For example, semantically we can decompose $splitAt\ \ n\ \ xs$ into the pair $(take\ n\ xs, drop\ n\ xs)$, however the standard *splitAt* definition makes only a single traversal of *xs* while a definition in terms of *take* and *drop* may make two traversals. Multiple traversals of a stream implies duplicating the work of the stream producer.

Not being able to define fusible functions with multiple stream outputs would appear to be a fundamental limitation of *unfoldr*-based fusion systems. Recall from Section 1.3.8 that one of the touted advantages of *unbuild*/*unfoldr* fusion over *foldr*/*build* fusion is that it can fuse functions such as *zip* that consume multiple lists. We might expect some kind of duality where *unfoldr*-based systems cannot fuse functions that produce multiple lists while *foldr*-based systems can. That is, we might expect that under the *foldr*/*build* system we could define *unzip* so that it is a good producer for both output lists.

$$\begin{aligned} zip &:: [a] \to [b] \to [(a,b)] \\ unzip &:: [(a,b)] \to ([a],[b]) \end{aligned}$$

This is not the case however[8], *foldr*/*build* fusion is also unable to fuse functions that produce multiple lists. The reason is simply that *build* produces a single list.

$$build :: \forall a.\ (\forall b.(a \to b \to b) \to b \to b) \to [a]$$

It is possible to extend however: it is straightforward to abstract a list producer like *unzip* over $(:)$ and $[\,]$, and it is also straightforward to write a version of *build* that builds a pair of lists.

$$\begin{aligned} &buildPair :: \forall a\ b.\ (\forall c\ d.(a \to c \to c) \to c \to \\ &\qquad\qquad (b \to d \to d) \to d \to (c,d)) \to ([a],[b]) \\ &buildPair\ g = g\ (:)\ [\,]\ (:)\ [\,] \\ &unzip\ xs = buildPair\ (\lambda cons\ nil\ cons'\ nil' \to \\ &\qquad foldr\ (\lambda(a,b)\ \sim(as,bs) \to (a\ `cons`\ as, b\ `cons'`\ bs))\ (nil, nil')\ xs) \end{aligned}$$

Note that we must abstract over the construction of the two lists separately, otherwise we would constrain the two lists to have the same type.

[8] Gill (1996, Section 3.5.3) lists *unzip* as a good producer but this was an oversight (confirmed by personal communication).

Chitil (2000) takes this idea much further and presents an inference algorithm to generate the appropriate 'build wrapper' for each function. Despite this significant generalisation Chitil does not present any method to fuse multiple consumers with such a producer – he only presents examples such as $foldr\ f\ z\ (fst\ (unzip \ldots))$ where one of the outputs is selected and consumed with a *foldr*. Thus it appears to remain an open question as to whether functions with multiple outputs can be effectively fused in a *foldr*-based system[9].

Reliance on memoisation in a list data structure

$$\begin{aligned} scanr &:: (a \to b \to b) \to b \to [a] \to [b] \\ scanr1 &:: (a \to a \to a) \to [a] \to [a] \end{aligned}$$

Interestingly, while it is straightforward to define stream versions of *scanl* and *scanl1*, it appears not to be possible to define stream versions of *scanr* and *scanr1* at all – with or without our additional conditions. The standard definition of *scanr* is

$$\begin{aligned} &scanr\ f\ q_0\ [\,] = [q_0] \\ &scanr\ f\ q_0\ (x : xs) = f\ x\ q : qs \quad \textbf{where}\ qs@(q : \_) = scanr\ f\ q_0\ xs \end{aligned}$$

This definition allows results to be produced incrementally if the function f is non-strict in its second argument[10]. This definition appears to rely on sharing within the structure of the list: note that it memoises qs – the result of *scanr* for the tail xs. It is not possible using *unfoldr* to construct lists that have sharing within the structure of the list.

Sharing between the input and output lists

$$\begin{aligned} tail &:: [a] \to [a] \\ cons &:: a \to [a] \to [a] \\ append &:: [a] \to [a] \to [a] \\ insert &:: Ord\ a \Rightarrow a \to [a] \to [a] \\ delete &:: Eq\ a \Rightarrow a \to [a] \to [a] \end{aligned}$$

We looked at the *tail* function previously in Section 4.7.11. The problem is that the original list versions functions do not satisfy the allocation constraint

[9] Hinze et al. (2011) present a 'parallel hylo-ana' rule which presumably could be dualised to a 'parallel cata-hylo' rule, but the practical implications remain unclear.

[10] For example $scanr\ (:)\ [\,]\ (1 : 2 : \bot) = (1 : 2 : \bot) : (2 : \bot) : \bot$

because they do not construct a completely fresh list, instead they return some tail of the input list. As we discussed in Section 4.7.11, this is not necessarily a fatal problem because in many cases the input list to these functions is freshly constructed. In such cases there is still some saving to be had, and there is no risk of loss of sharing by stream fusion. Accurately identifying these cases however adds complexity to a stream fusion implementation.

Sharing within the output list

$$\begin{aligned} &\mathit{repeat} :: a \to [a] \\ &\mathit{cycle} :: [a] \to [a] \end{aligned}$$

The *repeat* and *cycle* functions are interesting examples. The standard list versions of these functions construct cyclic list data structures

$$\begin{aligned} &\mathit{repeat}\ x = xs' \ \textbf{where}\ xs' = x : xs' \\ &\mathit{cycle}\ xs = xs' \ \textbf{where}\ xs' = xs +\!\!+ xs' \end{aligned}$$

Recall that the allocation constraint requires one $(:)$ constructor to be allocated per list element. The definition of *repeat* only allocates one $(:)$ for the entire sequence. Stream fusion will still be an optimisation in this case but it only saves one allocation in total, rather than one allocation per sequence element. Arguably the allocation constraint should be relaxed to allow this case. A relaxed constraint would have to distinguish this case, where fresh allocations are reused multiple times, from the examples above where the $(:)$ allocations from the input are reused in the output.

The definition of *cycle* is very similar to that of *repeat* but its stream equivalent is more subtle. The most obvious stream version would violate the linearity constraint by reusing the initial stream state each time round the sequence. It is right to exclude such a version because in general it could lead to unbounded duplication of work. Again, there will be special cases where the duplication is bounded and the stream version is still profitable. Alternatively, instead of saving the initial state a stream version could store the elements of the input sequence in some data structure. This would necessarily mean it does not satisfy the allocation constraint, though such a definition could at least be neutral in terms of allocation change.

No allocation advantage

$$reverse :: [a] \rightarrow [a]$$

We can write a stream version of *reverse* however it has to use a data structure to hold the accumulated input sequence before producing the result. The list version of *reverse* uses a list as the data structure for accumulating the input sequence and is able to reuse the same data structure as the result. The stream version on the other hand must traverse its internal data structure to produce the output sequence. If we do the allocation accounting we find that the stream version uses one more allocation per sequence element and thus after fusion the allocation change is exactly zero. So *reverse* is an example where stream fusion works but does not save any allocations. It may still be worthwhile to fuse, but to decide we would need either a more detailed cost model or empirical measurements.

Higher order stream input

$$\begin{array}{lll} concatMap :: (a \rightarrow [b]) \rightarrow [a] & \rightarrow [b] \\ concat \quad :: & [[a]] \rightarrow [a] \end{array}$$

The same constraint as above that excludes *unzip* etc. also excludes *concatMap*. In the case of *concatMap* however it is not a expressiveness limitation inherited from *unfoldr*, it is an additional limitation[11]. We can define *concatMap* on streams, indeed in Section 3.9.7 we proved that our definition respects the stream abstraction property. The problem is that we cannot guarantee that we can optimise uses of *concatMap* and we have taken the approach of constraining our definition of good producers and consumers to those that we can guarantee to optimise.

[11] It should be noted however that this is not a disadvantage compared to *unbuild/unfold* fusion as no method for expressing and optimising *concatMap* has been presented for this system.

4.8.3 The challenge of optimising *concatMap*

The function *concatMap* is important because it represents the entire class of nested list computations, including list comprehensions (Peyton Jones et al., 2003, Section 3.11). The *foldr*/*build* system is extremely effective for this class of functions. It is thus a major limitation of stream fusion that, at present, there is no known reliable method of optimising uses of *concatMap*.

The first stage involving the *stream*/*unstream* fusion rule is straightforward and works as expected. The list wrapper for the stream version of *concatMap* is

$$\begin{aligned}&\mathit{concatMap} :: (a \to [b]) \to [a] \to [b]\\&\mathit{concatMap}\, f = \mathit{unstream} \circ \mathit{concatMap}_s\ (\mathit{stream} \circ f) \circ \mathit{stream}\end{aligned}$$

Suppose we have good producers f and g, and a good consumer h

$$\begin{aligned}f &= \mathit{unstream} \circ f_s\\g &= \mathit{unstream} \circ g_s\\h &= h_s \circ \mathit{stream}\end{aligned}$$

If we directly compose these together with *concatMap* then we can follow the standard transformations from Section 4.2, that is, unfolding followed by applying the *stream*/*unstream* rule.

$$\begin{aligned}&h \circ \mathit{concatMap}\, f \circ g\\=\;&h_s \circ \mathit{stream} \circ \mathit{unstream} \circ \mathit{concatMap}_s\ (\mathit{stream} \circ \mathit{unstream} \circ f_s)\\&\quad \circ \mathit{stream} \circ \mathit{unstream} \circ g_s\\=\;&h_s \circ \mathit{concatMap}_s\, f_s \circ g_s\end{aligned}$$

The subsequent phases are problematic.

In previous work (Coutts et al., 2007b, Section 7.2) we described two techniques that are effective in some cases: the static argument transformation, and specialising on partial applications. We will not consider these two techniques in detail, rather we will briefly outline what makes the problem non-trivial and survey some possible alternative approaches.

The definition of $concatMap_s$ is

$$
\begin{aligned}
&concatMap_s :: (a \to Stream\ b) \to Stream\ a \to Stream\ b \\
&concatMap_s\ f\ (Stream\ next_a\ s_a) = Stream\ next\ (s_a, Nothing) \\
&\quad \textbf{where} \\
&\qquad next\ (s_a, Nothing) = \\
&\qquad\quad \textbf{case}\ next_a\ s_a\ \textbf{of} \\
&\qquad\qquad Done \to Done \\
&\qquad\qquad Skip\ s'_a \to Skip\ (s'_a, Nothing) \\
&\qquad\qquad Yield\ a\ s'_a \to Skip\ (s'_a, Just\ (f\ a)) \\
&\qquad next\ (s_a, Just\ (Stream\ next_b\ s_b)) = \\
&\qquad\quad \textbf{case}\ next_b\ s_b\ \textbf{of} \\
&\qquad\qquad Done \to Skip\ (s_a, Nothing) \\
&\qquad\qquad Skip\ s'_b \to Skip\ (s_a, Just\ (Stream\ next_b\ s'_b)) \\
&\qquad\qquad Yield\ b\ s'_b \to Yield\ b\ (s_a, Just\ (Stream\ next_b\ s'_b))
\end{aligned}
$$

Before looking at the details of the *next* stepper function, it is helpful to compare *concatMap* with a traditional nested loop construct. One can either see this as an analogy or as a goal since if a *concatMap* stream is consumed by a strict left fold then the ideal code to generate is exactly a nested loop. We can characterise the nested loop in imperative pseudocode as:

```
foreach (x in xs) {
  foreach (y in (f x)) {
    ...
  }
}
```

If we consider the evaluation states for a nested loop, we have two modes: in one mode we are in the outer loop performing calculations to decide if there will be a next iteration and setting up values that will be needed by the inner loop; in the other mode we are in the inner loop performing an iteration.

The stepper function of $concatMap_s$ follows this pattern. The outer mode has just the state of the outer stream. In the outer mode the stepper function pulls from the outer stream. When it obtains an element from the outer stream it applies a function to obtain the inner stream and it steps to the inner mode. The inner mode holds the current state of the outer stream and the entire inner stream – stepper and current state. In the inner mode the stepper function pulls from the inner stream. When the inner stream is exhausted it steps back to the outer mode.

In a typical nested loop, there is a single inner loop and it is parametrised by variables in the outer loop. In contrast, it is possible to have multiple independent inner loops, with independent sets of state variables, and there is the potential for each iteration of the outer loop to select a different inner loop.

```
foreach (x in xs) {
  switch (f x)
    0:  foreach (y in ...
    1:  foreach (y in ...
    2:  foreach (y in ...
    ...
}
```

The construction of $concatMap_s$ is sufficiently general to express these non-typical nested loops and it is this degree of expressiveness that makes it hard to optimise.

In the $concatMap_s$ stepper function, the inner mode contains the whole inner stream and thus in principle each iteration of the outer loop can have an inner loop with a different stepper function and even a different internal state type. We must write $concatMap_s$ with this level of generality if we are to give it its usual type. The type of $concatMap_s$ allows examples such as the following where we choose completely different streams – with different stepper functions and state types – depending on a runtime test.

$$concatMap_s\ (\lambda x \rightarrow \textbf{if}\ p\ x\ \textbf{then}\ Stream\ next_a\ (s_a\ x)\ \textbf{else}\ Stream\ next_b\ (s_b\ x))$$

It is however quite unreasonable to expect that we can optimise such examples.

As an aside, it is worth noting that the *foldr/build* system cannot handle this example either and for essentially the same reason. Under the *foldr/build* system the crux of this example would be a situation like

$$foldr\ c\ n\ (\textbf{if}\ p\ x\ \textbf{then}\ build\ (..)\ \textbf{else}\ build\ (..))$$

The problem would be that we could not make use of the *foldr/build* rule, since the *foldr* is not applied directly to the *build*.

By way of example, recall from Section 4.2 that we only expect to fuse terms where a good consumer is applied directly to a good producer. This requirement is so that we can apply the *stream*/*unstream* rule statically. It means we cannot hope to improve examples such as

$$\begin{aligned} &\mathit{consume}_s\ (\mathit{stream}\ (\textbf{if}\ \mathit{even}\ n\ \textbf{then}\ \mathit{unstream}\ (\ldots) \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \ \mathit{unstream}\ (\ldots))) \end{aligned}$$

Similarly, we require that good producers use a single *unstream* applied to a term that unfolds to a *Stream* term. We have this requirement so that we can statically apply stream consumer/producer or transformer/producer fusion.

Given this requirement on stream producers, then in the context of $\mathit{concatMap}_s$, we should only be faced with terms such as the following.

$$\mathit{concatMap}_s\ (\lambda x \rightarrow \mathit{Stream}\ \mathit{next}_b\ (f\ x))$$

This is a form that we can hope to optimise because it uses a single inner stream where the initial state is parametrised by the element value from the outer stream.

Possible solutions

Even if we restrict our attention to uses of $\mathit{concatMap}_s$ of the above form, finding a reliable optimisation is still a challenge. As mentioned previously, the techniques we have described elsewhere (Coutts et al., 2007b, Section 7.2) do work in many cases and further improvements along these lines are no doubt possible. The primary disadvantage of these implementation-focused techniques is complexity, both in implementation and explanation. It is hard to give an explanation, such as the one in this chapter, that sets out what form of input terms are required and gives an argument that the transformations can always be applied and that they lead to an improvement.

A further practical problem is that not only do these techniques do not work with all examples, but that this is discovered too late, after they are already committed. The result can be a significant pessimisation: programs are left with many extra intermediate data structures, the *Step* constructors and the constructors used for state shapes.

An alternative approach that could plausibly lead to a reasonable semi-formal argument is to return to the state machine form that we already know how to optimise. The aim would be to find a suitable form of state machine composition that takes a state machine for the outer stream and a parametrised inner state machine and gives a single flattened state machine.

Another plausible approach starts by looking at the form that we expect to handle

$$concatMap_s\ (\lambda x \to Stream\ next_b\ (f\ x))$$

The idea is to write a specialised variant of $concatMap_s$ that works for exactly this form. That is, we specify $concatMap'_s$ as

$$concatMap'_s\ next_b\ f = concatMap_s\ (\lambda x \to Stream\ next_b\ (f\ x))$$

The hope of course is that we can find a definition for $concatMap'_s$ that takes advantage of the fact that we now have just a single inner stepper function and a single inner state type. We can in fact do exactly that

$$\begin{array}{l}
concatMap'_s :: (a \to Step\ b\ s) \to (a \to s) \to Stream\ a \to Stream\ b \\
concatMap'_s\ next_b\ f\ (Stream\ next_a\ s_a) = Stream\ next\ (s_a, Nothing) \\
\quad \textbf{where} \\
\quad\quad next\ (s_a, Nothing) = \\
\quad\quad\quad \textbf{case}\ next_a\ s_a\ \textbf{of} \\
\quad\quad\quad\quad Done \to Done \\
\quad\quad\quad\quad Skip\ s'_a \to Skip\ (s'_a, Nothing) \\
\quad\quad\quad\quad Yield\ a\ s'_a \to Skip\ (s'_a, Just\ (f\ a)) \\
\quad\quad next\ (s_a, Just\ s_b) = \\
\quad\quad\quad \textbf{case}\ next_b\ s_b\ \textbf{of} \\
\quad\quad\quad\quad Done \to Skip\ (s_a, Nothing) \\
\quad\quad\quad\quad Skip\ s'_b \to Skip\ (s_a, Just\ s'_b) \\
\quad\quad\quad\quad Yield\ b\ s'_b \to Yield\ b\ (s_a, Just\ s'_b)
\end{array}$$

If we can guarantee that $f\ a$ statically constructs a state shape then this stream definition satisfies all the stream producer conditions.

The main difficulty of this approach is that it uses a non-trivial rewrite rule

$$concatMap_s\ (\lambda x \to Stream\ next_b\ (f\ x)) = concatMap'_s\ next_b\ f$$

This rewrite rule would have to be applied by the compiler at the appropriate point during compilation. The first difficulty is that it relies on matching under a lambda. Additionally, to be useful we must match not just on explicit functions that construct the initial state, but expressions with particular free variables. We actually want to match lambda expressions of the following form

$$\lambda x \to Stream\ (\langle stepper \rangle [\,])\ \ (\langle e \rangle [x])$$

The expression $\langle e \rangle$ may have x as a free variable, while we require that x is not a free variable in the stepper function. In the result, we want to capture and

substitute for the free variable x so that we can construct a suitable function to pass to $concatMap'_s$

$$concatMap'_s\ \langle stepper \rangle\ (\lambda y \rightarrow \langle e \rangle [x := y])$$

So while this approach is relatively promising, it relies on a rule matching and rewrite language that is rather richer than those of most compilers.

List comprehensions

It should be noted that while the meaning of Haskell's list comprehensions are specified in terms of *concatMap*, tackling *concatMap* directly is not the only approach to handling list comprehensions. The common translation does not use *concatMap* and the *foldr/build* system has a special translation into uses of *foldr* and a single *build*.

To date we have not found an alternative translation that makes it possible to handle list comprehensions with stream fusion. This is perhaps not surprising because such a translation would likely also give us a solution for *concatMap* itself, since *concatMap* can be defined straightforwardly as a list comprehension. The *foldr/build* system can already handle *concatMap*, its special translation for list comprehensions does not in principle add any new capability, it just improves the robustness of the optimisation by reducing the number of rewrite rules the compiler must apply.

4.8.4 Converting definitions from *unfoldr* to streams

It is possible to convert many – but not all – list producing functions that are an instance of *unfoldr* into stream versions satisfying the good producer conditions. The key is to convert the *unfoldr* stepper function into a stream stepper function that satisfies the stream producer conditions. Essentially, we translate by replacing recursive calls with skips to new states.

We can approach a translation by looking at the general form of definitions we have before and after stream consumer/producer fusion. An *unfoldr* stepper function has aspects of each. Given a stream producer *Stream next s*, applying stream consumer/producer fusion to *unstream* (*Stream next s*) gives us a set of mutually recursive function definitions

$$\begin{aligned} g_0\ x_0 \ldots x_{i_0} &= t_0 \\ &\vdots \\ g_n\ x_0 \ldots x_{i_n} &= t_n \end{aligned}$$

Each function body expression t_i takes the form

$$
\begin{aligned}
t ::= & \ [\,] \\
| & \ e : g_j \ e_0 \ldots e_{i_j} \\
| & \quad g_j \ e_0 \ldots e_{i_j} \\
| & \ \textbf{case} \ e \ \textbf{of} \ p_0 \rightarrow t_0; \ldots \\
| & \ \textbf{let} \ \langle binds \rangle \ \textbf{in} \ t
\end{aligned}
$$

The e terms are unrestricted expressions. There are many stepper functions *next* that would give rise to this set of recursive functions but a simple form with a clear structural similarity is

$$
\begin{aligned}
& next \ (State_0 \ x_0 \ldots x_{i_0}) = t_0 \\
& \vdots \\
& next \ (State_n \ x_0 \ldots x_{i_n}) = t_n
\end{aligned}
$$

We have a single non-recursive *Step* function with multiple simple clauses. The body expression t_i of each clause takes the form

$$
\begin{aligned}
t ::= & \ Done \\
| & \ Yield \ e \ (State_j \ e_0 \ldots e_{i_j}) \\
| & \ Skip \quad (State_j \ e_0 \ldots e_{i_j}) \\
| & \ \textbf{case} \ e \ \textbf{of} \ p_0 \rightarrow t_0; \ldots \\
| & \ \textbf{let} \ \langle binds \rangle \ \textbf{in} \ t
\end{aligned}
$$

Given an *unfoldr* term *unfoldr phi a*, the stepper function *phi* has aspects both of the stream stepper function *next* and of the recursive functions $g_0 \ldots g_n$. Like the function *next* it is a stepper function, but unlike *next*, it may be recursive and it may use auxiliary recursive functions similar to $g_0 \ldots g_n$.

Supposing that we can write *phi* as f_0 where $f_0 \ldots f_n$ take the following form

$$
\begin{aligned}
& f_0 \ x_0 \ldots x_{i_0} = t_0 \\
& \vdots \\
& f_n \ x_0 \ldots x_{i_n} = t_n
\end{aligned}
$$

$$
\begin{aligned}
t ::= & \ Nothing \\
| & \ Just \ (e, f_j \ e_0 \ldots e_{i_j}) \\
| & \ f_j \ e_0 \ldots e_{i_j} \\
| & \ \textbf{case} \ e \ \textbf{of} \ p_0 \rightarrow t_0; \ldots \\
| & \ \textbf{let} \ binds \ \textbf{in} \ t
\end{aligned}
$$

There is a one-to-one correspondence with the form of stepper functions above and it is reasonably clear that we can perform a translation. We start with

the top level functions $f_0 \ldots f_n$. Each one is translated into a clause of the final stepper function *next*. Each one gets a distinct data constructor for the stream state.

$$[\![\, f_i\ x_0 \ldots x_{i_0} = e_i \,]\!] \quad = \quad next\ (State_i\ x_0 \ldots x_{i_0}) = [\![\, e_i \,]\!]$$

The body term of each clause is translated as follows.

$$\begin{array}{lcl}
[\![\, Nothing \,]\!] & = & Done \\
[\![\, Just\ (e, f_i\ e_0 \ldots e_n) \,]\!] & = & Yield\ e\ (State_i\ e_0 \ldots e_n) \\
[\![\, f_i\ e_0 \ldots e_n \,]\!] & = & Skip \quad (State_i\ e_0 \ldots e_n) \\
[\![\, \textbf{case}\ e\ \textbf{of}\ p_0 \to t_0; \ldots \,]\!] & = & \textbf{case}\ e\ \textbf{of}\ p_0 \to [\![\, t_0 \,]\!]; \ldots \\
[\![\, \textbf{let}\ binds\ \textbf{in}\ t \,]\!] & = & \textbf{let}\ binds\ \textbf{in}\ [\![\, t \,]\!]
\end{array}$$

In particular note that calls to the functions $f_0 \ldots f_n$ translate into skips to new stream states.

Clearly, the form of definition we can translate from is still quite limited. There are a few ways we can increase the range of definitions that fit this form.

Firstly we may need to unfold definitions of library and auxiliary functions that are used in tail calls, since we need to translate their bodies.

Secondly we can perform full lambda lifting so instead of functions defined in local **let** clauses they are moved to top level functions. Otherwise local **let** values can only be used in argument positions, not in tail calls. That is we cannot translate the following pattern where x is a lambda or **let**-bound variable

$$[\![\, x\ e_0 \ldots e_n \,]\!] \quad \{\ \text{error}\ \}$$

For example we cannot translate the following stepper function

$$\begin{array}{l}
\textbf{let}\ x :: Maybe\ (x, s) \\
\quad x = \ldots \\
\textbf{in case}\ f\ a\ \textbf{of} \\
\qquad A \to x \\
\qquad B \to Nothing \\
\qquad C \to x
\end{array}$$

In this case we could duplicate x into both branches, but in general the problem remains.

A further technique is to lift expressions to new top level functions. In particular if we meet the following pattern

$$[\![\ \mathit{Just}\ (e, e')\ ]\!]$$

We can abstract over the free variables of e' and define a new top level function. By replacing e' with a call to the new function then it matches the existing pattern $\mathit{Just}\ (e, f_i\ e_0 \ldots e_n)$.

A similar technique would be to translate from general recursive list-generating definitions. We simply change the cases for where the stepper function returns *Nothing* and *Just* into

$$\begin{array}{lcl} [\![\ [\,]\ ]\!] & = & \mathit{Done} \\ [\![\ e : f_i\ e_0 \ldots e_n\ ]\!] & = & \mathit{Yield}\ e\ (\mathit{State}_i\ e_0 \ldots e_n) \end{array}$$

Of course there are still syntactic restrictions and such a translation does not cover consuming or transforming lists.

Chapter 5

Related work

The most closely related work is of course the *foldr/build* fusion system (Gill et al., 1993; Gill, 1996) and the *unbuild/unfoldr* fusion system (Svenningsson, 2002). We have covered both systems in detail in Chapter 1 and made frequent comparison to them in Chapters 3 and 4. In this chapter we briefly review other related work.

5.1 Stream fusion and shortcut fusion theory

5.1.1 Mechanised fixpoint induction proofs

Huffman (2009) has developed a formalisation of stream fusion using the formal proof tool Isabelle. The formalisation uses Isabelle/HOLCF (Regensburger, 1995) which is an extension of the standard Isabelle/HOL for working with continuous functions in CPOs. He proves that $stream\ (unstream\ s) \approx s$, equivalent to our Lemma 3.8.2, and also proves the abstraction property for a number of standard functions including *map*, *filter*, *foldr*, *enumFromTo*, *append*, *zipWith* and *concatMap*.

The most obvious difference with our presentation in Chapter 3 is that Huffman's proofs are machine-checked while those in Chapter 3 are by hand and are aimed at explaining the proof strategy. The other significant difference is that he does not define $\approx$ as a logical relation covering all types, rather it is defined as a relation at just the *Stream* type. Similarly, no general data abstraction relation is defined. As a consequence it is not possible to express the general stream fusion rule (Theorem 3.8.3). This reflects the focus of the formalisation which is in part to demonstrate the utility of Isabelle/HOLCF for proofs about lazy functional programs in Haskell, rather than the focus being on providing an end-to-end proof of correctness of stream fusion.

If the logical relation version of $\approx$ can be expressed reasonably in Isabelle/HOLCF then it seems likely that it would be relatively straightforward to express and prove the general stream fusion rule. Similarly, one would wish to express the data abstraction property as a logical relation and prove that stream functions satisfying the data abstraction property also satisfy the precondition for the stream fusion rule (see Section 3.8.3).

5.1.2 Proof techniques for abstract data types

Gibbons (2008) presents a theory of abstract data types that has application to streams and to stream fusion. The intention is to be able to describe and prove interesting properties about abstract types. He starts from the premise that abstract types have existential types, meaning that existential quantification is used to hide the internal representation type of an abstract type. He argues that a useful semantics for abstract types is co-data and that taking this view enables the use of proof techniques for co-recursive programs (Gibbons and Hutton, 2005), particularly techniques making use of the properties of final co-algebras. The final co-data arises by taking an ADT and making a (usually infinite) tree by repeatedly applying all operations in the ADT interface. Equality between ADTs then coincides with equality between the trees.

As an example, Gibbons (2008, Section 4) applies this idea to streams: equality on streams corresponds to equality on their unfolding to lists. He proves that $stream \circ unstream = id$ for non-skipping streams by showing it is an instance of the *unbuild/unfoldr* rule which he proves using free theorems. This is essentially equivalent to our proof of Theorem 3.5.2 given in Section 3.5.2. He gives an additional proof of the same theorem using the universal property of *unfoldr*.

He goes on to discuss skipping streams and equality of skipping streams as "equivalence modulo *Skips*", which is again equality of their unfolding to lists. As an example, he proves one of the monad laws for streams, which requires proving an equality between two stream terms. The proof makes use of the universal property of *unfoldr* via *unfoldr* fusion. It also makes use of fixpoint induction for a step involving a function that does the non-productive conversion from the *Step x s* functor to the *Maybe* (x, s) functor.

In Chapter 3 we did not directly need to address the issue of equality between streams because the proof of the fusion rule and the proofs of the abstraction property only involve equalities between lists.

Gill and Hutton (2009) describe the worker/wrapper transformation which is a simple and elegant approach to data refinement that covers a wide range

of examples. While *foldr/build* makes for a quite interesting instance of the system, stream fusion appears to be a rather uninteresting instance because stream functions are non-recursive and it is the ability of the worker/wrapper transformation to handle data refinements for recursive functions that is its most notable feature.

5.1.3 Shortcut fusion proofs based on hylomorphisms

Meijer et al. (1991) did much to popularise a style of calculating programs written in terms of recursion combinators, namely folds, unfolds and combinations thereof. They present a rich set of laws for manipulating and fusing functions described using these combinators. In particular they describe *hylomorphisms* which generalise folds and unfolds. A hylomorphism is specified using an algebra and a co-algebra. One can think of a hylomorphism as using an unfold to generate a virtual data structure followed by a fold that consumes the structure.

Inspired by the *foldr/build* fusion rule of Gill et al. (1993), Takano and Meijer (1995) give a generalisation of *foldr/build* to arbitrary inductive data. They also describe a dual law for unfolds which Gill (1996) subsequently calls the *unbuild/unfold* rule. They suggest the use of hylomorphisms for expressing and fusing a wider range of functions.

Hinze et al. (2011) present a fusion theory using hylomorphisms that has particular relevance to shortcut fusion including the *foldr/build*, *unbuild/unfoldr* and stream fusion systems. The motivation for the development of the theory is in part due to dissatisfaction with the use of syntactic methods, such as fixpoint induction, in proving fusion rules. Fixpoint induction is often seen as rather tedious and "low-level proof method" (Gibbons and Hutton, 2005, Section 3). Though the tedium may be manageable using by automated proof assistants, the greater complaint is that fixpoint induction tends not provide the insights that more structured methods can give.

Hinze et al. (2011, Section 3) define a hylomorphism for an F-algebra a and an F-co-algebra c as a function h that satisfies

$$h = a \circ F\ h \circ c$$

The major innovation compared to previous work on hylomorphism-based fusion is that by restricting c to a subcategory of co-algebras known as *recursive* co-algebras, then the above equation becomes the definition of a unique h. That is, given an F-algebra a and a recursive F-co-algebra c then there is a unique h satisfying $h = a \circ F\ h \circ c$. Just as uniqueness in the universal properties of fold

and unfold leads to fold and unfold fusion laws, the uniqueness property for these hylomorphisms leads to laws for fusing hylomorphisms with algebras, co-algebras and other hylomorphisms.

The restriction to recursive co-algebras has particular advantages in the category SET, or categories that are typically used for models of System F, since it allows algebras and co-algebras to be treated together in one framework. Even in CPOs where data and co-data coincide, the restriction is still useful due to the uniqueness property.

General fusion with algebras, co-algebras or hylomorphisms involves non-trivial side conditions. The key characteristic of shortcut fusion is the use of more limited but simpler fusion laws with no side conditions, which in turn makes it practical to apply shortcut fusion laws automatically in an optimising compiler. The hylo fusion system can express the *fold*/*build* and *unbuild*/*unfold* fusion laws (though the proof of each relies on the free theorem about *build* and *unbuild* respectively).

Hinze et al. (2011, Section 5.6) also apply the hylo system to stream fusion, including the details of skipping streams. A central idea is to express stream transformers using natural transformations on the *Step* functor. In their formulation: stream consumers are defined using algebras; stream producers using co-algebras and stream transformers using natural transformations. Repeated use of algebra/hylo and hylo/co-algebra fusion allows a linear pipeline of stream consumers, transformers and producers to be fused into a single hylomorphism.

A particularly satisfying aspect of this system is the precise characterisation of stream transformers, which is absent from Chapter 3 and only described informally in Chapter 4.

The system is described in a categorical style and much of the system is independent of the underlying category. Some assumptions have to be proved differently in different categories, for example how one proves that a co-algebra is recursive is different between the category SET and CPO.

It is interesting to compare the hylo system and the system described in Chapter 3 in terms of the proof obligations for library authors. Hinze et al. (2011, Section 5.6) give the example of *filter*. For our system in Chapter 3, each function requires a proof that it satisfies the abstraction property $f_s \; \mathcal{A}\, f$, which for *filter* is

$$unstream \circ filter_s\ p = filter\ p \circ unstream$$

This property is then proved directly using fixpoint induction. While it is not elegant, it is not terribly difficult and as Huffman (2009) has demonstrated, the proofs can be partially automated and machine checked.

To describe the proof obligations with the hylo system we first need a few definitions. Instead of writing *unstream*, we write *foldr fromStep* and *unfoldr toStep* to expose and emphasise the algebra *fromStep* and the co-algebra *toStep*.

$$
\begin{aligned}
&\mathit{toStep} &&:: [a] &&\to \mathit{Step}\ a\ [a] \\
&\mathit{fromStep} &&:: \mathit{Step}\ a\ [a] &&\to [a]
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{toStep}\ [\,] &&= \mathit{Done} \\
&\mathit{toStep}\ (x : xs) &&= \mathit{Yield}\ x\ xs \\
&\mathit{fromStep}\ \mathit{Done} &&= [\,] \\
&\mathit{fromStep}\ (\mathit{Skip}\ \ xs) &&= xs \\
&\mathit{fromStep}\ (\mathit{Yield}\ x\ xs) &&= x : xs
\end{aligned}
$$

Similarly, the stream transformer filter_s is written as $\mu\ \mathit{filterStep}$. The μ operator lifts a natural transformation to a fold, or alternatively there are also equivalent unfolds or hylos.

$$
\begin{aligned}
&\mathit{filterStep}\ p\ \mathit{Done} &&&&= \mathit{Done} \\
&\mathit{filterStep}\ p\ (\mathit{Skip}\ \ s) &&&&= \mathit{Skip}\ \ s \\
&\mathit{filterStep}\ p\ (\mathit{Yield}\ x\ s) &&\mid p\ x &&= \mathit{Yield}\ x\ s \\
&&&\mid \mathit{otherwise} &&= \mathit{Skip}\ \ s
\end{aligned}
$$

Finally, what Hinze et al. call the data abstraction property[1] for *filter* is

$$\mathit{foldr}\ \mathit{fromStep} \circ \mu\ \mathit{filterStep} \circ \mathit{unfoldr}\ \mathit{toStep} = \mathit{filter}$$

The claim is that since the left hand side is expressed in terms of folds, unfolds and natural transformations then using the various fusion and computation laws the proof is straightforward and it is left as an exercise to the reader. Note also that we have the obligation to prove that *filterStep* is indeed a natural transformation.

To summarise the proof obligations:

- for all functions we must prove the data abstraction property;
- for producers we must prove that the co-algebra is recursive;
- for transformers we must prove the step transformer is a natural transformation.

[1] This is a different and strictly weaker data abstraction property than the one used in Chapter 3.

In SET the recursive property of co-algebras is implied by termination. In CPO however co-algebras that produce infinite structures are possible and indeed useful. No suggestion is given as to how to prove that co-algebras are recursive.

While for a linear pipeline of consumers, transformers and producers, the system is relatively straightforward, the situation for tree shaped combinations is more complicated. We get trees when we have stream consumers or transformers that consume multiple streams, such as *zip*. Hinze et al. (2011, Section 5.4) present a special fusion law for *parallel hylo-ana fusion* – that is, the fusion of a transformer like *zip* with a pair of unfolds. The proof obligation for the transformer stepper function is that it must transform a pair of recursive co-algebras into a recursive co-algebras. No suggestion is given for how to prove such a property.

While it would be interesting to compare the typical proof strategies that library authors may use to discharge their various proof obligations – particularly in the context of CPOs – it remains as further work. It also remains to be seen how insights from the more structured approach using hylomorphisms might affect the practice of how shortcut fusion, streams or otherwise, may be implemented.

5.1.4 Parametricity and free theorems

Historically, shortcut fusion rules have been justified using free theorems (e.g. Gill et al., 1993, Section 3.4). Johann (2003, Section 1.2) pointed out that free theorems' "correctness has not yet been proved for the languages to which it is applied" because free theorems rely on the parametricity property of models of System F but the fusion rules were being applied in languages of partial functions such as Haskell. It is common practice during program development to apply rules about total functions in the context of languages of partial functions, and, perhaps somewhat surprisingly, this "fast and loose" reasoning turns out to be at least partially justified (Danielsson et al., 2006). This justification covers monomorphic functions and hence does not extend to free theorems.

Johann (2003) proves that the *foldr/build* rule is correct in the context of a language PolyPCF – a strict polymorphic lambda calculus with general recursion and lazy lists. A notion of parametricity is obtained by a careful construction of a logical relation that captures contextual equivalence. While Johann's development was is a step in the right direction, since lazy languages are not captured by PolyPCF, the problem remained of employing shortcut fusion in lazy languages without an adequate proof of correctness.

Svenningsson (2002, Section 3.3) suggests the approach taken by Johann (2003) as a method by which one might prove the *unbuild/unfoldr* rule, but of course this would still not justify the use of the rule in languages like Haskell.

Johann and Voigtländer have a series of papers on the topic of free theorems and shortcut fusion rules in the presence of *seq*. They demonstrate (Johann and Voigtländer, 2006), with counterexamples, that both the *foldr/build* and *unbuild/unfoldr* rules are wrong when the language contains polymorphic *seq*. In particular, for each rule they gives examples where application of the rule make the program less defined, and examples where the rule makes the program more defined. This contradicted the previous 'folk theorem' that "free theorems remains valid in the presence of *seq* if all of the functions appearing in it are strict and total". Johann and Voigtländer showed how the fusion rules can be fixed, but their revised rules require non-trivial side conditions on the strictness of the arguments of *foldr* and *unfoldr*. Voigtländer has also investigated alternative shortcut fusion rules that do not have problematic side conditions in the presence of *seq*, including an *unbuild/build* rule (Voigtländer, 2008a) and a rule called *pfold/buildp* (Voigtländer, 2008b). He also proposes reformulations of the *foldr/build* and *unbuild/unfoldr* rules which introduce extra terms in the result to avoid problems with *seq*. It remains to be seen in a practical implementation if these changes interfere with the subsequent optimisation phases – which as we know from Chapter 4 are not always trivial. The combined effect of Johann and Voigtländer's work is that it is no longer possible for implementors of fusion systems to get away with ignoring *seq* or with relying on naïve use of free theorems[2].

In Chapter 3 we have been able to side-step the issues that Johann and Voigtländer raise while still using a realistic semantic model that supports *seq*. Our fusion rule has a side condition that each fusible function must satisfy and this condition is much more demanding than the strictness properties described by Johann and Voigtländer (2006). Thus in some sense the system we propose is worse. What makes it manageable however is that each fusible function must in any case satisfy a data abstraction property, and this property subsumes the side condition for the fusion rule and thus only one proof per library function is required.

[2]Indeed, more recently Seidel and Voigtländer (2010) have automated the construction of counter-examples to naïve free theorems.

5.2 General deforestation techniques

The major difficulty in trying to eliminate intermediate data structures is recursion. Eliminating intermediate structures in non-recursive code is relatively straightforward, using simple local transformations such as case-of-case. There are two major approaches to tackling the recursion problem: one is to identify and capture recursion using combinators and the other is to tackle recursion directly using more sophisticated algorithms and transformations.

In Chapter 1 we discussed the Burstall and Darlington (1977) fold-unfold calculation system. The fold-unfold system is very general and many subsequent developments are instances of the system – though they are designed with automation in mind. One such system that we mentioned in Chapter 1 is the Wadler (1990b) deforestation algorithm.

Gill (1996, Section 6.1) makes a comparison between *foldr/build* and various extensions of Wadler's deforestation algorithm. Two classes that he identifies where deforestation removes lists that *foldr/build* cannot are functions such as *zip* that consume multiple input lists, and "irregular" consumers such as *foldr1*. It is interesting to note that *unfold*-based fusion handles both classes well.

Sørensen et al. (1994) compare several related systems: Wadler's deforestation algorithm, supercompilation, partial evaluation and 'generalised partial computation'. Deforestation is much more limited in its ambitions than the other techniques.

While many program optimisation transformations are an instance of partial evaluation, deforestation is not one of them. On the other hand, supercompilation has a great deal of overlap with both deforestation and with partial evaluation. Supercompilation was first introduced by Turchin (1986) and there have been several subsequent developments and variations on the theme. Supercompilation algorithms are program transformation algorithms that involve unfolding function definitions (including recursive definitions), evaluating, simplifying and emitting a residual program. One of the major challenges is ensuring termination and this is one of the main areas of variation between different supercompilation techniques.

There has been a recent resurgence of interest in supercompilation in the context of Haskell (Mitchell and Runciman, 2008; Mitchell, 2010; Bolingbroke and Peyton Jones, 2010). This holds out the possibility that supercompilation may become practically applicable for the same problems that are currently tackled using shortcut fusion. There are however a number of issues to address before this can come to fruition. In addition to building a practical implementation

with reasonable compile times, there is the important issue of control: supercompilation does a great deal more than deforestation, perhaps too much in some cases. In particular current supercompilation algorithms can cause a great deal of code duplication. Recent work by Jonsson and Nordlander (2011) suggests one automated approach to preventing excessive code duplication.

Ohori and Sasano (2007) take another direct approach to the problem of eliminating intermediate data structures in the presence of recursion. They argue that shortcut fusion is too limited because in practice much potentially fusible code is directly written using general recursion rather than in terms of recursion combinators. They propose another automated transformation system that can be seen as an instance of Burstall and Darlington fold-unfold calculation system. The key to the system is a transformation they call *fixed point promotion* which takes the composition of two recursive functions and produces a single recursive function. This enables many standard examples to be fused, including tail-recursive consumers with accumulating parameters such as *foldl*. Furthermore, the system is not limited to lists but works for any user-defined algebraic type. The system is not strictly more powerful than shortcut fusion however: a limitation imposed by the choice of termination condition prevents fusion in cases where a recursive function is composed with itself, as in $map\ f \circ map\ g$ for example. Overall, the system appears very promising and the next obvious step would be a full-scale implementation in a production compiler and an evaluation with more real programs.

A distinct advantage of the direct approaches to the recursion problem, such as supercompilation or the fixed point promotion system, is that since ordinary recursive definitions may be used, there is no need to prove that fusible library functions are equivalent to their usual definitions.

On the other hand, an advantage of shortcut fusion is that it can be reasonably predictable: the 'good producer / good consumer' promise that we make for stream fusion (Section 4.1.3), or that Gill (1996, Section 3.5.2) makes for *foldr/build* fusion, is relatively straightforward to understand and to use. For programmers to rely on these more general transformations it will be important to be able to make a similar clear promise about when the optimisation will be effective. It would be an interesting and somewhat ironic outcome if these more general transformations become practically effective but promises about their optimisation rely on the use of recursion combinators in the style of shortcut fusion.

A hybrid approach to the recursion problem is to use shortcut fusion but to automatically convert general recursive definitions into a form that uses the fusion combinators. Launchbury and Sheard (1995) were the first to present such

an approach. They present an algorithm to transform recursive definitions into instances of *foldr* and *build*, both for lists and generalised to other algebraic data types. Hu et al. (1996) describes an algorithm to transform recursive definitions into hylomorphisms and Onoue et al. (1997) extend this into an automated fusion system they call HYLO. We have mentioned previously (Section 4.8.1) the approach taken by Chitil (2000) to derive 'build wrappers' for ordinary recursive functions.

5.3 Optimisation and cost models

Hope and Hutton (2006) describe their solution to a challenge that is in many ways similar to the challenge we have attempted to tackle in Chapter 4. They imagine a fusion system based on hylomorphisms and consider the issue of whether the fusion transformation is actually an improvement in terms of maximum space usage. They note that elements must be consumed as they are generated otherwise fusion simply replaces a data structure with an equivalent structure of thunks[3]. For the special case of lists they suggest the use of left-hylomorphisms which are the composition of an unfold followed by a left fold. They define a hylo fusion rule and prove it correct in the context of CPOs using fixpoint induction. The remainder of the paper is dedicated to proving results about the maximum space use of functions using a technique to derive the space-use function from the definition of original function.

The crucial difference between our work and that of Hope and Hutton is that we use a cost measure that counts all the constructor allocations that are performed whereas Hope and Hutton consider only the maximum allocations required at any one time. We have to use our more detailed metric because the improvements that stream fusion make are not measurable with the latter metric. The maximum allocation metric considers an allocation followed by immediate deallocation to be neutral, but the transformation that stream fusion performs is exactly for the purpose of eliminating such allocation/deallocation pairs.

Gustavsson and Sands (2001) consider the problem of the effect of program transformations on space usage. They describe a detailed space semantics based on an abstract evaluation machine. It takes into account many operational details of how lazy functional languages are evaluated, including garbage collection. While the model is very detailed, the improvement relation that they

[3]Gill (1996, Section 3.5.6) discusses this problem in relation to the function *reverse*.

define – weak improvement ($\overset{\lhd}{\approx}$) – only captures "space improvement within a constant factor". Thus the cost measure is unable to capture the improvements of deforestation. As an example (Gustavsson and Sands, 2001, Section 6, case study 2), they give an elegant definition of the function *any* and a deforested definition

$$\begin{aligned} &any\ p = or \circ map\ p \\ &any'\ p\ [\,] \quad\ = False \\ &any'\ p\ (x : xs) = p\ x \vee any'\ p\ xs \end{aligned}$$

They sketch a proof of the proposition that $any\ p\ xs \overset{\lhd\rhd}{\approx} any'\ p\ xs$. That is, under the weak improvement relation the two definitions are equivalent, but to show that deforestation is an improvement we need a relation that distinguishes the two definitions.

In Chapter 4 we did not use a detailed formal cost model. In particular we did not use any abstract evaluation machine. Given the complexity of the transformations we are interested in, we judged that a detailed model would make the analysis too difficult. We picked an approximation that only considers the allocation of data constructors. It is interesting to note that in our context we can avoid the complexity of considering garbage collection since we are interested in the number of memory allocations performed during evaluation and not the maximum residency.

5.4 Applications of stream fusion

In addition to our own previous work (Coutts et al., 2007a,b) which introduced stream fusion and described applications to byte arrays and lists, there have been several other applications of stream fusion. Most applications have been in the context of arrays or array-like data structures.

The most high-profile application is as part of the ongoing *Data Parallel Haskell* project. In their status report, Chakravarty et al. (2007, Section 6.3) describe how the previous fusion system based on *functional array fusion* (Chakravarty and Keller, 2001) has been replaced by an improved system based on stream fusion. As we demonstrated (Coutts et al., 2007a, Section 5), stream fusion can achieve substantially higher performance than functional array fusion.

One feature that we implemented[4] for functional array fusion for byte arrays, was the ability to eliminate intermediate arrays in a pipeline of non-fusible

[4]This feature is not described in the paper but is described in the code that accompanies the paper (Coutts et al., 2007a, Reference 1).

operations by using in-place updates, thus saving unnecessary and expensive array copy operations. This feature was given up in the move from to stream fusion, but the deficiency was subsequently remedied by Leshchinskiy (2009) who described how to integrate this "recycling" feature with stream fusion for arrays.

Harper (2010) applied stream fusion in the context of an implementation of Unicode strings. Rather than being an afterthought, streams and stream fusion was integral to the prototyping and performance tuning effort. Operations on streams can be reused at many different concrete representations simply by defining new *stream* and *unstream* functions. By making use of stream fusion, such implementations can often perform as well as direct implementations but at considerably reduced effort. Thus it becomes practical to experiment with multiple concrete representations. Harper implemented and benchmarked three different string implementations using the three common Unicode encodings.

This stream-based approach is not a complete solution however, there are some operations that are cheap when implemented for concrete representation but that are expensive when implemented via streams. In particular, stream operations are unable to take advantage of sharing within a concrete representation.

Liu et al. (2009) describe a variation on the standard set of *arrow* combinators and laws that they call *causal commutative arrows*. They describe a rule-based optimisation procedure and they report that when combined with stream fusion, performance improvements that exceed two orders of magnitude are possible.

Chapter 6

Conclusion

6.1 Contributions

As this thesis is in part a product of the Programming Research Group, it seems apt to consider it in the context of the founding philosophy of the PRG.

> "It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing. One of the central aims of the Programming Research Group as a teaching and research group has been to set up an atmosphere in which this separation cannot happen."
>
> Christopher Strachey (1974)

Shortcut fusion is a research area that is particularly well-suited to making fruitful connections between theory and practice. The design and correctness of shortcut fusion systems depend on interesting theoretical properties of data and computation, while improving the performance of programs that people are interested in, is clearly of practical use.

The criteria by which we should evaluate fusion systems also combines theory and practice: the effect of a fusion system should be a correct program transformation that improves program performance in some way. A program optimisation that changes the program result is not generally useful. Neither is it useful – as a program optimisation – to have a correct program transformation that does not in fact improve performance. There are many published papers

that concern just one aspect and assume, either implicitly or explicitly, that the other aspect is straightforward or uninteresting. Both aspects are essential and both have interesting problems. In this thesis we have attempted to address both criteria.

Stream fusion is a reasonably successful and practical program optimisation. Our initial work on stream fusion (Coutts et al., 2007a,b) was primarily driven by practical concerns of performance. It neglected any serious formal treatment of correctness and concentrated on empirical evidence for the optimisation claim. In this thesis we have attempted to put the existing practical work on firmer theoretical ground by proving that the transformation is correct and by making a more universal and theoretical argument that the transformation can improve program performance.

6.1.1 Correctness

We have given a proof that the stream fusion transformation is correct in the context of CPOs. Our proof of the *stream*/*unstream* fusion rule is sufficiently general to cover stream fusion for arbitrary algebraic data types. We have also given a general criteria for the correctness of stream fusion for any abstract data type that can be viewed as a stream via suitable *stream* and *unstream* conversions. We have stated the data abstraction property that fusible functions must satisfy, and for the specific case of lists and arrays we have proved that a number of standard operations satisfy the property.

While our proof approach is not the most elegant, relying as it does on fixpoint induction, it has the practical advantage that it is valid in a semantic model that is realistic for the languages in which stream fusion is applied. The fact that our proof approach covers abstract types, such as arrays, is of practical importance because arrays are the most common application of stream fusion.

6.1.2 Optimisation

We have given a semi-formal argument that, subject to various reasonable syntactic conditions on fusible functions, stream fusion for lists is always possible and that it is strictly an improvement in terms of the number of data constructor allocations. Indeed we show that it is optimal in the sense that it eliminates all the list data constructors[1]. The optimisation argument covers not just the

[1]Strictly speaking we show only that we eliminate as many data constructor allocations as there are in the original lists. We do not track the identity of constructor allocations, just the

stream/*unstream* fusion rule itself, but the full transformation end-to-end – including the switch from standard definitions to fusible definitions. We have demonstrated that the syntactic conditions on fusible functions are reasonable by showing that many standard list functions can be defined in a way that satisfies the conditions.

We have not directly covered implementation issues, however there are several aspects of our optimisation argument that may be of use to practitioners. While our claim is just that an optimisation is possible, and we do not give a specific algorithm or prescribe a particular implementation approach, the argument that we give is constructive and may be useful as a basis for an implementation. We describe one particular sequence of transformations which closely follows those that are used in existing implementations of stream fusion. The argument necessitates giving a detailed description of how the optimisation works for this sequence of transformations. In addition, for our particular sequence of transformations, we give a clear set of conditions for what makes a good producer and a good consumer. This is directly useful as guidance on how to define fusible functions, or more generally, it may form the basis of the contract between the implementor of a stream fusion system, and those writing fusible definitions for use with the system.

6.2 Assessment of stream fusion

Stream fusion is now a reasonably mature and reasonably effective program optimisation with multiple independent implementations. It has been particularly successful in applications to array-like types.

While it was initially hoped that stream fusion could be strictly better than *foldr*/*build* fusion, and for example replace the use of *foldr*/*build* fusion in the list library in GHC, it is clear that there is no single best choice for all applications, given the state of the art in the two approaches.

The fact that we cannot yet effectively optimise uses of *concatMap* nor list comprehensions is a major limitation preventing the use of stream fusion in place of *foldr*/*build* for lists. By contrast, the *foldr*/*build* system works very effectively for list comprehensions. On the other hand, current implementations of *foldr*/*build* fusion do not effectively handle consumers with accumulating parameters such as *foldl*. In our previous empirical work we found that the main area where our implementation of stream fusion performed better than

number of allocations.

the GHC implementation of *foldr/build* fusion was in left folds. This is not a fundamental limitation of *foldr*-based fusion however; Gill (1996, Section 4.4) gives an analysis and a transformation that can turn *foldl* defined in terms of *foldr* into an efficient recursive definition.

While these practical and implementation problems persist, stream fusion remains better suited for array applications where loop-like strict left folds are the norm, while *foldr/build* remains better suited to lists where list comprehensions are important and it is tolerable to do without fusion for *foldl*.

The obvious future work in this area is to solve stream fusion's problem with *concatMap* and to implement Gill's arity analysis and then to perform new performance experiments. We conjecture that the difference in the results between the two approaches will become much less pronounced.

There remains fundamental limitations however: *foldr*-based fusion cannot express multiple consumers such as *zip* while *unfoldr*-based fusion cannot express multiple producers such as *unzip*. In this aspect it would appear that *unfoldr*-based fusion has the advantage because multiple consumers appear to be more common. Furthermore it is as yet unclear whether there are performance advantages to be gained from fusing multiple consumers. The issue is essentially one of 'buffering'. In a function such as *unzip*, the two output lists are produced in lock-step. Unless they can also be consumed in lock step – by separate consumers – then there is a mismatch and unconsumed elements from one list would have to be buffered, losing any of the allocation savings that arose from fusion.

It is interesting to observe that shortcut fusion approaches in general, and stream fusion in particular, trade off one kind of complexity for another: complexity in the implementation versus complexity in the optimisation argument. All three of the shortcut fusion systems that we have discussed consist of a sequence of local transformations, of which only one or two are classical fusion rules. The argument has been that these features make them relatively easy to implement and relatively easy to integrate into an optimising compiler because it is often possible to reuse existing infrastructure for many of the standard local transformations. The relative simplicity in implementation has made it possible to experiment easily with the design of stream fusion. On the other hand, since stream fusion requires a whole sequence of transformations, each step with its own pre and post conditions, the optimisation argument is relatively complex. For each transformation step we have had to argue that the transformation is always applicable and that it produces the desired result. It seems likely that a direct implementation, perhaps using a special state-machine representation, would be rather simpler to explain but it would be much less desirable from

the point of view of compiler engineering.

This observation on the apparent complexity tradeoff also suggests that it may be easier to give an optimisation argument for more holistic approaches such as supercompilation.

6.3 Further work

6.3.1 Theoretical

As mentioned in Section 5.1.1, it would be interesting to extend the existing Isabelle/HOLCF formal proofs to give an end-to-end proof for stream fusion as a whole. In particular, the remaining parts are: to express the abstraction property and the fusion rule in their full generality; to prove that the abstraction property is sufficient to satisfy the side condition in the fusion rule; and to show that the existing abstraction properties for individual functions are instances of the general abstraction property.

A useful extension would be to develop a search strategy for the fixpoint induction proofs of the abstraction property. In Section 3.9.8 we described some general heuristics for finding proofs, which essentially amounts to following the recursive call structure of the function in question. It seems likely that this approach could be partially or fully automated in the context of an interactive or scripted theorem prover.

In Chapter 4 we were unable to give a full formal optimisation argument. It would obviously be desirable to make the argument more formal, either to give greater confidence, to find a simpler argument or to give an argument with fewer approximations. If we follow the basic structure of the argument in Chapter 4 then there are two major parts to a formalisation:

1. the syntactic argument that the transformations are possible and that they do what we expect; and

2. the cost model.

As we discussed in Section 4.4.5, the state machine abstraction may give a method of formalising the transformation arguments. One would attempt to precisely specify an abstraction relation between the syntactic representation of stepper functions and the state machine representation. One would then aim to show that an appropriate form of state machine composition works. Finally,

it may be useful to have another relation between the state machine and the syntax of a stream consumer.

In Section 5.3 we looked at some existing work on cost models and optimisation, however it appears that no existing model is sufficiently precise to capture the improvements that deforestation achieves. In particular it is important to capture the cost of each allocation event, rather than the overall storage requirements.

6.3.2 Practical

The most significant remaining practical work on stream fusion itself is to resolve the problem with effectively optimising uses of *concatMap*. In Section 4.8.3 we suggest a number of avenues of which the most promising appears to be to use a specialised *concatMap* for exactly the situations we expect to be able to handle. The challenge with this approach is that a more sophisticated rule matching and rewriting language is required.

In Section 4.6.6 we discussed the problem of how to ensure that call pattern specialisation does actually specialise on all the stream state shapes. We suggested as a possible solution, to annotate the data types used for the state shapes, and as a special case in the call pattern specialisation heuristics, to always specialise patterns involving data constructors of such annotated data types.

Just as Chitil (2000) shows that it is often possible to derive fusible definitions from general recursive definitions for the *foldr/build* system, it seems likely that a similar technique may be possible for stream fusion. As with Chitil's technique, it would likely involve a combination of unfolding and a type analysis. If the outcome of the analysis is positive then there would follow a syntactic transformation similar to that described in Section 4.8.4.

As mentioned above, it would be desirable to implement Gill's arity analysis which should enable *foldr/build* fusion to be an effective optimisation for left folds[2].

Finally, with a solution to the *concatMap* problem and an implementation of the arity analysis, it would then be desirable to conduct new performance experiments, with a variety of fusion micro-benchmarks and real programs, to see how stream fusion and *foldr/build* fusion compare in practice.

[2] At the time of writing it appears that this transformation has been implemented in GHC, though it has yet to be tried in conjunction with *foldr/build*. `http://hackage.haskell.org/trac/ghc/ticket/4474`

Bibliography

Samson Abramsky and Achim Jung. Domain theory. In S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum, editors, *Semantic structures*, volume 3 of *Handbook of Logic in Computer Science*, pages 1–168. Oxford University Press, April 1995. ISBN 978-0-19-853762-5.

André Arnold. *Finite Transition Systems: semantics of communicating systems*. Prentice Hall, 1994. ISBN 0-13-092990-5.

E. S. Bainbridge, Peter J. Freyd, Andre Scedrov, and Philip J. Scott. Functorial polymorphism. *Theoretical Computer Science*, 70(1):35–64, January 1990. doi: 10.1016/0304-3975(90)90151-7.

Maximilian Bolingbroke and Simon Peyton Jones. Supercompilation by evaluation. In *Proceedings of the Haskell Symposium (Haskell '10)*, pages 135–146. ACM, 2010. doi: 10.1145/1863523.1863540.

Kim B. Bruce, Albert R. Meyer, and John C. Mitchell. The semantics of second-order lambda calculus. *Information and Computation*, 85(1):76–134, March 1990. doi: 10.1016/0890-5401(90)90044-I.

R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977. doi: 10.1145/321992.321996.

Manuel M. T. Chakravarty and Gabriele Keller. Functional array fusion. In Xavier Leroy, editor, *Proceedings of the International Conference on Functional Programming (ICFP '01)*, pages 205–216. ACM, 2001. doi: 10.1145/507635.507661.

Manuel M. T. Chakravarty and Gabriele Keller. An approach to fast arrays in Haskell. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming: International School (AFP '02), Revised Lectures*, volume 2638 of *Lecture Notes in Computer Science*, pages 27–58. Springer-Verlag, 2003. doi: 10.1007/978-3-540-44833-4_2.

Manuel M. T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, Gabriele Keller, and Simon Marlow. Data parallel Haskell: a status report. In *Proceedings of the workshop on Declarative Aspects of Multicore Programming (DAMP '07)*, pages 10–18. ACM, 2007. doi: 10.1145/1248648.1248652.

Olaf Chitil. Type-inference based short cut deforestation (nearly) without inlining. In Pieter Koopman and Chris Clack, editors, *Selected papers from the International Workshop on Implementation of Functional Languages (IFL '99)*, volume 1868 of *Lecture Notes in Computer Science*, pages 19–35. Springer-Verlag, 2000. doi: 10.1007/10722298_2.

Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the International Conference on Functional Programming (ICFP '00)*, pages 268–279. ACM, September 2000. doi: 10.1145/351240.351266.

Duncan Coutts. Bytestring technical report. Technical Report CL-RR-10-27, Oxford University Computing Laboratory, December 2010.

Duncan Coutts, Don Stewart, and Roman Leshchinskiy. Rewriting Haskell strings. In Michael Hanus, editor, *Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL '07)*, volume 4354 of *Lecture Notes in Computer Science*, pages 50–64. Springer-Verlag, January 2007a. doi: 10.1007/978-3-540-69611-7_3.

Duncan Coutts, Don Stewart, and Roman Leshchinskiy. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the International Conference on Functional Programming (ICFP '07)*, pages 315–326. ACM, October 2007b. doi: 10.1145/1291151.1291199.

Nils Anders Danielsson and Patrik Jansson. Chasing bottoms: A case study in program verification in the presence of partial and infinite values. In Dexter Kozen, editor, *Proceedings of the International Conference on Mathematics of Program Construction (MPC '04)*, volume 3125 of *Lecture Notes in Computer Science*, pages 85–109. Springer-Verlag, July 2004. doi: 10.1007/978-3-540-27764-4_6.

Nils Anders Danielsson, Jeremy Gibbons, John Hughes, and Patrik Jansson. Fast and loose reasoning is morally correct. In *Proceedings of the Symposium on Principles of Programming Languages (POPL '06)*, pages 206–217. ACM, January 2006. doi: 10.1145/1111037.1111056.

Olivier Danvy and Lasse R. Nielsen. Defunctionalization at work. In *Proceedings of the international conference on Principles and Practice of Declarative Programming (PPDP '01)*, pages 162–174. ACM, 2001. doi: 10.1145/773184.773202.

Kei Davis. Deforestation: Transformation of functional programs to eliminate intermediate trees. Master's thesis, Programming Research Group, Oxford University, 1987.

Martin Erwig and Simon Peyton Jones. Pattern guards and transformational patterns. In *Proceedings of the Haskell Workshop (Haskell '00)*, volume 41 of *Electronic Notes in Theoretical Computer Science*. Elsevier, August 2001. doi: 10.1016/S1571-0661(05)80540-7.

The GHC Team. The Glasgow Haskell Compiler (GHC), 2010. URL `http://www.haskell.org/ghc/`.

Jeremy Gibbons. Unfolding abstract datatypes. In *Proceedings of the International conference on Mathematics of Program Construction (MPC '08)*, volume 5133 of *Lecture Notes in Computer Science*, pages 110–133. Springer-Verlag, 2008. doi: 10.1007/978-3-540-70594-9_8.

Jeremy Gibbons and Graham Hutton. Proof methods for corecursive programs. *Fundamenta Informaticae*, 66(4):353–366, April/May 2005. ISSN 0169-2968.

Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings of the International Conference on Functional Programming (ICFP '98)*, pages 273–279. ACM, September 1998. doi: 10.1145/289423.289455.

Andrew Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, University of Glasgow, January 1996.

Andrew Gill, John Launchbury, and Simon Peyton Jones. A short cut to deforestation. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '93)*, pages 223–232. ACM, June 1993. doi: 10.1145/165180.165214.

Andy Gill and Graham Hutton. The worker/wrapper transformation. *Journal of Functional Programming*, 19(2):227–251, March 2009. doi: 10.1017/S0956796809007175.

Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and Types*. Cambridge University Press, 1989. ISBN 0-521-37181-3.

Jörgen Gustavsson and David Sands. Possibilities and limitations of call-by-need space improvement. In *Proceedings of the International Conference on Functional Programming (ICFP '01)*, pages 265–276. ACM, 2001. doi: 10.1145/507635.507667.

Tatsuya Hagino. *A Categorical Programming Language*. PhD thesis, University of Edinburgh, September 1987.

Thomas Harper. Stream fusion on Haskell Unicode strings. In Marco Morazán and Sven-Bodo Scholz, editors, *Revised selected papers from the International Symposium on Implementation and Application of Functional Languages (IFL '09)*, volume 6041 of *Lecture Notes in Computer Science*, pages 125–140. Springer-Verlag, 2010. doi: 10.1007/978-3-642-16478-1_8.

Ryu Hasegawa. Categorical data types in parametric polymorphism. *Mathematical Structures in Computer Science*, 4(1):71–109, 1991. doi: 10.1017/S0960129500000372.

Ralf Hinze, Tom Harper, and Daniel W. H. James. Theory and practice of fusion. In Jurriaan Hage, editor, *Revised selected papers from the International Symposium on Implementation and Application of Functional Languages (IFL '10)*, volume 6647 of *Lecture Notes in Computer Science*. Springer-Verlag, 2011. ISBN 978-3-642-24275-5.

Catherine Hope and Graham Hutton. Compact fusion. In *Proceedings of the Workshop on Mathematically Structured Functional Programming (MSFP '06)*, eWiC. BCS, July 2006.

Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings of the International Conference on Functional Programming (ICFP '96)*, pages 73–82. ACM, 1996. doi: 10.1145/232627.232637.

Brian Huffman. Stream fusion. In Gerwin Klein, Tobias Nipkow, and Lawrence Paulson, editors, *The Archive of Formal Proofs*. `http://afp.sf.net/entries/Stream-Fusion.shtml`, April 2009. Formal proof development.

John Hughes. Why functional programming matters. *Computer Journal*, 32(2):98–107, April 1989. doi: 10.1093/comjnl/32.2.98.

Graham Hutton. A tutorial on the universality and expressiveness of fold. *Journal of Functional Programming*, 9(4):355–372, July 1999. doi: 10.1017/S0956796899003500.

Patricia Johann. Short cut fusion is correct. *Journal of Functional Programming*, 13 (4):797–814, July 2003. doi: 10.1017/S0956796802004409.

Patricia Johann and Janis Voigtländer. The impact of *seq* on free theorems-based program transformations. *Fundamenta Informaticae*, 69(1–2):63–102, July 2006. ISSN 0169-2968.

Peter A. Jonsson and Johan Nordlander. Taming code explosion in supercompilation. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation (PEPM '11)*. ACM, January 2011. Accepted for publication.

John Launchbury and Tim Sheard. Warm fusion: deriving build-catas from recursive definitions. In *Proceedings of the conference on Functional Programming Languages and Computer Architecture (FPCA '95)*, pages 314–323. ACM, 1995. doi: 10.1145/224164.224223.

Roman Leshchinskiy. Recycle your arrays! In Andy Gill and Terrance Swift, editors, *Proceedings of the International Symposium on Practical Aspects of Declarative Languages (PADL '09)*, volume 5418 of *Lecture Notes in Computer Science*, pages 209–223. Springer-Verlag, 2009. doi: 10.1007/978-3-540-92995-6_15.

Hai Liu, Eric Cheng, and Paul Hudak. Causal commutative arrows and their optimization. In *Proceedings of the International Conference on Functional Programming (ICFP '09)*, pages 35–46. ACM, 2009. doi: 10.1145/1596550.1596559.

Simon Marlow and Philip Wadler. Deforestation for higher-order functions. In *Proceedings of the Workshop on Functional Programming*, Workshops in Computing, pages 154–165. Springer-Verlag, 1993. ISBN 3-540-19820-2.

Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In John Hughes, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '91)*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, August 1991. doi: 10.1007/3540543961_7.

Neil Mitchell. Rethinking supercompilation. In *Proceedings of the International Conference on Functional Programming (ICFP '10)*, pages 309–320. ACM, 2010. doi: 10.1145/1863543.1863588.

Neil Mitchell and Colin Runciman. A supercompiler for core Haskell. In Olaf Chitil, Zoltán Horváth, and Viktória Zsók, editors, *Revised selected papers from the International Symposium on Implementation and Application of Functional Languages (IFL '07)*, volume 5083 of *Lecture Notes in Computer Science*, pages 147–164. Springer-Verlag, 2008. doi: 10.1007/978-3-540-85373-2_9.

Atsushi Ohori and Isao Sasano. Lightweight fusion by fixed point promotion. In *Proceedings of the symposium on Principles of Programming Languages (POPL '07)*, pages 143–154. ACM, 2007. doi: 10.1145/1190216.1190241.

Y. Onoue, Z. Hu, M. Takeichi, and H. Iwasaki. A calculational fusion system HYLO. In *Proceedings of the IFIP TC2 WG 2.1 international workshop on Algorithmic Languages and Calculi*, pages 76–106. Chapman & Hall, 1997. ISBN 0-412-82050-1.

Simon Peyton Jones. Call-pattern specialisation for Haskell programs. In *Proceedings of the International Conference on Functional Programming (ICFP '07)*, pages 327–337. ACM, October 2007. doi: 10.1145/1291151.1291200.

Simon Peyton Jones and André Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1–3):3–47, September 1998. doi: 10.1016/S0167-6423(97)00029-4.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In Ralf Hinze, editor, *Preliminary Proceedings of the Haskell Workshop (HW '01)*, pages 203–233, September 2001. Utrecht University technical report UU-CS-2001-23.

Simon Peyton Jones et al. *The Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, May 2003. ISBN 0-521-82614-4. URL `http://www.haskell.org/definition/`.

Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. In Marc Bezem and Jan Groote, editors, *International Conference on Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*, pages 361–375. Springer-Verlag, 1993. doi: 10.1007/BFb0037118.

Franz Regensburger. HOLCF: Higher order logic of computable functions. In *Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 293–307. Springer-Verlag, 1995. doi: 10.1007/3-540-60275-5_72.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM annual conference (ACM '72)*, volume 2, pages 717–740. ACM, 1972. doi: 10.1145/800194.805852.

Giuseppe Rosolini and Alex Simpson. Using synthetic domain theory to prove operational properties of a polymorphic programming language based on strictness. Manuscript, 2004. URL `http://homepages.inf.ed.ac.uk/als/Research/usdt.pdf`.

Colin Runciman, Matthew Naylor, and Fredrik Lindblad. SmallCheck and Lazy SmallCheck: automatic exhaustive testing for small values. In *Proceedings of the Haskell Symposium (Haskell '08)*, pages 37–48. ACM, September 2008. doi: 10.1145/1411286.1411292.

André Santos. *Compilation by Transformation in Non-Strict Functional Languages*. PhD thesis, University of Glasgow, July 1995.

Daniel Seidel and Janis Voigtländer. Automatically generating counterexamples to naive free theorems. In Matthias Blume and German Vidal, editors, *Proceedings of the International Symposium on Functional and Logic Programming (FLOPS '08)*, volume 6009 of *Lecture Notes in Computer Science*, pages 175–190. Springer-Verlag, April 2010. doi: 10.1007/978-3-642-12251-4_14.

Morten Sørensen, Robert Glück, and Neil Jones. Towards unifying partial evaluation, deforestation, supercompilation, and GPC. In Donald Sannella, editor, *Proceedings of the European Symposium on Programming: Programming Languages and Systems (ESOP '94)*, volume 788, pages 485–500. Springer-Verlag, 1994. doi: 10.1007/3-540-57880-3_32.

Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *Proceedings of the International Conference on Functional Programming (ICFP '02)*, pages 124–132. ACM, 2002. doi: 10.1145/581478.581491.

Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '95)*, pages 306–313. ACM, 1995. doi: 10.1145/224164.224221.

Valentin Turchin. The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3):292–325, June 1986. doi: 10.1145/5956.5957.

Janis Voigtländer. Proving correctness via free theorems: The case of the *destroy/build*-rule. In Robert Glück and Oege de Moor, editors, *Proceedings of the Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '08)*, pages 13–20. ACM, January 2008a. doi: 10.1145/1328408.1328412.

Janis Voigtländer. Semantics and pragmatics of new shortcut fusion rules. In Jacques Garrigue and Manuel Hermenegildo, editors, *Proceedings of the International Symposium on Functional and Logic Programming (FLOPS '08)*, volume 4989 of *Lecture Notes in Computer Science*, pages 163–179. Springer-Verlag, April 2008b. doi: 10.1007/978-3-540-78969-7_13.

Philip Wadler. Theorems for free! In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*, pages 347–359. ACM, 1989. doi: 10.1145/99370.99404.

Philip Wadler. Recursive types for free! Unpublished manuscript, July 1990a. URL `http://homepages.inf.ed.ac.uk/wadler/papers/free-rectypes/free-rectypes.txt`.

Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, January 1990b. doi: 10.1016/0304-3975(90)90147-A.

G. C. Wraith. A note on categorical datatypes. In *Category Theory and Computer Science*, volume 389 of *Lecture Notes in Computer Science*, pages 118–127. Springer-Verlag, 1989. doi: 10.1007/BFb0018348.