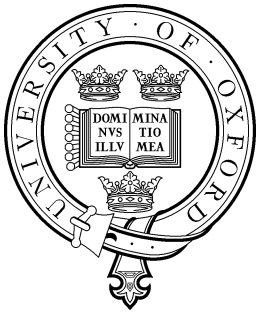
Abstract Interpretation of Domain-Specific Embedded Languages

Kevin Backhouse
Lincoln College

D.Phil. Thesis
Trinity Term, 2002



To Meghan

Abstract Interpretation of Domain-Specific Embedded Languages

Kevin Backhouse, Lincoln College

D.Phil. Thesis
Trinity Term, 2002

Abstract

A domain-specific embedded language (DSEL) is a domain-specific programming language with no concrete syntax of its own. Defined as a set of combinators encapsulated in a module, it borrows the syntax and tools (such as type-checkers and compilers) of its host language; hence it is economical to design, introduce, and maintain. Unfortunately, this economy is counterbalanced by a lack of room for growth. DSELs cannot match sophisticated domain-specific languages that offer tools for domain-specific error-checking and optimisation. These tools are usually based on syntactic analyses, so they do not work on DSELs.

Abstract interpretation is a technique ideally suited to the analysis of DSELs, due to its semantic, rather than syntactic, approach. It is based upon the observation that analysing a program is equivalent to evaluating it over an abstract semantic domain. The mathematical properties of the abstract domain are such that evaluation reduces to solving a mutually recursive set of equations. This dissertation shows how abstract interpretation can be applied to a DSEL by replacing it with an abstract implementation of the same interface; evaluating a program with the abstract implementation yields an analysis result, rather than an executable.

The abstract interpretation of DSELs provides a foundation upon which to build sophisticated error-checking and optimisation tools. This is illustrated with three examples: an alphabet analyser for CSP, an ambiguity test for parser combinators, and a definedness test for attribute grammars. Of these, the ambiguity test for parser combinators is probably the most important example, due to the prominence of parser combinators and their rather conspicuous lack of support for the well-known LL(k) test.

In this dissertation, DSELs and their signatures are encoded using the polymorphic lambda calculus. This allows the correctness of the abstract interpretation of DSELs to be proved using the parametricity theorem: safety is derived for free from the polymorphic type of a program. Crucially, parametricity also solves a problem commonly encountered by other analysis methods: it ensures the correctness of the approach in the presence of higher-order functions.

Contents

Acknowledgements

I would like to thank my supervisor, Bernard Sufrin. He gave me guidance when I needed it, but also always encouraged me to explore my own ideas. Bernard believes that a D.Phil is about learning to do research; he gave me the freedom to do so, even though I wandered so far from my original project description that this dissertation has almost nothing in common with it. I am very grateful for his trust.

I would like to thank the members of the Programming Tools Group and Oege de Moor in particular. As leader of the group he devoted huge amounts of time and effort, helping us with our work and involving us in his own. I would also like to thank Eric Van Wyk who, as someone who had recently finished his own doctorate, was an excellent source of advice. Ganesh Sittampalam and Iván Sanabria-Piretti, who started their doctorates at the same time and with whom I shared an office throughout, were great companions. More generally, the Oxford University Computing Laboratory has been a fantastic place to study; I would recommend it to anyone who is interested in computer science. Had I gone straight to industry, I might still be blissfully ignorant of many of the things that I have learned, but the computing lab has opened up a new world for me.

This dissertation was sponsored by Microsoft Research's Intentional Programming Group, who funded a project at Oxford to design *a Meta-Language for Intentional Programming*. I would like to thank Microsoft for giving me the opportunity to work on such an interesting project and also for inviting me to visit the company as a summer intern in 1999.

I would like to thank my examiners, Mark Jones and Richard Bird, for their helpful comments and suggestions, which have greatly improved this dissertation.

I am very grateful to my family for their help and advice and for encouraging me to study for a doctorate. My father deserves a special mention, because, as a professor of computer science, he was also able to help me academically. We did some joint work, which led to the material in Chapter 5, and wrote a paper on it together: something I hope we might do again in the future.

I met Meghan, my wife, approximately six months into my course. A doctoral student herself, she has been there with me through thick and thin. I hope that I can help her through writing up in the same way that she has helped me. I have few insights into ethnography to offer, but I can do a bit of IT support!

I have many good memories from the time I spent with the Oxford University Cycling Club and Wolfson College Boat Club. I have had a lot of fun at Oxford and a lot of it has been with these two clubs. Thanks to all my friends in Oxford!

Finally, I would like to thank Thomas Harte, who was my project supervisor during my Diploma in Computer Science at Cambridge. He reawakened my interested in doing a doctorate and encouraged me to look for a position. He also put a lot of effort into improving my written English. If this dissertation is bad, it was much worse before!

Statement of Authorship

I confirm that this dissertation is wholly my own work, apart from the material in Chapter 5, which was developed in collaboration with Professor Roland Backhouse, of Nottingham University. I suggested that proofs of the safety of abstract interpretations, that I had done earlier, could be simplified and shortened by the use of parametricity properties, and sought his advice on how to proceed. From subsequent joint work, it emerged that the connection between safety properties and parametricity was already known, and could be obtained by specialising Abramsky's uniformization lemma. The details were developed jointly in a series of email exchanges and meetings, spread over a period of several months.

Chapter 1

Introduction

A *domain-specific language* (DSL) is a programming language that is designed to be used in a particular problem domain. For example, HTML [84] is a domain specific language for writing web documents. Other examples are the Make utility [29] and the parser generator YACC [47]. Domain specific languages stand in contrast to *generic* programming languages such as C [52] and Java [8]. Any program written in a DSL could also be written in a generic language, but a well-designed DSL should offer a number of advantages:

- Clear notation specifically designed for the problem domain. Many DSLs are *declarative*, which can make programs much shorter and easier to understand than their generic counterparts.
- Domain specific error-checking and error-messages.
- Domain specific optimisations.

A particularly well-designed DSL might even be useable by a non-programmer who is familiar with the problem domain. A generic language, however, would not afford the same possibilities. An illustration of this might be a web designer who is familiar with HTML, but not Java.

A comprehensive survey of the literature on DSLs has been conducted by van Deursen, *et al.* [23]. They confirm the advantages listed above, but also report a number of disadvantages:

- Designing, implementing and maintaining a DSL can be costly.
- Educating DSL users may be costly.
- Programs written in a DSL may be less efficient than hand-written code.
- Finding an appropriate scope for the DSL, such that it is expressive but not bloated, can be difficult.

Initially, a DSL might be implemented with a simple preprocessor, but a solution with better error-reporting and optimisation is often required as the language becomes more

widely used. So the maintenance of a DSL could potentially involve the development of a complex compiler or interpreter. There is also a danger that the scope of a DSL will gradually expand as users request more features. This is because the division between generic and domain specific languages is often fuzzy. For example, Microsoft's Visual Basic [107] could be thought of as a domain specific language for scripting Windows components, but it is powerful enough to be considered a generic language. In fact many languages are launched as DSLs, but eventually evolve into generic languages. Witness Cobol, Fortran and Lisp, which were originally DSLs for business processing, numeric computation and symbol processing, respectively. The danger of escalating maintenance costs associated with language evolution is a strong reason against introducing new DSLs.

The alternative to a DSL is a subroutine library written in an existing generic language. This option is much cheaper than constructing a DSL, because it does not involve the development and maintenance of a compiler. There is also less danger of the library evolving beyond its intended scope, because other subroutine libraries can cater for other needs. In fact, van Deursen, *et al.* suggest that the first phase of implementing a DSL should be to construct a subroutine library. The second phase is to implement a translator from DSL programs to sequences of library calls. The main disadvantage of a subroutine library is that program structure is dictated by the semantics of the host language. This often means that programs cannot be written in a declarative style.

The concept of a *domain-specific embedded language* (DSEL) was identified by Hudak [43, 44]. A DSEL is just a subroutine library, but written in a functional language such as Haskell [78]. Hudak argues that in a language such as Haskell, domain specific programs *can* be written in a declarative style. He believes that the crucial features of Haskell that make this possible are higher-order functions, lazy evaluation, polymorphism and type classes.[1] Some convincing examples of DSELs are Swierstra and Duponcheel's parser combinators [96], Elliott and Hudak's animation library [25, 26] and a library for defining financial contracts, invented by Peyton Jones, *et al.* [80].

DSELs are a promising new method of constructing domain specific languages. They allow the construction of a *declarative* language, without the overhead of maintaining a compiler. However, two advantages of DSLs that are currently absent in DSELs are domain specific optimisations and error-checking. Neither advantage can be achieved without the ability to perform domain specific *program analyses* on DSELs. Domain specific program analyses are required for error-checking and for checking the side-conditions of optimisations. This dissertation contributes a methodology for performing program analyses on DSELs. The methodology is based on *abstract interpretation*: a technique invented by Cousot and Cousot [21]. An important aspect of the methodology, that ties in with the philosophy of DSELs, is that it does not require the construction of a complex domain specific tool for performing the analysis. The analysis can be performed by a generic facility, provided by the host

[1] Hudak includes type classes in this list for their syntactic benefits, but they do not make a semantic contribution. Type classes are not used in this dissertation.

language.

1.1 Overview

The methodology proposed in this dissertation for analysing DSELs relies on several areas of related work, which are reviewed in Chapters 1 to 4. The main theoretical result is presented in Chapter 5 and each of Chapters 6, 7, and 8 illustrates its use on a practical example. Chapter 9 concludes.

This first chapter gives a brief introduction to DSELs and explains why they are more difficult to analyse and optimise than traditional pre-processor based DSLs. The solution proposed by this dissertation is then sketched. The difficulty of analysing and optimising DSELs may appear to be a strong argument against their use, so some alternative frameworks are surveyed. In contrast to DSELs, many of these frameworks allow domain specific optimisations to be added relatively easily, but their lack of a semantic model leads to other problems that do not affect DSELs.

Chapter 2 introduces the programming notation that is used throughout this dissertation. The notation is based loosely on Haskell [78], but it does not follow Haskell's semantics because it is intended primarily as a mathematical notation, rather than as a real programming language. Instead, the much simpler semantics of a typed lambda calculus are adopted. In order to encode DSELs and their type signatures, a module system invented by Jones [50] is used. It is based on standard polymorphic types, which makes it easy to apply the main theoretical result to DSELs. The chapter contains a short tutorial on Jones's module system.

Chapter 3 discusses abstract interpretation and the semantic foundations upon which it is based, such as domain theory, least fixed points, and Galois connections. Chapter 4 reviews Reynolds' abstraction theorem [86], or *theorems for free* as Wadler [103] calls it.

The main theoretical result is obtained in Chapter 5 by combining *theorems for free* with abstract interpretation to create *abstract interpretation for free*. The first example of its use is an alphabet analysis for CSP, given in Chapter 6. A more substantial example is an ambiguity test for parser combinators, which is presented in Chapter 7. Abstract interpretation for free can also be applied to traditional examples of abstract interpretation, as illustrated by the development of a definedness test for attribute grammars in Chapter 8. Chapter 9 discusses possibilities for future applications of the methodology and areas for further research.

1.2 Domain Specific Embedded Languages

The fundamental idea underlying DSELs — articulated in 1966 by Landin [60] — is that a language's aptitude for a particular class of tasks is determined by its *primitives*, not its *structuring mechanisms*. Unfortunately, language designers are often distracted by the latter and confuse the boundary between the two. Landin argued that language design should be conducted within a powerful framework of generic

structuring mechanisms, so that the choice of primitives takes priority. The generic structuring mechanisms form a *family of languages* that can be adapted to different application domains. Landin proposed a system called ISWIM based on the lambda calculus to achieve this. The influence of ISWIM is still visible today in languages such as Haskell [78].

At the time of Landin's paper, it was not fully understood how to use the pure lambda calculus as a programming language due to its lack of imperative features. In the discussion at the end of Landin's paper, Strachey comments that: "we don't know how to do everything in pure declarative languages". Hence, Landin's ISWIM included imperative features, although he promoted the use of a pure subset of the language. Ironically, Landin's own ideas have helped to solve the problem: Wadler [104, 105, 106] has discovered that Moggi's *monads* [69, 70, 71] can be used to encode impure features such as input-output as *primitives*; Spivey [91] independently developed a similar encoding of exceptions. The use of these ideas has been pioneered in the functional language Haskell [78]: despite being a pure, strongly typed language it has extensive libraries for input-output and foreign function interfacing. This progress implies that most of the barriers to using the pure lambda calculus as a programming language are now gone. The pure lambda calculus is used for all the examples in this dissertation, although some syntactic sugar, based on Haskell's syntax, is introduced in Chapter 2 to improve the presentation.

If Landin's advice is to be followed, then a truly well-designed DSEL should have a very compact, yet extremely powerful, core of primitives. This core should be independent of any abstractions that might be built on top of it. DSEL design is therefore a perfectionist's art and can sometimes take years of tweaking. This is illustrated by the improvements that are still being made to *parser combinators* — most recently by Swierstra [94] and Leijen [64]. Parser combinators are a classic example of a DSEL, with a core consisting of just four combinators:[2]

$$
\begin{array}{l}
\mathit{empty} :: g, \\
\mathit{token} :: g \leftarrow \tau, \\
\mathit{choice} :: g \leftarrow (g,g), \\
\mathit{cat} :: g \leftarrow (g,g).
\end{array}
$$

The combinators *empty* and *token* are primitive parsers for empty and singleton strings, respectively. These elements can be combined to form more complicated parsers using *cat* (for sequential composition) and *choice* (for combining two alternatives). An arbitrary parser can be built in this way, but many abstractions can be introduced to facilitate the process. For example, *commaSep* is a useful abstraction for parsing comma separated lists:

$$
\begin{array}{lcl}
\mathit{commaSep} & :: & g \leftarrow g, \\
\mathit{commaSep}\ p & = & \mathit{choice}\ \ (p,\ \ \mathit{cat}\ \ (p,\ \ \mathit{cat}\ \ (\mathit{token}\ \mathit{Comma},\ \ \mathit{commaSep}\ p))).
\end{array}
$$

[2]Throughout this dissertation, function types are written backwards with $\leftarrow$, rather than forwards with $\rightarrow$, primarily because this is more natural when working with relations. (See §4.1.)

This is a function of type $g \leftarrow g$ because it takes a parser and produces a new parser, for comma separated lists. For instance, if *param* is a parser for procedure parameters, then *commaSep param* is a parser for comma-separated lists of procedure parameters. An important feature of *commaSep* is that it is defined in terms of the primitive combinators. Therefore it does not need to be incorporated into the core. Many sophisticated abstractions can be developed in this way. For example, Leijen's *Parsec* library [64] contains a useful abstraction for constructing expression parsers from a list of operators annotated with their precedence.

1.3 Analysing DSELs

DSELs do not satisfy many of the assumptions that are made by conventional program analyses. To illustrate, consider the ambiguity analysis performed by the parser generator YACC [47], which makes three crucial assumptions:

1. The *syntax* of the input allows YACC to analyse the structure of the grammar.

2. Higher-order abstractions do not exist. For example, it is not possible to define a new generic operator for parsing comma-separated lists.

3. YACC can apply a *global* program analysis, because it has access to the entire grammar.

None of these assumptions can be made in the embedded setting of a DSEL. Firstly it is not possible to apply a syntactic analysis, because the DSEL is embedded in the language and does not have its own distinctive syntax. Secondly, higher-order abstractions are unavoidable in DSELs, because they are integral to the tradition of separating the primitives from the structuring mechanisms. Thirdly, global program analysers for full scale programming languages are rare and inevitably limited in scope, due to the computational expense of analysing an entire program. So a scheme for analysing DSELs must abstain from many prevalent analysis techniques. It must not be based on syntax or a closed world assumption and it must be compatible with higher-order abstractions. A scheme that satisfies these criteria is the main contribution of this dissertation. It is based on *abstract interpretation*, a technique invented by Cousot and Cousot [21], which is ideally suited to the analysis of DSELs, because it is not based on syntax. Reynolds's *parametricity* result [86] is used, in Chapter 5, to prove that the scheme is valid in the presence of higher-order abstractions. The scheme is *compositional* in nature, so there is no need for a global program analyser, but this also means, unfortunately, that it cannot express non-compositional analyses. The only solution to this problem is to *derive* analyses that *are* compositional. One of the goals of Chapters 6, 7, and 8 is to illustrate some of the techniques that can be used to achieve this.

A rough introduction to the scheme follows below.

1.3.1 Abstract Interpretation and Abstract Datatypes

Cousot and Cousot's observation is that analysing a program can be viewed as *evaluating* it over an abstract domain. A program analyser can therefore be thought of as an *abstract interpreter* for the language. In the context of DSELs, this idea corresponds to replacing the core of primitives with *abstract* alternatives. Evaluating the program with these abstract primitives yields an analysis result.

The idea of replacing the core primitives is a variation on the theme of *abstract datatypes*. It is well known that different (but semantically equivalent) implementations of an abstract datatype can be swapped without changing the semantics of the program. The classical example is an interface that defines the datatype *Stack* and provides *push* and *pop* operations. One implementation of this interface might represent stacks as linked lists and another might represent them as arrays. These implementations are semantically equivalent, so the choice does not affect the semantics of programs that use the module.

The core primitives of a DSEL typically implement operations over an abstract datatype. For example, parser combinators are operations over the abstract type of a grammar. This allows multiple implementations of the DSEL to be provided. The approach in this dissertation is to give two implementations: the first is the *concrete* implementation of the DSEL for normal use and the second is an *abstract* version which is used for program analysis. In contrast to the stack example above, the semantics of the concrete and abstract implementations are *not* equivalent, because the abstract module is used to compute *static* properties of the program. This means that the semantics of a program using the DSEL are changed if the concrete module is swapped for the abstract.

In any form of program analysis, it is important to prove that the analysis never produces incorrect results. This property is often known as *safety*. In Chapter 5, the safety of the abstract interpretation of DSELs is proved with Reynolds's *parametricity* result [86]. This result, which has been dubbed *theorems for free* by Wadler [103], states that every polymorphic program satisfies a property that can be derived from its type. It can be applied to DSELs due to Jones's observation [49] that abstract datatypes can be encoded as polymorphic type variables. An important advantage of using parametricity is that it solves the problem of working with higher-order abstractions: the parametricity property of a program depends only on its type, so it does not matter if higher-order abstractions were used in its definition. Jones's encoding of abstract datatypes is explained in Chapter 2 and introductions to abstract interpretation and parametricity are given in Chapters 3 and 4. The safety result is proved in Chapter 5.

The abstract interpretation of a DSEL is illustrated by the commuting diagram in Figure 1.1. P is a program that is parameterised by a DSEL with interface I. C and A are the concrete and abstract implementations of I, respectively. Their semantics are related by the function *abs*. P_C and P_A are the two instantiations of P. The semantic relationship between P_C and P_A is given by a function called P_{abs}. The analysis is safe if the diagram commutes: this indicates that the analysis correctly predicts the behaviour of the program.

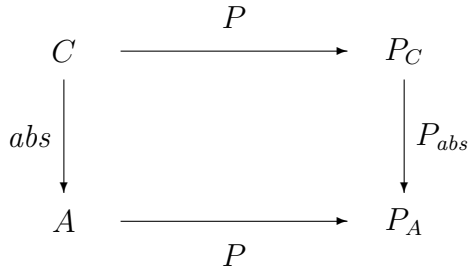


Figure 1.1: Two Instantiations of a Program P

1.3.2 Phantom Types

The importance of types in the design of DSELs is well-known and some domain-specific errors can be caught with types alone. An important example of this is Launchbury and Peyton Jones's use of types to ensure the safety of stateful computations in Haskell [62]. Their library allows programmers to create and modify mutable variables, while maintaining referential transparency. The library provides primitive operations over the abstract datatypes ST of computations and $MutVar$ of mutable variables:

$$
\begin{array}{ll}
returnST & :: \forall s,\alpha.\ ST\ s\ \alpha \leftarrow \alpha, \\
thenST & :: \forall s,\alpha.\ ST\ s\ \beta \leftarrow (ST\ s\ \beta \leftarrow \alpha) \leftarrow ST\ s\ \alpha, \\
newVar & :: \forall s,\alpha.\ ST\ (MutVar\ s\ \alpha) \leftarrow \alpha, \\
readVar & :: \forall s,\alpha.\ ST\ s\ \alpha \leftarrow MutVar\ s\ \alpha, \\
writeVar & :: \forall s,\alpha.\ ST\ s \leftarrow \alpha \leftarrow MutVar\ s\ \alpha, \\
runST & :: \forall \alpha.\ \alpha \leftarrow (\forall s.\ ST\ s\ \alpha).
\end{array}
$$

The functions $returnST$ and $thenST$ are the standard monadic combinators for constructing computations. Mutable variables are manipulated with $newVar$, $readVar$ and $writeVar$. The only purpose of the polymorphic type variable s, which is threaded through each of these operations, is to ensure safety. To evaluate a computation, $runST$ must be used. The nested quantification over s in the type of $runST$ prevents the computation from interacting with the outside world, so referential transparency is maintained.

Leijen and Meijer [63] refer to type variables such as s, that have no purpose other than ensuring safety, as *phantom types*. Their use can be a valuable technique in the design of DSELs. The main drawback of phantom types is that their expressiveness is limited by the expressiveness of the type system. For instance, Leijen and Meijer's example of phantom types requires an experimental type system feature for extensible records, invented by Gaster and Jones [33]. It would not be possible to encode their example in a standard ML-style type system. Hence, the ideas of this dissertation complement phantom types by offering an alternative when the type system is not expressive enough.

1.3.3 Active Libraries

This dissertation is closely related to a relatively new research field called *Active Libraries*. Veldhuizen and Gannon [102] identify this research category and give examples of the work that it encompasses. Most of their examples are in the area of scientific computing, where programmers wish to work with high level concepts such as matrices and Fast Fourier Transforms, but without any loss of performance. The strong performance criteria in scientific computing rule out the use of object oriented libraries, which are expressive but hard to optimise. The alternative of adding new features to the host language is not usually viable, because the proposed extensions are too specialised. This has led researchers to develop solutions based on macro-processing or C++ templates. These solutions are often ad-hoc and specific to one problem, so Veldhuizen and Gannon propose that there is a need for generic facilities for making libraries active. Such facilities would allow the library designer to specify library-specific optimisations — possibly dependent on information obtained by program analysis.

This dissertation is relevant to active libraries, because it proposes a generic way for library designers to specify program analyses. In combination with a notation for specifying rewrite rules this analysis information could be used to specify optimisations. The implementers of the Glasgow Haskell Compiler [34, pages 185–190] are currently experimenting with such a feature for specifying rewrite rules in Haskell.

1.4 Alternatives to Domain Specific Embedded Languages

Being able to develop domain specific languages by adapting an existing language is an enticing goal, so numerous ways of achieving it have been proposed. The issues that affect these approaches are often quite different to those that affect DSELs. The implementation of domain specific error checking and optimisations tends to be just an engineering problem. However, the semantic soundness that one might take for granted in DSELs can be difficult or impossible to verify.

Extending an existing compiler is possibly the simplest and most obvious route to adapting a language for domain specific applications, so this option is discussed first. The second topic is Intentional Programming (IP): a concept invented by C. Simonyi at Microsoft Research [90]. IP tries to address the limitations of extensible compilers by providing a highly structured compiler framework. The idea is that if language extensions (intentions) are designed correctly, then they can be safely added to the framework. Attribute grammar researchers have also tackled the problem of creating DSLs by adapting existing languages. They have proposed attribute grammar features to facilitate this. The theoretical work on using monads and monad transformers to modularise semantics is also of great interest here. Unfortunately, some of the results of this work merely emphasise the fundamental difficulties involved in trying to extend a programming language.

1.4.1 Extensible Compilers

As discussed in the introduction, the advantage of a DSEL is that it inherits the features of the host language. An alternative approach is to build an *extensible* compiler for the host language. A DSL is then created by writing a compiler module and plugging it in. This solution is lighter weight than creating a compiler from scratch, because the compiler for the host language is inherited. An advantage of this approach is that domain specific optimisations and error checking can be implemented by adding the necessary code to the DSL compiler module. There are also fewer limitations on the semantics of the DSL. For example, an imperative language could be extended with a declarative language for creating graphical user interfaces. Such a DSL might be difficult to write as a subroutine library.

When a DSL is designed as a subroutine library or DSEL, it is easy to take the module system of the host language for granted. Assuming that the host language has a well-designed module system, the DSEL will benefit from the following properties:

1. *Interface.* The DSEL has a published interface that specifies how it may be used.

2. *Encapsulation.* The DSEL is a *black box* that can only be accessed via its interface. This prevents DSELs from interfering with each other in unexpected ways.

3. *Implementation independence.* Two (semantically equivalent) implementations of the same interface can be swapped.

These properties have a number of important consequences. Firstly, a user of a DSEL only needs to understand its interface, not its implementation. Hence, the DSEL can be developed by an independent team. Secondly, the implementation of the DSEL can be updated without breaking existing programs. Thirdly, several DSELs can be used simultaneously, even if they were implemented by independent teams. Encapsulation prevents the DSELs from interfering with each other in unexpected ways.

In an extensible compiler, the properties listed above cannot be taken for granted. Consider Engler's [27] Magik system: Magik is a customised version of the lcc C compiler [31], that allows DSL designers to extend C by writing routines that modify the internal abstract syntax tree during the compilation process. Magik is probably not intended to be used with DSLs designed by independent teams, but such use would lead to some difficult questions. Which DSL should be allowed to modify the abstract syntax tree first? Will the modifications implemented by the first DSL affect the behaviour of the second DSL? These questions cannot be answered without a detailed understanding of the implementations of the two DSLs, so the DSLs cannot be designed by independent teams.

1.4.2 Intentional Programming

Intentional Programming (IP) is a concept invented by C. Simonyi at Microsoft Research [90]. IP attempts to tackle the encapsulation problems of extensible compilers

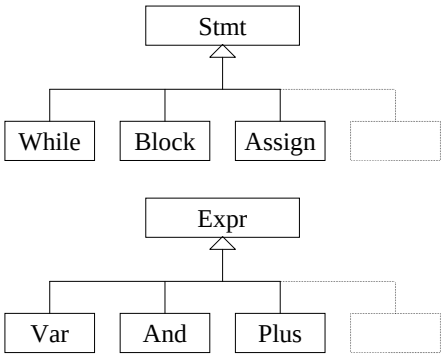


Figure 1.2: An object oriented representation of a grammar (depicted as a UML class diagram).

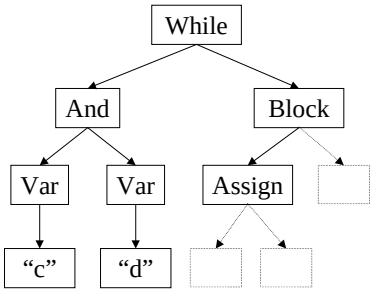


Figure 1.3: An object oriented representation of an AST

by providing a robust compiler framework. Within this framework, new language features are added by adding *intentions* to the system. An intention is roughly equivalent to a production in the grammar of the language. It is an encapsulated entity that defines both the syntax and the semantics of an element of the language. For simplicity of presentation, IP is explained here in terms of object oriented programming. This is not entirely accurate, because IP differs from OO in a number of ways, but it allows the use of familiar OO terminology.

In object oriented programming, an abstract syntax tree is typically represented by a tree of objects. The class hierarchy of the objects mirrors the grammar of the language. (Gamma, *et al.* [32, pages 243–255] explain this idiom in their discussion of the Interpreter design pattern.) For example, consider the following grammar:

$$\begin{array}{lcl} \mathit{Stmt} & ::= & \mathit{While} \mid \mathit{Block} \mid \mathit{Assign} \mid \ldots \\ \mathit{Expr} & ::= & \mathit{Var} \mid \mathit{And} \mid \mathit{Plus} \mid \ldots \end{array}$$

The object oriented class hierarchy for this grammar is illustrated by the UML class diagram in Figure 1.2. Stmt and Expr are interfaces implemented by the classes While, Block, Var, *etc.* An AST built using these classes is illustrated in Figure 1.3. Using OO terminology, an intention is a class and an intention instance is an object in the AST. In the IP framework, intention instances always have a back-pointer to their

parent, so that information can be easily passed up and down the tree. Compilation proceeds by calling the *code* method on the object at the root of the tree. This triggers a series of method calls throughout the tree and intermediate code is produced.

The key to developing an extensible compiler in IP is to establish a *framework*, consisting of a set of methods that all intentions should support and a protocol for their use. For example, a simple protocol might require that every intention defines two methods:

- A *code* method for computing intermediate code. This method recursively calls the children of the intention.
- An *env* method for computing the environment. This method recursively calls the parent of the intention.

Provided that an intention follows this protocol, it can be safely added to the language. Unfortunately, it is difficult to add new language features without adding new methods to the framework. Consider for example an imperative language with *while* statements but without *break* or *continue* statements. These new commands could be added as new intentions and implemented by reducing them to *goto* statements. For example, the code

```
while k < n do {
  if a[k] == 0 then
    break;
  else
    continue;
}
```

can be reduced to:

```
while k < n do {
  if a[k] == 0 then
    goto L1;
  else
    goto L2;
L1:
}
L2:
```

To implement this translation, the *break* and *continue* statements need to know the names or locations of the labels L1 and L2. Hence, new *breaklabel* and *continuelabel* methods need to be added to the framework. This is unfortunate, because every intention must now implement these methods, even though they are only relevant to *while*, *break* and *continue* intentions. To alleviate this problem, methods can be given default behaviour. For example, the default behaviour of *breaklabel* is to call *parent.breaklabel* and return the result. This solution is widely known as the *Chain of Responsibility* design pattern and is described by Gamma, *et al.* [32, pages 223–232].

Forwarding

The central goal of IP is to allow independently authored intentions to be simultaneously added to the language. It should also be possible to improve the implementation of the base language without breaking existing intentions. To achieve this goal, intentions need to be independent of the implementation of the base language. Otherwise, changing the base language is impossible without changing the intentions. One reason for a change to the base language might be the addition of a new feature, such as a pretty printing facility. However, sometimes changes need to be made to accommodate new intentions. For example, suppose one wishes to add intentions for exception handling to a language. This might be done by adding *try*, *throw* and *catch* commands (similar to those available in Java [8]) to the language. This is likely to involve major changes to the code generation mechanism, so the framework might change substantially. IP attempts to alleviate this problem by providing a mechanism called forwarding.

Forwarding seems to be the main innovative idea in the design of IP and can be separated from the other aspects of IP. It can be applied in other paradigms such as attribute grammars, as Van Wyk, de Moor, Kwiatkowski and I have shown [101]. Forwarding is used to define new intentions in terms of existing intentions. For example, if *goto* is an existing intention, then *break* can be defined by forwarding to *goto*:

break $\Longrightarrow$ **goto** *parent.breaklabel*

This is very similar to specifying a rewrite rule for *break*, but some of the methods of *break* are allowed to be defined manually if so desired. For example, it is better to define the pretty printing method manually, because otherwise breaks will be pretty printed as gotos.

The advantage of forwarding is that it makes new intentions less dependent on the implementation details of the base language. This means that new intentions can be added to the system with less risk of independently authored intentions interfering with each other. However, it is only possible to *guarantee* that two intentions will not interfere if they are defined *entirely* by forwarding. If this is the case, then one has to ask why those intentions could not have been defined as subroutines.

1.4.3 Modular Attribute Grammars

Attribute grammars (AGs) are a formalism for describing the syntax, semantics and translation of programming languages. The phase of compilation for which they are most commonly used is the translation of abstract syntax to intermediate code. They are also a powerful tool for implementing *syntax directed editors*. Attribute grammar systems such as The Synthesizer Generator [85] are capable of automatically generating such editors. AGs were introduced by Knuth [55, 56], who proved that they can be tested for circularity and closure. Unfortunately, Knuth's definition of attribute grammars does not include any kind of module system. Dueck and Cormack [24] quote several sources who have found that this makes AGs unsuitable for

Stmt	*code*	*env*
While		
Block		
Assign		
⋮		

Expr	*code*	*env*	*type*
Var			
And			
Plus			
⋮			

Table 1.1: Tables of method implementations

large projects. Therefore, Dueck and Cormack introduced the concept of a modular attribute grammar. Their aim was to give existing AGs a clearer structure. Adams [6] takes the idea further and uses AG modules to describe language components. He advocates the use of modular AGs for rapidly creating domain specific languages.

There are many similarities between intentional programming and attribute grammars. Method calls between intentions are analogous to attributes, only without the inherited/synthesised distinction. Many AG systems provide *default copy rules*, which are very similar to IP's defaulting mechanism. Other powerful notations for passing attributes around the tree have been proposed by Kastens and Waite [51].

Aspects

Modular attribute grammars solve a problem orthogonal to the problem solved by forwarding (although Van Wyk, de Moor, Kwiatkowski and I [101] have found that the two techniques can be easily integrated). The problem also occurs in object oriented programming, so OO terminology is used to describe it below.

Recall the object oriented representation of abstract syntax, discussed in §1.4.2. Stmt and Expr are interfaces implemented by the classes While, Block, Var, *etc.* This hierarchy was illustrated by the UML class diagram in Figure 1.2. Suppose the following methods are added to the Stmt and Expr interfaces:

Stmt: *code*, *env*.
Expr: *code*, *env*, *type*.

A new UML diagram reflecting this change is given in Figure 1.4. The subclasses (While, Block, Var, *etc*) must now provide implementations of these methods. Hence two tables full of implementations need to be provided, as depicted in Table 1.1. Most object oriented languages require the method implementations to be grouped by class. For example, the definition of While is written in Java as:

```
class While implements Stmt {
    ...
    public Code code() { ... }
    public Env env() { ... }
    ...
}
```

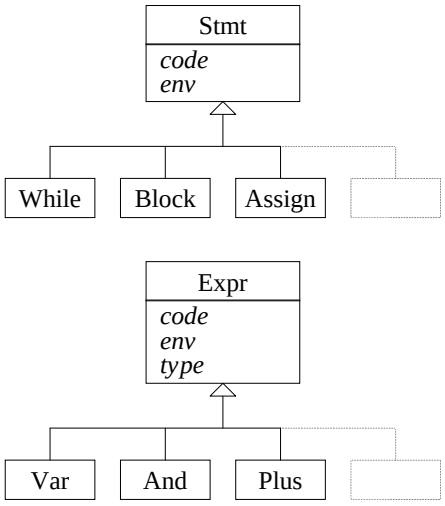


Figure 1.4: The UML class diagram with methods.

This class-based grouping can be inconvenient, because the implementations of *code* and *env* are spread over several class definitions. An alternative grouping with the implementations of the *code* method all given in one place is called *aspect-oriented*. This term was introduced by Kiczales [53], of AspectJ Inc. AspectJ is an extended version of Java with *aspect-oriented* programming features. It allows the programmer to define all the *code* methods in one place. These definitions are then *woven* into the class definitions.

Aspect-oriented and class-based groupings are both useful for developing an extensible compiler. Firstly, adding a new language feature involves adding new productions to the grammar. Class-based grouping encapsulates productions with their semantics, so it is ideal for this. Secondly, adding a new language feature might require a new aspect to be introduced. For example, a new error-checking aspect might need to be added. Ideally then, an environment for developing extensible compilers should allow both forms of grouping. A language such as AspectJ is such an environment. In the Oxford IP project, we have investigated adding aspect-oriented features to attribute grammars [72, 73].

As a grouping mechanism, aspects are superficially similar to modules, but their semantics seem ad-hoc. Any notion corresponding to abstract datatypes is, for instance, absent. If an aspect-oriented language allows both class-based and aspect-oriented groupings, then it seems reasonable that it should also allow arbitrary combinations of the two. For example, it should be valid to write a module that defines the methods While.*code*, Block.*code* and Block.*env*. These methods form neither a class or an aspect, so the type signature for the group must list them individually. This complexity of the type signatures does not bode well for the readability of large programs.

1.4.4 Modular Monadic Interpreters

Liang, *et al.* [66] discuss a highly sophisticated approach to constructing languages from components. Rather than compilers they build interpreters from components, but Harrison and Kamin [37] have shown that with the help of a partial evaluator the same techniques can be applied to the construction of compilers. Liang, *et al.* encapsulate the behaviour of an interpreter using a *monad*: a technique invented by Moggi [69, 70, 71]. A language component is encoded as a *monad transformer*. By applying such a transformer to an existing interpreter, a new interpreter is created.

Modular monadic interpreters differ in an important way from intentional programming and modular attribute grammars: monad transformers must be applied one at a time. Hence it is not possible to extend the language with two new features *simultaneously*. Moreover, the order in which the features are added can affect the semantics of the resulting language. Liang, *et al.* build on earlier work by Steele [92], who uses a non-determinism feature to illustrate this. Steele points out that none of the resulting interpreters is incorrect: they are the legitimate result of combining the language features in that order. However, it would be incorrect to claim that these features could be added to a language simultaneously. The semantics of such a combination would be undefined.

1.4.5 Summary

This section summarised some alternative techniques for the rapid creation of Domain Specific Languages. The approaches covered are all based on the idea of *extending* an existing language. This drastically reduces the amount of work, because existing language features can be reused. Unfortunately, the inherent modularity of a DSEL (that one might take for granted) does not come for free with these other approaches. If two new features are *simultaneously* added to a language, then there is a risk that those features might interact in an unexpected way. If this risk sounds unlikely, then consider the problems that can arise in any language that has an inadequate module system. This dissertation was produced with LaTeX, which is just such a system. LaTeX packages are not encapsulated, so using a new package in a document can often cause other packages to cease working!

Modular attribute grammars provide a module system that ensures the modularity of the attribute grammar. This is modularity at the *meta-level* though: it prevents the attribute computations from interfering in unexpected ways; it does not prevent the code generated by the attribute grammar from behaving unexpectedly. Suppose for example that two features are added to an imperative language:

1. A feature that generates profiling information for the program. It does this by adding extra code to the beginning and end of every procedure.

2. Exception handling. It uses long jumps to jump to the nearest enclosing exception handler.

Raising an exception might prevent the profiling code at the end of a procedure from being executed, thereby causing the profiler to generate odd information or even to

fail. Forwarding can prevent problems like these by separating language extensions from the details of code-generation. However, if a language feature can implemented entirely in terms of forwarding, then why can it not be implemented as a subroutine?

Modular monadic interpreters tackle the issues of modularity more directly, because interpreters have no meta-level. It is interesting then that this approach forces a priority to be assigned to the language extensions. Even more interesting is that changing the priority can change the semantics. This work demonstrates the difficulty or even impossibility of creating an extensible compiler framework that is semantically sound. DSELs avoid these difficulties by working within a framework that is already semantically sound.

Chapter 2

Programming in the Typed Lambda Calculus

Most existing domain-specific embedded languages have been written in the functional language Haskell [78]: an advanced language with a state-of-the-art type system. Many of Haskell's sophisticated features—type classes in particular—are, however, not required here. Also, Haskell's semantics are complicated and have never been formally defined. The primary requirement in this dissertation is for a mathematical notation with simple semantics, so instead of Haskell, a typed lambda calculus is used. To improve the readability of examples, this chapter introduces some syntactic sugar for the lambda calculus, based loosely on Haskell's syntax. In particular, the syntactic sugar provides a convenient notation for recursive function definitions. A simple desugaring process translates such definitions into lambda terms that use the least fixed point combinator, μ.

DSELs are modules, so a module system is needed. Jones [50] has shown that *first-class* modules with abstract datatypes can encoded very simply in polymorphic lambda calculi, using structures and polymorphic type variables. This chapter contains a short tutorial on Jones's encoding. An advantage of Jones's encoding is that it is based on standard polymorphic typing, so the underlying theory is well understood. This also means that the *parametricity* theorem can be applied at the module level. In later chapters, the parametricity theorem is used to derive properties of DSELs that have been encoded as first-class modules.

2.1 The Core Language

Following Landin's suggestion [60], the lambda calculus is used in this dissertation as a generic set of structuring mechanisms with which to manipulate primitives. Sometimes the primitives to be manipulated are complex entities such as parser combinators, but at the most atomic level the crucial ingredients are basic datatypes such as integers, lists and sets. For most programming tasks, it is also essential to include the least fixpoint operator, μ, as a primitive, because the *strong normalisation property* states that typed lambda terms always reduce in a finite number of steps. With-

out μ it is, therefore, impossible to write a program that uses general, rather than structural, recursion. An advantage of regarding μ as a primitive is that it allows the core language to retain its simple semantics, free from the complexities caused by recursion. For example, there is no need to distinguish between call-by-value and call-by-name, because strong normalisation states that both evaluation strategies lead to identical results. The semantics of μ are discussed separately in Chapter 3.

While a typed lambda calculus is perfect for the theoretical work in this dissertation, its syntax can be a little clumsy for writing non-trivial programs. It helps to have some syntactic sugar for dealing with the primitives mentioned above. As well as variables, abstraction, application, and let-bindings, which are standard features of the lambda-calculus, the syntax defined in Figure 2.1 includes three additional forms of notation. Constructors and case-expressions are used to manipulate algebraic data-types; they are discussed in §2.2. The Haskell-style use of recursive declarations as syntactic sugar for μ is discussed in §2.3. Structures are also discussed in §2.3.

The type system used in this dissertation is the polymorphic type system designed by Milner for ML [68]. The syntax of types is given in Figure 2.2. The symbols σ, τ and α represent polymorphic types, monomorphic types and type variables, respectively. This separation of monomorphic and polymorphic types prohibits types with nested polymorphism, such as $\forall\alpha.\ \alpha \leftarrow (\forall\beta.\ \beta \leftarrow \beta) \leftarrow \alpha$. The $\forall$-quantification may only appear at the top level, so only types such as $\forall\alpha.\ \forall\beta.\ \alpha \leftarrow \beta$ are allowed. Milner made this restriction to ensure that the most general type can be inferred by an automatic algorithm.[1]

2.1.1 Semantics

The approach to semantics in this dissertation is denotational, rather than, say, operational or algebraic. In the denotational approach, a mathematical value is assigned to every typeable term. This is done by a function $\mathcal{E}$, defined as follows:

$$
\begin{array}{lcll}
\mathcal{E}[\![x]\!]\ \Gamma & = & \Gamma(x), & \\
\mathcal{E}[\![\lambda x.e]\!]\ \Gamma & = & f, & \\
 & & \text{where} \quad f(v) & = \quad \mathcal{E}[\![e]\!]\ (\Gamma \oplus \{x \mapsto v\}), \\
\mathcal{E}[\![e_1\ e_2]\!]\ \Gamma & = & f(v), & \\
 & & \text{where} \quad f & = \quad \mathcal{E}[\![e_1]\!]\ \Gamma, \\
 & & \quad\ \ v & = \quad \mathcal{E}[\![e_2]\!]\ \Gamma.
\end{array}
$$

The function $\mathcal{E}$ takes a term, e, and an environment, Γ, which maps identifiers to values, and produces a value for e by structural recursion over its syntax. The rules given above are based on the assumption that the term is typeable in the environment provided by Γ. Otherwise, the first rule will fail if x is not defined in Γ and the third rule will fail if e_1 does not denote a function.

[1] Milner's type system for ML does not include signature types, such as those included in Figure 2.2, because they can be difficult to infer automatically. In this dissertation, structures are explicitly annotated with their types to avoid the need for an inference mechanism.

E	$::=$	x	*variable*
	$\mid$	C	*constructor*[†]
	$\mid$	$E\ E$	*application*
	$\mid$	$\lambda x.E$	*abstraction*
	$\mid$	**let** D^+ **in** E	*local definition*[‡]
	$\mid$	$(E, \ldots, E)$	*tuple*
	$\mid$	**case** E **of** $\{P \rightarrow E\}^+$	*case*
	$\mid$	$E.x$	*field selection*
	$\mid$	**struct** D^+ **end**	*structure*[‡]
	$\mid$	**open** E **in** E	*structure inclusion*
D	$::=$	$x\ P^* = E$	*declaration*
	$\mid$	$x :: \sigma$	*type annotation*
P	$::=$	$C\ P^*$	*constructor pattern*
	$\mid$	$(P, \ldots, P)$	*tuple pattern*
	$\mid$	x	*pattern variable*

[†]Constructors are reserved words and are, therefore, simple to distinguish from variables. (See §2.2.)

[‡]Lists of declarations are comma-separated.

Figure 2.1: The Syntax of the Language

τ	$::=$	α	*type variable*
	$\mid$	$[\tau]$	*list*
	$\mid$	$(\tau, \ldots, \tau)$	*tuple*
	$\mid$	$\tau \leftarrow \tau$	*function*[†]
	$\mid$	**sig** $\{x :: \tau\}^+$ **end**	*signature*[‡]
	$\mid$	$R\ \tau^*$	*algebraic type*
σ	$::=$	$\forall\alpha.\sigma$	*polytype*
	$\mid$	τ	*monotype*

[†]Function types are written backwards, for consistency with relations. (See §4.1.)

[‡]The declarations in a signature are comma-separated.

Figure 2.2: The Syntax of Types

Every function in this framework is total. That is, a function cannot go wrong or fail to terminate. It is possible, however, to *model* such effects. A program that would fail to terminate on a real machine can be represented mathematically by a function that computes a special value, $\perp$, that denotes failure or non-termination. This idea is discussed in more detail in Chapter 3.

New primitives can be added to the language very easily by adding new rules to the definition of $\mathcal{E}$. For example, an *plus* primitive merely requires the following new rule:

$$\begin{array}{lcl} \mathcal{E}[\![plus]\!]\ \Gamma & = & f, \\ & & \text{where} \quad f(x,y) \quad = \quad x + y \end{array}$$

This modularity in the definition of $\mathcal{E}$ is the reason why it is easy to separate the structuring mechanisms from the primitives: the semantics of each new primitive can be defined in isolation and easily incorporated into the whole.

2.2 Algebraic Datatypes

Haskell's notation is used to define new algebraic datatypes. For example, the *Maybe* datatype is defined as follows:

data *Maybe* α = *Just* α | *Nothing*.

In Haskell, datatype and function definitions can be interleaved, but in this dissertation, datatype definitions are regarded as strictly separate from program definitions, because datatype definitions create new primitives and are, therefore, part of the definition of the language, rather than the program. The above definition of *Maybe* contributes the following new primitives to the language:

- A type constructor, *Maybe*.
- Two data constructors, *Just* and *Nothing*.
- The ability to unwrap values of type *Maybe* α with a case-expression.

The final item in this list refers to the following notation, provided in Figure 2.1:

case m **of**
  Just x $\rightarrow f\ x$
  Nothing $\rightarrow y$.

This notation is syntactic sugar for an application of the primitive function *getMaybe*, defined as follows:

$$\begin{array}{l} getMaybe \qquad\qquad :: \forall\alpha,\beta.\ \beta \leftarrow \beta \leftarrow (\beta \leftarrow \alpha) \leftarrow Maybe\ \alpha, \\ getMaybe\ (Just\ x)\ f\ y = f\ x, \\ getMaybe\ Nothing\ f\ y = y. \end{array}$$

The case-expression, above, computes the same result as *getMaybe m f y*.

One of the reasons for allowing datatype and function definitions to be interleaved in Haskell, is to allow the module system to govern the visibility of datatypes. An *abstract datatype* is created by exporting the name of a datatype, but not its definition, from a module. In the framework described above, there is no such scoping of datatype definitions, because they are part of the language. Fortunately there is no need for scoping of datatype definitions in this dissertation, because Jones's module system [50], which is explained later in this chapter, uses polymorphism, rather than scoping, to make datatypes abstract.

An incidental advantage of regarding datatype definitions as part of the language definition is that constructor names, such as *Just* and *Nothing*, can be treated as reserved words in the syntax of the language. Hence, there is no need to follow Haskell's convention that constructors are capitalised and variables are not. For example, first class modules are often given capitalised names in this dissertation, which would be illegal in Haskell.

Sometimes it is useful to define type synonyms, which is again done using Haskell's notation:

$$\textbf{type}\ \mathit{Fix}\ \alpha\ =\ \alpha \leftarrow (\alpha \leftarrow \alpha).$$

A list is an algebraic datatype with two constructors, *Nil* and *Cons*, but it is convenient to adopt Haskell's notation. Hence, *Nil* is written [] and *Cons x xs* is written $x{:}xs$. The type of a list is written $[A]$, although the more verbose notation *List A* is sometimes used to improve readability when applying the parametricity theorem. Haskell's notation for tuples is also adopted. For example, $(0, \mathit{False})$ is a pair of type $(\mathit{Int}, \mathit{Bool})$.

Even though Haskell's notation is used to define algebraic datatypes, the resulting datatypes do not have equivalent semantics to their Haskell counterparts. Haskell datatypes are implicitly *lazy*, which means, for example, that it is possible to build a cyclic list. As discussed in §3.4, this is due to the implicit inclusion of a $\perp$ value in every datatype. The datatypes defined here contain no such implicit values: when they are required, they are defined explicitly.

2.3 Declarations and Structures

Lists of declarations are used in the style of Haskell to define functions. For example, the definition of *map* is:

$$\begin{array}{ll} \mathit{map} & :: \forall\alpha,\beta.\ [\beta] \leftarrow [\alpha] \leftarrow (\beta \leftarrow \alpha), \\ \mathit{map}\ f\ [] & = [], \\ \mathit{map}\ f\ (x : xs) & = f\ x\ :\ \mathit{map}\ f\ xs. \end{array}$$

In terms of the grammar given in Figure 2.1, this is a comma-separated list of declarations. Multiple declarations have been used here instead of a case-expression. An equivalent formulation is:

$$
\begin{array}{l}
map \;::\; \forall\alpha,\beta.\; [\beta] \leftarrow [\alpha] \leftarrow (\beta \leftarrow \alpha), \\
map \;=\; (\lambda f\; xs.\; \textbf{case}\; xs\; \textbf{of} \\
\qquad\qquad\qquad\qquad [] \quad\;\; \rightarrow [] \\
\qquad\qquad\qquad\qquad x : xs' \rightarrow f\; x \;:\; map\; f\; xs').
\end{array}
$$

The recursion in the definition of *map* is lowered to an application of μ, by first factoring out the recursive use of *map*:

$$
\begin{array}{l}
map \;::\; \forall\alpha,\beta.\; [\beta] \leftarrow [\alpha] \leftarrow (\beta \leftarrow \alpha), \\
map \;=\; (\lambda\, m\; f\; xs.\; \textbf{case}\; xs\; \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad [] \quad\;\; \rightarrow [] \\
\qquad\qquad\qquad\qquad\qquad x : xs' \rightarrow f\; x \;:\; m\; f\; xs')\;\; map.
\end{array}
$$

Any definition of the form $x = f\; x$ is syntactic sugar for $x = \mu\; f$, so the lowered definition of *map* is:

$$
\begin{array}{l}
map \;::\; \forall\alpha,\beta.\; [\beta] \leftarrow [\alpha] \leftarrow (\beta \leftarrow \alpha), \\
map \;=\; \mu\;\; (\lambda\, m\; f\; xs.\; \textbf{case}\; xs\; \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad [] \quad\;\; \rightarrow [] \\
\qquad\qquad\qquad\qquad\qquad x : xs' \rightarrow f\; x \;:\; m\; f\; xs').
\end{array}
$$

Any recursive use of declarations can be reduced to an application of μ in this way. The translation is slightly more complex if the declarations define a set of mutually recursive functions. Consider, for example, the following (slightly contrived) version of the factorial function:

$$
\begin{array}{l}
f \;\;\;::\; int \leftarrow int, \\
f\; 0 \;=\; 1, \\
f\; n \;=\; g\; n,
\end{array}
$$

$$
\begin{array}{l}
g \;\;\;::\; int \leftarrow int, \\
g\; n \;=\; n \cdot f\; (n{-}1).
\end{array}
$$

The solution in mutually recursive cases such as this is to use μ to compute a tuple of functions. The above definitions are equivalent to:

$$
\begin{array}{l}
f \;::\; int \leftarrow int, \\
f \;=\; fst\; (\mu\; F),
\end{array}
$$

$$
\begin{array}{l}
g \;::\; int \leftarrow int, \\
g \;=\; snd\; (\mu\; F),
\end{array}
$$

where the definition of F is:

$$
\begin{array}{l}
F \;::\; (int \leftarrow int,\; int \leftarrow int) \leftarrow (int \leftarrow int,\; int \leftarrow int), \\
F \;=\; \lambda\, m.\; \textbf{let}\; f' = fst\; m, \\
\qquad\qquad\qquad g' = snd\; m\;\; \textbf{in}\;\; (\lambda\, n.\; \textbf{case}\; n\; \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad 0 \rightarrow 1 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad n \rightarrow g'\; n,\;\; \lambda\, n.\; n \cdot f'\; (n{-}1)).
\end{array}
$$

In general, a strongly connected component analysis can be used to find the sets of mutually recursive functions.

2.3.1 Structures

A structure is nothing more than a tuple with named fields, but as explained later in this chapter, it can serve as a first-class module. A structure is created by an expression of the form **struct** $d_1 \ldots d_n$ **end**, where the d_i are declarations, as discussed above. The comments regarding the use of recursive declarations to define recursive functions also apply to structures: if one of the fields of a structure is a recursive function, then a recursive set of declarations can be used to define it. Two mechanisms are provided for accessing the fields of a structure. The first mechanism is selection: if M is a module with a field named x, then $M.x$ is the value of the field. The second mechanism is the **open** ... **in** ... construction, which Pascal programmers will recognise as being equivalent to Pascal's **with** ... **do** ... construction. It brings the fields of the module into scope, so that they can be accessed directly, without using the field selection operator. For example, if M has fields named x and y, then the expression **open** M **in** $x + y + 2$, is equivalent to the expression $M.x + M.y + 2$.

The type of a structure is called a signature in Figure 2.2. A signature lists the fields defined in the structure and their types. Note that the fields of the structure can only have monomorphic types, due to the restrictions in ML's type system.

2.4 First Class Modules

Prior to discussing Jones's module system [50], it is worth noting that first-class modules are already in widespread use: an object in an object-oriented language is a first-class module. It is first-class, because it can be created dynamically and passed as a parameter. It is a module, because it is an encapsulated unit with an abstract interface. Modules also often have a third characteristic, which is that they can be compiled separately. This is achieved by defining each module in a separate source file. Object-oriented languages usually allow objects to be compiled separately, though some, such as Java [8], enforce a stricter correspondence between objects and source file than others, such as C++ [93].

An object is, in essence, just a structure that can contain methods as well as values. Jones's module system is, in fact, just an idiomatic use of structures and types that achieves the equivalent of a complex module system. By allowing structures to contain functions and sub-structures, they can fulfil all the roles of a module while retaining their first-class status.

A key difference between objects and the modules presented here is that objects have state. In object oriented programming, method calls always have an implicit parameter: a reference to the object on which the method is being called. This enables the method to refer to the state of the object. There is no such implicit parameter here.

The real innovation in Jones's module system is in the use of polymorphic types to encode signatures and abstract datatypes. This leads to a simple type system for first-class modules that is ideally suited to the purposes of this dissertation. Due to the clean integration of first class modules with polymorphic typing and higher order

functions, it is also simple to create functor modules and parametric modules. In contrast, the template system in C++ [93, pages 327–354] is highly complicated and *generics* [16] are currently still absent from Java.

2.4.1 Using Structures as First-Class Modules

A structure in a higher-order language is more versatile than a record in a first-order language, such as Pascal, because it can contain functions and sub-structures. For example, a complex number module might be declared as follows:[2]

$$
\begin{array}{lcl}
\mathit{Complex} & = & \textbf{struct} \;\; \mathit{mk} \;\; = \; \lambda\, r\; i.\; \textbf{struct}\; \mathit{re} \; = r, \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{im} \; = i \\
 & & \qquad\qquad\qquad\qquad\quad \textbf{end}, \\
 & & \qquad\quad \mathit{plus} \;\; = \; \lambda\, x\; y.\; \mathit{mk}\; (x.\mathit{re} + y.\mathit{re})\; (x.\mathit{im} + y.\mathit{im}), \\
 & & \qquad\quad \mathit{minus} \; = \; \lambda\, x\; y.\; \mathit{mk}\; (x.\mathit{re} - y.\mathit{re})\; (x.\mathit{im} - y.\mathit{im}) \\
 & & \textbf{end}.
\end{array}
$$

This module contains three functions:

1. *mk* constructs a complex number from a real and an imaginary component.

2. *plus* adds two complex numbers.

3. *minus* subtracts two complex numbers.

Note that structures are used in two ways in this module. A structure is used to construct the module itself, but complex numbers are also represented as a structures (containing fields *re* and *im*). The latter use of structures corresponds to their traditional use in languages such as Pascal.

To facilitate the use of structures as modules, the syntax allows mutual recursion between the fields of a structure, as mentioned in §2.3.1. There is no recursion in the definition of *Complex*, but *mk* is used as a subroutine by *plus* and *minus*, which would not be allowed if the syntax were stricter.

2.4.2 Functor Modules

Structures can be used to define ML-style *functor modules*. A functor module is a module that is parameterised by another module. A typical example of this is a *Matrix* module. The *Matrix* module might support operations such as plus and minus, but the implementation of these operations depends on the type of the matrix elements: an implementation for complex numbers is different than an implementation for integers. This problem can be solved by parameterising *Matrix* by a module called *Element*:

[2]The type of *Complex* is discussed in §2.5.

$$
\begin{array}{l}
\mathit{Matrix} = \\
\quad \lambda\, \mathit{Element}.\ \textbf{struct} \\
\qquad\qquad \mathit{plus} \ = \ \lambda\, M\ N.\ \mathit{zipWith}\ (\mathit{zipWith}\ \mathit{Element.plus})\ M\ \ N, \\
\qquad\qquad \mathit{minus} \ = \ \lambda\, M\ N.\ \mathit{zipWith}\ (\mathit{zipWith}\ \mathit{Element.minus})\ M\ \ N \\
\qquad \textbf{end}.
\end{array}
$$

Element can be any module that contains functions called *plus* and *minus*. This allows *Matrix* to add or subtract two matrices by applying the operations from *Element* to the corresponding elements. The *Complex* module defined above is a suitable parameter for *Matrix*. (In this implementation, matrices are represented as lists of lists. The function *zipWith* is defined in Appendix A.2.)

2.4.3 Private Values

A convenient feature of practical module systems is that not all values defined in a module need to be exported. This allows local helper functions to be defined without them appearing in the interface to the module. For example, in Java a method or field can be designated *private*, which means that it is not visible outside the class. A let-binding can be used to achieve the same effect. Suppose for example that the *mk* function of the *Complex* module is not intended to be externally visible:

$$
\begin{array}{lll}
\mathit{Complex} \ = \ \textbf{let}\ \mathit{mk} \ = \ \lambda\, r\ i.\ \textbf{struct}\ \mathit{re} \ = r, & & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{im} \ = i & & \\
\qquad\qquad\qquad\qquad\quad \textbf{end}\ \textbf{in} & & \\
\qquad \textbf{struct}\ \ \mathit{plus} \ & = \ \lambda\, x\ y.\ \mathit{mk}\ (x.\mathit{re} + y.\mathit{re})\ (x.\mathit{im} + y.\mathit{im}), & \\
\qquad\qquad\ \ \mathit{minus} \ & = \ \lambda\, x\ y.\ \mathit{mk}\ (x.\mathit{re} - y.\mathit{re})\ (x.\mathit{im} - y.\mathit{im}) & \\
\qquad \textbf{end}. & &
\end{array}
$$

It would of course be straightforward to add syntactic sugar for private values. This would allow the definitions of private values to be interleaved with the other fields.

2.5 Abstract Datatypes as Polymorphic Types

In most module systems, a module defines a set of values, functions and types. The programmer can decide which of these definitions are to be made *public*. For example as discussed above, the *mk* function of the complex module could be kept private. Similarly, the definition of the complex number datatype could be kept private, thereby making it an *abstract datatype*. In OCaml [65] (an ML dialect), the signature of the *Complex* module is:

```
module type COMPLEX = sig
   type c

   val mk    :  float -> float -> c
   val plus  :  c -> c -> c
   val minus :  c -> c -> c
end
```

Type `c` is the abstract datatype of complex numbers. It might be implemented as a tuple or a record and it might store a cartesian or polar representation. The user is prevented from writing code that depends on the implementation of `c`. For example obtaining the real part of a complex number by applying the function `fst` is illegal, even if the programmer happens to know that complex numbers are represented as pairs.

2.5.1 Types and Values are Separate

In Jones's proposal, modules *do not* define datatypes. In OCaml, types and values in modules are intertwined in a fairly complex way, both syntactically and semantically: an intricate dependent type system is needed to define the typing of OCaml modules. The simplicity in Jones's model comes from completely separating the definitions of types and values. Hence, the decision in §2.2 to regard algebraic datatypes as primitives of the language is compatible with Jones's model.

To give the *Complex* module a signature, it is convenient to define the following type synonym:

$$\begin{array}{l}\textbf{type}\ \mathit{Cpx}\ \gamma = \textbf{sig}\ \mathit{mk} \quad :: \gamma \leftarrow \mathit{Float} \leftarrow \mathit{Float},\\ \qquad\qquad\qquad\quad \mathit{plus} \quad :: \gamma \leftarrow \gamma \leftarrow \gamma,\\ \qquad\qquad\qquad\quad \mathit{minus} :: \gamma \leftarrow \gamma \leftarrow \gamma\\ \qquad\qquad\quad \textbf{end}.\end{array}$$

The polymorphic type variable γ is used to leave the type of complex numbers undefined. However in the definition of the *Complex* module, complex numbers are represented as structures:

$$\begin{array}{l}\textbf{type}\ \mathit{cnum} = \textbf{sig}\ \mathit{re}\ :: \mathit{Float},\\ \qquad\qquad\qquad\quad \mathit{im} :: \mathit{Float}\\ \qquad\qquad\quad \textbf{end}.\end{array}$$

Therefore, the type of the *Complex* module is:

$$\mathit{Complex} :: \mathit{Cpx}\ \mathit{cnum}.$$

Here *Cpx* has been instantiated with $\gamma := \mathit{cnum}$. This is quite different from the OCaml signature for the same module, because the definition of γ is not hidden: the type *cnum* is quite clearly being used to represent complex numbers. However, consider a program that uses complex numbers:

$$\begin{array}{lcl}\mathit{addlist} & :: & \forall\gamma.\ \gamma \leftarrow [\gamma] \leftarrow \mathit{Cpx}\ \gamma,\\ \mathit{addlist}\ C\ [] & = & C.\mathit{mk}\ 0\ 0,\\ \mathit{addlist}\ C\ (x : xs) & = & C.\mathit{plus}\ x\ (\mathit{addlist}\ C\ xs).\end{array}$$

This program sums a list of complex numbers. It is parameterised by the complex numbers module, so that different implementations can be used. The polymorphic type of the program indicates that it treats complex numbers as an abstract datatype, which is how abstract datatypes are encoded in Jones's model.

2.5.2 Existential Types

The work in this dissertation is based on the idea of instantiating a program with different implementations of the same module signature. Therefore, it makes perfect sense for the module to be a parameter of the program, as in the definition of *addlist* above. In a practical system though, it would be convenient to be able to indicate that a datatype is abstract without needing to thread extra type parameters through the entire program. This can be done by introducing an $\exists$-quantifier and giving *Complex* the following type:

$$Complex :: \exists\gamma.\ Cpx\ \gamma.$$

This definition states that *Complex* is *an* implementation of the *Cpx* signature and it hides the implementation of complex numbers. An advantage of this definition is that the library designer can ensure that a type is abstract. In the definition of *addlist*, it was the library *user* who needed to ensure that complex numbers were treated as abstract.

2.6 Summary

Rather than a real programming language, such as Haskell [78], a polymorphically typed lambda calculus is used throughout this dissertation, because of its simple semantics. Following Landin's suggestion [60], a distinction is made between the core language and the primitives. The core language contains just variables, abstraction and application; added to this are the least fixed point operator, μ, and primitives for manipulating algebraic datatypes. Least fixed points are discussed further in Chapter 3. To improve the presentation, some notation has been borrowed from Haskell. In particular, multiple declarations are used to define the clauses of a function. Such syntactic sugar can be removed by a simple lowering process.

Jones [50] has shown how to encode first-class modules with abstract datatypes in a polymorphically typed lambda calculus, using just structures, which are tuples with named fields. His encoding is used in this dissertation to encode DSELs as first-class modules. An important consequence of this approach is that the *parametricity* theorem can be applied to DSELs.

Chapter 3

Abstract Interpretation

Abstract interpretation is a technique invented by Cousot and Cousot [21]. The technique is to analyse properties of programs by evaluating them with an *abstract* interpreter. To illustrate this with a simple example, consider a desk calculator. The calculator reads expressions such as

$$2 \times ((3 \times 5) + 7)$$

and computes their values. The calculator can be thought of as an interpreter for a trivial programming language consisting of integer constants, addition, multiplication, and parentheses. The grammar of the language is:

$$\begin{array}{lcll} E & ::= & C & \textit{integer constant} \\ & | & (E) & \textit{parenthesised expression} \\ & | & E + E & \textit{addition} \\ & | & E \times E & \textit{multiplication} \end{array}$$

An abstract syntax corresponding to this grammar is:

data $Expr = Const\ Int \mid Plus\ Expr\ Expr \mid Times\ Expr\ Expr.$

The definition of the calculator is then:

$$\begin{array}{lcl} calc & :: & Int \leftarrow Expr, \\ calc\ (Const\ k) & = & k, \\ calc\ (Plus\ e_1\ e_2) & = & calc\ e_1 + calc\ e_2, \\ calc\ (Times\ e_1\ e_2) & = & calc\ e_1 \cdot calc\ e_2. \end{array}$$

An *abstract* interpreter called *calcEven* can be used to predict if *calc* will produce an even result. The specification of *calcEven* is:

$$calcEven = even \circ calc,$$

where *even* is a function that returns *True* if its argument is even:

$$\begin{array}{lcl} even & :: & Bool \leftarrow Nat, \\ even\ n & = & (\exists k.\ n = 2 \cdot k). \end{array}$$

The function *even* is known as the *abstraction function*. In most examples of abstract interpretation, the abstraction function is the *lower adjoint* of a *Galois connection*. (These concepts are explained in §3.7.) The abstraction function is typically used to specify the analysis as has been done above. A concrete definition of the abstract interpreter is then derived from the specification. Here this process leads to the following definition of *calcEven*:

$$\begin{array}{lll} \mathit{calcEven} & :: & \mathit{Bool} \leftarrow \mathit{Expr}, \\ \mathit{calcEven}\ (\mathit{Const}\ k) & = & \mathit{even}\ k, \\ \mathit{calcEven}\ (\mathit{Plus}\ e_1\ e_2) & = & (\mathit{calcEven}\ e_1 \equiv \mathit{calcEven}\ e_2), \\ \mathit{calcEven}\ (\mathit{Times}\ e_1\ e_2) & = & (\mathit{calcEven}\ e_1 \vee \mathit{calcEven}\ e_2). \end{array}$$

This definition is based on the properties that $x \times y$ is even if either x or y is even and that $x + y$ is even if x and y are both even or both odd.

The calculator example does not demonstrate the full power of abstract interpretation, because the programming language does not have variables or parameters. Abstract interpretation comes into its own when the program is parameterised and therefore has an infinite number of different inputs. It is not possible to test the program on every possible input, so abstract interpretation can help by reducing the number of possible inputs. For example, a function of n integers has an infinite number of inputs, but a function of n Booleans has precisely 2^n input combinations, so an abstract interpreter based on Booleans could test every possible input.

For a full introduction to abstract interpretation, see Abramsky and Hankin [4] or Nielson *et al.* [76, pages 209–271]. In this chapter, a brief introduction to the topic is given by developing a strictness analyser for a simple first-order functional language. Along the way, posets and least fixed points are discussed so that the semantics of the first-order language can be properly defined. Galois connections, which are important in abstract interpretation, are also introduced.

3.1 A First-Order Functional Language

The classic application of abstract interpretation, discovered by Mycroft [74], is the strictness analysis of lazy functional programs. The material in this chapter is illustrated by developing a strictness analyser for a simple first-order functional language. This is achieved by following the standard process of abstract interpretation:

1. An interpreter for the language is defined.

2. The analysis is specified by means of a Galois connection.

3. The Galois connection is used to derive an *abstract* interpreter for the language.

The strictness analysis of a program is performed by evaluating the program with the abstract interpreter. As an introduction to the later chapters, the interpreter is defined as a polymorphic function, using the notation of Chapter 2.

A first-order language is a language that has functions, but not higher-order functions. That is, the language does not have lambda abstractions or currying. Instead, functions must be defined at the top level of the program. In other words, a program in the language has the following form:

$$\begin{array}{lll} \textbf{let} & f(x,\ldots,y) & = \ \ldots \\ & \vdots & \\ & g(x,\ldots,y) & = \ \ldots \\ \textbf{in} & & \\ \ldots & & \end{array}$$

This program consists of a set of first order function definitions and an expression to be evaluated. The function definitions are allowed to be mutually recursive.

A sample program in this language is:

$$\textbf{let}\ \mathit{fac}\ (n)\ =\ \textbf{if}\ n \geq 1\ \textbf{then}\ n \cdot \mathit{fac}\ (n-1)\ \textbf{else}\ 1\ \textbf{in}\ \mathit{fac}\ (10).$$

This program computes the tenth factorial.

3.1.1 Abstract Syntax

A program is a list of function definitions and an expression to be evaluated. Therefore, the abstract syntax of programs is:

$$\textbf{type}\ \mathit{Prog} = ([\mathit{FunDef}], \mathit{Exp}).$$

A function definition consists of a name, a list of parameter names and a function body:

$$\textbf{type}\ \mathit{FunDef} = (\mathit{String}, [\mathit{String}], \mathit{Exp}).$$

An expression is either a function application, a constant application or a variable:

$$\textbf{data}\ \mathit{Exp} = \mathit{Call}\ \mathit{String}\ [\mathit{Exp}] \mid \mathit{Const}\ \mathit{String}\ [\mathit{Exp}] \mid \mathit{Var}\ \mathit{String}.$$

This language has been kept intentionally simple by considering most language features to be constants. For example, *if-then-else* is a constant function with three parameters: the condition and two expressions. Simple constants, such as the integer constant 10, are constant functions with zero parameters.

3.2 An Interpreter

In this section, an interpreter for the first-order language is defined using the notation of Chapter 2. The interpreter can be used to execute programs written in the language, but it also serves as the denotational semantics of the language. A discussion of semantic domains, which is necessary for a formal treatment of the semantics, can be found in §3.3.

The interpreter is a function with the following type:

$$evalProg :: \forall v.\ v \leftarrow Prog \leftarrow ConstEnv\ v.$$

This function computes the value of a program, given an environment defining the constants. An important property of this function is that it is polymorphic in the type of values. This means that different types of values can be computed by instantiating the interpreter with different constant environments. In fact, this is precisely how strictness analysis is performed in §3.8.

3.2.1 Environments

An environment is a function or table that defines the values of variables or constants. In the interpreter three different environments are used: one for the constants, one for the functions and one for variables. The constants environment was mentioned already above. It has the following type:

$$\textbf{type}\ ConstEnv\ v = v \leftarrow [v] \leftarrow String.$$

Its inputs are the name of the constant and a (possibly empty) list of parameter values. Its output is the value computed by the constant. As discussed above the interpreter is polymorphic in the type v, which is the type of runtime values. Different constant environments can therefore use different types. To evaluate trivial examples like factorial the only runtime values that are needed are integers and Booleans, so the following datatype suffices:[1]

$$\textbf{data}\ Value = IVal\ Int \mid BVal\ Bool.$$

A minimal set of constants based on this datatype, is:

$$\begin{array}{lll}
consts & & :: ConstEnv\ Value, \\
consts\ \text{“}\cdot\text{”} & [IVal\ x,\ IVal\ y] & = IVal\ (x \cdot y), \\
consts\ \text{“}-\text{”} & [IVal\ x,\ IVal\ y] & = IVal\ (x - y), \\
consts\ \text{“}\geq\text{”} & [IVal\ x,\ IVal\ y] & = BVal\ (x \geq y), \\
consts\ \text{“if”} & [BVal\ b,\ x,\ y] & = \textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y, \\
consts\ \text{“true”} & [] & = BVal\ True, \\
consts\ \text{“false”} & [] & = BVal\ False, \\
consts\ num & [] & = IVal\ (intOfString\ num).
\end{array}$$

Note that this function is based on pattern matching.[2] If a constant is used with the wrong number (or type) of arguments, then it will fail. This will lead to the interpreter crashing without a helpful error-message! So for the time being, assume that the interpreter is only applied to correct programs. What it means for the interpreter to fail is discussed in §3.3.

The environment for functions is identical to the environment for constants:

[1] The definition of *Value* is revised in §3.4.

[2] The *num* pattern is a non-standard pattern used for this example only. It matches any string that contains only decimal digits.

$$\textbf{type}\ \mathit{FunEnv}\ v = v \leftarrow [v] \leftarrow \mathit{String}.$$

The environment for variables is represented, instead, as an association list, so that new bindings can be added easily by adding them to the list:

$$\textbf{type}\ \mathit{VarEnv}\ v = [(v, \mathit{String})].$$

A function called *lookup* is used to find values in a variable environment. It has type:

$$\mathit{lookup} :: \forall v.\ v \leftarrow \mathit{String} \leftarrow \mathit{VarEnv}\ v.$$

This function fails if it is asked to find a value that does not appear in the list. Again, the discussion of failure is deferred to §3.3. Finally, it is often convenient to package all three environments as a tuple:

$$\textbf{type}\ \mathit{Envs}\ v = (\mathit{ConstEnv}\ v,\ \mathit{FunEnv}\ v,\ \mathit{VarEnv}\ v).$$

3.2.2 Interpreters for Expressions, Functions and Programs

Expressions, functions and programs are syntactically distinct in the first-order language, so it is easiest to define three interpreters: one for expressions, one for functions and one for programs. The expression and function interpreters are subroutines of the program interpreter.

Expressions depend on all three kinds of environment, so the expression interpreter *evalExp* is passed a tuple of type *Envs v*:

$$\begin{array}{lll}
\mathit{evalExp} & :: & \forall v.\ v \leftarrow \mathit{Exp} \leftarrow \mathit{Envs}\ v, \\
\mathit{evalExp}\ \mathit{envs}\ e & = & \textbf{let}\ (\mathit{constEnv}, \mathit{funEnv}, \mathit{varEnv}) = \mathit{envs}\ \textbf{in} \\
 & & \quad \textbf{case}\ e\ \textbf{of} \\
 & & \qquad \mathit{Const}\ f\ \mathit{es} \rightarrow \mathit{constEnv}\ f\ (\mathit{map}\ (\mathit{evalExp}\ \mathit{envs})\ \mathit{es}) \\
 & & \qquad \mathit{Call}\ f\ \mathit{es} \rightarrow \mathit{funEnv}\ f\ (\mathit{map}\ (\mathit{evalExp}\ \mathit{envs})\ \mathit{es}) \\
 & & \qquad \mathit{Var}\ x \rightarrow \mathit{lookup}\ \mathit{varEnv}\ x.
\end{array}$$

If the expression is a constant, the interpreter evaluates the arguments (by calling itself recursively) and then invokes the environment of constants. Similarly if the expression is a function, the interpreter evaluates the arguments and then invokes the environment of functions. Finally, if the expression is a variable then the environment of variables is used to determine its value.

The task of the function interpreter is to construct an entry for the function environment. However, functions can call each other, so the function environment is also a parameter of the function interpreter! This fact is important in the discussion of the program interpreter below.

$$\begin{array}{l}
\mathit{evalFun} :: \forall v.\ ((v \leftarrow [v]),\ \mathit{String}) \leftarrow \mathit{FunDef} \leftarrow \mathit{FunEnv}\ v \leftarrow \mathit{ConstEnv}\ v \\
\mathit{evalFun}\ \mathit{constEnv}\ \mathit{funEnv}\ (\mathit{name}, \mathit{params}, e) = \\
\quad ((\lambda\, \mathit{vs}.\ \mathit{evalExp}\ (\mathit{constEnv},\ \mathit{funEnv},\ \mathit{zip}\ \mathit{vs}\ \mathit{params})\ e),\ \mathit{name}).
\end{array}$$

An entry in the function environment consists of the function's name and its value. Its value is a function: given a list of values *vs* (the parameters to the function), it computes the value of the body of the function. This is done by calling *evalExp* on the body of the function, with the values of the parameters bound in the variable environment. The function *zip* (Appendix A.2) is used to construct the variable environment.

The interpreter for programs uses *evalExp* and *evalFun* as subroutines and is parameterised by the environment of constants:

$$
\begin{array}{l}
\mathit{evalProg} :: \forall v.\ v \leftarrow \mathit{Prog} \leftarrow \mathit{ConstEnv}\ v \\
\mathit{evalProg}\ \mathit{constEnv}\ (\mathit{fs}, e) = \\
\quad \textbf{let}\ \mathit{funEnv} = \mathit{lookup}\ (\mathit{map}\ (\mathit{evalFun}\ \mathit{constEnv}\ \mathit{funEnv})\ \mathit{fs})\ \textbf{in} \\
\quad \mathit{evalExp}\ (\mathit{constEnv}, \mathit{funEnv}, [])\ e.
\end{array}
$$

This definition sets up the function environment and then uses the expression interpreter to evaluate the main expression. (Note that the variable environment for the main expression is empty.) The interesting aspect of this interpreter is the *cyclic* definition of *funEnv*. This is necessary, because the functions are allowed to be mutually recursive, so the function interpreter needs access to the full function environment. However, in order to explain the meaning of cyclic definitions such as this, least fixed points need to be introduced.

3.3 Domains and Least Fixed Points

Cyclic equations, such as the definition of *funEnv* above, have the following general form:

$$x = f\ x.$$

Even mutually recursive cyclic equations can be written in this form, using tuples. For example, the equations

$$
\begin{array}{lcl}
x & = & f\ y, \\
y & = & g\ z, \\
z & = & h\ x,
\end{array}
$$

can be written in the form

$$(x, y, z) = F\ (x, y, z),$$

by defining F as:

$$F\ (x, y, z) = (f\ y, g\ z, h\ x).$$

A solution x of the equation $x = f\ x$ is known as a *fixed point* of f. Some equations do not have unique solutions though. For example, the trivial equation $x = x$ has many solutions. Ill-defined equations such as this cause programs to enter infinite loops. As an illustration, consider this simple function:

$$f\ n\ =\ f\ (n{-}1).$$

One valid solution to this equation is:

$$\mathrm{f}\ 0 = 1,\ \ \mathrm{f}\ 1 = 1,\ \ \mathrm{f}\ 2 = 1,\ \ \mathrm{f}\ 3 = 1,\ \ \ldots$$

but another equally valid solution is

$$\mathrm{f}\ 0 = 2,\ \ \mathrm{f}\ 1 = 2,\ \ \mathrm{f}\ 2 = 2,\ \ \mathrm{f}\ 3 = 2,\ \ \ldots$$

A computer will fail to find either of these solutions. Executing the program leads to non-termination.

3.3.1 A Semantic View

The semantic explanation of non-termination is that program execution corresponds to the computation of a *least* fixed point. The *least* solution to the definition of f, above, is:

$$\mathrm{f}\ 0 = \bot,\ \ \mathrm{f}\ 1 = \bot,\ \ \mathrm{f}\ 2 = \bot,\ \ \mathrm{f}\ 3 = \bot,\ \ \ldots$$

The value $\bot$, pronounced *bottom*, is an abstract value, representing non-termination or failure. The set of integers, augmented with this special value $\bot$, is sometimes written $Int_\bot$. It is called a *domain*. The semantic view is that programs compute *least fixed points* over *domains*. If the equations are well-defined, then the solution is as expected. If not, the solution is $\bot$.

3.3.2 Posets and Domains

A *partially ordered set* (poset) is a pair, consisting of a set and a partial ordering on the set.[3] The notation $(A, \preceq)$ is used to indicate the set and its ordering. A *domain* is a poset with the additional property that least fixed points of monotonic functions are guaranteed to exist. *Complete lattices*, which are discussed below, are one type of poset that satisfy this requirement.

The semantics of many programming languages are based on *flat* domains. These simple domains have the form illustrated in Figure 3.1 (although the top element is sometimes omitted). There are two special elements $\bot$ and $\top$, such that $\bot \preceq x$ and $x \preceq \top$, for all x; the other elements are unordered. This is the structure used in the $Int_\bot$ domain. Though one might expect that $0 \preceq 1 \preceq 2 \preceq \cdots$, these values are in fact unordered; the semantic and arithmetic orderings are unrelated.

[3] A partial ordering is a relation that is reflexive, anti-symmetric and transitive.

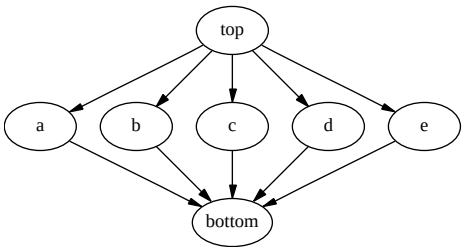


Figure 3.1: The structure of a flat domain.

3.3.3 Complete Lattices

Least fixed points were introduced to avoid the problem of there being *more than one* solution, but it is also desirable to avoid a situation in which there are *zero* solutions. A domain is a poset in which least fixed points are guaranteed to exist. The domains used in this dissertation are all *complete lattices*. There are alternatives to complete lattices, such as *complete partial orders* (CPOs), which have less restrictive definitions, but they are not used in this dissertation, because complete lattices have simpler properties.

A complete lattice is a domain $(A, \preceq)$ with two operators $\sqcup$ and $\sqcap$. If X is a subset of A, then $\sqcup X$ is the least upper bound of the elements of X and $\sqcap X$ is their greatest lower bound. That is:

$$\sqcup X \preceq u \quad \equiv \quad (\forall x \in X.\ x \preceq u)$$

and

$$l \preceq \sqcap X \quad \equiv \quad (\forall x \in X.\ l \preceq x)$$

Note that these rules imply that a complete lattice always has top and bottom elements: $\top$ and $\bot$.

A common example of a complete lattice is a flat domain, as illustrated in Figure 3.1: the elements of the domain are unordered, except with respect to the top and bottom elements. Another example is $(\mathcal{P}X, \subseteq)$, where $\mathcal{P}X$ denotes the power set of the set X. The elements of this lattice are sets, ordered by the subset relation. The least upper bound operator is union and the greatest lower bound operator is intersection.

3.3.4 Lifted Domains

Most function definitions use recursion on *functions* not *values*. For example, *fac* is defined by a recursive equation and it has type $Int \leftarrow Int$. Therefore, the ordering on functions needs to be defined. There is a canonical way of lifting an ordering to a function space, called a *pointwise* ordering. If $(A, \preceq)$ is a domain and B is an arbitrary type, then the pointwise ordering on $A \leftarrow B$ is:

$$f \mathrel{\dot{\preceq}} g \ \equiv \ (\forall x.\ f\ x \preceq g\ x).$$

If $\preceq$ is a partial order, then it is straightforward to prove that $\dot{\preceq}$ is a partial order. If $(A, \preceq)$ is a complete lattice, then lifted versions of the upper and lower bounds can be defined:

$$\dot{\sqcup} F = (\lambda x.\ \sqcup \{\ f\, x \mid f \in F\ \})$$
$$\dot{\sqcap} F = (\lambda x.\ \sqcap \{\ f\, x \mid f \in F\ \})$$

It is straightforward to prove that $(A \leftarrow B, \dot{\preceq})$ is a complete lattice with these operators.

Posets can also be lifted through algebraic datatypes such as lists. This is stated as a formal lemma in §5.2.2, after *relators* have been introduced in §4.2. Unfortunately, complete lattices do not always lift though algebraic datatypes. For example, the lifted partial ordering on lists does not have a top or bottom element. On the other hand, complete lattices do lift through tuples and structures. Hence, least fixed points over algebraic datatypes need to be checked carefully to make sure that they are valid. If fixpoints over a problematic datatype are unavoidable then the datatype definition can always be patched by adding extra top and bottom elements. For example, the lists used in this dissertation differ from lists in Haskell, because Haskell lists have a bottom element. Recursively defined lists are therefore valid in Haskell (and in fact frequently used). In Chapter 8, recursively defined lists are needed to define the semantics of attribute grammars. This problem is solved by using *sequences* rather than lists.

3.3.5 The Knaster-Tarski Theorem

The *Knaster-Tarski theorem* [54, 97] states that a fixed point equation over a complete lattice always has a least fixed point. The least fixed point of the equation $x = f\ x$ is written as μf. Sometimes the notation $(\mu x.\ e)$ is used as a shorthand for $\mu\ (\lambda x.\ e)$.

Theorem 1 (Knaster-Tarski) *Suppose $(A, \preceq)$ is a complete lattice and f is a monotonic function of type $A \leftarrow A$. Then f has a least fixed point, written μf. It satisfies:*

$$\mu f = f\ (\mu f) \quad \wedge \quad (\forall x.\ \mu f \preceq x \ \Leftarrow\ f\ x \preceq x)$$

The left- and right-hand properties are known as the *computation* and *induction* rules, respectively. Note that the induction rule implies that μf is the *least* fixed point, because $f\ x = x$ implies $\mu f \preceq x$. The induction rule can also be used to eliminate μ from an equation, thereby simplifying it. This proof technique is similar to induction, so it is called *fixpoint induction*. The fixpoint theorems given in §3.5 are all proved with fixpoint induction. (From now on least fixed points are often referred to as fixpoints, for brevity.)

3.4 Fixpoints and Functional Programming

Posets are an important semantic tool, but it is rare for them to be mentioned in a programming language manual. Despite this, the value $\bot$ is discussed at length by Bird [14] and in the Haskell Report [78]. While $\bot$ remains an invisible value, which is implicitly defined for every datatype, Haskell cannot be understood without it. Advanced Haskell programmers also sometimes use strictness annotations to improve the performance of their programs by altering the way that $\bot$ is manipulated. This use of Haskell is often regarded as impure, but the opposite view is taken here. Rather than initially sweeping $\bot$ under the carpet and later manipulating it with low-level, performance-oriented language features, it is always explicitly defined where it is needed: any datatype that is used in a least fixed point computation is a complete lattice.

In light of the above, some of the definitions of §3.2 need to be corrected. Most importantly, the definition of the datatype *Value* needs to be augmented with elements *Bot* and *Top*, to make it a complete lattice:

$$\textbf{data}\ \mathit{Value} = \mathit{Bot} \mid \mathit{IVal}\ \mathit{Int} \mid \mathit{BVal}\ \mathit{Bool} \mid \mathit{Top}.$$

The ordering on this datatype is a flat ordering, as illustrated in Figure 3.1. The mathematical equation is:

$$x \preceq y \ \equiv\ x = \mathit{Bot} \vee y = \mathit{Top} \vee x = y.$$

The definition of *consts* need to be revised, to take the values *Bot* and *Top* into account:

$$\begin{array}{llll}
\mathit{consts} & & & :: \mathit{ConstEnv}\ \mathit{Value}, \\
\mathit{consts} & \text{“}\cdot\text{”} & [\mathit{Bot}, y] & = \mathit{Bot}, \\
\mathit{consts} & \text{“}\cdot\text{”} & [x, \mathit{Bot}] & = \mathit{Bot}, \\
\mathit{consts} & \text{“}\cdot\text{”} & [\mathit{IVal}\ x,\ \mathit{IVal}\ y] & = \mathit{IVal}\ (x{\cdot}y), \\
\mathit{consts} & \text{“}-\text{”} & [\mathit{Bot}, y] & = \mathit{Bot}, \\
\mathit{consts} & \text{“}-\text{”} & [x, \mathit{Bot}] & = \mathit{Bot}, \\
\mathit{consts} & \text{“}-\text{”} & [\mathit{IVal}\ x,\ \mathit{IVal}\ y] & = \mathit{IVal}\ (x{-}y), \\
\mathit{consts} & \text{“}\geq\text{”} & [\mathit{Bot}, y] & = \mathit{Bot}, \\
\mathit{consts} & \text{“}\geq\text{”} & [x, \mathit{Bot}] & = \mathit{Bot}, \\
\mathit{consts} & \text{“}\geq\text{”} & [\mathit{IVal}\ x,\ \mathit{IVal}\ y] & = \mathit{BVal}\ (x \geq y), \\
\mathit{consts} & \text{“if”} & [\mathit{Bot}, x, y] & = \mathit{Bot}, \\
\mathit{consts} & \text{“if”} & [\mathit{BVal}\ b,\ x,\ y] & = \textbf{if}\ b\ \textbf{then}\ x\ \textbf{else}\ y, \\
\mathit{consts} & \text{“true”} & [] & = \mathit{BVal}\ \mathit{True}, \\
\mathit{consts} & \text{“false”} & [] & = \mathit{BVal}\ \mathit{False}, \\
\mathit{consts} & \mathit{num} & [] & = \mathit{IVal}\ (\mathit{intOfString}\ \mathit{num}), \\
\mathit{consts} & s & [\ldots, \mathit{Top}, \ldots] & = \mathit{Top}, \\
\mathit{consts} & s & xs & = \mathit{Bot}.
\end{array}$$

This definition contains many more clauses than the previous one, because the behaviour of the operators when one or more operand is undefined must be specified.

Some of the clauses in this definition overlap: those higher up the list take precedence. The operators $\cdot$, $-$, and $\geq$ depend on both arguments, so they evaluate to *Bot* if either operand is *Bot*. The extra clause for *if* indicates that it depends on its first parameter, but not necessarily on the second and third: at most one branch is evaluated. The final clause is a *catch-all* corresponding to failure of the function. This might happen if a non-existent constant is called or if the wrong number of parameters is passed. The penultimate clause may seem peculiar though. Firstly, the pattern $[\ldots, \mathit{Top}, \ldots]$ is an informal notation, invented for this example only. It matches any list containing the value *Top*. The sole purpose of this clause is to make *consts* a monotonic function, as required by the Knaster-Tarski theorem. This is a drawback of working with complete lattices, that does not arise when working with complete partial orders, because a complete partial order is not required to have a top element. *Top* is, however, very useful in the specification of strictness analysis.

In §3.2.1, the function *lookup* was used to find values in association lists. If the value does not appear in the list, then *lookup* should return $\bot$. Unfortunately, *lookup* is polymorphic so it does not know the value of $\bot$. This problem can be solved by passing the value of $\bot$ as an extra parameter to *lookup*. A refinement of this solution would be to use Haskell-style *type classes* to pass the extra parameter automatically and invisibly. This idea is discussed further in §3.6.1.

3.4.1 Functional Programming with Non-Flat Domains

An advantage of explicitly defining the partial order for every datatype is that, in principle, orderings can be used that are not permitted in languages such as Haskell. For example, the following program uses sets ordered by inclusion:

$$x :: \mathit{Set\ String},$$
$$x = \{\text{“dog”}\} \cup x.$$

In Haskell, x would evaluate to $\bot$, but the correct (least) solution is the set $\{$“dog”$\}$. It is assumed throughout this dissertation that it possible to implement this solution in a real programming language. Evidence that this assumption is realistic is provided by the many program analysis algorithms that have been written. Such algorithms frequently need to evaluate least fixed points over non-flat domains and the techniques for doing so are well-known. Nielsen, *et al.* [76, pages 363–385] include a chapter on computing least fixed points. Naturally, there are some restrictions because not all least fixed points are computable. The algorithms described by Nielsen, *et al.* require the domains involved to satisfy the *ascending chain condition*. This condition ensures that the algorithms (based on iteration) will terminate. Unfortunately, most function spaces do not satisfy the condition: the domain $(A \leftarrow B,\ \dot{\preceq})$ only satisfies the condition if $(A, \preceq)$ satisfies the condition and B is a *finite* set. So care is required when combining non-flat domains with functions.

3.5 Fixpoint Theorems

The Knaster-Tarski theorem provides two algebraic rules for manipulating fixpoints: the *computation rule* and the *induction rule*. Proofs using the induction rule can be cumbersome, because they often involve mutual inclusion.[4] An alternative approach, advocated by the Eindhoven MPC Group [67], is to use a collection of higher level fixpoint rules. The fixpoint rules, stated below, are powerful enough to prove most fixpoint results without resort to fixpoint induction.

An assumption is made in the theorems below that the fixpoints involved exist. A simple way to guarantee this is by working with complete lattices.

Theorem 2 (Rolling Rule) *If f is a monotonic function of type $B \leftarrow A$ and g is a monotonic function of type $A \leftarrow B$ then:*

$$\mu\ (f \circ g)\ =\ f\ (\mu\ (g \circ f)).$$

The rolling rule can be useful for fusing a function into a fixpoint. It complements the more complicated μ-fusion rule (given below), because it is applicable under different conditions.

The abstraction rule, below, uses a well-known combinator, S, which is defined as follows:

$$\begin{array}{ll} S & :: \ \forall\alpha,\beta,\gamma.\ (\gamma \leftarrow \alpha) \leftarrow (\beta \leftarrow \alpha) \leftarrow (\gamma \leftarrow \beta \leftarrow \alpha), \\ S\ f\ g\ x & =\ f\ x\ (g\ x). \end{array}$$

Theorem 3 (Abstraction Rule) *Suppose f is a function of type $B \leftarrow B \leftarrow A$. If f is monotonic in both its arguments then:*

$$\mu\ (S\ f)\ =\ (\lambda\, x.\ \mu\ (f\ x)).$$

The abstraction rule is also a form of fusion rule. It allows function application to be fused into a fixpoint. To see this, it helps to write the rule as:

$$(\forall x.\ \mu\ (S\ f)\ x\ =\ \mu\ (f\ x)).$$

An example of the use of the abstraction rule is given in §3.5.1.

Theorem 4 (Diagonal Rule) *Suppose f is a function of type $A \leftarrow A \leftarrow A$. If f is monotonic in both its arguments then:*

$$(\mu\, x.\ (\mu\, y.\ f\ x\ y))\ =\ (\mu\, x.\ f\ x\ x).$$

The diagonal rule is useful for collapsing nested fixpoints into one.

Theorem 5 (Mutual Recursion Rule) *Suppose f is a monotonic function of type $(A, B) \leftarrow (A, B)$. Then:*

[4] If $X \leq Y$ and $Y \leq X$ then $X = Y$ by mutual inclusion.

$$
\begin{array}{lcl}
\mu\ f & = & \textbf{let}\ f_1 \quad = fst \circ f, \\
 & & \quad\ f_2 \quad = snd \circ f, \\
 & & \quad\ p\ x\ =\ (\mu\, v.\ f_2\ (x,v)), \\
 & & \quad\ q\ y\ =\ (\mu\, u.\ f_1\ (u,y))\ \textbf{in} \\
 & & ((\mu\, x.\ f_1\ (x,\ p\ x)),\ (\mu\, y.\ f_2\ (q\ y,\ y))).
\end{array}
$$

The mutual recursion rule allows least fixed point computations to be factorised.

Theorem 6 (μ-Fusion) *Suppose f is the lower adjoint in a Galois connection between the domains $(A, \preceq)$ and $(B, \sqsubseteq)$. (Galois connections are discussed in §3.7.) Suppose also that g and h are monotonic functions with types $B \leftarrow B$ and $A \leftarrow A$, respectively. Then:*

$$f\ (\mu\ g)\ =\ \mu\ h\ \Leftarrow\ f \circ g = h \circ f.$$

and

$$f\ (\mu\ g)\ \preceq\ \mu\ h\ \Leftarrow\ f \circ g\ \dot{\preceq}\ h \circ f.$$

The pointwise ordering $\dot{\preceq}$ was defined in §3.3.4. Both versions of this theorem are useful. The first version is more precise than the second, but the second is more often used in abstract interpretation.

3.5.1 An Example: The Abstraction Rule

To illustrate the use of these rules, consider the function *fix*, defined using the notation of Chapter 2:

$$
\begin{array}{l}
fix\ \ \ ::\ A \leftarrow (A \leftarrow A), \\
fix\ f\ =\ f\ (fix\ f).
\end{array}
$$

This definition is based on the computation rule for μ; it is sometimes used by functional programmers to define the fixpoint combinator, using the recursive notation of their language. To prove that $\mu = fix$, note first that the above definition is syntactic sugar for:

$$fix\ =\ (\mu\, Fix.\ (\lambda f.\ f\ (Fix\ f))).$$

The key step is to use the abstraction rule:

$$
\begin{array}{ll}
 & (\mu\, Fix.\ (\lambda f.\ f\ (Fix\ f))) \\
= & \qquad \{\ \text{definitions } S \text{ and } app \text{ (Appendix A.3)}\ \} \\
 & (\mu\, Fix.\ S\ app\ Fix) \\
= & \qquad \{\ \text{use point-free notation}\ \} \\
 & \mu\ (S\ app) \\
= & \qquad \{\ \text{abstraction rule (§3.5)}\ \} \\
 & (\lambda f.\ \mu\ (app\ f))
\end{array}
$$

$$
\begin{aligned}
&= \quad \{ \text{ definition } \mathit{app} \ \} \\
&\quad (\lambda f.\ \mu\ f) \\
&= \quad \{ \text{ use point-free notation } \} \\
&\quad \mu.
\end{aligned}
$$

A point to note about this derivation is that before the application of the abstraction rule, μ is used with type:

$$(A \leftarrow A) \leftarrow ((A \leftarrow A) \leftarrow (A \leftarrow A))$$

Afterwards, its type is simply $A \leftarrow (A \leftarrow A)$. In the first instance, a pointwise ordering is being used, as explained in §3.3.4. Part of the power of the fixpoint theorems is that they can be used to change the domain over which a fixpoint is computed.

3.6 Consequences of Programming with Fixpoints

Programming languages often place restrictions on the use of recursion to ensure that it is implementable. For example, most ML dialects only allow recursion over functions, not values. In contrast, Haskell places almost no restrictions on the use of recursion, but at the expense of requiring lazy evaluation. Restricting the use of recursion corresponds to the semantic restriction that a least fixed point must be over an appropriate domain. Haskell's solution corresponds to implicitly adding a special value $\perp$ to every datatype, to ensure that every datatype is partially ordered. Such implicit values are not assumed in this dissertation, so the use of recursion is only valid when a partial order has been explicitly defined.

This section discusses some of the finer points of fixpoints. It explains that μ is often thought of as a polymorphic function, but that its polymorphic type signature is flawed. Incorrect use of recursion is, therefore, not prevented by the type system. A possible solution to this problem, based on type classes, is discussed.

3.6.1 The Type of μ is Wrong

The type of μ is often thought of as being:

$$\mu :: \forall\alpha.\ \alpha \leftarrow (\alpha \leftarrow \alpha),$$

but, even though μ can be applied to many different types, this polymorphic type is flawed, because μ can only be applied to domains. If the type α does not have a partial order associated with it, then μ is undefined. This problem closely resembles the problem caused by polymorphic equality. (See Wadler [103].[5]) Some functional

[5]Wadler discusses these problems in the specific context of parametricity. Polymorphic equality operators are incompatible with the parametricity theorem and the same is true for μ. This problem is discussed further in §4.4.

languages, such as Miranda [100] and OCaml [65], provide a polymorphic equality operator with type $\forall\alpha.\ \mathit{Bool} \leftarrow \alpha \leftarrow \alpha$. This operator is not valid on all types; it cannot, for example, be applied to functions, because testing functions for equality is undecidable in general. Standard ML (see Paulson [77, page 97]) solves this problem with *equality type variables*. The equality operator is given the type $\forall^{=}\alpha.\ \mathit{Bool} \leftarrow \alpha \leftarrow \alpha$, to indicate that it only works on types that have an equality test. In Haskell, the problem is solved with *type classes*. (See Jones [49] for an introduction.) Equality is assigned the constrained type $\forall\alpha.\ \mathit{Eq}\ \alpha \Rightarrow (\mathit{Bool} \leftarrow \alpha \leftarrow \alpha)$, in which α must be a member of the *Eq* type class. The problem with fixpoints could be solved in a similar way, either by assigning μ the SML-style type:

$$\forall^{\leq}\alpha.\ \alpha \leftarrow (\alpha \leftarrow \alpha),$$

or the Haskell-style type:

$$\forall\alpha.\ \mathit{Fix}\ \alpha \Rightarrow \alpha \leftarrow (\alpha \leftarrow \alpha).$$

These types are intended to indicate that the type α is a domain $(\alpha, \leq)$ in which least fixed points are guaranteed to exist. Such formal notation is not used in this dissertation, but any restrictions on the type variables are always mentioned.

Launchbury and Paterson [61] have investigated the type class approach. They show that such types are simple to implement using existing type class technology. The necessary annotations could therefore be discovered by an automatic type-inference algorithm.

3.7 Galois Connections and Pair Algebras

The fusion theorem, described in §3.5, is important in abstract interpretation, because it changes the domain over which a fixpoint is computed. This is the principle on which abstract interpretation is based: a program is analysed by evaluating it over a simpler domain. If the domain is simple enough, then the analysis can be computed automatically in a finite amount of time. The fusion theorem depends on Galois connections, so Galois connections are equally important in abstract interpretation.

Pair algebras (another name for Galois connections) were proposed by Hartmanis and Stearns [38, 39]. A relation R on the posets $(A, \preceq)$ and $(B, \sqsubseteq)$ is called a Pair Algebra if there are functions $f :: A \leftarrow B$ and $g :: B \leftarrow A$, such that:

$$(\forall x,y.\ (x,y) \in R \equiv f\ x \preceq y),$$

and

$$(\forall x,y.\ (x,y) \in R \equiv x \sqsubseteq g\ y).$$

This relationship between f (the lower adjoint) and g (the upper adjoint) is called a Galois connection between the posets $(A, \preceq)$ and $(B, \sqsubseteq)$. In the literature on abstract interpretation, the relation R is often called the *correctness relation* and f and g are called the *abstraction* and *concretisation* functions, respectively.

Galois connections are an interesting area of mathematics that have turned out to be important in the theory of programming. This dissertation only explores a limited area of the field, so Aart's dissertation [1] is recommended for further reading.

3.7.1 Abstraction and Concretisation Functions

In abstract interpretation, Galois connections are used to relate the *concrete* domain to the *abstract* domain. In the interpreter for the first-order language, the concrete domain is the type *Value*. The abstract domain used in strictness analysis is the Boolean domain: *False* represents *Bot* and *True* represents any other value. Booleans form a complete lattice with implication as the ordering: a is less than or equal to b if $a \Rightarrow b$. (In other words, *False* is less than *True*.) The abstraction function converts values from concrete to abstract:

$$\begin{aligned} &abs && :: Bool \leftarrow Value, \\ &abs\ Bot && = False, \\ &abs\ v && = True. \end{aligned}$$

Information is lost going from concrete to abstract, so the concretisation function has to be conservative, meaning that it computes a value that is greater than or equal to the original value.

$$\begin{aligned} &con && :: Value \leftarrow Bool, \\ &con\ False && = Bot, \\ &con\ True && = Top. \end{aligned}$$

It is a straightforward case analysis to prove that *abs* and *con* form a Galois connection:

$$(\forall v,b.\ \ abs\ v \Rightarrow b \equiv v \preceq con\ b).$$

In the following section, this Galois connection is used to derive a strictness analysis for the first-order language.

3.8 Strictness Analysis

Possibly one of the most successful applications of abstract interpretation has been the strictness analysis of lazy functional programs. This application was discovered by Mycroft [74]. Peyton Jones [79, pages 380-395] gives an excellent introduction to strictness analysis and a discussion of its practical use in compilers. For a more formal treatment, see Burn, *et al.* [18].

The goal of strictness analysis is to predict when a program will compute the value $\perp$. A function f is *strict* if $f \perp = \perp$. A strict function can be compiled to more efficient code than a lazy function because there is no need to wrap its argument in an expensive *thunk*: the argument can, instead, be evaluated before the function is called. If evaluating the argument in advance causes non-termination then the function would not have terminated anyway, so this is a safe optimisation.

Flashing forward to the results of Chapter 5, the derivation of an abstract interpreter that performs strictness analysis on the first order language is almost trivial. Due to its polymorphic type, *evalProg* satisfies the following free theorem:[6]

[6]Wadler [103] introduced the terminology *free* for theorems that are derived from types.

$$\forall p.\ evalProg\ consts\ p \preceq con\ (evalProg\ consts'\ p).$$

provided that *consts* and *consts′* are related as follows:

$$\forall s.\ consts'\ s = abs \circ consts\ s \circ map\ con.$$

The only work that remains to be done is the derivation of *consts′* using this specification. The interpretation of the free theorem is that *evalProg consts* is the concrete interpreter of the language and *evalProg consts′* is the abstract interpreter. The former evaluates the program and produces a *Value*. The latter evaluates the program and produces a *Bool*. The theorem states that the abstract interpreter is a *safe* approximation of the concrete interpreter. This means that the results of the abstract interpreter are never wrong, but they may sometimes be overly conservative. Suppose b is the Boolean value produced by the abstract semantics. If b is *False*, the program will always evaluate to *Bot*. On the other hand, if b is *True* then the program will evaluate to a value less than or equal to *Top*. Every value (including *Bot*) is less than or equal to *Top*, so this statement says nothing about the behaviour of the program. The analysis is *conservative*, because it might compute *True* when the program actually evaluates to *Bot*.

3.8.1 Constants for the Abstract Semantics

Due to the polymorphic design of *evalProg*, the only difference between the concrete and abstract interpreters is the set of constants used. The concrete semantics uses *consts*, which manipulates *Values*, and the abstract semantics uses *consts′*, which manipulates *Booleans*. One of the advantages of abstract interpretation is that it is a methodology for *deriving* analyses, rather than guessing an algorithm and then attempting to prove its correctness. The side condition of the free theorem can be viewed as a specification of *consts′*. Its definition can be derived by expanding the definitions of *consts* (page 16), *abs* and *con*:

$$\begin{array}{lll}
consts' & & :: ConstEnv\ Bool, \\
consts'\ \text{“.”} & [False, y] & = False, \\
consts'\ \text{“.”} & [x, False] & = False, \\
consts'\ \text{“}-\text{”} & [False, y] & = False, \\
consts'\ \text{“}-\text{”} & [x, False] & = False, \\
consts'\ \text{“}\geq\text{”} & [False, y] & = False, \\
consts'\ \text{“}\geq\text{”} & [x, False] & = False, \\
consts'\ \text{“if”} & [False, x, y] & = False, \\
consts'\ \text{“true”} & [] & = True, \\
consts'\ \text{“false”} & [] & = True, \\
consts'\ num & [] & = True, \\
consts'\ s & [\ldots, True, \ldots] & = True, \\
consts'\ s & xs & = False.
\end{array}$$

Evaluating the factorial function (page 14) with these constants produces the result *Top*, as expected. The abstract interpreter also correctly predicts that the nonterminating function (page 15) evaluates to *Bot*.

For the purposes of compiler implementation, the strictness of the functions is more interesting than the strictness of the entire program. As mentioned above, strict functions can be implemented without *thunks*, which makes them more efficient. The function *evalFun* satisfies a similar theorem to *evalProg*, so *evalFun consts′* can be used to analyse the strictness of functions.

3.9 A Comparison with Other Program Analysis Techniques

Abstract interpretation is just one of many approaches to program analysis. "Principles of Program Analysis," a textbook by Nielson, *et al.* [76], is an excellent guide to four of the main approaches to program analysis: data flow analysis, constraint based analysis, abstract interpretation and type and effect systems. The authors successfully highlight the similarities between the different methods: though the methods differ widely in notation, the resulting algorithms differ very little. In particular, each technique is based on the computation of a least fixed point, though some techniques compute the least fixed point by iteration, while others proceed by solving equations. For example, data flow analysis is usually based on iteration, whereas type and effect systems are based on equation solving.

An advantage of abstract interpretation is that program analyses can be *derived* in a systematic way. This can be a mixed blessing though. While the constructive approach sounds attractive, it can lead to dauntingly complex derivations. If polymorphic types are used judiciously in the design of an abstract interpretation, then much of this complexity can be eliminated.

Many computer scientists find data flow and constraint based analysis more intuitive than abstract interpretation. It is often fairly easy to guess the constraint equations for an analysis algorithm. In practice, programmers might, therefore, find it easier to annotate their DSEL with constraints than to derive an abstract implementation. Of course, any analysis should, ideally, be proved correct. Such a correctness proof is no less complicated than the derivation of an abstract interpretation.

It is less clear how type and effect systems could be used in domain specific applications. An important goal is to allow library designers to define their own analyses. Type and Effect systems are based on the idea of adding analyses by enriching the type system. Modifications to the type system can only be made by the language implementers, not by library designers.

3.10 Summary

This chapter introduced abstract interpretation by developing a strictness analyser for a simple first-order functional language. The abstract interpretation approach to this problem is to develop two interpreters for the language: a *concrete* interpreter and an *abstract* interpreter. In anticipation of the results of Chapter 5, both interpreters were defined in terms of one polymorphic interpreter. The interpreter uses recursion,

which motivated a discussion of domains and least fixed points. In fact, the primary difference between the two interpreters is the *domain* over which they compute their results.

An important aspect of the work by Cousot and Cousot is that they use *Galois connections* to formally define the relationship between the concrete and abstract interpreters. There are many reasons why Galois connections are well-suited to this purpose, but least fixed points are a particularly important factor. A Galois connection is required for the fusion theorem (§3.5) to hold and fusion allows the domain over which fixpoints are computed to be changed. The evaluation of programs over simplified semantic domains is the principle on which abstract interpretation is founded.

Chapter 4

The Parametricity Theorem

This chapter is a brief introduction to Reynolds's *abstraction theorem* [86] and *relation algebra*, on which it is based. It is intended as a primer for Chapter 5, which goes into greater depth. The abstraction theorem is a formalisation of what it means for a language to be *parametrically polymorphic*, so it is often known as the *parametricity theorem*. It has also been dubbed *theorems for free* by Wadler [103], because it can be used to derive properties of programs by considering only their type. For example the type of the well-known function *filter* (Appendix A.2) is:

$$\mathit{filter} :: \forall\alpha.\ [\alpha] \leftarrow [\alpha] \leftarrow (\mathit{Bool} \leftarrow \alpha).$$

Just by looking at the type of this function, Wadler deduces the following theorem:

$$\forall f,p.\ \ \mathit{map}\ f \circ \mathit{filter}\ (p \circ f) = \mathit{filter}\ p \circ \mathit{map}\ f.$$

Intuitively, the polymorphic type of *filter* indicates that it does not modify the elements of the list; it can only change the *structure* of the list by rearranging, duplicating or removing elements. Therefore it does not matter whether the function f is mapped over the list before or after *filter* has been applied. Wadler gives many more examples of free theorems for well-known functions such as *fold*, *sort* and *zip*.

Wadler obtains his theorems by instantiating the parametricity theorem according to an automatable procedure. The resulting theorems are less general than the full parametricity result, but much easier to understand and use. Similarly in Chapter 5 an automatable instantiation of the parametricity theorem is proposed. Some generality is lost, but the results are immediately applicable to abstract interpretation.

4.1 Relation Algebra

The parametricity theorem depends heavily on the use of relations, so it cannot be understood without a knowledge of relation algebra. The textbook by Bird and de Moor [15] contains an excellent introduction to this topic and much of the material in this dissertation follows their approach. This section introduces some of the basics and also discusses the relationship between functions and relations.

A relation R of type $A \sim B$ is a set of pairs of type (A, B). Hence, R relates x to y if the pair (x, y) is in the set R:

$$(x, y) \in R.$$

(In these equations, x has type A and y has type B.)

The converse $R^{\cup}$ of a relation R, is defined by:

$$(x, y) \in R^{\cup} \equiv (y, x) \in R$$

If R has type $A \sim B$, then $R^{\cup}$ has type $B \sim A$. The definition of relation composition is:

$$(x,z) \in R \circ S \equiv (\exists y.\ (x,y) \in R \wedge (y,z) \in S).$$

The type of this operator is:

$$(\circ) :: \forall \alpha,\beta,\gamma.\ (\gamma \sim \alpha) \leftarrow ((\gamma \sim \beta),\ (\beta \sim \alpha)).$$

In other words, composition is a *function* that operates on *relations*. The rules concerning such interactions between functions and relations are discussed in §4.1.1.

A function f of type $A \leftarrow B$ is considered to be a special kind of relation:

$$(x,y) \in f \equiv x = f\ y.$$

(The reason why the backwards notation $A \leftarrow B$ is more convenient than $B \rightarrow A$, is that it corresponds more naturally to the order of the variables x and y in this equation.) The following *shunting rules* hold for any function f and any relations R and S:

$$f \circ R \subseteq S \equiv R \subseteq f^{\cup} \circ S \tag{4.1}$$
$$R \circ f^{\cup} \subseteq S \equiv R \subseteq S \circ f \tag{4.2}$$

Proofs of these shunting rules are given by Bird and de Moor [15, page 89].

4.1.1 Functions and Relations

This dissertation never strays into the territory of non-deterministic lambda calculi, even though it may sometimes seem that functions and relations are used in such close proximity that they are interchangeable. Hence it is important to declare the boundaries between functions and relations. It was mentioned above that a function is special case of a relation, but in general a relation is not a function. This means for example that it is not valid to pass a relation as a parameter when a function is expected. However relations are *values* (sets of pairs to be precise), so they can be manipulated by functions. An example of this is the composition operator that is a function from a pair of relations and to a relation. Functions on relations are restricted by the limited set of available operators: relations can be composed or inverted, but never *applied* to a value. Relation application, which produces a *set* of results, is permitted in non-deterministic lambda calculi, but it is not allowed here: this dissertation uses the pure lambda calculus, in which all functions are *total*.

The dual of defining a function on relations (such as composition) is to define a relation on functions. This is precisely how the parametricity theorem formalises the parametric properties of functions.

4.2 Relators

Relators were invented by Backhouse, *et al.* [11] to facilitate the application of the parametricity theorem to arbitrary datatypes. All standard datatypes, such as tuples, lists, and Booleans, are instances of the concept. Strictly speaking, relators are superfluous, because it is well-known that familiar types such as Booleans, pairs, lists, and natural numbers can be defined as types constructed from just $\leftarrow$ and $\forall$. These constructions are often unintuitive though, so relators are a convenient alternative.

A *relator* is a canonical operation corresponding to a particular datatype. For those familiar with category theory, a relator is a *functor* with some additional properties. Following the convention in category theory, a relator is given the same name as its datatype. For instance, the relator corresponding to the *List* datatype is also called *List*. Relators always operate in the category of relations, meaning that relators are mappings from relations to relations. The definition of *List* is:

$$\begin{array}{lcl} \mathit{List} & :: & \forall\alpha,\beta.\ (\mathit{List}\ \alpha \sim \mathit{List}\ \beta) \leftarrow (\alpha \sim \beta), \\ \mathit{List}\ R & = & \{\ ([],[])\ \} \cup \\ & & \quad \{\ (y' : ys',\ y : ys)\ \mid\ (y', y) \in R \wedge (ys', ys) \in \mathit{List}\ R\ \}. \end{array}$$

Two lists are related by *List* R if they have equal length and their elements are related by R. In general, a relator F is a function from a tuple of relations $(R, \ldots, S)$ to a relation $F(R, \ldots, S)$. It satisfies the standard properties of a functor:

- $F(id, \ldots, id) = id$,
- $F(R_1, \ldots, S_1) \circ F(R_2, \ldots, S_2) = F((R_1 \circ R_2), \ldots, (S_1 \circ S_2))$,

but it also satisfies some additional properties:

- It commutes with converse: $(F(R, \ldots, S))^{\cup} = F(R^{\cup}, \ldots, S^{\cup})$.
- It is monotonic: $F(R, \ldots, S) \subseteq F(R', \ldots, S') \Leftarrow R \subseteq R' \wedge \ldots \wedge S \subseteq S'$.
- It distributes through binary intersections: $F(R, \ldots, S) \cap F(R', \ldots, S') = F(R \cap R', \ldots, S \cap S')$.

The final property that relators distribute through binary intersections is non-standard. It is included here to ensure that relators preserve partial orderings.[1] This extra restriction is not problematic, because Hoogendijk [42] has proved that all *regular* datatypes have relators satisfying all of the above properties.

Relators are often generalisations of well-known functions. For example, *List* is a generalisation of the function *map*. That is, if f is a function then *List* f = *map* f. Using the properties listed above, it is straightforward to prove that relators map functions to functions. It follows that relators are functors in the category of functions.

[1] A partial ordering is a relation that is reflexive, anti-symmetric and transitive.

4.2.1 Examples of Relators

To further illustrate relators, three more examples are given. First consider the product relator. The definition of $R \times S$ is:

$$R \times S \;=\; \{\; ((x', y'), (x,y)) \mid (x', x) \in R \wedge (y', y) \in S \;\}.$$

The type of this combinator is:

$$(\times) \;::\; \forall\alpha,\beta,\gamma,\delta.\; ((\alpha,\gamma) \sim (\beta,\delta)) \leftarrow ((\alpha \sim \beta),\; (\gamma \sim \delta)).$$

In other words, two pairs are related if their left components are related and their right components are related. As expected, this definition is a generalisation of a well-known function. If f and g are functions, then the definition of $f \times g$ reduces to:

$$\begin{array}{ll} (\times) & :: \;\forall\alpha,\beta,\gamma,\delta.\; (\alpha,\gamma) \leftarrow (\beta,\delta) \leftarrow ((\alpha \leftarrow \beta),\; (\gamma \leftarrow \delta)), \\ (f \times g)\;(x,y) & =\; (f\;x,\; g\;y). \end{array}$$

Bird [14, page 42] calls this function *cross*.

The second example is the *Either* datatype (see the Haskell Report [78, page 101]):

$$\textbf{data}\ \mathit{Either}\ \alpha\ \beta = \mathit{Left}\ \alpha \mid \mathit{Right}\ \beta.$$

The relator for this datatype is:

$$\begin{array}{lll} \mathit{Either} & :: & \forall\alpha,\beta,\gamma,\delta.\; (\mathit{Either}\ \beta\ \delta \sim \mathit{Either}\ \alpha\ \gamma) \\ & & \qquad \leftarrow (\delta \sim \gamma) \leftarrow (\beta \sim \alpha), \\ \mathit{Either}\ L\ R & = & \{\; (\mathit{Left}\ l',\ \mathit{Left}\ l) \mid (l', l) \in L \;\} \cup \\ & & \quad \{\; (\mathit{Right}\ r',\ \mathit{Right}\ r) \mid (r', r) \in R \;\}. \end{array}$$

In other words, e' and e are related if they are either both *Left* and related by L or both *Right* and related by R. Again, if the relator is applied to functions, the result is a familiar function:

$$\begin{array}{lll} \mathit{Either} & :: & \forall\alpha,\beta,\gamma,\delta.\; \mathit{Either}\ \beta\ \delta \leftarrow \mathit{Either}\ \alpha\ \gamma \leftarrow \\ & & \qquad (\delta \leftarrow \gamma) \leftarrow (\beta \leftarrow \alpha), \\ \mathit{Either}\ f\ g\ (\mathit{Left}\ l) & = & \mathit{Left}\ (f\ l), \\ \mathit{Either}\ f\ g\ (\mathit{Right}\ r) & = & \mathit{Right}\ (g\ r). \end{array}$$

The final example is the nullary relator *Bool*. *Bool* takes no parameters, so its definition is trivial:

$$\begin{array}{l} \mathit{Bool} \;::\; \mathit{Bool} \sim \mathit{Bool}, \\ \mathit{Bool} \;=\; \{\; (b', b) \mid b' = b \;\}. \end{array}$$

All nullary relators are equivalent to the identity relation.

4.3 Reading Types as Relations

Surprisingly, the mathematical properties of polymorphic functions are easiest to state using relations! Reynolds [86] discovered this by studying algebraic approaches in which interpretations are related by the homomorphisms between the algebras. He found that the algebraic approach is intrinsically first-order, but solved the problem by generalising homomorphisms from functions to relations. Wadler's summary of Reynolds's method is that the key to extracting theorems from types is to read types as relations. For example, the type of *filter* can be read as a relation by substituting the relation R for the type variable α:[2]

$$\forall R.\ \mathit{List}\ R \leftarrow \mathit{List}\ R \leftarrow (\mathit{Bool} \leftarrow R).$$

This expression is assigned a meaning by overloading the notation of types. The meaning of $\mathit{List}\ R$ when R is a relation was explained in §4.2. In a similar fashion, the meanings of $\forall X.\ R(X)$ and $R \leftarrow S$ are defined on relations. The definition of $\forall X.\ R(X)$ is relatively straightforward:

$$(x,y) \in (\forall X.\ R\ (X)) \equiv (\forall X.\ (x,y) \in R\ (X)).$$

In other words, x and y are related by $\forall X.R(X)$ if x and y are related by $R(X)$ for any substitution of the relation X in $R(X)$.

Reynolds's reason for generalising from functions to relations stems from his definition of $R \leftarrow S$:

$$(f', f) \in R \leftarrow S \ \equiv\ (\forall x', x.\ (f'\ x',\ f\ x) \in R \Leftarrow (x', x) \in S).$$

This definition is inherently relational because $f \leftarrow g$ is often a relation, even if f and g are both functions. Reynolds's explanation of the definition is that functions are related if they map related arguments into related results. It can also be viewed as a generalised concept of monotonicity. If $f = f'$ it states that f is monotonic:

$$(f,f) \in (\sqsubseteq) \leftarrow (\subseteq) \ \equiv\ (\forall x', x.\ f\ x' \sqsubseteq f\ x \Leftarrow x' \subseteq x).$$

Here, the relations R and S have been written as $\sqsubseteq$ and $\subseteq$, respectively, to emphasise that this is a monotonicity equation.

4.3.1 Theorems for Free

If e has type t and R is the relation corresponding to t, then the parametricity theorem for e is simple to state:

$$(e, e) \in R$$

To illustrate this on a concrete example, the parametricity theorem for *filter* is:

[2] As mentioned in §2.2, the notation $\mathit{List}\ \alpha$ is sometimes used in preference to $[\alpha]$ to prevent overuse of square brackets.

$$(\mathit{filter}, \mathit{filter}) \in (\forall R.\ \mathit{List}\ R \leftarrow \mathit{List}\ R \leftarrow (\mathit{Bool} \leftarrow R)).$$

This property can be transformed into a more familiar form by expanding the definition of the relation:

$$\begin{array}{ll}
& (\mathit{filter}, \mathit{filter}) \in (\forall R.\ \mathit{List}\ R \leftarrow \mathit{List}\ R \leftarrow (\mathit{Bool} \leftarrow R)) \\
\equiv & \quad \{ \text{ definition } \forall X.\ R \ \} \\
& (\forall R.\ (\mathit{filter}, \mathit{filter}) \in \mathit{List}\ R \leftarrow \mathit{List}\ R \leftarrow (\mathit{Bool} \leftarrow R)) \\
\equiv & \quad \{ \text{ definition } \leftarrow \ \} \\
& (\forall R.\ (\forall p', p.\ (\mathit{filter}\ p',\ \mathit{filter}\ p) \in (\mathit{List}\ R \leftarrow \mathit{List}\ R) \\
& \qquad\qquad \Leftarrow (p', p) \in (\mathit{Bool} \leftarrow R))) \\
\equiv & \quad \{ \text{ definition } \leftarrow \ \} \\
& (\forall R.\ (\forall p', p.\ (\forall xs', xs.\ (\mathit{filter}\ p'\ xs',\ \mathit{filter}\ p\ xs) \in \mathit{List}\ R \\
& \qquad\qquad \Leftarrow (xs', xs) \in \mathit{List}\ R) \ \Leftarrow \\
& \qquad\qquad (\forall x', x.\ (p'\ x',\ p\ x) \in \mathit{Bool} \ \Leftarrow \ (x', x) \in R))) \\
\equiv & \quad \{ \mathit{Bool} \text{ is a nullary relator } \} \\
& (\forall R.\ (\forall p', p.\ (\forall xs', xs.\ (\mathit{filter}\ p'\ xs',\ \mathit{filter}\ p\ xs) \in \mathit{List}\ R \\
& \qquad\qquad \Leftarrow (xs', xs) \in \mathit{List}\ R) \ \Leftarrow \\
& \qquad\qquad (\forall x', x.\ p'\ x' = p\ x \ \Leftarrow \ (x', x) \in R))).
\end{array}$$

Unfortunately, this expanded version is hardly more comprehensible than the original statement! Wadler's idea is to consider the special case where R is a function; this often helps to simplify the equations. Substituting f for R:

$$\begin{array}{ll}
& (\forall f.\ (\forall p', p.\ (\forall xs', xs.\ \mathit{filter}\ p'\ xs' = \mathit{List}\ f\ (\mathit{filter}\ p\ xs) \\
& \qquad\qquad \Leftarrow xs' = \mathit{List}\ f\ xs) \ \Leftarrow \\
& \qquad\qquad (\forall x', x.\ p'\ x' = p\ x \ \Leftarrow \ x' = f\ x))) \\
\equiv & \quad \{ \text{ Substitute for } xs' \text{ and } x' \ \} \\
& (\forall f.\ (\forall p', p.\ (\forall xs.\ \mathit{filter}\ p'\ (\mathit{List}\ f\ xs) = \mathit{List}\ f\ (\mathit{filter}\ p\ xs)) \\
& \qquad\qquad \Leftarrow (\forall x.\ p'\ (f\ x) = p\ x))) \\
\equiv & \quad \{ \text{ Use function composition to eliminate } xs \text{ and } x \ \} \\
& (\forall f.\ (\forall p', p.\ \mathit{filter}\ p' \circ \mathit{List}\ f = \mathit{List}\ f \circ \mathit{filter}\ p \ \Leftarrow \ p' \circ f = p)) \\
\equiv & \quad \{ \text{ Substitute for } p \ \} \\
& (\forall f.\ (\forall p'.\ \mathit{filter}\ p' \circ \mathit{List}\ f = \mathit{List}\ f \circ \mathit{filter}\ (p' \circ f))).
\end{array}$$

This much simpler statement is the *theorem for free* given by Wadler.

4.4 Fixpoints and Parametricity

Wadler [103] points out that fixpoints are not compatible with the parametricity theorem. This is due to the reasons discussed in §3.6.1: fixpoints are not fully polymorphic. The parametricity theorem for μ states that:

$$\forall R.\ (\mu, \mu) \in R \leftarrow (R \leftarrow R).$$

To demonstrate that μ does not satisfy this theorem for all R, consider the special case where R is the function f:

$$
\begin{array}{ll}
 & (\mu,\mu) \in f \leftarrow (f \leftarrow f) \\
\equiv & \quad \{ \text{ definition } \leftarrow \} \\
 & (\forall h,k.\ (\mu\ h,\ \mu\ k) \in f \ \Leftarrow\ (\forall x,y.\ (h\ x,\ k\ y) \in f \ \Leftarrow\ (x,y) \in f)) \\
\equiv & \quad \{ \ f \text{ is a function } \} \\
 & (\forall h,k.\ \ \mu\ h = f\ (\mu\ k) \ \Leftarrow\ (\forall x,y.\ \ h\ x = f\ (k\ y) \ \Leftarrow\ x = f\ y)) \\
\equiv & \quad \{ \text{ eliminate } x \ \} \\
 & (\forall h,k.\ \ \mu\ h = f\ (\mu\ k) \ \Leftarrow\ (\forall y.\ \ h\ (f\ y) = f\ (k\ y))) \\
\equiv & \quad \{ \text{ use composition } \} \\
 & (\forall h,k.\ \ \mu\ h = f\ (\mu\ k) \ \Leftarrow\ h \circ f = f \circ k).
\end{array}
$$

This is recognisable as the fusion theorem (§3.5), but the fusion theorem does not hold for an arbitrary function f. (A sufficient condition is that f is a lower adjoint.) So μ does not satisfy its parametricity theorem for all f, let alone for all R. The solution proposed in Chapter 5 is to apply the parametricity theorem to a family of relations based on Galois connections. In this case, μ satisfies its parametricity theorem, so the theorem holds for any function that uses recursion.

4.4.1 Avoiding the Use of Fixpoints

Wadler suggests that one solution to the non-parametricity of fixpoints is to simply not use fixpoints! He promotes the idea of a programming language that only allows structural recursion, because then the parametricity theorem would apply in its full generality. This solution is no good here, because many of the examples in this dissertation need general recursion. There is another way to circumvent the problem though: *pass the fixpoint operator as an external parameter*. Using an example from Chapter 3, the definition of *evalProg* (§3.2.2) could be changed to:[3]

$$
\begin{array}{l}
\mathit{evalProg} :: \forall v.\ v \leftarrow \mathit{Prog} \leftarrow \mathit{ConstEnv}\ v \leftarrow \mathit{Fix}\ (\mathit{FunEnv}\ v) \\
\mathit{evalProg}\ \mathit{fix}\ \mathit{constEnv}\ (\mathit{fs}, e) = \\
\quad \textbf{let}\ \ \mathit{funEnv} = \mathit{fix}\ (\lambda\, \mathit{env}.\ \mathit{lookup}\ (\mathit{map}\ (\mathit{evalFun}\ \mathit{constEnv} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{env})\ \mathit{fs}))\ \ \textbf{in} \\
\quad \mathit{evalExp}\ (\mathit{constEnv}, \mathit{funEnv}, [])\ e.
\end{array}
$$

To evaluate a program, μ must now be passed as an extra parameter to *evalProg*. Factoring out the fixpoint in this way makes *evalProg* structurally recursive, so it is fully parametric. The new free theorem for *evalProg* has an extra side condition though: it states that *fix* must satisfy the following property:

$$(\mathit{fix}, \mathit{fix}) \in \mathit{Fix}\ (\mathit{FunEnv}\ R).$$

[3]The type *Fix* is a synonym: $\textbf{type}\ \mathit{Fix}\ \alpha = \alpha \leftarrow (\alpha \leftarrow \alpha)$.

This side-condition expands to a complicated property that must be proved, so nothing is gained by factoring out the fixpoint. If the type *FunEnv* was not a function type, then the approach might have been more beneficial. Chapter 8 discusses an example where the side-condition on *fix* turns out to be equivalent to the fusion theorem. This allows a more precise abstract interpretation result to be proved.

4.5 Summary

The parametricity theorem is an extremely powerful property of parametrically polymorphic type systems to which this chapter has given a brief introduction. Wadler popularised the parametricity theorem by demonstrating that it can be used to derive useful theorems about polymorphic programs *for free*. Wadler's theorems do not represent the full power of Reynolds's result though, as they are obtained by *specialisation*: Wadler replaces relations by functions to simplify the equations. The advantage of Wadler's theorems is that they are easier to comprehend. This was illustrated in §4.3.1 by comparing Wadler's free theorem for *filter* with its full parametric property. Chapter 5 develops a different specialisation of Reynolds's theorem by replacing relations with *pair algebras*. This leads to a different set of free theorems which are applicable to abstract interpretation. An important benefit of this specialisation is that it solves the problem caused by fixpoints; μ is not compatible with parametricity, but it is compatible with the restricted version introduced in Chapter 5.

Chapter 5

Abstract Interpretation for Free

In this chapter a theory is developed that allows abstract interpretations to be derived for free from the types of programs. The idea is to instantiate the parametricity theorem with pair algebras (Galois connections). The goal of this chapter is to emulate Wadler [103] and develop a theory that is intuitive and easy to use, despite the heavy theoretical foundations upon which it is built. Wadler made parametricity intuitive by substituting functions for relations; the free theorems developed here are different from Wadler's, but the same idea of simplifying the results by substituting functions for relations is applied.

The material in this chapter is joint work with R. Backhouse and was presented at MPC 2002 [10]. It builds upon the work of Mycroft and Jones [75] and particularly that of Abramsky [3], who have shown how to use logical relations to construct abstract interpretations. The main contribution here is to specialise Abramsky's theorems to Galois connections. Although the resulting framework is less general than his, it satisfies a number of simple laws that make it easy to work with.

Logical relations and parametricity are closely related topics, so this chapter begins by explaining the connection. Logical relations are then specialised to *logical pair algebras*. This specialisation reduces the generality of the parametricity theorem, but it has an important benefit: it makes the parametricity theorem applicable to programs that use general recursion.

5.1 Logical Relations

As explained in §4.3, Reynolds's abstraction theorem [86] is based on reading types as relations. For example, the type of *filter* is read as the relation:

$$\forall R.\ List\ R \leftarrow List\ R \leftarrow (Bool \leftarrow R).$$

The reading of the type of *filter* as a relation is an instance of a *Logical Relation*. Logical relations were introduced by Plotkin [83] and Reynolds [86] in order to reason about the polymorphic lambda calculus and are integral to the parametricity theorem. A logical relation is family of relations, indexed by types. In other words, if V is a

logical relation then V_t is a relation for all types t. Using the definition of $R \leftarrow S$ given in §4.3, a logical relation V should satisfy for all types u and v:

$$V_{u \leftarrow v} = V_u \leftarrow V_v. \tag{5.1}$$

Similarly, if F is a relator, V should satisfy:

$$V_{F(u \ldots v)} = F(V_u \ldots V_v). \tag{5.2}$$

There is also a rule concerning types of the form $\forall\alpha.\sigma$, but there is no need for it when using Milner's type system for ML [68] (which is the type system used in this dissertation), because nested quantifiers are not allowed.

5.1.1 Relation Assignments

Rather than as a family of relations, a logical relation can also be understood as a substitution operation. Suppose V is a relation assignment. That is, it is a mapping from type variables to relations, such as:

$$V = (\alpha \mapsto R, \beta \mapsto S).$$

Then V_t denotes the application of the substitution V to the type t. This is done by recursively applying the rules (5.1) and (5.2) and replacing every occurrence of a type variable x by the relation V_x. Note that if the type t contains a free type variable that is not in the domain of V, then V_t is undefined. This side-condition on the domain of V is implicitly assumed throughout this chapter.

The inductive substitution process can also be used to substitute partially ordered sets (posets) for type variables. A poset assignment is a mapping from type variables to posets, such as:

$$V = (\alpha \mapsto (A, \preceq), \beta \mapsto (B, \sqsubseteq)).$$

Again, (5.1) and (5.2) can be used to extend V to an arbitrary type t. That the result V_t is also a poset is proved in §5.2.2. Many of the results in this chapter rely on monotonicity, so the poset $(A, \preceq) \leftarrow (B, \sqsubseteq)$ is restricted, by definition, to the set of *monotonic* functions of type $A \leftarrow B$.

Poset assignments are useful for defining the types of relation assignments. If V_a has type $A_a \sim B_a$ for all a then V_t has type $A_t \sim B_t$ for all t.

Converse and Composition of Relation Assignments

It is convenient to define composition and converse on relation assignments. Firstly, the converse $V^\cup$ of a relation assignment is defined by:

$$(V^\cup)_a = (V_a)^\cup, \tag{5.3}$$

Secondly the composition $V \circ W$ of two relation assignments is defined by:

$$(V \circ W)_a = V_a \circ W_a. \tag{5.4}$$

Please note that these definitions only apply when a is a simple type variable, that is in the domain of both V and W. It can be proved that (5.3) extends to non-trivial types, but the same is not true for (5.4).

5.1.2 The Parametricity Theorem, Using Logical Relations

Using logical relations, the parametricity theorem can be stated very concisely. Suppose e is a value with type $\forall\alpha_1 \ldots \alpha_n.t$, where t is a monotype.[1] Then for any logical relation V:

$$(e, e) \in V_t.$$

If, as explained above, the logical relation is constructed from a relation assignment, then the relation assignment should be of the form:

$$V = (\alpha_1 \mapsto R_1, \ldots, \alpha_n \mapsto R_n).$$

Using this notation, the parametricity theorem for *filter* can be restated as:

$$(\mathit{filter}, \mathit{filter}) \quad \in \quad (\alpha \mapsto R)_{\mathit{List}\,\alpha \,\leftarrow\, \mathit{List}\,\alpha \,\leftarrow\, (\mathit{Bool} \,\leftarrow\, \alpha)}.$$

As before, this property holds for all R.

5.1.3 A Binary Logical Operator

It is often useful to distinguish between *positive* and *negative* occurrences of type variables. An occurrence of a type variable is positive if it is the result type of a function and negative if it is the argument type. For example, here are some types annotated with signs:

$$\begin{array}{l}
\alpha^+ \leftarrow \beta^-, \\
\alpha^+ \leftarrow (\beta^- \leftarrow \gamma^+), \\
\alpha^+ \leftarrow (\beta^- \leftarrow \gamma^+) \leftarrow (\beta^- \leftarrow \gamma^+), \\
\mathit{List}\,\alpha^+ \leftarrow \mathit{List}\,\alpha^- \leftarrow (\mathit{Bool} \leftarrow \alpha^+).
\end{array}$$

Note that there is a double-negation rule, so γ is positive in these examples.

The binary logical operator $[V, W]_t$ applies V to the positive type variables of t and W to the negative type variables of t . It is defined inductively as follows:

$$\begin{array}{rcll}
[V, W]_a & = & V_a, & (5.5) \\
[V, W]_{u \leftarrow v} & = & [V, W]_u \leftarrow [W, V]_v, & (5.6) \\
[V, W]_{F(u \ldots v)} & = & F([V, W]_u \ldots [V, W]_v). & (5.7)
\end{array}$$

To illustrate its behaviour, it can be used to substitute R for the positive and S for the negative occurrences of α in the type of *filter*:

$$\begin{array}{l}
\quad [(\alpha \mapsto R), (\alpha \mapsto S)]_{\mathit{List}\,\alpha \,\leftarrow\, \mathit{List}\,\alpha \,\leftarrow\, (\mathit{Bool} \,\leftarrow\, \alpha)} \\
= \\
\quad \mathit{List}\,R \;\leftarrow\; \mathit{List}\,S \;\leftarrow\; (\mathit{Bool} \;\leftarrow\; R).
\end{array}$$

[1] In Milner's type system for ML [68], all types have this form. See Figure 2.2, page 9.

The primary motivation for introducing this binary operator is that it can be used to construct functions (as opposed to relations), which is not possible with standard logical relations. This is done by replacing every positive type variable with a function and every negative type variable with the inverse of a function.

As with standard logical relations, the binary operator works equally with on poset assignments. If for all a, V_a has type $A_a \sim B_a$ and W_a has type $C_a \sim D_a$ then $[V, W]_t$ has type $[A, C]_t \sim [B, D]_t$ for all t.

5.2 Basic Properties

This section covers some basic properties of the operators introduced above. These properties are useful for proving the main results of the chapter. Throughout the chapter, relation algebra is frequently conducted using point-free notation. For example, the defining equations of a pair algebra are written as:

$$R = f^\cup \circ \preceq \quad \wedge \quad R = \sqsubseteq \circ g,$$

rather than the more verbose:

$$(\forall x,y.\ (x,y) \in R \ \equiv\ f\ x\ \preceq\ y),$$

and

$$(\forall x,y.\ (x,y) \in R \ \equiv\ x\ \sqsubseteq\ g\ y).$$

The concise point-free notation makes the theorems easier to state and prove.

5.2.1 Properties of the Arrow Operator

The definition of $R \leftarrow S$ was given in §4.3 when the parametricity theorem was first discussed, but it is repeated here for convenience:

$$(f', f) \in R \leftarrow S \ \equiv\ (\forall x', x.\ (f'\ x',\ f\ x)\ \in\ R\ \Leftarrow\ (x', x) \in S).$$

A number of properties can be easily derived from this definition. If f and g are (monotonic) functions, then $f \leftarrow g^\cup$ is a (monotonic) function:

$$f \leftarrow g^\cup = (\lambda h.\ f \circ h \circ g). \tag{5.8}$$

Converse distributes through the arrow operator:

$$(R \leftarrow S)^\cup = R^\cup \leftarrow S^\cup. \tag{5.9}$$

Composition also distributes through the arrow operator, provided that some of the relations involved are functions:

$$(R \circ f) \leftarrow (S \circ g^\cup) \ =\ (R \leftarrow S) \circ (f \leftarrow g^\cup), \tag{5.10}$$

$$(f^\cup \circ R) \leftarrow (g \circ S) \ =\ (f^\cup \leftarrow g) \circ (R \leftarrow S). \tag{5.11}$$

Arrow preserves identity:

$$id \leftarrow id = id. \tag{5.12}$$

If the arrow operator is restricted to *monotonic* functions, then its definition becomes much simpler. If f and f' are monotonic, then:

$$(f, f') \ \in \ \preceq \leftarrow \sqsubseteq \quad \equiv \quad (\forall x.\ f\ x \preceq f'\ x).$$

In other words the ordering $\preceq \leftarrow \sqsubseteq$ (on monotonic functions) is equivalent to the pointwise ordering $\dot{\preceq}$ discussed in §3.3.4. This fact is important for making the results of this chapter easy to apply in practical situations. In §5.4.1, the result is extended to arbitrary logical relations of the form $\preceq_t$.

5.2.2 Properties of Logical Relations

This section introduces the basic properties of logical relations. The most important properties here are:

1. Lemma 1: $[f, g^{\cup}]_t$ is a function.
2. Corollary 1: if $(A_a, \preceq_a)$ is a poset for all a then $(A_t, \preceq_t)$ is a poset.

The first property, Lemma 1, is one of the main motivations for introducing the binary operator $[\_,\_]$. The second property, Corollary 1, is very important and has already been mentioned in §3.3.4 and §5.1.1.

The binary operator $[\_,\_]$ is a generalisation of standard logical relations because:

$$V_t = [V, V]_t. \tag{5.13}$$

The proof is a simple induction over the structure of type expressions. By a similar proof, it is straightforward to show that:

$$(V_t)^{\cup} \ = \ (V^{\cup})_t, \tag{5.14}$$

$$([V, W]_t)^{\cup} \ = \ [V^{\cup}, W^{\cup}]_t. \tag{5.15}$$

These rules justify the notation $V_t^{\cup}$ and $[V, W]_t^{\cup}$, in which the parentheses have been omitted.

Property (5.8) states that $f \leftarrow g^{\cup}$ is a function. This result can be generalised:

Lemma 1 *Suppose f and g are type assignments such that (for all type variables a) f_a and g_a are functions of type $A_a \leftarrow B_a$ and $C_a \leftarrow D_a$, respectively. Then, for all type expressions t, $[f, g^{\cup}]_t$ is a function of type $[A, D]_t \leftarrow [B, C]_t$.*

The proof of this Lemma is a straightforward induction using (5.8) and the preservation of functions by relators. This Lemma is one of the main motivations for introducing the operator $[V, W]$, because logical relations of the form f_t are not functions in general. Properties that can be expressed using functions are usually much easier to comprehend than those that use relations.

Lemma 1 does not mention monotonicity, because this is covered separately by Lemma 2 below.

Lemma 2 *Suppose f and g are type assignments such that (for all type variables a) f_a and g_a are monotonic functions of type $(A_a, \leq_a) \leftarrow (B_a, \sqsubseteq_a)$ and $(C_a, \preceq_a) \leftarrow (D_a, \trianglelefteq_a)$, respectively. Then, for all type expressions t, $[f, g^{\cup}]_t$ is a monotonic function of type $([A, D]_t, [\leq, \trianglelefteq]_t) \leftarrow ([B, C]_t, [\sqsubseteq, \preceq]_t)$.*

The proof of this Lemma is a rather neat application of the parametricity theorem. The types in Lemma 1 indicate that $[\_, (\_)^{\cup}]$ is a polymorphic operator: it works for function assignments f and g of any type. So the parametricity theorem states that for all relations R, S, T and V:

$$\begin{array}{l} \quad ([f, g^{\cup}]_t,\ [f, g^{\cup}]_t)\ \in\ [R, U]_t \leftarrow [S, T]_t \\ \Leftarrow \\ \quad (\forall a.\ (f_a, f_a) \in R_a \leftarrow S_a\ \wedge\ (g_a, g_a) \in T_a \leftarrow U_a). \end{array}$$

But as was said in §4.3, the statement that $(f, f) \in R \leftarrow S$ is equivalent to the statement that f is monotonic, because by definition:

$$(f,f) \in (\sqsubseteq) \leftarrow (\subseteq)\ \equiv\ (\forall x', x.\ f\ x' \sqsubseteq f\ x\ \Leftarrow\ x' \subseteq x).$$

So substituting $R := \leq$, $S := \sqsubseteq$, $T := \preceq$ and $U := \trianglelefteq$ proves Lemma 2. That $[\leq, \trianglelefteq]_t$ and $[\sqsubseteq, \preceq]_t$ are partial orders is established in Lemma 5.

Lemma 3 *If f, g, h and k are defined on all the free variables in t, then:*

$$[f, g^{\cup}]_t \circ [h, k^{\cup}]_t = [f \circ h, (k \circ g)^{\cup}]_t.$$

(Note the contravariance in the second argument.)

Proof. Straightforward induction on the structure of type expressions. The basis follows immediately from (5.5) and the induction step uses (5.10) and (5.7).

Lemma 4 *If $(A, \preceq) \ldots (B, \sqsubseteq)$ are posets and F is a relator, then $(F(A \ldots B), F(\preceq \ldots \sqsubseteq))$ is a poset.*

Proof. It is easy to verify that transitivity and reflexivity of $F(\preceq \ldots \sqsubseteq)$ follow from transitivity and reflexivity of $\preceq \ldots \sqsubseteq$. In point-free notation, anti-symmetry of $F(\preceq \ldots \sqsubseteq)$ is the property that:

$$F(\preceq \ldots \sqsubseteq) \cap (F(\preceq \ldots \sqsubseteq))^{\cup}\ \subseteq\ F(id \ldots id).$$

This follows from the anti-symmetry of $\preceq \ldots \sqsubseteq$ and that F commutes with converse and is monotonic, and the assumption that F preserves binary intersections.

The assumption that F preserves binary intersections was made in §4.2 when relators were introduced. It is not a standard property of relators, but it is necessary for this Lemma to hold. Fortunately, Hoogendijk [42] has proved that all *regular* datatypes satisfy this property, so it is a reasonable assumption to make in the context of modern programming languages. Sadly, complete lattices do not always lift though relators in the same way that posets do. For example, the poset $([A], \mathit{List}(\preceq))$ does not have a top or bottom element. This means that relators do not always make suitable domains for the computation of least fixed points.

Lemma 5 *If* $(A_a, \preceq_a)$ *and* $(B_a, \sqsubseteq_a)$ *are posets for all type variables* a*, then* $([A, B]_t, [\preceq, \sqsubseteq]_t)$ *is a poset for all types* t*.*

The proof of this lemma is a simple induction over the structure of types, using Lemma 4. The following important corollary follows from V_t being equal to $[V, V]_t$:

Corollary 1 *If* $(A_a, \preceq_a)$ *is a poset for all type variables* a *then* $(A_t, \preceq_t)$ *is a poset for all types* t*.*

5.3 Logical Pair Algebras

In this section, the main result of this chapter is proved. It is Theorem 7, which states that:

> If R_a is a pair algebra for all type variables a, then R_t is a pair algebra for all types t.

As explained in §3.7, a pair algebra is the relation corresponding to a Galois connection. Theorem 7 also defines the upper and lower adjoints of the Galois connection using the binary operator $[\_,\_]$. This means that when the parametricity theorem is instantiated with the logical relation R in §5.4, a concise safety property drops out.

The crucial result which makes the proof of Theorem 7 possible is Abramsky's *uniformization theorem* [3, Proposition 6.4]. It is stated below as Lemma 7, but specialised to Galois connections. (Abramsky does not assume that the function g has an upper adjoint.) First a preliminary lemma is needed:

Lemma 6 *For all partial orderings* $\sqsubseteq$ *and* $\preceq$ *and all functions* g*:*

$$\sqsubseteq \leftarrow (\preceq \circ g) \;=\; \sqsubseteq \leftarrow g,$$

and

$$\sqsubseteq \leftarrow (g^{\cup} \circ \preceq) \;=\; \sqsubseteq \leftarrow g^{\cup}.$$

Proof. For all monotonic functions h and k:

$$\begin{array}{ll} & (h, k) \;\in\; \sqsubseteq \leftarrow (\preceq \circ g) \\ = & \qquad \{ \text{ Definition of } \leftarrow \} \\ & (\forall u, v.\; h\, u \sqsubseteq k\, v \Leftarrow u \preceq g\, v) \\ = & \qquad \{ \; (\Rightarrow) \text{ reflexivity of } \preceq \\ & \qquad\quad (\Leftarrow) \; h \text{ is monotonic, transitivity of } \sqsubseteq \; \} \\ & (\forall v.\; h\,(g\, v) \sqsubseteq k\, v) \\ = & \qquad \{ \text{ Definition of } \leftarrow \} \\ & (h, k) \;\in\; \sqsubseteq \leftarrow g. \end{array}$$

The second claim is proved similarly. (This is where monotonicity of k is used.)

Lemma 6 is often used in combination with the distributivity property (5.10). Its most immediate application is that the ordering $\preceq \leftarrow \sqsubseteq$ on monotonic functions is equivalent to $\preceq \leftarrow id$, which is equivalent to the pointwise ordering $\dot{\preceq}$, defined in §3.3.4.

Lemma 7 (Uniformization) *If f is a monotonic function with range $(A, \sqsubseteq)$ and the pair $(g, g^\sharp)$ is a Galois connection between the posets $(B, \preceq)$ and $(C, \leq)$ then*

$$(f^\cup \circ \sqsubseteq) \leftarrow (g^\cup \circ \preceq) \;=\; (f^\cup \leftarrow g^\sharp) \circ (\sqsubseteq \leftarrow \preceq).$$

Also,

$$(\sqsubseteq \circ f) \leftarrow (\leq \circ g^\sharp) \;=\; (\sqsubseteq \leftarrow \leq) \circ (f \leftarrow g^\cup).$$

Proof. That $(g, g^\sharp)$ is a Galois connection means that $g^\cup \circ \preceq \;=\; \leq \circ g^\sharp$. Therefore:

$$\begin{array}{ll} & (f^\cup \circ \sqsubseteq) \leftarrow (g^\cup \circ \preceq) \\ = & \qquad \{\ (g, g^\sharp) \text{ is a Galois connection } \} \\ & (f^\cup \circ \sqsubseteq) \leftarrow (\leq \circ g^\sharp) \\ = & \qquad \{\ \text{Distributivity (5.11)} \ \} \\ & (f^\cup \leftarrow id_C) \circ (\sqsubseteq \leftarrow (\leq \circ g^\sharp)) \\ = & \qquad \{\ \text{Lemma 6} \ \} \\ & (f^\cup \leftarrow id_C) \circ (\sqsubseteq \leftarrow g^\sharp) \\ = & \qquad \{\ \text{Distributivity (5.11)} \ \} \\ & (f^\cup \circ \sqsubseteq) \leftarrow g^\sharp \\ = & \qquad \{\ \text{Distributivity (5.11)} \ \} \\ & (f^\cup \leftarrow g^\sharp) \circ (\sqsubseteq \leftarrow id_B) \\ = & \qquad \{\ \text{Lemma 6} \ \} \\ & (f^\cup \leftarrow g^\sharp) \circ (\sqsubseteq \leftarrow \preceq). \end{array}$$

The second property is proved similarly.

Uniformization is now used to establish that the upper and lower adjoints of the new Galois connection are defined by the operator $[\_,\_]$.

Lemma 8 *Suppose that for each type variable a, f_a and g_a form a Galois connection between the posets $(A_a, \sqsubseteq_a)$ and $(B_a, \preceq_a)$. Suppose also that h_a and k_a form a Galois connection between the posets $(C_a, \trianglelefteq_a)$ and $(D_a, \leq_a)$. Then, for all types t,*

$$[f, k^\cup]_t^\cup \circ [\sqsubseteq, \trianglelefteq]_t \;=\; [\preceq, \leq]_t \circ [g, h^\cup]_t.$$

That is, the pair of functions $([f, k^\cup]_t, [g, h^\cup]_t)$ forms a Galois connection between the posets $([A, C]_t, [\sqsubseteq, \trianglelefteq]_t)$ and $([B, D]_t, [\preceq, \leq]_t)$.

Proof. Lemma 1 (page 27) establishes that $[f, k^\cup]_t$ and $[g, h^\cup]_t$ are functions (of the right type) and Lemma 5 that $[\sqsubseteq, \trianglelefteq]_t$ and $[\preceq, \leq]_t$ are partial ordering relations. So it suffices to prove the equality. This is done by induction on types. The induction hypothesis is the equality stated above, together with the symmetric equality:

$$[h, g^\cup]_t^\cup \circ [\trianglelefteq, \sqsubseteq]_t \quad = \quad [\leq, \preceq]_t \circ [k, f^\cup]_t.$$

For all type variables a, the pair of functions (f_a, g_a) is a Galois connection between the posets $(A_a, \sqsubseteq_a)$ and $(B_a, \preceq_a)$, so:

$$f_a^\cup \circ \sqsubseteq_a \quad = \quad \preceq_a \circ g_a.$$

Similarly:

$$h_a^\cup \circ \trianglelefteq_a \quad = \quad \leq_a \circ k_a.$$

Combined with (5.5), these establish the basis of the proof.

For the case $t = u \leftarrow v$ uniformization is used. This depends on $[h, g^\cup]_v$ and $[k, f^\cup]_v$ forming a Galois connection, which is true by the induction hypothesis.

$$\begin{array}{ll}
 & [f, k^\cup]_t^\cup \circ [\sqsubseteq, \trianglelefteq]_t \\
= & \qquad \{\ t = u \leftarrow v, \text{ definition (5.6)}\ \} \\
 & ([f, k^\cup]_u \leftarrow [k^\cup, f]_v)^\cup \circ ([\sqsubseteq, \trianglelefteq]_u \leftarrow [\trianglelefteq, \sqsubseteq]_v) \\
= & \qquad \{\ \text{converse: (5.9) and (5.15)}\ \} \\
 & ([f, k^\cup]_u^\cup \leftarrow [k, f^\cup]_v) \circ ([\sqsubseteq, \trianglelefteq]_u \leftarrow [\trianglelefteq, \sqsubseteq]_v) \\
= & \qquad \{\ \text{uniformization (Lemma 7), induction hypothesis}\ \} \\
 & [f, k^\cup]_u^\cup \circ [\sqsubseteq, \trianglelefteq]_u \leftarrow [h, g^\cup]_v^\cup \circ [\trianglelefteq, \sqsubseteq]_v \\
= & \qquad \{\ \text{induction hypothesis}\ \} \\
 & [\preceq, \leq]_u \circ [g, h^\cup]_u \leftarrow [\leq, \preceq]_v \circ [k, f^\cup]_v \\
= & \qquad \{\ \text{uniformization (Lemma 7), induction hypothesis}\ \} \\
 & ([\preceq, \leq]_u \leftarrow [\leq, \preceq]_v) \circ ([g, h^\cup]_u \leftarrow [h, g^\cup]_v^\cup) \\
= & \qquad \{\ \text{converse (5.15)}, t = u \leftarrow v, \text{ definition (5.6)}\ \} \\
 & [\preceq, \leq]_t \circ [g, h^\cup]_t.
\end{array}$$

The proof of the second equality is completely symmetric.

Finally, the case $t = F(u \ldots v)$ is a straightforward application of the distributivity of relators through composition and their commutativity with converse.

Theorem 7 (Logical Pair Algebras) *Suppose for all type variables a, the pair of functions (abs_a, con_a) forms a Galois connection between the posets $(A_a, \sqsubseteq_a)$ and $(B_a, \preceq_a)$ and R_a is the corresponding pair algebra. That is:*

$$R_a = abs_a^\cup \circ \sqsubseteq_a \quad \wedge \quad R_a = \preceq_a \circ con_a. \tag{5.16}$$

For all types t,

$$R_t = [abs, con^\cup]_t^\cup \circ \sqsubseteq_t \quad \wedge \quad R_t = \preceq_t \circ [con, abs^\cup]_t.$$

That is, the logical relation R defines a pair algebra for all types t between the posets $(A_t, \sqsubseteq_t)$ and $(B_t, \preceq_t)$. The corresponding lower and upper adjoints are $[abs, con^{\cup}]_t$ and $[con, abs^{\cup}]_t$, respectively.

Proof. Lemma 8 establishes the Galois connection. (Take $f = h = abs$, $g = k = con$, $\sqsubseteq = \trianglelefteq$ and $\preceq = \leq$.) All that remains is to establish that R_t is the corresponding pair algebra. The proof is by induction on the structure of types. The base case follows from assumption (5.16). For $t = u \leftarrow v$:

$$
\begin{aligned}
& [abs, con^{\cup}]_t^{\cup} \circ \sqsubseteq_t \\
= & \qquad \{\ t = u \leftarrow v,\ (5.1) \text{ and } (5.6)\ \} \\
& ([abs, con^{\cup}]_u \leftarrow [con^{\cup}, abs]_v)^{\cup} \circ (\sqsubseteq_u \leftarrow \sqsubseteq_v) \\
= & \qquad \{\ \text{converse: } (5.9) \text{ and } (5.15)\ \} \\
& ([abs, con^{\cup}]_u^{\cup} \leftarrow [con, abs^{\cup}]_v) \circ (\sqsubseteq_u \leftarrow \sqsubseteq_v) \\
= & \qquad \{\ \text{uniformization (Lemma 7), induction hypothesis}\ \} \\
& [abs, con^{\cup}]_u^{\cup} \circ \sqsubseteq_u \leftarrow [abs, con^{\cup}]_v^{\cup} \circ \sqsubseteq_v \\
= & \qquad \{\ \text{induction hypothesis, } t = u \leftarrow v,\ (5.1)\ \} \\
& R_t.
\end{aligned}
$$

Finally, the case $t = F(u \ldots v)$ is a straightforward application of the distributivity of relators through composition and their commutativity with converse.

In their chapter on abstract interpretation, Nielson, *et al.* [76, pages 244-256] discuss the *Systematic Design of Galois connections*. Given a set of base Galois connections, they explain how to construct new Galois connections. For example, they discuss the sequential composition of Galois connections, pairs of Galois connections and Galois connections over monotone function spaces. Most of the examples that they discuss are instances of Theorem 7.

5.4 Safety for Free

Instantiating the parametricity theorem with the logical pair algebras of Theorem 7 leads to a concise theorem that expresses the safety property of abstract interpretation.

Theorem 8 (Safety for Free) *Suppose e is a value with polymorphic type $\forall \alpha_1 \ldots \alpha_n.t$, where t is a monotype. Suppose also that for all type variables a in $\alpha_1 \ldots \alpha_n$, the pair of functions (abs_a, con_a) forms a Galois connection between the posets $(A_a, \sqsubseteq_a)$ and $(B_a, \preceq_a)$. Then:*

$$e \ \preceq_t \ [con, abs^{\cup}]_t \ e, \tag{5.17}$$

$$[abs, con^{\cup}]_t \ e \ \sqsubseteq_t \ e. \tag{5.18}$$

Proof. If the logical relation R is defined as in Theorem 7, then the parametricity theorem states simply that:

$$(e, e) \in R_t.$$

Therefore, (5.17) and (5.18) follow immediately from Theorem 7.

In the equations above, e is instantiated with different types on the left and right hand sides of the equations. On the left hand side, e has type t, with $B_{\alpha_1} \dots B_{\alpha_n}$ substituted for $\alpha_1 \dots \alpha_n$, whereas on the right hand side e has type t, with $A_{\alpha_1} \dots A_{\alpha_n}$ substituted for $\alpha_1 \dots \alpha_n$. In other words, the left hand e operates in the concrete domain and the right hand e operates in the abstract domain. Of the two equations, (5.17) tends to be the more useful, because it predicts the value of e in the concrete domain, using the value of e in the abstract domain, whereas (5.18) does the opposite.

In order to make Theorem 8 theorem usable, two more facts are needed. Firstly, Lemma 9 shows that the orderings used in Theorem 8 are equivalent to pointwise orderings. Secondly, Lemma 10 shows that programs that use general recursion also satisfy Theorem 8.

5.4.1 Higher Order Pointwise Orderings

The simple pointwise ordering $\dot{\preceq}$ was discussed in §3.3.4 and on page 26 it was noted that it is equivalent to the ordering $\preceq \leftarrow \sqsubseteq$ on monotonic functions. Pointwise orderings can be generalised to higher-order types:

$$\dot{\preceq}_a = \preceq_a, \tag{5.19}$$

$$\dot{\preceq}_{u \leftarrow v} = \dot{\preceq}_u \leftarrow id_v, \tag{5.20}$$

$$\dot{\preceq}_{F(u \dots v)} = F(\dot{\preceq}_u \dots \dot{\preceq}_v). \tag{5.21}$$

Equations that use the ordering $\dot{\preceq}_t$ expand to a much simpler form than those that use $\preceq_t$, but they are in fact equivalent:

Lemma 9 (Higher Order Pointwise Orderings) *If $(A_a, \preceq_a)$ is a poset for each type variable a, then for all types t:*

$$\dot{\preceq}_t = \preceq_t.$$

Proof. The proof is a straightforward induction on the structure of type expressions. Lemma 6 provides the crucial induction step.

5.4.2 Least Fixed Points

The fusion theorem (Theorem 6) is one of the main motivations for introducing Galois connections. Abstract interpretation relies heavily on fusion, because it enables the domain over which a fixpoint is computed to be changed. Theorem 7 offers another argument in favour of Galois connections: Galois connections lift through logical relations. In this section, fusion is used in conjunction with logical pair algebras to prove that safety for free holds even when the expression e uses general recursion.

Lemma 10 *If R is a pair algebra, then $(\mu, \mu) \in R \leftarrow (R \leftarrow R)$.*

Proof. R is a pair algebra, so there are partial orders $\preceq$ and $\sqsubseteq$ and adjoints f and g, such that:

$$f^{\cup} \circ \preceq = R \quad \wedge \quad R = \sqsubseteq \circ g.$$

Therefore:

$$\begin{array}{ll} & (\mu,\mu) \in R \leftarrow (R \leftarrow R) \\ \equiv & \qquad \{ \text{ definition } \leftarrow \} \\ & (\forall h,k.\ (\mu\ h,\ \mu\ k) \in R \Leftarrow (\forall x,y.\ (h\ x,\ k\ y) \in R \Leftarrow (x,y) \in R)) \\ \equiv & \qquad \{\ R = f^{\cup} \circ \preceq\ \} \\ & (\forall h,k.\ f\ (\mu\ h) \preceq \mu\ k \Leftarrow (\forall x,y.\ f\ (h\ x) \preceq k\ y \Leftarrow f\ x \preceq y)) \\ \equiv & \qquad \{\ \preceq \text{ is a partial order, } k \text{ is monotonic } \} \\ & (\forall h,k.\ f\ (\mu\ h) \preceq \mu\ k \Leftarrow (\forall x.\ f\ (h\ x) \preceq k\ (f\ x))) \\ \equiv & \qquad \{ \text{ Theorem 6 } (\mu\text{-Fusion}) \} \\ & \mathit{true}. \end{array}$$

In other words, the statement that $(\mu, \mu) \in R \leftarrow (R \leftarrow R)$ is equivalent to the fusion theorem.

As noted in §4.4, functions that use the fixpoint operator μ do not satisfy the parametricity theorem, because it is *not* true for all R that:

$$(\mu, \mu) \in R \leftarrow (R \leftarrow R).$$

However, Lemma 10 ensures the truth of this statement, provided that R is a pair algebra. In the proof of Theorem 8 (safety for free), every relation involved is a pair algebra. This fact follows from Theorem 7 (logical pair algebras): for all types t, R_t is a pair algebra. Therefore, safety for free holds for all values e, including those that use general recursion.

5.5 An Example

The results of this chapter can be applied to the strictness analysis example of §3.8. Recall that *evalProg* has type:

$$\mathit{evalProg} :: \forall v.\ v \leftarrow \mathit{Prog} \leftarrow \mathit{ConstEnv}\ v.$$

Therefore, by safety for free:

$$\mathit{evalProg} \quad \preceq_t \quad [\mathit{con}, \mathit{abs}^{\cup}]_t\ \mathit{evalProg}, \tag{5.22}$$

with $t = v \leftarrow \mathit{Prog} \leftarrow \mathit{ConstEnv}\ v$. This equation can be shown to be equivalent to the safety property stated in §3.8. First use Lemma 9 to simplify the ordering $\preceq_t$:

$$
\begin{array}{ll}
 & \preceq_{v \leftarrow Prog \leftarrow ConstEnv\ v} \\
= & \quad \{ \text{ Lemma 9 } \} \\
 & \dot{\preceq}_{v \leftarrow Prog \leftarrow ConstEnv\ v} \\
= & \quad \{ \text{ Higher order pointwise orderings: (5.19) and (5.20). } \} \\
 & \preceq_v \leftarrow id \leftarrow id.
\end{array}
$$

Therefore, (5.22) is equivalent to:

$$
\begin{array}{ll}
(\forall consts, p. & evalProg\ consts\ p \\
 & \preceq_v \quad [con, abs^{\cup}]_t\ evalProg\ consts\ p).
\end{array}
\tag{5.23}
$$

All that remains is to expand the definition of the function $[con, abs^{\cup}]_t$ with $t = v \leftarrow Prog \leftarrow ConstEnv\ v$. Using the rules of the logical operator $[\_,\_]$, this is an entirely mechanical expansion. The result is that:

$$
\begin{array}{ll}
 & [con, abs^{\cup}]_t\ evalProg\ consts\ p \\
= & \quad \{ \text{ (5.5), (5.6), (5.7), (5.8) and (5.15) } \} \\
 & con_v\ (evalProg\ consts'\ p),
\end{array}
$$

where:

$$consts' = [abs, con^{\cup}]_{ConstEnv\ v}\ consts.$$

Recalling that $ConstEnv\ v$ is a type synonym for $v \leftarrow List\ v \leftarrow String$, the definition of $consts'$ expands to:

$$consts' = (\lambda s.\ abs_v \circ consts\ s \circ List\ con_v).$$

So the free theorem is that for all values $consts$ and p:

$$evalProg\ consts\ p \quad \preceq_v \quad con_v\ (evalProg\ consts'\ p), \tag{5.24}$$

where $consts'$ is defined as above. This is precisely the safety property stated in §3.8.

Expanding safety for free on concrete examples such as *evalProg* is an entirely mechanical process and it would be simple to build a tool that automates the process. On most examples, however, the result is so simple and intuitive that it can be written down immediately, without any need for calculation.

5.6 Related Work

Most of the results in this chapter have already been observed by Abramsky [3]. The main contribution here is to specialise Abramsky's theorems to pair algebras (Galois connections). Rather than assuming the properties of a pair algebra, Abramsky separates the properties that a relation might satisfy into six categories: strictness, inductiveness, $\bot$-reflection, $\top$-universality, S-monotonicity and L-monotonicity. He then proves that these properties are *inherited* by logical relations. This allows him

to work with relations that are not pair algebras. The stricter criteria of pair algebras seem to be a worthwhile trade-off though, because they lead to very simple and concise results.

A second difference with Abramsky's work is one of focus. Abramsky is interested in the abstract interpretation of the typed lambda calculus. For example, he shows how to analyse higher-order functions for strictness. He does this by evaluating the lambda calculus with a different set of constants, just as was done with the first order functional language in §3.8. In contrast, the work presented here has an extra level of indirection: the typed lambda calculus is used to define an interpreter (denotational semantics) for a second language. An abstract interpretation is then derived for the *second* language. This extra level of indirection allows the results to be applied to any language rather than just lambda calculi. It also allows one to make effective use of a powerful and widely available tool: the polymorphic type-checker.

Cousot and Cousot [22] are dismissive of Abramsky's work on logical relations. They complain (page 110, bottom right) that:

> Moreover, for logical relations [Abramsky], the approximation process is tied up with the standard computation ordering and the type system in the abstraction process. Application to logic programming with e.g. declarative semantics then becomes a bit tortuous. Moreover, it freezes approximation to a few paradigms (such as "approximate pairs by pairs", "approximate functions by functions") which should leave the place to a broader palette of possible choices, such as "approximate functions by pairs, functions, relations, ..., up to a Galois connection) as abundantly illustrated in this paper.

The extra level of indirection discussed above may help to alleviate some of Cousot and Cousot's concerns. The framework described in this chapter is much less rigid than this quote suggests. It is true that safety for free approximates pairs by pairs and functions by functions, but this need not prevent the use of alternative paradigms. Take the example of Chapter 3, which was revisited in §5.5. Safety for free greatly reduces the amount of work that needs to be done, but the definition of $consts'$ still needs to be derived by hand. The polymorphic type of *evalProg* has divided the abstract interpretation process into two parts: the first part follows from safety for free, but the second part must be derived by hand. So polymorphic types are not a hindrance to abstract interpretation but rather a tool for eliminating noise from the calculations. They form a boundary between the interesting part of the calculation and the part that is completely canonical.

It is well-known in the abstract interpretation literature that there is a canonical way of constructing Galois connections. For example, Nielson, *et al.* [76, pages 244-256] discuss the *Systematic Design of Galois connections*. The relevance of logical relations to abstract interpretation has also been known since the 1985 paper of Mycroft and Jones [75]. However, it seems not to be widely known that systematically designed Galois connections *are* logical relations. For example Cousot and Cousot [22, page 105/106] refer to, "... a definite advantage of the Galois connection approach to abstract interpretation over its variant formalization using logical relations".

One of the most successful applications of abstract interpretation has been the strictness analysis of lazy functional programs. A problem that needed to be solved in this field was the strictness analysis of *polymorphic* programs. Abramsky [2] solved the problem by proving that strictness analysis is a *polymorphic invariant*. This means that the strictness of a polymorphic function is the same for all instances of that function. Therefore it suffices to analyse the strictness of a simple instance of the function, using standard (monomorphic) techniques. Subsequently, Abramsky and Jensen [5] have found a much simpler way of proving the same result. Their proof is based on very similar techniques to those used in this chapter, but again their goals are slightly different. The results in this chapter are not oriented towards the abstract interpretation of the polymorphic lambda calculus itself, but languages whose semantics are defined using it.

5.7 Summary

The two main results of this chapter are Theorem 7 (logical pair algebras) and Theorem 8 (safety for free). The first formalises the well-known notion (Nielson, *et al.* [76, pages 244-256]) that Galois connections can be *systematically designed*. Such systematic constructions correspond precisely to the concept of a logical relation. Safety for free is the corollary obtained by instantiating the parametricity theorem with these logical relations. The result is very easy to apply in practice, as demonstrated in §5.5. This is particularly due to the introduction of the logical operator $[\_,\_]$, which produces a function (rather than a relation) when used in the form $[f, g^{\cup}]_t$.

In contrast to Wadler's [103] free theorems, Theorem 8 does not suffer from the problems caused by general recursion (§4.4). That Theorem 8 holds for *all* values, including those that use general recursion, is a pleasing consequence of working with Galois connections.

The three chapters following this one illustrate the use of safety for free. Chapters 6 and 7 apply the theorem to DSELs and Chapter 8 applies it to a traditional example of abstract interpretation: the strictness analysis of attribute grammars.

Chapter 6

Communicating Sequential Processes

This chapter discusses Hoare's language of Communicating Sequential Processes [40] (CSP) as an example of a DSEL. CSP is language for writing programs that use concurrency and communication. To a large extent, CSP can be encoded very neatly as a DSEL, but *alphabets* are difficult to model in the style intended by Hoare. The solution offered in this chapter is to analyse the alphabets separately by abstract interpretation.

The DSEL for CSP introduced in this chapter is simplistic in that it only covers the material discussed in Chapter 2 of Hoare's book. In other words, non-determinism is omitted. This subset of CSP is nonetheless relevant to some existing DSELs. For example, Elliot and Hudak [25, 26] use a deterministic event model similar to CSP's in their Functional Reactive Animation library.

6.1 A Brief Introduction to CSP

Two of Hoare's [40] classic examples are used to illustrate CSP. Vending machines are used to introduce the notation and alphabets are demonstrated with the dining philosophers in §6.2.2. In Hoare's vending machine example, there are two processes. The first is a customer who repeatedly inserts a coin into the machine and requests chocolate; the second is a vending machine that will dispense either chocolate or toffee after a coin has been paid. Using Hoare's notation, the two processes are defined as follows:

$$\begin{array}{lcl} \mathit{Customer} & = & \mathit{coin} \rightarrow \mathit{choc} \rightarrow \mathit{Customer}, \\ \mathit{Vend} & = & \mathit{coin} \rightarrow (\mathit{choc} \rightarrow \mathit{Vend} \mid \mathit{toffee} \rightarrow \mathit{Vend}). \end{array}$$

The capitalised words are processes and the words written in lower case are events. If P is a process and x an event, then $x \rightarrow P$ is a process that first engages in the event x and then behaves like P. The process $(x \rightarrow P \mid y \rightarrow Q)$ offers a choice. Either it engages in x and then continues as P or it engages in y and then continues as Q. Hoare generalises these constructions to the general choice operator $(x : B \rightarrow P(x))$.

Here B is a finite set of events. The process first engages in one of the events in B and then continues as $P(x)$.

A second method of constructing processes is concurrency. The process $P \parallel Q$ is the synchronisation of the processes P and Q. That is, an event can only happen if both P and Q can simultaneously engage in that event. For example, if *Customer* and *Vend* are run concurrently then the vending machine will never vend toffee, because the customer never requests it. The two processes do agree on the events *coin* and *choc* though, so *Customer* $\parallel$ *Vend* is a process that engages repeatedly in *coin* followed by *choc*. In other words it is equivalent to the *Customer* process.

6.2 The Interface to the Library

Below is the signature of a simple CSP library:

$$\begin{array}{l}\textbf{type } \mathit{CSP}\ \alpha\ \rho = \textbf{sig } \mathit{choice} \quad :: \rho \leftarrow [(\alpha,\rho)],\\ \qquad\qquad\qquad\qquad\quad \mathit{par} \quad\quad :: \rho \leftarrow (\rho,\rho),\\ \qquad\qquad\qquad\qquad\quad \mathit{extend} \quad :: \rho \leftarrow \rho \leftarrow [\alpha],\\ \qquad\qquad\qquad\qquad\quad \mathit{alphabet} :: \rho \leftarrow \rho \leftarrow [\alpha]\\ \qquad\qquad\qquad\qquad \textbf{end}.\end{array}$$

This signature has been encoded using the notation and ideas introduced in Chapter 2. The interface contains four methods: the general choice operator, the concurrency operator, an operator for extending the alphabet of a process, and an operator for asserting the alphabet of a process. This set of combinators differs intentionally from Hoare's, to make the manipulation of alphabets more explicit. In Hoare's CSP, two processes with different alphabets can be joined with the concurrency operator, but that is not allowed here; before applying *par*, *extend* should be applied to both processes so that their alphabets are equal. The effect of *extend xs P* is to modify the behaviour of P such that it ignores any event in *xs* and behaves as before on other events. The method *alphabet* has no effect on the behaviour of a process: it merely allows a process to be annotated with its alphabet. Its purpose is similar to explicit type annotations in ML: ML programs do not need to be annotated with types, because ML has full type inference, but type annotations often make programs easier for humans to read. During type checking, ML implementations confirm that the annotated types agree with the inferred types. Similarly, the alphabet analysis derived later in this chapter confirms that the inferred alphabets and the annotated alphabets are the same.

The types α and ρ are abstract datatypes, α being the type of an event and ρ being the type of a process. Throughout this chapter the assumption is made that the set of events (of type α) is finite and that they can be tested for equality. To improve readability in the examples of processes given below, *par* is often written using the binary operator $\parallel$.

Alphabets are types, but there is no natural way of encoding them in an ML-style type system. For example, to accurately encode the type of *extend*, a union operation

on types is needed. With such an operation, the type of *extend* might look something like this:

$$extend :: \rho(A \cup B) \leftarrow \rho(A) \leftarrow B$$

In other words, given a process with alphabet A and a set of events, B, *extend* produces a process with alphabet $A \cup B$. Unfortunately, it is not possible to encode this information in an ML-style type system. The solution proposed in this chapter is to analyse alphabets separately using abstract interpretation.

6.2.1 Using the Library

The method *choice* corresponds to Hoare's general choice operator ($x : B \rightarrow P(x)$). A CSP process such as $(x \rightarrow P \mid y \rightarrow Q \mid z \rightarrow R)$ is encoded as:

$$choice\ [(x,P),(y,Q),(z,R)].$$

Two common special cases of *choice* are guarded processes such as $x \rightarrow P$ and the *stop* process that refuses to engage in any event. It is convenient to define a functor module (§2.4.2) to introduce operators for these special cases. The functor module, below, also defines an operator *extendFrom*, that extends a process from alphabet *xs* to alphabet *ys*. The function *diff* is used to remove any elements of *xs* from *ys*.

$$\begin{array}{lcl}
mkCSPutils & :: & \forall\alpha,\rho.\ CSPutils\ \alpha\ \rho \leftarrow CSP\ \alpha\ \rho, \\
mkCSPutils\ C & = & \textbf{struct}\ stop = C.choice\ [], \\
 & & \quad guard\ x\ P = C.choice\ [(x,P)], \\
 & & \quad extendFrom\ xs\ ys\ P = extend\ (diff\ xs\ ys) \\
 & & \qquad\qquad (alphabet\ xs\ P) \\
 & & \textbf{end},
\end{array}$$

where

$$\begin{array}{l}
\textbf{type}\ CSPutils\ \alpha\ \rho = \textbf{sig}\ stop :: \rho, \\
\qquad\qquad guard :: \rho \leftarrow \rho \leftarrow \alpha, \\
\qquad\qquad extendFrom :: \rho \leftarrow \rho \leftarrow [\alpha] \leftarrow [\alpha] \\
\qquad \textbf{end}.
\end{array}$$

The input to *mkCSPutils* is an implementation of the *CSP* signature and its output is a module of type *CSPutils*, containing the *stop* process, the *guard* operator, and the *extendFrom* operator. In the examples below, *guard x P* is written using the infix notation $x \rightsquigarrow P$. Hoare's notation $x \rightarrow P$ is not used to avoid confusion with function types.

Encoding the *Customer* and *Vend* processes is straightforward. Firstly, the type of events is a simple enumeration:

$$\textbf{data}\ A = coin \mid choc \mid toffee.$$

The definitions of the processes are:

$$
\begin{array}{l}
\mathit{Customer} \;\; :: \; \forall \rho.\; \rho \leftarrow \mathit{CSP}\ A\ \rho, \\
\mathit{Customer}\ C \;=\; \mathbf{open}\ \mathit{mkCSPutils}\ C\ \mathbf{in} \\
\qquad\qquad\quad \mathit{coin} \rightsquigarrow \mathit{choc} \rightsquigarrow \mathit{Customer}\ C,
\end{array}
$$

$$
\begin{array}{l}
\mathit{Vend} \;\; :: \; \forall \rho.\; \rho \leftarrow \mathit{CSP}\ A\ \rho, \\
\mathit{Vend}\ C \;=\; \mathbf{open}\ C\ \mathbf{in} \\
\qquad\qquad\quad \mathbf{open}\ \mathit{mkCSPutils}\ C\ \mathbf{in} \\
\qquad\qquad\quad \mathit{coin} \rightsquigarrow \mathit{choice}\ [(\mathit{choc},\ \mathit{Vend}\ C),\ (\mathit{toffee},\ \mathit{Vend}\ C)].
\end{array}
$$

The **open** directives are used here to bring the methods of C and *mkCSPutils* C into scope. Note that *Customer* and *Vend* are not polymorphic in the type A of events, because they explicitly refer to the events *coin*, *choc* and *toffee*. However they are still polymorphic in ρ (the type of a process), because they are independent of the implementation of the library. In other words ρ is treated as an abstract datatype.

6.2.2 The Dining Philosophers

To properly illustrate the use of alphabets, a more complex example is needed. Below is Hoare's description of a scenario involving five dining philosophers:

> In ancient times, a wealthy philanthropist endowed a College to accommodate five eminent philosophers. Each philosopher had a room in which he could engage in his professional activity of thinking; there was also a common dining room, furnished with a circular table, surrounded by five chairs, each labelled by the name of the philosopher who was to sit in it. The names of the philosophers were $PHIL_0$, $PHIL_1$, $PHIL_2$, $PHIL_3$, $PHIL_4$, and they were disposed in this order anticlockwise round the table. To the left of each philosopher there was laid a golden fork, and in the centre stood a large bowl of spaghetti, which was constantly replenished.
>
> A philosopher was expected to spend most of his time thinking; but when he felt hungry, he went to the dining room, sat down in his own chair, picked up his own fork on his left, and plunged it into the spaghetti. But such is the tangled nature of spaghetti that a second fork is required to carry it to the mouth. The philosopher therefore had also to pick up the fork on his right. When he was finished he would put down both his forks, get up from his chair, and continue thinking. Of course, a fork can be used by only one philosopher at a time. If the other philosopher wants it, he just has to wait until the fork is available again.

To encode this scenario, a datatype of events must first be defined. The events are when a philosopher stands up or sits down or when a fork is picked up or put down:

$$\mathbf{data}\ E = \mathit{Sit}\ \mathit{Int} \mid \mathit{Stand}\ \mathit{Int} \mid \mathit{PickUp}\ \mathit{Int}\ \mathit{Int} \mid \mathit{PutDown}\ \mathit{Int}\ \mathit{Int}.$$

The type *Int* is used here to number the philosophers and forks. The operations *incr* and *decr* are used to add and subtract modulo five:

$$
\begin{array}{l}
incr \;\; :: \; Int \leftarrow Int, \\
incr \; 4 \; = \; 0, \\
incr \; k \; = \; k{+}1,
\end{array}
$$

$$
\begin{array}{l}
decr \;\; :: \; Int \leftarrow Int, \\
decr \; 0 \; = \; 4, \\
decr \; k \; = \; k{-}1.
\end{array}
$$

The philosophers can then be defined by a single function, parameterised by a number:

$$
\begin{array}{l}
phil \;\; :: \; \forall \rho.\; \rho \leftarrow Int \leftarrow CSP \; E \; \rho, \\
phil \; C \; k \; = \; \textbf{open} \; mkCSPutils \; C \; \textbf{in} \\
\qquad Sit \; k \rightsquigarrow PickUp \; k \; k \rightsquigarrow PickUp \; k \; (incr \; k) \rightsquigarrow PutDown \; k \; k \\
\qquad\quad \rightsquigarrow PutDown \; k \; (incr \; k) \rightsquigarrow Stand \; k \rightsquigarrow phil \; C \; k.
\end{array}
$$

The alphabet of one of these philosophers is defined by:

$$
\begin{array}{l}
philE \;\; :: \; [E] \leftarrow Int, \\
philE \; k \; = \; [Sit \; k, \; PickUp \; k \; k, \; PickUp \; k \; (incr \; k), \\
\qquad PutDown \; k \; k, \; PutDown \; k \; (incr \; k), \; Stand \; k].
\end{array}
$$

The behaviour and alphabets of the forks can be defined similarly:

$$
\begin{array}{l}
fork \;\; :: \; \forall \rho.\; \rho \leftarrow Int \leftarrow CSP \; E \; \rho, \\
fork \; C \; k \; = \; \textbf{open} \; C \; \textbf{in} \\
\qquad \textbf{open} \; mkCSPutils \; C \; \textbf{in} \\
\qquad choice \; C \; [(PickUp \; k \; k, \; PutDown \; k \; k \rightsquigarrow fork \; C \; k), \\
\qquad\qquad\quad (PickUp \; (decr \; k) \; k, \\
\qquad\qquad\qquad PutDown \; (decr \; k) \; k \rightsquigarrow fork \; C \; k)],
\end{array}
$$

$$
\begin{array}{l}
forkE \;\; :: \; [E] \leftarrow Int, \\
forkE \; k \; = \; [PickUp \; k \; k, \; PickUp \; (decr \; k) \; k, \\
\qquad PutDown \; k \; k, \; PutDown \; (decr \; k) \; k].
\end{array}
$$

Finally, the processes can be combined to form a college:

$$
\begin{array}{l}
college \; :: \; \forall \rho.\; \rho \leftarrow CSP \; E \; \rho \\
college \; C \; = \\
\quad \textbf{open} \; C \; \textbf{in} \\
\quad \textbf{open} \; mkCSPutils \; C \; \textbf{in} \\
\quad \textbf{let} \; p \; = \; \lambda k.\; extendFrom \; (philE \; k) \; collegeE \; (phil \; k) \; \textbf{in} \\
\quad \textbf{let} \; f \; = \; \lambda k.\; extendFrom \; (forkE \; k) \; collegeE \; (fork \; k) \; \textbf{in} \\
\quad (p \; 0 \parallel p \; 1 \parallel p \; 2 \parallel p \; 3 \parallel p \; 4) \parallel (f \; 0 \parallel f \; 1 \parallel f \; 2 \parallel f \; 3 \parallel f \; 4).
\end{array}
$$

The alphabet *collegeE* used here is an amalgamation of all the events in the system:

$$
\begin{array}{l}
\mathit{collegeE} \;::\; [E], \\
\mathit{collegeE} \;=\; \textbf{let}\ \mathit{ns} = [0,1,2,3,4]\ \textbf{in} \\
\qquad\qquad \textbf{let}\ \mathit{ns}' = \mathit{map}\ \mathit{incr}\ \mathit{ns}\ \textbf{in} \\
\qquad\qquad \mathit{map}\ \mathit{Sit}\ \mathit{ns} +\!\!+ \mathit{map}\ \mathit{Stand}\ \mathit{ns} +\!\!+ \mathit{zipWith}\ \mathit{PickUp}\ \mathit{ns}\ \mathit{ns} \\
\qquad\qquad\quad +\!\!+ \mathit{zipWith}\ \mathit{PutDown}\ \mathit{ns}\ \mathit{ns} +\!\!+ \\
\qquad\qquad\quad \mathit{zipWith}\ \mathit{PickUp}\ \mathit{ns}\ \mathit{ns}' +\!\!+ \mathit{zipWith}\ \mathit{PutDown}\ \mathit{ns}\ \mathit{ns}'.
\end{array}
$$

Note how the alphabets of the processes are extended before they are concurrently combined. The *extendFrom* operator not only extends the alphabets, but also adds *alphabet* annotations.

6.3 Traces

Hoare defines the semantics of CSP using *traces*. A trace xs of the process P is a finite sequence of events that P might engage in. Traces are represented here as lists, so the concatenation of the traces xs and ys is $xs +\!\!+ ys$. Traces are partially ordered, xs being less than ys if it is a prefix of ys:

$$xs \leq ys \;\equiv\; (\exists ts.\ xs +\!\!+ ts = ys).$$

If ts is a trace of a process P and $rs \leq ts$, then rs is also a trace of P. Hoare defines the semantics of a process P as the set of all traces of P. That is:

$$\textbf{type}\ \mathit{Process}\ \alpha = \mathit{Set}\ [\alpha].$$

The type parameter α is the type of events. Given a type of events A, *Process* A is a complete lattice under the set inclusion ($\subseteq$) ordering. The bottom element of the lattice is the empty set $\emptyset$ and the top element is *all* A, where *all* is defined as follows:

$$
\begin{array}{l}
\mathit{all} \;::\; \forall\alpha.\ \mathit{Process}\ \alpha \leftarrow \mathit{Set}\ \alpha, \\
\mathit{all}\ X \;=\; \{\ \mathit{xs} \mid \mathit{elems}\ \mathit{xs} \subseteq X\ \}.
\end{array}
$$

The function *elems* converts a list to a set; it is defined in Appendix A.2.

The trace-semantics of CSP are given by the following module definition:

$$
\begin{array}{lll}
\mathit{Tr} & :: & \forall\alpha.\ \mathit{CSP}\ \alpha\ (\mathit{Process}\ \alpha), \\
\mathit{Tr} & = & \textbf{struct}\ \ \mathit{choice} = \mathit{foldr}\ (\cup)\ \{\,[]\,\} \circ \mathit{map}\ \mathit{cons}, \\
 & & \qquad\quad P \parallel Q = P \cap Q, \\
 & & \qquad\quad \mathit{extend}\ \ \mathit{xs}\ P = \{\ \mathit{ys} \mid \mathit{diff}\ \mathit{xs}\ \mathit{ys} \in P\ \}, \\
 & & \qquad\quad \mathit{alphabet}\ \mathit{xs}\ P = P \\
 & & \textbf{end}.
\end{array}
$$

The definition of *choice* uses *foldr* (Appendix A.2) to compute the union of the traces of each individual choice and also adds the empty trace. The set of traces for each individual choice is computed by *cons*, defined as follows:

$$
\begin{array}{l}
\mathit{cons} \;::\; \forall\alpha.\ \mathit{Process}\ \alpha \leftarrow (\alpha,\ \mathit{Process}\ \alpha), \\
\mathit{cons}\ (x,P) \;=\; \{\ x : \mathit{xs} \mid \mathit{xs} \in P\ \}.
\end{array}
$$

The parallel composition of two processes contains the set of traces that both processes agree on. In other words, it contains the intersection of their traces. The alphabet of a process is extended by interleaving the new events in every possible way with the old events. Finally, *alphabet* has no effect. Its only use is as an assertion mechanism in the alphabet analysis.

Tr can be used to evaluate the semantics of the processes defined earlier. For example, to evaluate *Customer*, note first that the recursive definition of *Customer* is syntactic sugar for the following application of μ (see Chapter 2):

$$\begin{array}{l} \mathit{Customer} \quad :: \ \forall\rho.\ \rho \leftarrow \mathit{CSP}\ A\ \rho, \\ \mathit{Customer}\ C\ = \ \mathbf{open}\ \mathit{mkCSPutils}\ C\ \mathbf{in} \\ \qquad\qquad\qquad (\mu\, X.\ \mathit{coin} \rightsquigarrow \mathit{choc} \rightsquigarrow X), \end{array}$$

Hence, when applied to *Tr*, *Customer* expands as follows:

$$\begin{array}{ll} & \mathit{Customer}\ \mathit{Tr} \\ = & \qquad \{\ \text{Definitions } \mathit{Customer} \text{ and } \mathit{mkCSPUtils}\ \} \\ & (\mu\, X.\ \mathit{Tr.choice}\ \ [(\mathit{coin},\ \ \mathit{Tr.choice}\ \ [(\mathit{choc}, X)])]) \\ = & \qquad \{\ \text{Definition } \mathit{Tr.choice}\ \} \\ & (\mu\, X.\ \{\,[]\,\}\ \cup\ \mathit{cons}\ \ (\mathit{coin},\ \{\,[]\,\} \cup \mathit{cons}\ \ (\mathit{choc}, X))) \\ = & \qquad \{\ \text{Definition } \mathit{cons}\ \} \\ & (\mu\, X.\ \{\,[]\,\} \cup \{\ [\mathit{coin}]\ \} \cup \{\ \mathit{coin} : \mathit{choc} : \mathit{xs} \mid \mathit{xs} \in X\ \}). \end{array}$$

The solution to this least fixed point is the set $\{[], [\mathit{coin}], [\mathit{coin}, \mathit{choc}], \ldots\}$, as expected.

6.4 Analysing the Alphabet

An analysis to determine the alphabet of a process can be derived by abstract interpretation. The abstraction function is the function *events*, which determines the set of events that a process might engage in:

$$\begin{array}{l} \mathit{events} \quad :: \ \forall\alpha.\ \mathit{Set}\ \alpha \leftarrow \mathit{Process}\ \alpha, \\ \mathit{events}\ P\ = \ (\bigcup \mathit{xs}\colon\ \mathit{xs} \in P\colon\ \mathit{elems}\ \mathit{xs}). \end{array}$$

Notice that *events* forms a Galois connection with *all*:

$$\mathit{events}\ P\ \subseteq\ X \quad \equiv \quad P\ \subseteq\ \mathit{all}\ X.$$

Therefore, given a process such as *Customer* of type $\forall\rho.\ \rho \leftarrow \mathit{CSP}\ A\ \rho$, safety for free (page 29) states that:

$$\mathit{Customer}\ \mathit{Tr}\ \ \subseteq\ \ [(\rho \mapsto \mathit{all}), (\rho \mapsto \mathit{events}^{\cup})]_t\ \mathit{Customer}\ \mathit{Tr},$$

where $t = \rho \leftarrow \mathit{CSP}\ A\ \rho$. This statement is equivalent to:

$$\mathit{events}\ (\mathit{Customer}\ \mathit{Tr})\ \ \subseteq\ \ \mathit{Customer}\ \mathit{Ev},$$

where *Ev* is an implementation of the *CSP* interface, satisfying:

$$
\begin{array}{lll}
Ev & :: & \forall\alpha.\ CSP\ \alpha\ (Set\ \alpha), \\
Ev & = & \mathbf{struct}\ \ choice \quad\quad\quad\ = events \circ Tr.choice \circ map\ (id \times all), \\
 & & \quad\quad\ par \quad\quad\quad\ = events \circ Tr.par \circ (all \times all), \\
 & & \quad\quad\ extend\ \ xs\ = events \circ Tr.extend\ xs \circ all, \\
 & & \quad\quad\ alphabet\ xs\ = events \circ Tr.alphabet\ xs \circ all \\
 & & \mathbf{end}.
\end{array}
$$

This definition can be simplified, after two observations about *events* have been made. Firstly,

$$events\ (P \cup Q)\ =\ events\ P \cup events\ Q, \tag{6.1}$$

and secondly,

$$events \circ cons \circ (id \times all)\ =\ insert, \tag{6.2}$$

where *insert* is a function that adds an element to a set:

$$
\begin{array}{l}
insert \quad\quad :: \forall\alpha.\ Set\ \alpha \leftarrow (\alpha,\ Set\ \alpha), \\
insert\ (x,X)\ =\ \{x\} \cup X.
\end{array}
$$

The derivation of *Ev.choice* is then as follows:

$$
\begin{array}{ll}
 & Ev.choice \\
= & \quad \{\ \text{Definition } Ev\ \} \\
 & events \circ Tr.choice \circ map\ (id \times all) \\
= & \quad \{\ \text{Definition } Tr.choice\ \} \\
 & events \circ foldr\ (\cup)\ \{\,[]\,\} \circ map\ cons \circ map\ (id \times all) \\
= & \quad \{\ \text{Fusion (see below)}\ \} \\
 & foldr\ (\cup)\ \emptyset \circ map\ events \circ map\ cons \circ map\ (id \times all) \\
= & \quad \{\ map \text{ is a functor}\ \} \\
 & foldr\ (\cup)\ \emptyset \circ map\ (events \circ cons \circ (id \times all)) \\
= & \quad \{\ (6.2)\ \} \\
 & foldr\ (\cup)\ \emptyset \circ map\ insert.
\end{array}
$$

In the second step of this calculation *fold fusion* is used, followed by *fold-map fission*. These rules of *foldr* are explained by Bird [14, page 131]. They are applicable here due to the distributivity property (6.1).

The remaining calculations of *Ev.par*, *Ev.extend*, and *Ev.alphabet* are straightforward; they lead to the following definition of *Ev*:

$$
\begin{array}{lll}
Ev & :: & \forall\alpha.\ CSP\ \alpha\ (Set\ \alpha), \\
Ev & = & \mathbf{struct}\ \ choice \quad\quad\quad\ = foldr\ (\cup)\ \emptyset \circ map\ insert, \\
 & & \quad\quad\ P \parallel Q \quad\quad\quad\ = P \cap Q, \\
 & & \quad\quad\ extend\ \ xs\ P\ = \mathbf{let}\ X\ =\ elems\ xs\ \mathbf{in}\ X \cup P, \\
 & & \quad\quad\ alphabet\ xs\ P\ = P \\
 & & \mathbf{end}.
\end{array}
$$

Of course, the purpose of the alphabet analysis is to produce error messages when Hoare's alphabet rules are not followed correctly. The above implementation only calculates the alphabets; it does not produce any error messages. A simple solution to this problem is to add some assertions to *Ev*:

$$
\begin{array}{lll}
Ev & :: & \forall\alpha.\ CSP\ \alpha\ (Set\ \alpha), \\
Ev & = & \textbf{struct}\ \ choice = foldr\ (\cup)\ \emptyset \circ map\ insert, \\
 & & \qquad\quad P \parallel Q = \textbf{assert}\ P = Q\ \textbf{in}\ P \cap Q, \\
 & & \qquad\quad extend\ \ xs\ P = \textbf{let}\ X = elems\ xs\ \textbf{in} \\
 & & \qquad\qquad\qquad\qquad\ \ \textbf{assert}\ X \cap P = \emptyset\ \textbf{in}\ X \cup P, \\
 & & \qquad\quad alphabet\ xs\ P = \textbf{assert}\ P = elems\ xs\ \textbf{in}\ P \\
 & & \textbf{end}.
\end{array}
$$

While evaluating a process with *Ev*, the implementation should check the assertions and print an error message if any of them fails. The assertions state, respectively, that the alphabets of two concurrent processes should be equal, that *extend* should only be used to add new events to a process's alphabet, and that the alphabet of an annotated process should satisfy its specification.

To illustrate the use of *Ev*, consider again the *Customer* example. The evaluation steps are very similar to the evaluation of *Customer Tr*, leading this time to:

$$Customer\ Ev = (\mu X.\ \{\ coin\ \} \cup \{\ choc\ \} \cup X).$$

In other words, $Customer\ Ev = \{coin, choc\}$, as expected. The evaluation process could be automated. This would allow examples such as the dining philosophers, which are too large to evaluate by hand, to be checked automatically.

6.5 Summary

Hoare's language of Communicating Sequential Processes has proved particularly amenable to static analysis. Powerful tools such as FDR, which is developed by Formal Systems Ltd. [30] and described in Roscoe's book [87] on concurrency, can be used to check a CSP process against a specification. In comparison, the alphabet analysis presented here is rather trivial. It does however solve a problem that is common in the design of DSELs: the ML-style type system of the host language is inappropriate for the embedded language. The *alphabets* of CSP cannot be modelled properly with an ML-style type system, because such type systems do not offer a union operation on types. It is in unfortunate situations like this that the *phantom type* approach of Leijen and Meijer [63] (§1.3.2) is not immediately applicable. The problem can instead be solved by developing an abstract interpretation that is tailored to the needs of the embedded language.

As further work, it would be interesting to explore more sophisticated analyses of CSP. A particularly interesting candidate would be the analysis performed by FDR, which checks that one CSP process is a refinement of another. This would involve adding non-determinism to the semantics of CSP.

Chapter 7

Parser Combinators

Parser combinator libraries are a well-studied topic in functional programming and are one of the best known examples of Domain-Specific Embedded Languages. Parser combinators have many advantages in comparison to traditional parser generators such as YACC [47]. In particular, it is very easy to define new combinators for common parsing problems such as comma-separated lists and optional fields. Parsers written using parser combinators are also very readable, due to Wadler's observation [104] that parsers form an instance of a *monad*. Monads allow the *actions* of the parser to be written in a very natural way.[1]

The one major drawback of parser combinators is that they lack the error-checking facilities of tools such as YACC, which perform (conservative) analyses to check that the grammar is non-ambiguous and can be implemented efficiently. In contrast, no such facilities are available for parser combinators. This means that there is a risk of writing a parser that is ambiguous, inefficient or even non-terminating. The reasons why parsers written with parser combinators are difficult to test are discussed in more detail in §7.7, but the fundamental problem is the embedded nature of the combinators: it is not possible to *syntactically* analyse the grammar. The solution is to develop an analysis that is not based on syntax. This is done by abstract interpretation.

For an introduction to monadic parser combinators, see Hutton and Meijer [46]. Practical Haskell implementations of parser combinators are maintained by Swierstra [94, 95, 96] and Leijen [64].

7.1 Using Parser Combinators

Below is the signature of a simple parser combinator library:

[1] An alternative to monads are *arrows*: see Hughes [45]. Swierstra's combinators [94, 95, 96] use this approach.

$$
\begin{array}{l}
\textbf{type}\ \mathit{Parsing}\ \tau\ g = \textbf{sig}\ \mathit{empty} :: g, \\
\qquad\qquad\qquad\qquad\quad \mathit{token}\ :: g \leftarrow \tau, \\
\qquad\qquad\qquad\qquad\quad \mathit{choice} :: g \leftarrow (g,g), \\
\qquad\qquad\qquad\qquad\quad \mathit{cat}\ \ :: g \leftarrow (g,g) \\
\qquad\qquad\qquad\quad \textbf{end}.
\end{array}
$$

The types τ and g are abstract datatypes, τ being the type of a token and g being the type of a parser. Throughout this chapter the assumption is made that the set of tokens (of type τ) is finite and that they can be tested for equality. A typical example is the datatype *Char*. The library contains primitive parsers *empty* and *token*, which parse empty and singleton strings, respectively. The union of two languages is computed by *choice* and their sequential composition by *cat*. In the examples of parsers given below, the binary operators | and $\cdot$ are used for *choice* and *cat*, respectively. This is intended to improve the readability of the examples. However in calculations the function names are used.

A simplification has been made in this module signature, because it is not *monadic*. Monadic parser combinators allow parsers to be annotated with *computations* (also known as *actions*), which means that they can produce a result (such as an abstract syntax tree). With the module signature defined above, the result type of the parser is defined by the module implementation, so it is not possible for the parser to return a user-defined result. The reason for introducing this simplification is discussed further in §9.2, but it does not actually affect the results of this chapter, because the computations of a parser are not related to the question of its ambiguity.

7.1.1 Examples of Parsers

To see how the combinators can be used to define a parser, consider a simple grammar:

$$S \ ::= \ \mathrm{a}S \mid \mathrm{b}$$

This grammar defines a language containing the words "b", "ab", "aab", and so on. Using the combinators, a parser for this language is:

$$
\begin{array}{l}
\mathit{aaab}\ \ :: \forall g.\ g \leftarrow \mathit{Parsing}\ \mathit{Char}\ g, \\
\mathit{aaab}\ G\ = \textbf{open}\ G\ \textbf{in} \\
\qquad\quad \textbf{let}\ a\ = \mathit{token}\ \text{`a'}, \\
\qquad\qquad\ \ b\ = \mathit{token}\ \text{`b'}, \\
\qquad\qquad\ \ S\ = (a \cdot S) \mid b\ \textbf{in}\ S.
\end{array}
$$

Note that this function is polymorphic in the type g, which means that the type of a parser is treated as an abstract datatype.

The second example is the grammar:

$$S \ ::= \ S\mathrm{a} \mid \mathrm{b}$$

The language defined by this grammar contains the words "b", "ba", "baa", and so on. Using the combinators, its definition is:

$$
\begin{array}{lcl}
\mathit{baaa} & :: & \forall g.\ g \leftarrow \mathit{Parsing}\ \mathit{Char}\ g, \\
\mathit{baaa}\ G & = & \mathbf{open}\ G\ \mathbf{in} \\
 & & \quad \mathbf{let}\ a = \mathit{token}\ \text{`a'}, \\
 & & \qquad\ \ b = \mathit{token}\ \text{`b'}, \\
 & & \qquad\ \ S = (S \cdot a) \mid b\ \mathbf{in}\ S.
\end{array}
$$

Unfortunately, this parser does not work with standard combinators. The problem is caused by the grammar being *left-recursive*. Most implementations enter an infinite loop when trying to evaluate S. A semantic explanation of this problem is given in §7.2. One of the goals of the analysis presented here is to statically detect left-recursion.

Aho and Ullman [7, pages 344–345] describe a simple technique for eliminating left-recursion from a grammar. The left-recursive grammar given above is equivalent to:

$$
\begin{array}{lcl}
S & ::= & \mathrm{b}S' \\
S' & ::= & \mathrm{a}S' \mid \epsilon
\end{array}
$$

This grammar, which is not left-recursive, can be encoded as follows:

$$
\begin{array}{lcl}
\mathit{baaa}' & :: & \forall g.\ g \leftarrow \mathit{Parsing}\ \mathit{Char}\ g, \\
\mathit{baaa}'\ G & = & \mathbf{open}\ G\ \mathbf{in} \\
 & & \quad \mathbf{let}\ a = \mathit{token}\ \text{`a'}, \\
 & & \qquad\ \ b = \mathit{token}\ \text{`b'}, \\
 & & \qquad\ \ e = \mathit{empty}, \\
 & & \qquad\ \ S = b \cdot S', \\
 & & \qquad\ \ S' = (a \cdot S') \mid e\ \mathbf{in}\ S.
\end{array}
$$

The final example illustrates *ambiguity*. It is based on the well-known *dangling else problem* caused by conditional statements in imperative languages such as C. (See Kernighan and Ritchie [52, page 56].) Grammars for such languages contain productions similar to the following:

$$
\begin{array}{lcl}
\mathit{Stmt} & ::= & \text{if } \mathit{Expr} \text{ then } \mathit{Stmt} \\
 & \mid & \text{if } \mathit{Expr} \text{ then } \mathit{Stmt} \text{ else } \mathit{Stmt} \\
 & \mid & \ldots
\end{array}
$$

A statement such as

$$\text{if } P \text{ then if } Q \text{ then S else T}$$

is ambiguous, because it could be read as either

$$
\begin{array}{l}
\text{if } P \text{ then if } Q \text{ then S} \\
\qquad \text{else T}
\end{array}
$$

or

if P then if Q then S
else T

From a grammatical perspective, this string is ambiguous because it can be derived in two distinct ways. To keep the example concise, it can be simplified to a grammar containing just three terminal symbols:

S ::= aS | aSbS | c

An example of an ambiguous word in this language is "aacbc". Using the combinators, this grammar is written as:

$$
\begin{array}{l}
\mathit{cond} \;\; :: \; \forall g.\; g \leftarrow \mathit{Parsing}\ \mathit{Char}\ g, \\
\mathit{cond}\ G \;=\; \textbf{open}\ G\ \textbf{in} \\
\qquad \textbf{let}\ a \;=\; \mathit{token}\ \text{`a'}, \\
\qquad\quad\; b \;=\; \mathit{token}\ \text{`b'}, \\
\qquad\quad\; c \;=\; \mathit{token}\ \text{`c'}, \\
\qquad\quad\; S \;=\; (a \cdot S) \mid (a \cdot S \cdot b \cdot S) \mid c\ \textbf{in}\ S.
\end{array}
$$

The analysis presented later in this chapter warns that there might be an ambiguity in this grammar.

7.2 Implementing the Combinators

The standard implementation technique used by combinator libraries is to represent a parser as a (partial) function:

type $\mathit{Parse}\ \tau = [[\tau]] \nleftarrow [\tau]$.

The input is a list of tokens — the string to be parsed. The parser attempts to parse *prefixes* of this list. For example, if the input is the string "aaaaabcc", then the parser *aaab* (defined above) will successfully parse the prefix "aaaaab" and return the suffix ["cc"]. If no prefix of the string is in the grammar then the result is the empty list. If there is more than one way of parsing a prefix of the string, then a list of suffixes is returned. For example, given the input "acbcdd", the parser *cond* returns the list ["bcdd", "dd"]. If the entire string can be parsed, then one of the suffixes returned is the empty string: "". This indicates that the string is in the language defined by the parser. For example, given the input "baaa", $baaa'$ returns ["", "a", "aa", "aaa"]. One of the results is "", so "baaa" is in the language defined by $baaa'$.

This representation of grammars is convenient to implement in a functional language, because *choice* and *cat* can be implemented by concatenation and function composition, respectively. The implementation is given below.

$$
\begin{array}{lll}
P \;::\; \forall\tau.\; \mathit{Parsing}\ \tau\ (\mathit{Parse}\ \tau), & & \\
P \;=\; \textbf{struct}\ \mathit{empty}\ \mathit{xs} & & = [\mathit{xs}], \\
\qquad\quad \mathit{token}\ t & [] & = [], \\
\qquad\quad \mathit{token}\ t & (x : \mathit{xs}) & = \textbf{if}\ x = t\ \textbf{then}\ [\mathit{xs}]\ \textbf{else}\ [], \\
\qquad\quad \mathit{choice}\ (g,h) & & = \mathit{append} \circ (g \vartriangle h), \\
\qquad\quad \mathit{cat}\ (g,h) & & = \mathit{concat} \circ \mathit{List}\ h \circ g \\
\quad \textbf{end}. & &
\end{array}
$$

The $\triangle$ operator (pronounced *split*) used in the definition of *choice* is defined in Appendix A.1. It is used to apply both parsers to the same input. The results of the two parsers are then appended. Sequential composition is implemented by applying the second parser to every result returned by the first and concatenating the results.

7.2.1 Partial Functions and Left-Recursion

As mentioned above, the type of a parser is a *partial* function from input strings to successes. One way of modelling a partial function is as a total function that might compute the value $\bot$. Here a different approach turns out to more convenient: partial functions are viewed as *relations*. From this point onwards the type of a parser is considered to be a relation:

$$\textbf{type}\ \mathit{Parse}\ \tau = [[\tau]] \sim [\tau].$$

Notice that the parser combinator definitions given above are perfectly valid on relations. Firstly, the parsers defined by *empty* and *token* are *total* functions, which can be viewed as relations. Secondly, $G \mid H$ and $G \cdot H$ are defined for relations G and H, because they are defined in terms of *List*, $\triangle$ and $\circ$. Each of these operations is defined on relations as well as functions.

Relations ordered by subset inclusion form a complete lattice, so recursive definitions are interpreted as least fixed points in this lattice. This model can be used to explain why *left-recursive* parsers fail. Consider the left-recursive grammar *baaa*, defined above. It defines S by the recursive equation:

$$S = (S \cdot a) \mid b,$$

which expands to:

$$S = \mathit{append} \circ ((\mathit{concat} \circ \mathit{List}\ S \circ a) \triangle b).$$

The least solution of this equation is the empty relation $\emptyset$, because *List*, $\triangle$ and $\circ$ are strict in $\emptyset$. That is, for all relations R:

$$\begin{aligned} \mathit{List}\ \emptyset &= \emptyset, \\ R \circ \emptyset &= \emptyset, \\ \emptyset \circ R &= \emptyset, \\ R \triangle \emptyset &= \emptyset. \end{aligned}$$

That $S = \emptyset$ means that S fails on all inputs.

7.2.2 Ambiguity

A grammar is *ambiguous* if there are strings in the language that can be derived in more than one way. The parser *cond* is ambiguous, because the string "aacbc" can be derived in the following two ways:

$$\begin{aligned} S &\to \mathrm{a}S &&\to \mathrm{aa}S\mathrm{b}S &&\to \mathrm{aacbc}, \\ S &\to \mathrm{a}S\mathrm{b}S &&\to \mathrm{aa}S\mathrm{b}S &&\to \mathrm{aacbc}. \end{aligned}$$

In the implementation described above, this ambiguity manifests itself by duplicate results appearing in the results list. Applying the function *cond* to the input "aacbc" produces the result ["bc", "", ""]. The double appearance of the empty string in this list indicates that "aacbc" was parsed in two distinct ways.

A parser is considered to be ambiguous if there exists an input string for which it produces duplicate results. The analysis presented here is a *conservative* analysis of ambiguity. If a parser passes the test, then it is *definitely* non-ambiguous. If it fails the test, then it *might* be ambiguous. A second property of the analysis is that it conservatively predicts failure of the parser. (Failure is typically caused by left-recursion.) If the analysis predicts failure on a given input, then the parser will *definitely* fail on that input, but the parser *may* also fail on inputs that were not predicted by the analysis.

7.3 The Analysis

The analysis is derived in two parts to simplify the calculations. The first part is the crucial abstract interpretation. The second abstract interpretation (presented in §7.5) is a fairly mundane transformation which is required to make the domains finitely computable.

LL(k) analysis is based on analysing the *first* and *follow* sets of the non-terminals of a grammar. For a thorough introduction to LL(k) analysis, see Aho and Ullman [7, pages 334–368]. The analysis presented here is based on similar ideas, but the two analyses are not necessarily interchangeable. The focus is on proving the analysis correct with respect to the semantics of the combinators, rather than proving that it is equivalent to Aho and Ullman's analysis. The relationship between the two analyses and why it is not possible to fit Aho and Ullman's analysis into the abstract interpretation framework is discussed in §7.7.

The key to the analysis is the function *take*. It retains the first k elements of a list and discards the rest:

$$\begin{array}{lcl} \mathit{take} & :: & \forall\alpha.\ [\alpha]_k \leftarrow [\alpha] \leftarrow (k : \mathit{Nat}), \\ \mathit{take}\ 0\ \mathit{xs} & = & [], \\ \mathit{take}\ n\ [] & = & [], \\ \mathit{take}\ n\ (x : \mathit{xs}) & = & x\ :\ \mathit{take}\ (n{-}1)\ \mathit{xs}. \end{array}$$

An informal *dependent type* notation has been used in the type of *take*, to indicate that it produces a list of length $\leq k$. These annotations are added purely as a hint; the formal type of *take* is:

$$\mathit{take} :: \forall\alpha.\ [\alpha] \leftarrow [\alpha] \leftarrow \mathit{Nat}.$$

The idea is to analyse parsers by considering just the first k tokens of their input and the first k tokens of their output. This idea corresponds to the first sets and follow sets of Aho and Ullman's analysis. The result of analysing a parser p is a relation of type:

type $Parse_k\ \tau = [[\tau]_k] \sim [\tau]_k$.

Dependent type notation has been used here again to indicate that the strings are limited to length k. Note though that the list of results can still have an arbitrary length. The relation relates all string prefixes to all results lists that might result from parsing a string with that prefix. Formally the analysis is specified by the following abstraction function:

$$\begin{array}{lcl} abs & :: & \forall\tau.\ Parse_k\ \tau \leftarrow Parse\ \tau \leftarrow (k : Nat), \\ abs\ k\ p & = & List\ (take\ k) \circ p \circ (take\ k)^{\cup}. \end{array}$$

The relation $(take\ k)^{\cup}$ relates a prefix to all strings that have that prefix. In other words, $abs\ k\ p$ generates all possible strings, applies p to them all and then simplifies the results with $take\ k$. As usual, abs is not a computable function. Instead it is a mathematical tool for deriving an analysis that *is* computable.

7.3.1 How to Interpret the Results

It is perhaps not immediately obvious how abs can be used to prove that a grammar is unambiguous. Suppose the grammar p is ambiguous. This means that, for some input string, say "abcdefgh", it produces duplicate results, such as: ["fgh", "fgh"]. Suppose the analysis is being conducted with $k = 2$. Then one of the pairs in the relation $abs\ k\ p$ will be (["fg", "fg"], "ab"). So by inspecting the relation $abs\ k\ p$, it is possible to determine that input strings prefixed by "ab" might lead to an ambiguous parse. If, on the other hand, the range of $abs\ k\ p$ does not contain any lists with duplicates, then the parser is definitely not ambiguous.

The results can also be used to detect failure (possibly caused by left recursion). If, for example, the prefix "ab" is not in the domain of the relation, then any string with the prefix "ab" causes the parser to fail. If there is a possibility that strings prefixed by "ab" will not cause the parser to fail, then "ab" will be in the domain of the relation.

Finally the results can sometimes be used to optimise the parser. The standard implementation of $G \mid H$ applies both G and H to the input and their results are concatenated. Typically, though, only one of the two parsers will return any results, so any time spent executing the other is wasted. This can often be predicted by looking at the analysis results for G and H. For example:

$$\begin{array}{lcl} G & = & \{\ [\text{“b”}] \leftarrowtail \text{“a”},\ [] \leftarrowtail \text{“a”},\ [] \leftarrowtail \text{“b”}\ \} \\ H & = & \{\ [\text{“b”}] \leftarrowtail \text{“b”},\ [] \leftarrowtail \text{“a”},\ [] \leftarrowtail \text{“b”}\ \} \end{array}$$

It is clear from these analysis results that G returns no results unless the first token of input is 'a' and H only returns results if the first token is 'b'. Therefore the implementation could be optimised by adding a conditional statement to inspect the first token and choose either G or H accordingly. A second property that can be deduced from these results is that the parser never returns more than one result. This means that a list is not needed to encode the results. Instead the simpler *Maybe* datatype could be used.

7.4 Deriving the Analysis

The concretisation function corresponding to the abstraction function given above is:

$$\begin{array}{l} con \quad :: \ \forall\tau.\ Parse\ \tau \leftarrow Parse_k\ \tau \leftarrow (k : Nat), \\ con\ k\ q \ = \ List\ ((take\ k)^{\cup})\ \circ\ q\ \circ\ take\ k. \end{array}$$

The proof that these functions form a Galois connection is a simple application of function shunting (§4.1):

$$\begin{array}{ll} & abs\ k\ p \subseteq q \\ \equiv & \quad \{ \text{ Definition } abs \text{ } \} \\ & List\ (take\ k)\ \circ\ p\ \circ\ (take\ k)^{\cup} \subseteq q \\ \equiv & \quad \{ \text{ Function shunting } \} \\ & p\ \circ\ (take\ k)^{\cup} \subseteq List\ ((take\ k)^{\cup})\ \circ\ q \\ \equiv & \quad \{ \text{ Function shunting } \} \\ & p \subseteq List\ ((take\ k)^{\cup})\ \circ\ q\ \circ\ take\ k \\ \equiv & \quad \{ \text{ Definition } con \text{ } \} \\ & p \subseteq con\ k\ q. \end{array}$$

This Galois connection can be used to instantiate safety for free (§5.4). A typical parser S has type $\forall g.\ g \leftarrow Parsing\ T\ g.$ for some type T of tokens. Safety for free states that for any implementation P of the *Parsing* interface:

$$S\ P \quad \subseteq \quad [(g \mapsto abs\ k), (g \mapsto (con\ k)^{\cup})]_t\ S\ P$$

(With $t = g \leftarrow Parsing\ T\ g.$) This statement can equivalently be written as:

$$abs\ k\ (S\ P) \quad \subseteq \quad S\ Q$$

Where Q is an implementation of the *Parsing* interface, specified by:

$$\begin{array}{lll} Q & :: & \forall\tau.\ Parsing\ \tau\ (Parse_k\ \tau), \\ Q & = & \textbf{struct}\ \ empty \qquad\quad = \ abs\ k\ (P.empty), \\ & & \qquad\quad token\ \ t \qquad = \ abs\ k\ (P.token\ t), \\ & & \qquad\quad choice\ (g,h) \ = \ abs\ k\ \ (P.choice\ \ (con\ k\ g,\ con\ k\ h)), \\ & & \qquad\quad cat \quad\ (g,h) \ = \ abs\ k\ \ (P.cat\ \ (con\ k\ g,\ con\ k\ h)) \\ & & \textbf{end}. \end{array}$$

This specification of Q is expanded below, using the definition of P given on page 37.

7.4.1 Deriving the Implementation of Q

Each of the four methods of Q can be derived individually by substituting the definitions of *abs*, *con* and P. A simple cancellation property of *take* is useful in many of the derivations:

$take\ k \circ (take\ k)^{\cup} :: \forall \alpha.\ [\alpha]_k \leftarrow [\alpha]_k,$
$take\ k \circ (take\ k)^{\cup} = id.$

This is by no means a unique property of *take*. It holds for any function f, provided that for all x there exists a y such that $x = f\ y$. This means that the range of f includes every element of the result type. The type signature given above is therefore quite important, because *take* can produce any list, provided that its length is $\leq k$. It is in this context that the property is applied below.

The definition of $Q.empty$ is the easiest to derive:

$$
\begin{array}{ll}
 & Q.empty \\
= & \quad \{\ \text{Specification } Q\ \} \\
 & abs\ k\ P.empty \\
= & \quad \{\ \text{Definitions } abs, P.empty\ \} \\
 & List\ (take\ k) \circ (\lambda\, xs.\ [xs]) \circ (take\ k)^{\cup} \\
= & \quad \{\ \text{Cancellation property of } take\ \} \\
 & \lambda\, xs.\ [xs].
\end{array}
$$

Note that the definition of $Q.empty$ is identical to the definition of $P.empty$. This also turns out to be the case for $Q.choice$ and $Q.cat$. Only the definitions of $Q.token$ and $P.token$ differ.

$Q.choice$ is derived below and is found to have the same definition as $P.choice$:

$$
\begin{array}{ll}
 & Q.choice\ (g,h) \\
= & \quad \{\ \text{Specification } Q\ \} \\
 & abs\ k\ \ (P.choice\ (con\ k\ g,\ con\ k\ h)) \\
= & \quad \{\ \text{Definition } abs\ \} \\
 & List\ (take\ k) \circ (P.choice\ (con\ k\ g,\ con\ k\ h)) \circ (take\ k)^{\cup} \\
= & \quad \{\ \text{Definition } P.choice\ \} \\
 & List\ (take\ k) \circ append \circ (con\ k\ g\ \vartriangle\ con\ k\ h) \circ (take\ k)^{\cup} \\
= & \quad \{\ \text{Free Theorem of } append \text{ (see Wadler [103])}\ \} \\
 & append \circ (List\ (take\ k) \times List\ (take\ k)) \circ \\
 & \quad (con\ k\ g\ \vartriangle\ con\ k\ h) \circ (take\ k)^{\cup} \\
= & \quad \{\ \text{Definition } con\ \} \\
 & append \circ (List\ (take\ k) \times List\ (take\ k)) \circ \\
 & \quad ((List\ ((take\ k)^{\cup}) \circ g \circ take\ k)\ \vartriangle \\
 & \qquad (List\ ((take\ k)^{\cup}) \circ h \circ take\ k)) \circ (take\ k)^{\cup} \\
= & \quad \{\ \text{Property } \vartriangle\ \} \\
 & append \circ (List\ (take\ k) \times List\ (take\ k)) \circ \\
 & \quad (List\ ((take\ k)^{\cup}) \times List\ ((take\ k)^{\cup})) \circ (g \vartriangle h) \circ \\
 & \quad take\ k \circ (take\ k)^{\cup} \\
= & \quad \{\ take\ k \circ (take\ k)^{\cup} = id\ \} \\
 & append \circ (g \vartriangle h).
\end{array}
$$

The derivation of $Q.cat$ proceeds similarly:

$$
\begin{array}{ll}
 & Q.cat\ (g,h) \\
= & \quad \{\ \text{Specification } Q\ \} \\
 & abs\ k\ \ (P.cat\ \ (con\ k\ g,\ con\ k\ h)) \\
= & \quad \{\ \text{Definition } abs\ \} \\
 & List\ \ (take\ k)\ \circ\ (P.cat\ \ (con\ k\ g,\ con\ k\ h))\ \circ\ (take\ k)^{\cup} \\
= & \quad \{\ \text{Definition } P.cat\ \} \\
 & List\ \ (take\ k)\ \circ\ concat\ \circ\ List\ \ (con\ k\ h)\ \circ\ con\ k\ g\ \circ\ (take\ k)^{\cup} \\
= & \quad \{\ \text{Free Theorem of } concat \text{ (see Wadler [103])}\ \} \\
 & concat\ \circ\ List\ \ (List\ \ (take\ k))\ \circ\ List\ \ (con\ k\ h)\ \circ\ con\ k\ g\ \circ \\
 & \quad (take\ k)^{\cup} \\
= & \quad \{\ \text{Definition } con\ \} \\
 & concat\ \circ\ List\ \ (List\ \ (take\ k))\ \circ \\
 & \quad List\ \ (List\ \ ((take\ k)^{\cup})\ \circ\ h\ \circ\ take\ k)\ \circ\ List\ \ ((take\ k)^{\cup}) \\
 & \quad \circ\ g\ \circ\ take\ k\ \circ\ (take\ k)^{\cup} \\
= & \quad \{\ List\ R \circ List\ S\ =\ List\ (R \circ S)\ \} \\
 & concat\ \circ\ List\ \ (List\ \ (take\ k \circ (take\ k)^{\cup}))\ \circ\ List\ h\ \circ \\
 & \quad List\ \ (take\ k \circ (take\ k)^{\cup})\ \circ\ g\ \circ\ take\ k\ \circ\ (take\ k)^{\cup} \\
= & \quad \{\ take\ k \circ (take\ k)^{\cup}\ =\ id\ \} \\
 & concat \circ List\ h \circ g.
\end{array}
$$

As mentioned above, $Q.token$ is the only method of Q that differs from its counterpart in P. Expanding the definitions reveals that $Q.token$ is the following relation:[2]

$$
\begin{array}{l}
(zss, xs)\ \in\ abs\ k\ \ (token\ t)\ \equiv \\
\quad (zss = [] \ \wedge\ (\nexists ys.\ xs\ =\ t : ys))\ \vee \\
\quad\quad (\exists ys.\ zss\ =\ [ys]\ \wedge\ xs\ =\ take\ k\ \ (t : ys)).
\end{array}
$$

This is the definition of a *non-deterministic* relation, which is a symptom of the analysis not being exact: information is lost in the analysis of the *token* operation. To illustrate the difference between $P.token$ and $Q.token$, consider a simple situation in which there are just two tokens 'a' and 'b' and the analysis is conducted with $k = 2$. Examples of pairs related by the total function $P.token$ 'a' are:

$$
\begin{array}{rcl}
[\text{“bba”}] & \mapsto & \text{“abba”} \\
[] & \mapsto & \text{“bab”} \\
[\text{“aaaaa”}] & \mapsto & \text{“aaaaaa”} \\
[\text{“”}] & \mapsto & \text{“a”} \\
[] & \mapsto & \text{“”}
\end{array}
$$

The relation $Q.token$ 'a' contains only a finite set of pairs, all of which are listed below:

[2] In the $\exists$-clauses used here, ys is a list of length $\leq k$.

$$
\begin{array}{rcl@{\qquad}rcl}
[] & \mapsto & \text{“”} & [\text{“ba”}] & \mapsto & \text{“ab”} \\
[\text{“”}] & \mapsto & \text{“a”} & [\text{“bb”}] & \mapsto & \text{“ab”} \\
[\text{“a”}] & \mapsto & \text{“aa”} & [] & \mapsto & \text{“b”} \\
[\text{“aa”}] & \mapsto & \text{“aa”} & [] & \mapsto & \text{“ba”} \\
[\text{“ab”}] & \mapsto & \text{“aa”} & [] & \mapsto & \text{“bb”} \\
[\text{“b”}] & \mapsto & \text{“ab”} & & &
\end{array}
$$

Note that the inputs “aa” and “ab” produce non-deterministic results. This is due to “aa” and “ab” being approximations, resulting from the application of *take* 2 to an input string. The length of this string is unknown, so it could be two tokens long or it could be a longer string such as “aabab” or “aaab” . This lack of information about the third token leads to the non-determinism in $Q.token$.

7.5 Ensuring Termination

If the analysis is be implemented by iteratively computing a least fixed point, then the set of possible results must be finite to ensure termination. The type $Parse_k$ does not satisfy this criterion, because the list of results can be arbitrarily long (if elements are duplicated). A simple solution to this problem is to immediately stop iterating if duplicates appear in the results list. This guarantees termination, because the set of possible results is finite if duplication is not allowed. It also produces correct results: iteration produces an *increasing* sequences of results, which means that pairs are added to the relation but never removed. Therefore duplications never disappear after they have appeared, so it is a safe to announce failure as soon as a duplication appears.

If the computation of least fixed points were a built-in language feature, then it would not be possible to modify the iteration algorithm so that it bails out early in certain cases. Therefore the proper solution is to find a representation in which the set of possible results is finite. This solution requires slightly more work, because a second abstract interpretation is needed.

The new representation is based on the observation that the only relevant properties of the results list are the set of elements in the list and the set of elements that are *duplicated*. So no relevant information is lost by applying the function *crop* to the results list:

$$
\begin{array}{l}
crop \;::\; \forall\alpha.\; Crop\ \alpha \leftarrow [\alpha], \\
crop \;=\; removeTriples \circ sort.
\end{array}
$$

where *Crop* is a type synonym for lists:

$$\textbf{type}\ Crop\ \alpha = [\alpha].$$

The implicit understanding is that values of type *Crop* are sorted lists with no more than two occurrences of every element. The function *removeTriples*, used above, filters a list such that there are no more than two copies of every element. Its definition is omitted. The number of cropped results lists is finite, because no element can

appear more than twice. Sorting the list is an additional optimisation that reduces the number of possibilities by ensuring that different permutations of the same list are considered equal.

The abstraction function based on *crop* is:

$$\begin{aligned} &abs &&:: \forall\tau.\ Analysis_k\ \tau \leftarrow Parse_k\ \tau, \\ &abs\ R &&= crop \circ R. \end{aligned}$$

where:

$$\textbf{type}\ Analysis_k\ \tau\ =\ (Crop\ [\tau]_k)\ \sim\ [\tau]_k.$$

The corresponding concretisation function is:

$$\begin{aligned} &con &&:: \forall\tau.\ Parse_k\ \tau \leftarrow Analysis_k\ \tau, \\ &con\ R &&= crop^{\cup} \circ R. \end{aligned}$$

The proof that these functions form a Galois connection is almost identical to the proof in §7.4.

7.5.1 Properties of *crop*

In the derivation of the second abstract interpretation some properties of *crop* are needed. The proofs of these properties are straightforward and are, therefore, omitted. Firstly,

$$crop \circ append\ =\ crop\ \circ\ append\ \circ\ (crop \times crop).$$

In other words, performing a preliminary crop before appending has no effect on the final results. A more advanced version of this property would be to define a function *merge*, such that:

$$crop \circ append\ =\ merge\ \circ\ (crop \times crop).$$

The implementation of *merge* could be more efficient than simply evaluating $crop \circ append$, because it could take advantage of the lists already being sorted. The second property is a similar property concerning *concat*:

$$crop \circ concat\ =\ crop\ \circ\ concat\ \circ\ crop\ \circ\ List\ crop.$$

The more advanced version of this property is:

$$crop \circ concat\ =\ mergeAll\ \circ\ crop\ \circ\ List\ crop.$$

Here, *mergeAll* is similar function to *merge* that merges a list of lists.

Finally, a relator *Crop* is needed, satisfying the specification that for all relations R:

$$Crop\ R\ =\ crop\ \circ\ List\ R\ \circ\ crop^{\cup}.$$

(This property is used in the derivation of the *cat* operator.) This specification of *Crop* needs to be transformed to an executable implementation:

$$
\begin{array}{ll}
 & (as, bs) \in (crop \circ List\ R \circ crop^{\cup}) \\
\equiv & \quad \{ \text{ Composition } \} \\
 & (\exists xs, ys.\ (xs, ys) \in List\ R \wedge as = crop\ xs \wedge bs = crop\ ys) \\
\equiv & \quad \{ \text{ Define } zs = zip\ xs\ ys \} \\
 & (\exists zs.\ elems\ zs \subseteq R \wedge as = crop\ (map\ fst\ zs) \wedge \\
 & \qquad bs = crop\ (map\ snd\ zs)) \\
\equiv & \quad \{ \text{ Define } zs' = crop\ zs \} \\
 & (\exists zs'.\ elems\ zs' \subseteq R \wedge as = crop\ (map\ fst\ zs') \wedge \\
 & \qquad bs = crop\ (map\ snd\ zs')).
\end{array}
$$

(The function *elems* converts a list to a set. It is used here to convert a list of pairs to a relation.) This expression is computable because there are only a finite number of zs' to be tested.

One further property that is used below is that $crop \circ crop^{\cup} = id$. This property is true for the same reasons as the cancellation property of *take*, discussed above. As with *take* there is a side-condition that the input to $crop \circ crop^{\cup}$ must be a cropped list for the property to hold.

7.5.2 Deriving the Analysis

The second abstract interpretation proceeds in exactly the same way as the first. The goal this time is derive (from Q) a new parsing module R that operates on values of type $Analysis_k$:

$$R :: \forall \tau.\ Parsing\ \tau\ (Analysis_k\ \tau).$$

Empty and singleton lists are left unchanged by *crop*, so the definitions of *R.empty* and *R.token* are identical to their counterparts in Q.

R.choice is derived as follows:

$$
\begin{array}{ll}
 & R.choice\ (g,h) \\
= & \quad \{ \text{ Specification } R \} \\
 & abs\ (Q.choice\ (con\ g,\ con\ h)) \\
= & \quad \{ \text{ Definitions } abs, con \text{ and } Q.choice \} \\
 & crop \circ append \circ ((crop^{\cup} \circ g) \vartriangle (crop^{\cup} \circ h)) \\
= & \quad \{ \text{ Property of } crop \circ append \} \\
 & merge \circ (crop \times crop) \circ ((crop^{\cup} \circ g) \vartriangle (crop^{\cup} \circ h)) \\
= & \quad \{\ crop \circ crop^{\cup} = id \ \} \\
 & merge \circ (g \vartriangle h).
\end{array}
$$

The derivation of *R.cat* is similar:

$$
\begin{array}{ll}
 & R.cat\ (g,h) \\
= & \quad \{ \text{ Specification } R \ \} \\
 & abs\ \ (Q.cat\ \ (con\ g,\ con\ h)) \\
= & \quad \{ \text{ Definitions } abs, \text{ } con \text{ and } Q.cat \ \} \\
 & crop \circ concat \circ List\ (crop^{\cup} \circ h) \circ crop^{\cup} \circ g \\
= & \quad \{ \text{ Property of } crop \circ concat \ \} \\
 & mergeAll \circ crop \circ List\ crop \circ List\ (crop^{\cup} \circ h) \circ crop^{\cup} \circ g \\
= & \quad \{ \ crop \circ crop^{\cup} = id \ \} \\
 & mergeAll \circ crop \circ List\ h \circ crop^{\cup} \circ g \\
= & \quad \{ \text{ Definition of } Crop \ \} \\
 & mergeAll \circ Crop\ h \circ g.
\end{array}
$$

This concludes the derivation of the parsing module R. It can be used to analyse grammars and the iteration process is guaranteed to terminate. For simplicity though the less sophisticated approach is used in the examples below.

7.6 Examples

The results of analysing the grammars *aaab*, *baaa*, *baaa′* and *cond* are discussed here briefly.

Recall that the *aaab* grammar is:

$$S \ ::= \ \mathrm{a}S \mid \mathrm{b}$$

Iteration is used to compute the value of S. In other words, the algorithm starts with S equal to the empty relation. The new value of S is then computed repeatedly with the equation given above until it ceases to change. If $k = 1$ then the result of this process is a relation containing the following pairs:

[]	↤	“”	[“b”]	↤	“a”
[]	↤	“a”	[“”]	↤	“b”
[“”]	↤	“a”	[“a”]	↤	“b”
[“a”]	↤	“a”	[“b”]	↤	“b”

There are no duplicated items in these results lists, so the grammar is not ambiguous.

The analysis results can sometimes be used to optimise the parser. The choice in the definition of *aaab* can be optimised, because aS only succeeds if the first token is ‘a’ and the primitive parser b only succeeds if the first token is ‘b’. This is apparent from the results of the analysis, because the relation for aS is:

[]	↤	“”	[“a”]	↤	“a”
[]	↤	“a”	[“b”]	↤	“a”
[“”]	↤	“a”	[]	↤	“b”

and the relation for b is:

[]	↤	“”		[“”]	↤	“b”	
[]	↤	“a”		[“a”]	↤	“b”	
				[“b”]	↤	“b”	

It is clear from these results that the choice can be made based on only the first token of input. It is also clear that the parser never returns more than one result, so the result could be encoded as a *Maybe* rather than a list.

The relation produced by analysing *baaa* with any positive value of k is empty, which means that *baaa* will fail on any input. This is due to it being left-recursive. In contrast, the relation for $baaa'$ (with $k = 1$) is:

[]	↤	“”		[“”,“a”]	↤	“b”	
[]	↤	“a”		[“b”]	↤	“b”	
[“”]	↤	“b”		[“b”,“a”]	↤	“b”	

This relation indicates that the parser is not ambiguous.

Finally, *cond* fails the test as expected. The analysis bails out with the following relation:

[]	↤	“”		[“b”,“a”]	↤	“a”	
[]	↤	“a”		**[“b”,“b”]**	↤	“a”	
[“”]	↤	“a”		[“b”,“c”]	↤	“a”	
[“”,“”]	↤	“a”		[“c”]	↤	“a”	
[“”,“a”]	↤	“a”		[“c”,“”]	↤	“a”	
[“”,“b”]	↤	“a”		[“c”,“a”]	↤	“a”	
[“”,“c”]	↤	“a”		[“c”,“b”]	↤	“a”	
[“a”]	↤	“a”		**[“c”,“c”]**	↤	“a”	
[“a”,“”]	↤	“a”		[]	↤	“b”	
[“a”,“a”]	↤	“a”		[“”]	↤	“c”	
[“a”,“b”]	↤	“a”		[“a”]	↤	“c”	
[“a”,“c”]	↤	“a”		[“b”]	↤	“c”	
[“b”]	↤	“a”		[“c”]	↤	“c”	
[“b”,“”]	↤	“a”					

Some of the result lists in this relation contain duplicates, so *cond* fails the test. It also fails with larger values of k.

7.7 Related Work

The LL(k) analysis described by Aho and Ullman [7, pages 334–368] is divided into two stages. In the first stage, *first sets* are computed for every non-terminal in the grammar. The first set of a non-terminal N contains the prefixes (of length $\leq k$) of strings accepted by N. For example if $k = 2$, then the first set of the *aaab* grammar is {“b”, “ab”, “aa”}. In the second stage, *follow sets* are computed for every non-terminal in the grammar. The follow set of a non-terminal N is computed by

examining the contexts in which N appears. For example, if N appears in the production $S \rightarrow N\,T$, then the first set of T is included in the follow set of N. Follow sets are important for analysing ambiguity, because ambiguity can be caused by context. For example, a grammar that defines the strings "a" and "ab" is unambiguous on its own, but if it is used in a context where it might be followed by either "c" or "bc", then the string "abc" is ambiguous.

Aho and Ullman's algorithm is very difficult to apply to parser combinators due to its reliance on context to compute the follow sets. To compute the follow set of a non-terminal, one must know *every* context in which that non-terminal is used. A YACC-style parser generator could obtain this information by a simple syntactic analysis of the grammar, but this is not possible with embedded combinators. It would be particularly difficult if higher order functions were used in the definition of the grammar. The analysis presented here uses relations to avoid the need for contextual information. Reynolds's *abstraction theorem* [86] guarantees that the analysis is still valid in the presence of higher order functions. It seems that the analysis produces very similar results to Aho and Ullman's on standard grammars, but the conjecture that the analyses are, in fact, equivalent is unproved at present. Instead the correctness of the analysis has been proved with respect to the semantics of the combinators.

Swierstra and Duponcheel [96] have previously investigated the analysis of first and follow sets in a functional setting and encountered similar problems. They have shown how to compute first sets in Haskell[3], but have not found a method for analysing follow sets. On the page 12 of their paper (page 195 of the volume) they sum the problem up as follows:

> We may wonder why we cannot compute the union of all possible follow sets, so that we may statically decide if parsing will always be deterministic. In order to be able to compute this set however, we need access to all the places in the code where the function corresponding to this non-terminal is called. Unfortunately this information can be neither directly or indirectly computed from the functional parsers since we do not have an explicit representation of the parsers at hand. The programs we write may look like a grammar, but keep in mind that they actually are function definitions. The fact that at a specific location we call a specific function, which corresponds to the use of a specific non-terminal in the right hand side of production, is information which is not available to us explicitly. For this we would need a language with some form of reflection.

However, Swierstra and Duponcheel's primary goal is *optimisation*, not error checking. One of the benefits of LL(k) analysis is that the results can be used to optimise the parser. Aho and Ullman explain how to use the results to encode the parser as a finite state machine. The machine only needs to inspect the next k tokens of input to decide which state to move to next. Even with only first set analysis at their disposal, Swierstra and Duponcheel [96] manage to use the same ideas to improve

[3]Their method of computing first sets is an abstract interpretation, although they do not refer to it as such.

the performance of their parser combinators. Their observation is that, in order to improve performance, only the *current* context is relevant. This context can be computed dynamically and passed as an extra parameter. Swierstra and Duponcheel use this information to make their parsers *deterministic*, based on the assumption that the grammar is LL(1). The analysis presented here could be used to go one step further and statically optimise the parser. This could be done by applying conditional rewrite rules, based on the results of the analysis. Research on such rewriting technology is currently being done by the implementers of the Glasgow Haskell Compiler [34, pages 185–190], who are experimenting with a feature for specifying rewrite rules in Haskell.

7.8 Summary

Functional programmers are currently presented with a choice when they wish to write a parser: use a parser generator and benefit from the ambiguity test or use parser combinators and benefit from the superior expressive power. With the analysis presented here the best of both worlds can be achieved. The techniques that have been applied to parser combinators are also applicable to other DSELs. They could be used to improve error checking and enable more optimisations in a wide variety of domain specific areas.

Grammars are perhaps not the most exciting domain specific language, but they are important because parsing is such a well studied topic: if grammars can be handled well then this is good advert for DSELs in general. A problem that has prevented the LL(k) analysis described by Aho and Ullman [7, pages 334–368] from being applied to parser combinators is its reliance on context. A key contribution here is to derive a new analysis that does not rely on context.

Chapter 8

Attribute Grammars

The material in this chapter is a traditional example of abstract interpretation, rather than a DSEL. The semantics of attribute grammars are defined using the notation of Chapter 2. Then an abstract interpretation of the semantics is derived and used to test the definedness of attribute grammars. This chapter demonstrates the use of safety for free (§5.4) in a traditional application of abstract interpretation: a language is defined and then an abstract interpreter for the language is derived. Safety for free drastically simplifies the derivation of the abstract interpretation.

The definition of the semantics of attribute grammars given here is rather different from Knuth's [55] original formulation. The functional definition given here allows a calculational proof style to be employed. This is illustrated by proving Chirica and Martin's result [20] that attribute grammars can be evaluated by structural recursion. Using the results of Chapter 5 a new definedness test is also derived. The definedness test turns out to be slightly more powerful than Knuth's circularity test [55, 56]. It tests circularity, but it also encompasses Knuth's closure test

The material in this chapter was presented at TACAS 2002 [9]. The definition of attribute grammars given here is based on earlier work with de Moor and Swierstra [72].

8.1 A Brief Introduction to Attribute Grammars

Bird's *repmin* problem [13] is used as an illustration, throughout this chapter. Kuiper and Swierstra [58] have noted that repmin can be easily written as an attribute grammar and, due to its simplicity, it makes a nice example. In this section, the topic of attribute grammars is introduced by writing repmin as an attribute grammar. The input to repmin is a binary tree of integers. In the output tree, the leaf values have been replaced by the minimum leaf value. A sample run of repmin is shown in Figure 8.1.

Attribute Grammars are a formalism for annotating abstract syntax trees with attribute values. For example, an attribute grammar could be used to implement a type-checker by annotating every expression node with its type. The semantic rules of an attribute grammar are used to compute the attribute values. Every production of the grammar has its own semantic rules, so different productions can behave in

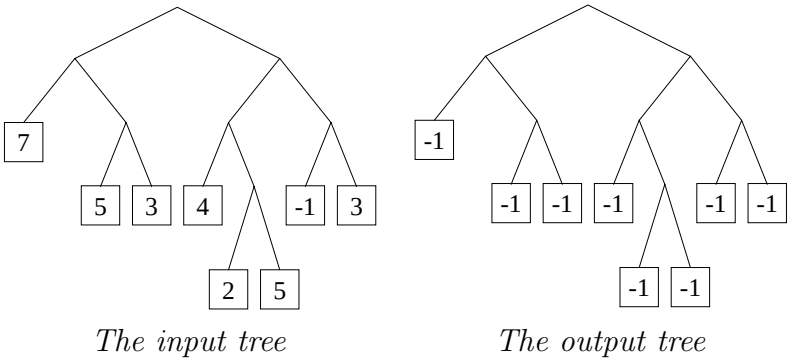


Figure 8.1: An example of the use of *repmin*

different ways. A distinction is drawn between inherited and synthesised attributes, so that they can be used to pass data both up and down the tree. The inherited attributes of a node are defined by the semantic rules of the production above, whereas the synthesised attributes are defined by the production below. Data is, therefore, passed down the tree by means of inherited attributes and up the tree by means of synthesised attributes.

To introduce attribute grammars by example, an attribute grammar for repmin is presented below, written using a simple pseudo code. The attribute grammar uses two synthesised attributes, named *ntree* (for new tree) and *locmin* (for local minimum). Furthermore, there is one inherited attribute, named *gmin* (for global minimum). The strategy is to first recursively compute the local minimum on all nodes. The global minimum, which equals the local minimum at the root, is then broadcast to all the leaves and the new tree is built recursively.

The production named *Root* rewrites the start symbol *Start* to *Tree*. It is in this production that the global minimum is defined to be equal to the local minimum at the root. The computation of the *ntree* synthesised attribute is a simple application of the *Root* constructor.

$$\begin{aligned} \mathit{Root} &: \mathit{Start} \rightarrow \mathit{Tree}, \\ \mathit{Start.ntree} &= \mathit{Root}\ \mathit{Tree.ntree}, \\ \mathit{Tree.gmin} &= \mathit{Tree.locmin}. \end{aligned}$$

At each binary node, the local minimum is obtained by taking the minimum of both subtrees; the global minimum is broadcast from the parent to both children. Here and below, indices are used to refer to successive occurrences of the non-terminal *Tree*.

$$\begin{aligned} \mathit{Fork} &: \mathit{Tree}_0 \rightarrow \mathit{Tree}_1\ \mathit{Tree}_2, \\ \mathit{Tree}_0\mathit{.ntree} &= \mathit{Fork}\ \mathit{Tree}_1\mathit{.ntree}\ \mathit{Tree}_2\mathit{.ntree}, \\ \mathit{Tree}_0\mathit{.locmin} &= \mathit{min}\ \mathit{Tree}_1\mathit{.locmin}\ \mathit{Tree}_2\mathit{.locmin}, \\ \mathit{Tree}_1\mathit{.gmin} &= \mathit{Tree}_0\mathit{.gmin}, \\ \mathit{Tree}_2\mathit{.gmin} &= \mathit{Tree}_0\mathit{.gmin}. \end{aligned}$$

Finally, the local minimum of a leaf is its value, and the new tree is a leaf with the global minimum as its value:

$$\begin{aligned} \mathit{Leaf} &: \mathit{Tree} \to \mathit{Val}, \\ \mathit{Tree.ntree} &= \mathit{Leaf}\ \ \mathit{Tree.gmin}, \\ \mathit{Tree.locmin} &= \mathit{Val.value}. \end{aligned}$$

The notation used above is modelled on the notation used in attribute grammar systems such as the Synthesiser Generator [85] or LRC [59]. How the semantic rules of an attribute grammar can be encoded as a function is discussed in §8.4.

8.2 Sequences

The definition of the semantics of attribute grammars given in this chapter uses sequences rather than lists, because lists cannot be used in least fixed point computations. (See the discussion in §3.3.4.) Sequences are important in the semantics of attribute grammars, because they are used in the definition of rose trees, which are used to define abstract syntax trees.

The definition of a sequence is:[1]

$$\textbf{type}\ \mathit{Seq}\ \alpha = \alpha \leftarrow \mathit{Nat}^+.$$

That is, a sequence is a function. Given the position of an element (as a natural number), the sequence returns the value stored at that position. This representation of sequences allows for infinitely long sequences, but every sequence used in this chapter is finite. The length of a finite sequence xs is the smallest integer n such that $xs\ k = \bot$, for all $k > n$.

Recursive definitions of sequences are valid, because they are ordered pointwise as discussed in §3.3.4. If A is a poset then $\mathit{Seq}\ A$ is a poset and if A is a complete lattice then $\mathit{Seq}\ A$ is a complete lattice.

8.2.1 Operations on Sequences

In this section, the counterparts of the list operations *map* and *zipWith* are defined on sequences. For the duration of this chapter, the names *map* and *zipWith* refer to the functions defined here. There should be no confusion with the functions defined in Appendix A.2, because normal lists are not used in this chapter.

The definition of *map* on sequences simply uses function composition:

$$\begin{aligned} &\mathit{map} \quad :: \forall\alpha,\beta.\ \mathit{Seq}\ \beta \leftarrow \mathit{Seq}\ \alpha \leftarrow (\beta \leftarrow \alpha), \\ &\mathit{map}\ f\ xs = f \circ xs. \end{aligned}$$

The definition of *zipWith* on sequences uses the S' combinator, which some readers might recognise from Turner's work on implementing functional languages [98, 99]. The definition of S' is given in Appendix A.3.

[1] Nat^+ is the type of positive, non-zero integers.

$$zipWith \;::\; \forall\alpha,\beta,\gamma.\; Seq\ \gamma \leftarrow Seq\ \beta \leftarrow Seq\ \alpha \leftarrow (\gamma \leftarrow \beta \leftarrow \alpha),$$
$$zipWith \;=\; S'.$$

A specialisation of *zipWith* that is frequently useful is *appSeq*. It is used to apply a sequence of functions pointwise to a sequence of values:

$$appSeq \;::\; \forall\alpha,\beta.\; Seq\ \beta \leftarrow Seq\ \alpha \leftarrow Seq\ (\beta \leftarrow \alpha),$$
$$appSeq \;=\; zipWith\ app.$$

A simple expansion of this definition shows that *appSeq* is equivalent to the well-known S combinator. (See Appendix A.3.) The S combinator also appeared in the abstraction rule, given in §3.5. In the context of sequences, the abstraction rule has the following form:

$$(\forall xs.\ \mu\ (appSeq\ xs) \;=\; map\ \mu\ xs).$$

In other words, computing the fixed point over the whole sequence at once is equivalent to computing the fixed point of each element individually.

The equivalence of *map*, *zipWith*, and *appSeq* to such simple combinators makes them easy to manipulate. For example, here is a proof of a little lemma that is used later.

Lemma 11 *For all binary operators* $\oplus$ *and* $\otimes$ *and sequences* xs, ys *and* zs*:*

$$zipWith\ (\oplus)\ xs\ (zipWith\ (\otimes)\ ys\ zs) \;=$$
$$\quad appSeq\ (zipWith\ (\circ)\ (map\ (\oplus)\ xs)\ (map\ (\otimes)\ ys))\ zs.$$

Proof. The proof is by showing that, for every k, the two sequences have equal elements at position k. For conciseness, the functions *zipWith*, *appSeq* and *map* have been replaced with their combinator definitions in the calculation below.

$$\begin{aligned}
S'\ c_1\ xs\ (S'\ c_2\ ys\ zs)\ k &= c_1\ (xs\ k)\ (S'\ c_2\ ys\ zs\ k) \\
&= c_1\ (xs\ k)\ (c_2\ (ys\ k)\ (zs\ k)) \\
&= (c_1\ (xs\ k)\ \circ\ c_2\ (ys\ k))\ (zs\ k) \\
&= (S'\ (\circ)\ (c_1 \circ xs)\ (c_2 \circ ys)\ k)\ (zs\ k) \\
&= S\ (S'\ (\circ)\ (c_1 \circ xs)\ (c_2 \circ ys))\ zs\ k
\end{aligned}$$

In contrast to this proof, the function names *map*, *zipWith* and *appSeq* are used in preference to the combinators for the rest of this chapter, because the function names tend to read more intuitively than the combinators.

8.3 Rose Trees

Rose trees are a well-known datatype in Haskell. (See Bird [14, pages 195–201].) The definition given here is slightly different, because sequences are used instead of lists. A node of a rose tree contains a datum and zero or more children:

$$\textbf{data}\ \mathit{Rose}\ \alpha\ =\ \mathit{Node}\ \alpha\ (\mathit{Forest}\ \alpha).$$

A forest is a sequence of trees:

$$\textbf{type}\ \mathit{Forest}\ \alpha\ =\ \mathit{Seq}\ (\mathit{Rose}\ \alpha).$$

Although this definition offers the possibility of a rose tree being infinitely deep or infinitely wide, the rose trees used in this chapter are neither. The ordering on rose trees is the standard pointwise ordering, which means that if A is a complete lattice, then *Rose* A is also a complete lattice.

8.3.1 Operations on Rose Trees

Rose trees can be taken apart with *split*:

$$\begin{array}{l}\mathit{split} :: \forall\alpha.\ (\alpha,\ \mathit{Forest}\ \alpha) \leftarrow \mathit{Rose}\ \alpha,\\ \mathit{split}\ (\mathit{Node}\ x\ ts)\ =\ (x, ts).\end{array}$$

They can be put back together with *build*:

$$\begin{array}{l}\mathit{build} :: \forall\alpha.\ \mathit{Rose}\ \alpha \leftarrow (\alpha,\ \mathit{Forest}\ \alpha),\\ \mathit{build}\ (x, ts)\ =\ \mathit{Node}\ x\ ts.\end{array}$$

A useful shorthand is the *root* function:

$$\begin{array}{l}\mathit{root} :: \forall\alpha.\ \alpha \leftarrow \mathit{Rose}\ \alpha,\\ \mathit{root}\ =\ \mathit{fst} \circ \mathit{split}.\end{array}$$

The counterpart of *map* on rose trees is *mapRose*:

$$\begin{array}{l}\mathit{mapRose} :: \forall\alpha,\beta.\ \mathit{Rose}\ \beta \leftarrow \mathit{Rose}\ \alpha \leftarrow (\beta \leftarrow \alpha),\\ \mathit{mapRose}\ f\ (\mathit{Node}\ x\ ts)\ =\ \mathit{Node}\ (f\ x)\ (\mathit{map}\ (\mathit{mapRose}\ f)\ ts).\end{array}$$

The counterpart of *appSeq* is:

$$\begin{array}{l}\mathit{appRose} :: \forall\alpha,\beta.\ \mathit{Rose}\ \beta \leftarrow \mathit{Rose}\ \alpha \leftarrow \mathit{Rose}\ (\beta \leftarrow \alpha)\\ \mathit{appRose}\ (\mathit{Node}\ f\ fs)\ (\mathit{Node}\ x\ xs)\ =\\ \quad \mathit{Node}\ (f\ x)\ (\mathit{zipWith}\ \mathit{appRose}\ fs\ xs).\end{array}$$

The fold function for rose trees is:

$$\begin{array}{l}\mathit{foldRose} :: \forall\alpha,\beta.\ \beta \leftarrow \mathit{Rose}\ \alpha \leftarrow (\beta \leftarrow (\alpha,\ \mathit{Seq}\ \beta)),\\ \mathit{foldRose}\ R\ =\ R \circ (\mathit{id} \times \mathit{map}\ (\mathit{foldRose}\ R)) \circ \mathit{split}.\end{array}$$

Note that this definition is also valid if R is a relation. Therefore *foldRose* can also be given a different type, indicating that it is a function from relations to relations:

$$\mathit{foldRose} :: \forall\alpha,\beta.\ (\beta \sim \mathit{Rose}\ \alpha) \leftarrow (\beta \sim (\alpha,\ \mathit{Seq}\ \beta)).$$

Bird and de Moor [15, pages 142–143] discuss the use of folds (catamorphisms) over relations and give many examples. They also prove a theorem (pages 142–143) that is very important in discussion of the definedness test for attribute grammars:

Theorem 9 (Hylomorphism) *Suppose R and S are relations:*

$$R :: C \sim (B,\ Seq\ C),$$
$$S :: A \sim (B,\ Seq\ A).$$

Then the expression

$$(foldRose\ R) \circ (foldRose\ S)^{\cup} :: C \sim A.$$

is equal to:

$$(\mu\, H.\ R \circ (id \times map\ H) \circ S^{\cup}).$$

This theorem is also known by functional programmers as *deforestation*, because it can be used to eliminate intermediate data-structures from programs.

8.4 The Semantics of Attribute Grammars

The approach in this section is to represent abstract syntax trees as rose trees. The semantics of attribute grammars are then defined as a least fixed point computation over rose trees.

8.4.1 Abstract Syntax Trees as Rose Trees

An abstract syntax tree can be represented as a rose tree of production labels:

type $AST = Rose\ ProdLabel.$

The type *ProdLabel* is a sum type representing the different productions of the grammar. In repmin, The abstract syntax tree is a binary tree, so *ProdLabel* is:

data $ProdLabel = Root \mid Fork \mid Leaf\ Int.$

A single *Root* production appears at the top of the tree. It has one child, which is the binary tree. *Fork* nodes have two children and *Leaf* nodes have zero. Note that the type *ProdLabel* can contain terminal information such as integer constants and strings, but it does not contain non-terminal information. The non-terminals of a production are represented as children in the rose tree.

8.4.2 Attributes

A drawback of using of rose trees is that every node in the tree must have the same set of attributes. That is, if leaf nodes have a *code* attribute then root nodes have a *code* attribute too. However, attributes can be left undefined and the definedness test ensures that undefined attributes are not used. Repmin has one inherited attribute for the global minimum leaf value:

type $Inh = Flat\ Int.$

The *Flat* datatype (Appendix A.5) is used so that the attribute can be left undefined if necessary. Repmin has two synthesised attributes: the minimum leaf value for the current subtree and the output tree for the current subtree. The *Flat* datatype is used here again:

$$\textbf{type } Syn \;=\; (Flat\ Int,\ Flat\ AST).$$

In an attribute grammar with more attributes, one might use records or lookup tables, rather than tuples to store the attributes. The definition of attribute grammars given below uses polymorphism to leave this choice open.

8.4.3 Attribute Grammars as Functions

In the following definition of an attribute grammar, the inherited and synthesised attributes are represented by the type parameters α and β, respectively. As explained above, this means that the implementation of attributes is left unspecified. The use of polymorphism also means that safety for free (§5.4) can be applied later.

$$\textbf{type } AG\ \alpha\ \beta \;=\; SemRule\ \alpha\ \beta \leftarrow ProdLabel.$$

$$\textbf{type } SemRule\ \alpha\ \beta \;=\; (\beta,\ Seq\ \alpha) \leftarrow (\alpha,\ Seq\ \beta).$$

This definition states that an attribute grammar is a set of semantic rules, indexed by the productions of the grammar. The concept of a semantic rule used here deviates slightly from the traditional one. Traditionally, there is one semantic rule per attribute. Here, a single semantic rule defines all the attribute values. It is a function from the input attributes to the output attributes, as illustrated in Figure 8.2. The input attributes are the inherited attributes of the parent node and the synthesised attributes of the children. The output attributes are the synthesised attributes of the parent and the inherited attributes of the children. The use of a single function to define all the attribute values does not mean that they have to be computed simultaneously. In Haskell, for example, attribute values would be evaluated on demand due to the lazy semantics. In Farrow's model [28], attribute values are computed iteratively, so the function would be called several times.

Repmin

Repmin is encoded as a value of type AG as follows:

$$\begin{array}{lll}
repAG & :: & AG\ Inh\ Syn, \\
repAG\ Root\ (_, syns) & = & \textbf{let}\ (gmin, ntree) = syns\ 1\ \textbf{in} \\
 & & ((Bot,\ Flat\ mkRoot\ ntree),\ [gmin]), \\
repAG\ Fork\ (gmin, syns) & = & \textbf{let}\ (lmin_1, ntree_1) = syns\ 1, \\
 & & \quad (lmin_2, ntree_2) = syns\ 2\ \textbf{in} \\
 & & ((Flat_2\ min\ lmin_1\ lmin_2, \\
 & & \ Flat_2\ mkFork\ ntree_1\ ntree_2), \\
 & & [gmin, gmin]), \\
repAG\ (Leaf\ k)\ (gmin, _) & = & ((Mid\ k,\ Flat\ mkLeaf\ gmin),\ []).
\end{array}$$

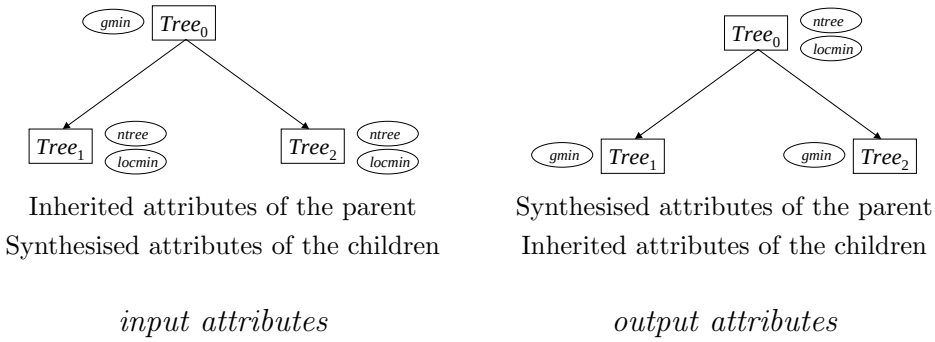


Figure 8.2: Input and Output Attribute Families

where the definitions of the auxiliary functions *mkRoot*, *mkFork* and *mkLeaf* are:

$$\begin{aligned} &mkRoot \;\; :: \; AST \leftarrow AST, \\ &mkRoot \; t \; = \; Node \; Root \; [t]. \end{aligned}$$

$$\begin{aligned} &mkFork \;\; :: \; AST \leftarrow AST \leftarrow AST, \\ &mkFork \; l \; r \; = \; Node \; Fork \; [l,r]. \end{aligned}$$

$$\begin{aligned} &mkLeaf \;\; :: \; AST \leftarrow Int, \\ &mkLeaf \; k \; = \; Node \; (Leaf \; k) \; []. \end{aligned}$$

This should be read as follows. The local minimum of the root production is undefined, because it will remain unused. The output tree of the root production is obtained by applying *mkRoot* to the output tree of its subtree. In the root production, the local minimum of the subtree becomes the inherited global minimum attribute. (The *gmin* attribute of the root node is ignored.) In the fork production, the local minimum is the minimum of the two local minima of the subtrees. The output tree is built from the output trees of the two subtrees and the global minimum is passed unchanged to the subtrees. The local minimum in the leaf production equals the value at the leaf and the new tree is a leaf node with the global minimum as its value.

Semantic Trees

An attribute grammar can be used to convert an AST into a semantic tree, which is a tree containing semantic functions. This is done by simply mapping the attribute grammar over the tree:

$$\textbf{type} \; SemTree \; \alpha \; \beta \; = \; Rose \; (SemRule \; \alpha \; \beta).$$

$$\begin{aligned} &mkSemTree \; :: \; \forall \alpha,\beta.\; SemTree \; \alpha \; \beta \leftarrow AST \leftarrow AG \; \alpha \; \beta, \\ &mkSemTree \; = \; mapRose. \end{aligned}$$

The main reason for introducing semantic trees is that they make many of the theorems and proofs in this chapter more concise.

8.4.4 Shifting Attributes Up and Down the Tree

The goal of an attribute grammar evaluator is to produce a tree containing the attribute values. In other words the goal is to produce a value of type $Rose\ (\alpha, \beta)$, where α and β are the types of the inherited and synthesised attributes. However, this format is not easily compatible with the semantic rules defined earlier. Instead the following two formats to are used to represent annotated trees:

$$\textbf{type}\ \mathit{InputTree}\ \alpha\ \beta\ =\ \mathit{Rose}\ (\alpha,\ \mathit{Seq}\ \beta).$$

$$\textbf{type}\ \mathit{OutputTree}\ \alpha\ \beta\ =\ \mathit{Rose}\ (\beta,\ \mathit{Seq}\ \alpha).$$

In the input format, the synthesised attributes of a node are stored by the node's parent. The parent uses a sequence to store the synthesised attributes of all its children. Similarly, in the output format, the inherited attributes of a node are stored by the node's parent. The output tree is the result of applying the semantic tree to the input tree. Note that in the input format, the synthesised attributes of the root node are not stored. Similarly, in the output format the inherited attributes are not stored. Therefore, this data needs to be stored separately.

The *shift* function is used to convert from the output to the input format. It encapsulates the idea that inherited attributes are passed down the tree and synthesised attributes are passed up. Note that *shift* merely moves attributes around; the calculation of the attribute values is done by the semantic tree.

$$\begin{aligned}&\mathit{shift}\ ::\ \forall\alpha,\beta.\ \mathit{InputTree}\ \alpha\ \beta \leftarrow \mathit{OutputTree}\ \alpha\ \beta \leftarrow \alpha\\ &\mathit{shift}\ a\ (\mathit{Node}\ (b, as)\ ts)\ =\\ &\quad \textbf{let}\ bs\ =\ \mathit{map}\ (\mathit{fst} \circ \mathit{root})\ ts\ \textbf{in}\ \mathit{Node}\ (a, bs)\ (\mathit{zipWith}\ \mathit{shift}\ as\ ts).\end{aligned}$$

Note that the inherited attributes of the root node are the first parameter of *shift*. The inherited attributes of the root node are always an external parameter to an attribute grammar.

8.4.5 The Semantics as a Least Fixed Point Computation

Recall that a semantic tree is obtained by mapping an AG over an AST. Given a semantic tree *semtree* and the inherited attributes a of the root node, the input and output trees should satisfy the following equations:

$$\mathit{inputTree} = \mathit{shift}\ a\ \mathit{outputTree} \tag{8.1}$$

$$\mathit{outputTree} = \mathit{appRose}\ \mathit{semtree}\ \mathit{inputTree} \tag{8.2}$$

Equation (8.1) states that the input tree can be computed from the output tree with the *shift* function. Equation (8.2) states that the output tree can be computed by applying the semantic tree to the input tree. Equations (8.1) and (8.2) form a pair of simultaneous equations that can be used to compute *inputTree* and *outputTree*. Following Chirica and Martin [20], the semantics of attribute grammars are defined as the least solution of these equations. This choice has no effect if the attributes are

not circularly defined, because the solution to the equations is unique. If a flat domain is used, then circularly defined attributes will evaluate to $\perp$. Farrow [28] discusses some interesting applications of working with non-flat domains. The least solution can be computed with the following function:

$$\begin{array}{l} \mathit{eval} \quad :: \ \forall\alpha,\beta.\ \mathit{OutputTree}\ \alpha\ \beta \leftarrow \alpha \leftarrow \mathit{SemTree}\ \alpha\ \beta, \\ \mathit{eval}\ \mathit{semtree}\ a\ =\ \mu\ (\mathit{appRose}\ \mathit{semtree} \circ \mathit{shift}\ a). \end{array}$$

This function is the definition of the semantics of attribute grammars. Please note, though, that it is not proposed as an efficient implementation! In the following section, a more convenient version called *io* is defined and demonstrated on the repmin problem in §8.5.2.

8.5 AG Semantics as a Structural Recursion

The *eval* function given above computes attribute values for the entire tree. In practice, the synthesised attributes at the root node of the tree are often the only values that are of interest. Therefore, it is convenient to define the function *io*:

$$\begin{array}{l} \mathit{io} \quad :: \ \forall\alpha,\beta.\ \beta \leftarrow \alpha \leftarrow \mathit{SemTree}\ \alpha\ \beta, \\ \mathit{io}\ \mathit{semtree}\ a\ =\ \mathit{fst}\ (\mathit{root}\ (\mathit{eval}\ \mathit{semtree}\ a)). \end{array}$$

The name *io* stands for input-output. It converts the semantic tree into a function from inherited attributes to synthesised attributes.

Johnsson [48] and Kuiper and Swierstra [58] have shown that AGs can be viewed as an idiom of lazy functional programming. Using the idea that a semantic tree is simply a function from inherited to synthesised attributes, they recursively build the input-output function for the entire AST. The idea that input-output functions can be computed by structural recursion was also stated much earlier by Chirica and Martin [20]. The idea can be stated formally with the following theorem:

Theorem 10 *The following definition of io is equivalent to the one given above:*

$$\mathit{io} = \mathit{foldRose}\ \mathit{knit}$$

where:

$$\begin{array}{lll} \mathit{knit} & :: \ \forall\alpha,\beta.\ (\beta \leftarrow \alpha) \leftarrow (\mathit{SemRule}\ \alpha\ \beta,\ \mathit{Seq}\ (\beta \leftarrow \alpha)), \\ \mathit{knit}\ (\mathit{semrule}, \mathit{fs})\ a & = & \textbf{let}\ (b, \mathit{as}) = \mathit{semrule}\ (a, \mathit{bs}), \\ & & \quad\ \ \mathit{bs} \quad\ = \mathit{appSeq}\ \mathit{fs}\ \mathit{as}\ \textbf{in}\ b. \end{array}$$

Note that in the definition of *knit*, the sequence *as* is recursively defined. Therefore, the definition of *knit* could equivalently be written as:

$$\mathit{knit}\ (\mathit{semrule}, \mathit{fs})\ a\ =\ \mathit{fst}\ \ (\mu\,(b, \mathit{as}).\ \mathit{semrule}\ \ (a,\ \mathit{appSeq}\ \mathit{fs}\ \mathit{as})).$$

The definition of *knit* is rather unintuitive, but it has appeared in different forms in many papers on attribute grammars. For example, see Gondow & Katayama [36] and Rosendahl [88]. Very briefly, *as* and *bs* are respectively the inherited and synthesised attributes of the subtrees. Their values are mutually dependent (using *semrule*), so they are computed by a least fixed point computation. For a better understanding of *knit*, see Johnsson [48].

The proof given here is essentially the same as the proof given by Chirica and Martin [20], but uses rather different notation. The most important step is to use the mutual recursion rule (§3.5). Lemma 12 shows how the mutual recursion rule can be applied to rose trees. A second important step is to use the abstraction rule.

Lemma 12 (The Mutual Recursion Rule on Rose Trees) *If f is a monotonic function of type Rose $A \to$ Rose A, then:*

$$root\ (\mu\, f)\ =\ (\mu\, x.\ g_1\ (x,\ (\mu\, xs.\ g_2\ (x, xs)))).$$

where:

$$\begin{aligned} g_1 &= fst \circ g, \\ g_2 &= snd \circ g, \\ g &= split \circ f \circ build. \end{aligned}$$

Proof:

$$\begin{aligned} & root\ (\mu\, f) \\ = & \qquad \{\ build \circ split = id\ \} \\ & root\ (\mu\ (build \circ split \circ f)) \\ = & \qquad \{\ \text{Rolling Rule (§3.5)}\ \} \\ & root\ (build\ (\mu\ (split \circ f \circ build))) \\ = & \qquad \{\ root \circ build = fst, \text{ Definition } g\ \} \\ & fst\ (\mu\ g) \\ = & \qquad \{\ \text{Mutual Recursion Rule (§3.5)}\ \} \\ & (\mu\, x.\ g_1\ (x,\ (\mu\, xs.\ g_2\ (x, xs)))). \end{aligned}$$

Using Lemma 12, Theorem 10 can now be proved and the definition of *knit* derived in the process. The proof strategy is to solve the following equation with *knit* as the unknown:

$$io = foldRose\ knit.$$

By the universal property of *foldRose* (see Bird & de Moor [15, page 46]), this equation is equivalent to:

$$(\forall s, ss, a.\ \ io\ (Node\ s\ ss)\ a\ =\ knit\ (s,\ map\ io\ ss)\ a).$$

The proof continues by manipulating the expression $io\ (Node\ s\ ss)\ a$ until an expression is obtained from which it is easy to derive the definition of *knit*:

$$
\begin{array}{ll}
 & \mathit{io}\ (\mathit{Node}\ s\ ss)\ a \\
= & \qquad \{\ \text{Definition } \mathit{io}\ \} \\
 & \mathit{fst}\ \ (\mathit{root}\ \ (\mu\ \ (\mathit{appRose}\ \ (\mathit{Node}\ s\ ss)\ \circ\ \mathit{shift}\ a))).
\end{array}
$$

The mutual recursion rule on rose trees (Lemma 12) can be applied to this expression. Expanding the definitions of g_1 and g_2, as specified in Lemma 12, establishes that:

$$
\begin{array}{l}
g_1\ ((b, as),\ ts)\ =\ s\ \ (a,\ \mathit{map}\ \ (\mathit{fst} \circ \mathit{root})\ \ ts), \\
g_2\ ((b, as),\ ts)\ =\ \mathit{zipWith}\ \ \mathit{appRose}\ \ ss\ \ (\mathit{zipWith}\ \ \mathit{shift}\ \ as\ \ ts).
\end{array}
$$

Therefore, Lemma 12 implies that $\mathit{io}\ (\mathit{Node}\ s\ ss)\ a$ is equal to:

$$
\mathit{fst}\ \ (\mu\,(b, as).\ g_1\ \ ((b, as),\ \ (\mu\,ts.\ g_2\ \ ((b, as),\ ts)))).
$$

(With g_1 and g_2 as defined above.) Consider the sub-expression containing g_2:

$$
\begin{array}{ll}
 & (\mu\,ts.\ g_2\ \ ((b, as),\ ts)) \\
= & \qquad \{\ \text{Definition } g_2\ \} \\
 & (\mu\,ts.\ \mathit{zipWith}\ \ \mathit{appRose}\ \ ss\ \ (\mathit{zipWith}\ \ \mathit{shift}\ \ as\ \ ts)) \\
= & \qquad \{\ \text{Lemma 11}\ \} \\
 & (\mu\,ts.\ \mathit{appSeq}\ \ (\mathit{zipWith}\ \ (\circ)\ \ (\mathit{map}\ \ \mathit{appRose}\ \ ss)\ \ (\mathit{map}\ \ \mathit{shift}\ \ as))\ \ ts) \\
= & \qquad \{\ \text{Abstraction Rule (§3.5)}\ \} \\
 & (\lambda\,k.\ \mu\ \ (\mathit{zipWith}\ \ (\circ)\ \ (\mathit{map}\ \ \mathit{appRose}\ \ ss)\ \ (\mathit{map}\ \ \mathit{shift}\ \ as)\ \ k)) \\
= & \qquad \{\ \text{Definitions } \mathit{zipWith} \text{ and } \mathit{map}\ \} \\
 & (\lambda\,k.\ \mu\ \ (\mathit{appRose}\ \ (ss\ \ k)\ \circ\ \mathit{shift}\ \ (as\ \ k))) \\
= & \qquad \{\ \text{Definition } \mathit{eval}\ \} \\
 & (\lambda\,k.\ \mathit{eval}\ \ (ss\ \ k)\ \ (as\ \ k)).
\end{array}
$$

This result can be used in the main expression:

$$
\begin{array}{ll}
 & \mathit{fst}\ \ (\mu\,(b, as).\ g_1\ \ ((b, as),\ \ (\mu\,ts.\ g_2\ \ ((b, as),\ ts)))) \\
= & \qquad \{\ \text{Substitute the new sub-expression}\ \} \\
 & \mathit{fst}\ \ (\mu\,(b, as).\ g_1\ \ ((b, as),\ \ (\lambda\,k.\ \mathit{eval}\ \ (ss\ \ k)\ \ (as\ \ k)))) \\
= & \qquad \{\ \text{Definition } g_1\ \} \\
 & \mathit{fst}\ \ (\mu\,(b, as).\ s\ \ (a,\ \ \mathit{map}\ \ (\mathit{fst} \circ \mathit{root})\ \ (\lambda\,k.\ \mathit{eval}\ \ (ss\ \ k)\ \ (as\ \ k)))) \\
= & \qquad \{\ \text{Definitions } \mathit{map} \text{ and } \mathit{io}\ \} \\
 & \mathit{fst}\ \ (\mu\,(b, as).\ s\ \ (a,\ \ (\lambda\,k.\ \mathit{io}\ \ (ss\ \ k)\ \ (as\ \ k)))) \\
= & \qquad \{\ \text{Definitions } \mathit{appSeq} \text{ and } \mathit{map}\ \} \\
 & \mathit{fst}\ \ (\mu\,(b, as).\ s\ \ (a,\ \ \mathit{appSeq}\ \ (\mathit{map}\ \ \mathit{io}\ \ ss)\ \ as)) \\
= & \qquad \{\ \text{Definition } \mathit{knit}\ \} \\
 & \mathit{knit}\ \ (s,\ \mathit{map}\ \ \mathit{io}\ \ ss)\ \ a.
\end{array}
$$

The final step of this calculation can be viewed as a derivation of the definition of *knit*.

8.5.1 A More Convenient Version of the *io* Function: *trans*

The *io* function operates on semantic trees. Semantic trees were useful for keeping the proof of Theorem 10 concise, but a function called *trans* (short for translate) is often more useful in practice:

$$\begin{aligned} &trans &&:: \forall\alpha,\beta.\ \beta \leftarrow \alpha \leftarrow AST \leftarrow AG\ \alpha\ \beta, \\ &trans\ ag &&= io \circ mkSemTree\ ag. \end{aligned}$$

In the next section, it is important to know that *trans* can be written as a fold:

$$trans\ ag = foldRose\ (knit \circ (ag \times id)).$$

The proof is a simple application of fold-map fusion. (See Bird [14, page 131]).

8.5.2 Repmin as a Recursive Function

It is illustrative to expand *trans repAG* and see that it is equivalent to the functional program for repmin. Expanding the definitions of *trans* (above), *knit* (above), *foldRose* (§8.3.1) and *repAG* (§8.4.3.1) leads to:

$$\begin{aligned} &repmin :: Syn \leftarrow Inh \leftarrow AST \\ &repmin\ (Node\ p\ ts)\ gmin = \\ &\quad \textbf{case}\ p\ \textbf{of} \\ &\qquad Root \rightarrow \textbf{let}\ (gmin, ntree) = repmin\ (ts\ 1)\ gmin\ \textbf{in} \\ &\qquad\qquad (Bot,\ Flat\ mkRoot\ ntree) \\ &\qquad Fork \rightarrow \textbf{let}\ (lmin_1, ntree_1) = repmin\ (ts\ 1)\ gmin, \\ &\qquad\qquad\qquad (lmin_2, ntree_2) = repmin\ (ts\ 2)\ gmin\ \textbf{in} \\ &\qquad\qquad (Flat_2\ min\ lmin_1\ lmin_2,\ Flat_2\ mkFork\ ntree_1\ ntree_2) \\ &\qquad Leaf\ k \rightarrow (Mid\ k,\ Flat\ mkLeaf\ gmin). \end{aligned}$$

This is indeed very similar to the program given by Bird [13]. Of course Bird's emphasis is on the recursive definition of *gmin* at the root of the tree, which is valid in a language with lazy evaluation.

8.6 The Definedness Test

When Knuth [55] introduced attribute grammars, he also gave algorithms for testing them for closure and circularity. Closure is the property that there are no missing semantic rules for attributes. An attribute grammar is circular if there exists an input tree on which the attributes are circularly defined. A new algorithm is presented here which is very similar to the circularity test, but encompasses the roles of both the closure and the circularity tests. It computes equivalent results to the circularity test and improves upon the closure test (which is based on a very shallow analysis of the dependency structure) by reducing the number of false negatives. This improvement has been found to be essential in other work with Van Wyk and de Moor on extending attribute grammars [101].

In the model given here, an undefined attribute is an attribute with the value $\bot$. Similarly, circularly defined attributes over a flat domain will evaluate to $\bot$, due to the least fixed point semantics.[2] So the closure and the circularity tests are both techniques for detecting attributes that might evaluate to $\bot$. The definedness test presented here performs the tasks of both tests by analysing strictness. The concept of strictness is very similar to the concept of dependence used in the circularity test. A function f is strict if $f\ \bot = \bot$. An undefined attribute is an attribute with the value $\bot$, so a strict function depends on its parameter. However, the expression $f\ x$ is said to depend on x even if f is not strict, so the two concepts are not identical. Luckily, semantic rules in a traditional attribute grammar are always strict in the attributes that they depend on.

The derivation of the definedness test consists of two parts. In the first part, safety for free (§5.4) is used to derive the abstract interpretation of the *trans* function. In the second part, the hylomorphism theorem (§8.3.1) is used to apply the strictness analysis to all possible input trees.

8.6.1 Abstract Interpretation of Attribute Grammars

Applying safety for free (§5.4) to the function *trans*, leads to the following theorem:

Theorem 11 *Suppose ag is a function of type* AG A B. *Suppose also that there are Galois Connections between* $(A, \preceq)$ *and* $(A', \sqsubseteq)$ *and between* $(B, \leq)$ *and* $(B', \trianglelefteq)$*:*

$$abs_A\ a \sqsubseteq a' \ \equiv\ a \preceq con_A\ a',$$
$$abs_B\ b \trianglelefteq b' \ \equiv\ b \leq con_B\ b'.$$

Then:

$$(\forall t,a.\ \ trans\ ag\ t\ a \ \leq\ con_B\ (trans\ ag'\ t\ (abs_A\ a))).$$

where the definition of ag' *is:*

$$ag'\ p \ =\ (abs_B \times (map\ abs_A)) \circ ag\ p \circ (con_A \times (map\ con_B)).$$

This theorem could be worded as follows: if ag' is the abstract interpretation of ag, then $trans\ ag'$ is the abstract interpretation of $trans\ ag$.

Repmin

In the repmin example, the types A and B are *Flat Int* and $(Flat\ Int, Flat\ AST)$, respectively. Therefore, appropriate abstraction and concretisation functions for the strictness analysis of repmin are:

[2] If the domain is non-flat, then circular attributes may not evaluate to $\bot$, but in this situation circularity is usually intended rather than erroneous. Farrow [28] suggests some useful applications of circular attributes over non-flat domains.

$$
\begin{aligned}
abs_A &= abs,\\
con_A &= con,\\
abs_B &= (abs \times abs),\\
con_B &= (con \times con).
\end{aligned}
$$

Where *abs* is the standard abstraction function used in strictness analysis:

$$
\begin{aligned}
abs &:: \forall\alpha.\ Bool \leftarrow Flat\ \alpha,\\
abs\ Bot &= False,\\
abs\ \_ &= True.
\end{aligned}
$$

and *con* is the corresponding concretisation function:

$$
\begin{aligned}
con &:: \forall\alpha.\ Flat\ \alpha \leftarrow Bool,\\
con\ False &= Bot,\\
con\ True &= Top.
\end{aligned}
$$

Given these abstraction and concretisation functions for repmin, Theorem 11 defines $repAG'$ in terms of $repAG$ (§8.4.3.1). The result is:

$$
\begin{aligned}
&repAG' && :: AG\ Bool\ (Bool, Bool),\\
&repAG'\ Root\ (\_, syns) && = \textbf{let}\ (gmin', ntree') = syns\ 1\ \textbf{in}\\
&&& \quad ((False, ntree'), [gmin']),\\
&repAG'\ Fork\ (gmin', syns) && = \textbf{let}\ (lmin'_1, ntree'_1) = syns\ 1,\\
&&& \quad\ \ (lmin'_2, ntree'_2) = syns\ 2\ \textbf{in}\\
&&& \quad ((lmin'_1 \wedge lmin'_2,\ ntree'_1 \wedge ntree'_2),\\
&&& \quad\ [gmin', gmin']),\\
&repAG'\ (Leaf\ k)\ (gmin', \_) && = ((True, gmin'), []).
\end{aligned}
$$

This function is used later to analyse the definedness of repmin.

8.6.2 A More Exact Analysis

A technique for applying the full parametricity theorem to functions that use fixpoints was discussed in §4.4.1. The idea was to pass the fixpoint operator as an external parameter to the function. In the case of the *trans* function, this actually leads to a more precise version of Theorem 11. The definition of the semantics of attribute grammars, given in §8.4, only uses one fixpoint: the fixpoint in the definition of *eval* (page 47). This fixpoint could be factorised out to create a function $trans'$ with type:[3]

$$trans' :: \forall\alpha,\beta.\ \beta \leftarrow \alpha \leftarrow AST \leftarrow AG\ \alpha\ \beta \leftarrow Fix\ (OutputTree\ \alpha\ \beta).$$

Applying the standard parametricity theorem to this function delivers the following result:

Theorem 12 *Suppose ag is a function of type* AG A B. *Suppose also that there are Galois Connections between* $(A, \preceq)$ *and* $(A', \sqsubseteq)$ *and between* $(B, \leq)$ *and* $(B', \trianglelefteq)$*:*

[3]The type *Fix* is a synonym: **type** $Fix\ \alpha = \alpha \leftarrow (\alpha \leftarrow \alpha)$.

$$abs_A\ a \sqsubseteq a' \equiv a \preceq con_A\ a',$$
$$abs_B\ b \trianglelefteq b' \equiv b \leq con_B\ b'.$$

Then:

$$(\forall t,a.\ abs_B\ (trans'\ \mu\ ag\ t\ a) = trans\ \mu\ ag'\ t\ (abs_A\ a)).$$

provided that ag and ag′ are related as follows:

$$(\forall p.\ (abs_B \times (map\ abs_A)) \circ ag\ p = ag'\ p \circ (abs_A \times (map\ abs_B))).$$

This theorem is more precise than Theorem 11, because the two sides of the equation are equal. To prove the theorem, one extra side-condition must be proved:

$$(\mu,\mu) \in Fix\ (OutputTree\ abs_A\ abs_B),$$

but this just expands to an instance of the fusion theorem (§3.5).

The functions *repAG* and *repAG′* satisfy the stronger requirements of this theorem, so *trans repAG′* produces exact analysis results for *trans repAG*.

8.6.3 Computing the Strictness for all Possible Trees

In general an attribute grammar will have n inherited attributes and m synthesised attributes. Therefore, the abstract types *Inh′* and *Syn′* are typically tuples of Booleans:

$$\begin{array}{lcl} \textbf{type}\ Inh' & = & (\ \overbrace{Bool, \ldots, Bool}^{n \text{ times}}\ ) \\ \textbf{type}\ Syn' & = & (\ \underbrace{Bool, \ldots, Bool}_{m \text{ times}}\ ) \end{array}$$

This means that there are 2^n different values of type *Inh′* and 2^m of type *Syn′*. Consequently, there are only a finite number of functions of type $Inh' \to Syn'$. So even though there are an infinite number of input trees, *trans ag′* can only have a finite number of values. In this section an algorithm is derived, very similar to the traditional circularity test algorithm, for computing those values in a finite amount of time.

The Grammar

Until this point, abstract syntax trees have not been constrained by a grammar: the rose tree representation of ASTs allows grammatically incorrect trees to be created. The goal of the definedness test is only to check the attribute grammar on grammatically correct trees, so the grammar needs to be defined.

A grammar consists of a set of symbols and an associated set of productions. Just like productions, the symbols can be represented as a simple enumeration datatype. In repmin, there are just two symbols:

data *Symbol* = *Start* | *Tree*.

Every symbol in the grammar is associated with zero or more productions. Every production has a sequence of symbols on its right hand side. This can be represented as a partial function with the following type:

$$grammar :: Symbol \nleftarrow (ProdLabel, Seq\ Symbol).$$

This function is partial, because it only accepts inputs where the sequence of symbols matches the production. For example, the grammar for repmin is:

$$\begin{aligned} grammar\ (Root, [Tree]) &= Start, \\ grammar\ (Fork, [Tree, Tree]) &= Tree, \\ grammar\ (Leaf\ k, []) &= Tree. \end{aligned}$$

The following function checks whether a tree is grammatically correct:

$$foldRose\ grammar :: Symbol \nleftarrow AST.$$

This function is also partial. It is only defined on grammatically correct trees.

Generating Grammatically Correct Trees

The set of all grammatically correct trees can be generated by inverting the grammar:

$$\begin{aligned} &trees :: AST \sim Symbol, \\ &trees = (foldRose\ grammar)^{\cup}. \end{aligned}$$

As indicated by the type signature, *trees* is a relation. Relations can be thought of as set-valued functions, so given a symbol, *trees* produces the set of all grammatically correct trees with that symbol-type.

Testing all Grammatically Correct Trees

The two necessary ingredients for the definedness test are now in place. An individual tree can be analysed and the set of all grammatically correct trees can be generated. These two elements can be put together as follows:

$$\begin{aligned} &definedness :: (Syn' \leftarrow Inh') \sim Symbol, \\ &definedness = trans\ ag' \circ trees. \end{aligned}$$

Given a symbol, *definedness* produces the test results for all trees of that type. These results form a set of finite size, despite the possibility of the number of intermediate trees being infinite. So the question is: how can the results be computed in a finite amount of time? The answer is provided by the hylomorphism theorem (§8.3.1). It is applicable, because *trans* can be written as a fold and *trees* as the inverse of a fold. Therefore, an equivalent definition of *definedness* is:

$$definedness = (\mu\, D.\ knit \circ (ag' \times map\ D) \circ grammar^{\cup}).$$

The result of this fixpoint computation can be computed by iteration (§3.4.1). In the context of computing *definedness*, D is a relation over a finite domain, so it can be represented as a set of pairs. The iteration starts with the empty set and continues until D ceases to change.

Testing repmin

To evaluate *definedness* for repmin, start with D equal to the empty relation. After one iteration, D' equals:

$$\begin{aligned} D' &= knit \circ (repAG' \times map\ D) \circ grammar^{\cup}, \\ &= ((\lambda\, a.\ (True, a)) \leftarrowtail Tree). \end{aligned}$$

After the second iteration:

$$D'' = ((\lambda\, a.\ (False, True)) \leftarrowtail Start,\ (\lambda\, a.\ (True, a)) \leftarrowtail Tree).$$

On the third iteration, it turns out that $D''' = D''$, so the iteration has stabilised and $definedness = D''$. What does this value of *definedness* say about repmin? On the root node, the *lmin* attribute will never be defined, regardless of whether the inherited *gmin* attribute is defined, but the *ntree* attribute is always defined. On internal nodes of the tree, *lmin* is always defined and *ntree* is defined if *gmin* is defined.

8.7 Summary

This chapter defined the semantics of attribute grammars using the notation of Chapter 2. This formulation was used to give a new proof of Chirica and Martin's important result [20] and to derive a definedness test which is very similar to the traditional circularity test. The derivation of the definedness test is a traditional example of abstract interpretation: first the semantics of a language is defined and then an abstract semantics is derived. The derivation was made much easier by the use of safety for free (§5.4).

An unusual feature of the derivation of the definedness test, is the use of the hylomorphism theorem (§8.3.1). Abstract interpretation produces an analysis that can be applied to one AST, but there are an infinite number of possible ASTs. The hylomorphism theorem is used to apply the analysis to all possible ASTs in a finite amount of time.

The decision to use rose trees comes with one major drawback: loss of type safety. If algebraic datatypes had been used to define the abstract syntax, then they would automatically be grammatically correct. There would have been no need to discuss *the Grammar* in §8.6.3.1. It is also common for different grammar symbols to have different attributes. For example, in *repmin* the *Start* symbol does not need a *locmin* or a *gmin* attribute. In the model given here, it is not possible to distinguish between the symbols in this way. It would be interesting to explore the use of category theory to extend the results to an arbitrary recursive datatype.

Chapter 9

Summary and Further Work

The ideas underlying *domain-specific embedded languages* date back to Landin's work in 1966 [60], but they have become increasingly viable in recent years due to advances in functional programming. The lazy functional language Haskell [78] has been particularly influential, because it has proved to be an excellent host for DSELs. A number of high-quality implementations of Haskell are available, which offer extensive support for input-output. This has allowed the development of DSELs such as Elliot and Hudak's animation library *Fran* [26], which dispel the myth that functional languages are unsuitable for writing interactive programs. However, if DSELs are to be a realistic vehicle for language development, then much better support for domain-specific error-checking and optimisation is needed. Without such support, DSELs are destined to remain toys, not tools.

Error-checking and optimisations are dependent upon program analysis, but two problems make DSELs difficult to analyse:

1. *Syntactic* analyses cannot be used due to the embedded nature of DSELs.

2. *Higher-order functions* are pervasive in DSEL design and use.

The solution proposed here uses *abstract interpretation* to tackle the first problem and *parametricity* to tackle the second. A bond between these two concepts is created by means of *abstract datatypes*. If abstract datatypes are employed in the design of a DSEL, then it can be paired with an *abstract* module that implements the same interface. Evaluating a program with the abstract module yields an analysis result. The correctness of this result is guaranteed, even in the presence of higher-order functions, by Reynolds's *abstraction theorem* [86] (or *theorems for free* as Wadler [103] calls it). Reynolds's theorem is applicable to DSELs due to Jones's observation [50] that abstract datatypes can be encoded as polymorphic type variables. In this encoding, modules are first-class values, so DSELs can be passed as parameters to programs. This allows abstract and concrete implementations of DSELs to be interchanged easily. Abstract interpretation was discussed in Chapter 3 and Reynolds's theorem in Chapter 4. The combination of these two concepts in Chapter 5 led to *safety for free*, which ensures the correctness of the analysis results.

Safety for free has been illustrated on three examples. The first example of the abstract interpretation of DSELs was the alphabet analysis of CSP, developed in Chapter 6. This is a straightforward example, but it is representative of a common problem in DSEL design: CSP's type system is incompatible with the type system of the host language, so the *phantom type* approach of Leijen and Meijer [63] (§1.3.2) is not applicable. The precise problem is that CSP requires a union operation on types, but ML-style type systems do not provide one. The solution is to analyse alphabets using abstract interpretation. A more advanced example was tackled in Chapter 7: an ambiguity test for parser combinators. Parser combinators are based on LL parsing, but the well-known LL(k) test described by Aho and Ullman [7, pages 334–368] cannot be used due to its reliance on contextual information. A new test was derived that does not suffer from this problem. The third example was a *definedness test* for attribute grammars, which was developed in Chapter 8. This is an example of abstract interpretation in its traditional form: a language is defined and then an abstract interpreter for the language is derived. Safety for free takes much of the toil out of this process.

This chapter discusses three areas for further research. Firstly, in §9.1, some new examples that safety for free might be applied to are suggested. Secondly, in §9.2, the problem that safety for free cannot currently be applied to *monadic* DSELs is discussed. Solving this problem will involve foundational work on extending the parametricity theorem to Girard's system F_ω [35]. Finally, in §9.3, some suggestions are made for implementing the ideas in a real programming language.

9.1 Future Applications

The application of safety for free to DSELs has been chosen as the primary focus of this dissertation for two reasons: firstly, because it illustrates the power of safety for free, and secondly, because no other solutions to the problem of analysing DSELs exist, apart from the phantom types of Leijen and Meijer [63]. There is no reason, though, why safety for free should only be applied to the abstract interpretation of DSELs. In fact, there is also no reason why it should only be applied to abstract interpretation. The result is valid for any problem involving Galois connections. Given how important Galois connections are in computer science (see, for example, Hoare and Jifeng [41]), safety for free must be of potential relevance to a much wider range of problems.

In the field of DSELs, the most frequent application of safety for free is likely to be the development of domain-specific type checkers, such as the alphabet analysis for CSP, presented in Chapter 6. For instance, an area of current interest is the document language XML [17], which uses types based on regular expressions. New programming languages with specially designed type systems, such as CDuce [12], are being designed to enable XML documents to be manipulated in a type-safe way. If XML's type system could, instead, be encoded as an abstract interpretation, then there would be no need for new programming languages, just new DSELs.

Safety for free can also be applied to more sophisticated analyses, such as the ambiguity analysis for parser combinators, presented in Chapter 7. The challenge

there was to find a compositional alternative to the traditional LL(k) test, which uses a global analysis. Many other known analyses are likely to pose similar problems, so it would be useful to develop a catalogue of mathematical techniques that can be used to solve them. This catalogue will grow naturally as new analyses are cast into the abstract interpretation framework. Suitable examples to tackle might be a test for deadlock in CSP processes or an analysis of the freshness of meta-variables, as is done in FreshML [89].

9.2 Monadic DSELs

One of the most important innovations in the recent history of functional programming is the use of *monads*. Monads were introduced to computer science by Moggi [69, 70, 71], who used them to structure denotational semantics. Wadler [104, 105, 106] then showed that they are of great practical use in functional programming, particularly for encoding impure features such as state and I/O. Spivey [91] independently developed a monadic encoding of exceptions.

Many DSELs have monadic interfaces, to allow a greater degree of interaction with the program: a monadic interface enables the program to pass its own data through the library. There is still a type-safe boundary between the program and the DSEL, but it is much less restrictive than was previously possible. As an illustration, most parser combinator libraries have monadic interfaces so that parsers can be annotated with computations.

Unfortunately, there is a theoretical gap that needs to be bridged before the ideas of this dissertation can be applied to monadic DSELS. Jones's module system [50] can be used to encode monadic DSELs but *higher kinded polymorphic type variables* are required. Such types are well understood, but no work seems to have been done on their parametric properties. Without a parametricity theorem for higher kinded polymorphic type variables there is no theoretical basis for the abstract interpretation of monadic DSELs. Jones's encoding of monadic modules is explained below.

9.2.1 Higher-Kinded Types

As an example of a module signature that requires higher kinded types, consider the signature of a stack module, with the type of stacks being an abstract datatype. Suppose that the stacks should also be polymorphic. That is, it should be possible to build a stack of integers and a stack of strings with the same module. The signature of such a stack module is as follows:

$$
\begin{array}{lll}
\textbf{type}\ \mathit{Stack}\ s\ = \ \textbf{sig} & \mathit{empty} & :: \forall\alpha.\ s\ \alpha, \\
& \mathit{push} & :: \forall\alpha.\ s\ \alpha \leftarrow s\ \alpha \leftarrow \alpha, \\
& \mathit{pop} & :: \forall\alpha.\ (\alpha,\ s\ \alpha) \leftarrow s\ \alpha, \\
& \mathit{map} & :: \forall\alpha,\beta.\ s\ \beta \leftarrow s\ \alpha \leftarrow (\beta \leftarrow \alpha) \\
\textbf{end}. &&
\end{array}
$$

The type s is the type of stacks and α is the type contained in the stack. The type variable s differs from normal type variables such as α, because it is *higher kinded*. A suitable instantiation for s would a unary relator, such as the list datatype.

Jones uses the type system of Girard's F_ω calculus [35][1], in which type variables are assigned *kinds*. In the stack example, α has kind $*$, because it is a normal type variable, but s has kind $* \leftarrow *$, because it takes types to types. Such polymorphic type variables with higher kinds such as $* \leftarrow *$ are essential for encoding many interesting abstract datatypes.

9.2.2 A Partial Solution

In the stack example above, the *map* operation was included to illustrate the full power of higher-kinded types. It is possible to define a stack signature that does not use higher-kinded types, but then the *map* operation cannot be included. The restricted version of the stack signature is:

$$\begin{array}{lll} \textbf{type } \mathit{Stack}'\ s\ \alpha\ = \textbf{sig} & \mathit{empty} & :: s\ \alpha, \\ & \mathit{push} & :: s\ \alpha \leftarrow s\ \alpha \leftarrow s\ \alpha, \\ & \mathit{pop} & :: (\alpha,\ s\ \alpha) \leftarrow s\ \alpha \\ \textbf{end}. & & \end{array}$$

Higher-kinded type variables are also essential for *monadic* programming. The signature of a monadic library is:

$$\begin{array}{lll} \textbf{type } \mathit{Monad}\ m\ = \textbf{sig} & \mathit{unit} & :: \forall\alpha.\ m\ \alpha \leftarrow \alpha, \\ & \mathit{bind} & :: \forall\alpha,\beta.\ m\ \beta \leftarrow (m\ \beta \leftarrow \alpha) \leftarrow m\ \alpha \\ \textbf{end}. & & \end{array}$$

The nested quantifiers in this signature are essential. The power of the monadic interface would be lost if the type variables α and β were lifted out as was done in the stack example.

9.3 Implementation

To implement the ideas of this dissertation, a functional language is needed that is capable of computing least fixed points over non-flat domains. Techniques for computing such fixpoints are well-known in the field of program analysis, but little work seems to have been done on incorporating them into programming languages. Cai and Paige [19] have developed a language called SQ^+ that can compute least fixed points over sets, but it is not a higher-order language.

The textbook on program analysis by Nielsen, *et al.* [76, pages 363–385] includes a chapter on algorithms for computing least fixed points. Certain restrictions apply when using these algorithms. For example, *iteration* can only be used if the *ascending*

[1] An excellent introduction to system F_ω can be found in the textbook by Pierce [82, pages 439–466].

chain condition is satisfied; otherwise the algorithm might not terminate. So programs that use fixpoints are not always implementable. Compilers and interpreters need to be able to distinguish the implementable programs and derive appropriate implementations.

It has already been observed in §3.6 that the fixpoint operator poses similar problems to the polymorphic equality operator. This observation seems to extend to implementation: like the fixpoint operator, polymorphic equality is not always implementable. For example, comparing two functions is undecidable in general. This problem has been solved through the use of *equality type variables* in Standard ML and *type classes* in Haskell. A type inference process is used to ensure that polymorphic equality is implementable. For example, in Haskell rules such as the following are employed:

```
instance (Eq a, Eq b) => Eq (a,b) where
   (x1,y1) == (x2,y2) = x1 == x2 && y1 == y2
```

This rule states that if values of type `a` can be tested for equality and values of type `b` also, then pairs of type `(a,b)` can be tested for equality. The implementation of equality on pairs is also defined by this declaration. It is defined in terms of the equality operators for `a` and `b`. The same approach could be used in the implementation of least fixed points: inference rules determine when and how the fixpoint operator is implementable.

Appendix A

Commonly Used Functions and Datatypes

This appendix defines the standard functions that are used throughout the dissertation. Most of these functions are well-known by functional programmers and their definitions are given by Bird [14] and the Haskell Report [78].

A.1 Tuples

The functions *fst* and *snd* (Haskell Report [78, page 103], Bird [14, page 41]) are used to extract the first and second element of a pair:

$$\begin{array}{l} \mathit{fst} \quad :: \forall\alpha,\beta.\ \alpha \leftarrow (\alpha,\beta), \\ \mathit{fst}\ (x,y) = x. \end{array}$$

$$\begin{array}{l} \mathit{snd} \quad :: \forall\alpha,\beta.\ \beta \leftarrow (\alpha,\beta), \\ \mathit{snd}\ (x,y) = y. \end{array}$$

The product relator takes a pair of relations to a relation on pairs:

$$(\times) :: \forall\alpha,\beta,\gamma,\delta.\ ((\alpha,\gamma) \sim (\beta,\delta)) \leftarrow ((\alpha \sim \beta),\ (\gamma \sim \delta)).$$

Its definition (also given in §4.2) is:

$$((a,c),(b,d)) \in (R \times S) \equiv (a,b) \in R \wedge (c,d) \in S.$$

When applied to functions, this definition is equivalent to the standard product functor:

$$\begin{array}{l} (\times) \quad :: \forall\alpha,\beta,\gamma,\delta.\ (\alpha,\gamma) \leftarrow (\beta,\delta) \leftarrow ((\alpha \leftarrow \beta),\ (\gamma \leftarrow \delta)), \\ (f \times g)\ (x,y) = (f\ x,\ g\ y). \end{array}$$

A similar operator is $\triangle$ (pronounced *split*), with type:

$$(\triangle) :: \forall\alpha,\beta,\gamma.\ ((\alpha,\beta) \sim \gamma) \leftarrow ((\alpha \sim \gamma),\ (\beta \sim \gamma)).$$

Its definition is:

$$((a,b),c) \in (R \vartriangle S) \equiv (a,c) \in R \wedge (b,c) \in S.$$

Like the product relator, $\vartriangle$ takes functions to functions:

$$\begin{array}{l}(\vartriangle) :: \forall\alpha,\beta,\gamma.\ (\alpha,\beta) \leftarrow \gamma \leftarrow ((\alpha \leftarrow \gamma),\ (\beta \leftarrow \gamma)),\\ (f \vartriangle g)\ x = (f\ x,\ g\ x).\end{array}$$

A.2 Lists

The function *map* applies a function to every element of a list. It is a special case of the relator *List* (§4.2):

$$\begin{array}{l}map :: \forall\alpha,\beta.\ [\beta] \leftarrow [\alpha] \leftarrow (\beta \leftarrow \alpha),\\ map\ f\ [] = [],\\ map\ f\ (x : xs) = f\ x : map\ f\ xs.\end{array}$$

The function *filter* removes elements that do not satisfy a predicate:

$$\begin{array}{l}filter :: \forall\alpha.\ [\alpha] \leftarrow [\alpha] \leftarrow (Bool \leftarrow \alpha),\\ filter\ p\ [] = [],\\ filter\ p\ (x : xs) = \textbf{let}\ ys = filter\ p\ xs\ \textbf{in if}\ p\ x\ \textbf{then}\ x : ys\ \textbf{else}\ ys.\end{array}$$

The function *foldr* is a general encoding of structural recursion over lists. It is known in category theory as a *catamorphism*:

$$\begin{array}{l}foldr :: \forall\alpha,\beta.\ \beta \leftarrow [\alpha] \leftarrow \beta \leftarrow (\beta \leftarrow (\alpha,\beta)),\\ foldr\ f\ z\ [] = z,\\ foldr\ f\ z\ (x : xs) = f\ (x,\ foldr\ f\ z\ xs).\end{array}$$

The function *append* is used to concatenate two lists:

$$\begin{array}{l}append :: \forall\alpha.\ [\alpha] \leftarrow ([\alpha],[\alpha]),\\ append\ ([], ys) = ys,\\ append\ (x : xs,\ ys) = x : append\ (xs, ys).\end{array}$$

Often the infix notation $+\!\!+$ is used for *append*.

The function *concat* concatenates a list of lists:

$$\begin{array}{l}concat :: \forall\alpha.\ [\alpha] \leftarrow [[\alpha]],\\ concat = foldr\ append\ [].\end{array}$$

The function *zip* converts a pair of lists into a list of pairs:

$$\begin{array}{lll}zip & & :: \forall\alpha,\beta.\ [(\alpha,\beta)] \leftarrow [\beta] \leftarrow [\alpha],\\ zip\ xs & [] & = [],\\ zip\ [] & ys & = [],\\ zip\ (x : xs) & (y : ys) & = (x,y) : zip\ xs\ ys.\end{array}$$

The length of the resulting list is equal to the length of the shorter of the two lists.

A generalisation of *zip* is *zipWith*, which uses a higher order parameter to combine the elements:

$$
\begin{array}{lll}
\mathit{zipWith} & :: & \forall\alpha,\beta,\gamma.\ [\gamma] \leftarrow [\beta] \leftarrow [\alpha] \\
 & & \qquad \leftarrow (\gamma \leftarrow (\alpha,\beta)), \\
\mathit{zipWith}\ (\oplus)\ \mathit{xs}\ \ [] & = & [], \\
\mathit{zipWith}\ (\oplus)\ []\ \ \mathit{ys} & = & [], \\
\mathit{zipWith}\ (\oplus)\ (x:\mathit{xs})\ (y:\mathit{ys}) & = & (x \oplus y) : \mathit{zipWith}\ (\oplus)\ \mathit{xs}\ \mathit{ys}.
\end{array}
$$

The function *take* retains the first k elements of a list and discards the rest:

$$
\begin{array}{lll}
\mathit{take} & :: & \forall\alpha.\ [\alpha] \leftarrow [\alpha] \leftarrow \mathit{Nat}, \\
\mathit{take}\ 0\ \mathit{xs} & = & [], \\
\mathit{take}\ n\ [] & = & [], \\
\mathit{take}\ n\ (x:\mathit{xs}) & = & x : \mathit{take}\ (n{-}1)\ \mathit{xs}.
\end{array}
$$

The function *elems* converts a list to a set:

$$
\begin{array}{lll}
\mathit{elems} & :: & \forall\alpha.\ \mathit{Set}\ \alpha \leftarrow [\alpha], \\
\mathit{elems}\ [] & = & \emptyset, \\
\mathit{elems}\ (x:\mathit{xs}) & = & \{x\} \cup \mathit{elems}\ \mathit{xs}.
\end{array}
$$

A.3 Combinators

In this section, some simple combinators are described that can be very useful in function definitions and in algebraic manipulations. Composition is used regularly:

$$
\begin{array}{lll}
(\circ) & :: & \forall\alpha,\beta,\gamma.\ (\gamma \leftarrow \alpha) \leftarrow ((\gamma \leftarrow \beta),\ (\beta \leftarrow \alpha)), \\
(f \circ g)\ x & = & f\ (g\ x).
\end{array}
$$

The more general definition of composition when applied to relations is:

$$
\begin{array}{lll}
(\circ) & :: & \forall\alpha,\beta,\gamma.\ (\gamma \sim \alpha) \leftarrow ((\gamma \sim \beta),\ (\beta \sim \alpha)), \\
(x,z) \in R \circ S & \equiv & (\exists y.\ (x,y) \in R \wedge (y,z) \in S).
\end{array}
$$

So is the identity function:

$$
\begin{array}{lll}
\mathit{id} & :: & \forall\alpha.\ \alpha \leftarrow \alpha, \\
\mathit{id}\ x & = & x.
\end{array}
$$

Occasionally, it is also useful to encode function application as a combinator:

$$
\begin{array}{lll}
\mathit{app} & :: & \forall\alpha,\beta.\ \beta \leftarrow \alpha \leftarrow (\beta \leftarrow \alpha), \\
\mathit{app}\ f\ x & = & f\ x.
\end{array}
$$

The S and K combinators are important in the lambda calculus, because any lambda term can be written using S and K. Turner [98, 99] uses this idea to implement lazy functional languages. An introduction to Turner's techniques is given by Peyton Jones [79, pages 260-280]. The K combinator is known as *const* in Haskell:

$$
\begin{array}{l}
K \quad :: \forall\alpha,\beta.\ \alpha \leftarrow \beta \leftarrow \alpha, \\
K\ x\ y\ =\ x.
\end{array}
$$

The S combinator shares one argument between two functions. Its importance in this dissertation is mainly due to its appearance in the abstraction rule for fixpoints (§3.5).

$$
\begin{array}{l}
S \quad :: \forall\alpha,\beta,\gamma.\ (\gamma \leftarrow \alpha) \leftarrow (\beta \leftarrow \alpha) \leftarrow (\gamma \leftarrow \beta \leftarrow \alpha), \\
S\ f\ g\ x\ =\ f\ x\ (g\ x).
\end{array}
$$

Another combinator in the S-K family is S':

$$
\begin{array}{l}
S' \quad :: \forall\alpha,\beta,\gamma,\delta.\ (\delta \leftarrow \alpha) \leftarrow (\gamma \leftarrow \alpha) \leftarrow \\
\qquad\qquad\qquad\qquad (\beta \leftarrow \alpha) \leftarrow (\delta \leftarrow \gamma \leftarrow \beta), \\
S'\ c\ f\ g\ x\ =\ c\ (f\ x)\ (g\ x).
\end{array}
$$

A.4 Type Synonyms

Eq is convenient shorthand for the type of an equality operator:

$$\textbf{type}\ \mathit{Eq}\ \alpha\ =\ \mathit{Bool} \leftarrow \alpha \leftarrow \alpha.$$

Similarly, *Fix* is a useful shorthand for the type of a fixpoint operator:

$$\textbf{type}\ \mathit{Fix}\ \alpha\ =\ \alpha \leftarrow (\alpha \leftarrow \alpha).$$

A.5 The *Flat* Datatype

The *Flat* datatype is a useful wrapper for converting a datatype into a flat domain. It does this by adding top and bottom elements and leaving the other elements unordered. Its definition is:

$$\textbf{data}\ \mathit{Flat}\ \alpha\ =\ \mathit{Bot}\ \mid\ \mathit{Mid}\ \alpha\ \mid\ \mathit{Top}.$$

The ordering relation is:

$$x \leq y\ \equiv\ x = \mathit{Bot}\ \vee\ y = \mathit{Top}\ \vee\ x = y.$$

The *Flat* functor is useful for lifting functions to operate on values of type *Flat*:

$$
\begin{array}{lcl}
\mathit{Flat} & :: & \forall\alpha,\beta.\ \mathit{Flat}\ \beta \leftarrow \mathit{Flat}\ \alpha \leftarrow (\beta \leftarrow \alpha), \\
\mathit{Flat}\ f\ \mathit{Bot} & = & \mathit{Bot}, \\
\mathit{Flat}\ f\ (\mathit{Mid}\ x) & = & \mathit{Mid}\ (f\ x), \\
\mathit{Flat}\ f\ \mathit{Top} & = & \mathit{Top}.
\end{array}
$$

Similarly, it is often useful to be able to lift binary functions:

$$
\begin{array}{lll}
\mathit{Flat}_2 & :: & \forall\alpha,\beta,\gamma.\ \mathit{Flat}\ \gamma \leftarrow \mathit{Flat}\ \beta \leftarrow \mathit{Flat}\ \alpha \\
 & & \qquad \leftarrow (\gamma \leftarrow \beta \leftarrow \alpha), \\
\mathit{Flat}_2\ f\ (\mathit{Mid}\ x)\ (\mathit{Mid}\ y) & = & \mathit{Mid}\ (f\ x\ y), \\
\mathit{Flat}_2\ f\ \mathit{Top}\ y & = & \mathit{Top}, \\
\mathit{Flat}_2\ f\ x\ \mathit{Top} & = & \mathit{Top}, \\
\mathit{Flat}_2\ f\ \mathit{Bot}\ y & = & \mathit{Bot}, \\
\mathit{Flat}_2\ f\ x\ \mathit{Bot} & = & \mathit{Bot}.
\end{array}
$$

Bibliography

[1] C. J. Aarts. Galois connections presented calculationally. Afstudeer verslag (Graduating Dissertation), Department of Computing Science, Eindhoven University of Technology. `http://www.cs.nott.ac.uk/~rcb/MPC/galois.ps.gz`, 1992.

[2] S. Abramsky. Strictness analysis and polymorphic invariance. In H. Ganzinger and N. D. Jones, editors, *Workshop on Programs as Data Objects, Copenhagen, 1985*, volume 217 of *Lecture Notes in Computer Science*, pages 1–23. Springer-Verlag, 1986.

[3] S. Abramsky. Abstract interpretation, logical relations and Kan extensions. *Journal of Logic and Computation*, 1(1):5–41, July 1990.

[4] S. Abramsky and C. Hankin. An introduction to abstract interpretation. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, chapter 1, pages 9–31. Ellis Horwood, 1987.

[5] S. Abramsky and T. P. Jensen. A relational approach to strictness for higher-order polymorphic functions. In *18th POPL, Orlando, Florida*, pages 49–54. ACM Press, January 1991. `http://www.irisa.fr/lande/jensen/papers/popl91.ps`.

[6] S. Adams. *Modular Grammars for Programming Language Prototyping*. PhD thesis, University of Southampton, March 1991. `http://www.swiss.ai.mit.edu/~adams/MAG/`.

[7] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling; Volume 1: Parsing*. Series in Automatic Computation. Prentice-Hall, Englewood Cliff, N.J., 1972.

[8] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language*. Addison-Wesley, Reading, MA, third edition, 2000.

[9] K. S. Backhouse. A functional semantics of attribute grammars. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002*, volume 2280 of *Lecture Notes in Computer*

Science, pages 142–157. Springer-Verlag, 2002. http://web.comlab.ox.ac.uk/oucl/work/kevin.backhouse/papers/TACAS02/abstract.html.

[10] K. S. Backhouse and R. C. Backhouse. Logical relations and Galois connections. In *Mathematics of Program Construction (MPC 2002)*, volume 2386 of *Lecture Notes in Computer Science*, pages 23–39. Springer-Verlag, 2002. http://web.comlab.ox.ac.uk/oucl/work/kevin.backhouse/papers/MPC2002/abstract.html.

[11] R. C. Backhouse, P. J. de Bruin, P. Hoogendijk, G. Malcolm, T. S. Voermans, and J. van der Woude. Polynomial relators. In M. Nivat, C. S. Rattray, T. Rus, and G. Scollo, editors, *Proceedings of the 2nd Conference on Algebraic Methodology and Software Technology, AMAST'91*, Workshops in Computing, pages 303–326, London, May 1992. Springer-Verlag.

[12] V. Benzaken, G. Castagna, and A. Frisch. CDuce: an XML-centric general-purpose language. In *8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003)*, Uppsala, Sweden, August 2003. http://www.cduce.org/papers/cduce-design.ps.gz.

[13] R. S. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21:239–250, 1984.

[14] R. S. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, 2 edition, 1998.

[15] R. S. Bird and O. de Moor. *Algebra of Programming*, volume 100 of *International Series in Computer Science*. Prentice Hall, September 1996. http://www.comlab.ox.ac.uk/oucl/publications/books/algebra/.

[16] G. Bracha. Add generic types to the Java^{TM} programming language. Technical Report JSR 14, Java Specification Requests, August 2001. http://www.jcp.org/jsr/detail/14.jsp.

[17] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible markup language (XML) 1.0 (second edition). Technical report, W3C, October 2000. http://www.w3.org/TR/REC-xml.

[18] G. L. Burn, C. Hankin, and S. Abramsky. Strictness analysis for higher-order functions. *Science of Computer Programming*, 7:249–278, 1986.

[19] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11(3):197–261, April 1989.

[20] L. M. Chirica and D. F. Martin. An order-algebraic definition of Knuthian semantics. *Mathematical Systems Theory*, 13(1):1–27, 1979.

[21] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY. `http://www.di.ens.fr/~cousot/COUSOTpapers.shtml#1977`.

[22] P. Cousot and R. Cousot. Higher-order abstract interpretation (and application to comportment analysis generalizing strictness, termination, projection and PER analysis of functional languages). In *Proceedings of the 1994 International Conference on Computer Languages, ICCL'94*, pages 95–112, Toulouse, France, 16–19 May 1994. IEEE Computer Society Press. `http://www.di.ens.fr/~cousot/COUSOTpapers.shtml#1994`.

[23] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices*, 35(6):26–36, June 2000. `http://www.cwi.nl/~arie/papers/dslbib.pdf`.

[24] G. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *Computer Journal*, 33(2):164–172, April 1990.

[25] C. Elliott. Modeling interactive 3D and multimedia animation with an embedded language. In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*, pages 285–296, Berkeley, October 15–17 1997. USENIX Association. `http://www.research.microsoft.com/~conal/papers/default.htm`.

[26] C. Elliott and P. Hudak. Functional reactive animation. In *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming*, pages 263–273, Amsterdam, The Netherlands, 9–11 June 1997. `http://www.research.microsoft.com/~conal/papers/default.htm`.

[27] D. R. Engler. Incorporating application semantics and control into compilation. In *Proceedings of the Conference on Domain-Specific Languages (DSL-97)*, pages 103–118, Berkeley, October 15–17 1997. USENIX Association.

[28] R. Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *SIGPlan '86 Symposium on Compiler Construction*, pages 85–98, Palo Alto, CA, June 1986. Association for Computing Machinery, SIGPlan.

[29] S. I. Feldman. Make – A program for maintaining computer programs. Technical Report Computing Science Technical Report No. 57, Bell Laboratories, August 1978.

[30] Formal Systems (Europe) Ltd. Failures-Divergences Refinement: FDR2. `http://www.fsel.com/documentation/fdr2/fdr2manual.ps.gz`, May 2000.

[31] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Benjamin/Cummings Pub. Co., Redwood City, CA, USA, 1995. This book is a literate program (see Knuth [57]), describing the lcc C compiler. See also: `http://www.cs.princeton.edu/software/lcc/`.

[32] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, Reading, MA, 1995.

[33] B. R. Gaster and M. P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, Department of Computer Science, University of Nottingham, University Park, Nottingham NG7 2RD, UK, November 1996. `http://www.cse.ogi.edu/~mpj/pubs/polyrec.html`.

[34] The GHC Team. The Glasgow Haskell Compiler user's guide, version 5.04. `http://www.haskell.org/ghc/docs/latest/users_guide.ps`.

[35] J.-Y. Girard. *Interprétation Fonctionnelle et Élimination des Coupures dans l'Arithmétique d'Ordre Supérieur*. PhD thesis, Université Paris VII, 1972.

[36] K. Gondow and T. Katayama. Attribute grammars as record calculus — a structure-oriented denotational semantics of attribute grammars by using cardelli's record calculus. *Informatica*, 24(3):287–299, 2000. `http://www-sop.inria.fr/oasis/WAGA00/proceedings/gondow/gondow.ps`.

[37] W. L. Harrison and S. N. Kamin. Metacomputation-based compiler architecture. In R. C. Backhouse and José Nuno Oliveira, editors, *Mathematics of Program Construction: 5th International Conference, MPC 2000, Ponte de Lima, Portugal*, volume 1837 of *Lecture Notes in Computer Science*. Springer-Verlag, July 2000.

[38] J. Hartmanis and R. E. Stearns. Pair algebras and their application to automata theory. *Information and Control*, 7(4):485–507, 1964.

[39] J. Hartmanis and R. E. Stearns. *Algebraic Structure Theory of Sequential Machines*. Prentice Hall, Englewood Cliffs, 1966.

[40] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, NJ, 1985.

[41] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall Europe, Campus 400, Maylands Avenue, Hemel Hempstead, Hertfordshire, HP2 7EZ, UK, 1998.

[42] P. F. Hoogendijk. *A Generic Theory of Datatypes*. PhD thesis, Deptartment of Mathematics and Computing Science, Eindhoven University of Technology, 1997. Available from: `http://www.cs.nott.ac.uk/~rcb/MPC/HoogendijkThesis.ps.gz`.

[43] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4es):196–196, December 1996. http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/a196-hudak/.

[44] P. Hudak. Modular domain specific languages and tools. In P. Devanbu and J. Poulin, editors, *Proceedings: Fifth International Conference on Software Reuse*, pages 134–142. IEEE Computer Society Press, 1998. http://www.cs.yale.edu/HTML/YALE/CS/HyPlans/hudak-paul/hudak-dir/icsr98.ps.

[45] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, May 2000. See also: http://www.haskell.org/arrows/.

[46] G. Hutton and E. Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998. http://www.cs.nott.ac.uk/~gmh/bib.html#pearl.

[47] S. C. Johnson. Yacc: Yet another compiler compiler. In *UNIX Programmer's Manual*, volume 2, pages 353–387. Holt, Rinehart, and Winston, New York, NY, USA, 1979. Available from: http://citeseer.nj.nec.com/johnson79yacc.html.

[48] T. Johnsson. Attribute grammars as a functional programming paradigm. In Gilles Kahn, editor, *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, volume 274 of *LNCS*, pages 154–173, Portland, Oregon, September 1987. Springer-Verlag. http://citeseer.nj.nec.com/johnsson87attribute.html.

[49] M. P. Jones. Functional programming with overloading and higher-order polymorphism. In *First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden*, volume 925 of *Lecture Notes in Computer Science*, pages 97–136. Springer-Verlag, May 1995. http://www.cse.ogi.edu/~mpj/pubs/springschool.html.

[50] M. P. Jones. Using parameterized signatures to express modular structure. In *Conference Record of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*, pages 68–78, St. Petersburg Beach, Florida, 1996. ACM Press. http://www.cse.ogi.edu/~mpj/pubs/paramsig.html.

[51] U. Kastens and W. M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31(7):601–627, 1994.

[52] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs (NJ), USA, 1988.

[53] G. Kiczales. Aspect-oriented programming. *ACM Computing Surveys*, 28A(4), 1996. See also: http://eclipse.org/aspectj/.

[54] B. Knaster. Un théorèm sur les fonctions d'ensembles. *Annals Soc. Pol. Math*, 6:133–134, 1928.

[55] D. E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2:127–146, 1968.

[56] D. E. Knuth. Semantics of context-free languages: Correction. *Mathematical Systems Theory*, 5:95–96, 1971.

[57] D. E. Knuth. *Literate Programming*. Number 27 in CSLI Lecture Notes. Stanford University Center for the Study of Language and Information, Stanford, CA, USA, 1992.

[58] M. Kuiper and S. D. Swierstra. Using attribute grammars to derive efficient functional programs. In *Computing Science in the Netherlands CSN '87*, 1987. `ftp://ftp.cs.ruu.nl/pub/RUU/CS/techreps/CS-1986/1986-16.ps.gz`.

[59] Matthijs Kuiper and João Saraiva. Lrc - A Generator for Incremental Language-Oriented Tools. In Kay Koskimies, editor, *7th International Conference on Compiler Construction, CC/ETAPS'98*, volume 1383 of *LNCS*, pages 298–301. Springer-Verlag, April 1998. `http://www.di.uminho.pt/~jas/Research/Papers/CC98/cc98.ps.gz`.

[60] P. J. Landin. The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, March 1966.

[61] J. Launchbury and R. Paterson. Parametricity and unboxing with unpointed types. In H. R. Nielson, editor, *Programming Languages and Systems - ESOP'96, 6th European Symposium on Programming, Linköping, Sweden, April 22-24, 1996*, volume 1058 of *Lecture Notes in Computer Science*, pages 204–218. Springer, 1996. `http://www.cse.ogi.edu/~jl/Papers/pointed.ps`.

[62] J. Launchbury and S. L. Peyton Jones. Lazy functional state threads. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–35, Orlando, June 1994. `http://research.microsoft.com/Users/simonpj/Papers/papers.html#monads`.

[63] D. Leijen and E. Meijer. Domain-specific embedded compilers. In *Proceedings of the 2nd Conference on Domain-Specific Languages*, pages 109–122, Berkeley, CA, October 3–5 1999. USENIX Association. `http://www.cs.uu.nl/~daan/papers/dsec.ps`.

[64] D. Leijen and E. Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Department of Computer Science, Universiteit Utrecht, October 2001. `http://www.cs.uu.nl/~daan/parsec.html`.

[65] X. Leroy (with D. Doligez, J. Garrigue, D. Rémy and J. Vouillon). The Objective Caml system release 3.04. Technical report, Institut National de Recherche en Informatique et en Automatique, December 2001. http://caml.inria.fr/.

[66] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Conference record of POPL '95, 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, January 22–25, 1995*, pages 333–343, New York, NY, USA, 1995. ACM Press. http://www.cse.ogi.edu/~mpj/pubs/modinterp.html.

[67] Eindhoven University of Technology Mathematics of Program Construction Group. Fixed-point calculus. *Information Processing Letters Special Issue on The Calculational Method*, 53:131–136, 1995.

[68] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[69] E. Moggi. An abstract view of programming languages. Technical Report ECS-LFCS-90-113, Edinburgh University, 1989. http://www.disi.unige.it/person/MoggiE/publications.html.

[70] E. Moggi. Computational lambda-calculus and monads. In *Proceedings 4th Annual IEEE Symposium on Logic in Computer Science, LICS'89, Pacific Grove, CA, USA, 5–8 June 1989*, pages 14–23. IEEE Computer Society Press, Washington, DC, 1989. http://www.disi.unige.it/person/MoggiE/publications.html.

[71] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991. http://www.disi.unige.it/person/MoggiE/publications.html.

[72] O. de Moor, K. Backhouse, and S. D. Swierstra. First-class attribute grammars. *Informatica*, 24(3):329–341, 2000. http://web.comlab.ox.ac.uk/oucl/work/kevin.backhouse/papers/waga00/abstract.html.

[73] O. de Moor, S. Peyton Jones, and E. Van Wyk. Aspect-oriented compilers. In *First International Symposium on Generative and Component-based Software Engineering*, Lecture Notes in Computer Science. Springer-Verlag, 1999. http://web.comlab.ox.ac.uk/oucl/work/oege.demoor/papers/aspects.ps.gz.

[74] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs*. PhD thesis, Department of Computer Science, University of Edinburgh, 1981.

[75] A. Mycroft and N. D. Jones. A relational framework for abstract interpretation. In H. Ganzinger and N. D. Jones, editors, *Workshop on Programs as Data Objects, Copenhagen, 1985*, volume 217 of *Lecture Notes in Computer Science*,

pages 112–135. Springer-Verlag, 1986. http://www.cl.cam.ac.uk/users/am/papers/cph85.dvi.

[76] F. Nielson, H. R. Nielson, and C. L. Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999.

[77] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, Cambridge, UK, second edition, 1996.

[78] S. Peyton Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Report on the programming language Haskell 98. http://www.haskell.org/, 1999.

[79] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Series in Computer Science. Prentice Hall, 1987. See also: Peyton Jones and Lester [81].

[80] S. L. Peyton Jones, J.-M. Eber, and J. Seward. Composing contracts: an adventure in financial engineering, functional pearl. In *Proceedings of the ACM Sigplan International Conference on Functional Programming (ICFP-00)*, volume 35.9 of *ACM Sigplan Notices*, pages 280–292, N.Y., September 18–21 2000. ACM Press. http://research.microsoft.com/Users/simonpj/Papers/contracts-icfp.htm.

[81] S. L. Peyton Jones and D. R. Lester. *Implementing functional languages: a tutorial*. Series in Computer Science. Prentice Hall, 1992. http://research.microsoft.com/Users/simonpj/Papers/pj-lester-book/.

[82] B. C. Pierce. *Types and Programming Languages*. The MIT Press, February 2002. http://www.cis.upenn.edu/~bcpierce/tapl/index.html.

[83] Gordon D. Plotkin. Lambda-definability in the full type hierarchy. In Jonathan P. Seldin and J. Roger Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 363–373. Academic Press, London, 1980.

[84] D. Raggett, A. Le Hors, and I. Jacobs. Html 4.01 specification. Technical report, W3C, December 1999. http://www.w3.org/TR/1999/REC-html401-19991224/.

[85] T. W. Reps and T. Teitelbaum. *The Synthesizer Generator: A system for constructing language-based editors*. Texts and Monographs in Computer Science. Springer-Verlag, 1989.

[86] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Proceedings 9th IFIP World Computer Congress, Information Processing '83, Paris, France, 19–23 Sept 1983*, pages 513–523. Elsevier Science

Publishers B.V., Amsterdam, North Holland, 1983. http://www-2.cs.cmu.edu/afs/cs/user/jcr/ftp/typesabpara.pdf.

[87] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science, 1998. See also: http://web.comlab.ox.ac.uk/oucl/publications/books/concurrency/.

[88] M. Rosendahl. Strictness analysis for attribute grammars. In *PLILP'92*, volume 631 of *LNCS*, pages 145–157. Springer-Verlag, 1992. http://www.dat.ruc.dk/~madsr/webpub/plilp92-ag.pdf.

[89] M. Shinwell, A. Pitts, and M. Gabbay. FreshML: Programming with binders made simple. In *8th ACM SIGPLAN International Conference on Functional Programming (ICFP 2003)*, Uppsala, Sweden, August 2003. http://www.cl.cam.ac.uk/users/amp12/papers/index.html#frepbm.

[90] C. Simonyi. Intentional programming: Innovation in the legacy age. Presented at IFIP Working group 2.1, 1996.

[91] J. M. Spivey. A functional theory of exceptions. *Science of Computer Programming*, 14(1):25–43, July 1990.

[92] G. L. Steele, Jr. Building interpreters by composing monads. In *Conference record of POPL '94, 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: Portland, Oregon, January 17–21, 1994*, pages 472–492, New York, NY, USA, 1994. ACM Press. http://citeseer.nj.nec.com/steele94building.html.

[93] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[94] S. D. Swierstra. Combinator parsers: From toys to tools. In G. Hutton, editor, *Electronic Notes in Theoretical Computer Science*, volume 41.1. Elsevier Science Publishers, 2001. http://www.cs.uu.nl/~doaitse/Papers/2000/HaskellWorkshop.pdf.

[95] S. D. Swierstra and P. R. Azero Alcocer. Fast, error correcting parser combinators: A short tutorial. In J. Pavelka, G. Tel, and M. Bartosek, editors, *SOFSEM'99 Theory and Practice of Informatics, 26th Seminar on Current Trends in Theory and Practice of Informatics*, volume 1725 of *LNCS*, pages 111–129, November 1999. http://www.cs.uu.nl/~doaitse/Papers/1999/SofSem99.pdf.

[96] S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced functional programming: second international school, Olympia, WA, USA, August 26–30, 1996: tutorial text*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207, New York, NY, USA, 1996. Springer-Verlag. http://www.cs.uu.nl/~doaitse/Papers/1996/LL1.pdf.

[97] A. Tarski. A lattice-theoretic fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[98] D. A. Turner. Another algorithm for bracket abstraction. *Journal of Symbolic Logic*, 44(2):267–270, June 1979.

[99] D. A. Turner. A new implementation technique for applicative languages. *Software – Practice and Experience*, 9:31–49, 1979.

[100] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J. Jouannaud, editor, *Proceedings IFIP International Conference on Functional Programming Languages and Computer Architectures, Nancy, France*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag, New York, NY, September 1985.

[101] E. Van Wyk, O. de Moor, K. Backhouse, and P. Kwiatkowski. Forwarding in attribute grammars for modular language design. In R. N. Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002*, volume 2304 of *Lecture Notes in Computer Science*, pages 128–142. Springer-Verlag, 2002. `http://web.comlab.ox.ac.uk/oucl/work/kevin.backhouse/papers/CC2002/abstract.html`.

[102] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing (OO'98)*. SIAM Press, 1998. `http://www.osl.iu.edu/~tveldhui/papers/`.

[103] P. Wadler. Theorems for free! In *FPCA'89, London, England*, pages 347–359. ACM Press, September 1989. `http://www.research.avayalabs.com/user/wadler/topics/parametricity.html#free`.

[104] P. Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992. `http://www.research.avayalabs.com/user/wadler/topics/monads.html#monads`.

[105] P. Wadler. The essence of functional programming (invited talk). In *19'th ACM Symposium on Principles of Programming Languages*, Albuquerque, New Mexico, January 1992. `http://www.research.avayalabs.com/user/wadler/topics/monads.html#essence`.

[106] P. Wadler. How to declare an imperative. *ACM Computing Surveys*, 29(3):240–263, September 1997. `http://www.research.avayalabs.com/user/wadler/topics/monads.html#monadsdeclare`.

[107] M. Williams Zimmerman, editor. *Microsoft Visual Basic 6.0 Programmer's Guide*. Microsoft Press, 1998.