A Semantics for Aspects by Compositional Translation

Sam B. Sanjabi

Exeter College

Doctor of Philosophy

Oxford University Computing Laboratory

October 2008

To Bec

Contents

Acknowledgements

Producing a Doctoral Thesis has been, and may prove to remain, the most daunting task I have undertaken. To thank everyone who I feel contributed to it seems for the moment nearly as challenging: I feel as justified mentionning colleagues and friends as I am the barkeep that served me the ale I was drinking in a quiet epiphanic moment. In a selfish attempt to ease the task of writing this statement, I will let brevity and generality guide my hand.

My own experience was perhaps typically colored by periods of advancement, and others of stagnation. Throughout the former, my supervisor Dr. Luke Ong stands out as the voice of calm, providing a guiding hand or skillfully standing back as the situation required. His judgement, as well as the feedback and inspiration from the members of the FoCS and Tools groups at the Computing Laboratry, were invaluable. When research was slow and my confidence waned, the support of my family and friends, now scattered accross the world and thankfully too numerous to mention, cannot be quantified. I hope that you know who you are, and I assure you that if you so much as sent me an e-mail in the last few years you are in my thoughts now.

I would also like to specially mention Dr. Samson Abramsky who served as my co-supervisor in my first year, and Dr. Oege de Moor and Dr. Chris Hankin who served as my examiners and who's respective expertise in the practical and theoretical facets of the pages which follow yielded a breadth of feedback that I am taking very much to heart.

Abstract

We analyse the semantics of aspect-oriented extensions to functional languages by presenting compositional translations of these primitives into languages with traditional notions of state and control. As a first step, we examine an existing semantic description of aspects which allows the labelling of program points. We show that a restriction of these semantics to aspects which do not preempt the execution of code can be fully abstractly translated into a functional calculus with higher order references, but that removing this restriction requires a notion of exception handling to be added to the target language in order to yield a sound semantics. Next, we proceed to show that abandoning the labelling technique, and consequently relaxing the so-called "obliviousness" property of aspectual languages, allows preemptive aspects to be included in the general references model without the need for exceptions. This means that the game model of general references is inherited by the aspect calculus.

The net result is a clean semantic description of aspect-orientation, which mirrors recently published techniques for their implementation, and thereby provides theoretical justification for these systems. The practical validity of our semantics is demonstrated by implementing extensions to the basic calculus in Standard ML, and showing how a number of useful aspect-oriented features can be expressed using general references alone. Our theoretical methodology closely follows the proof structure that often appears in the game semantics literature, and therefore provides an operational perspective on notions such as "bad variables" and factorisation theorems.

Part I

Introduction and Background

Chapter 1

Introduction

The increase in widely available computational power has led to a corresponding increase in program complexity. No longer the task of small groups, software development is a collaborative effort requiring the integration of independently developed components. This often leads to multiple programmers modifying the same piece of code, possibly simultaneously. Moreover, modern applications must endure a seemingly perpetual sequence of updates, bug fixes, feature enhancements, and so on. In short, a modern piece of software is a dynamic entity. It is practically impossible to accommodate such a situation with a low-level programming language: the need for readable, reusable code is crucial in order to successfully create functioning complex systems.

Procedural programs (like those written in C and Pascal) improved on assembly languages by providing high-level syntactic constructs for looping, branching, and – more importantly – easing code reuse. The ability to name a piece of code, and to execute it with a procedure call enabled the creation of *libraries*: commonly used functions that did not need to be re-written by later programmers.

Procedural libraries suffer from two major deficiencies. First, library users can directly change any data associated with the functions. This is a problem when the li-

brary's underlying implementation was changed, but its interface was not. Second, libraries were not easily extensible: if a user wanted to modify a library, then the actual library code would need to be touched. Consider the case when a 2D-point library is available, but a 3D-point library is needed.

These issues were resolved by the next major innovation in reusable code: Object-Oriented programming languages. Languages such as C++ and Java provided facilities that encapsulated both the functions and the data of a particular library into an entity called an *object*. Access to private data could be restricted, and each user that requires the use of the resources would instantiate his own copy of the object, and then access it through interface functions. Thus, underlying implementations could be changed without affecting user code. The extensibility issue was resolved by *inheritance*, which allowed fields and functions to be added to each object, and previously defined functions to be over-written without changing the original code.

All of these innovations can be viewed as attempts to adapt programming languages to support the following software development principle:

Changes to correct, working code should be minimised.

Aspect-oriented programming (AOP) can be viewed as an attempt to further evolve programming languages according to this principle.

1.1 Aspect Oriented Programming

The genesis of AOP accommodates practical software development practices, i.e. it is a response to the observed behaviour of "real world" software development projects. Specifically, once design and planning stages are completed, implementation tends to to follow two separate stages:

1. The functioning "core" of the software is written. This is the code that actually performs the operation that the system is intended to accomplish.
2. The "bells and whistles" of logging, error handling, etc. are added.

Notice how this process violates the principle alluded to above: the core functionality of the software is coded and tested, then changed in order to add all of the supporting features. This process readily leads to new bugs creeping into functioning code. Even a maximally modular object oriented program requires additional procedure calls in order to (for instance) add a log entry at the appropriate time.

Aspects attempt to rectify this situation by allowing programmers to specify that a piece of code needs to be executed whenever a particular predicate is satisfied during execution. The predicate could be anything from "procedure X is called", to "any procedure is called from within procedure Y" and various others. Note the contrast between this and non-aspect oriented languages in which any code execution must be explicitly invoked. Filman and Friedman [18] propose a qualitative characterisation of an ideal Aspect-Oriented Programming Language (AOPL) as one that allows a programmer to write *quantified* code which affects programs written by *oblivious* programmers.

Quantification refers to the ability to write a single piece of code that executes at many loci. An example of quantified program behaviour is a class constructor that is called upon the instantiation of any subclass. The obliviousness property states that such code should be definable and executable after the code which it affects (or "base" code) without the base code programmer being aware of it.

Such functionality allows programmers to work at the policy level directly, i.e. AOP allows them to concisely express general requirements that they wish programs to respect. For example, the following are readily expressible in modern aspect languages:

- A policy in an object-oriented language which asks that calls to any method in an object and its subclasses be logged.

- A security policy in an AOP version of C which asks that any calls to `strcpy()` be replaced with calls to `strncpy()`

The programmer merely writes the appropriate code and associates it with a predicate (in some specification language) that captures the required program points at which it needs to execute. This not only reduces code changes, but actually modularises code more than ever before possible. For example, *all* logging code would be located in one place in this framework unlike in the OO model where the definitions of logging functions would appear in the class, but the calls to those functions would be spread over multiple classes (see Fig 1.1).

1.1.1 Terminology

A *concern* (or *cross-cutting concern*) refers to the facet of a software system that is appropriate for using an aspect to address. The term is meant to indicate that aspects are appropriate for accomplishing similar tasks that are required across the breadth of the whole program regardless of the actual underlying functionality of the code (hence the term "cross-cutting"). In addition to the aforementioned examples of logging and error handling, common concerns also include debugging code, profiling, and timing.

Practical AOPLs have distinguished eligible points in the code that can be interrupted by an aspect invocation. These are known as *join points*, and can in principle be anywhere, but often involve procedure calls in some way. A predicate describing a set of join points is known as a *pointcut descriptor* or simply a *pointcut*, and is the mechanism for expressing which join points should be interrupted. The actual code that is executed when a pointcut is matched is called *advice*. An *aspect* is the pairing of a piece of advice together with a pointcut describing when the advice should execute.

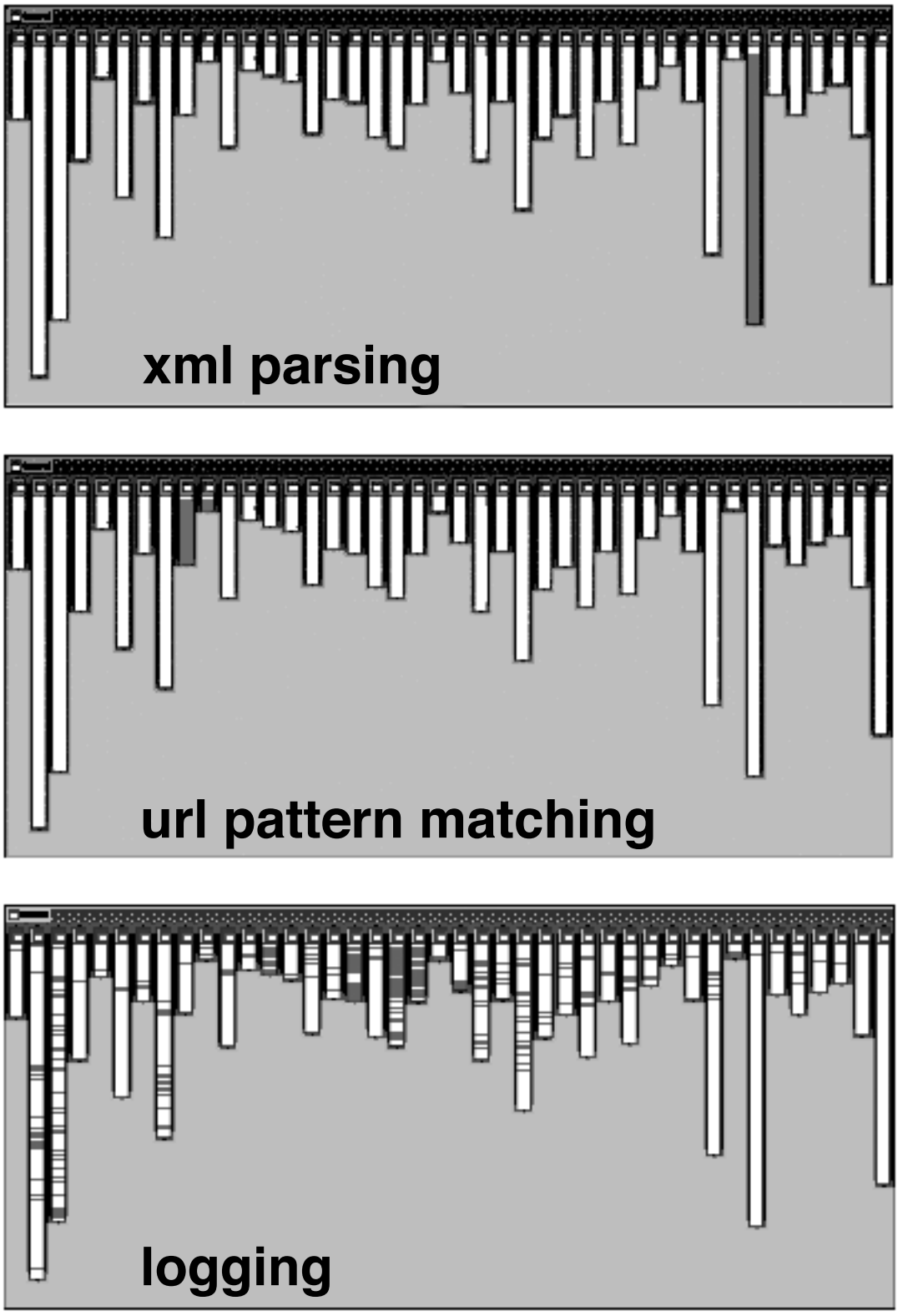


Figure 1.1: *Contrasting the localisation of parsing, pattern matching, and logging code in the Tomcat web server. Each vertical bar represents a class file, and the highlights show bytecode that deals with each task.*

1.1.2 A Brief Introduction to AspectJ

It is useful to be familiar with a concrete example of an AOPL in order to grasp the manner in which the concepts introduced above are realised in practice, and also to have a reference point against which any theoretical research can be compared. This section describes AspectJ – an aspect-oriented extension to Java, and the most popular AOP implementation – at a high level. It describes some of the language's join points, how to specify a subset of them using pointcut descriptors, and how to bundle these with advice in order to form aspects. The interested reader can refer to [32] for a more complete overview.

Join Points and Pointcuts

The valid join points in an AspectJ program are designated points in the *execution* of the underlying Java program. A pointcut descriptor (PCD) can therefore be dynamic, i.e. predicated on the runtime state of the program. Join points in AspectJ include method calls and executions, field reads and writes, and class and object initialisation.

Primitive pointcuts matching the join points are specified by unquantified logical formulae consisting of atoms matching individual join points. For instance the `call(S)` atom matches any calls of methods whose signature matches `S`, while `set(S)` matches writes to the field matching `S`. Other atoms of this language include `execution(S)` which matches the execution of a method, `get(S)` for field reads, and `cflow(P)` which matches any join points occurring within the dynamic context of another pointcut descriptor `P`. So for example, the PCD `cflow(call(A.foo()))` matches any join points occurring between a call to method `foo()` in class `A` and its termination. This is an example of a dynamic PCD, and can for example be used to match the top call in a recursive stack.

Advice

A piece of AspectJ advice is just a piece of Java code. Associating this piece of code with a pointcut causes the code to be executed every time a join point matching the PCD is reached during execution. The exact time at which this predicate should be true is slightly vague, however. Take the `set(S)` pointcut described above, should the advice execute immediate before the field is set, or immediately after? AspectJ provides syntax allowing the aspect programmer to specify this. For instance, the advice

```
before() : P {...}
```

results in the execution of the body immediately before any join point matching `P`. Note that the advice is subject to the same join point model as the base code, i.e. executing advice could trigger further advice.

In addition to `before()` and `after()` advice, AspectJ also supports something called `around()` advice, which provides the capability to selectively preempt the normal computation at the given join point. The base code can be executed at any time in the body of the advising code by calling the `proceed()` command, which returns the result of the underlying computation. Therefore, the advice can be executed in any way "around" the base code. Furthermore, `around()` advice can potentially prevent the base computation from executing at all by not calling `proceed()`. While `before()` and `after()` advice strictly add to the base code, `around()` advice may remove some of it.

Aspects

Aspects are declared as an encapsulation of point-cuts, advice, and any other usual declarations (such as local fields and methods) which may be declared in a class. The following declaration keeps track of when a particular method executes:

```
aspect TrackExecution
{
  static boolean flag = false;

  static boolean testAndClear()
    {
      boolean result = flag;
      flag = false;
      return result;
    }

  after() : execution(void A.foo())
    {
      flag = true;
    }
}
```

The aspect defines a local bit `flag`, and a method `testAndClear()` which retrieves its current contents and resets it. The pertinent part is the advice declaration, which states that the execution of `A.foo()` sets the flag. The method could (as in [32]) be one which a graphics package calls to move an element in a picture. In this case, the flag can simply be seen as a boolean indicating whether the screen needs to be refreshed.

Note that it is possible for a single join point to trigger multiple pieces of advice. This implies that there must be some way to give some precedence to advice, and indeed AspectJ has some internal precedence rules, but the resolution of many potential conflicts are left to the implementation. Chapter 5 discusses precedence in a functional setting in more detail.

1.2 Survey of Formal Semantics of Aspects

There have been three major research threads studying the formal semantics of AOP since its inception. The first sought to study the pointcut and advice mechanism in its purest form, and culminates in a minimal aspect calculus called μABC [10]. The second

attempts to emulate the structure of AspectJ by extending Abadì and Cardelli's object calculus with aspects [14]. The third studies aspect oriented extensions to functional languages, focussing on developing feature rich AOPLs with strong theoretical foundations. A seminal paper in this area was published in 2003 by Walker, Zdancewic, and Ligatti [59] and defines the semantics of an small ML-like AOPL by translation into a typed λ-calculus.

Notation In the following, the notation $\overline{\mathrm{X}_i}$ is used to denote the syntactic, comma separated sequence $\mathrm{X}_1, \ldots, \mathrm{X}_n$ of X's indexed by the set $\mathrm{I} = \{1, \ldots, n\}$. If another delimiter is required, this is indicated by appending it, so for example $\overline{\mathrm{X}_i;}$ denotes the same sequence separated by semicolons.

1.2.1 A minimal aspect calculus

μABC was defined in the last of a series of papers by Glenn Bruns, Radha Jagadeesan, Alan Jeffrey, and James Riely [28, 29, 10] which present a number of core calculi for aspect-oriented programming. Of these, μABC [10] is the most minimal in terms of features, while the typed language of [28] and the untyped language of [29] are constructed around a multi-threaded class-based language.

The language is built around a countable partial order of *roles*, with top element $\top$, which are dynamically declared. A program consists of a sequence of declarations followed by a returned role. Declarations include role declarations, named advice declarations (whose names are disjoint from roles), and what the authors call *message sends*, which take the form `let` $x = p \to q : \overline{m_i}$ where $p, q, \overline{m_i}$ is a sequence of roles. The notation is meant to call to mind the sending of message $\overline{m_i}$ by p to q. However, this is an unfortunate choice of words, as a "message send" doesn't send a message at all, but simply acts as a placeholder against which advice can be matched (i.e. a join point).

The declaration `advice` $a[\phi] = \sigma x.\tau y.\pi b.Q$ associates a piece of advice Q, which is a μABC term, with an advice name a and a pointcut descriptor ϕ. The binders σ, τ, and π are mnemonics for source, target, and proceed and come into play in advice substitution. PCDs are quantified logical formulae built from atoms $p \to q : l$. For example, if ϕ is the negated atom $\neg(p \to q : l)$, then the advice is triggered by any message sends that are not of the form $p \to q : \overline{m_i}, l$, i.e. any message whose source is not p, whose target is not q, or whose last message is not l. When an aspect fires, it replaces the message which triggered it with Q, substituting the message's source name for the σ-bound variable in Q, the target name for the τ-bound variable, and the sequence of message names (excluding the last one) for the π-bound variable. More complicated pointcuts allow multiple messages to trigger a piece of advice. For example, the formula $\exists x \preceq \top.\exists y \preceq p.[x \to y : l]$ matches any message that sends l to p or one of its subroles.

Essentially, μABC is nothing but join points (which have no semantic function other than the role names they contain), and advice declarations (which provide a sort of substitution semantics). It is a language of pure aspects in the same sense that Featherweight Java [27] is a language of pure objects. It is also expressive, as it can encode the untyped λ-calculus, which is both surprising and predictable depending on one's point of view: surprising because of μABC's simplicity, but expected because of the substitution behaviour of advice.

1.2.2 Aspects and objects

On the opposite end of the minimality spectrum lies Clifton, Leavens, and Wand's parametrised aspect calculus $\varsigma_{\mathsf{A}}(\mathrm{M})$ [14]. Their language formalises aspects on top of Abadì and Cardelli's functional object calculus [1], which models an object as a set $\{\overline{l_i = \varsigma(x_i)b_i}\}$ of labelled methods, where l_i is the label, x_i represents the "self" parameter of the object, and b_i is the body of the method. A method l_i is selected from an object

o using the notation $o.l_i$, which evaluates to $b_i[o/x_i]$, i.e. the body of l_i with the object itself replacing every instance of the self parameter in the body. The term $o.l_i \Leftarrow \varsigma(x)b$ replaces the method labelled l_i in o with $\varsigma(x)b$. The values of the language include objects as well as constants d taken from a set `Consts`. A "field" in the object calculus is just a method that ignores its self parameter and returns a basic constant. As an example, suppose that $\texttt{Consts} = \mathbb{N}$, then the object $\{\texttt{n} = \varsigma(\ )1\ ,\ \texttt{val} = \varsigma(\texttt{p})\texttt{p.n}\}$ stores a natural number in the field `n`, and provides a method `val` which queries its current value. Note that this treatment of objects does not deal with issues like sub-typing or inheritance, or even private fields.

The aspectual features which extend the object calculus to $\varsigma_{\mathsf{A}}(\mathrm{M})$ include join points and advice, but abstract the specific pointcut description language, which is passed as a parameter $\mathrm{M} = \langle \mathscr{G}, \texttt{match} \rangle$. Loosely speaking, $\mathscr{G}$ is the grammar defining the syntax of PCDs, and `match` is a function defining their semantics, i.e. taking an aspect and a join point and returning a sequence of operations (essentially unlabelled methods) to be executed in place of the base code. The join points include method selections, method updates, and base values which are identified by their signatures: a constant's signature is itself, and an object's signature is its set of labels. A program pairs a term (the base code) with a sequence of advice $\overline{\texttt{pcd}_i \triangleright \varsigma(y_i)b_i}$, which each in turn pair a pointcut descriptor from $\mathscr{G}$ with a method defining the advice that needs to execute when the pointcut matches. The operational semantics are defined in big-step style, and keep track of the evaluation context in which a term reduces. This means the context is available to join points, and can be used in M to form pointcuts. It is therefore possible to conditionally execute advice based on the run-time state of the program, much like the ability that AspectJ's `cflow()` pointcut provides. The primary features of $\varsigma_{\mathsf{A}}(\mathrm{M})$ as a research language are summarised below:

- The fact that pointcut descriptors are abstracted from the language makes it quite

a flexible framework in which to study various kinds of AOP.

- It uses a static list of aspects and a distinguished base term on top of an object calculus, making it much like AspectJ.
- It is untyped and therefore suffers from similar flaws as μABC with respect to denotational semantics study.
- It's definition uses a rather complex scheme for advice substitution, and particularly for proceeding onto further advice, which is implemented by first forming closures (essentially continuations) out of `proceed` commands and later executing them.

1.2.3 Aspects in functional languages

Considering the popularity of languages like ML and Haskell in academic circles, and their connection to typed λ-calculi, not to mention the success of game semantics in characterising them, it's no surprise that many aspect researchers turned to them as a basis on which to build structured AOP calculi. This section discusses the features of these languages at a high level.

MinAML and the Labelled Aspect Calculus

In their ICFP paper [59], Walker et al. define a simple aspect-oriented programming language called MinAML; it extends a small functional language with AspectJ-like constructs such as `before`, `after`, and `around` advice, using function calls as the only join points. The semantics of MinAML is defined by translation to a core aspect calculus (referred to here as λ_L), which extends the simply typed λ-calculus with a distinguished type $\texttt{lab}[\tau]$, whose values come from a countable set of *labels*. New labels are created

(in block structured scopes) by a `new` command. λ_{L} also has a type $\mathtt{asp}[\tau]$ of *aspects*, whose values take the form $\{\alpha.x \to M\}$, where α is a label, and M (the *advice*) is a term of the same type as the variable x. Terms are evaluated in an environment comprising a sequence of aspects A, which can be extended with the $M \gg N$ expression, which appends the aspect M to the end of the sequence and then evaluates N.

Join points are formed by explicitly labelling subterms using the syntax $\alpha\langle\!\langle M \rangle\!\rangle$. After M has been evaluated to a value V, the term $\alpha\langle\!\langle M \rangle\!\rangle$ causes any advice associated with the label α to be triggered. If there is an aspect of the form $\{\alpha.x \to N\}$ in A, then $\alpha\langle\!\langle M \rangle\!\rangle$ reduces to $N[V/x]$. These simple features endow λ_{L} with the ability to write non-terminating programs(see [59] or the next chapter), and in fact all partial recursive functions. The paper also proposes a term that uses aspects to implement a reference cell. Indeed the purpose of chapter 3 is to make this connection precise, by formally showing that not only can these aspectual features emulate references, but that the reverse is also possible.

Another feature of λ_{L} is the `return` expression, which takes a label and a value, and immediately passes the value to the nearest enclosing occurrence of the label. The MinAML semantics of [59] uses this primitive specifically to encode `around()` advice which does not call `proceed()`, i.e. which completely replaces the base code that it advises.

The fact that λ_{L} is built as an extension to the simply typed λ-calculus makes it an ideal language for semantics research, as this enables comparison to calculi which add state, control, non-determinism etc. Additionally, it it is an excellent framework on which to add further AOP features orthogonally. In fact, Walker et al. have developed a number of such extensions:

- A typed version of ς-calculus style objects are added to MinAML and λ_{L} by routine modification of the syntax and operational semantics.

- More complex pointcuts, in the form of *sets* of labels, allowing users to install advice of the form $\{\{\alpha_1, \ldots, \alpha_n\}.x \to M\}$ which executes at any point labelled by one of the α_i.

- The ability to dynamically store values on a stack, and to match the current stack against a pattern, enabling dynamic point-cut descriptors like `cflow()`.

Polymorphic Aspects

The fact that advice must share a type with the program points they advise in MinAML is clearly a departure from AspectJ, but it is not clear how debilitating a restriction it is. As shown in later chapters, some useful aspects can certainly be written using it, but many others cannot. A prototypical example of the latter is an aspect which only executes a side-effect at (say) every function call, for example for logging. PolyAML [16] is a polymorphic extension of MinAML: pointcuts are sets of function names which are collectively given a polymorphic type. It is defined, as MinAML was, by translation into a core calculus F_A, which extends λ_L with System F (i.e. type abstraction and application) and polymorphic labels.

To label a term with a label α of type $\mathtt{lab}[\overline{\mathrm{Z}_i}.\tau]$ (where the Zs are type variables bound in τ), the programmer uses the syntax $\alpha[\overline{\tau_i}]\langle\!\langle M \rangle\!\rangle$ which now causes variable *and* type substitution in advice associated with α. Furthermore, F_A imposes a partial ordering on labels, reserving a special label at the top of the hierarchy. This allows programmers to write aspects which, for example, are triggered at every labelled control flow point regardless of type.

Jagadeesan et al. [30] also study the interaction of polymorphism and AOP in an object-oriented setting.

Type Extension

AspectJ allows programmers to add fields and methods to objects within aspects, and have the additions hold only within the aspect's scope. This feature has not been studied formally at all, but its interaction with the functional paradigm has been explored in AspectualCAML [56]: an implementation of aspects in a language similar to Objective CAML.

To illustrate the type extension feature, consider the example from [56] of a simple language interpreter which allows integers, variables, addition, and `let` binding. The interpreter consists of a type declaration establishing the syntax of terms, and an evaluation function defining their semantics:

```
type id = I of string

type t = Num of int
       | Var of id
       | Add of t * t
       | Let of id * t * t

let rec eval env t = (* code for eval *)
```

Now suppose that a programmer wishes to extend the language with a new term constructor for subtraction, AspectualCAML allows this to be done *externally* by defining and installing the following aspect:

```
aspect AddSubtraction
  type+ t = ... | Sub of t * t
  advice eval_sub = [around call eval env t]
                      match t with
                            Sub(t1,t2) -> (* code for subtraction *)
                          | _ -> proceed t
end
```

The first line declares that (within the aspect's code), the type `t` has an additional two argument constructor `Sub`. The advice executes around a call to the evaluation function, and handles the case where the given term is a subtraction term, passing control to the original evaluator (via a call to `proceed`) in other cases.

Module Systems and Aspects

A serious problem with AOP is that its unrestricted use erodes modularity boundaries that were enforced in the language to which it is added. For instance, any assumptions that programmers could make about objects in Java– say about the behaviour of methods in its interface – may as well be thrown out if the code can be advised with AspectJ. Furthermore, the obliviousness of the language means that the base code programmer is given no indication that such effects may be present.

In an excellent ECOOP paper [9], Jonathan Aldrich introduced Open Modules in an effort to preserve the desirable features provided by AOP, while exerting some level of control over them so that the aspect language maintains some desirable properties. Open Modules lets a programmer define functions and pointcuts inside of ML-style structures, but also requires them to *explicitly export* those which can be externally advised. This is achieved using a syntax similar to ML's module system, with a sealing operator `:>` allowing a structure's interface to be restricted to those declared in a signature. For example, the declaration

```
structure S = struct
      val f1 = fn ...
      val f2 = fn ...
      val f3 = fn ...
      pointcut p1 = call(f1)
      pointcut p2 = call(f3)
end :> sig
          f1 : A1 -> B1
```

```
    f2 : A2 -> B2
    p1 : pc(A1 -> B1)
end
```

exposes the functions `f1`,`f2` and the pointcut `p1` to external advice, but keeps `f3` and `p2` internal to the structure and therefore not advisable from outside it.

This seemingly straightforward extension of sealing to advice retains much of the usefulness of AOP, but restricts it to a well behaved subset: Aldrich defines a set of inference rules for a logical equivalence relation that is conservative with respect to observational equivalence. The results in chapter 5 use similar ideas, but build a fully abstract semantics of a MinAML-style language.

1.3 Thesis Summary

This thesis is concerned with giving formal semantics for aspect oriented programming languages by translation into traditional paradigms such as state and control. In particular, the aim is to quantify the tightness of the semantics by proving that they satisfy certain criteria, which are defined in chapter 2. Of particular interest is the construction of *fully abstract* translations, i.e. those which can be used to reason about observational equivalence soundly and completely. These results are made possible by restricting AOP in the following ways:

- The languages studied are generally not oblivious. As mentioned above, this is not a serious restriction, as it is now accepted as a necessary (and indeed desirable) restriction. As such, it generally falls in line with other theoretical studies.

- Aspects are *dynamically allocated*, i.e. evaluating a term may cause further aspects to be installed. This is a departure from AspectJ and languages such as $\varsigma_A(M)$, which evaluate terms in the presence of a static list of aspects. However, dynamic

allocation does fit snugly with the functional paradigm, and is used in most programming language research in this area [59, 9, 56, 15, 61, 31].

- Pointcuts are *in-scope*, meaning that every pointcut descriptor refers to the join points they advise (usually function calls) explicitly by name. This is not a trivial restriction, and the prospects of relaxing it are discussed in further detail in chapter 7. In-scope aspects are used in much of the recent theoretical research in AOP [59, 9, 62, 31, 53].

The starting point of the research is Walker et al.'s MinAML and its associated core language λ_L, both of which are defined formally in the next chapter.

Chapter 3 restricts λ_L to a fragment not including the `return` primitive, and shows that the resulting language can soundly be translated into a language with higher-order storage. Furthermore, if a standard object-oriented view of labels and references is taken, the translation can be made fully abstract. This implies that the translation function can be pre-composed with the game semantics model of higher order store [3] yielding a fully abstract game model of the restricted aspect calculus. This model is, for similar reasons, also an adequate model of the fragment of MinAML not requiring `return` to encode.

Chapter 4 considers the full aspect calculus (including `return`) and shows that it cannot be modelled using higher-order store alone. A translation is defined into a language which also includes exception handling, effectively resulting in a core calculus for ML. The translation is proved adequate, and a reasonable conjecture states that similar techniques to those of chapter 3 can make it fully abstract. The absence of a fully abstract model of the target language meant an inability to prove the full abstraction result for the translation (hence the conjecture).

Chapter 5 proposes an alternate semantics of the MinAML constructs which *can* be fully abstractly translated into higher-order storage even in the presence of non-proceeding around advice. The ideas used in this translation lead to a natural implementation in Standard ML, which is described in chapter 6, and used to demonstrate few extensions – such as some dynamic pointcuts and a simple integration into the ML module system – to the bare-bones MinAML features. These additional features could relatively easily be incorporated into the model. A number of examples using the implementation explore it's utility and limitations.

The final chapter discusses the restriction to in-scope pointcuts, and what would be required to remove it, and also summarises some of the issues around the extension of this work to formal verification.

Chapter 2

Technical Background

This chapter presents the formal background required to understand the results that follow it. In particular, the first section introduces the concept of *compositional translation* [45, 52] as a means of defining the semantics of programming languages. The focus is on defining criteria whereby the merit of such semantics can be quantified, i.e. a list of desirable properties that express how tightly the operational behaviour of the target language emulate that of the source.

The second half of the chapter formally defines the aspect language MinAML [59], the basis of this thesis' research. The operational semantics of the language are themselves defined by translation into a core calculus of aspects λ_{L}.

2.1 Semantics of Languages by Compositional Translation

A *denotational semantics* for a language $\mathcal{L}$ is a function $\mathcal{M}[\![-]\!] : \mathcal{L} \to \mathcal{D}$, which assigns an element $\mathcal{M}[\![M]\!] \in \mathcal{D}$ of some mathematical structure $\mathcal{D}$ to each valid phrase $M \in \mathcal{L}$. Typically, one asks that this function respects some properties that show a correspondence between the mathematical interpretation of the terms and the operational semantics of

the language. For instance, if a term M evaluates to a value V, one might ask that M and V have the same interpretation in $\mathcal{D}$, i.e. that $\mathcal{M}[\![M]\!] = \mathcal{M}[\![V]\!]$. If the range $\mathcal{D}$ is itself a language, then the semantic function $\mathcal{M}[\![-]\!]$ becomes a translation between two languages, but similar properties can be established so long as there is a suitable notion of equivalence between terms of the language.

A translation is *compositional* if the translate of a program is a composite of the translates of its component phrases; compositional translations are typically defined by recursion on program syntax. If a translation $\mathcal{F}[\![-]\!]$ is compositional, a denotational semantics $\mathcal{M}[\![-]\!]$ of the target language can be transformed to a denotational semantics of the source language by precomposing the semantic function with the translation map, i.e. the interpretation of a term M in the source language is simply $\mathcal{M}[\![\mathcal{F}[\![M]\!]]\!]$. Further, the more faithful the translation, the greater the extent to which goodness-of-fit properties of $\mathcal{M}[\![-]\!]$ are inherited by the source-language semantics $\mathcal{F}[\![-]\!]$.

Quantifying the faithfulness of translations

Fix typed source and target languages $\mathcal{S}$ and $\mathcal{T}$. Let M, N range over their well typed terms, V over a subset of terms called *values*, and σ, τ over types. The phrase $\Gamma \vdash M : \tau$ means that the term M has type τ under typing assumptions Γ. Omitting either the type or the context implies an existential quantification over the missing component, e.g. $\vdash M : \tau$ means that there is some context such that M has type τ.

Assume that each language has an operational semantics defined as an evaluation relation (i.e. relating terms and values) denoted by $\Downarrow$. Write $M \Downarrow V$ to mean "M *evaluates* to value V", and $M\Downarrow$ to mean "M *converges*", i.e. it evaluates to some value. An important and compelling notion of program equivalence is observational equivalence. Intuitively two terms are *observationally equivalent*, written $M \simeq N$, just if one can be replaced by the other in *all* programs without causing any observable difference in the

computational outcome. Slightly more formally, two terms M, N of type τ are equivalent if for all contexts $\mathcal{C}[-]$ – i.e. terms with an arbitrary number of holes of type τ – the result of plugging M into the holes (denoted $\mathcal{C}[M]$) converges if and only if $\mathcal{C}[N]$ converges.

A basic property one asks of a translation $[\![-]\!]$ from terms and types of $\mathcal{S}$ to those of $\mathcal{T}$ is the following:

Property E (Evaluation Equivalence)

For every M and V, $M{\Downarrow_{\mathcal{S}}}V$ if, and only if for some V', $[\![M]\!]{\Downarrow_{\mathcal{T}}}V'$ and $V' \sim_{\mathcal{T}} [\![V]\!]$

A different notion of equivalence ($\sim$) is assumed for values because, in the languages considered in the rest of the thesis, contextual equivalence isn't necessarily well-defined for values. For the moment, the $\sim$ relation is left undefined, and should just be taken as a statement that it relates values that are "the same" according to some measure. In case a translation satisfies property E, the target language can be regarded as an emulator of the evaluation relation of the source language with respect to $\sim$. In other words, a term in either language can be soundly evaluated to a value by translating it into the other and using the interpreter for the target language. A weaker version of this property only requires that the termination relation is preserved and reflected, and is often sufficient to prove stronger properties:

Property T (Termination Equivalence)

For every M, $M{\Downarrow_{\mathcal{S}}}$ if, and only if $[\![M]\!]{\Downarrow_{\mathcal{T}}}$

Call a translation *adequate* if it reflects observational equivalence, and *fully abstract* if, in addition, it preserves observational equivalence. An important aim of this thesis is to

exhibit fully abstract translations of aspect calculi. A useful technical condition, often a pre-condition of full abstraction, is the following:

Property D (Definability)

For every $M \in \mathcal{T}$, there is some $N \in \mathcal{S}$ such that $[\![N]\!] \simeq_{\mathcal{T}} M$

This states that, modulo observational equivalence, the translation map is a surjection. Another requirement, as well as a precondition of properties E, T, and D is the following substitution result:

Property S (Substitution)

Given a term M with free variable x and a term N of the same type as x:

1. $\vdash M[N/x]$ *if, and only if* $\vdash [\![M]\!][[\![N]\!]/x]$
2. *If* $\vdash M[N/x]$ *then* $[\![M[N/x]]\!] = [\![M]\!][[\![N]\!]/x]$

Every translation defined in the following chapters translates a variable x in the source language to itself. Therefore, capture of variables by a substitution is a non-issue: capture occurs in the source if and only if it occurs in the translate. Therefore, by viewing the holes in a context as a free variable, the following corollary of property S immediately follows:

Property C (Compositionality)

Given a context $\mathcal{C}[-]$ *and a term* M *of the same type as the hole(s):*

1. $\vdash \mathcal{C}[M]$ *if, and only if* $\vdash [\![\mathcal{C}]\!][[\![M]\!]]$
2. *If* $\vdash \mathcal{C}[M]$ *then* $[\![\mathcal{C}[M]]\!] = [\![\mathcal{C}]\!][[\![M]\!]]$

The following proof shows that properties C and T are sufficient conditions for adequacy:

Property A (Adequacy)

For every M *and* N*, if* $[\![M]\!] \simeq_\tau [\![N]\!]$ *then* $M \simeq_s N$

Proof. Let M and N be such that $[\![M]\!] \simeq [\![N]\!]$, and let $\mathcal{C}[-]$ be an arbitrary context such that both $\mathcal{C}[M]$ and $\mathcal{C}[N]$ are well typed programs. By property C.1, $[\![\mathcal{C}]\!][-]$ is a valid program context for $[\![M]\!]$ and $[\![N]\!]$. Since observational equivalence is a congruence, we have $[\![\mathcal{C}]\!][[\![M]\!]] \simeq [\![\mathcal{C}]\!][[\![N]\!]]$; but then by property C.2 we have $[\![\mathcal{C}]\!][[\![M]\!]] = [\![\mathcal{C}[M]]\!]$ and similarly for N. Hence $[\![\mathcal{C}[M]]\!] \simeq [\![\mathcal{C}[N]]\!]$. Now suppose $\mathcal{C}[M]\Downarrow$; by property T $[\![\mathcal{C}[M]]\!]\Downarrow$. Since $[\![\mathcal{C}[M]]\!] \simeq [\![\mathcal{C}[N]]\!]$, we have $[\![\mathcal{C}[N]]\!]\Downarrow$. Hence property T again implies that $\mathcal{C}[N]\Downarrow$. Symmetrically, we can prove that if $\mathcal{C}[N]$ converges then $\mathcal{C}[M]$ converges. Therefore $M \simeq N$ as desired. □

Notice that only the forward direction of property C.1 was required for this proof. Full abstraction also requires the reverse direction of C.1, as well as property D:

Property F (Full Abstraction)

For every M and N, $M \simeq_{\mathcal{S}} N$ if, and only if $[\![M]\!] \simeq_{\mathcal{T}} [\![N]\!]$.

Proof. It remains to prove the forward direction: that the translation preserves observational equivalence. We prove the contrapositive. Suppose for some M and N that $[\![M]\!] \not\simeq [\![N]\!]$. By definition, there exists a closing context $\mathcal{C}[-]$ such that (without losing generality) $\mathcal{C}[[\![M]\!]]$ converges but $\mathcal{C}[[\![N]\!]]$ diverges. By property D, $\mathcal{C}[-] \simeq [\![\mathcal{D}[-]]\!]$ for some context $\mathcal{D}[-]$ of $\mathcal{S}$. Since $\simeq$ is a congruence, we know that $[\![\mathcal{D}]\!][[\![M]\!]]$ converges and $[\![\mathcal{D}]\!][[\![N]\!]]$ does not. Now apply property C.1 to deduce that $\mathcal{D}[M]$ and $\mathcal{D}[N]$ are well-typed terms and that

$$[\![\mathcal{D}[M]]\!] \Downarrow \quad \text{and} \quad \neg([\![\mathcal{D}[N]]\!] \Downarrow).$$

But we are now done, because property T implies that $\mathcal{D}[M]$ terminates and $\mathcal{D}[N]$ diverges. Therefore $M \not\simeq N$ as required. □

Recall that a denotational model for a language is fully abstract if the equational theory of the model coincides with observational equivalence. Fully abstract models are thus powerful and highly accurate models. An important reason for studying fully abstract translation is that it offers a way to build a fully abstract semantics for a language. McCusker [45] showed that if the translation is adequate (respectively fully abstract), then the semantics inherited by the source language by composition of the valuation function is also adequate (respectively fully abstract).

2.2 Formal Definition of MinAML

This section is a summary of the main results of [59], which presents MinAML, an aspectual extension of a small ML-like language, and defines its operational semantics by translation into a core calculus. The presentation restricts itself to the most basic form of MinAML presented in that paper. It first defines the syntax of MinAML, then the syntax and operational semantics of the core calculus, and finally the translation from the former to the latter.

Syntax

The only types of MinAML are function types $\sigma \to \tau$ built from a ground type of booleans. The valid terms are those which can be constructed from the following grammar:

$$M, N ::= x \mid \texttt{tt} \mid \texttt{ff} \mid \texttt{if}\ M\ \texttt{then}\ N_1\ \texttt{else}\ N_2 \mid \texttt{let}\ D\ \texttt{in}\ M \mid M \cdot N$$

where the `let` statement binds a sequence D of *declarations* in the term M, which are constructed as follows

$$\begin{array}{rcl} D & ::= & (\texttt{bool}\ x = M)\ D \\ & \mid & (\texttt{fun}\ f(x : \sigma) : \tau = M)\ D \\ & \mid & A\ D \end{array}$$

$$\begin{array}{rcl} A & ::= & \texttt{before}\ f(x) = M \\ & \mid & \texttt{after}\ f(x) = M \\ & \mid & \texttt{around}\ f(x) = M \\ & \mid & \texttt{around}\ f(x) = M; \texttt{proceed}\ y \to N \end{array}$$

The function identifiers f, which form the join points of the language, are simply taken to be ordinary variables. Term typing is defined in the usual way, built up from boolean

constants and variables, with functions forming the abstractions in the language. For a function of type $\sigma \to \tau$, advice is typed according to its execution time:

- `before` $f(x) = M$ is typed so that $x : \sigma \vdash M : \sigma$, i.e. the input and body match the source type of the function.
- `after` $f(x) = M$ is typed so that $x : \tau \vdash M : \tau$, similarly matching the function's output type.
- `around` $f(x) = M$ is typed so that $x : \sigma \vdash M : \tau$, i.e. it takes the source type as input, but returns the target type.
- `around` $f(x) = M$; `proceed` $y \to N$ is typed so that $x : \sigma \vdash M : \sigma$ and $y : \tau \vdash N : \tau$ i.e. M and N are typed as `before` and `after` advice respectively.

Example 2.2.1 (MinAML Semantics)**.** The semantics of `before` and `after` advice are quite intuitive: the former binds f's input to x and returns the result of M which is fed back the function, while the latter similarly binds f's result to x. For example, momentarily assuming the presence of integer arithmetic in the language, consider the term

$$\begin{aligned}&\texttt{let fun incr}(x : \texttt{int}) : \texttt{int} = x + 1\\ &\quad tm\ \texttt{incr}(x) = x^2\\ &\texttt{in incr}(3)\end{aligned}$$

which declares an increment function and a piece of advice with execution time tm. The result of subsequently evaluating `incr`(3) is 10 if $tm =$ `before`, and 16 if $tm =$ `after`, reflecting the fact that the squaring takes place before and after the increment respectively.

The semantics of `around` advice, which comes in two forms, is more subtle. The non-proceeding form causes the body of the advised function to be completely replaced by

the body of the advice, so setting $tm = \texttt{around}$ in the above causes $\texttt{incr}(3)$ to evaluate to 9 because the the body of `incr` never executes. The second form of `around` advice is essentially equivalent to installing `before` advice with body M and `after` advice with body N simultaneously. The former is evaluated with the call point input to f bound to x. The result is fed back to f (indicated by the `proceed` keyword), the result of which is bound to y in N. For example, assuming some basic I/O primitives, the following advice logs entries and exits to `incr` without altering its computational behaviour:

$$\texttt{around incr}(x) = \texttt{print ``Enter : incr''}; x;$$
$$\texttt{proceed}\ y \rightarrow \texttt{print ``Leave : incr''}; y$$

The variables x and y must follow the `print` functions so that the correct values are passed to the proceeding function and returned as the result.

Definition of the labelled aspect calculus

Most of the languages studied in this thesis are defined as extensions to λ: the simply typed, call-by-value λ-calculus with boolean and unit types. The types and expressions of this language are given by the following grammars:

$$\begin{array}{rcl} \sigma, \tau & ::= & \texttt{unit} \mid \texttt{bool} \mid \tau_1 \times \tau_2 \mid \sigma \rightarrow \tau \\ M, N & ::= & x \mid \texttt{skip} \mid \texttt{tt} \mid \texttt{ff} \mid \texttt{cond}\ M\ N_1\ N_2 \mid \\ & & \langle M, N \rangle \mid \pi_i(M) \mid \lambda x : \tau.M \mid M \cdot N \end{array}$$

The type system and operational semantics are standard. and presented in appendix A.1. The semantics is defined as both a "small step" transition style relation $\longrightarrow_\lambda$, and as a "big step" evaluation relation $\Downarrow_\lambda$ between expressions and *values*, which are terms gen-

erated by the following grammar:

$$V ::= \texttt{skip} \mid \texttt{tt} \mid \texttt{ff} \mid \langle V_1, V_2 \rangle \mid \lambda x : \tau.M$$

The subscripts on the operational relations are usually omitted, and terms are equated up to α-renaming of bound names. Given a sequence $\overline{V_i}$ of identically typed function values, the notation $V_1 \circ \cdots \circ V_n$ is used to mean the functional composition of the terms in the sequence:

$$\lambda x : \tau.(V_1 \cdot (V_2 \cdot (\ \cdots\ \cdot (V_n \cdot x)\ \cdots\ )))$$

By convention, using the metavariable d as the bound variable in a lambda term indicates a dummy (ignored) input. Formally, a term $\lambda d : \tau.M$ implicitly assumes that $d \notin \texttt{fv}(M)$.

Extend this language to the aspect calculus λ_{L} by assuming a countable set of *label names* ranged over by α, and add a type of labels and a type of aspects, together with constructs to generate (locally scoped) labels, label program points, create and install aspects, and return a value to a label:

$$\begin{aligned}
\sigma, \tau &::= \cdots \mid \texttt{lab}[\tau] \mid \texttt{asp}[\tau] \\
M, N &::= \cdots \mid \alpha \mid \texttt{newlab}[\tau] \mid \{M.x \to N\} \\
&\quad \mid M\langle\!\langle N \rangle\!\rangle \mid M \gg N \mid \texttt{return}\ M\ \texttt{to}\ N \\
V &::= \cdots \mid \alpha \mid \{V.x \to M\}
\end{aligned}$$

The typing rules extend those of λ with the rules in table 2.1, and are parametrised by a map L from label names to types. Though label names appear in the syntax, they only arise as the outcome of an evaluation, and may not appear in user programs. Note in particular the rule for `return`, which states that as long as the the types of the label and the returned value match, the return statement can take any type. This means that a

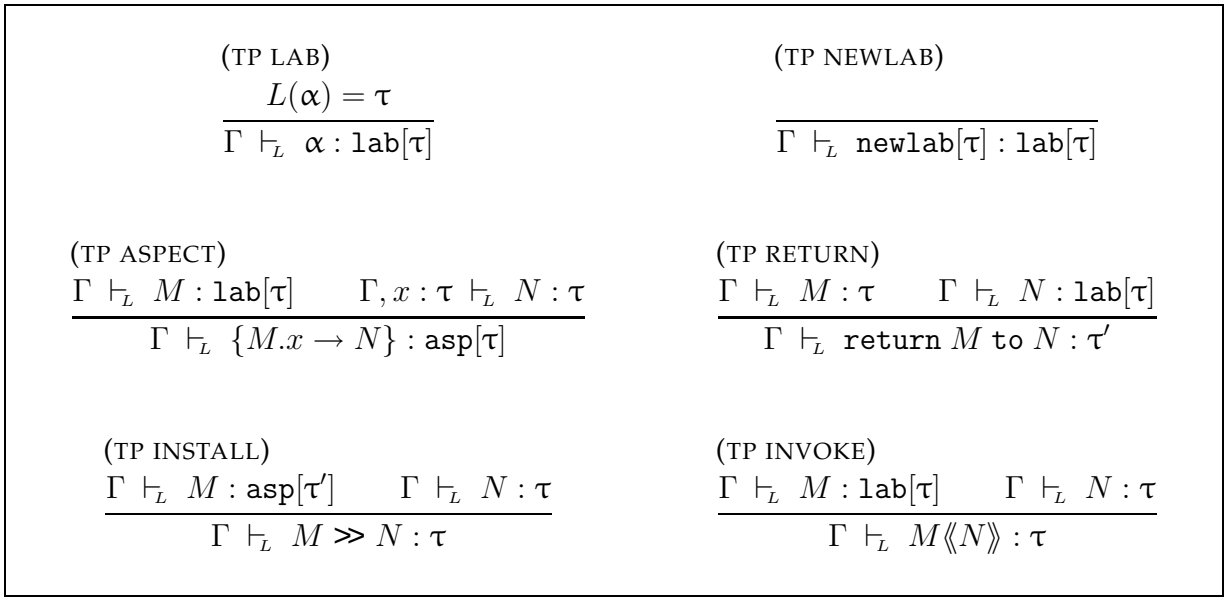


Table 2.1: *Additional typing rules for* λ_{L}

`return` statement may appear in any evaluation context, similar to raising an exception in ML. This connection is explored in greater detail in chapter 4.

A *configuration* is a triple (L, A, M) comprising a map L from labels to types, a sequence A of aspects, and a term M to be evaluated; a *value configuration* is one whose term component is a value, and a *returning configuration*'s term has the form `return` V `to` α. The operational semantics is specified by a relation $\longrightarrow$ between configurations, and is identical to the one defined in [59], save for the following alterations:

- The aspect installation primitive which prepends to the head of the aspect sequence is omitted. While it would be easy to include it, as it would not affect any technical results, it is not used in the encoding of MinAML, nor is it needed to encode general references. It would serve only to clutter the presentation.

- Walker et al.'s semantics terminate if a configuration was reached which evaluated a `return` statement that was not enclosed by its label, i.e. one whose term had

the form $E[\texttt{return}\ V\ \texttt{to}\ \alpha]$ where α does not enclose the `return`. The semantics here add an additional rule which takes such a configuration to `return` V `to` α immediately, eliminating its surrounding context.

The formal rules appear in table 2.2. The rules defining the operational semantics of λ are naturally lifted to the corresponding configurations: they make no direct changes to the L or A components. The "::" symbol is used to concatenate elements to the ends of sequences, and the empty sequence is written as $[\,]$. The set $\texttt{dom}(F)$ is the domain of the map F, and $F[x \mapsto n]$ is the same map as F except that x is remapped to n. The notation $\texttt{id}[\tau]$ is a shorthand for the identity function $\lambda x : \tau.x$ of type τ.

A term is *closed* if it contains no free variables, and *open* otherwise, and terms that contain no label names are *user terms*. A *program* is a closed user term of ground type. Some auxiliary definitions are used in the evaluation rules:

- A family of maps $\texttt{lookup}\langle A, L\rangle(-)$ which, given a label α, return the term M obtained by functionally composing all α-labelled advice in A in sequential order.
- A map $\texttt{stack}()$ taking an evaluation context and returning the sequence of labels that surround the hole. This function provides some dynamic information about the execution state and is used in the reduction rules for `return`.

A configuration is said to be well typed if the map L correctly realises the types of labels in the term M, and if every element of A is a well typed aspect with respect to L. Formally, this is stated as follows:

Definition 2.2.1 (λ_{L} Well Typed Configuration)**.** A configuration (L, A, M) is well typed, denoted $\vdash (L, A, M)$, if for some Γ and τ, the following conditions are met:

1. $\Gamma \vdash_L M : \tau$ is provable
2. $\Gamma \vdash_L \{\alpha.x \to M\} : \texttt{asp}[L(\alpha)]$ is provable for each aspect $\{\alpha.x \to M\} \in A$,

EVALUATION CONTEXTS

$$\begin{array}{rcl} E & ::= & \cdots \mid E \gg M \mid E\langle\!\langle M \rangle\!\rangle \mid \alpha\langle\!\langle E \rangle\!\rangle \mid \{E.x \to M\} \\ & & \mid \texttt{return}\ E\ \texttt{to}\ M \mid \texttt{return}\ V\ \texttt{to}\ E \end{array}$$

STACK FUNCTION

$$\texttt{stack}([-]) = [\,] \qquad \texttt{stack}(\alpha\langle\!\langle E \rangle\!\rangle) = \texttt{stack}(E) :: \alpha$$

β-REDUCTION

$$\begin{array}{rcll} (L, A, \texttt{newlab}[\tau]) & \xrightarrow{\beta} & (L[\alpha \mapsto \tau], A, \alpha) & [\alpha \notin \texttt{dom}(L)] \\ (L, A, V \gg M) & \xrightarrow{\beta} & (L, A :: V, M) & \\ (L, A, \alpha\langle\!\langle E[\texttt{return}\ V\ \texttt{to}\ \alpha] \rangle\!\rangle) & \xrightarrow{\beta} & (L, A, \alpha\langle\!\langle V \rangle\!\rangle) & [\alpha \notin \texttt{stack}(E)] \\ (L, A, \alpha\langle\!\langle V \rangle\!\rangle) & \xrightarrow{\beta} & (L, A, M[V/x]) & [\texttt{lookup}\langle L, A\rangle(\alpha) = \lambda x.M] \end{array}$$

ASPECT LOOKUP

$$\begin{array}{rcll} \texttt{lookup}\langle L, [\,]\rangle(\alpha) & \triangleq & \texttt{id}[L(\alpha)] & [\alpha \in \texttt{dom}(L)] \\ \texttt{lookup}\langle L, \{\alpha.x \to M\} :: A\rangle(\alpha) & \triangleq & \texttt{lookup}\langle L, A\rangle(\alpha) \circ \lambda x.M & \\ \texttt{lookup}\langle L, \{\alpha'.x \to M\} :: A\rangle(\alpha) & \triangleq & \texttt{lookup}\langle L, A\rangle(\alpha) & [\alpha \neq \alpha'] \end{array}$$

REDUCTION RULE

$$\frac{(L, A, M) \xrightarrow{\beta} (L', A', M')}{(L, A, E[M]) \longrightarrow (L', A', E[M'])} \qquad \frac{\alpha \notin \texttt{stack}(E)}{(L, A, E[\texttt{return}\ V\ \texttt{to}\ \alpha]) \longrightarrow (L, A, \texttt{return}\ V\ \texttt{to}\ \alpha)}$$

Table 2.2: *Operational semantics of* λ_{L}

The operational rules closely follow their intuitive descriptions in section 1.2.3. The `newlab` expression generates a fresh label and adds it to the label map L, the installation primitive simply evaluates the aspect and appends it to the end of the sequence A, and the invocation expression looks up the appropriate advice in A and applies the resulting function to the input value. The `return` rule finds the nearest enclosing occurrence of the label α by inspecting the `stack`(). Syntactic sugar for sequential composition, `let` binding, and block structured labels (`newlab` ... `in` ...) are defined using λ abstraction in the typical call-by-value fashion:

$$\begin{aligned}
\texttt{let}\ x = M_1\ \texttt{in}\ M_2 &\triangleq (\lambda x.M_2) \cdot M_1 \\
\texttt{newlab}\ \tau : x\ \texttt{in}\ M &\triangleq (\lambda x : \tau.M) \cdot \texttt{newlab}[\tau] \\
M_1; M_2 &\triangleq (\lambda d.M_2) \cdot M_1 \quad [d \notin \texttt{fv}(M_2)]
\end{aligned}$$

Example 2.2.2 (Non Termination)**.** The addition of these primitives to the λ-calculus strictly increase its expressive power, as they allow non-terminating terms. Consider the following:

$$\texttt{newlab}\ \ell : \texttt{bool}\ \texttt{in}\ \{\ell.x \to \ell\langle\!\langle x \rangle\!\rangle\} \gg \ell\langle\!\langle \texttt{tt} \rangle\!\rangle$$

This term declares a label, installs a piece of advice on it, then evaluates the labelled value `tt`. Note that, stripped of its AOP elements, this term is already a value. However, let $L = \{\ell \mapsto \texttt{bool}\}$ and $A = [\{\ell.x \to \ell\langle\!\langle x \rangle\!\rangle\}]$, then after declaring the label and installing the advice, evaluation proceeds as follows[1]:

$$(L, A, \ell\langle\!\langle \texttt{tt} \rangle\!\rangle) \longrightarrow (L, A, (\lambda x.\ell\langle\!\langle x \rangle\!\rangle) \cdot \texttt{tt}) \longrightarrow (L, A, \ell\langle\!\langle \texttt{tt} \rangle\!\rangle)$$

where the first step follows because $\texttt{lookup}\langle A, L\rangle(\ell) = \lambda x.\ell\langle\!\langle x \rangle\!\rangle$. In other words, even

[1]There is a slight abuse of notation here as the label name generated by `newlab` is identified with the variable name to which it is declared

though the base code is already a value, the presence of aspects causes this term to diverge.

The usual notion of observational equivalence can now be defined on this language. Its formal definition is as follows:

Definition 2.2.2 (Observational Equivalence)**.** Well-typed λ_{L} user terms M_1 and M_2 are *observationally equivalent*, denoted $M_1 \simeq M_2$, if for any context $\mathcal{C}[-]$: `unit` such that $\mathcal{C}[M_i]$ are programs, we have (for some L_1, L_2, A_1, A_2)

$$(\bot, \bot, \mathcal{C}[M_1]) \longrightarrow^* (L_1, A_1, \texttt{skip}) \quad \Longleftrightarrow \quad (\bot, \bot, \mathcal{C}[M_2]) \longrightarrow^* (L_2, A_2, \texttt{skip})$$

where $\longrightarrow^*$ is the reflexive and transitive closure of $\longrightarrow$.

The core labelled calculus was proved type-safe by Walker et al. [59] via the standard progress and preservation theorems:

Theorem 2.2.1 (λ_{L} Progress)**.** *If C is a well typed configuration, then either it is a value configuration, a returning configuration, or there exists another configuration C' such that $C \longrightarrow C'$.*

Theorem 2.2.2 (λ_{L} Preservation)**.** *If (L, A, M) is well typed and $(L, A, M) \longrightarrow (L', A', M')$ then L' extends L and (L', A', M') is well typed.*

Operational semantics of MinAML by translation

In order to define the semantics of MinAML, the meaning of "before" and "after" a function call must be made precise. Walker et al. define the former as the point immediately after the evaluation of the function's arguments, and the latter as the point immediately after the evaluation of the body.

The formal semantics are defined by three translation functions $\mathcal{T}[\![-]\!]$, $\mathcal{D}[\![-]\!]$, and $\mathcal{A}[\![-]\!]$, defined recursively on the syntax of terms, declarations, and aspect declarations

respectively. The term translation function takes a MinAML expression of type τ to a λ_L term of the same type. It's operation on booleans, variables, conditionals, and applications is the obvious one. Translation of the MinAML `let` expression is passed to the $\mathcal{D}[\![-]\!]$ function:

$$\mathcal{T}[\![\texttt{let } D \texttt{ in } M]\!] \triangleq \mathcal{D}[\![D; M]\!]$$

Unsurprisingly this function consists of the most interesting clauses of the semantics, particularly that of function declarations. Intuitively, a MinAML function is translated to an abstraction which has two labels at the before and after points acting as "hooks" on which advice can be attached. In order to have access to these program points, an abstraction $\lambda x.M$ is expanded into the equivalent $\lambda x.\texttt{let } x = x \texttt{ in } M$, which can then be annotated with before and after labels β and α:

$$\lambda x.\alpha\langle\!\langle \texttt{let } x = \beta\langle\!\langle x \rangle\!\rangle \texttt{ in } M \rangle\!\rangle$$

This is used when translating a function declaration, which declares fresh labels for each function:

$$\begin{aligned}\mathcal{D}[\![(\texttt{fun } f(x : \tau_1) : \tau_2 = M)\ D; N]\!] \triangleq\ & \texttt{newlab } f_\beta : \tau_1 \texttt{ in} \\ & \texttt{newlab } f_\alpha : \tau_2 \texttt{ in} \\ & \texttt{let } f = \lambda x.f_\alpha\langle\!\langle \texttt{let } x = f_\beta\langle\!\langle x \rangle\!\rangle \texttt{ in } \mathcal{T}[\![M]\!] \rangle\!\rangle \texttt{ in} \\ & \mathcal{D}[\![D; N]\!]\end{aligned}$$

This means that for each function variable in the MinAML typing context, two label variables must be added in the corresponding λ_L context. Using this labelled function, advice is assigned with the installation operator:

$$\mathcal{D}[\![A\ D; M]\!] \triangleq \mathcal{A}[\![A]\!] \gg \mathcal{D}[\![D; M]\!]$$

which delegates the responsibility of deciding exactly what to install to the $\mathcal{A}[\![-]\!]$ translation:

$$\begin{aligned}
\mathcal{A}[\![\texttt{after } f(x) = M]\!] &= \{f_\alpha.x \to \mathcal{T}[\![M]\!]\} \\
\mathcal{A}[\![\texttt{before } f(x) = M]\!] &= \{f_\beta.x \to \mathcal{T}[\![M]\!]\} \\
\mathcal{A}[\![\texttt{around } f(x) = M_1; \texttt{proceed } y \to M_2]\!] &= \{f_\beta.x \to \mathcal{T}[\![M_1]\!]\} \gg \{f_\alpha.y \to \mathcal{T}[\![M_2]\!]\} \\
\mathcal{A}[\![\texttt{around } f(x) = M]\!] &= \{f_\beta.x \to \texttt{return } \mathcal{T}[\![M]\!] \texttt{ to } f_\alpha\}
\end{aligned}$$

These definitions correspond with the intuitive semantics of advice presented when introducing MinAML's syntax: before and after advice append themselves to the appropriate program point, proceeding around advice does both, and non-proceeding around advice evaluates its body before executing the function call then jumps past it using the `return` statement. Figure 2.1 is a graphical representation of the net effect of aspect installation in MinAML according to these semantics.

Example 2.2.3. Contrast the translations of the terms discussed in example 2.2.1. When $tm = \texttt{before}$, the resulting translation is

$$\begin{aligned}
&\texttt{newlab incr}_\beta : \texttt{int in} \\
&\texttt{newlab incr}_\alpha : \texttt{int in} \\
&\texttt{let incr} = \lambda x.\texttt{incr}_\alpha\langle\!\langle \texttt{let } x = \texttt{incr}_\beta\langle\!\langle x \rangle\!\rangle \texttt{ in } x + 1 \rangle\!\rangle \texttt{ in} \\
&\{\texttt{incr}_\beta.x \to x^2\} \gg \texttt{incr} \cdot 3
\end{aligned}$$

After declaring the labels, installing the advice, and substituting the input value 3 into the body of the `incr` function, the resulting configuration is (L, A, M) with

$$\begin{aligned}
L &= \{\texttt{incr}_\beta, \texttt{incr}_\alpha\} \\
A &= [\{\texttt{incr}_\beta.x \to x^2\}] \\
M &= \texttt{incr}_\alpha\langle\!\langle \texttt{let } x = \texttt{incr}_\beta\langle\!\langle 3 \rangle\!\rangle \texttt{ in } x + 1 \rangle\!\rangle
\end{aligned}$$

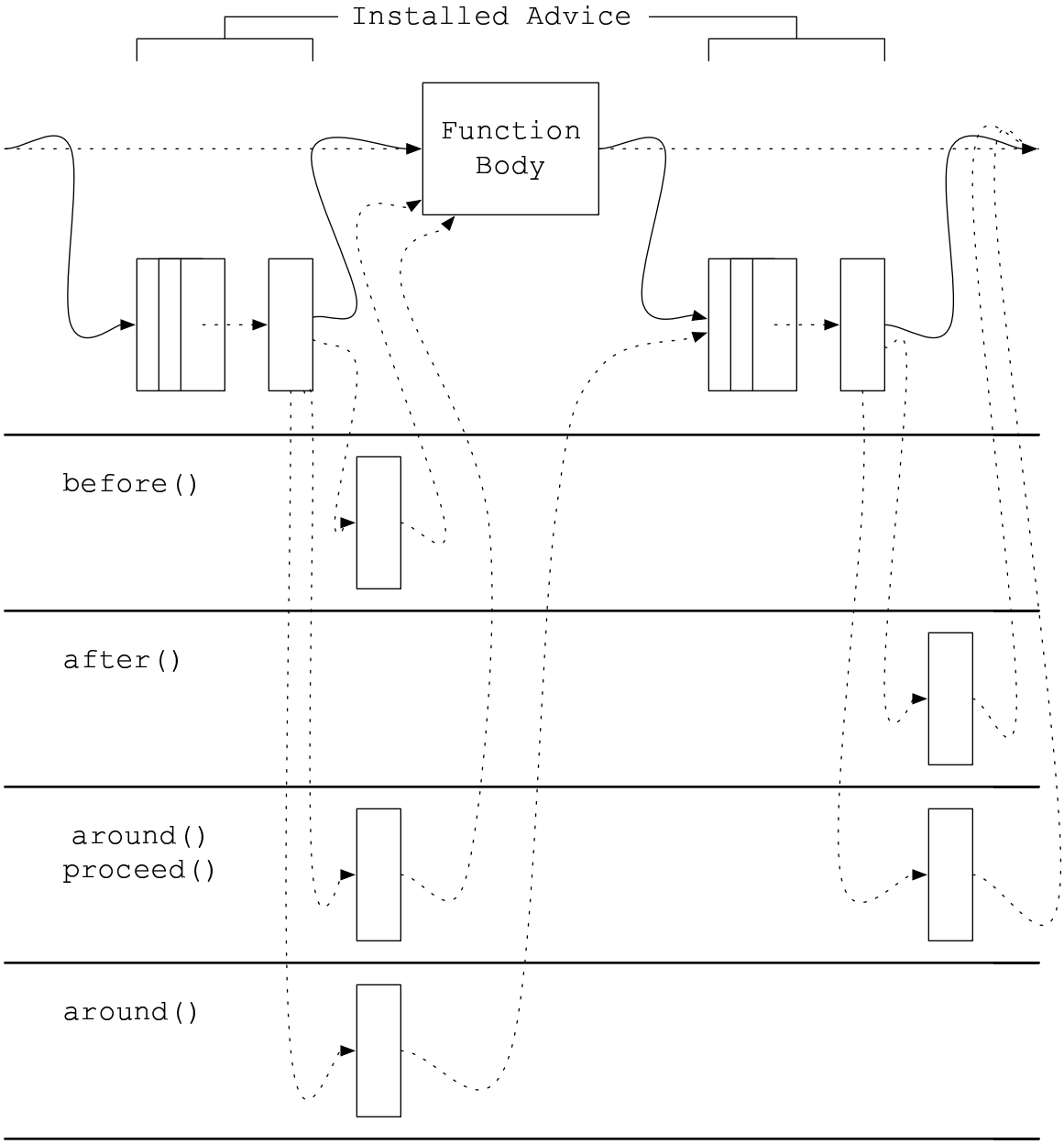


Figure 2.1: *Follow the dotted line: the effect of installing various kinds of advice on a function in MinAML. The upper section depicts the result of calling the function with previously installed advice. Each of the lower sections shows the result of adding indicated advice to the function.*

Since there is only one aspect in A, the `lookup` function returns it when evaluating $\texttt{incr}_\beta\langle\!\langle 3 \rangle\!\rangle$, and hence M reduces to

$$\texttt{incr}_\alpha\langle\!\langle (\lambda x.x^2 \cdot 3) + 1 \rangle\!\rangle$$

and then immediately to $\texttt{incr}_\alpha\langle\!\langle 10 \rangle\!\rangle$. The `lookup` function now returns the identity because there are no aspects labelled with $\texttt{incr}_\alpha$ in A. The whole term therefore simply becomes $(\lambda x.x) \cdot 10$ which finally reduces to 10 as expected.

Now consider the translate of the same term when $tm = \texttt{around}$:

$$\begin{array}{l}
\texttt{newlab incr}_\beta : \texttt{int in} \\
\texttt{newlab incr}_\alpha : \texttt{int in} \\
\texttt{let incr} = \lambda x.\texttt{incr}_\alpha\langle\!\langle \texttt{let } x = \texttt{incr}_\beta\langle\!\langle x \rangle\!\rangle \texttt{ in } x + 1 \rangle\!\rangle \texttt{ in} \\
\{\texttt{incr}_\beta.x \rightarrow \texttt{return } x^2 \texttt{ to incr}_\alpha\} \gg \texttt{incr} \cdot 3
\end{array}$$

Notice that the terms are identical except for the advice that's installed on the last line. In fact evaluating the term until we get to the base code as before yields an identical result save the aspect appearing in A. From this point however, the behaviour of the term changes radically: looking up the advice results in the input value 3 being applied to a very different function than before:

$$\texttt{incr}_\alpha\langle\!\langle (\lambda x.\texttt{return } x^2 \texttt{ to incr}_{\texttt{aft}} \cdot 3) + 1 \rangle\!\rangle$$

After making the substitution, the remaining term is $\texttt{incr}_\alpha\langle\!\langle (\texttt{return } 9 \texttt{ to incr}_\alpha) + 1 \rangle\!\rangle$. By the evaluation rule for `return`, the context $[-] + 1$ around the statement is "zapped", resulting in the term $\texttt{incr}_\alpha\langle\!\langle 9 \rangle\!\rangle$ which evaluates to 9 because (again) there are no aspects in A labelled with $\texttt{incr}_\alpha$. The net result is that the actual body of the original function

is circumvented because of the invocation of `return`.

A pleasing consequence of Walker et al.'s translation approach is that a safety result can straightforwardly be inferred from the type safety of the core calculus λ_{L}. One only needs to prove the following straightforward preservation lemma:

Lemma 2.2.3. *For any MinAML term M, if $\vdash M : \tau$ then $\vdash \mathcal{T}[\![M]\!] : \tau$.*

Combining this lemma with the type safety result for λ_{L} then yields

Theorem 2.2.4 (MinAML Safety)**.** *For any closed MinAML term M, either $\mathcal{T}[\![M]\!]$ diverges or there is a value or returning configuration C such that $(\bot, \bot, \mathcal{T}[\![M]\!]) \longrightarrow^* C$*

A key result of this thesis is that the "additive" fragment of λ_{L} – which doesn't include `return` – can be fully abstractly translated into a functional language with higher-order locally declared storage. Consequently, the fully abstract denotational games model for the latter is inherited by this fragment, implicitly constructing an adequate game semantics for the fragment of MinAML which excludes non-proceeding around advice.

2.3 Chapter and Related Work Summary

This chapter has introduced the key concepts relating to compositional translation as a means of specifying semantics of programming languages. It presented a series of criteria that can be used to quantify the level of correspondence that a given translation achieves between its source and target:

Evaluation Equivalence (E) The evaluation relations of the two languages emulate one another under the translation, i.e. the translation preserves and reflects them.

Termination Equivalence (T) The termination relations of the languages are preserved and reflected by the translation. Implied by evaluation equivalence.

Definability (D) Every term of the target language has an emulator in the source modulo observational equivalence.

Substitution (S) Capture avoiding substitutions are preserved and reflected by the translation, which also commutes with substitution.

Compositionality (C) Context plugging is preserved and reflected by the translation, which also commutes with the operation. For the translations considered in the following, this is implied by the substitution property.

Adequacy (A) The translation reflects observational equivalence. This is the usual notion of correctness in compilation semantics, and is implied by termination equivalence and compositionality, although only the forward direction of C.1 is required.

Full Abstraction (F) The translation preserves and reflects observational equivalence. Implied by termination equivalence, adequacy, definability, and compositionality.

The proof outline presented in the previous sections closely mirrors the one often used in proofs of the full abstraction of game semantics models [6, 4, 7, 26, 36]. In these cases, termination in the target language corresponds to the non-bottom denotation of a term in the model. Some of the terminology varies when speaking about game semantics rather than languages. Primarily, the latter proofs often proceed by showing preservation of equivalence (often called *soundness* in game semantics papers [8]), and reflection of termination (referred to as *computational adequacy* [3]). Invariably, the key result in all these proofs is the proof of definability. In game semantics, this is often done by what is known as a factorisation theorem, which expresses the elements of the model under consideration in terms of elements of a previous one.

This general proof strategy has also been used when considering operational translations, as in this thesis. Notably, McCusker [45] presents a fully abstract model of the lazy λ-calculus by translation into FPC, whose model is then inherited by the source language. McCusker's proof strategy very closely follows our own, and in fact was a major influence of the work which follows. Confusingly, his paper refers to the adequacy property as "soundness", perhaps alluding to the roots of game semantics in the logic community. More recently, [54] generalises many of the above notions, presenting a framework in which fully abstract translations of λ calculi with non-deterministic features can be defined.

The second half of the chapter formally defined the MinAML language, the object of study in later chapters, as it was presented in [59]: by translating it into a core calculus of aspects. In this case, translation is used as a means of *defining* the semantics of the language itself, which yields adequacy inherently.

Two important reasons for the choice of MinAML/λ_L as a starting point are its extensibility, and its well-behaved typed system. The fact that it extends the simply-typed λ-calculus closely mirrors many of the languages for which game semantics models already exist [3, 34]. Many of the options surveyed in the previous chapter were dismissed due to excess minimality, complexity, or lack of formality.

For example, μABC's [10] lack of a semantically meaningful base language against which advice is matched makes it difficult to illuminate how such a language behaves in the presence of advice. Second, in contrast to languages in the bulk of modern programming language semantics research, it is untyped, making it difficult to put into any meaningful context. Third, even finding a reasonable type system for it has proved to be a significant challenge [25], indicating that it's a language that is currently too unstructured to use as a basis for further research.

By contrast, while $\varsigma_{\mathsf{A}}(\mathrm{M})$ [14] would clearly be useful to a researcher who wants to explore various join point and advice models in a unified framework, it's complexity makes it difficult to use a basis for denotational work. It is preferable to start from a small typed core, and include more and more features incrementally. PolyAML [16] (which was succeeded by AspectML [17]) was eliminated for similar reasons. Aspectual CAML [56] is also fairly complex, but of even greater detriment is the fact that it was published as an implementation, and has no formally defined type system.

Finally, Aldrich's Open Modules [9] wasn't considered because it was only published after this project began. However, the semantics presented in chapter 5 has some interesting connections with Aldrich's work which are discussed therein.

Therefore, despite the fact that MinAML is missing many features that would be considered part of practical AOPLs, it was deemed a suitable starting point for denotational research. For instance, MinAML aspects can only advise pointcuts with identical types, there is also no module system, and hence no concept of type extension. However, due to it's basis in the λ-calculus, and its desirable theoretical properties, this was deemed an acceptable compromise.

Part II

Labelled Calculi

Chapter 3

Additive Advice with State

This chapter and the next aim to express Walker et al.'s semantics of MinAML using traditional notions of state and control. This is accomplished by encoding the core aspect calculus λ_{L} into a functional language with imperative features. As a first approximation, consider the fragment of λ_{L} not including the `return` primitive, and call the resulting language λ_{A}. The subscript indicates that the language fragment can encode only the "additive" fragment of MinAML using the translation from [59], that is, just those aspects which only add to the base code. The key results in this chapter establish the following:

- A translation from λ_{A} to a functional language with general references. This translation is evaluation equivalent and compositional, and thus reflects observational equivalence (properties E, C, and A from the previous chapter), and as such is an emulator of λ_{A} and thus MinAML.

- The translation also satisfies definability (property D), but is curiously not fully abstract.

- If λ_{A} is modified to take a standard (in the denotational semantics community)

"object-oriented" view of labels (and in fact *names* in general), then the translation becomes fully abstract (property F).

- The existence of this fully abstract translation means that the fully abstract games model of higher order references [3] is also a fully abstract for the modified λ_{A}, and thus adequate for MinAML.

The last section examines the properties satisfied by translations between the languages with and without the object-oriented view of names, so that the effects of taking this view can be understood. To begin, the syntax and semantics of the translation's target language are defined in the next section.

3.1 General References

The language λ^{*}_{R} is defined by extending λ with higher-order store in the style of ML-references (the "*" superscript in the language name indicates the presence of "bad variables", the popular idiom for the object-orientation mentioned above). Given a countable set of *references* whose elements are ranged over by ρ, the expressions of the language are defined by extending the expression grammar of λ with the following productions:

$$M ::= \cdots \mid \rho \mid \texttt{newref}[\tau](M) \mid \rho := N \mid !\rho$$

The typing rules and operational semantics of these constructs are defined in table 3.1. A term is typed with respect to a map R from references to types.

Rather than introducing a distinguished type for references, the type $\texttt{ref}[\tau]$ is viewed as a product of a "read method" and a "write method", *à la* Reynolds [51]. The write method assigns a value to the location and has type $\tau \to \texttt{unit}$; the read method retrieves

the value currently stored there and has type $\mathtt{unit} \to \tau$:

$$\mathtt{ref}[\tau] \triangleq (\tau \to \mathtt{unit}) \times (\mathtt{unit} \to \tau)$$

A consequence of the identification is the presence of the aforementioned bad variables. Even though every term of type $\mathtt{ref}[\tau]$ can be assigned to and dereferenced by calling the appropriate function, not all terms of the type behave as *bona fide* references (for example, reads need not be causally related to writes). Many *fully abstract* models of Algol-like languages interpret reference types as products of read and write methods. Consequently bad variables live in them. Interested readers may wish to consult [3, §2.4] for a discussion on whether this should be regarded as a defect. Essentially, their presence amounts to the inability of the language to define equality of references, i.e. a function of type $(\mathtt{ref}[\tau] \times \mathtt{ref}[\tau]) \to \mathtt{bool}$, which returns $\mathtt{tt}$ if and only if its inputs are references to the same location in the store. This result is quite intuitive, as it would make no sense to ask whether bad references denote the same location when they don't denote a proper location at all.

The operational semantics of λ^*_{R} is defined as an evaluation relation between configurations (R, S, M) comprising a map R from references to types, a store S mapping references to values, and a term M to be evaluated. A well typed configuration is defined similarly to the corresponding definition in λ_{L}:

Definition 3.1.1 (λ^*_{R} Well Typed Configuration)**.** A configuration (R, S, M) is well typed, denoted $\vdash (R, S, M)$, if for some Γ and τ the following conditions hold:

1. $\Gamma \vdash_R M : \tau$ is provable
2. $\Gamma \vdash_R N : R(\rho)$ is provable for each $\rho \in \mathtt{dom}(S)$ where $S(\rho) = N$.

TYPE SYSTEM

$$\frac{R(\rho) = \tau}{\Gamma \vdash_R \rho : \mathtt{ref}[\tau]} \text{ (TP REF)} \qquad \frac{\Gamma \vdash_R M : \tau}{\Gamma \vdash_R \mathtt{newref}[\tau](M) : \mathtt{ref}[\tau]} \text{ (TP NEWREF)}$$

$$\frac{\Gamma \vdash_R \rho : \mathtt{ref}[\tau]}{\Gamma \vdash_R !\rho : \tau} \text{ (TP DEREF)} \qquad \frac{\Gamma \vdash_R \rho : \mathtt{ref}[\tau] \qquad \Gamma \vdash_R M : \tau}{\Gamma \vdash_R \rho := M : \mathtt{unit}} \text{ (TP ASSIGN)}$$

OPERATIONAL SEMANTICS

$$\frac{M \Downarrow (R, S, V)}{\mathtt{newref}[\tau](M) \Downarrow (R[\rho \mapsto \tau], S[\rho \mapsto V], \rho)} \; [\rho \notin \mathtt{dom}(R)] \quad (\Downarrow \text{ NEWREF})$$

$$\frac{M \Downarrow (R, S, V)}{\rho := M \Downarrow (R, S[\rho \mapsto V], \mathtt{skip})} \; (\Downarrow \text{ ASSIGN}) \qquad \frac{}{(R, S, !\rho) \Downarrow S(\rho)} \; (\Downarrow \text{ DEREF})$$

$$\frac{M \Downarrow \rho}{\pi_1(M) \Downarrow \lambda x : \tau.[\rho := x]} \; (\Downarrow \text{ REFPROJ}_1) \qquad \frac{M \Downarrow \rho}{\pi_2(M) \Downarrow \lambda d : \mathtt{unit}.[!\rho]} \; (\Downarrow \text{ REFPROJ}_2)$$

Table 3.1: *Typing rules and operational semantics of* λ^*_{R}

The grammar of values extends those of λ with reference names:

$$V ::= \cdots \mid \rho$$

By definition a reference is a pair, so it must be possible to project it to retrieve its components; the ($\Downarrow$REFPROJ$_i$) rules provide this capability. This means that the assignment

and dereferencing constructs over arbitrary terms can simply be encoded as projections on their subterms:

$$M := N \triangleq \pi_1(M) \cdot N$$
$$!M \triangleq \pi_2(M) \cdot \mathtt{skip}$$

The language λ_{R} (without the "*" superscript) is defined by regarding the reference type $\mathtt{ref}[\tau]$ and the above shortcuts (and their derived rules) as primitive, adding the usual rules for assigning and dereferencing terms of reference type, and removing those for projections over reference names.

Observational equivalence of λ^*_{R} terms is defined as for λ_{A}, as are the shorthands for the `let` construct, sequential composition, and the identity function. The `new ... in ...` expression for references is defined as

$$\mathtt{newref}\ x : \tau = M\ \mathtt{in}\ N \triangleq (\lambda x : \mathtt{ref}[\tau].N) \cdot \mathtt{newref}[\tau](M)$$

The language λ^*_{R} is identical to the language of Abramsky et al. [3] except for the initialisation of newly created references, the absence of a distinguished reference type, and hence also the absence of an explicit "bad variable" constructor `mkvar`. These differences have no semantic consequences. In fact, the game model of the language of Abramsky et al. is fully abstract for λ^*_{R}.

Remark 3.1.1. In the language of Abramsky et al., for each type τ, there is a *primitive* reference type $\mathtt{ref}[\tau]$; in addition there is a "bad variable" constructor that takes functions $\mathtt{write} : \tau \to \mathtt{unit}$ and $\mathtt{read} : \mathtt{unit} \to \tau$ and casts them as a reference $\mathtt{mkvar}\,\langle \mathtt{write}, \mathtt{read}\rangle : \mathtt{ref}[\tau]$. Crucially the reference type $\mathtt{ref}[\tau]$ is isomorphic to the product type $(\tau \to \mathtt{unit}) \times (\mathtt{unit} \to \tau)$

$$\mathtt{ref}[\tau] \underset{\mathtt{mkvar}}{\overset{\mathscr{P}(-)}{\rightleftarrows}} (\tau \to \mathtt{unit}) \times (\mathtt{unit} \to \tau)$$

The isomorphism is witnessed in one direction by the bad variable constructor $\mathtt{mkvar}$; in the other direction, a term M of reference type $\mathtt{ref}[\tau]$ can be transformed to a pair of type $(\tau \to \mathtt{unit}) \times (\mathtt{unit} \to \tau)$ as follows:

$$\mathscr{P}(M) \triangleq \mathtt{let}\ x = M\ \mathtt{in}\ \langle \lambda y : \tau.[x := y]\ ,\ \lambda d : \mathtt{unit}.[!x] \rangle$$

The two transformations are inverse of each other, modulo observational equivalence; that is to say, the equations

$$\begin{aligned} \mathtt{mkvar}\ \mathscr{P}(M) &\simeq M \quad : \quad \mathtt{ref}[\tau] \\ \mathscr{P}(\mathtt{mkvar}\ \langle f, g \rangle) &\simeq \langle f, g \rangle \quad : \quad (\tau \to \mathtt{unit}) \times (\mathtt{unit} \to \tau). \end{aligned}$$

hold. It follows that the types $\mathtt{ref}[\tau]$ and $(\tau \to \mathtt{unit}) \times (\mathtt{unit} \to \tau)$ – which are distinct syntactic objects – must have isomorphic denotations in every fully abstract model. In the game model in [3] the types have the same denotations. The λ^*_{R} calculus achieves the same effect by explicitly identifying the reference type with the pair.

3.2 Translating Aspects to References

This section introduces a compositional translation $[\![-]\!]_{\mathsf{A}}$ from λ_{A} into λ^*_{R}; see Table 3.2 for the key clauses of the definition. The main intuition is to translate a λ_{A} label of type τ into a λ^*_{R} reference location of type $[\![\tau \to \tau]\!]_{\mathsf{A}}$, and to translate an aspect into a pair comprising the translates of its pointcut (a label) and advice (a function). Installing an aspect then corresponds to composing the advice with the current content of the location, and invoking an aspect simply dereferences it and applies the result to the value of the underlying term, which corresponds to the term labelled in the source language. The translation of the types of labels and aspects appear at the top of table 3.2. The

Types

$$\begin{array}{rcl} [\![\, \mathtt{lab}[\tau] \,]\!] & \triangleq & \mathtt{ref}[\, [\![\tau \to \tau]\!] \,] \\ [\![\, \mathtt{asp}[\tau] \,]\!] & \triangleq & [\![\, \mathtt{lab}[\tau] \,]\!] \times [\![\tau \to \tau]\!] \end{array}$$

Terms

$$\begin{array}{rcl} [\![\, \alpha \,]\!] & \triangleq & \mathscr{L}_{\mathsf{A}}(\rho_\alpha) \\ [\![\, \mathtt{newlab}[\tau] \,]\!] & \triangleq & \mathtt{newref}\ x : [\![\tau \to \tau]\!] = (\lambda y.y)\ \mathtt{in}\ \mathscr{L}_{\mathsf{A}}(x) \\ [\![\, \{M.x \to N\} \,]\!] & \triangleq & \langle [\![M]\!], \lambda x.[\![N]\!] \rangle \\ [\![\, M\langle\!\langle N \rangle\!\rangle \,]\!] & \triangleq & ![\![M]\!] \cdot [\![N]\!] \\ [\![\, M \gg N \,]\!] & \triangleq & \mathtt{let\ a} = [\![M]\!]\ \mathtt{in}\ \pi_1(\mathtt{a}) := \pi_2(\mathtt{a}); [\![N]\!] \end{array}$$

Environment

$$\begin{array}{rcl} [\![\, L[\alpha \mapsto \tau] \,]\!] & \triangleq & [\![L]\!][\rho_\alpha \mapsto [\![\tau \to \tau]\!]] \\ [\![\, L[\alpha \mapsto \tau], A \,]\!] & \triangleq & [\![L, A]\!][\rho_\alpha \mapsto [\![\mathtt{lookup}\langle L, A\rangle(\alpha)]\!]] \end{array}$$

Configuration

$$[\![(L, A, M)]\!] \triangleq ([\![L]\!], [\![L, A]\!], [\![M]\!])$$

Table 3.2: *Translation* $[\![-]\!]_{\mathsf{A}}$ *from* λ_{A} *to* λ^*_{R}. *The subscript* A *is omitted from the translation brackets for clarity.*

translation acts compositionally on the remaining types of λ_{A}: it is the identity map on ground types, and the translate of a product type is the product of the translates of its components, similarly for a function type.

For the translation of the terms of λ_{A}, assume a bijection between label names and references names, and write ρ_α as the image of α under the bijection. The semantics stores the composite of all the advice associated to a label α in the corresponding reference location. Since the only way to access advice in λ_{A} is by accessing the entire

composite, it can be composed statically without losing any information. This is a direct consequence of the fact that the pointcuts are in-scope: because these don't allow pointcuts to match functions that are not yet defined (using wildcards for example), every join point at which the advice must execute is accessible at installation time. The translation uses a shorthand

$$\mathscr{L}_{\mathsf{A}}(\mathtt{r}) \triangleq \langle\, \lambda \mathtt{New} : \tau \to \tau.[\mathtt{let\ Old} = !\mathtt{r\ in\ r} := \mathtt{New} \circ \mathtt{Old}]\ ,\ \lambda \mathtt{d} : \mathtt{unit}.[\lambda \mathtt{x}.!\mathtt{r} \cdot \mathtt{x}] \,\rangle$$

This term takes an input of type $\mathtt{ref}[\tau \to \tau]$, and is at the heart of the translation; it defines a bad variable of type $\tau \to \tau$ whose assignment function composes its input with the current contents rather than overwriting it. This allows aspect installation to be modelled by assignment, and aspect invocation by dereferencing. Note the subtle use of `let` bindings and η expansion in the two components: when assigning, the `let` statement assures that the new advice is composed to the the advice present *at installation time*, and when dereferencing the η-expansion makes sure that the advice is only looked up if a value (corresponding to the value of the underlying labelled term) is applied to it.

Label creation, $\mathtt{newlab}[\tau]$, translates to the creation of a new reference of the above kind initialised to the appropriately typed identity function. Since there is exactly one `newref` statement in $[\![M]\!]_{\mathsf{A}}$ for each `newlab` statement in M, it is assumed that if the new label chosen by evaluating M is α, the corresponding `newref` in $[\![M]\!]_{\mathsf{A}}$ generates ρ_α.

The bottom two sections of the table extend the translation to act on configurations of λ_{A}, producing configurations of λ^*_{R}. The environment is translated by iterating through the map L, producing an R map by assigning the type $[\![\tau \to \tau]\!]_{\mathsf{A}}$ to the reference corresponding to each label of type τ. A second pass produces a store by looking up the advice for each label and assigning its translate to the appropriate cell. Configurations

are then simply translated pointwise. The translation satisfies the following properties:

Proposition 3.2.1.

(a) If $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash_L M : \tau$ *is valid in* λ_A *then* $x_1 : [\![\tau_1]\!]_A, \ldots, x_n : [\![\tau_n]\!]_A \vdash_{[\![L]\!]_A} [\![M]\!]_A : [\![\tau]\!]_A$ *is valid in* λ_R^*.

(b) If (L, A, V) *is a* λ_A *value configuration, then* $[\![(L, A, V)]\!]_A$ *is a* λ_R^* *value configuration.*

(c) If (L, A, M) *is well typed in* λ_A *then* $[\![(L, A, M)]\!]_A$ *is well typed in* λ_R^*.

Proof. (a) is proved by a straightforward induction on the structure of M. (b) follows immediately from the definition of the translation over the core calculus λ and the fact that $\mathscr{L}_A(V)$ is a value for any value V. (c) follows from the fact that every label α in the environment is translated into ρ_α, and each label in M is translated into $\mathscr{L}_A(\rho_\alpha)$, which itself only contains the reference ρ_α. □

3.3 Adequacy and Definability

How faithful is the compositional translation from λ_A into λ_R^*? This section examines the extent to which behavioural properties of user terms of λ_A are preserved (and reflected) by the translation. As it turns out, the translation is adequate (property A from chapter 2), but fails to be fully abstract; a simple counterexample explains why. This is rectified in the next section, where a fully abstract translation is achieved by a slight modification to λ_A.

Preliminary lemmas

Property S helps reason about typing judgements involving substitution. The first two lemmas prove that the translation satisfies some of its conditions. In the following, let

M and V be well-typed user terms of λ_{A} where V is a value.

Lemma 3.3.1 ($[\![-]\!]_{\mathsf{A}}$ Property S.1 $\Rightarrow$)**.** *If* $\vdash M[V/x]$ *holds in* λ_{A}*, then* $\vdash [\![M]\!]_{\mathsf{A}}[[\![V]\!]_{\mathsf{A}}/x]$ *holds in* λ^*_{R}.

Proof. By assumption there are Γ, σ, τ such that

$$\Gamma, x : \sigma \vdash M : \tau \quad \text{and} \quad \Gamma \vdash V : \sigma$$

are valid. By Proposition 3.2.1, we have

$$[\![\Gamma]\!]_{\mathsf{A}}, x : [\![\sigma]\!]_{\mathsf{A}} \vdash [\![M]\!]_{\mathsf{A}} : [\![\tau]\!]_{\mathsf{A}} \quad \text{and} \quad [\![\Gamma]\!]_{\mathsf{A}} \vdash [\![V]\!]_{\mathsf{A}} : [\![\sigma]\!]_{\mathsf{A}}$$

are valid in λ^*_{R}. Then, since $[\![V]\!]_{\mathsf{A}}$ is a value, we know that $[\![M]\!]_{\mathsf{A}}[[\![V]\!]_{\mathsf{A}}/x]$ is a well-typed term of λ^*_{R}. □

The converse of the lemma does not hold i.e. there are terms M and V such that $[\![M]\!]_{\mathsf{A}}[[\![V]\!]_{\mathsf{A}}/x]$ is well-typed in λ^*_{R}, but $M[V/x]$ is not well-typed in λ_{A}. For instance, let M be a variable x of type $\mathtt{lab}[\tau]$ and

$$V \;\equiv\; \langle \lambda y : \tau \to \tau.\mathtt{skip}\ ,\ \lambda y : \mathtt{unit}.\mathtt{id}[\tau] \rangle \;:\; \mathtt{ref}[\tau \to \tau].$$

This is precisely the defect that is later exploited to show the failure of full abstraction. The remedy proposed in the following section is designed to ensure that the translation preserves types, making the proof of property S.1 trivial. The next lemma shows that property S.2 holds for $[\![-]\!]_{\mathsf{A}}$:

Lemma 3.3.2 ($[\![-]\!]_{\mathsf{A}}$ Property S.2)**.** *If* $\vdash M[V/x]$ *holds in* λ_{A} *then* $[\![M[V/x]]\!]_{\mathsf{A}} = [\![M]\!]_{\mathsf{A}}[[\![V]\!]_{\mathsf{A}}/x]$ *holds in* λ^*_{R}.

Proof. By the result of the previous lemma, we know that both sides of the desired equality are well-typed terms of λ^*_{R}. The equality is proved by an easy induction on the derivation of $\Gamma, x : \sigma \vdash M : \tau$, whose existence is implied by $\vdash M[V/x]$. The proof requires a standard weakening lemma in the case of application. □

Evaluation equivalence

Recall from chapter 2 that the next step to proving adequacy is to establish that the evaluation of *user terms* is preserved, i.e. the forward direction of property E. A stronger induction hypothesis is needed to push the argument through; the result is therefore proved for all possible configurations, and with respect to syntactic equality (up to renaming of bound variables) rather than observational equivalence.

Proposition 3.3.3 ($[\![-]\!]_{\mathsf{A}}$ Property E $\Rightarrow$)**.** *for every well-typed* λ_{A} *configuration* C*, if* $C \longrightarrow^* C'$ *for some value configuration* C'*, then* $[\![C]\!]_{\mathsf{A}} \Downarrow [\![C']\!]_{\mathsf{A}}$

Proof. The clearest way to present this proof is in two stages. First, the small-step semantics $\longrightarrow$ of λ_{A} are re-cast in big step style, and proved equivalent by induction in the standard way. The result is then proved by an induction on the derivation of a λ_{A} evaluation. The complete proof appears in appendix B.1.1. □

The next proposition proves the converse of the preceding result, thus showing that that the translation from λ_{A} to λ^*_{R} *preserves and reflects evaluation* in the sense of property E. It follows that the translation also *preserves and reflects termination* of configurations.

Proposition 3.3.4 ($[\![-]\!]_{\mathsf{A}}$ Property E $\Leftarrow$)**.** *For any well-typed* λ_{A} *configuration* C*, if* $[\![C]\!]_{\mathsf{A}} \Downarrow C'$ *for some* λ^*_{R} *value configuration* C'*, then* $C \longrightarrow^* C''$ *such that* $C' = [\![C'']\!]_{\mathsf{A}}$

Proof. By induction on the derivation of $[\![C]\!]_{\mathsf{A}} \Downarrow C'$. Since only translates of λ_{A} configurations need to be considered, the cases are enumerated by the structure of the term component of C. Care must be taken to assure that the induction hypothesis is only applied to sub-derivations which satisfy this property, and the substitution property is integral in this regard. See appendix B.1.2 for details. □

Example 3.3.1 (Commutativity)**.** The propositions above are equivalent to a statement that the translation $[\![-]\!]_{\mathsf{A}}$ commutes with the operational semantics of the languages. In other words, it makes no difference whether a configuration is translated first or evaluated first, the result is the same λ^*_{R} configuration. As an example, consider the λ_{A} term

$$\begin{aligned} M \equiv\ & \texttt{newlab}\, l_1 : \texttt{bool}\ \texttt{in} \\ & \texttt{newlab}\, l_2 : \texttt{bool} \to \texttt{bool}\ \texttt{in} \\ & \{l_1.x \to \neg x\} \gg \\ & \{l_2.f \to \lambda x.\neg f(x)\} \gg \\ & \{l_1.x \to \neg x \vee x\} \gg \\ & \lambda x.l_2\langle\!\langle \lambda z.l_1\langle\!\langle z \rangle\!\rangle \rangle\!\rangle \end{aligned}$$

which uses logical operators that are all easily encodable with conditionals, so the translation behaves as the identity on them. The term isn't really significant in any way, but uses many of λ_{A}'s salient features, and in this regard is quite illustrative. Evaluating this term using the λ_{A} evaluation rules results in the configuration (L, A, U) where

$$\begin{aligned} L &\equiv \{\alpha_1 \mapsto \texttt{bool}, \alpha_2 \mapsto \texttt{bool} \to \texttt{bool}\} \\ A &\equiv [\{\alpha_1.x \to \neg x\}, \{\alpha_2.f \to \lambda x.\neg f(x)\}, \{\alpha_1.x \to \neg x \vee x\}] \\ U &\equiv \lambda x.\alpha_1\langle\!\langle \lambda z.\alpha_2\langle\!\langle z \rangle\!\rangle \rangle\!\rangle \end{aligned}$$

Now translate M into λ^*_{R} to get $[\![M]\!]_{\mathsf{A}}$ (the variables l_i have been renamed to r_i to high-

light the fact that they are now *references*):

$$\texttt{let}\ r_1 : \texttt{ref}[\texttt{bool} \to \texttt{bool}] = [\texttt{newref}\ x = \lambda y.y\ \texttt{in}\ \mathscr{L}_{\mathsf{A}}(x)]\ \texttt{in}$$
$$\texttt{let}\ r_2 : \texttt{ref}[(\texttt{bool} \to \texttt{bool}) \to (\texttt{bool} \to \texttt{bool})] = [\texttt{newref}\ x = \lambda y.y\ \texttt{in}\ \mathscr{L}_{\mathsf{A}}(x)]\ \texttt{in}$$
$$r_1 := (\lambda x.\neg x);$$
$$r_2 := (\lambda f.\lambda x.\neg f(x));$$
$$r_1 := (\lambda x.\neg x \vee x);$$
$$\lambda x.!r_2 \cdot (\lambda z.!r_1 \cdot z)$$

The assignment statements in the above term assigns to bad variables of the form $\mathscr{L}_{\mathsf{A}}(\rho_i)$, so evaluating this term up to the last element in the sequential composition yields configuration (R, S, V) where

$$\begin{aligned} R &\equiv \{\rho_1 \mapsto \texttt{bool} \to \texttt{bool}, \rho_2 \mapsto (\texttt{bool} \to \texttt{bool}) \to (\texttt{bool} \to \texttt{bool})\} \\ S &\equiv \{\rho_1 \mapsto (\lambda x.\neg x \vee x) \circ (\lambda x.\neg x) \circ (\texttt{id}[\texttt{bool}]), \\ &\qquad \rho_2 \mapsto (\lambda f.\lambda x.\neg f(x)) \circ (\texttt{id}[\texttt{bool} \to \texttt{bool}])\} \\ V &\equiv \lambda x.!\mathscr{L}_{\mathsf{A}}(\rho_2) \cdot (\lambda z.!\mathscr{L}_{\mathsf{A}}(\rho_1) \cdot z) \end{aligned}$$

Finally, verify that $[\![(L, A, U)]\!]_{\mathsf{A}} = (R, S, V)$, i.e. the correspondence between the two terms is exact (up to renaming the bound names).

Adequacy

Having shown that the translation over user terms satisfies property E (implied by propositions 3.3.3 and 3.3.4), and property S.2 and the forward direction of S.1 (lemmas 3.3.2 and 3.3.1), the proof from chapter 2 can be used to conclude that the translation is adequate:

Theorem 3.3.5 ($[\![-]\!]_{\mathsf{A}}$ Adequacy)**.** *If* $[\![M_1]\!]_{\mathsf{A}} \simeq [\![M_2]\!]_{\mathsf{A}}$ *then* $M_1 \simeq M_2$*, where* M_1 *and* M_2 *are*

well-typed open user terms of λ_{A}.

The theorem says that one can soundly reason about the observational equivalence of λ_{A}-terms by reasoning about the observational equivalence of their respective translates in λ^*_{R}. Since the translation is compositional and adequate, the game model of λ^*_{R} is inherited by λ_{A}.

Why full abstraction still fails

The translation from λ_{A} to λ^*_{R} is adequate, and in fact also satisfies the definability property ([53]); however, it is *not* fully abstract i.e. it does not preserve observational equivalence. This is somewhat surprising, especially in view of the definability result which indicates a bijection between the two languages. It suggests that the translation is not fully abstract for a subtle reason. Take the following λ_{A} user terms and context:

$$\begin{aligned} M_1 &= \lambda z : \mathtt{lab}[\tau].\{z.x \to x\} \gg \mathtt{skip} \\ M_2 &= \lambda z : \mathtt{lab}[\tau].\mathtt{skip} \\ \mathcal{C}[-] &= [-] \cdot \langle \lambda x.\Omega, \lambda x.\Omega \rangle \end{aligned}$$

where Ω denotes a divergent term. Note that $\mathcal{C}[-]$ (viewed as a term with a single free variable) paired with either of the M_i's, forms a witness to the failure of the converse of Lemma 3.3.1. Now consider the latter terms, the first takes a label as input and then installs the trivial identity advice to it before evaluating to `skip`, while the second simply evaluates to `skip` right away, so they are equivalent in λ_{A}. Now consider their respective translates in λ^*_{R}: $[\![M_1]\!]_{\mathsf{A}}$ assigns to the reference that is bound to z, whereas $[\![M_2]\!]_{\mathsf{A}}$ does not. Therefore, if z is bound to a bad reference whose components immediately diverge (say by plugging the M_i into $\mathcal{C}[-]$), the first term ($\mathcal{C}[[\![M_1]\!]_{\mathsf{A}}]$) diverges while the second ($\mathcal{C}[[\![M_2]\!]_{\mathsf{A}}]$) does not. Therefore the translates of the two terms are not observationally

equivalent in λ^*_{R}, violating full abstraction.

Proposition 3.3.6 ($[\![-]\!]_{\mathsf{A}}$ Non full-abstraction)**.** *There are* λ_{A} *user terms* M_1 *and* M_2 *such that* $M_1 \simeq M_2$ *but* $[\![M_1]\!]_{\mathsf{A}} \not\simeq [\![M_2]\!]_{\mathsf{A}}$.

Essentially, full abstraction breaks down because λ_{A}'s type structure is not preserved by the translation: the types $\mathtt{lab}[\tau]$ and $\mathtt{ref}[\tau \to \tau]$ are distinct in the source (one is a label and one is a pair), but collapsed together in the target. This fact is exploited in the above counter-example by forcing these types to "mix" when translated (note that $\mathcal{C}[-]$, while a valid λ_{A} context, is not correctly typed for M_1 and M_2). The next section alters λ_{A} to avoid this difficulty.

3.4 Definability and Full Abstraction

The failure of full abstraction can be traced to a fundamental mismatch between labels and their translates, which are pairs of read and write methods. The fix is to "object-orientate" the types of labels in λ_{A} into their component accessor methods, introducing "bad labels" – the analogue of bad variables – into the language.

Definition 3.4.1. The language λ^*_{A} is the exact analogue of λ^*_{R} for labels. It is defined by viewing the type of labels as a shorthand for the following:

$$\mathtt{lab}[\tau] \triangleq (\tau \to \tau) \to \mathtt{unit} \times \mathtt{unit} \to (\tau \to \tau)$$

The distinguished type of aspects is also removed. The reason for selecting this particular type identification follows an easy intuition: a label pairs an *installation* function – which takes a piece of advice and installs an associated aspect into the environment – with an *invocation* function – which returns the composite of all the advice associated with the label.

The expression syntax is modified to reflect this change: the aspect construct is removed, and aspect installation and invocation primitives over arbitrary terms are replaced with analogous constructs installing them on raw label names. The aspect installation operator $\gg$ is replaced with a standalone operator $\triangleright$ which takes a label of type τ and a piece of advice, i.e. a term of type $\tau \to \tau$, installs the advice on the label, and evaluates to `skip`. This particular change isn't really significant, as the two operators are mutually encodable, but makes the translation a little neater. Precisely, the syntax of λ^*_{A} extends that of λ as follows:

$$M ::= \cdots \mid \alpha \mid \texttt{newlab}[\tau] \mid \alpha \triangleright M \mid \alpha\langle\!\langle M \rangle\!\rangle$$

Primitives which apply to arbitrary terms can be recovered as in λ^*_{R} by projecting over labels:

$$\begin{aligned} M \triangleright N &\triangleq \pi_1(M) \cdot N \\ M\langle\!\langle N \rangle\!\rangle &\triangleq (\pi_2(M) \cdot \texttt{skip}) \cdot N \end{aligned}$$

The β rules and evaluation contexts for these constructs are updated as follows:

$$E ::= \cdots \mid \alpha \triangleright E \mid \alpha\langle\!\langle E \rangle\!\rangle$$

$$\begin{aligned} (L, A, \alpha \triangleright V) &\xrightarrow{\beta} (L, A :: \langle \alpha, V \rangle, \texttt{skip}) \\ (L, A, \pi_1(\alpha)) &\xrightarrow{\beta} (L, A, \lambda x.\alpha \triangleright x) \\ (L, A, \pi_2(\alpha)) &\xrightarrow{\beta} (L, A, \lambda d.\lambda x.\alpha\langle\!\langle x \rangle\!\rangle) \end{aligned}$$

The rules for `newlab`$[\tau]$ and invocation are unchanged. The primary consequence of these changes is that the new translation function $[\![-]\!]^*_{\mathsf{A}}$ is just the identity on types, and so, there is no further need to distinguish between a λ^*_{A} type τ and its translate $[\![\tau]\!]^*_{\mathsf{A}}$. The converse of Lemma 3.3.1 is therefore immediate, and stated without proof, establishing

property S (and consequently property C):

Lemma 3.4.1 ($[\![-]\!]^*_{\mathsf{A}}$ Property S $\Leftarrow$)**.** *Let M and V be user terms of λ^*_{A} where V is a value. If $\vdash [\![M]\!]^*_{\mathsf{A}}[[\![V]\!]^*_{\mathsf{A}}/x]$ holds in λ^*_{R} then $\vdash M[V/x]$ holds in λ^*_{A}.*

The translation rules for the modified expression constructs are as follows:

$$\begin{array}{rcl} [\![\ \alpha \triangleright M\ ]\!]^*_{\mathsf{A}} & \triangleq & \mathtt{let\ New} = [\![M]\!]^*_{\mathsf{A}}\ \mathtt{in} \\ & & \mathtt{let\ Old} = !\rho_\alpha\ \mathtt{in}\ \rho_\alpha := \mathtt{New} \circ \mathtt{Old} \\ \\ [\![\ \alpha\langle\!\langle M \rangle\!\rangle\ ]\!]^*_{\mathsf{A}} & \triangleq & \mathtt{let\ Base} = [\![M]\!]^*_{\mathsf{A}}\ \mathtt{in}\ !\rho_\alpha \cdot \mathtt{Base} \end{array}$$

As before, `let` statements are used to assure that the components of each operation are evaluated at the appropriate time. When installing, the advice must be evaluated first, then existing advice must be retrieved, then the two must be composed. Similarly when invoking advice, the base code must be evaluated before the advice is looked up. Notice that, rather than translating the label α and assigning, these rules "short-circuit" the additional pairing and assign/dereference the storage location corresponding to α directly. In addition, the usual label emulation term $\mathscr{L}_{\mathsf{A}}(V)$ has to be amended to the similar:

$$\begin{array}{rcl} \mathscr{L}_{\mathsf{A}}(\mathtt{r}) & = & \langle \mathtt{install}, \mathtt{invoke} \rangle \\ \mathtt{install} & = & \lambda \mathtt{x.let\ New} = \mathtt{x\ in\ let\ Old} = !\mathtt{r\ in\ r} := \mathtt{New} \circ \mathtt{Old} \\ \mathtt{invoke} & = & \lambda \mathtt{d}.\lambda \mathtt{x.let\ Base} = \mathtt{x\ in}\ !\mathtt{r} \cdot \mathtt{Base} \end{array}$$

This is identical to the previous definition except for an additional `let` abstraction of the input variable in the body of each function. The reason for these changes is that they allow the syntactic equality in the reflection of evaluation result to hold true for λ^*_{A}. To illustrate, consider the following example:

Example 3.4.1. (An Alternative Translation) Suppose that the translation of aspect in-

stallation were defined in a more natural way as

$$[\![\alpha \triangleright M]\!]^*_{\mathsf{A}} \quad \triangleq \quad [\![\alpha]\!]^*_{\mathsf{A}} := [\![M]\!]^*_{\mathsf{A}} \quad = \quad \mathscr{L}_{\mathsf{A}}(\rho_\alpha) := [\![M]\!]^*_{\mathsf{A}}$$

the intuition being to delegate the responsibility of aspect composition to the $\mathscr{L}_{\mathsf{A}}(-)$ operator as before. A problem arises when projections over labels are considered: the λ^*_{A} term $\pi_1(\alpha)$ evaluates to $\lambda x.\alpha \triangleright x$ which would translate to

$$\lambda x.[\mathscr{L}_{\mathsf{A}}(\rho_\alpha) := x] \quad \triangleq \quad \lambda x.[\pi_1(\ \mathscr{L}_{\mathsf{A}}(\rho_\alpha)\ ) \cdot x] \tag{3.1}$$

but $[\![\pi_1(\alpha)]\!]^*_{\mathsf{A}} = \pi_1([\![\alpha]\!]^*_{\mathsf{A}}) = \pi_1(\mathscr{L}_{\mathsf{A}}(\rho_\alpha))$ which in turn evaluates to

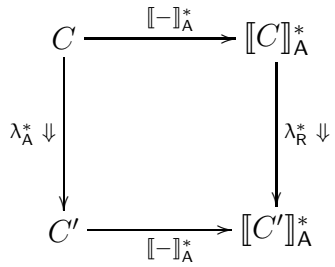
$$\lambda \mathtt{x}.\mathtt{let\ New} = \mathtt{x\ in\ let\ Old} = !\rho_\alpha\ \mathtt{in}\ \rho_\alpha := \mathtt{New} \circ \mathtt{Old}$$

This term – the evaluation of the translate – is not syntactically equal to (3.1) – the translate of the evaluation. The discrepancy occurs because evaluation of $\pi_i(\alpha)$ results in λ-abstractions that did not previously exist in the term's pre-evaluation syntax. These abstractions delay the evaluation of the body, and so the subterms aren't reduced to syntactic equality as before.

It should be noted that the adequacy theorem would surely still hold with such a translation. While the above translation is not evaluation equivalent, it is termination equivalent, and this is still sufficient for adequacy. Proving the latter was the route chosen in the published version of this work [53], but proving termination equivalence alone requires more machinery to be put in place. Furthermore, the updated translation has the benefit of the more general evaluation result. It is stated as a commutativity condition:

Proposition 3.4.2 ($[\![-]\!]^*_{\mathsf{A}}$ Property E)**.** *For any λ^*_{A} configuration C, the following diagram*

commutes:



Consequently, the proof of adequacy (theorem 3.3.5) remains unchanged.

Definability

Another way to interpret the adequacy theorem is to view it as a statement that there are at least as many observationally distinct terms in λ^*_{R} as there are in λ^*_{A}: an injectivity property. Definability (property D) is the converse of this statement: it says that there are no more distinct terms in λ^*_{R} than λ^*_{A}, or equivalently that the translation map is surjective with respect to observational equivalence:

Proposition 3.4.3 ($[\![-]\!]^*_{\mathsf{A}}$ Definability)**.** *For every open* λ^*_{R} *user term* M*, there exists a* λ^*_{A} *user term* $\mathsf{A}[\![M]\!]$ *such that* $[\![\mathsf{A}[\![M]\!]]\!]^*_{\mathsf{A}} \simeq M$*.*

This result is proved by induction on the structure of M. Since only user terms are considered, nearly all the cases follow immediately from the induction hypothesis, as they come from the underlying λ-calculus. The only exception is the case for $\texttt{newref}(M)$, which requires the construction of a λ^*_{A} term which translates to a λ^*_{R} term observationally equivalent to `newref`. Following [59], a reference of type τ is modelled by creating a label of type τ and using the ability to preserve local state to mimic a reference cell. Omitting type annotations, consider the λ^*_{A} term

$$\begin{array}{rcl} \mathsf{A}[\![\texttt{newref}(M)]\!] & \triangleq & \texttt{let Init} = \mathsf{A}[\![M]\!] \texttt{ in} \\ & & \texttt{let Cell} = \texttt{newlab in} \\ & & \langle\, \lambda\texttt{Val.Cell} \triangleright \lambda d.\texttt{Val}\ ,\ \lambda d.\texttt{Cell}\langle\!\langle \texttt{Init} \rangle\!\rangle \,\rangle \end{array}$$

The intuition is that `Cell` will be the label modelling the newly created reference cell, `Init` will be the initial value of the reference cell, and `Val` will be bound to the value assigned to the cell. Thus, to prove definability, it suffices to show that the translate of this term is equivalent to $\texttt{newref}(M)$.

Proving contextual equivalence of programs in languages with higher-order store is notoriously hard because of the quantification over all contexts. There are essentially two approaches one can take in order to attempt it. The operational approach usually defines an approximation to $\simeq$ which can be proved without such a quantification, and which implies observational equivalence [44, 50]. The drawback to this technique is that it produces false negatives because the approximation fails to account for the behaviour of local state. For instance, the logical relation of [50] would reject $[\![\mathsf{A}[\![\texttt{newref}(M)]\!]]\!]^*_{\mathsf{A}}$ as inequivalent to $\texttt{newref}(M)$ because the latter creates a storage cell of type τ and the former a storage cell of type $\tau \to \tau$. It cannot detect that a user of $[\![\mathsf{A}[\![\texttt{newref}(M)]\!]]\!]^*_{\mathsf{A}}$ can only access this cell by applying its contents to a dummy value (see below), and so essentially behaves as a cell of type τ.

The second approach is the denotational one, which involves comparing the interpretations of the two terms in some mathematical model. The most accurate model of λ^*_{R} is the game semantics model of [3]. Because it is fully abstract, it doesn't suffer from the approximation deficiency above. It's drawback is that computing a term's denotation is quite complicated and considered fairly inaccessible. However, as its the only available option, it is the one that must be used. Appendix C presents a summary of the game model $\mathcal{G}[\![-]\!]$ of λ^*_{R}, and then argues that $\mathcal{G}[\![\ [\![\mathsf{A}[\![\texttt{newref}(M)]\!]]\!]^*_{\mathsf{A}}\ ]\!]$ and $\mathcal{G}[\![\texttt{newref}(M)]\!]$ respond identically to any move played by the environment. This is achieved by a meticulous analysis of the strategies of the two terms in question.

To understand the reason for the equivalence intuitively, consider $[\![\mathsf{A}[\![\texttt{newref}(M)]\!]]\!]^*_{\mathsf{A}}$, the above term's translation into λ^*_{R}:

$$\begin{array}{l}\texttt{let Cell} = \texttt{newref}(\lambda y.y)\ \texttt{in}\\ \texttt{let Init} = [\![\mathsf{A}[\![M]\!]]\!]_{\mathsf{A}}\ \texttt{in}\ \langle \texttt{assign}, \texttt{deref}\rangle\end{array}$$

where

$$\begin{array}{rcl}\texttt{assign} & = & \lambda\texttt{Val.let}\\ & & \quad \texttt{Asp} = \langle\ \mathscr{L}_{\mathsf{A}}(\texttt{Cell})\ ,\ \lambda d.\texttt{Val}\ \rangle\\ & & \texttt{in}\ \pi_1(\texttt{Asp}) := \pi_2(\texttt{Asp}); \texttt{skip}\\ \\ \texttt{deref} & = & \lambda\texttt{d.!}\mathscr{L}_{\mathsf{A}}(\texttt{Cell}) \cdot \texttt{Init}\end{array}$$

The $\mathscr{L}_{\mathsf{A}}(-)$'s appear because of the shorthand used to encode the `let ... in` constructs. Recall that formally, aspect installation and labelling in λ^*_{A} and assignment and dereferencing in λ^*_{R} are actually nothing more than projections. They have been presented here using their usual sugared syntax to aid in reading the intuitive argument.

Check that this term indeed has a reference type, and observe that $\langle \mathscr{L}_{\mathsf{A}}(\texttt{Cell}), \lambda d.\texttt{Val}\rangle$ is already a value. Furthermore, sequential composition of a term of type `unit` with `skip` is equivalent to just the term. Using these facts, and the induction hypothesis that $M \simeq [\![\mathsf{A}[\![M]\!]]\!]^*_{\mathsf{A}}$, simplify $[\![\mathsf{A}[\![\texttt{newref}(M)]\!]]\!]^*_{\mathsf{A}}$ to

$$\begin{array}{l}\texttt{let Cell} = \texttt{newref}(\lambda y.y)\ \texttt{in}\\ \texttt{let Init} = M\ \texttt{in}\\ \langle\ \lambda\texttt{Val.}\mathscr{L}_{\mathsf{A}}(\texttt{Cell}) := (\lambda\texttt{d.Val})\ ,\ \lambda\texttt{d.!}\mathscr{L}_{\mathsf{A}}(\texttt{Cell}) \cdot \texttt{Init}\ \rangle\end{array}$$

The only way to store any other value into the location created here is by applying a value to the first component of the reference pair. By the definition of $\mathscr{L}_{\mathsf{A}}(-)$, the only thing that can ever subsequently be assigned to the location is a constant function composed with its current contents. The new location will therefore only ever contain a constant function. The following observations complete the argument:

- The term has the same termination behaviour as $\texttt{newref}(M)$ because of the first `let`, i.e. it converges if and only if M converges.

- Suppose the new reference location created by evaluating this term is ρ. There are three possible imperative actions one can take on this location:

    1. If this is immediately dereferenced, then the operational semantics evaluate $(!\rho)\cdot\texttt{Init}$, which (since ρ contains the identity function in this case) evaluates to `Init`: the result of evaluating M. So an immediately accessed reference returns its initial value as expected.

    2. Any attempt to assign a value V will compose the *constant function* returning V to the contents of ρ.

    3. Any subsequent dereferencing results in the application of the initial value `Init` to the contents of ρ, but since each function in the composition !ρ ignores its input, this application immediately returns the constant value of the last function in the sequence (i.e. the last value assigned).

Projecting the reference cell returns the function which was shown above to correspond exactly to assignment and dereferencing to the cell. Therefore these points exactly describe the observational behaviour of $\texttt{newref}(M)$. Furthermore, because of the additional substitution property, the reasoning of chapter 2 can be used to conclude full abstraction:

Theorem 3.4.4 ($[\![-]\!]^*_{\mathsf{A}}$ Full abstraction)**.** $M_1 \simeq M_2$ *if and only if* $[\![M_1]\!]^*_{\mathsf{A}} \simeq [\![M_2]\!]^*_{\mathsf{A}}$ *for any well-typed* λ^*_{A} *user terms* M_1 *and* M_2*.*

A pleasing consequence of the theorem is that the fully abstract semantics of the target language λ^*_{R} (used to prove definability) is at once inherited by the source language λ^*_{A}. Since observational equivalence of λ^*_{A} coincides with the equational theory of the fully abstract model, one may reason about the aspect language using the games model of [3].

3.5 Completing the Picture with Good Variables

Having considered compositional translations from λ_A to λ^*_R (section 3.3), and from λ^*_A to λ^*_R (section 3.4), is is natural to ask if there is a reasonable translation from these aspect calculi to λ_R (with only good variables). Furthermore, it is illustrative to see what effect the presence or absence of bad names in the source or target might have on the properties satisfied by these translation.

In answer to the first question, one need only observe that the translation $[\![-]\!]_A$ from λ_A to λ^*_R is also a well defined translation from λ_A to λ_R as long as one remembers that, formally, assignment and dereferencing of terms in the target are actually just projections. Specifically, the clauses for aspect installation and invocation in table 3.2 should be read as follows:

$$
\begin{array}{rcl}
[\![\ M\langle\!\langle N \rangle\!\rangle\ ]\!] & \triangleq & (\pi_2([\![M]\!]) \cdot \mathtt{skip}) \cdot [\![N]\!] \\
[\![\ M \gg N\ ]\!] & \triangleq & \mathtt{let\ Asp} = [\![M]\!] \\
 & & \mathtt{in}\ \pi_1(\pi_1(\mathtt{Asp})) \cdot \pi_2(\mathtt{Asp}); [\![N]\!]
\end{array}
$$

The proofs of preservation and reflection of evaluation, as well as the proof of substitution, and thus the proof of adequacy that apply to $[\![-]\!]_A$ can therefore remain identical. For similar reasons, the translation $[\![-]\!]^*_A$ is also a valid translation from λ^*_A to λ_R, satisfying the same properties. However, neither translation has any terms of type $\mathtt{ref}[\tau]$ (the "good" version!) in its image, making it impossible for them to satisfy the definability property. The properties satisfied by the various versions of these semantics are summarised by the following theorem.

Theorem 3.5.1. *Properties satisfied by the respective translations from λ_A/λ^*_A to λ_R/λ^*_R are as follows:*

	Properties				
	E	**T**	**D**	**A**	**F**
$\lambda_A \to \lambda_R$	*Yes*	*Yes*	*No*	*Yes*	*No*
$\lambda_A \to \lambda_R^*$	*Yes*	*Yes*	*Yes*	*Yes*	*No*
$\lambda_A^* \to \lambda_R$	*Yes*	*Yes*	*No*	*Yes*	*No*
$\lambda_A^* \to \lambda_R^*$	*Yes*	*Yes*	*Yes*	*Yes*	*Yes*

Reading of the table: E.g. the second row summarises properties satisfied by the λ_A-to-λ_R^* translation (see Section 3.3), which satisfies Properties E, T, D and A, but not F[1].

The table suggests a pattern: so long as the imperative features of the target language are sufficiently expressive to *emulate* those of the source (as references can emulate labels), the presence of bad variables in the target enables the definability property. This is fairly intuitive: labels mimic the behaviour of references as pairs, so identifying references and pairs in the target language allows the reference emulator of λ_A to be translated into a term of the correct type. Similarly, bad labels in the source only have an effect if bad variables are already present in the target language. This is also intuitive: making labels bad can be viewed as decreasing the expressive power of the source (label equality becomes undefinable [3]), it therefore stands to reason that there is no change to adequacy or definability when translating into λ_R. However, when translating into λ_R^*, bad labels collapse the type structure in the source language, preventing the exploit discussed above and enabling full abstraction.

[1]Note that the table does not apply to all possible translations, just the ones documented here, i.e. a "No" in the table does not necessarily imply that no translation exists which satisfies the corresponding property

3.6 Chapter and Related Work Summary

The previous sections have expressed the semantics of the additive (i.e. `return`-less) fragment of Walker et al.'s aspect calculus in terms of ordinary general references in the style of [3]. They proved that there exists a translation between the two languages that is adequate, and therefore that the game semantics model of the target language is a model of λ_A (and thus the additive fragment of MinAML). If the source language is modified to allow "bad labels", then there is a fully abstract translation, and therefore the game semantics of general references form a fully abstract model of the modified aspect calculus.

The use of this technique eliminates much of the complexity of the definability proof. In fact, the only difference between the user term syntax of λ^*_A and λ^*_R is their respective `new` operators, all of the other productions are inherited from the underlying functional calculus. In essence, this is is an operational reflection of the structure of the game semantics model of λ^*_R [3]. As a result, the definability proof is reduced to proving that the `newref` operator can be emulated in λ^*_A, and in this respect is reminiscent of the factorisation theorems used in the game semantics literature [6, 5, 8]. These papers often use an explicit constructor (`mkvar`) in order to cast pairs as imperative types to enforce the required isomorphism. The same technique could have been used here, but would have required adding `mkvar` and its analogue for labels (`mklab`) to both languages in order to assure type preservation.

By contrast, McCusker's proof of definability for FPC characterises program equivalence by considering its compact terms [45], while ours resorts to existing denotational semantics. In this regard, our proof is akin to those of Riecke [52], who used domain-theoretic models of λ-calculi to introduce the notion of translations between languages. Finally, the aforementioned paper by Schmidt-Schauss et al. only considers fully ab-

stract translations between a language and its extensions, thereby easing the proof of definability significantly (cf. [54, Definition 3.5]). Other proof techniques that were considered for this purpose included the operational strategies proposed by Mason and Talcott [44] and also Pitts [49]. However these proved insufficient for our purposes: their approximations proved too coarse to successfully equate the terms we were examining.

Chapter 4

Around Advice with Exceptions

This chapter extends the previous analysis to encompass the `return` primitive from the aspect calculus defined in chapter 2 and [59]. The previous chapter's results showed that bad names in both the source and target language were required to enable a fully abstract translation. The situation is no different here: if bad labels are not included, the type mismatch between source and target could be exploited as in the additive case. The translations below therefore only consider languages which include bad names, be they labels, references, or exceptions. Much like the previous chapter, translations can be constructed between languages which have no bad names to achieve properties analogous to those of theorem 3.5.1, but an in-depth discussion of these is omitted in order to avoid too much repetition.

The semantics below translates λ^*_{L}, i.e. Walker's core aspect calculus with bad labels, into a language including general references and locally scoped exceptions in the style of [34]. Unfortunately, the target language used currently has no fully abstract model, therefore a definability result required to achieve full abstraction remains open. In light of the results presented in the rest of the thesis, and because of the considerable explosion in the length required to present such a model, this result has been left for future

work. However, the end of this chapter discusses a possible avenue that researchers could take for finding such a model, and chapter 7 discusses why there is a good chance that this strategy could be successful.

Before continuing, it is reasonable to ask whether the drastic extension to exceptions is actually necessary: can the `return` primitive be expressed in terms of mutable storage alone? For that matter, can exceptions? To answer this question, consider the following terms of type $(\texttt{unit} \to \texttt{unit}) \to \texttt{unit}$:

$$\begin{aligned} M &\triangleq \lambda f.[f \cdot \texttt{skip}; \Omega] \\ N &\triangleq \lambda f.\Omega \end{aligned}$$

where Ω is a divergent term. Viewed as λ^*_{A} terms, it is fairly easy to see that $M \simeq N$. Take an arbitrary context $\mathcal{C}[-]$, the only way that $\mathcal{C}[M]$ can possibly converge is if M is never called. This is clearly also the case for N, as it simply immediately diverges if called. However, if M and N are taken as terms of λ^*_{L}, then `return` might appear in the context and be passed into the input variable f. Calling f may then allow M to avoid the divergent term, but not N because it ignores its input. For example, take the context $\mathcal{C}[-]$ to be

$$\mathcal{C}[-] \triangleq \texttt{newlab}\ l\ \texttt{in}\ l\langle\!\langle [-] \cdot \lambda d.\texttt{return skip to}\ l \rangle\!\rangle$$

and notice that $\mathcal{C}[M]$ converges, but $\mathcal{C}[N]$ does not, so $M \not\simeq N$ in λ^*_{L}. It follows that the model of λ^*_{R} (and thus λ^*_{A}) is not a model of λ^*_{L}, i.e. references can't model `return`. A very similar argument can be made to show that the exceptions introduced below can't be modelled by references either.

4.1 Bad Returning Labels

Recall from chapter 2 that the syntax of λ_L extends the productions of λ_A with the `return` M `to` N construct. This means that the language provides three separate operators on labels: installation, labelling and returning. Therefore, in order to add bad labels to this language to form λ_L^*, the type of labels must be identified with a 3-tuple of methods to accommodate the `return` operator added to λ_A^*.

What should the types of these methods be? The aspect installation primitive $\alpha \triangleright M$ is as in the additive case: it takes a piece of advice of type $\tau \to \tau$ and returns `unit`. The "returner" function, however, presents a slight difficulty. Given a value, it needs to evaluate to the expression returning the value to the label. Now the `return` expression, while a possible end result of evaluation, is not itself a value, so the only possible output type that would force a function to return it is the empty type `nil`. Once this type is added to the language, the returning component of a label of type τ is typed as $\tau \to$ `nil`, accepting a value and producing the `return` expression carrying it.

The labelling function is more problematic. As in the additive case, what is required is a function that accepts the value of the base code, and returns the result of passing this value to all of the advice associated to the label. If the label has type τ, this corresponds to a function of type $\tau \to \tau$[1]. In λ_L^*, there is the possibility that the labeller's argument is a `return` expression. This would not cause a problem in a call-by-name language, but in call-by-value the evaluation of the argument to a `return` results in the entire call evaluating to `return`, making it impossible for any function of type $\tau \to \tau$ to properly "catch" the returning expression. It is therefore necessary to delay the evaluation of the argument until it has been passed to the labeller by thunking the input type, making the labelling function have type $(\texttt{unit} \to \tau) \to \tau$.

[1] In the previous chapter, the delayed type $\texttt{unit} \to (\tau \to \tau)$ was used solely for the aesthetic reason that it equated the type $\texttt{lab}[\tau]$ with the type $\texttt{ref}[\tau \to \tau]$

To summarise, the imperative type $\texttt{lab}[\tau]$ in λ^*_{L} is defined as the following 3-tuple:

$$\begin{array}{rcll} \texttt{lab}[\tau] & \triangleq & (\tau \to \tau) \to \texttt{unit} & \times \quad \text{(Aspect Installation)} \\ & & (\texttt{unit} \to \tau) \to \tau & \times \quad \text{(Program Point Labelling)} \\ & & \tau \to \texttt{nil} & \quad \text{(Value Return)} \end{array}$$

The syntax adds the $\texttt{return}\ M\ \texttt{to}\ \alpha$ expression, whose semantics are as usual, to the terms of λ^*_{A}. The three projections on raw label names have the β rules:

$$\begin{array}{rcl} (L, A, \pi_1(\alpha)) & \xrightarrow{\beta} & (L, A, \lambda x.\alpha \triangleright x) \\ (L, A, \pi_2(\alpha)) & \xrightarrow{\beta} & (L, A, \lambda f.\alpha\langle\!\langle f \cdot \texttt{skip} \rangle\!\rangle) \\ (L, A, \pi_3(\alpha)) & \xrightarrow{\beta} & (L, A, \lambda x.\texttt{return}\ x\ \texttt{to}\ \alpha) \end{array}$$

Of note is the second projection, which dethunks its input after it's been labelled. The usual aspectual operations on arbitrary expressions can be recovered by projecting terms of label type:

$$\begin{array}{rcl} M \triangleright N & \triangleq & \pi_1(M) \cdot N \\ M\langle\!\langle N \rangle\!\rangle & \triangleq & \pi_2(M) \cdot \lambda d.N \\ \texttt{return}\ M\ \texttt{to}\ N & \triangleq & \texttt{let}\ x = M\ \texttt{in}\ \pi_3(N) \cdot x \end{array}$$

The `let` expression in the third projection is used to assure that M is evaluated before N as in Walker et al.'s definition of `return`.

4.2 Locally Declared Exceptions

Due to the presence of general references and exceptions, denote the translation's target language λ^*_{ML} as it forms a concise core of the major features of ML. Its syntax extends

the productions of λ^*_{R} with the following:

$$\begin{array}{rcl} M & ::= & \cdots \mid \xi \mid \texttt{newexn}[\tau] \mid \texttt{handle}\,\xi\,M \mid \texttt{raise}\,\xi\,M \\ V & ::= & \cdots \mid \xi \end{array}$$

The greek ξ ranges over an infinite set of *exception names* disjoint from the set of reference names. The language is similar to a call-by-value version of the one studied in [34] extended with higher order references. As in λ^*_{L}, the `nil` type is added to the ground types of λ^*_{R}. The type judgements for the control constructs are defined with respect to a map X from exception names to types in addition to the reference map R. These are presented in table 4.1, and use a type $\texttt{exn}[\tau]$ as the type of exception names. This is a shortcut for the pair consisting of the exception's "raiser" and "handler" methods:

$$\texttt{exn}[\tau] \triangleq \overbrace{\tau \to \texttt{nil}}^{\texttt{raise}} \times \overbrace{(\texttt{unit} \to \tau) \to \tau}^{\texttt{handle}}$$

In a call-by-name language these would be the mutually inverse $\tau \to \texttt{nil}$ and $\texttt{nil} \to \tau$, but for the same reason as the labelling function in λ^*_{L} the input to the handler must be thunked in call-by-value. Intuitively, the raiser takes an exception name and propagates it up the syntax tree until it reaches a handler, which then simply evaluates to the value given to the `raise` function when it was called if the exception being handled matches the one that was raised.

Configurations of λ^*_{ML} are 4-tuples of the form (X, R, S, M), with X and R as above and S defined as a λ^*_{R} store. A well typed configuration is defined analogously to the ones studied previously: the term M is typed with respect to the X and R components, and the types of the store's contents are also checked. A value configuration is defined as before, and an *error configuration* is one whose term component is a `raise` expression. The big-step operational semantics is defined as a pair of relations:

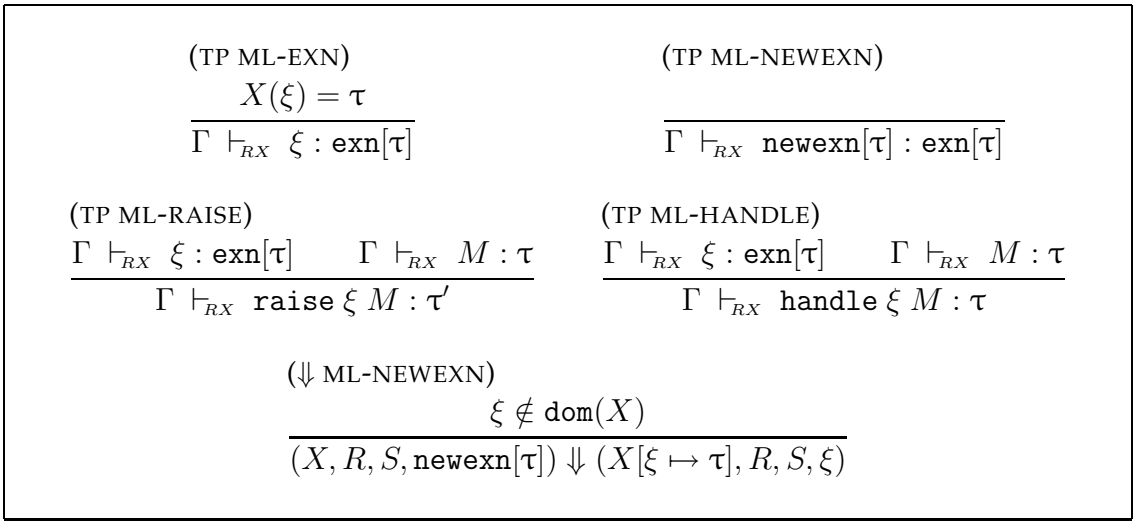


Table 4.1: λ^*_{ML} *type system*

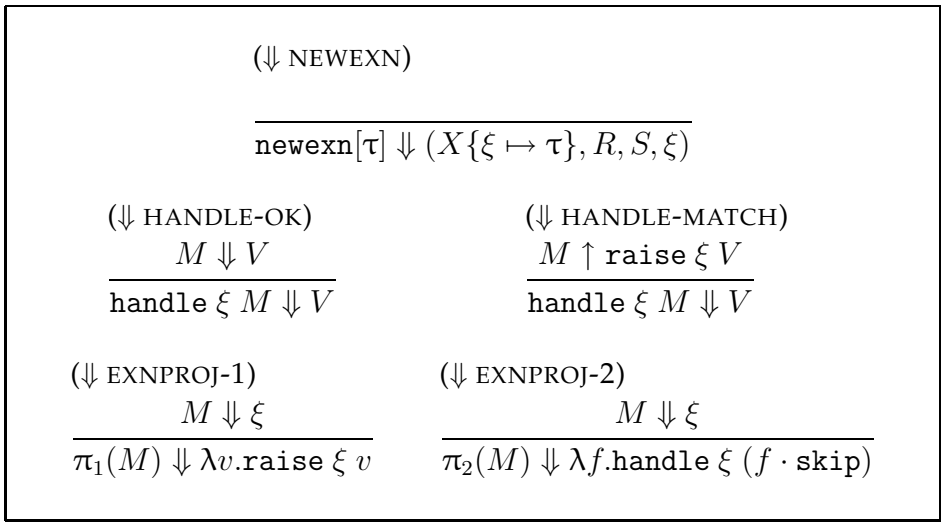


Table 4.2: λ^*_{ML} *evaluation rules*

- The $\Downarrow$ notation relates configurations to value configurations as usual. Its additional operational rules appear in table 4.2.
- The $\uparrow$ notation relates configurations and error configurations, and can be read as *raises*. Its operational semantics are in table 4.3.

$$(\uparrow \text{RAISE}) \quad \frac{M \Downarrow V}{\mathtt{raise}\ \xi\ M \uparrow \mathtt{raise}\ \xi\ V} \qquad (\uparrow \text{PROP-HANDLE}) \quad \frac{M \uparrow \mathtt{raise}\ \xi'\ V}{\mathtt{handle}\ \xi\ M \uparrow \mathtt{raise}\ \xi'\ V}\ [\xi \neq \xi']$$

$$(\uparrow \text{PROP-COND}_0) \quad \frac{M \uparrow \mathtt{raise}\ \xi\ V}{\mathtt{cond}\ M\ N_{\mathtt{tt}}\ N_{\mathtt{ff}} \uparrow \mathtt{raise}\ \xi\ V} \qquad (\uparrow \text{PROP-COND}_b) \quad \frac{M \Downarrow b \qquad N_b \uparrow \mathtt{raise}\ \xi\ V}{\mathtt{cond}\ M\ N_{\mathtt{tt}}\ N_{\mathtt{ff}} \uparrow \mathtt{raise}\ \xi\ V}\ [b \in \{\mathtt{tt}, \mathtt{ff}\}]$$

$$(\uparrow \text{PROP-PAIR}_i) \quad \frac{M_i \uparrow \mathtt{raise}\ \xi\ V}{\langle M_1, M_2\rangle \uparrow \mathtt{raise}\ \xi\ V} \qquad (\uparrow \text{PROP-PROJ}_i) \quad \frac{M \uparrow \mathtt{raise}\ \xi\ V}{\pi_i(M) \uparrow \mathtt{raise}\ \xi\ V}\ [i \in \{1,2\}]$$

$$(\uparrow \text{PROP-APP}_1) \quad \frac{M \uparrow \mathtt{raise}\ \xi\ V}{M \cdot N \uparrow \mathtt{raise}\ \xi\ V} \qquad (\uparrow \text{PROP-APP}_2) \quad \frac{M \Downarrow V \qquad N \uparrow \mathtt{raise}\ \xi\ V}{M \cdot N \uparrow \mathtt{raise}\ \xi\ V}$$

$$(\uparrow \text{PROP-NEWREF}) \quad \frac{M \uparrow \mathtt{raise}\ \xi\ V}{\mathtt{newref}[\tau](M) \uparrow \mathtt{raise}\ \xi\ V} \qquad (\uparrow \text{PROP-ASSIGN}) \quad \frac{M \uparrow \mathtt{raise}\ \xi\ V}{\rho := M \uparrow \mathtt{raise}\ \xi\ V}$$

Table 4.3: λ^*_{ML} *exception propagation*

The usual conventions for omitting the environment components X, R, and S apply to both tables. Observational equivalence can be defined for user terms of λ^*_{ML} just as for λ^*_{L} and λ^*_{R}, i.e. two terms are equivalent if they can be interchanged in all program contexts without changing termination behaviour. Because raising an exception is not convergence, this definition of equivalence equates an uncaught exception with divergence, i.e. if $M \uparrow$ then $M \simeq \Omega$.

4.3 Translation from Labels to Exceptions

The semantics of returning and labelling in λ^*_{L} is analogous to the control primitives of λ^*_{ML}, with `return` viewed as a raiser and labelling as a handler. In a sense, a label encapsulates the features of a reference cell and an exception into a single imperative type.

As in the additive case, assume bijections taking label names to reference and exception names respectively, and write ρ_α for the reference corresponding to label α and ξ_α for the exception. The principal clauses of the $[\![-]\!]^*_{\mathsf{L}}$ translation function, taking terms of λ^*_{L} to λ^*_{ML}, are shown in Table 4.4. The semantics are very similar to those presented for the additive case, with a few key differences:

- The label emulator $\mathscr{L}$ accepts both an exception and a reference
- The translation of program point labelling $\alpha\langle\!\langle M \rangle\!\rangle$ now wraps the translate of M inside a handler which looks for the exception corresponding to α. Once this handler is evaluated, the result is passed to the contents of the reference corresponding to α, which stores the advice as before.
- The additional `return` construct is modelled straightforwardly by raising an exception with the appropriate value.

The translation takes a λ^*_{L} label to a triple of type $\mathtt{lab}[\tau]$ which uses a reference and an exception to mimic the behaviour of labels. The emulator $\mathscr{L}_{\mathsf{L}}$ is defined as follows:

$$\mathscr{L}_{\mathsf{L}}(\mathtt{r},\mathtt{x}) \triangleq \langle\, \mathtt{install}\,,\, \mathtt{invoke}\,,\, \mathtt{return} \,\rangle$$

$$\begin{aligned}
\mathtt{install} &\triangleq \lambda \mathtt{f}.\mathtt{let}\ \mathtt{New} = \mathtt{f}\ \mathtt{in}\ \mathtt{let}\ \mathtt{Old} = !\mathtt{r}\ \mathtt{in}\ \mathtt{r} := \mathtt{New} \circ \mathtt{Old} \\
\mathtt{invoke} &\triangleq \lambda \mathtt{f}.\mathtt{let}\ \mathtt{Base} = \mathtt{handle}\ \mathtt{x}\ (\mathtt{f} \cdot \mathtt{skip})\ \mathtt{in}\ !\mathtt{r} \cdot \mathtt{Base} \\
\mathtt{return} &\triangleq \lambda \mathtt{Val}.\mathtt{raise}\ \mathtt{x}\ \mathtt{Val}
\end{aligned}$$

$$
\begin{array}{rcl}
[\![\, \alpha \,]\!] & \triangleq & \mathscr{L}_{\mathsf{L}}(\, \rho_\alpha \,,\, \xi_\alpha \,) \\
[\![\, \texttt{newlab}[\tau] \,]\!] & \triangleq & \texttt{let} \\
 & & \quad \texttt{Ref} = \texttt{newref}[\tau \to \tau](\texttt{id}[\tau]) \\
 & & \quad \texttt{Exc} = \texttt{newexn}[\tau] \\
 & & \texttt{in}\ \mathscr{L}_{\mathsf{L}}(\texttt{Ref}, \texttt{Exc}) \\
[\![\, \alpha \triangleright M \,]\!] & \triangleq & \texttt{let} \\
 & & \quad \texttt{New} = [\![M]\!] \\
 & & \quad \texttt{Old} = !\rho_\alpha \\
 & & \texttt{in}\ \rho_\alpha = \texttt{New} \circ \texttt{Old} \\
[\![\, \alpha\langle\!\langle M \rangle\!\rangle \,]\!] & \triangleq & \texttt{let} \\
 & & \quad \texttt{Base} = \texttt{handle}\ \xi_\alpha\ [\![M]\!] \\
 & & \texttt{in}\ !\rho_\alpha \cdot \texttt{Base} \\
\mathrm{split}[\![\, \texttt{return}\ M\ \texttt{to}\ \alpha \,]\!] & \triangleq & \texttt{raise}\ \xi_\alpha\ [\![M]\!]
\end{array}
$$

Table 4.4: *Term translation* $[\![-]\!]^*_{\mathsf{L}}$ *from* λ^*_{L} *to* λ^*_{ML}

The translation of `newlab` is then just this term with the reference and exception bound to fresh names. The translations of the three operators on label names correspond to the three components of the emulator. Here, references and exceptions are bound to the image of the source label under the corresponding bijection. Essentially, the key idea is to explicitly split a label into its two roles:

1. As a marker corresponding to installed advice (modelled by a reference)

2. As a point where a value may be returned (modelled by an exception)

Establishing how full configurations are translated completes the semantics. This is achieved in an essentially pointwise fashion similar way to the corresponding definition for λ^*_{A}. However, two auxiliary functions must now be used to take each label to *both* a reference and an exception. The formal rules for translating configurations are

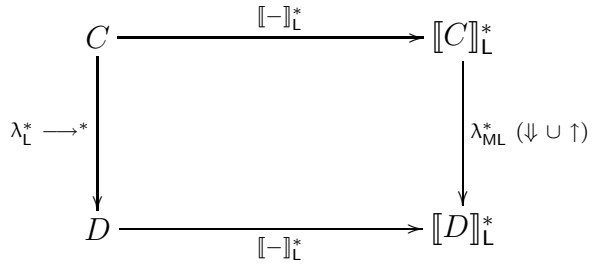
$$
\begin{array}{rcl}
[\![\ L[\alpha \mapsto \tau]\ ]\!]_X & \triangleq & [\![L]\!]_X\ [\xi_\alpha \mapsto \tau] \\
[\![\ L[\alpha \mapsto \tau]\ ]\!]_R & \triangleq & [\![L]\!]_R\ [\rho_\alpha \mapsto \tau \to \tau] \\
\\
[\![\ L[\alpha \mapsto \tau]\ ,\ A\ ]\!]^*_{\mathsf{L}} & \triangleq & [\![L, A]\!]^*_{\mathsf{L}}\ [\rho_\alpha \mapsto [\![\mathtt{lookup}\langle L, A\rangle(\alpha)]\!]^*_{\mathsf{L}}] \\
\\
[\![\ L, A, M\ ]\!]^*_{\mathsf{L}} & \triangleq & (\ [\![L]\!]_X\ ,\ [\![L]\!]_R\ ,\ [\![L, A]\!]^*_{\mathsf{L}}\ ,\ [\![M]\!]^*_{\mathsf{L}}\ )
\end{array}
$$

Table 4.5: *The $[\![-]\!]^*_{\mathsf{L}}$ translation function as it applies to configurations. The functions $[\![-]\!]_X$ and $[\![-]\!]_R$ are used to transform the label map L into an exception map and reference map respectively.*

presented in table 4.5.

As expected, this translation satisfies the desired commutativity property: a λ^*_{L} configuration evaluates to a final form D if and only if its translate terminates (whether by successful evaluation or raising an exception) to $[\![D]\!]^*_{\mathsf{L}}$:

Proposition 4.3.1 ($[\![-]\!]^*_{\mathsf{L}}$ Commutativity)**.** *For any λ^*_{L} configuration C, the following diagram commutes:*



Proof. The proof of this property is analogous to the corresponding proof for $[\![-]\!]^*_{\mathsf{A}}$. As such it is proved in three stages: first, the semantics of λ^*_{L} are expressed in big-step style and proved equivalent to Walker's transition semantics (cf. Appendix A.3). Next, preservation and reflection of evaluation is proved by an induction on the derivations of the source and target language terms respectively (cf. Appendix B.3). For the latter, note that only those λ^*_{ML} configurations which are translates of λ^*_{L} ones need be considered. Therefore the cases for the induction can be enumerated (as in the preservation result)

by the structure of the term component of the source configuration. Care must be taken in this case to assure that the induction hypothesis is only applied to λ^*_{ML} configurations which respect this property. □

A direct corollary of this result is that the $[\![-]\!]^*_{\mathsf{L}}$ translation function is termination equivalent:

Corollary. *For any λ^*_{L} user term M, $M\Downarrow$ if, and only if $[\![M]\!]^*_{\mathsf{L}}\Downarrow$.*

Furthermore, the translation is compositional by inspection of its definition[2], therefore by the reasoning of Chapter 2, it can be concluded that the translation function defines an adequate semantics of Walker et al.'s aspect calculus with bad labels.

Theorem 4.3.2. *For any λ^*_{L} user terms M and N, if $[\![M]\!]^*_{\mathsf{L}} \simeq [\![N]\!]^*_{\mathsf{L}}$ then $M \simeq N$.*

Just as in the additive case, a similar result can be established without using bad labels, references, and exceptions. These were included in the languages here to maintain continuity between the last chapter and the next, and also because their presence removes one obstacle in the path of a full abstraction result. Unfortunately, a proof of the latter for this language remains elusive despite this. Primarily, this is because the target language λ^*_{ML} currently has no known fully abstract game model. Consequently, this model can't be used to prove a definability property, and thus full abstraction can not be established.

With respect to the primary goals of this thesis, namely establishing foundational semantics for aspect oriented programming, this does not turn out to be a large setback. The next chapter shows that a fully abstract semantics for pre-emptive `around()` advice (i.e. precisely the advice modelled by Walker's `return` primitive) can be constructed without resorting to exceptions at all. The brunt of the work required to find such a

[2]To be totally formal, the substitution property can easily be proved with a routine induction on the type derivation of M

model is therefore left for future work, as it falls outside of the scope of this thesis' aims, and would result in a large explosion in its length.

The discovery of a model for λ^*_{ML} is however of considerable interest to the larger TCS community, as the language forms a concise core of Standard ML. The next section is therefore devoted to outline a strategy for constructing one, and argues why it is likely to succeed.

4.4 On Finding a Model of Exceptions and H.O. State

Since game semantics was successfully applied to the simply typed λ-calculus with recursion (PCF), yielding its first fully abstract denotational model [4, 26], models for more feature-rich languages have been found by simple modifications to the original model. More specifically, the model for PCF is in a sense a very restrictive one, reflecting the highly structured nature of purely functional programming. Its elements must meet a number of stringent properties in order to achieve a tight enough correspondence between the model and the language. It was later discovered that the *relaxation* of each of these properties yielded models of other traditional language features such as state and control. As a result of this elegant connection between the constraints placed upon the model and the language features under consideration, the vast majority of papers which introduce game semantics models of languages take an existing model of some language which is close to the one being studied, and prove that removing a constraint results in a model of the new language.

Applying this strategy to the problem of constructing game semantics for λ^*_{ML} therefore requires such a starting point. The language closest to λ^*_{ML}, which also happens to have fully abstract game semantics, is the one studied by Laird in [34]. Call this language λ^*_{X}. There are a number of important differences between the features of this

language and those of λ^*_{ML}[3]:

- The `raise` expression of λ^*_{ML} carries a value with the raised exception, whereas those of λ^*_X do not. When successfully handled, the latter's exceptions just evaluate to `skip`.

- The `handle` expression of λ^*_{ML} allows the examined term to successfully terminate, but the corresponding expression in λ^*_X *requires* its sub-term to raise an exception. This constraint is enforced by the type system.

- λ^*_{ML} allows higher order references cells, but Laird's language only permits ground type references.

One sensible strategy for constructing the desired model of λ^*_{ML} would therefore proceed in two stages:

1. Consider the language of [34] extended to allow general references, and call it λ^*_{RX}. This language could also be viewed as an extension of λ^*_R with local exceptions of the above kind. Construct a model of this language by "combining" the game models of λ^*_R and λ^*_X.

2. Bridge the gap between λ^*_{ML} and λ^*_{RX} by using the translation techniques used in this thesis. If this translation can be proved fully abstract (using the model from step 1), then the desired model of λ^*_{ML} will be inherited from λ^*_{RX}.

The rest of this section is devoted to describing a strategy for accomplishing the second step. For a more detailed discussion of why step 1 is likely to be successful, please consult the relevant section in chapter 7.

[3]One difference not mentioned in this list is that the language studied in [34] is call-by-name. However, as mentioned in the paper itself, a call-by-value version of the model can be constructed using standard techniques ([6])

Denote the exception type, `new` constructor for exceptions, and `raise` and `handle` expressions of λ^*_{RX} by $\mathtt{exn}_0$, $\mathtt{newexn}_0$, $\mathtt{raise}_0\ \xi$, and $\mathtt{handle}_0\ \xi\ M$ respectively. Note that the exception type is not qualified by τ, and that the raiser no longer accepts a value. The type $\mathtt{exn}_0$ in this language is therefore the following:

$$\mathtt{exn}_0 \quad \triangleq \quad \overbrace{\mathtt{unit} \to \mathtt{nil}}^{\mathtt{raise}_0} \quad \times \quad \overbrace{(\mathtt{unit} \to \mathtt{nil}) \to \mathtt{unit}}^{\mathtt{handle}_0}$$

Namely, the raiser takes no input and raises an exception, and the handler accepts a thunked exception (enforced by requiring the `nil` type as input) and evaluates to `skip` if it matches the exception name being handled.

One way to construct an adequate translation from λ^*_{ML} to λ^*_{RX} is to do so through an intermediate language λ^*_{V} which only changes one feature of the control expressions of λ^*_{ML}. Denote these expressions by $\mathtt{newexn}_{\mathtt{v}}[\tau]$, $\mathtt{handle}_{\mathtt{v}}\ \xi\ M$, and $\mathtt{raise}_{\mathtt{v}}\ \xi\ M$, and define the type of exceptions of λ^*_{V} by

$$\mathtt{exn}_{\mathtt{v}}[\tau] \quad \triangleq \quad \overbrace{\tau \to \mathtt{nil}}^{\mathtt{raise}_{\mathtt{v}}} \quad \times \quad \overbrace{(\mathtt{unit} \to \mathtt{nil}) \to \tau}^{\mathtt{handle}_{\mathtt{v}}}$$

Contrast this with the types $\mathtt{exn}[\tau]$ and $\mathtt{exn}_0$: the λ^*_{V} raiser carries a value of type τ, and its handler evaluates to the carried value on a successful match. However, the handler here still requires its input to raise an exception. A translation from λ^*_{ML} to λ^*_{RX} can now be defined by composing translations $[\![-]\!]^*_{\mathsf{ML}} : \lambda^*_{\mathsf{ML}} \to \lambda^*_{\mathsf{V}}$ and $[\![-]\!]^*_{\mathsf{V}} : \lambda^*_{\mathsf{V}} \to \lambda^*_{\mathsf{RX}}$.

Note that the three types of exceptions under consideration are not all equivalent (i.e. inter-encodable) on their own: the fact that higher order references are available in all can be crucial to achieving sound translations. In fact, Laird briefly proposes a candidate translation for $[\![-]\!]^*_{\mathsf{V}}$ (cf. [34, §2]). The translation works by declaring a λ^*_{RX} exception and a reference of type τ for each λ^*_{V} reference of type $\mathtt{exn}_{\mathtt{v}}[\tau]$. Let ρ_ξ denote

the reference corresponding to exception name ξ, it can be used to model λ^*_{V}'s raiser and handler functions as follows:

$$[\![\, \mathtt{raise}_{\mathtt{v}}\ \xi\ M\ ]\!]^*_{\mathsf{V}} \triangleq \rho_\xi := [\![M]\!]^*_{\mathsf{V}}; \mathtt{raise}_{\mathtt{0}}\ \xi$$

$$[\![\, \mathtt{handle}_{\mathtt{v}}\ \xi\ M\ ]\!]^*_{\mathsf{V}} \triangleq \mathtt{handle}_{\mathtt{0}}\ \xi\ [\![M]\!]^*_{\mathsf{V}}\ ; !\rho_\xi$$

The idea is to store the value that the exception must carry into the reference, and then to retrieve it once it has been successfully handled. Since only one exception can be raised at any given time during execution, and it is immediately propagated until either handled or it reaches the top layer of the syntax tree, there can be no conflict with regard to the value being stored in the cell.

Expressing the control constructs of λ^*_{ML} in terms of those of λ^*_{V} is simpler. It can be accomplished by simply wrapping the handler's underlying term into a dummy exception to meet λ^*_{V}'s restrictive typing:

$$[\![\, \mathtt{raise}\ \xi\ M\ ]\!]^*_{\mathsf{ML}} \triangleq \mathtt{raise}_{\mathtt{v}}\ \xi\ [\![M]\!]^*_{\mathsf{ML}}$$

$$[\![\, \mathtt{handle}\ \xi\ M\ ]\!]^*_{\mathsf{ML}} \triangleq \mathtt{handle}_{\mathtt{v}}\ \xi\ (\mathtt{raise}_{\mathtt{v}}\ \xi\ [\![M]\!]^*_{\mathsf{ML}})$$

The translation of `raise` is straightforward. For `handle`, note that if the term $[\![M]\!]^*_{\mathsf{ML}}$ raises an exception which does not match ξ, then it is propagated as desired. If it raises ξ, then it is immediately caught by the surrounding $\mathtt{handle}_{\mathtt{v}}$ which evaluates to whatever value was carried by the exception. Similarly, if it evaluates normally, then it is immediately raised on the back of ξ and caught by the outer handler, which will evaluate to the desired value.

Assuming the $[\![-]\!]^*_{\mathsf{ML}}$ and $[\![-]\!]^*_{\mathsf{V}}$ are formally defined and proved adequate, the three translations can be composed to yield an adequate translation from λ^*_{L} to λ^*_{RX} (see figure 4.1). Definability for the overall translation could then be established if the afore-

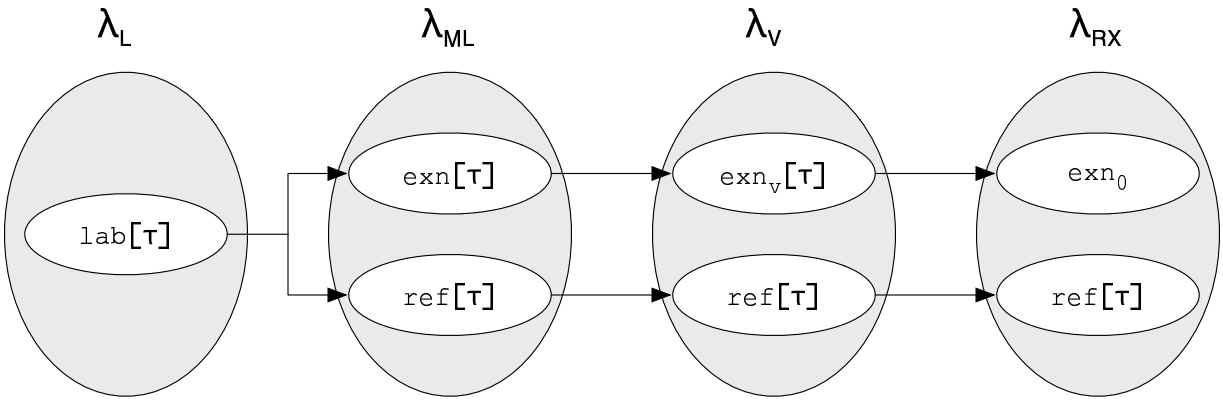


Figure 4.1: *Translating λ^*_{L} to λ^*_{RX} in three stages. After labels are split into exceptions and references by $[\![-]\!]^*_{\mathsf{L}}$, the other two stages leave the stateful expressions of the language untouched and only alter the exceptions.*

mentioned model of λ^*_{RX} is successfully constructed. Intuitively, modelling λ^*_{RX}'s exceptions using the aspect calculus seems straightforward: a raiser which does not carry a value would be modelled by a `return` which simply carries `skip`, and the handler would simply need to be wrapped into an abstraction which only permits `nil` type inputs.

It should be mentioned that, while $[\![-]\!]^*_{\mathsf{ML}}$ can be proved to be commutative using similar techniques used throughout the rest of the thesis, the $[\![-]\!]^*_{\mathsf{V}}$ translation cannot. This is because successfully caught exceptions results in information loss. To see why, consider the following example:

Example 4.4.1 (Non-Commutativity of $[\![-]\!]^*_{\mathsf{V}}$)**.** Let $X = \{\xi \mapsto \tau\}$, and V be an arbitrary value of type τ. Consider the λ^*_{V} configuration

$$C \equiv (\bot, X, \bot, \mathtt{handle}_{\mathsf{v}}\ \xi\ (\mathtt{raise}_{\mathsf{v}}\ \xi\ V))$$

The term is evaluated in an empty store and simply raises an exception which is immediately handled, so the result of evaluation is $C' = (\bot, X, \bot, V)$. According to the above description of $[\![-]\!]^*_{\mathsf{V}}$, each exception in the source corresponds to an exception and a ref-

erence in λ^*_{RX}. Therefore, when translating configurations, each exception $\xi \in \mathtt{dom}(X)$ must have a reference ρ_ξ of the same type added to the store in the image in order to keep track of any values with which it might be raised. Suppose this reference is initialised with some arbitrary value U (such a value can inductively be defined for each type), then C translates to:

$$[\![C]\!]^*_{\mathsf{V}} = (\bot, \{\xi\}, \{\rho_\xi \mapsto U\}, \mathtt{handle}_0\ \xi\ (\rho_\xi := [\![V]\!]^*_{\mathsf{V}}; \mathtt{raise}_0\ \xi)\ ; !\rho_\xi)$$

Notice that the exception component of the environment here is a set rather than a map because exceptions in λ^*_{RX} are not associated with types. Evaluating this configuration yields the desired value $[\![V]\!]^*_{\mathsf{V}}$, but it also leads to the the reference ρ_ξ being remapped to this value as well. The translation of C' however sets this reference to the default value U, violating commutativity.

The issue basically arises due to the locality of state in the languages: because reference and exception names can be extruded, unused references cannot be "garbage collected" as in Idealised Algol. One also will find that attempts to get around the issue by weakening the commutativity condition to exclude the problem cases will be unfruitful. The weakened induction hypothesis will be insufficient to push the proof through. The only choice appears to be to attempt to prove termination equivalence directly, but doing so requires some knowledge of the evaluation context in which a term appears in order to deduce the correct termination behaviour.

Interestingly, this is precisely the sort of problem studied by Pitts and Stark [50], although they did not deal with exceptions. Their technique extends usual configurations with an additional component, namely a continuation which encodes the evaluation context, and defines a unary termination relation directly on these extended configurations. Loosely speaking, let S be a store and M a term, a relation $\downarrow$ can be defined on

triples $(S, E[-], M)$ such that

$$(S, E[-], M)\downarrow \quad \Longleftrightarrow \quad (S, E[M])\Downarrow$$

Where $\Downarrow$ is the normal evaluation relation in the language. Then, taking $E[-]$ to be the empty context, one has a characterisation of termination in terms of the relation $\downarrow$. The advantage of the latter is that each rule in its definition forms its conclusion structurally from its premises, and therefore inductive proofs over the extended configurations are straightforward.

This technique appears a likely way to prove termination equivalence for λ^*_{V}. In order to apply it here, one would need to extend Pitts and Starks' techniques to account for exceptions. Extended configurations would need to be defined for λ^*_{V} and λ^*_{RX}, as would translations between them, and proofs of the alternate semantics' correspondence with the usual ones would have to be provided. As a result of this considerable amount of additional theoretical machinery, these tasks have been left as future work for those interested in computing a model of λ^*_{ML}.

4.5 Chapter and Related Work Summary

This chapter established that Walker et al.'s `return` construct, and hence their semantics of `around()` advice, cannot be encoded using general references alone. It extended the target language of chapter 3 with locally declared exceptions, and defined an adequate translation from λ^*_{L} into the resulting language. The usual definability property was not proved because no fully abstract denotational model of the translation's target language exists, so the full abstraction theorem is contingent on a proof of the definability property being found. Finding such a model is considered out of the scope of this thesis pri-

marily because the next chapter discusses an alternative definition of `around()` advice which can be modelled without exceptions. However, a model of λ^*_{ML} is of considerable interest to programming language researchers in general, as the language forms a core of many of the features of ML. The strategy for constructing such a model is discussed further in chapter 7.

The decision to use λ^*_{ML} as a target language for λ^*_{L} was fairly easy to make. It's connection to ML, the evident similarity between its control constructs and the behaviour of `return`, and the existence of the fully abstract model of λ^*_{RX} were all contributing factors. While there are a number of other control operators for which such models exist – [8, §6] and the work of Laird [33, 35, 37] describe a few – none seem to relate to the semantics of `return` as directly as the one that was chosen.

Some of the follow-up papers to Walker et al.'s paper on λ_{L} [59] explicitly omit the `return` primitive (and thus pre-emptive advice) from the language. For instance, their paper on "harmless" advice [15], as well as preliminary versions of PolyAML [16] do not consider such advice. The former paper proves a sort of non-interference property for an aspect calculus which does not exchange data with its underlying code, and justifies the omission of `around()` advice due to incompatibility with such a philosophy. Given the results of the next chapter, which shows how such advice can be modelled without exceptions, it is unclear if this restriction would still need to be applied.

Part III

Explicitly Advisable Functions

Chapter 5

Semantics of Around Advice with State

The previous two chapters explored the denotational semantics of Walker's core calculus of aspects by translation into higher order references and locally declared exceptions. It is important to remember that the λ_L calculus is a low level language intended to be expressive enough to encode the higher level AOP features of MinAML. The results of the previous chapters show that λ_L can itself be expressed with traditional notions of state and control. This chapter expresses MinAML-like constructs with imperative features *directly*, i.e. without going through an intermediate calculus. In fact, by defining an alternative semantics for `around` advice, the need for exceptions – used previously to encode `around` advice which does not `proceed` – is eliminated entirely.

Aside from the simplicity and elegance of the semantics, there are two further reasons for defining advice. First, while the semantics of chapter 3 are fully abstract for λ_A^*, and those of chapter 4 are (likely to be) for λ_L^*, neither of the models form fully abstract semantics for their respective fragments of MinAML itself. The discrepancy comes at the stage where the latter is translated into λ_L [59]. While this translation is inherently adequate – it forms the very definition of MinAML's operational semantics – it cannot possibly satisfy the definability property. Surprisingly, the failure isn't due to MinAML's

inability to express references or exceptions; indeed, references can readily be defined using constant functions using the technique introduced in chapter 3, and exceptions can similarly be instrumented if the `nil` type is added to the language. The problem is that plain λ abstractions are undefinable because all MinAML terms of function type are subject to advice. Therefore, contexts can always be devised which will use advice to distinguish them from unlabelled abstractions in λ_{L}. In fact, this is precisely the property that makes MinAML an oblivious language: there is no mechanism whereby a programmer can choose if his code can be advised. This choice can be provided by adding plain λ-abstractions to MinAML and giving named functions a distinguished type, effectively annulling the obliviousness property. The language introduced below does exactly this.

The second motivation for altering the semantics of advice is more practical. Namely, using `return` to encode non-proceeding `around` advice leads to some strange behaviour when multiple pieces of advice are concerned: once such advice is installed on a function, any further `before` advice installed on it will not execute. This effect can be seen almost immediately by examining figure 2.1. For instance, evaluating the MinAML term

```
let
  (fun f(x) = M_f)
  (around f(x) = M_a)
  (before f(x) = M_b)
in f()
```

results in the execution of $M_{\mathtt{a}}$, but not $M_{\mathtt{f}}$ (because the around advice replaces it) or $M_{\mathtt{b}}$ (because it is installed after the around advice). This kind of "forward interference", in which advice determines the execution (or non-execution) of later advice, seems at odds with good software development principles. Common applications of aspects include adding features or repairing bugs, but this sort of behaviour may cause such changes to be ineffective due to past code.

Still, one might argue that this semantics for aspects is just as good as any other, as any notion of "proper usage" of aspects is still not established. Another might say that the problem is easily repairable by simply reversing the precedence of aspects, effectively causing the control flow jump introduced by `return` to circumvent advice that was installed before it rather than after. However, further problems arise when *proceeding* around advice is involved. For example, substitute the following advice for the `before` advice in the above term:

$$(\texttt{around f}(x) = M_1;\ \texttt{proceed y} \rightarrow M_2)$$

In this case, when the function is called, the body of the first advice $M_{\mathtt{a}}$ replaces the function body as usual, but then control jumps to the "after" label of the function, at which point M_2 is executed (but not M_1, which is skipped over for the same reasons as above). This leaves the programmer in the bizarre situation in which only half of the code he just wrote executes. Not only that, but reversing the order of evaluation won't rectify the problem. The issue is inherent in the use of `return` to encode advice.

The rest of this chapter defines `around` advice a different way, and uses the techniques developed in previous chapters to define its semantics. Aside from solving the problem above, the benefits of the new semantics are:

- The `proceed` keyword can be called in the body of `around` advice an arbitrary number of times, including zero.
- The language can be fully abstractly encoded into λ^*_{R}, i.e. it does not require exceptions to encode non-proceeding arounds.
- As a result, there is a fully abstract games model of familiar AOP primitives.

The price paid for these properties is that the aspect language is not oblivious. This is

an acceptable cost, as other recent research seems to indicate that relaxing obliviousness allows for strong theoretical properties, and thus may not even be a desirable feature of AOP [9, 13, 17].

5.1 Advisable Functions as a Primitive

This section defines an imperative extension of the calculus λ in which local names are interpreted as advisable functions; that is, functions on which advice may be installed. Use ϕ to range over a countable set of *function names*, the syntax of λ_{F}^{*} extends that of λ with the following productions[1]:

$$\begin{array}{rcl} M & ::= & \cdots \mid \phi \mid \texttt{fun}\,(x:\tau)\{M\} \\ & & \mid \texttt{around}\,\phi(x)\{M\} \\ V & ::= & \cdots \mid \phi \end{array}$$

In order to keep the presentation concise, this language only includes `around` advice. Various extensions (including `before` and `after` advice) can straightforwardly be included, and many of these are discussed in the next chapter. The typing rules and operational semantics for these constructs are presented in table 5.1 and 5.2 respectively. The rules use $\sigma \Rightarrow \tau$ as the type for advisable functions, which is a shortcut for the object of accessor functions discussed below. The judgement is parametrised by a map F from function names to pairs of types (representing the input and output types of the advisable function). There are two intriguing points to note about the type system:

1. The rule (TP FUNAPP) "overloads" the application construct, allowing advisable functions to be called just as ordinary functions.

2. The rule (TP AROUND) binds a variable `proceed` in its subterm N which does not

[1]The "*" superscript, as usual, indicates that the imperative names of the language are "bad"

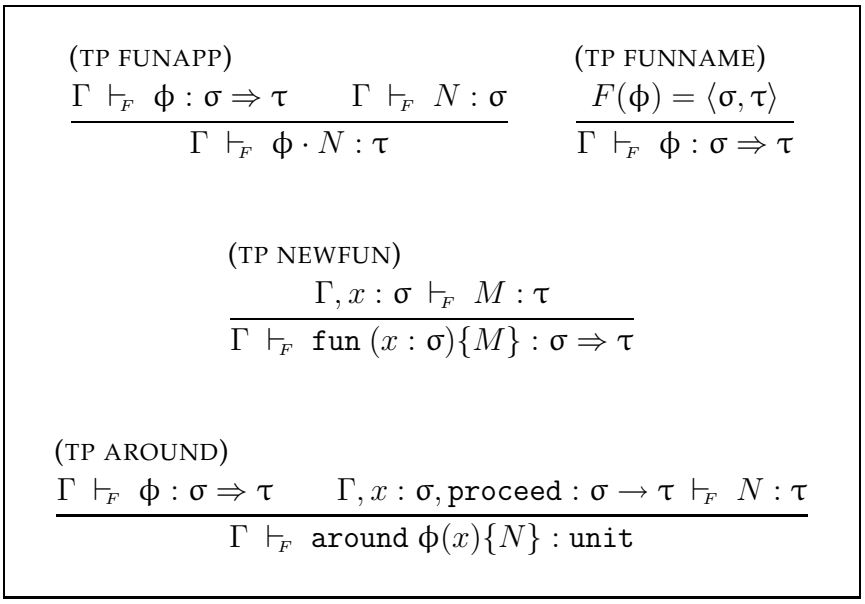


Table 5.1: *Type system for* λ^*_{F}

appear in the syntax of the `around` primitive itself. The semantics will bind this variable to the a function representing the advised code, allowing the keyword to be invoked arbitrarily many times in the body.

Configurations in λ^*_{F} take the form (F, A, M), where F is the map from the type rules. The sequence A is slightly different from the sequence in λ^*_{L}: it consists of a sequence of abstractions labelled by a function name, some of which have an additional tag "`around`" indicating that they are the result of an aspect installation. Those not labelled in this way represent the body of an advisable function as it was originally declared. The tagged form is added to the tail of the sequence using the aspect installation primitive, whose semantics are defined by the ($\Downarrow$ INSTALL) rule. Untagged abstractions are appended in the ($\Downarrow$ NEWFUN) rule, when the function is declared. The latter form is this language's equivalent of the `new` expression for advisable functions, i.e. it simply evaluates to a fresh name and appropriately updates the environment.

($\Downarrow$ NEWFUN)
$$\frac{\Gamma, x : \sigma \vdash_F M : \tau}{(F, A, \texttt{fun}\ (x : \sigma)\{M\}) \Downarrow (\ F[\phi \mapsto \langle \sigma, \tau \rangle]\ ,\ A :: \ \phi(x)\{M\}\ ,\ \phi\ )} \quad [\phi \notin \texttt{dom}(F)]$$

($\Downarrow$ FUNAPP)
$$\frac{N \Downarrow (F, A, V') \qquad M'[V'/x] \Downarrow V}{(F, A, \phi \cdot N) \Downarrow V} \quad [\texttt{lookup}\langle A \rangle(\phi) = \lambda x.M']$$

($\Downarrow$ INSTALL)
$$\frac{}{\texttt{around}\ \phi(x)\{N\} \Downarrow (F, A :: \texttt{around}\ \phi(x)\{N\}, \texttt{skip})}$$

($\Downarrow$ FUNPROJ$_1$)
$$\frac{M \Downarrow \phi}{\pi_1(M) \Downarrow \lambda f.\texttt{around}\ \phi(x)\{f \cdot \texttt{proceed} \cdot x\}}$$

($\Downarrow$ FUNPROJ$_2$)
$$\frac{M \Downarrow \phi}{\pi_2(M) \Downarrow \lambda x.\phi \cdot x}$$

Table 5.2: *Operational semantics of* λ^*_{F}

The type $\sigma \Rightarrow \tau$ of advisable functions is a shortcut for the pair

$$((\sigma \to \tau) \to \sigma \to \tau) \to \texttt{unit} \quad \times \quad \sigma \to \tau$$

The first element represents the aspect installation component, and takes a piece of `around` advice as input. One interpretation for the type of advice is as a function with two arguments: the raw function which is being advised (of type $\sigma \to \tau$), and the input which it was given when called (of type σ). An equivalent way to look at this type is to parenthesise it as a function of type $(\sigma \to \tau) \to (\sigma \to \tau)$. Under this view, a piece of `around` advice updates the behaviour of the entire code it advises: the input function is the base code, and the result is a new function of the same type including the additional advice. Figure 5.1 represents this interpretation of around advice as a diagram.

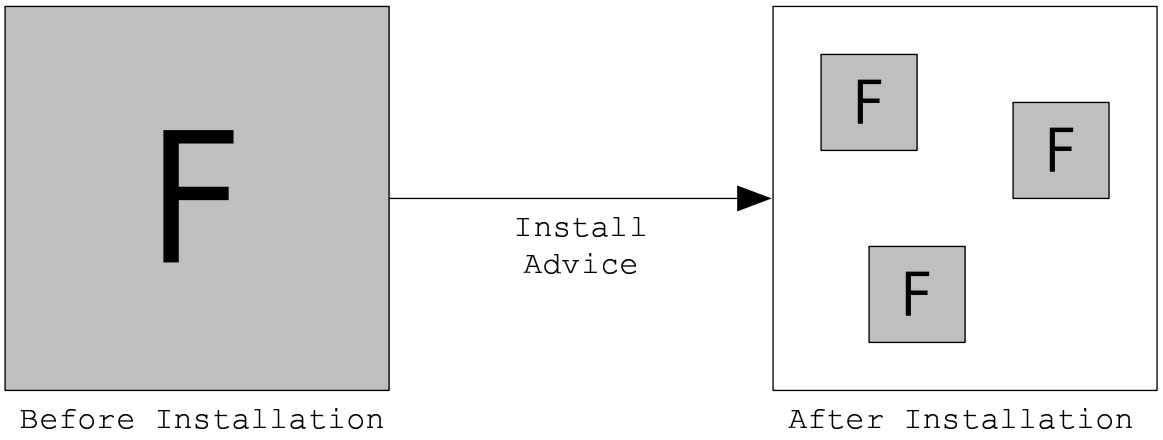


Figure 5.1: *Installing advice in* λ^*_{F}. *The grey boxes marked* F *represent the code executed when the advised function is called* before *installation. After installation, the result of calling the function is a larger function which may use* F *arbitrarily many times.*

The second component of $\sigma \Rightarrow \tau$ is simply the application operator. The projections on function names reflect these semantics, note the application in the ($\Downarrow$ FUNPROJ$_1$) rule, which binds the `proceed` keyword to the advice's argument.

Aspect installation and advisable function application over arbitrary expressions can therefore be encoded as projections in the usual way. Formally, let M be an arbitrary term of type $\sigma \Rightarrow \tau$, then:

$$\begin{aligned} \texttt{around}\ M(x)\{N\} &\triangleq \pi_1(M) \cdot (\lambda \texttt{proceed}.\lambda x.N) \\ M \cdot N &\triangleq \pi_2(M) \cdot N \end{aligned}$$

The rule ($\Downarrow$ FUNAPP) is the application rule for advisable functions, which is like the ordinary application rule except that the abstraction in the substitution premise is looked up in the aspect table using the `lookup` function defined as follows:

$$
\begin{array}{rcl}
\texttt{lookup}\langle A :: \phi(x)\{M\}\rangle(\phi) & = & \lambda x.M \\
\texttt{lookup}\langle A :: \texttt{around}\ \phi(x)\{M\}\rangle(\phi) & = & \lambda x.M[\ \texttt{lookup}\langle A\rangle(\phi)\ /\ \texttt{proceed}\ ] \\
\texttt{lookup}\langle A :: \phi'(x)\{M\}\rangle(\phi) & = & \texttt{lookup}\langle A\rangle(\phi) \qquad [\phi \neq \phi'] \\
\texttt{lookup}\langle A :: \texttt{around}\ \phi'(x)\{M\}\rangle(\phi) & = & \texttt{lookup}\langle A\rangle(\phi) \qquad [\phi \neq \phi']
\end{array}
$$

Given a name ϕ, this function iterates through the aspect sequence A in reverse order looking for advice on ϕ. When an element of the form "$\texttt{around}\ \phi(x)\{M\}$" is found, it constructs an abstraction by recursively replacing the `proceed` variable in the body M with the result of `lookup` on the rest of the list. The process ends when the abstraction representing the original declaration of ϕ is reached. Since the pointcuts in our language are in-scope, any advice associated with ϕ must have been declared after the function itself, so this definition is well formed.

Notice that λ^*_{F} is not oblivious because it gives the user the choice of whether or not to expose functions to advice: any attempt to advise an ordinary function results in a type error. In many ways, this property makes the language similar to Jonathan Aldrich's Open Modules (cf. § 1.2.3 of chapter 1 and [9]), where the choice is provided via a restriction operator rather than explicit typing. In fact, the semantics of `around` advice presented here are like those Aldrich proposed. The next section takes the paradigm one step further by fully abstractly encoding λ^*_{F} into λ^*_{R}.

5.2 Translation into State

The central idea of the encoding is to use a reference to store the current state of the join point. As usual, assume a bijection between the source and target language names. In this case, write ρ_ϕ for the reference corresponding to the advisable function name ϕ. This reference must be used to emulate the behaviour of the type $\sigma \Rightarrow \tau$ just as the $\mathscr{L}$ functions from previous chapters emulated the various sorts of labels. Define

the function $\mathscr{F}$ taking a value of type $\texttt{ref}[\sigma \to \tau]$ and producing the desired advisable function:

$$\begin{array}{rcl} \mathscr{F}(\texttt{r}) & \triangleq & \langle \texttt{install}, \texttt{apply} \rangle \\ \\ \texttt{install} & \triangleq & \lambda f.\texttt{r} := (\lambda p.\lambda x.f \cdot p \cdot x) \cdot !\texttt{r} \\ \texttt{apply} & \triangleq & \lambda x.\texttt{let Arg} = x \texttt{ in } !\texttt{r} \cdot \texttt{Arg} \end{array}$$

Once again, there appears to be unnecessary `let` statements and η expansions in the emulator's definition: the term $\lambda p.\lambda x.f \cdot p \cdot x$ in the installer is just two η expansions of f, and the input to the applier is `let` bound. Their role, as in previous encodings, is to ensure that syntactic equality holds when projections of names are translated. Intuitively, the reference holds the raw function that will execute when the advisable function is called. Installing a new piece of advice applies the current contents of this cell to the new advice, thereby binding it to the `proceed` variable, and stores the result back into the cell. The application function just does the obvious thing: given a value, it applies it to the contents of the cell, which by definition is the code that must be run. Advisable function creation is then translated by binding V to a freshly created cell. The translations of function names, aspect installation, and advisable function application use the cell ρ_ϕ in a similar way. The formal term translation clauses for the aspectual features of λ^*_{F} are in table 5.3. Observe that the `let` statement in the case for application is no longer superfluous: it assures that the argument is evaluated before the cell is de-referenced (i.e. the advice is looked up).

Translating the λ^*_{F} environments to λ^*_{R} works similarly to the corresponding translation of λ^*_{A}: for each name in the source, a reference is created storing the translate of the advice associated with that name. In this case, however, an advisable function ϕ of type $\sigma \Rightarrow \tau$ (represented by a type pairing $\langle \sigma, \tau \rangle$ in F) is translated into a cell of type $\sigma \to \tau$ containing the translate of $\texttt{lookup}\langle A \rangle(\phi)$. The formal clauses defining the translation of

$$
\begin{array}{rcl}
[\![\ \phi\ ]\!] & \triangleq & \mathscr{F}(\rho_\phi) \\
[\![\ \texttt{fun}\ (x:\sigma)\{M\}\ ]\!] & \triangleq & \texttt{let jp} = \texttt{newref}(\lambda x.[\![M]\!])\ \texttt{in}\ \mathscr{F}(\texttt{jp}) \\
[\![\ \texttt{around}\ \phi(x)\{M\}\ ]\!] & \triangleq & \rho_\phi := (\lambda\texttt{proceed}.\lambda x.[\![M]\!]) \cdot\, !\rho_\phi \\
[\![\ \phi \cdot M\ ]\!] & \triangleq & \texttt{let Arg} = [\![M]\!]\ \texttt{in}\ !\rho_\phi \cdot \texttt{Arg}
\end{array}
$$

Table 5.3: *Term translation $[\![-]\!]^*_{\mathsf{F}}$ from λ^*_{F} to λ^*_{R}. The sub- and super-script from the translation brackets is omitted for clarity.*

$$
\begin{array}{rcll}
[\![\ F[\phi \mapsto \langle\sigma,\tau\rangle]\ ]\!] & \triangleq & [\![F]\!][\rho_\phi \mapsto (\sigma \to \tau)] & (F \text{ Map}) \\
[\![\ F[\phi \mapsto \langle\sigma,\tau\rangle], A\ ]\!] & \triangleq & [\![F, A]\!][\rho_\phi \mapsto [\![\texttt{lookup}\langle A\rangle(\phi)]\!]] & (\text{Aspect Sequence}) \\
[\![\ (F, A, M)\ ]\!] & \triangleq & (\ [\![F]\!]\ ,\ [\![F, A]\!]\ ,\ [\![M]\!]\ ) & (\text{Configurations})
\end{array}
$$

Table 5.4: *Environment and configuration translation from λ^*_{F} to λ^*_{R}.*

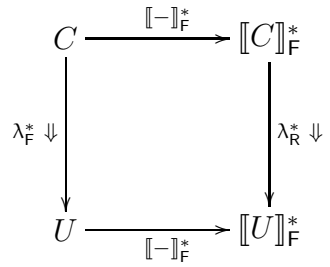
full configurations are presented in table 5.4.

5.3 Full Abstraction

The proof that $[\![-]\!]^*_{\mathsf{F}}$ is fully abstract follows the same blueprint as the one for proving similar results in previous chapters. Since the translation is type preserving, property S (and thus compositionality), follows from its definition. Furthermore, because of the way the function emulator $\mathscr{F}$ is defined, the translation function commutes with the operational semantics of the source and target language:

Proposition 5.3.1 ($[\![-]\!]^*_{\mathsf{F}}$ Commutativity)**.** *For every well typed configuration C of λ^*_{F}, $C \Downarrow U$ if and only if $[\![C]\!]^*_{\mathsf{F}} \Downarrow [\![U]\!]^*_{\mathsf{F}}$ for some value configuration U. Equivalently, the following diagram*

commutes:



Proof. See appendix B.4 for details of the preservation and reflection of evaluation proofs. □

Termination equivalence follows as a direct corollary to this proposition, and by the reasoning of chapter 2, adequacy follows closely on its heels.

Corollary ($[\![-]\!]^*_{\mathsf{F}}$ Property T)**.** *For any well typed user term M of λ^*_{F}, $M\Downarrow$ if and only if $[\![M]\!]^*_{\mathsf{F}}\Downarrow$*

Proof. Restrict the above proposition to well typed user terms, the result follows. □

Theorem 5.3.2 ($[\![-]\!]^*_{\mathsf{F}}$ Adequacy)**.** *If $[\![M]\!]^*_{\mathsf{F}} \simeq [\![N]\!]^*_{\mathsf{F}}$, then $M \simeq N$ for any well typed user terms M and N of λ^*_{F}*

Proof. By compositionality and termination equivalence. □

The only result left to prove is the definability property, which in conjunction with compositionality and adequacy implies full abstraction. Recall that the only formal syntactic and semantic difference between user terms of the two languages is their respective `new` statements (`fun` $(x : \sigma)\{M\}$ for λ^*_{F} and `newref`$[\tau](M)$ for λ^*_{R}). It therefore suffices to show that there exists a term of λ^*_{F} which emulates new reference creation, i.e. a term $\mathsf{F}[\![\texttt{newref}[\tau](M)]\!]$ such that

$$[\![\mathsf{F}[\![\texttt{newref}[\tau](M)]\!]]\!]^*_{\mathsf{F}} \simeq \texttt{newref}[\tau](M)$$

The same idea used to accomplish this feat for $[\![-]\!]^*_{\mathsf{A}}$ is used here. Precisely, constant functions are used as advice to store the contents of the cell, and function application

(rather than program point labelling) is used to provide dereferencing. Consider the following λ^*_{F} term:

$$\begin{array}{rcl}
\mathsf{F}[\![\texttt{newref}(M)]\!] & \triangleq & \texttt{let} \\
& & \quad \texttt{Init} = \mathsf{F}[\![M]\!] \\
& & \quad \texttt{Cell} = \texttt{fun}\,(\texttt{d} : \texttt{unit})\{\texttt{Init}\} \\
& & \texttt{in}\,\langle \texttt{assign}, \texttt{deref}\rangle \\
\\
\texttt{assign} & \triangleq & \lambda\texttt{Val}.\texttt{around}\ \texttt{Cell}(\texttt{d})\{\texttt{Val}\} \\
\texttt{deref} & \triangleq & \lambda\texttt{d}.\texttt{Cell} \cdot \texttt{skip}
\end{array}$$

This term evaluates the initial value of the cell and creates an advisable function of type $\texttt{unit} \Rightarrow \tau$ which immediately returns it when called. In fact, calling the function is exactly what the dereferencing component does. Assignment installs a new piece of advice which immediately returns the value `Val` that the user wishes to store. Note that because this advice never invokes `proceed`, any previous behaviour of the function is completely ignored. Therefore the next time `Cell` is called, `Val` is returned.

The formal proof proceeds by translating this term into λ^*_{R} and showing that its strategy in the fully abstract game model of [3] responds to each opponent move in the same way as the strategy for $\texttt{newref}(M)$. In fact, this proof is simpler than the corresponding proof in the additive case from chapter 3. To see why, it is useful to examine the term's translate. Recall that, formally, the aspectual expressions in the assignment and dereferencing functions are formally projections of the advisable function `Cell`:

$$\begin{array}{rcl}
\mathsf{F}[\![\texttt{newref}(M)]\!] & \triangleq & \texttt{let} \\
& & \quad \texttt{Init} = \mathsf{F}[\![M]\!] \\
& & \quad \texttt{Cell} = \texttt{fun}\,(\texttt{d} : \texttt{unit})\{\texttt{Init}\} \\
& & \texttt{in}\,\langle \texttt{assign}, \texttt{deref}\rangle \\
\\
\texttt{assign} & \triangleq & \lambda\texttt{Val}.\pi_1(\texttt{Cell}) \cdot (\lambda\texttt{proceed}.\lambda\texttt{d}.\texttt{Val}) \\
\texttt{deref} & \triangleq & \lambda\texttt{d}.\pi_2(\texttt{Cell}) \cdot \texttt{skip}
\end{array}$$

Therefore, assuming inductively that $[\![\mathsf{F}[\![M]\!]]\!]^*_{\mathsf{F}} \simeq M$, the term's translation into λ^*_{R} is

syntactically identical to it except for the translation of the $\texttt{fun}\,(\texttt{d} : \texttt{unit})\{\texttt{Init}\}$ subterm, which is

$$\texttt{let jp} = \texttt{newref}(\lambda\texttt{d}.\texttt{Init})\ \texttt{in}\ \mathscr{F}(\texttt{jp})$$

By definition of $\mathscr{F}$, any attempt to assign a value V to the reference amounts to evaluating

$$\rho := (\lambda p.\lambda x.(\lambda\texttt{proceed}.\lambda\texttt{d}.V) \cdot p \cdot x) \cdot !\rho$$

where ρ is the fresh name generated by the `newref` statement. Now, because `proceed` does not appear in V, the game corresponding to $!\rho$ in the game denotation of the term is never queried. Therefore, the strategy corresponding to the assignment statement above is isomorphic to the strategy for $\rho := \lambda\texttt{d}.V$. In other words, the semantics of `around` advice effectively cause each assigned value – in the form of a constant function which is de-thunked when dereferenced – to replace the previous one as desired. This insight enables the proof of definability, and thus full abstraction:

Proposition 5.3.3 ($[\![-]\!]^*_{\mathsf{F}}$ Definability)**.** *For each λ^*_{R} user term M, there exists a λ^*_{F} term $\mathsf{F}[\![M]\!]$ such that $[\![\mathsf{F}[\![M]\!]]\!]^*_{\mathsf{F}} \simeq M$*

Proof. Appendix C □

Theorem 5.3.4 ($[\![-]\!]^*_{\mathsf{F}}$ Full Abstraction)**.** *$M \simeq N$ if and only if $[\![M]\!]^*_{\mathsf{F}} \simeq [\![N]\!]^*_{\mathsf{F}}$, for any λ^*_{F} user terms M and N*

Proof. By compositionality, adequacy, termination equivalence, definability, and the reasoning of chapter 2. □

5.4 Chapter and Related Work Summary

By using an alternate semantics of `around` advice, a fully abstract semantics of such advice was constructed which:

- Allows the use of the `proceed` keyword an arbitrary number of times in the body of the advice.
- As a result, accounts for the possibility of non-proceeding around advice.
- Can be encoded using only general references (no exceptions)

This interpretation resolves a number of issues arising from the use of the `return` primitive in Walker et al.'s semantics. The price paid for these benefits is that the aspectual language is not oblivious, as its type structure explicitly distinguishes advisable and unadvisable functions, placing the language on an equal footing with other aspect restrictions from the literature.

Specifically, the semantics presented here bear a similarity with those proposed in Aldrich's Open Modules [9]. Both languages, whether by explicit typing in the case of λ^*_{F} or by the restriction operator in Open Modules, give the programmer the choice of exposing join points to advice or not. Aldrich's semantics also implements the `proceed()` keyword as a free variable bound to a continuation of the previous "state" of the given function. The two languages seem to have arrived at the same conclusion – namely that sacrificing obliviousness leads to a much better behaved language – from two different directions. In Aldrich's work, Open Modules was designed to address the issue of modularity in AOP, while λ^*_{F} is an attempt to express pre-emptive advice without using the control primitives of Walker et al.'s semantics, thereby simplifying its denotational model to achieve a full abstraction result. As a result, Open Modules is a much more complete implementation of modules: pointcuts and advice can be arbitrarily hidden

or exposed in nested modules, while λ_F^* simply requires each potential join point to be flatly advisable or not in any scope, but satisfies a very strong theoretical property.

Another language that indirectly yields the same effect is the one analysed by Jagadeesan, Pitcher and Riely [31]. In their case, the language allows pointcut descriptors (i.e. function names) to be locally scoped, thereby allowing raw λ-abstractions to be encoded by "hiding" the function name using the scoping operators. This language also uses similar ideas (partially influenced by μABC [10]) to implement the behaviour of `proceed()`, i.e. by binding it to a variable in the body of advice. The authors define a bisimulation relation which is fully abstract with respect to context equivalence in their language. However, a major difference between their work and ours is that λ_F^* is a typed calculus, whereas the language of [31] is not. The untyped language of Wand et al. [60] is also very similar in spirit to these ideas.

More recently, Clifton and Leavens [13] present an imperative language with aspect oriented features that specifically studies alternative semantics to `proceed()`. They prove a type safety result in their language, and it would be interesting to see if the denotational results presented here can be related to their language.

Chapter 6

Validating Advisable Functions by Example

This chapter aims to show that the advisable function model of aspects can reasonably be used to write useful aspect-oriented programs. It does so by extending λ_F^* to include more complex AOP primitives without sacrificing its theoretical properties. Indeed, the translation λ_F^* into λ_R^* suggests a rather simple strategy for implementing aspect oriented features into imperative functional languages. Such an implementation is used below as a vehicle for demonstrating the utility of the advisable function semantics of aspects. Specifically, λ_F^* is encoded into Standard ML of New Jersey as an ADT effectively defining the type $\sigma \Rightarrow \tau$ introduced previously. The core implementation is then extended with various practical features including:

- the ability to assign `before` and `after` advice in addition to `around` advice
- a more complex pointcut description language which allows disjunctions of function executions
- pointcuts also include a PCD which matches a given function call only if it is called

from within the dynamic scope of another

These extensions are can easily be added to λ^*_F without affecting the full abstraction result. This chapter demonstrates the features of the implementation via a series of examples.

The implementation is set in the functional realm because this is the easiest way to demonstrate the connection between the extended language and the theoretical results. It is therefore unrealistic to expect that it would be useful in a real world software development environment. As a result, no concerted effort has been made to make it efficient, and so would not withstand scrutiny against a run-time analysis. Rather, it is intended as a proof of the concept that advisable functions with in-scope pointcut descriptors can express useful AOP features.

6.1 An Aspect ADT

Central to the implementation is a `structure` called `AOP` with the following signature:

```
type     ('A,'B) Function

datatype ('A,'B) Pointcut
                 = execution  of ('A,'B) Function
                 | fromwithin of ('A,'B) Function * ('A,'B) Function
                 | ||         of ('A,'B) Pointcut * ('A,'B) Pointcut

val newfun : ('A -> 'B)  ->  ('A,'B) Function
val Before : ('A,'B) Pointcut -> ('A -> 'A) -> unit
val After  : ('A,'B) Pointcut -> ('B -> 'B) -> unit
val Around : ('A,'B) Pointcut -> (('A -> 'B) -> 'A -> 'B) -> unit
val @      : ('A,'B) Function * 'A -> 'B
```

This declares an abstract type `('A,'B) Function` of advisable functions from `'A` to `'B`. A value of this type is constructed by calling the `newfun` constructor, which takes an ordinary

function of the appropriate type and produces the corresponding advisable version. The signature also exposes a datatype of pointcuts, which allows the user to construct complex pointcut expressions from advisable functions. A value of type `('A,'B) Pointcut` is a pointcut descriptor which matches a set of program points of type `('A,'B) Function`, and can be constructed using one of the following:

- `execution(f)` matches the period between the point where the the arguments of the advisable function `f` have been evaluated, and the point when the function returns
- `f fromwithin g` is identical to `execution(f)`, but only matches if it occurs within the dynamic context of a call to `g`
- $\texttt{pcd}_1$ `||` $\texttt{pcd}_2$ matches either of the pointcuts $\texttt{pcd}_1$ or $\texttt{pcd}_2$

The other methods in the signature correspond to the accessor methods of the advisable function: `Before`, `After`, and `Around` each take a pointcut and an appropriately typed piece of advice to install. For instance, the advice given to the `Around` function has type

```
('A -> 'B) -> 'A -> 'B
```

corresponding exactly to the type of `around` advice from λ^*_{F}. The advice given to `Before` and `After` correspond to the source and target types of the function being advised respectively, and work similarly to the additive advice of λ^*_{A}. The symbol `@` (used because SML/NJ does not allow overloading of juxtaposition) is an infix operator corresponding to application, taking an advisable function and an argument, and producing the result of calling the function and all of the advice matching that particular call.

6.2 Examples

Suppose for the moment that only the syntactic constructs of λ^*_{F} (i.e. `around` advice on single functions) were included in the language. The signature of the `AOP` structure in such an imple-

mentation would just be:

```
type     ('A,'B) Function
datatype ('A,'B) Pointcut = execution of ('A,'B) Function

val newfun : ('A -> 'B)  ->  ('A,'B) Function
val Around : ('A,'B) Pointcut -> (('A -> 'B) -> 'A -> 'B) -> unit
val @      : ('A,'B) Function * 'A -> 'B
```

In this case, the definition of the abstract type `('A,'B) Function` is precisely the definition of $\sigma \Rightarrow \tau$ from the previous chapter:

```
abstype ('A,'B) Function = mkfun of
                           (('A -> 'B) -> 'A -> 'B) -> unit *
                            ('A -> 'B)
```

This provides a constructor `mkfun` which casts the installation/application pair as an advisable function. The `newfun` function then uses this to create one which behaves as one would expect it to (i.e. a "good" advisable function):

```
fun newfun ( base : 'A -> 'B ) : ('A,'B) Function =
    let
        val joinpt : ('A -> 'B) ref = ref base
    in
        mkfun (
        fn advice => joinpt := (advice (!joinpt)),
        fn input  => (!joinpt) input
        )
    end
```

Note how closely this corresponds to the translation of `fun` $(x : \tau)\{M\}$ into state (short of the extraneous `let` bindings and η expansions used to push the commutativity result through). The implementations of the `Around` and `@` functions are then simply reduced to projections of this type:

```
infix @
fun op @ ( mkfun(install,apply) , input ) = apply input
fun Around execution( mkfun(install,apply) ) advice  = install advice
```

In addition to demonstrating the usage of the additional features of the implementation, the examples below also describe how the basic skeleton above is extended to provide them. They all use I/O as a device to demonstrate the aspectual control flow, but more "computational" programs could easily be constructed.

Example 1: `Before` and `After` advice

The following commands implement the expository term from the beginning of chapter 5:

```
val f = newfun(fn d : unit => print "F");
Around execution(f) (fn proceed => fn d => print "A");
Before execution(f) (fn d => print "B");
f@()
```

If entered into the interpreter after loading and opening the `AOP` structure, these commands define a new advisable function `f`, install two pieces of advice on it, and call the function. The first piece of advice is a non-proceeding `around` that replaces the original function, and the second is an ordinary piece of `before` advice. When the function is called, the body of the `before` advice is executed, and then the body of the `around`, so the output is "`BA`" as expected. Similarly replacing the `before` advice with the proceeding `around` advice

```
Around execution(f) (fn proceed => fn d =>
       print "BefProc";
       proceed();
       print "AftProc"
       );
```

results in the string "`BefProc`", followed by "`A`" (from the body of the non-proceeding around invoked by the call to `proceed`), and then "`AftProc`". In other words, the first advice doesn't interfere with the execution of the second, as it would using the `return` semantics of advice.

The semantics takes the view that the join point corresponding to an advisable function comprises the body of the function itself, and all of the `around` advice surrounding it, and that `before` and `after` advice installed on it is disjoint from the join point. This means that `before` advice executes *strictly before* any attempt is made to execute the `around` advice, and similarly `after` advice executes strictly after all the `around` advice has successfully terminated. This is a design decision which clearly delineates the roles of the various kinds of advice. Indeed, one could instrument "interfering" versions of `before` and `after` advice by simply using `around` advice and the `proceed` keyword. For instance, let `<body>` be a piece of `before` advice of type `'A -> 'A`, then the advice

```
Around <pcd> (fn proceed => fn x => proceed(<body>(x)))
```

is equivalent to a piece of `before` advice which is suppressed if a piece non-proceeding `around` is later installed on a pointcut matching `<pcd>`. A similar technique can be used to encode interfering `after` advice, simply feed the result of `proceed(x)` to the desired advice of type `'B -> 'B`.

Implementing this feature is simply a matter of extending the abstract type to include installation functions for `before` and `after` advice:

```
abstype ('A,'B) Function = mkfun of
                           ('A -> 'A) -> unit *
                           ('B -> 'B) -> unit *
                          (('A -> 'B) -> 'A -> 'B) -> unit *
                           ('A -> 'B)
```

and similarly modifying the definition of `newfun` to provide the new functions, and properly invoking advice when the function is called:

```
fun newfun ( base : 'A -> 'B ) : ('A,'B) Function =
    let
        val pre    : ('A -> 'A) ref = ref (fn x => x)
        val post   : ('B -> 'B) ref = ref (fn x => x)
        val joinpt : ('A -> 'B) ref = ref base
    in
        mkfun (
        fn advice => pre    := (advice o (!pre)),
        fn advice => post   := (advice o (!post)),
        fn advice => joinpt := (advice (!joinpt)),
        fn input  => ((!post) o (!joinpt) o (!pre)) input
        )
    end
```

Finally, the accessor functions `Before`, `After`, `Around`, and `@` are again provided by projecting this type. The code above simply stores the `before` and `after` advice in separate cells, and the join point itself (i.e. the collection of around advice installed on the function) in a third. New `before` and `after` advice is installed as in λ_A^* by functionally composing new advice to the previous contents of the appropriate cell. When called, the application function simply composes the various the contents of the three cells in the correct order and applies the provided input to the resulting function. This extension can straightforwardly be added to λ_F^* without altering the full abstraction result in any way.

Example 2: Disjunctive Pointcuts

The next incremental extension of the language allows a disjunction operator `||` to be used to connect `execution(-)` pointcuts. The implementation adds a clause to the `Pointcut` datatype:

```
datatype ('A,'B) Pointcut
                = execution  of ('A,'B) Function
                | ||         of ('A,'B) Pointcut * ('A,'B) Pointcut
```

As a simple example of its usage, consider a family of advisable functions `PrintI` for `I` $\in$ $\{$`0`$, \cdots,$ `9`$\}$ which print the integer `I` when called:

```
val PrintI = newfun(fn _ : unit => print "I");
```

so the call `Print4@()` prints the string "4" to the output. The programmer may now define a single pointcut `PrintDigit` which matches calls to any functions in this family:

```
val PrintDigit : (unit,unit) Pointcut = (execution(Print0) ||
                                         execution(Print1) ||
                                         execution(Print2) ||
                                         execution(Print3) ||
                                         execution(Print4) ||
                                         execution(Print5) ||
                                         execution(Print6) ||
                                         execution(Print7) ||
                                         execution(Print8) ||
                                         execution(Print9))
```

And subsequently advise the entire collection with a single statement. For instance, installing the following advice results in each call to one of the `PrintI`'s to be followed by a new line:

```
After (PrintDigit) (fn _ => print "\n");
```

Ironically, the implementation of this feature proceeds by a sort of *conjunction* of commands. Precisely, installing advice on a disjunction $\texttt{pcd}_1$ `||` $\texttt{pcd}_2$ just calls the installation function on each pointcut separately:

```
fun After pcd advice =
    case pcd of execution(mkfun(ibef,iaft,ijp,app)) => iaft advice
              | pcdl || pcdr => (After pcdl advice; After pcdr advice)
```

Effectively, a disjunction of `execution()` pointcuts is interpreted as a list of advisable functions, and installing a piece of advice on it maps the appropriate installation function onto the list. This works because the pointcuts in the language are in-scope, and therefore the advice for each function can be locally stored with each of them. A similar extension can therefore be added to λ_F^* using the same idea.

Example 3: Capturing Runtime Predicates

The AOP structure also provides a third pointcut descriptor in the form of an infix operator `fromwithin` which takes two advisable functions as arguments. The pointcut descriptor `F fromwithin G` matches all executions of the function `F` that occur within the dynamic context of a call to `G`. Consider the following advisable function declarations:

```
val F = newfun(fn _ : unit => ( print "Entering F" ));
val G = newfun(fn _ : unit => ( print "Entering G" ; F@() ));
val H = newfun(fn _ : unit => ( print "Entering H" ; G@() ; F@() ));
```

Each of these functions "log" the point where they're called by printing a string containing their name to the output buffer. The first one then does nothing, the second calls the first, and the third calls the second then the first. Now install the following two pieces of advice on `F`:

```
After (F fromwithin G) (fn _ => print "from within G");
After (F fromwithin H) (fn _ => print "from within H");
```

Finally, consider a call to the function `H`. What should the output from evaluating this call be? Ignoring pretty-printing and whitespace issues, the call prints the following:

```
Entering H
Entering G
Entering F from within G from within H
Entering F from within H
```

The reason for the first two lines are obvious. The third line is a result of the call to `G` inside `H`, which calls `F` in turn. Since the last call is executed before `H` and `G` terminate, both pieces of advice are run. The last line is a result of the direct call of `F` from `H`.

In addition to extending the datatype of pointcuts, the implementation of `fromwithin` requires three modifications to the AOP structure. The first adds a boolean flag (and a method to test it) to the advisable function ADT. The second modifies the application function to set and

reset this flag for each function at an appropriate time. The third modifies the exposed installation functions to wrap each piece of advice assigned to `F fromwithin G` inside a test of `G`'s flag.

The changes to the `abstype` and the `Before`, `After`, and `Around` functions is fairly straightforward. Assume a function `ison: ('A,'B) Function -> bool` which tests the flag for the given function (i.e. tests if the function "is on"). The code for the `After` (say) installation function is extended as follows:

```
fun After pcd advice =
     case pcd of ...
               | F fromwithin G => (After (execution(F))
                                          (fn x => if ison(G) then
                                                      advice x
                                                   else x))
```

This installs code on `F` which actually executes the given advice only if the `G` is on *when `F` is called*. If `G` is not on at that point in time, then the advice behaves as the identity.

The second modification, i.e. to the application component in `newfun`, brings up the question of just when the flag should be set. One might be tempted to implement the application function as follows:

```
fn input =>
   let
      val prev = (!imon)
   in
      imon := true;
      let
          val Result = ((!post) o (!join) o (!pre)) input
      in
          imon := prev;
          Result
      end
   end
```

Where `imon` is the reference storing the boolean flag for the function. This code stores the previous value of the flag, then sets it before executing the function with all of its advice as before, and finally resets the flag to its previous value before returning the result.

This implementation conflicts with the principle introduced in example 2. Namely, that `before` and after `after` advice are considered disjoint from the actual join point. In order to remain consistent with this view, the flag must be set immediately after the `before` advice is executed, and reset immediately before the `after` advice is executed. To this end, the implementation used by the ADT is the following one:

```
fn input =>
   let
      val prev = (!imon)
   in
      let
          val BeforeRes = (!pre) input
          val JoinPtRes = (imon := true; !join) BeforeRes
          val AfterRes  = (imon := prev; !post) JoinPtRes
      in
          AfterRes
      end
   end
```

This decision has non-trivial consequences when recursive advisable functions, discussed below, are considered. Note, however, that as usual this feature has been entirely implemented using state. Extending the λ^*_F model with it would only involve making the corresponding changes to the definition of the $\sigma \Rightarrow \tau$ type and the clauses of the translation. These changes would significantly complicate the proofs of commutativity and definability because of the additional structure included in the types, but in principle would not require any significantly new insight.

Example 4: Recursion

The next example concerns itself with the problem of writing a recursive function which exposes every call to advice. This is an issue with this implementation because SML/NJ only allows

values of *ordinary* function type (i.e. whose type uses the `->` connective) to be declared recursive. Therefore, when declaring an advisable function of the form

```
val F = newfun(<init>)
```

the name `F` cannot appear in `<init>` because the `rec` keyword can't be used to indicate that it's recursive. This restriction would seem to make it impossible to define the desired function. Fortunately, despite the SML/NJ developers' well-intentioned attempts to prevent the programmer doing so, there is an easy workaround. The trick is to declare the function as the identity, and then immediately "replace" it using a non-proceeding `around`. For example, here is a fully advisable implementation of the factorial function:

```
val Fact = newfun(fn x : int => x);
Around (execution(Fact))
       (fn p => fn x =>
                if (x = 0) then (1) else (x * (Fact@(x-1))));
```

As expected, a call to `Fact@(n)` does indeed evaluate to `n`! for an integer `n`. Additionally, the programmer is now free to install advice on `Fact` as usual:

```
Before execution(Fact) (fn x : int =>
       print "Calling Fact(" ^ Int.toString(x) ^ ")" ; x)
```

"Under the hood", what's going on here is that general references are being exploited to create a cycle in the store. Calling `Fact@(4)` after installing this advice produces the expected output:

```
Calling Fact(4)
Calling Fact(3)
Calling Fact(2)
Calling Fact(1)
Calling Fact(0)
val it = 24 : int
```

Note that the body of the advice must be sure to return the input value `x` to make sure the calculation remains correct. Interestingly, the programmer can now even define advice on the pointcut `(Fact fromwithin Fact)`, which matches all the recursive calls to `Fact` except the top level one. For example, installing

```
Before (Fact fromwithin Fact)
       (fn x : int =>
               print "from within Fact" ; x)
```

and evaluating `F@(4)` again produces the following output:

```
Calling Fact(4)
Calling Fact(3) from within Fact
Calling Fact(2) from within Fact
Calling Fact(1) from within Fact
Calling Fact(0) from within Fact
val it = 24 : int
```

This kind of pointcut descriptor is very similar to the sort of thing that can be expressed using AspectJ's `cflowbelow` PCD.

Example 5: Integration into Modules

The last example attempts to create, at least structurally, a practical example showing the cross cutting abilities of the `AOP` abstract type implementation. To illustrate them, some of the aspect-oriented features discussed above are integrated into the SML/NJ module system. The base code which will be advised consists of a pair of ML modules representing simple implementations of a two dimensional point, and a line segment respectively. Their signatures are the following:

```
signature Point =
sig
    type Point
    val new : int * int -> Point
    val getX : Point -> int
    val getY : Point -> int
    val setX : (Point * int , unit) Function
    val setY : (Point * int , unit) Function
end;

signature Line =
sig
    type Line
    val new : Point * Point -> Line
    val getP1 : Line -> Point
    val getP2 : Line -> Point
    val setP1 : (Line * Point , unit) Function
    val setP2 : (Line * Point , unit) Function
end;
```

The structures are implemented somewhat naïvely. A `Point` locally stores two integer references, the getter functions simply dereference the corresponding cell, and the setter functions assign to them. The `Line` structure, similarly, locally stores two `Point`s. Here, the setter functions take an entire `Point` structure as input and simply replace the contents of the appropriate location. This semantics was chosen because it most simply illuminates the cross-cuts that `AOP` provides. Particulars of the implementation aside, the most important thing to note about the signatures above is the types of the methods: the setter functions are advisable while the getter functions (and the constructors `Point.new` and `Line.new`) are not.

This shows that a separate structure can be declared which advises functions in both `Point` and `Line`. The program text for each structure can exist in separate files across a possibly large system, and simply fed into the SML/NJ compilation manager, effectively producing an AspectJ-like situation in which the cross cutting concerns are localised. To illustrate, consider the following structure, which installs advice that "logs" calls to the setter functions in the `Point` and `Line` modules:

```
structure LoggingAspect = struct local

   (* Point advice *)
   val _ = Before (execution(Point.setX) || execution(Point.setY))
                  (fn (p,n) => (print "Moving a Point";(p,n)))

   (* Line advice *)
   val _ = Before (execution(Line.setP1) || execution(Line.setP2))
                  (fn (l,p) => (print "Moving a Line";(l,p)))

in val _ = () end;
end;
```

Each piece of installed advice is nothing more than an identity function which prints a string. The manner in which the advice is installed is a bit of an abuse of the module system. Aspects are only installed when the appropriate command is executed[1], and the code above simply wraps these with an empty `structure` so that they are recognised by the compilation manager, which determines that they must be evaluated after the declaration of the `Point`, `Line`, and `AOP` structures. This feature leaves the system in the familiar aspect-oriented scenario depicted in figure 6.1. Once all of these structures are fed into the compilation manager, any call to (say) `Point.setX` results in the update of the given coordinate, and the printing of the string "`Moving a Point`" to the output. However, no `print` commands, or indeed explicit calls to functions which call `print` commands, appear in the program text of `Line` or `Point`. This relatively simple implementation has successfully factored out a cross-cutting concern into a localized file, and in so doing provided one of the major benefits of aspect-oriented programming.

6.3 Summary and Discussion

The `AOP` implementation achieves a significant portion of aspect-oriented functionality, but it still suffers from some important restrictions. Let's enumerate the primary benefits the AO paradigm provides programmers:

[1]Because all the languages in this thesis use dynamically allocated aspects, as opposed to the static lists used by AspectJ and μABC

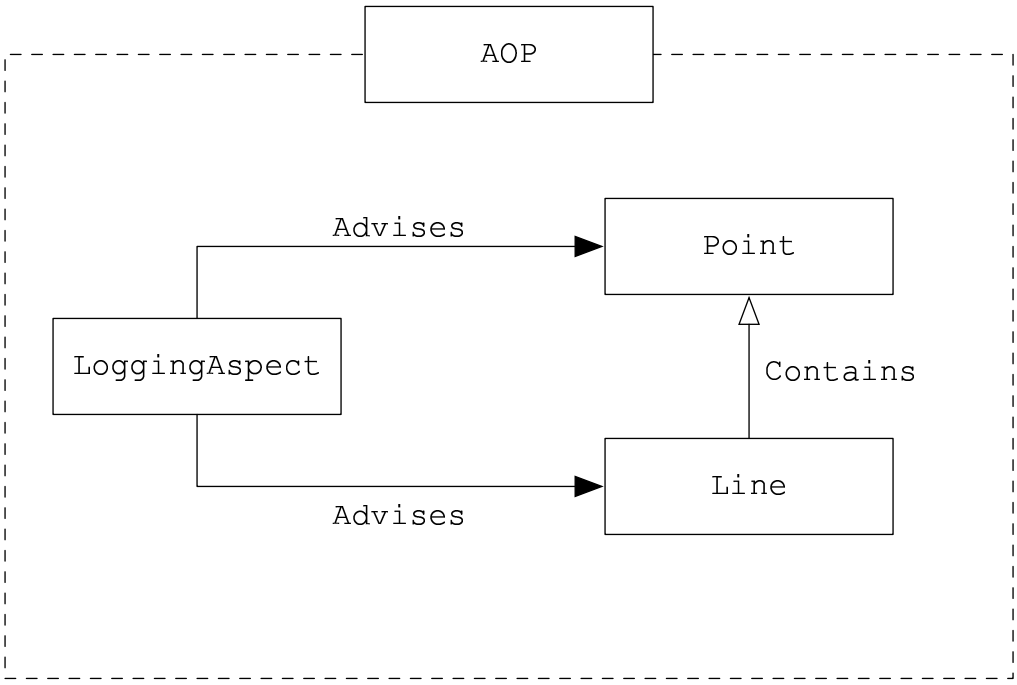


Figure 6.1: *A modular aspect implementation: the diagram shows the logical relationship between the* `structures` *in the example. The logging concern is factored out of the* `Point` *and* `Line` *modules, which make no mention of logging at all. The whole system is executed under the umbrella of the* `AOP` *structure, which is required by all of them.*

Localisation Program instructions relating to cross-cutting concerns are factored out of the base code and encapsulated into their own units. This feature solves the problems discussed in chapter 1 (see figure 1.1).

Conciseness Single pieces of advice can be associated to pointcuts which match many program points. These points may have different types as long as the advice can consistently interact with them. For instance, a piece of side-effecting advice can apply to program points of many types because it does not exchange values with the advised code. This property can be viewed as a more specific statement of the *quantification* of AOP [18].

Forward Applicability Pointcut descriptors may match code that does not yet exist. For example, in AspectJ, the user can use wild cards to create PCDs that match a (potentially infinite) set of method names. If a method is later included into the system that matches it,

then the advice applies equally to the new method. This endows the language with great deal of robustness: the programmer can almost literally implement code at the policy-level which applies to both current and future states of the system.

How does the system introduced in this chapter stand up to scrutiny according to these criteria? As example 5 showed, a localised program structure can be implemented quite successfully in the language as it stands. This is extremely important, as it represents a solution to the problem which originally inspired AOP. The fact that this feature has been implemented using only general references is a pleasing result.

The language also achieves a certain level of conciseness (or quantification), as multiple program points can be formed into a disjunction and associated to a single piece of advice. However, it's performance in this respect is limited, as every element of the PCD must have the same type. Even if advice only executes side-effects, a separate installation instruction must be written for each function type which must be advised. Solving this problem requires the introduction of polymorphism into the language. The extension of the stateful model aspects with polymorphism is left for future work at the moment, but the issues surrounding this extension are discussed in the following chapter in the context of a translation of PolyAML [16].

The last criterion (forward applicability) is virtually non-existent in the current model because the pointcuts used are in-scope. Each point-cut must name the function it advises explicitly, therefore making it impossible to have a PCD match function definitions that are introduced later on. This property is where the languages studied in this thesis truly diverge from "real world" aspect-oriented programming. To achieve this functionality, advice cannot be stored with each function at compile time; the language would need to revert to using a global aspect table which is only queried when a join point is reached.

It is important to distinguish forward applicability with the aforementioned "forward interference" effect encountered when the `return` primitive was used to encode non-proceeding around advice. In the latter case, the control primitive sometimes caused strange effects where only parts of a particular advice body were executed. Forward applicability similarly may result

in the suppression of future code, but the interference is better behaved: a function or piece of advice introduced by a programmer will either entirely run, or entirely not run. Still, a language designer must carefully consider whether such functionality has any true benefit to users, as debugging may require programmers to inspect the entire aspect table to find the cause of errors. While arguments can be made for both sides, the debate will likely not be resolved unless aspects gain wider usage in the development community. It is entirely possible that the benefits of forward applicability may be outweighed by its consequences, and that many programmers find that they prefer an aspectual language which only provides localisation, but satisfies stronger modularity properties.

It should be noted that the similar semantics to those introduced for `proceed()` in the previous chapter (and implemented here) have begun to appear in industrial implementations of AOP. Some links to these implementations are:

1. **Spring AOP** (`http://static.springframework.org`)
2. **AspectWerkz** (`http://aspectwerkz.codehaus.org/`)
3. **Alia** (`http://www.alia4j.org/`)

The work presented in this thesis – and particularly the last two chapters – can be seen as providing some theoretical justification for the ideas implemented in these systems.

Summing up, this chapter has presented an implementation of a variety of aspect features in the framework of an ML abstract type definition. The heart of the implementation was strongly inspired by the semantics of λ^*_{F}, and its subsequent fully abstract translation into general references. In fact, the previous section argues many of the included features can easily be added to λ^*_{F} without affecting the full abstraction result, yielding a model of a useful fragment of AOP. The examples showed that disjunctive pointcuts and even some dynamic effects can be statically captured using general references alone. The resulting language also allows cross cutting concerns to be localised into separate modules, providing an important feature of AO paradigm.

Part IV

Conclusions and Further Directions

Chapter 7

Conclusions and Future Work

The primary contribution of this thesis can be broadly described as a detailed analysis of two predominant semantics for AOP in typed functional languages. The semantic analysis is carried out by compositional translation, and often yields an inherited denotational model of the functional aspect calculus being examined. The conclusions of this analysis seem to indicate that the advisable function interpretation of advice is, at least from a semantics point of view, a preferable one as it is more elegant, has a fully abstract denotational model, and leads to a simple and expressive implementation. While conducting the study, the following specific results were demonstrated:

- A systematic method of constructing adequate translations between functional calculi with imperative locally bound names. By allowing "bad" names in both the source and the target, type preserving translations can be defined, leading to a commutativity result and thus adequacy.

- Adequate translations of both the labelled and advisable function implementations of AOP into languages with higher order references and exceptions, and only higher order store respectively.

- Due the existence of fully abstract game models for general references, the translations of

the additive fragment of the labelled style, and the full advisable function calculus, can be proved fully abstract. The game model of references is therefore inherited by these calculi.

- A sample implementation, intended to validate the advisable function model, showing that various useful aspect oriented features can feasibly be expressed with references alone.

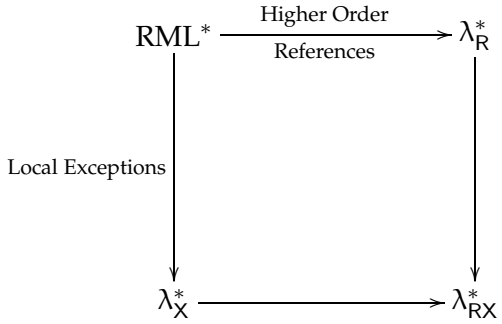
There are still a number of areas in need of further study. For one, the restriction to in-scope pointcuts is a fairly severe one, and an extension of the above techniques to allow a relaxation of this condition is highly desirable. Second, the labelled calculus still has no denotational model. Because of its connection to exceptions and references, the construction of such a model would be of interest not only to AOP researchers, but programming language semanticists in general. Third, the current state of the implementation isn't particularly viable until it has been integrated into a more practical paradigm such as OOP. These and other avenues of possible research are discussed in greater detail below.

7.1 Further Directions in Theory

The first branch of potential study focuses on completing the theoretical work developed here. Concerning the labelled style of part II, the primary concern is the construction of a denotational model for exceptions and general references. Another thrust of theoretical research is the development of denotational models of aspectual calculi with "good" names. Finally, a considerable gap still exists between the languages considered in this thesis and real world AOP languages. The most notable difference is the restriction to in-scope pointcuts which, while allowing aspect lookup to be resolved statically, significantly restricts the programmer's ability to express many useful pointcut predicates concisely.

Denotational Model of the Full Labelled Calculus

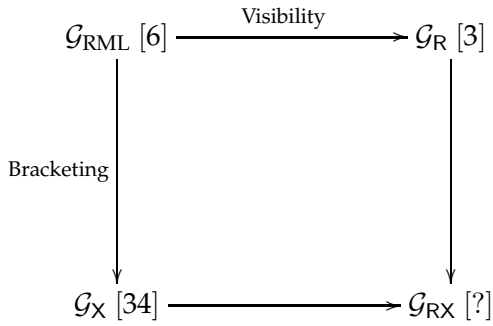
It might be argued that finding a model for λ^*_{L} is unnecessary because the advisable function model is already fully abstract and far simpler than the labelled one. However, the translation results of chapter 4 indicate that it could be achieved by finding a model or λ^*_{RX}, which would of itself be of considerable interest to the semantics community. Indeed, the latter's choice as a target language for the semantics of λ^*_{L} is not an arbitrary one. Control primitives in programming languages come in many different forms. For some of these, fully abstract game semantics models have been developed [34, 35, 38, 37] chiefly by James Laird. However, λ^*_{RX} occupies a very special place in the semantics space: it can be viewed as both an extension of λ^*_{R} with local exceptions, or as an extension of the language studied in [34]. The latter, referred to here as λ^*_{X} following the naming convention used in the rest of the thesis, is identical to λ^*_{RX} except that it only permits ground type references[1]. Furthermore, both λ^*_{R} and λ^*_{X} can be viewed as extensions of Reduced ML (RML*), a typed call-by-value λ calculus extended with a ground store, with bad variables. The relationship between these languages can be represented by the following "syntactic square":



The vertical arrows indicate the extension of the source language with local exceptions *à la* [34], and the horizontal ones represent an extension from ground type references to higher order references. In the traditional game semantics fashion, this diagram corresponds to a so-called

[1] As noted in chapter 4, the language presented in [34] is also a call-by-name language, but a model for the call-by-value version can be constructed using the techniques developed by Abramsky and McCusker [6]

"semantic square":



The nodes here represent fully abstract game models of the languages of the previous diagram, with the citations indicating where they were presented. The arrows represent the relaxation of certain conditions on the models. For instance, the model of RML* is given by strategies of dialogue games which satisfy certain conditions, among them *visibility* and *bracketing*. What these conditions formally mean is inconsequential, the salient point is that by relaxing the visibility condition on the model of RML*, one recovers a fully abstract model of λ^*_{R}: the visibility condition was precisely what was preventing the higher order `newref` from being expressed by the strategies. Similarly, by relaxing the bracketing condition (and adding some structure), one obtains the model of λ^*_{X}. The hope is that by relaxing *both* conditions, one would obtain a model of λ^*_{RX}, and thus by the results from chapter 4, of λ^*_{L} assuming the model confirms the definability conjecture. There is reason for optimism: in similar situations – notably that of PCF, ground type store, and the `call/cc` control operator [6] – game semantics models have been shown to compose in just this way.

Denotational Model of Good Advisable Functions

The fully abstract translations studied in this thesis all require that imperative names be given simple types rather than distinguished ones (i.e. they require "bad" names). When only good names are considered in the source languages, the translations, and hence the inherited game models, only reflect observational equivalence (adequacy), but do not preserve it. Fully abstract models of imperative languages without bad names had been an open problem until the work of James Laird [36], and more recently Nikos Tzevelekos [58]. In fact, [58] presents a fully abstract

game semantics for λ_R by casting the semantics into a monadic framework [46].

Unfortunately, the technique used in this thesis cannot be used to inherit the full abstraction result, as bad names are required to assure that the translation maps are type preserving. However, the results of the previous chapters do show that dynamically allocated aspect oriented languages with in-scope point cuts are nothing more than a special form of storage, or more generally a form of side-effect. Speaking in terms of monads, a stateful computation TB of type B (viewed as the set of terms of that type) is essentially a map from a set of possible stores $\mathcal{S}$ producing a value of type B and an updated store [46]:

$$TB = (B \times \mathcal{S})^{\mathcal{S}}$$

Analogously, an aspect oriented computation of type B takes an aspect table (from a set $\mathcal{A}$ of possible aspect sequences) instead of a traditional store, but similarly produces a value and an updated table:

$$TB = (B \times \mathcal{A})^{\mathcal{A}}$$

In other words, at the monadic level, these computations are identical save for the specific structure of the "store", which is a map from names to values in the case of general references, and a sequence of name/function pairings (i.e. aspects) in the case of AOP. It is therefore likely that a fully abstract model of λ_A or λ_F could be constructed *directly* using techniques similar to those presented in [58].

Beyond In-Scope Pointcuts

Encoding more complicated pointcut descriptors requires a number of alterations to the languages studied in this thesis. For example, a measure of forward applicability can be achieved with regular expressions as pointcut descriptors. In order to correctly resolve aspect invocation, the individual pieces of advice cannot be stored with the functions they advise, as these functions may not exist yet. Similarly, a function may already have advice that matches it at

declaration time, requiring that each function's aspects be populated when they are created. These issues truly separate the static stateful aspects studied here from the ones that need a global aspect table, enabling run-time resolution of the advice that matches each join point.

Adding this feature to MinAML is relatively straightforward: an additional component (such as a string) is stored with each declared function, against which a pattern can be matched. To include it in a model that uses storage cells to model the aspect table is much more complicated: the entire aspect table would somehow need to be stored in a distinguished global cell so that it can be accessed when advice is invoked at run-time. However, since the aspects in the table have different types, some form of polymorphism must be included in the language in order to successfully use this technique.

Another need for polymorphism in AOP is to allow an aspect to advise join points of different types. For example, advice which only executes side effects (such as logging) exchanges no data with its particular invocation points, but the simply typed languages studied so far would still require a different syntactic aspect installation for each type of join point which is to be advised. Ideally, an AOP programmer should be able to write advice of the form:

$$\texttt{before}\ \{*\}\ \texttt{print}\ \langle\texttt{some\_string}\rangle$$

The $\{*\}$ pointcut is a wildcard matching all function calls, but in principle could be any regular expression. The language PolyAML [16] is a polymorphic version of MinAML endowed with the ability to write these and other aspects whose pointcut descriptors match join points of different types. Its definition is given by a translation into a core calculus F_{A}, a polymorphic extension of λ_{A} (cf. ch 2).

In order to extend the translation-based analysis featured in this thesis to such languages, a suitable target language must be constructed which incorporates both polymorphic and stateful features together. Combining these features in a setting where types are inferred is known to be rife with difficulty [57, 40, 55, 39, 23]. However, in a language where terms are explicitly typed (essentially extending λ_{R} with explicitly type abstraction and application from Girard's

System F [21, ch. 11]), some inroads can be made. For a type variable Z, type expression τ and term M possibly containing free occurrences of Z, a term with type $\forall Z.\tau$ is a type abstraction $\Lambda Z.M$ which, when applied to a type σ, requires M to be of type $\tau[\sigma/Z]$ for every possible σ. For example, there are no terms of type $\forall Z.Z$ because there is no single term M that can take an arbitrary type in the proposed language[2]. A positive example is the type $\forall Z.Z \to Z$, which is populated by the polymorphic identity $\Lambda Z.\lambda x : Z.x$.

This last example is of particular relevance to AOP, as all forms of advice studied in this thesis are functions with identical input and output types. Indeed, one can imagine a sequence of polymorphic identity functions (possibly executing side-effects) as advice, each paired with a regular expression as a pointcut. A function call $f(x)$ invoking advice could then be implemented by traversing the aspect table, and passing each advice whose pattern matches f the type of x, and then x itself, passing the result of each executed piece of advice (which will always be x because all of the advice functionally behaves as the identity) to the next. Since the type of f is arbitrary, this procedure successfully implements side-effecting advice which could be executed at call points of different types.

To apply a similar strategy to advice which may manipulate data (i.e. which may do more than just execute side-effects) requires a mechanism whereby types can be examined at run-time (a `typecase` expression). This allows the advice to manipulate its input depending on the type of the join point which invoked it. Crucially, each piece of advice must have a default action in order to insure that all possible input types are accounted for. For example, consider the following type abstraction:

$$
\begin{aligned}
\Lambda Z.\lambda x : Z.\ &\texttt{typecase}\ Z\ \texttt{of} \\
&\texttt{int => } x + 1 \\
&\texttt{bool => false} \\
&\texttt{bool} \times \texttt{int => } \langle \pi_1(x), \pi_2(x) * 3 \rangle \\
&\texttt{default => } x
\end{aligned}
$$

[2]Note that if the local exceptions of λ^*_{RX} were included, the `raise` expression would be such a term

This term can safely be typed as $\forall Z.Z \to Z$ because for any τ to which it is applied (via type application), it returns a correctly typed λ abstraction of type $\tau \to \tau$. Advice of this form is free to branch on, and therefore compute with, as many potential call point inputs as it might like, leaving values of any types that it does not examine unchanged. A `typecase` expression appears in F_A for precisely this reason. In a language with assignment and dereferencing rather than labels, a sequence of such abstractions could, at least in principle, be stored into a distinguished global reference cell of type $(\forall Z.Z \to Z)$ `list` to emulate a polymorphic aspect table.

In sum, a state-based target language would seem to require the following features in order to soundly encode a polymorphic AOPL such as PolyAML:

1. Type abstraction and type application *à la* System F in order to encode polymorphic functions of type $\forall Z.Z \to Z$ which can advise call points of multiple types.

2. Lists used to construct a sequence of abstractions of the above kind in order to encode a dynamically queriable aspect table.

3. Reference cells which are not only higher order, but also able to store type abstractions, so that emulated aspect sequence can be altered at run-time.

4. A `typecase` construct for run time type identification, allowing polymorphic advice to exchange information with the call points which invoke them.

5. A mechanism whereby complex pointcuts can be constructed. This could be as complex as function name strings as a target and full regular expressions as a pattern, or as simple as simply collecting multiple advisable functions in a set.

Once the target language is defined, further work should proceed as follows:

- Some basic properties, such as type safety, would need to be established. One possible strategy to consider here may be to attempt to define a sound translation into F_A, which is already known to be type safe [16]. However, it isn't obvious whether polymorphic labels can be used to encode polymorphic references as in the monomorphic case.

- Encode polymorphic aspects by in turn adequately translating F_A into the language, thereby establishing an alternative semantics of PolyAML by composition with the translation from [16].
- Encode high level polymorphic aspect constructs similar to those of PolyAML directly into the proposed language, essentially extending the analysis of chapter 5. Ideally, the source language should minimise the appearance of polymorphic syntax.
- Explore the possibility of constructing a game semantics model of the target language. This would possibly involve combining the oft-mentioned model of higher order store [3] with the hypergame semantics of Hughes [24] which establishes a fully abstract model for System F

7.2 Further Directions in Practice

While the primary focus of this thesis has been theoretical, the implementation presented in chapter 6, and the connections to game semantics suggest a number of more pragmatic research directions. These include the exploitation of the game models of λ^*_L and λ^*_A for verification and model checking, and also the extension and polishing of the ADT implementation to potentially achieve a realistically usable AOP language.

Formal Verification and Model Checking

One of the major reasons why game semantics has been such a success is due to its connections with model checking [12, 11, 2] and static analysis [41, 42, 22, 43]. In the case of the former, restricting the order of terms in various state-based calculi yields fragments of these languages for which observational equivalence is decidable. For instance, restricting the order of terms in Idealised Algol [51] allows enough structure to be removed from the strategies corresponding to programs to generate pushdown decidable (at third order [48]) or even regular (at second order or below [20]) language semantics. Similar techniques have also been used to attain decidability

results for call-by-value languages [19], and a language with local state [47]. This last result shows that RML* is already undecidable at second order, but can be restricted to a decidable fragment.

Applying these results to the AOP languages presented in this thesis poses a significant challenge. One would need to explore the algorithmic properties of the game model of λ^*_{R}, as all of the above results only apply to imperative languages with ground type stores. The translations of the previous chapters show that join points of order n can be modelled with references of order $n + 1$. It might therefore be tempting to explore the possibility of attaining a decidability result by restricting the order of terms that can be placed in the store: even first order functions would allow `before` and `after` advice to be installed on first order advisable functions. Unfortunately, the factorisation result used to prove definability for the game model of λ^*_{R} [3, Proposition 5] states that a finite number first order cells (in fact cells of type $\texttt{unit} \to \texttt{unit}$) are sufficient to encode a reference cell of arbitrary order. This means that simply restricting the order of the store would not enable any simplification of the game model.

However, that is not to say that no inroads in this area are possible. The argument above only applies to one possible restriction of the models described in the rest of the thesis. It is entirely possible alternative restrictions (for instance, allowing only side-effecting advice) to the aspect languages may lead to far simpler models.

Implementation Improvements

Chapter 6 explored the extent to which aspect-oriented features can be expressed statically with state. The intention of the work was to validate the theoretical models which preceded it by showing that useful aspectual primitives can be expressed using the same ideas. It would be useful to have a more complete picture of just how far statically resolvable aspects could be pushed. For instance, the `fromwithin` point-cut constructor defined in the ADT cannot be used with complex pointcuts such as disjunction. There are also a wealth of AOP features that have yet to be explored: type extensions, conjunctive pointcuts, general dynamic PCDs such as

`cflow()`. Each could potentially be restricted enough to allow for a static semantics.

It is also quite unrealistic to expect the ADT implementation to be useful in a practical software development environment: not only is it embedded in a functional language, but it is also not particularly efficient. However, the fact that the aspects are collated and stored at installation time indicates possible performance gains in certain conditions. Namely, a system where aspects are invoked much more than they are installed could benefit from the static installation schemes proposed in this thesis. In-scope pointcut descriptors allow the language to circumvent the need for dynamic lookup of aspects on every invocation. It would therefore be interesting to implement the scheme in an object-oriented language such as Java, using the OOP features to emulate the store, and compare its performance (in the in-scope setting) with practical languages such as AspectJ to see if they are comparable. It may even be optimistic to expect a complete implementation to perform well against AspectJ, as the latter uses a static aspect list and has been under development for quite some time. Simple restrictions – for example, to side-effecting aspects, which would only require lists of instructions to be maintained in the store rather than function compositions – might however yield some fruitful results.

Part V

Appendices

Appendix A

Formal Language Definitions

A.1 Formal Definition of λ

λ Type System

(TP TT)
$$\frac{}{\Gamma \vdash \mathtt{tt} : \mathtt{bool}}$$

(TP FF)
$$\frac{}{\Gamma \vdash \mathtt{ff} : \mathtt{bool}}$$

(TP COND)
$$\frac{\Gamma \vdash M : \mathtt{bool} \qquad \Gamma \vdash N_1 : \tau \qquad \Gamma \vdash N_2 : \tau}{\Gamma \vdash \mathtt{cond}\ M\ N_1\ N_2 : \tau}$$

(TP VAR)
$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

(TP ABST)
$$\frac{\Gamma, x : \tau \vdash M : \sigma}{\Gamma \vdash \lambda x : \tau . M : \tau \to \sigma}$$

(TP APP)
$$\frac{\Gamma \vdash M : \tau \to \sigma \qquad \Gamma \vdash N : \tau}{\Gamma \vdash M \cdot N : \sigma}$$

(TP PAIR)
$$\frac{\Gamma \vdash M : \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash \langle M, N \rangle : \tau \times \sigma}$$

(TP PROJ$_i$)
$$\frac{\Gamma \vdash M : \tau_1 \times \tau_2}{\Gamma \vdash \pi_i(M) : \tau_i} \quad [i \in \{1, 2\}]$$

λ Big Step Semantics

$$\frac{}{V \Downarrow V}\ (\Downarrow \text{VAL}) \qquad \frac{M \Downarrow \lambda x : \tau.M' \qquad N \Downarrow V' \qquad M[V'/x] \Downarrow V}{M \cdot N \Downarrow V}\ (\Downarrow \text{APP})$$

$$\frac{M \Downarrow V \qquad N \Downarrow U}{\langle M, N\rangle \Downarrow \langle V, U\rangle}\ (\Downarrow \text{PAIR}) \qquad \frac{M \Downarrow \langle V_1, V_2\rangle}{\pi_i(M) \Downarrow V_i}\ (\Downarrow \text{PROJ}_i)$$

$$\frac{M \Downarrow \texttt{tt} \qquad N_1 \Downarrow V}{\texttt{cond}\ M\ N_1\ N_2 \Downarrow V}\ (\Downarrow \text{COND-TT}) \qquad \frac{M \Downarrow \texttt{ff} \qquad N_2 \Downarrow V}{\texttt{cond}\ M\ N_1\ N_2 \Downarrow V}\ (\Downarrow \text{COND-FF})$$

λ Small Step Semantics

$$E ::= [-] \mid \texttt{cond}\ E\ N_1\ N_2 \mid E \cdot N \mid V \cdot E \mid \langle E, N\rangle \mid \langle V, E\rangle \mid \pi_i(E)$$

$$(\lambda x : \tau.M) \cdot V \xrightarrow{\beta} M[V/x]$$
$$\texttt{cond}\ \texttt{tt}\ N_1\ N_2 \xrightarrow{\beta} N_1$$
$$\texttt{cond}\ \texttt{ff}\ N_1\ N_2 \xrightarrow{\beta} N_2$$
$$\pi_i(\langle V_1, V_2\rangle) \xrightarrow{\beta} V_i$$

$$\frac{M \xrightarrow{\beta} M'}{E[M] \longrightarrow E[M']}$$

Theorem A.1.1 (λ Equivalence of Semantics)**.** *$M \Downarrow V$ if and only if $M \longrightarrow^* V$ for any well typed term M of λ, where $\longrightarrow^*$ is the transitive closure of $\longrightarrow$.*

Proof. Standard. □

A.2 Big-Step Semantics of λ_A

To define the evaluation semantics of the additive aspect calculus with good labels, extend the evaluation rules of λ with the rules in the table below. The rules for λ are extended naturally to configurations.

λ_A Big Step Semantics

($\Downarrow$ A-NEWLAB)

$$\frac{}{(L, A, \texttt{newlab}[\tau]) \Downarrow (L[\alpha \mapsto \tau], A, \alpha)} \quad [\alpha \notin \texttt{dom}(L)]$$

($\Downarrow$ A-ASPECT)

$$\frac{M \Downarrow V}{\{M.x \to N\} \Downarrow \{V.x \to N\}}$$

($\Downarrow$ A-INSTALL)

$$\frac{M \Downarrow (L, A, U) \qquad (L, A :: U, N) \Downarrow V}{M \gg N \Downarrow (L, A :: U, V)}$$

($\Downarrow$ A-INVOKE)

$$\frac{M \Downarrow \alpha \qquad N \Downarrow (L, A, V') \qquad M'[V'/x] \Downarrow V}{M\langle\!\langle N \rangle\!\rangle \Downarrow V} \quad [\texttt{lookup}\langle L, A\rangle(\alpha) = \lambda x.M']$$

The proof of the correspondence of the big and small step semantics is just a restriction of the corresponding proof for λ_L^* (presented in the next section), with some trivial modifications to account for the slightly different installation and invocation primitives, and the presence of aspects of the form $\{M.x \to N\}$.

A.3 Big-Step Semantics of λ_L^*

The following extend the rules of λ.

λ_{L}^{*} Big Step Semantics

Let B be a returning or value configuration

$$\frac{}{(L, A, \mathtt{newlab}[\tau]) \Downarrow (L[\alpha \mapsto \tau], A, \alpha)}\ (\Downarrow \textsc{newlab})\ [\alpha \notin \mathtt{dom}(L)]$$

$$\frac{M \Downarrow (L, A, V)}{\alpha \triangleright M \Downarrow (L, A :: \langle \alpha, V \rangle, \mathtt{skip})}\ (\Downarrow \textsc{install}) \qquad \frac{M \Downarrow \mathtt{return}\ V\ \mathtt{to}\ \alpha'}{\alpha\langle\!\langle M \rangle\!\rangle \Downarrow \mathtt{return}\ V\ \mathtt{to}\ \alpha'}\ (\Downarrow \textsc{inv-ret-nok})\ [\alpha \neq \alpha']$$

$$\frac{M \Downarrow (L, A, \mathtt{return}\ V'\ \mathtt{to}\ \alpha) \qquad M'[V'/x] \Downarrow B}{\alpha\langle\!\langle M \rangle\!\rangle \Downarrow B}\ (\Downarrow \textsc{inv-ret-ok})\ [\mathtt{lookup}\langle L, A\rangle(\alpha) = \lambda x.M']$$

$$\frac{M \Downarrow (L, A, V') \qquad M'[V'/x] \Downarrow B}{\alpha\langle\!\langle M \rangle\!\rangle \Downarrow B}\ (\Downarrow \textsc{inv-ok})\ [\mathtt{lookup}\langle L, A\rangle(\alpha) = \lambda x.M']$$

$$\frac{M \Downarrow \alpha}{\pi_1(M) \Downarrow \lambda x.\alpha \triangleright x}\ (\Downarrow \textsc{projlab}_1) \qquad \frac{M \Downarrow \alpha}{\pi_2(M) \Downarrow \lambda f.\alpha\langle\!\langle f \cdot \mathtt{skip} \rangle\!\rangle}\ (\Downarrow \textsc{projlab}_2) \qquad \frac{M \Downarrow \alpha}{\pi_3(M) \Downarrow \lambda x.\mathtt{return}\ x\ \mathtt{to}\ \alpha}\ (\Downarrow \textsc{projlab}_3)$$

λ_{L}^{*} Big Step Semantics Continued

Let D be a returning configuration

$$\frac{M \Downarrow V}{\texttt{return}\ \alpha\ \texttt{to}\ M \Downarrow \texttt{return}\ \alpha\ \texttt{to}\ V}\ (\Downarrow \text{RET})$$

$$\frac{M \Downarrow D}{\texttt{return}\ \alpha\ \texttt{to}\ M \Downarrow D}\ (\Downarrow \text{RET-RET}) \qquad \frac{M \Downarrow D}{\alpha \triangleright M \Downarrow D}\ (\Downarrow \text{RET-INST}) \qquad \frac{M \Downarrow D}{\texttt{cond}\ M\ N_1\ N_2 \Downarrow D}\ (\Downarrow \text{RET-COND})$$

$$\frac{M \Downarrow D}{\pi_1(M) \Downarrow D}\ (\Downarrow \text{RET-PROJ-1}) \qquad \frac{M \Downarrow D}{\pi_2(M) \Downarrow D}\ (\Downarrow \text{RET-PROJ-2}) \qquad \frac{M \Downarrow D}{M \cdot N \Downarrow D}\ (\Downarrow \text{RET-APP-FUNC})$$

$$\frac{M \Downarrow D}{\langle M, N\rangle \Downarrow D}\ (\Downarrow \text{RET-PAIR-1}) \qquad \frac{M \Downarrow U \qquad N \Downarrow D}{\langle M, N\rangle \Downarrow D}\ (\Downarrow \text{RET-PAIR-2}) \qquad \frac{M \Downarrow U \qquad N \Downarrow D}{M \cdot N \Downarrow D}\ (\Downarrow \text{RET-APP-ARG})$$

Theorem A.3.1 (λ_{L}^{*} Equivalence of Semantics)**.** *$M \Downarrow B$ if and only if $M \longrightarrow^{*} B$ for any well typed term M of λ_{L}^{*}, for some final configuration B.*

Proof. The forward direction is by induction on the derivation of $M \Downarrow B$. The cases for the underlying λ calculus are simple extensions of the classic proof to configurations.

Case ($\Downarrow$ NEWLAB)**.** $(L, A, \texttt{newlab}[\tau]) \longrightarrow (L[\alpha \mapsto \tau], A, \alpha)$ in a single step.

Case ($\Downarrow$ INSTALL)**.** $M \longrightarrow^{*} (L, A, V)$ by the induction hypothesis, therefore taking $E[-] \equiv \alpha \triangleright [-]$ we have $E[M] \longrightarrow^{*} (L, A, E[V])$ and by the small step semantics this reduces in one step to

$(L, A :: \langle \alpha, V \rangle, \texttt{skip})$ as desired.

Case ($\Downarrow$ INV-RET-NOK)**.** $M \longrightarrow^* \texttt{return}\ V\ \texttt{to}\ \alpha'$ by the induction hypothesis, therefore taking $E[-] \equiv \alpha\langle\!\langle [-] \rangle\!\rangle$ we have $E[M] \longrightarrow^* (L, A, \alpha\langle\!\langle \texttt{return}\ V\ \texttt{to}\ \alpha' \rangle\!\rangle)$. Since $\alpha \notin \texttt{stack}(E)$ this reduces immediately to $\texttt{return}\ V\ \texttt{to}\ \alpha'$.

Case ($\Downarrow$ INV-RET-OK)**.** $M \longrightarrow^* \texttt{return}\ V'\ \texttt{to}\ \alpha$ by the induction hypothesis, therefore taking $E[-] \equiv \alpha\langle\!\langle [-] \rangle\!\rangle$ we have $E[M] \longrightarrow^* (L, A, \alpha\langle\!\langle \texttt{return}\ V'\ \texttt{to}\ \alpha \rangle\!\rangle)$. By the side condition, $\texttt{lookup}\langle L, A \rangle(\alpha) = \lambda x.M'$, so this reduces to $(L, A, M'[V'/x])$ because $\alpha \in \texttt{stack}(E)$. Then by the induction hypothesis, this reduces to V.

Case ($\Downarrow$ INV-OK)**.** $M \longrightarrow^* (L, A, V')$ by the induction hypothesis, therefore taking $E[-] \equiv \alpha\langle\!\langle [-] \rangle\!\rangle$ we have $E[M] \longrightarrow^* (L, A, \alpha\langle\!\langle V' \rangle\!\rangle)$. By the side condition, $\texttt{lookup}\langle L, A \rangle(\alpha) = \lambda x.M'$, so this reduces to $(L, A, M'[V'/x])$, but by the induction hypothesis, this reduces to V.

Case ($\Downarrow$ PROJLAB$_i$)**.** For each of these rules we have $M \longrightarrow^* \alpha$ by the induction hypothesis, therefore taking $E[-] \equiv \pi_i([-])$ we have $E[M] \longrightarrow^* \pi_i(\alpha)$. Then use the appropriate β rule for $\pi_i \alpha$ to get the desired result.

The cases for the (RET-*) rules are similar, a typical example is presented:

Case ($\Downarrow$ RET-COND-T)**.** $M \longrightarrow^* \texttt{tt}$ by the induction hypothesis, and taking $E[-] \equiv \texttt{cond}\,[-]\,N_1\,N_2$ yields $E[M] \longrightarrow^* \texttt{cond}\ \texttt{tt}\ N_1\ N_2 \longrightarrow N_1$, which by the induction hypothesis on the second premise reduces to D.

The backward direction is by induction on the length of $M \longrightarrow^* B$. For the base cases of length 0 reductions the result is immediate as these correspond to final forms. Otherwise, M is not a value so can be decomposed as $E[N]$ for some evaluation context E. Now proceed by case analysis:

Case ($E[-] = [-]$)**.** In this case the first step in the reduction is a β rule, and so must be one of the following:

Subcase $(L, A, \pi_i(\langle V_1, V_2 \rangle)) \xrightarrow{\beta} (L, A, V_i)$**:** Follows from ($\Downarrow$ PROJ$_i$).

Subcase $(L, A, \pi_i(\alpha)) \xrightarrow{\beta} (L, A, V_i)$**:** Follows from ($\Downarrow$ PROJLAB$_i$).

Subcase $(L, A, \texttt{newlab}[\tau]) \xrightarrow{\beta} (L[\alpha \mapsto \tau], A, \alpha)$ **with** $\alpha \notin \texttt{dom}(L)$**:** Follows from ($\Downarrow$ NEWLAB).

Subcase $(L, A, \alpha \triangleright V) \xrightarrow{\beta} (L, A :: \langle \alpha, V \rangle, \texttt{skip})$**:** Follows from ($\Downarrow$ INSTALL).

Subcase $(L, A, (\lambda x.N) \cdot V) \xrightarrow{\beta} (L, A, N[V/x])$**:** By the induction hypothesis $(L, A, N[V/x]) \Downarrow B$, so the result follows from ($\Downarrow$ APP).

Subcase $(L, A, \texttt{cond tt}\ N_1\ N_2) \xrightarrow{\beta} (L, A, N_1)$**:** By the induction hypothesis $(L, A, N_1) \Downarrow B$, so the result follows from ($\Downarrow$ COND-TT).

Subcase $(L, A, \texttt{cond ff}\ N_1\ N_2) \xrightarrow{\beta} (L, A, N_1)$**:** By the induction hypothesis $(L, A, N_2) \Downarrow B$, so the result follows from ($\Downarrow$ COND-FF).

Subcase $(L, A, \alpha\langle\!\langle V \rangle\!\rangle) \xrightarrow{\beta} (L, A, N[V/x])$ **with** $\texttt{lookup}\langle L, A\rangle(\alpha) = \lambda x.N$**:** The induction hypothesis yields that $(L, A, N[V/x]) \Downarrow B$, so the result follows from rule ($\Downarrow$ INV-OK).

Subcase $(L, A, \alpha\langle\!\langle E'[\texttt{return}\ \alpha\ \texttt{to}\ V] \rangle\!\rangle) \xrightarrow{\beta} (L, A, \alpha\langle\!\langle V \rangle\!\rangle)$ **where** $\alpha \notin \texttt{stack}(E')$**:** The proof is by a sub-induction on the structure of E' showing that $E'[\texttt{return}\ \alpha\ \texttt{to}\ V] \Downarrow \texttt{return}\ \alpha\ \texttt{to}\ V$. The result then follows by applying the induction hypothesis to the rest of the reduction sequence and the rule ($\Downarrow$ INV-RET-OK). For the sub-induction, each case uses the appropriate ($\Downarrow$ RET-*) rule corresponding to E' to draw its conclusion. The base case is when $E'[-] = [-]$ for which the result is immediate. A typical inductive case is where $E'[-] = E''[-] \cdot N$ where the sub-inductive hypothesis yields that $E''[\texttt{return}\ \alpha\ \texttt{to}\ V] \Downarrow \texttt{return}\ \alpha\ \texttt{to}\ V$ because α cannot appear in $\texttt{stack}(E'')$ since it is not in $\texttt{stack}(E')$. The rule ($\Downarrow$ RET-APP-FUN) then yields the result.

Case $(E[-] = \texttt{cond}\ E'[-]\ N_1\ N_2)$**.** In this case $M = \texttt{cond}\ E'[N]\ N_1\ N_2$. Since M terminates then so must $E'[N]$. If the latter results in a returning configuration D, then so does the whole term, and so $D = B$ since otherwise the assumption that $M \longrightarrow^* B$ would be contradicted. In this case, the induction hypothesis yields that $E'[N] \Downarrow B$, and so by rule ($\Downarrow$ RET-COND) $M \Downarrow B$. If $E'[N]$ reduces to a value V, then suppose without losing generality that $V = \texttt{tt}$, then by assumption

we have $N_1 \longrightarrow^* B$. Applying the induction hypothesis to the two reduction sequences we have $E'[N] \Downarrow \mathtt{tt}$ and $N_1 \Downarrow B$, and so by rule ($\Downarrow$ COND-TT) we have $M = \mathtt{cond}\ E'[N]\ N_1\ N_2 \Downarrow B$ as required.

Case ($E[-] = V \cdot E'[-]$)**.** $M = V \cdot E'[N]$, and by typing we know that $V = \lambda x.N'$. Similar to the above, we know that $E'[N]$ reduces to some final form D which is either a value V' or a returning configuration $D = B$. In the latter case apply the induction hypothesis to this reduction and use rule ($\Downarrow$ RET-APP-ARG) to conclude the result. For the former case, we also know that $N'[V'/x] \longrightarrow^* B$, so applying the induction hypothesis to this reduction and the evaluation of $E'[N]$ to V' allows rule ($\Downarrow$ APP) to be used to yield the result.

Case ($E[-] = E'[-] \cdot N$)**.** Here we have $M = E'[M'] \cdot N$. We know that $E'[M']$ must either reduce to a value V or to a returning configuration $D = B$. If a returning configuration, the result follows by applying the induction hypothesis to this reduction sequence and using ($\Downarrow$ RET-APP-FUNC). If a value, then use the induction hypothesis on the reduction and proceed as in the previous case.

Case ($E[-] = \langle V, E'[-]\rangle$)**.** We have $M = \langle V, E'[N]\rangle$ and $E'[N] \longrightarrow^* B'$ for some final form B'. If this is a returning configuration, then it must be B and the result follows from the induction hypothesis and rule ($\Downarrow$ RET-PAIR-2). Otherwise, use the induction hypothesis and rule ($\Downarrow$ PAIR).

Case ($E[-] = \langle E'[-], N\rangle$)**.** We have $M = \langle E'[N'], N\rangle$ so $E'[N']$ must reduce to a final form as usual. If this is a returning configuration it must be B, so use the induction hypothesis and rule ($\Downarrow$ RET-PAIR-1). If it is a value, then use the induction hypothesis on the reduction and proceed as in the above case.

Case ($E[-] = \pi_i(E'[-])$)**.** We have $M = \pi_i(E'[N])$, so $E'[N]$ must reduce to a final form. If it is a returning configuration, it must be B, therefore the result follows from the induction hypothesis and rule ($\Downarrow$ RET-PROJ-i). If it is a value, then it is of the form $\langle V_1, V_2\rangle$, and $B = V_i$ so use the induction hypothesis and rule ($\Downarrow$ PROJ$_i$).

Case ($E[-] = \alpha \triangleright E'[-]$)**.** We have $M = \alpha \triangleright E'[N]$ and $E'[N]$ must evaluate to a final form by assumption. If this is a returning configuration then it must be B and the result follows from the induction hypothesis and rule ($\Downarrow$ RET-INST). If it is a value V, then $B = (L, A :: \langle \alpha, V \rangle, \texttt{skip})$, and the result follows by using the induction hypothesis and rule ($\Downarrow$ INSTALL).

Case ($E[-] = \alpha\langle\!\langle E'[-] \rangle\!\rangle$)**.** We have $M = \alpha\langle\!\langle E'[N] \rangle\!\rangle$ and that $E'[N]$ evaluates to a final form. If this is a returning configuration $\texttt{return}\ \alpha'\ \texttt{to}\ V'$, there are 2 subcases:

1. if $\alpha \neq \alpha'$ then the reasoning is similar to other cases, i.e. $B = \texttt{return}\ \alpha'\ \texttt{to}\ V'$ and the result follows from the induction hypothesis and rule ($\Downarrow$ INV-RET-NOK).

2. if $\alpha = \alpha'$ then the next step in the reduction goes to $N'[V'/x]$ where $\texttt{lookup}\langle L, A \rangle(\alpha) = \lambda x.N'$, and by assumption this must reduce to B. Applying the induction hypothesis to the reduction of $E'[N]$ and $N'[V'/x]$ and rule ($\Downarrow$ INV-RET-OK) yields the result.

If $E'[N]$ results in a value V', then the next reduct is $N'[V'/x]$ where $\texttt{lookup}\langle L, A \rangle(\alpha) = \lambda x.N'$, which in turn must evaluate to B. Using the induction hypothesis on the reductions of $E'[N]$ and $N'[V'/x]$ and rule ($\Downarrow$ INV-OK) yields the desired result.

Case ($E[-] = \texttt{return}\ E'[-]\ \texttt{to}\ \alpha$)**.** We have $M = \texttt{return}\ E'[N]\ \texttt{to}\ \alpha$ so $E'[N]$ must evaluate to a final form. If this is a returning configuration, it must be B by the same reasoning as the previous cases, and rule ($\Downarrow$ RET-RET) and the induction hypothesis yields the result. If it is a value, then the induction hypothesis and rule ($\Downarrow$ RET) yields the result.

□

Appendix B

Commutativity Proofs

The proofs in this section use the big-step versions of the semantics for λ_{A}, λ^*_{A} and λ^*_{L} because it simplifies the presentation. The big-step semantics for λ_{A} and λ^*_{A}, and the corresponding correspondence proof with the small-step semantics, restrict trivially from those for λ^*_{L} presented in appendix A.3.

B.1 Commutativity for $[\![-]\!]_{\mathsf{A}}$

The proof proceeds in two stages, corresponding to the proofs of propositions 3.3.3 and 3.3.4 respectively.

B.1.1 Proof of Proposition 3.3.3

We wish to prove that for any λ_{A} configuration (L, A, M), if $(L, A, M) \Downarrow (L', A', V)$, then $[\![L, A, M]\!]_{\mathsf{L}} \Downarrow [\![L', A', M]\!]_{\mathsf{L}}$ in λ^*_{R}. The proof is by induction on the derivation of the antecedent. The base case of value configurations follows immediately from the fact that the translation preserves values, i.e. proposition 3.2.1(b). Cases for the rules stemming from λ follow easily from the induction hypothesis, here is the case for application:

Case ($\Downarrow$ APP)**.** Apply the induction hypothesis to the premises of this rule to get derivations of

$[\![M]\!] \Downarrow \lambda x.[\![M']\!]$, $[\![N]\!] \Downarrow [\![V']\!]$, and $[\![M'[V'/x]]\!] \Downarrow [\![V]\!]$. Apply property S.2 (proposition 3.3.2) to the last of these to deduce that $[\![M']\!][[\![V']\!]/x] \Downarrow [\![V]\!]$. Combine this derivation with the first two derivations, and use the ($\Downarrow$ APP) rule in λ^*_{R} to yield the result.

The remaining cases are label creation, aspect installation, aspects, and join points. The case for rule ($\Downarrow$ A-ASPECT) follows immediately from the inductive hypothesis as above, the remaining cases are presented below:

Case ($\Downarrow$ A-NEWLAB)**.** Follows routinely from the definition of the translation and the operational semantics of the two languages, simply check that the property is satisfied.

Case ($\Downarrow$ A-INSTALL)**.** The induction hypothesis yields λ^*_{R} derivations of $[\![M]\!] \Downarrow ([\![L]\!], [\![L, A]\!], [\![U]\!])$ and $([\![L]\!], [\![L, A :: U]\!], [\![N]\!]) \Downarrow [\![V]\!]$. Suppose without losing generality that $U = \{\alpha.x \to M'\}$, so $[\![U]\!] = \langle \mathscr{L}(\rho_\alpha), \lambda x.[\![M']\!] \rangle$. By definition, $\texttt{lookup}\langle L, A :: U\rangle(\alpha) = \texttt{lookup}\langle L, A\rangle(\alpha) \circ \lambda x.M'$, and so by the translation $[\![L, A :: U]\!](\rho_\alpha) = [\![\texttt{lookup}\langle L, A\rangle(\alpha)]\!] \circ \lambda x.[\![M']\!]$. Note that this is precisely the result of assigning $\lambda x.[\![M']\!]$ to $\mathscr{L}(\rho_\alpha)$ in environment $[\![L, A]\!]$, so we have

$$\pi_1([\![U]\!]) := \pi_2([\![U]\!]); [\![N]\!] \Downarrow [\![V]\!]$$

and since $[\![U]\!]$ is already a value by proposition 3.2.1(b), it can be `let` abstracted in the assignment without affecting the evaluation, yielding

$$\big[\texttt{let}\ x = [\![U]\!]\ \texttt{in}\ \pi_1(x) := \pi_2(x)\big]; [\![N]\!] \Downarrow [\![V]\!]$$

Finally, since $[\![M]\!] \Downarrow [\![U]\!]$, we can conclude that

$$\big[\texttt{let}\ x = [\![M]\!]\ \texttt{in}\ \pi_1(x) := \pi_2(x)\big]; [\![N]\!] \Downarrow [\![V]\!]$$

which is precisely the desired derivation of $[\![M \gg N]\!] \Downarrow [\![V]\!]$

Case ($\Downarrow$ A-INVOKE)**.** The induction hypothesis yields derivations of $[\![M]\!] \Downarrow \mathscr{L}(\rho_\alpha)$, $[\![N]\!] \Downarrow ([\![L]\!], [\![L, A]\!], [\![V']\!])$, and $[\![M'[V'/x]]\!] \Downarrow [\![V]\!]$. Now note that dereferencing $\mathscr{L}(\rho_\alpha)$ yields $\lambda x.!\rho_\alpha \cdot x$, but then substitut-

ing $[\![V']\!]$ in the body yields $[\![M']\!][[\![V']\!]/x]$ because by the definition of the translation and the side condition, $[\![L, A]\!](\rho_\alpha) = \lambda x.[\![M']\!]$. Use this together with the derivations above and property S to construct a derivation of $![\![M]\!] \cdot [\![N]\!] \Downarrow [\![V]\!]$ which is exactly the required derivation of $[\![M\langle\!\langle N \rangle\!\rangle]\!] \Downarrow [\![V]\!]$

B.1.2 Proof of Proposition 3.3.4

We want to show that for any λ_{A} configuration M, if $[\![M]\!]_{\mathsf{L}} \Downarrow V$ in λ^*_{R}, then $M \Downarrow U$ in λ_{A} such that $[\![U]\!]_{\mathsf{L}} = V$. We proceed by induction on the derivation of $[\![M]\!] \Downarrow V$. The base case occurs when M is a value, and is trivially true by Proposition 3.2.1(b). As in the previous proof, the rules of the underlying language λ follow relatively straightforwardly from the induction hypothesis, the most complex case being application (which requires the substitution lemma):

Case ($\Downarrow$ APP)**.** The induction hypothesis on the first two premises implies that $[\![M]\!]$ must evaluate to $\lambda x.[\![M']\!]$ for some λ_{A} term M', that $[\![N]\!]$ evaluates to $[\![V']\!]$ for some λ_{A} value V', and that $M \Downarrow \lambda x.M'$ and $N \Downarrow V'$ in λ_{A}. For the third premise, note that since the original term was well typed in λ_{A}, then $M[V'/x]$ must also be well-typed. Now apply property S.2 to yield that $[\![M']\!][[\![V']\!]/x] = [\![M'[V'/x]]\!]$, and apply the induction hypothesis to the third premise to conclude that the substitution must evaluate to $[\![V]\!]$ for some λ_{A} term V, and that $M'[V'/x] \Downarrow V$ is derivable in λ_{A}. Use these derivations as premises of rule ($\Downarrow$ APP) in λ_{A} to construct the desired derivation of $M \cdot N \Downarrow V$.

The remaining cases are those enumerated by the rules added to λ to define λ_{A}. The rule ($\Downarrow$ A-ASPECT) follows easily from the induction hypothesis, leaving the cases for the creation of new labels, and installing and invoking advice.

Case ($\Downarrow$ A-NEWLAB)**.** Simply translate $(L, A, \texttt{newlab}[\tau])$ and evaluate it in λ^*_{R} and check that the result is the translate of evaluating the λ_{A} configuration.

Case ($\Downarrow$ A-INSTALL)**.** Assume the antecedent and evaluate the translate of $M \gg N$ to get a

derivation of:

$$\mathcal{E} \equiv \big[\texttt{let}\ x = [\![M]\!]\ \texttt{in}\ \pi_1(x) := \pi_2(x)\big]; [\![N]\!] \Downarrow V$$

for some value configuration V. The first command in the sequence (i.e. $\texttt{let}\ x = \cdots$) was derived from an evaluation of $[\![M]\!]$. Apply the induction hypothesis to this derivation, yielding that it evaluates to a value configuration $([\![L]\!], [\![L, A]\!], [\![U]\!])$ which is the translate of a λ_{A} aspect U, and that $M \Downarrow (L, A, U)$. Supposing (wlog) that $U = \{\alpha.x \to M'\}$, then the body of the $\texttt{let}$ command amounts to an evaluation of

$$([\![L]\!], [\![L, A]\!], \mathscr{L}(\rho_\alpha) := \lambda x.[\![M']\!]) \Downarrow ([\![L]\!], [\![L, A]\!][\rho_\alpha \mapsto [\![\texttt{lookup}\langle L, A\rangle(\alpha)]\!] \circ \lambda x.[\![M']\!]], \texttt{skip})$$

by the definition of $\mathscr{L}(-)$. This then, is the environment under which $[\![N]\!]$ must have been evaluated in order to yield the derivation $\mathcal{E}$. By the definition of the translation, the λ^*_{R} configuration

$$([\![L]\!], [\![L, A]\!][\rho_\alpha \mapsto [\![\texttt{lookup}\langle L, A\rangle(\alpha)]\!] \circ \lambda x.[\![M']\!]], [\![N]\!])$$

is precisely $[\![L, A :: U, N]\!]$, so apply the induction hypothesis to its evaluation to yield that it results in some value configuration $V = [\![V']\!]$, and that $(L, A :: U, N) \Downarrow V'$. Use this with the derivation of $M \Downarrow (L, A, U)$ to construct the desired derivation of $M \gg N \Downarrow V'$ using rule ($\Downarrow$ A-INSTALL).

Case ($\Downarrow$ A-INVOKE)**.** Assuming the antecedent, translate $M\langle\!\langle N \rangle\!\rangle$ and evaluate it in λ^*_{R} to yield the following derivation:

$$\mathcal{E} \equiv\ ![\![M]\!] \cdot [\![N]\!] \Downarrow V$$

for some value V. This must have been derived from a derivation that $![\![M]\!]$ evaluates to some lambda term G, and this must have in turn been derived from an evaluation of $[\![M]\!]$. Apply the induction hypothesis to the latter to deduce that $[\![M]\!] \Downarrow \mathscr{L}(\rho_\alpha)$ and that $M \Downarrow \alpha$ for some label α. The second premise of the evaluation of $![\![M]\!]$ must therefore have been the application of $\texttt{skip}$ to the second component of $\mathscr{L}(\rho_\alpha)$ (since dereferencing is formally just a projection), so

$G = \lambda x.!\rho_\alpha \cdot x$ by definition of $\mathscr{L}(-)$. Next, apply the induction hypothesis to the subevaluation of $[\![N]\!]$ to yield that $[\![N]\!] \Downarrow ([\![L]\!], [\![L, A]\!], [\![V']\!])$ for some λ_{A} value V', and that $N \Downarrow (L, A, V')$. $\mathcal{E}$'s third premise must therefore be an evaluation of $!\rho_\alpha \cdot [\![V']\!] \Downarrow V$. This means that ρ_α must contain a lambda term in the store $[\![L, A]\!]$, and by the translation this must be $[\![\texttt{lookup}\langle L, A\rangle(\alpha)]\!]$, i.e. $[\![L, A]\!](\rho_\alpha) = \lambda x.[\![M']\!]$ and

$$\mathcal{D} = [\texttt{lookup}\langle L, A\rangle(\alpha) = \lambda x.M']$$

The third premise must therefore have itself been derived from a derivation of $[\![M']\!][[\![V']\!]/x] \Downarrow V$. Using property S.2 and applying the induction hypothesis yields that $V = [\![U]\!]$ for some λ_{A} value U such that $M'[V'/x] \Downarrow U$. Use this in conjunction with the evaluations of M and N, and the side condition $\mathcal{D}$ to construct a derivation of $M\langle\!\langle N \rangle\!\rangle \Downarrow U$.

B.2 Commutativity for $[\![-]\!]^*_{\mathsf{A}}$ (Proposition 3.4.2)

The proof of commutativity for the calculus of bad additive aspects is carried out in a similar fashion to the corresponding proof for λ_{A}, i.e. by proving preservation and reflection of the evaluation relation separately. The cases for the λ calculus are identical, so the proofs below only present the inductive cases for the modified aspect constructs present in λ^*_{A}, and the cases for projections over label names.

Preservation of Evaluation (Property E $\Rightarrow$)

We show that for any λ^*_{A} configuration M, if $M \Downarrow V$ for some value configuration V, then $[\![M]\!]^*_{\mathsf{A}} \Downarrow [\![V]\!]^*_{\mathsf{A}}$ in λ^*_{R}. As usual, we proceed by induction on the derivation of $M \Downarrow V$, with the case for the rule ($\Downarrow$ NEWLAB) being identical to the case for new label creation in the previous proof (i.e. following routinely from the definition of the translation and the operational semantics of the languages). Below, we present the cases for the remaining aspectual rules, which are taken as the appropriate subset of the big step rules of λ^*_{L}.

Case ($\Downarrow$ INSTALL)**.** Recall that

$$\begin{aligned}[\![\alpha \triangleright M]\!] \quad \triangleq \quad & \texttt{let NewAdvice} = [\![M]\!] \texttt{ in} \\ & \texttt{let OldAdvice} = !\rho_\alpha \texttt{ in } \rho_\alpha := \texttt{NewAdvice} \circ \texttt{OldAdvice}\end{aligned}$$

The induction hypothesis states that $[\![M]\!] \Downarrow ([\![L]\!], [\![L, A]\!], [\![V]\!])$, and by the definition of the translation we know that $[\![L, A]\!](\rho_\alpha) = [\![\texttt{lookup}\langle L, A\rangle(\alpha)]\!]$. Therefore by the operational semantics of λ^*_{R}, the above term evaluates to

$$([\![L]\!], [\![L, A]\!][\rho_\alpha \mapsto ([\![V]\!] \circ [\![\texttt{lookup}\langle L, A\rangle(\alpha)]\!])], \texttt{skip})$$

Now observe that this is precisely a translation of $(L, A :: \langle \alpha, V\rangle, \texttt{skip})$ as required.

Case ($\Downarrow$ INV-OK)**.** By the translation, we have that $[\![\alpha\langle\!\langle M\rangle\!\rangle]\!] = \texttt{let Base} = [\![M]\!] \texttt{ in } !\rho_\alpha \cdot \texttt{Base}$,

and the induction hypothesis states that $[\![M]\!] \Downarrow ([\![L]\!], [\![L, A]\!], [\![V']\!])$. Further, we know that the contents of ρ_α in $[\![L, A]\!]$ is the translate of a λ term $\lambda x.M' = \texttt{lookup}\langle L, A\rangle(\alpha)$. Therefore the evaluation of $[\![\alpha\langle\!\langle M\rangle\!\rangle]\!]$ has a subderivation of $(\lambda x.[\![M']\!]) \cdot [\![V']\!]$, which in turn has a derivation of $[\![M']\!][[\![V']\!]/x]$ as one of its premises. Use property S and the induction hypothesis on the second premise of ($\Downarrow$ INV-OK) to deduce that the latter evaluates to $[\![V]\!]$, and use this to construct the desired derivation of $[\![\alpha\langle\!\langle M\rangle\!\rangle]\!] \Downarrow [\![V]\!]$.

Case ($\Downarrow$ PROJLAB$_1$)**.** By the translation we know that $[\![\pi_1(\alpha)]\!] = \pi_1(\mathscr{L}^*_{\mathsf{A}}(\rho_\alpha))$, which by the definition of $\mathscr{L}^*_{\mathsf{A}}$ evaluates to

$$\lambda\texttt{x.let New} = \texttt{x in let Old} = !\rho_\alpha \texttt{ in } \rho_\alpha := \texttt{New} \circ \texttt{Old}$$

Now simply verify that this is exactly the translation of $\lambda x.\alpha \triangleright x$, the result of evaluating $\pi_1(\alpha)$ in λ^*_{A}. Note that the `let` bindings in the above term are crucial to achieve this result.

Case ($\Downarrow$ PROJLAB$_2$)**.** The translation tells us that $[\![\pi_2(\alpha)]\!] = \pi_2(\mathscr{L}^*_{\mathsf{A}}(\rho_\alpha))$, which by definition of $\mathscr{L}^*_{\mathsf{A}}$ evaluates to

$$\lambda\texttt{d.}\lambda\texttt{x.let Base} = \texttt{x in } !\rho_\alpha \cdot \texttt{Base}$$

As above, one can routinely check that this is precisely the translation of $\lambda d.\lambda x.\alpha\langle\!\langle x\rangle\!\rangle$, the result of evaluating $\pi_2(\alpha)$ in λ^*_{A}.

Reflection of Evaluation (Property E $\Leftarrow$)

We now show the converse result, namely that for any λ^*_{A} configuration M, if $[\![M]\!]^*_{\mathsf{A}} \Downarrow V$ then $M \Downarrow U$ such that $[\![U]\!]^*_{\mathsf{A}} = V$. As for the corresponding proof for λ_{A}, we proceed by induction on the derivation of $[\![M]\!]^*_{\mathsf{A}} \Downarrow V$ where the cases are enumerated by the structure of M. We present the same cases as the proof of preservation of evaluation, as the remaining cases are straight forward.

Case ($M \equiv \alpha \triangleright N$)**.** By supposition we know that

$$\begin{aligned}[\![\alpha \triangleright N]\!] \quad \triangleq \quad & \texttt{let NewAdvice} = [\![N]\!] \texttt{ in} \\ & \texttt{let OldAdvice} = !\rho_\alpha \texttt{ in } \rho_\alpha := \texttt{NewAdvice} \circ \texttt{OldAdvice}\end{aligned}$$

evaluates to some value configuration V. Using the induction hypothesis on the subevaluation of $[\![N]\!]$, we retrieve derivations of $N \Downarrow (L, A, U)$ and $[\![N]\!] \Downarrow ([\![L]\!], [\![L, A]\!], [\![U]\!])$. By the definition of the translation we know that ρ_α contains $[\![\texttt{lookup}\langle L, A\rangle(\alpha)]\!]$ in $[\![L, A]\!]$, therefore

$$V \equiv ([\![L]\!], [\![L, A]\!][\rho_\alpha \mapsto [\![U]\!] \circ [\![\texttt{lookup}\langle L, A\rangle(\alpha)]\!]], \texttt{skip})$$

Now use the evaluation of N in λ^*_{A} to construct a derivation of $M \Downarrow (L, A :: \langle \alpha, U\rangle, \texttt{skip})$ using rule ($\Downarrow$ INSTALL) and verify that V is precisely the translate of this configuration as desired.

Case ($M \equiv \alpha\langle\!\langle N \rangle\!\rangle$)**.** Here, the supposition states that

$$[\![\alpha\langle\!\langle N \rangle\!\rangle]\!] \quad \triangleq \quad \texttt{let Base} = [\![N]\!] \texttt{ in } !\rho_\alpha \cdot \texttt{Base}$$

evaluates to some value configuration V. Again, using the induction hypothesis on the subderivation of $[\![N]\!]$ yields that $[\![N]\!] \Downarrow ([\![L]\!], [\![L, A]\!], [\![V']\!])$ in λ^*_{R} and $N \Downarrow (L, A, V')$ in λ^*_{A}. Therefore the contents of ρ_α in $[\![L, A]\!]$ must be the translate of some λ^*_{A} term $\lambda x.M'$ (since $[\![L, A]\!]$ is itself a translate). By definition of the translation this is also $[\![\texttt{lookup}\langle L, A\rangle(\alpha)]\!]$, and so we must have that $\texttt{lookup}\langle L, A\rangle(\alpha) = \lambda x.M'$. Now, a subderivation of the application in the above term must therefore be an evaluation of $[\![M']\!][[\![V']\!]/x] \Downarrow V$. Applying the substitution property and the induction hypothesis to this derivation, we obtain a derivation of $M'[V'/x] \Downarrow U$ in λ^*_{A} such that $[\![U]\!] = V$. Therefore use this derivation, together with the evaluations of N and the deduction that $\texttt{lookup}\langle L, A\rangle(\alpha) = \lambda x.M'$ to form the premises and side condition of rule ($\Downarrow$ INV-OK), and construct the required derivation of $\alpha\langle\!\langle N \rangle\!\rangle \Downarrow U$.

Case ($M \equiv \pi_i(\alpha)$)**.** For each of these three cases, it simply suffices to note that $[\![\pi_i(\alpha)]\!] = \pi_i(\mathscr{L}^*_{\mathsf{A}}(\rho_\alpha))$ evaluates to some λ term using the λ^*_{R} rules, and to verify that this term is indeed

simply the translation of the λ term to which $\pi_i(\alpha)$ evaluates in λ^*_{A}. This triviality is a direct consequence of the definition of $\mathscr{L}^*_{\mathsf{A}}$, which uses seemingly unnecessary η expansions and `let` bindings enabling the result.

B.3 Commutativity for $[\![-]\!]^*_{\mathsf{L}}$ (Proposition 4.3.1)

This translation takes configurations of λ^*_{L} to those of a calculus λ^*_{ML} of general references and ML-style exception constructs. Recall that the distinguishing features of these exceptions are the following:

- The exceptions carry values, thus the `raise` expression takes 2 arguments, and a successfully handled exception evaluates to the value it carries.
- The `handle` expression allows the expression it wraps to evaluate normally.

We prove that for any λ^*_{L} configuration C, $C \longrightarrow^* C'$ if and only if $[\![C]\!]\ (\Downarrow \cup \uparrow)\ [\![C']\!]$. In other words, the result of translating C then evaluating it is the same as the result of first evaluating it and translating the result. We proceed, as in previous commutativity proofs, by induction on the derivation of the evaluation of C in the forward direction (preservation of evaluation), and by induction on the derivation of the evaluation of $[\![C]\!]$ in the reverse direction (reflection of evaluation). Since the cases for both directions are enumerated by the structure of the term component of C, we prove both directions of the equivalence simultaneously in each case to preserve space. We use the big step operational semantics of λ^*_{L} to prove the result, as these make the presentation of the proof much simpler.

The cases for the underlying λ calculus are straight forward, and generally follow fairly directly from the induction hypothesis. The most complex case is that of application, and is performed as follows:

Case ($M = M_1 \cdot M_2$)**.** For the forward direction, we know that $C \Downarrow C'$ must have been derived from one of the rules ($\Downarrow$ RET-APP-FUNC), ($\Downarrow$ RET-APP-ARG), or ($\Downarrow$ APP). For the first two, we apply the induction hypothesis to the premises of the corresponding rule to get either $[\![M_1]\!] \uparrow [\![D]\!]$ in the first case, or $[\![M_1]\!] \Downarrow [\![U]\!]$ and $[\![M_2]\!] \uparrow [\![D]\!]$ in the second case. Either way, use these derivations to derive that $M_1 \cdot M_2 \uparrow [\![D]\!]$ using the corresponding exception raising rule, recalling that a returning configuration translates to an error configuration. The same technique can be applied

in the case where the application was derived using ($\Downarrow$ APP), deriving $[\![C]\!] \Downarrow [\![C']\!]$ or $[\![C]\!] \uparrow [\![C']\!]$ depending on whether the original evaluation resulted in a value or a `return` expression.

For the reverse direction, we know that $[\![C]\!]$ $(\Downarrow \cup \uparrow)$ B. If B is an error configuration, then we know that the derivation came from either rule ($\uparrow$ PROP-APP$_1$), ($\uparrow$ PROP-APP$_2$), or ($\Downarrow$ APP) where the last premise raises B. In the first case, the induction hypothesis yields a λ^*_{L} derivation of $M_1 \Downarrow D$ where $[\![D]\!] = B$, which can be used as a premise of ($\Downarrow$ RET-APP-FUNC) to construct the desired derivation of $M_1 \cdot M_2 \Downarrow D$. The other two cases are similar, using the induction hypothesis on the derivations of the premises of the corresponding rules to construct the required λ^*_{L} derivation. Care must be taken for the case of ($\Downarrow$ APP) to assure that the subcases where B is a value configuration and an error configuration are treated separately, although the reasoning for each is identical.

The majority of the cases for the aspectual features of the calculus are straightforward, using a combination of the techniques used to reason about the store (used in the proof of λ^*_{A}), and those to reason about returning configurations (used in the case above). The only interesting case is that of term labelling, which must account for the possibility of the labelled term evaluating to a `return` expression. The case of new label creation, while not very complex, is also presented as it is where the actual "splitting" of the label occurs.

Case ($M = \texttt{newlab}[\tau]$)**.** For the forward direction, we have $(L, A, \texttt{newlab}[\tau]) \Downarrow (L[\alpha \mapsto \tau], A, \alpha)$ for fresh α. We can simply check that $[\![\ (L, A, \texttt{newlab}[\tau])\ ]\!]$ evaluates to

$$(\ [\![L]\!]_X[\xi_\alpha \mapsto \tau]\ ,\ [\![L]\!]_R[\rho_\alpha \mapsto \tau \to \tau]\ ,\ [\![L, A]\!][\rho_\alpha \mapsto \texttt{id}[\tau]]\ ,\ \mathscr{L}_{\mathsf{L}}(\rho_\alpha, \xi_\alpha)\ )$$

This is exactly the translation of $(L[\alpha \mapsto \tau], A, \alpha)$ as required. Indeed, as no inductive hypothesis was required, this also proves the reverse direction.

Case ($M = \alpha\langle\!\langle M \rangle\!\rangle$)**.** In the forward direction, we have $\alpha\langle\!\langle M \rangle\!\rangle \Downarrow B$ for some final configuration B. This must have been derived from one of three inference rules, which form the three subcases

below. In the following, recall that

$$\begin{array}{rcl}[\![\alpha\langle\!\langle M\rangle\!\rangle]\!] & \triangleq & \texttt{let} \\ & & \quad \texttt{Base} = \texttt{handle}\ \xi_\alpha\ [\![M]\!] \\ & & \texttt{in}\ !\rho_\alpha \cdot \texttt{Base}\end{array}$$

Subcase (⇓ INV-RET-NOK) The induction hypothesis yields that $[\![M]\!] \uparrow \texttt{raise}\ \xi_{\alpha'}\ [\![V]\!]$. By the definition of the translation, the semantics of the target language, and the induction hypothesis, we know that $[\![M]\!] \uparrow \texttt{raise}\ \xi_{\alpha'}\ [\![V]\!]$, which is precisely the translation of $\texttt{return}\ \alpha'\ \texttt{to}\ V$ as required.

Subcase (⇓ INV-RET-OK) Apply the induction hypothesis to the first premise to get a derivation of $[\![M]\!] \uparrow [\![L, A, \texttt{return}\ V'\ \texttt{to}\ \alpha]\!]$. This means, by rule (⇓ ML-HANDLE-MATCH), that the variable Base in $[\![\alpha\langle\!\langle M\rangle\!\rangle]\!]$ will be bound to $[\![V']\!]$. Similarly, the inductive hypothesis on the second premise, combined with the substitution property, yields a derivation of

$$[\![M']\!]\ [\ [\![V']\!]\ /\ x\ ]\ (\Downarrow \cup \uparrow)\ [\![B]\!]$$

for some final configuration B. By the translation on environments and the side condition, we have

$$[\![L, A]\!](\xi_\alpha) = \lambda x.[\![M']\!]$$

These facts can be combined to complete the derivation that $[\![\alpha\langle\!\langle M\rangle\!\rangle]\!]$ terminates as $[\![B]\!]$ using the (⇓ APP) or (⇓ RET-APP-ARG) rule in the target language, depending on whether B is a value or returning configuration.

Subcase (⇓ INV-OK) Apply the induction hypothesis to the first premise for a derivation of $[\![M]\!] \Downarrow [\![L, A, V']\!]$, which by rule (⇓ ML-HANDLE-OK), implies that Base is bound to $[\![V']\!]$. This is very similar to the previous case except in the rule used to deduce what is bound to the Base variable. In fact, the rest of the proof proceeds identically.

Now turning our attention to the backward direction of the proof, the antecedent provides a derivation of $[\![\alpha\langle\!\langle M \rangle\!\rangle]\!] \Downarrow D$ for some final configuration D. We must now prove that $\alpha\langle\!\langle M \rangle\!\rangle \Downarrow B$ in λ^*_{L} such that $[\![B]\!] = D$. There are now two subcases, each of which contains two branches:

Subcase: D is a value configuration. There are two ways that this could have occurred. First, the subevaluation of $[\![M]\!]$ terminated normally in some value V which was then bound to `Base` and applied to the contents of ρ_α. Use the induction hypothesis on this evaluation, the definition of the translation on the contents of the cell, and induction hypothesis and the substitution property on the relevant premise of the application to construct the premises and side condition of rule ($\Downarrow$ INV-OK) to construct the desired derivation. In the second case, the subevaluation $[\![M]\!]$ terminated in an error configuration of the from $\texttt{raise}\,\xi_\alpha\,V$, in which case V is again bound to `Base`. In this case the induction hypothesis, substitution property, and translation definition are used to construct the premises of rule ($\Downarrow$ INV-RET-OK) to achieve the result.

Subcase: D is an error configuration. One way for this to occur is for $[\![M]\!]$ to terminate in an error configuration of the same form as the second branch of the previous subcase, but then to have the subsequent application raise an error. This case is actually covered by the aforementioned proof by accounting for the fact that B (the final result of evaluation $\alpha\langle\!\langle M \rangle\!\rangle$ using rule ($\Downarrow$ INV-RET-OK) is a returning configuration. The second case is where $[\![M]\!]$ raises an error which is not caught by it's surrounding handler, in which case we apply the induction hypothesis to this derivation and use it to form the premise of rule ($\Downarrow$ INV-RET-NOK) to yield the result.

B.4 Commutativity for $[\![-]\!]^*_{\mathsf{F}}$ (Proposition 5.3.1)

Proving commutativity for the translation of λ^*_{F} to λ^*_{R} is very similar in flavour to the corresponding proof for λ^*_{A}. Since the target language does not include exceptions, many of the pitfalls encountered when dealing with the translation of λ^*_{L} are avoided. Again, we prove preservation and reflection of the evaluation relation, omitting the cases for the underlying calculus λ, and presenting only those for the aspectual features of λ^*_{F}.

Preservation of Evaluation (Property E $\Rightarrow$)

We show that for any λ^*_{F} configuration M, if $M \Downarrow V$ for some value configuration V, then $[\![M]\!]^*_{\mathsf{F}} \Downarrow [\![V]\!]^*_{\mathsf{F}}$ in λ^*_{R}. As usual, we proceed by induction on the derivation of $M \Downarrow V$.

Case ($\Downarrow$ NEWFUN)**.** The λ^*_{R} configuration $[\![F, A, \texttt{fun}\ (x : \sigma)\{M\}]\!]$ evaluates to

$$([\![F]\!][\rho_\phi \mapsto (\sigma \to \tau)], [\![F, A]\!][\rho_\phi \mapsto \lambda x.[\![M]\!]], \mathscr{F}(\rho_\phi))$$

where M (and by type preservation $[\![M]\!]$) has type τ. On the λ^*_{F} side, we have

$$(F, A, \texttt{fun}\ (x : \sigma)\{M\}) \Downarrow (F[\phi \mapsto \langle \sigma, \tau \rangle, A :: \ \phi(x)\{M\}, \phi) \quad [\phi \notin \texttt{dom}(F)]$$

by rule ($\Downarrow$ NEWFUN). Since ϕ is fresh, it does not appear in F or A, therefore

$$\texttt{lookup}\langle A :: \ \phi(x)\{M\}\rangle(\phi) = \lambda x.M$$

so the λ^*_{R} configuration above is the translate of the result of the λ^*_{F} one as desired.

Case ($\Downarrow$ INSTALL)**.** The translated configuration $[\![F, A, \texttt{around}\ \phi(x)\{M\}]\!]$ is an evaluation of

$$([\![F]\!], [\![F, A]\!], \rho_\phi := (\lambda\texttt{proceed}.\lambda x.[\![M]\!]) \cdot\, !\rho_\phi)$$

By definition, the contents of ρ_ϕ in $[\![F, A]\!]$ is $[\![\texttt{lookup}\langle A\rangle(\phi)]\!]$, therefore the result of the evaluation

above is

$$([\![F]\!], [\![F, A]\!][\rho_\phi \mapsto \lambda x.[\![M]\!][[\![\texttt{lookup}\langle A\rangle(\phi)]\!]/\texttt{proceed}], \texttt{skip})$$

Observe that the contents ρ_ϕ in this configuration, by the definition of the translation and the substitution property, is precisely

$$[\![\texttt{lookup}\langle A :: \texttt{around}\ \phi(x)\{M\}\rangle(\phi)]\!]$$

Therefore the configuration above is the required translation of the result of evaluating the original term in λ^*_{F}:

$$(F, A :: \texttt{around}\ \phi(x)\{M\}, \texttt{skip})$$

Case ($\Downarrow$ FUNAPP)**.** By the translation, we have that $[\![\phi \cdot M]\!] = \texttt{let Arg} = [\![M]\!]\ \texttt{in}\ !\rho_\phi \cdot \texttt{Arg}$, and the induction hypothesis states that $[\![M]\!] \Downarrow ([\![F]\!], [\![F, A]\!], [\![V']\!])$. Further, we know that the contents of ρ_α in $[\![F, A]\!]$ is the translate of a λ term $\lambda x.M' = \texttt{lookup}\langle A\rangle(\phi)$. Therefore the evaluation of $[\![\phi \cdot M]\!]$ has a subderivation of $(\lambda x.[\![M']\!]) \cdot [\![V']\!]$, which in turn has a derivation of $[\![M']\!][[\![V']\!]/x]$ as one of its premises. Use property S and the induction hypothesis on the second premise of ($\Downarrow$ FUNAPP) to deduce that the latter evaluates to $[\![V]\!]$, and use this to construct the desired derivation of $[\![\phi \cdot M]\!] \Downarrow [\![V]\!]$.

Case ($\Downarrow$ FUNPROJ$_i$)**.** Much like the corresponding cases for previous commutativity proofs such as that of λ^*_{A}, preservation of evaluation for projections over advisable function names follows easily from the induction hypothesis on the evaluation of M, and the definition of the λ^*_{R} function emulator $\mathscr{F}$. Simply translate the source term, evaluate it using the λ^*_{R} interpreter, and check that the result is indeed the translate of the evaluation of the original term in λ^*_{A}.

Reflection of Evaluation (Property E $\Leftarrow$)

We now show that for any λ^*_{F} configuration M, if $[\![M]\!]^*_{\mathsf{F}} \Downarrow V$ then $M \Downarrow U$ such that $[\![U]\!]^*_{\mathsf{F}} = V$. As usual, proceed by induction on the derivation of $[\![M]\!]^*_{\mathsf{F}} \Downarrow V$, enumerating the cases by the

structure of M. Note that the cases for function creation and aspect installation in the converse proof also form proofs of this case. The case for projections over function name trivially follow from the induction hypothesis. This leaves the case for advisable function application, which is presented below:

Case ($M \equiv \alpha \cdot N$)**.** Here, the antecedent states that

$$[\![\alpha \cdot N]\!] \quad \triangleq \quad \mathtt{let\ Arg} = [\![N]\!]\ \mathtt{in}\ !\rho_\phi \cdot \mathtt{Arg}$$

evaluates to some value configuration V. Using the induction hypothesis on the subderivation of $[\![N]\!]$ yields that $[\![N]\!] \Downarrow ([\![F]\!], [\![F, A]\!], [\![V']\!])$ in λ^*_{R} and $N \Downarrow (F, A, V')$ in λ^*_{F}. Therefore the contents of ρ_ϕ in $[\![F, A]\!]$ must be the translate of some λ^*_{F} term $\lambda x.M'$ (since $[\![F, A]\!]$ is itself a translate). By definition of the translation this is also $[\![\mathtt{lookup}\langle A\rangle(\phi)]\!]$, and so we must have that $\mathtt{lookup}\langle A\rangle(\phi) = \lambda x.M'$. Now, a subderivation of the application in the above term must therefore be an evaluation of $[\![M']\!][[\![V']\!]/x] \Downarrow V$. Applying the substitution property and the induction hypothesis to this derivation, we obtain a derivation of $M'[V'/x] \Downarrow U$ in λ^*_{F} such that $[\![U]\!] = V$. Therefore use this derivation, together with the evaluations of N and the deduction that $\mathtt{lookup}\langle A, \phi\rangle(=)\lambda x.M'$ to form the premises and side condition of rule ($\Downarrow$ FUNAPP), and construct the required derivation of $\alpha \cdot N \Downarrow U$.

Appendix C

Definability

This appendix presents a calculation which shows that the game semantics denotations of the terms $\texttt{newref}[\tau](M)$ and $[\![\, \mathsf{F}[\![\texttt{newref}[\tau](M)]\!] \,]\!]^*_{\mathsf{F}}$ are equivalent. This completes the proof of the definability result for the $[\![-]\!]^*_{\mathsf{F}}$ translation, and therefore establishes that it is fully abstract. The corresponding proof for $[\![-]\!]^*_{\mathsf{A}}$ is nearly identical. Details of where it differs from the one below will be discussed in the text. Before presenting the strategies, the definition of the game model of λ^*_{R} is briefly reviewed. The reader is asked to consult Abramsky, Honda and McCusker [3] for greater detail about the model as it pertains to general references in particular. Those requiring further background in the game models call-by-value languages, or indeed game semantics in general, should refer to [6] and [8] respectively.

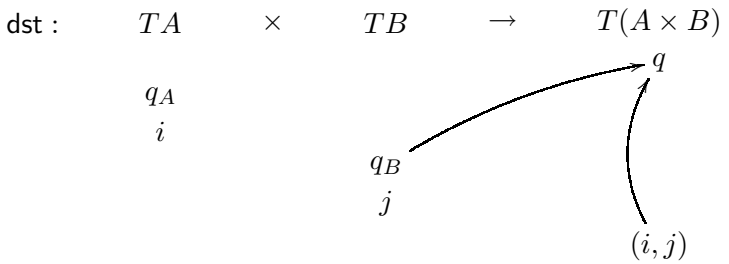
The Game Model of λ^*_{R}

The model of λ^*_{R} is built upon the standard interpretation of a call-by-value language in a Cartesian Closed Category with a strong monad. The objects in the category in this case are families of arenas $\{A_i \mid i \in I\}$ indexed by a set I where an arena is the familiar game semantics triple of a set of moves, a labelling function, and an enabling relation. A morphism between families $\{A_i \mid i \in I\}$ and $\{B_j \mid j \in J\}$ consists of a function $f : I \to J$ and a family $\{\sigma_i \mid i \in I\}$ where

each σ_i is a *thread independent* strategy over the function arena $A_i \Rightarrow B_{f(i)}$. Thread independence is a relaxation of the usual notion of innocence, and states that σ_i can make its moves based on all moves hereditarily justified by the same initial move as it's last one. The product construction $A \times B$ on arenas simply inherits the enabling and labelling components directly from it's components, and the function construction $A \Rightarrow B$ reverses the roles of player and opponent in A, and the initial moves in A are enabled by the initial moves in B. These constructs can be lifted to families of arenas as follows:

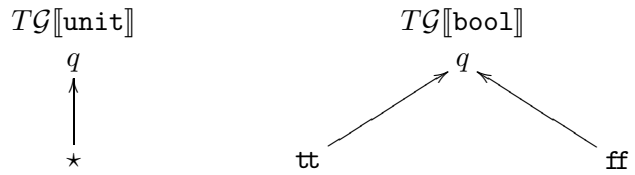
$$\begin{aligned} \{A_i \mid i \in I\} \times \{B_j \mid j \in J\} &= \{A_i \times B_j \mid (i,j) \in I \times J\} \\ \{A_i \mid i \in I\} \Rightarrow \{B_j \mid j \in J\} &= \{\Pi_{i \in I}(A_i \Rightarrow B_{f(j)}) \mid f : I \to J\} \end{aligned}$$

The monadic structure of the category is obtained by using a co-product construction on families of arenas. Specifically, given a family $A = \{A_i \mid i \in I\}$, a single arena $\Sigma_{i \in I} A_i$ can be constructed, whose moves include a unique initial question, which enables $|I|$ moves which will be denoted by the elements of I itself. Each of these moves will in turn enable the initial move in the corresponding arena A_i. A computation TA, constructed from the family A, is then defined as the singleton family $\{\Sigma_{i \in I} A_i\}$. The lifting operation of the monad takes a morphism $f : A \to TB$ to $f^* : TA \to TB$. The latter map simply responds to the initial question in TB with the initial question q in TA. Once a response i is given, f^* just plays as f and ignores the prefix $q \cdot i$ in the TA component. This transformation can be used to define the double-strength morphism $\mathsf{dst} : TA \times TB \to T(A \times B)$ which emulates the left-to-right evaluation discipline of our call-by-value language. A typical play of this strategy is as follows:



Thereafter the play proceeds as the identity over $(A_i \times B_j)$, playing copycat between the input and output components. Similarly, the unit $\eta : A \to TA$ of the monad, taking a value to a computation, is defined by the family $\{\mathsf{in}_i \mid i \in I\}$, where the in_i maps are the obvious strategies over $A_i \Rightarrow \Sigma_i A_i$ which responds to the initial question in $\Sigma_i A_i$ with i and thereafter plays copycat between the two components.

The interpretation $\mathcal{G}[\![-]\!]$ of λ^*_{R} in this category is based upon denoting the `unit` type as the empty arena $\mathcal{G}[\![\mathtt{unit}]\!] = \mathbf{1}$, whose only valid strategy is the empty strategy. The type `bool` is then interpreted as the family $\mathcal{G}[\![\mathtt{bool}]\!] = \mathbf{2} = \{1_{\mathtt{tt}}, 1_{\mathtt{ff}}\}$, which contains a copy of the empty arena for each value of the boolean. Notice that *computations* $T\mathbf{1}$ and $T\mathbf{2}$ built upon these arenas, i.e. prepending an initial question to each family along with a single response for each of its members, take on the familiar dialogue form now standard in game semantics (the arrows here can be read as the "enabled by" relation):



Given these interpretations as base cases, the interpretations of the compound types (i.e. products and functions) of λ^*_{R} are defined inductively:

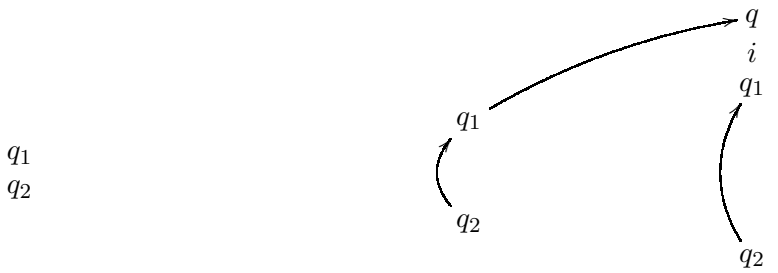
$$\begin{aligned} \mathcal{G}[\![\sigma \times \tau]\!] &= \mathcal{G}[\![\sigma]\!] \times \mathcal{G}[\![\tau]\!] \\ \mathcal{G}[\![\sigma \to \tau]\!] &= \mathcal{G}[\![\sigma]\!] \Rightarrow T\mathcal{G}[\![\tau]\!] \end{aligned}$$

Letting Γ range over typing contexts of the form $x_1 : \tau_1, \ldots, x_n : \tau_n$, a well typed user term $\Gamma \vdash M : \tau$ is interpreted as a map $\mathcal{G}[\![\tau_1]\!] \times \cdots \times \mathcal{G}[\![\tau_n]\!] \to T\mathcal{G}[\![\tau]\!]$ from our category. A free variable is modelled simply by the projection strategy – playing copycat between the appropriate

component and the output – composed with the unit η:

$$\mathcal{G}[\![x_1 : \tau_1, \ldots, x_n : \tau_n \vdash x_i : \tau_i]\!] =$$

$$\mathcal{G}[\![\tau_1]\!] \times \cdots \times \mathcal{G}[\![\tau_i]\!] \times \cdots \times \mathcal{G}[\![\tau_n]\!] \xrightarrow{\pi_i} \mathcal{G}[\![\tau_i]\!] \xrightarrow{\eta} T\mathcal{G}[\![\tau_i]\!]$$

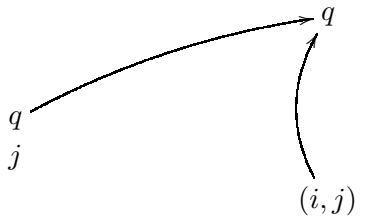


The arrows now signify the justification relation, and are omitted if the move in question is justified by the previous move in the sequence. Pairing is modelled by composing the double strength morphism with the pair comprising the denotations of the two components:

$$\mathcal{G}[\![\Gamma \vdash \langle M, N\rangle : \sigma \times \tau]\!] = \langle \mathcal{G}[\![\Gamma \vdash M : \sigma]\!], \mathcal{G}[\![\Gamma \vdash N : \tau]\!]\rangle ; \mathsf{dst}$$

A typical play of this strategy therefore proceeds as follows:

$$T\mathcal{G}[\![\Gamma]\!] \longrightarrow T\mathcal{G}[\![\tau]\!] \times T\mathcal{G}[\![\sigma]\!] \xrightarrow{\mathsf{dst}} T\mathcal{G}[\![\sigma \times \tau]\!]$$

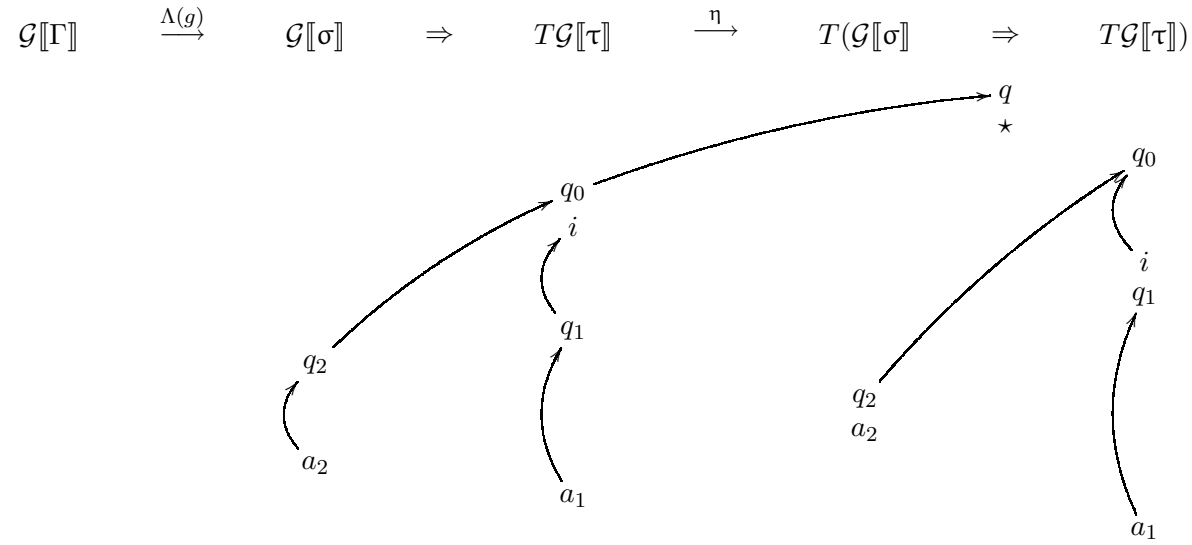


Thereafter, the strategy plays copycat between the right and the left hand side. Keep in mind, however, that each of the two components on the left logically have their own copy of the context $\mathcal{G}[\![\Gamma]\!]$ which are overlaid in the overall strategy. This arises as a result of the definition of pairing of strategies, which in turn is based on the definition of the categorical product, see [3, §3.5] for details. Modelling projections is done by composing the strategy for the pair with the lifted

projection:

$$\mathcal{G}[\![\Gamma \vdash \pi_i(M) : \tau_i]\!] = \mathcal{G}[\![\Gamma \vdash M : \tau_1 \times \tau_2]\!]; \pi_i^*$$

Currying and decurrying is accomplished by simply relabelling the moves in the disjoint union of moves comprising the sub-arenas. Given a map $f : A \times B \to C$, the map $\Lambda(f)$ is the same family of strategies as f (i.e. the same sets of sequences of moves) but interpreted as being played over $A \to (B \Rightarrow C)$. Similarly, the strategy ev is identical to the copycat strategy over $(A \Rightarrow B) \to (A \Rightarrow B)$ played on the arena $(A \Rightarrow B) \times A \to B$. These constructs are used to interpret λ abstraction and application in our language. Let $g = \mathcal{G}[\![\Gamma, x : \sigma \vdash M : \tau]\!]$, a typical play of $\mathcal{G}[\![\Gamma \vdash \lambda x.M : \sigma \to \tau]\!]$ is as follows:



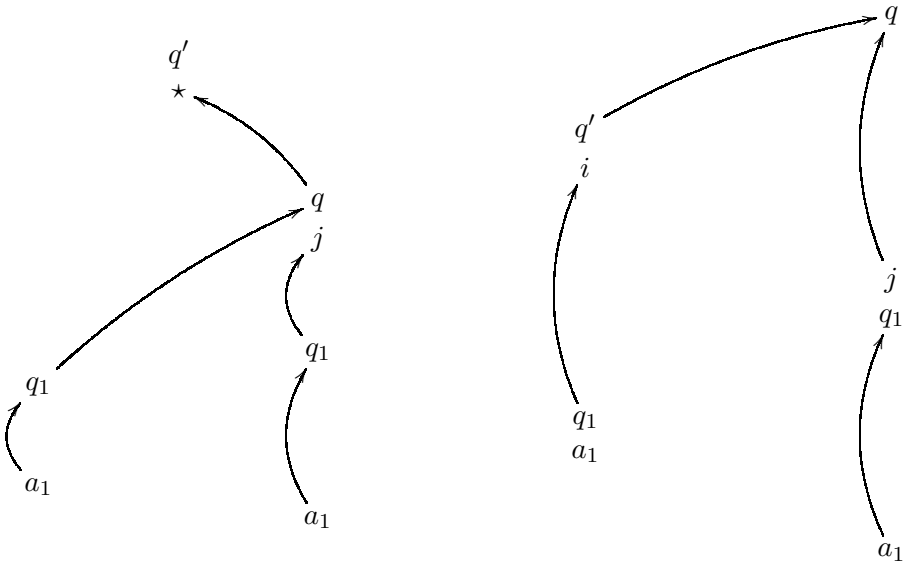
Note that by the definition of T, the family $T\mathcal{G}[\![\tau]\!]$ is a singleton, therefore there is only one possible reindexing function in $\mathcal{G}[\![\sigma]\!] \Rightarrow T\mathcal{G}[\![\tau]\!]$, and so there is only one possible response (written as "$\star$") to the initial question. The play in $\mathcal{G}[\![\sigma]\!]$ corresponds to queries for the value of the free variable in x in g. These are copied to the output component for a response, reflecting the fact that this value comes from the context in which this term lives via an application. The latter is a strategy $[\![G]\!][\![\Gamma \vdash M \cdot N : \tau]\!]$ constructed by composing the paired strategies for M and N with

the double strength morphism and the lifted evaluation strategy ev^*:

$$\mathcal{G}[\![\Gamma \vdash M \cdot N : \tau]\!] = \langle \mathcal{G}[\![\Gamma \vdash M : \sigma \to \tau]\!], \mathcal{G}[\![\Gamma \vdash N : \sigma]\!] \rangle ; \mathsf{dst}; \mathsf{ev}^*$$

A typical play of this strategy is as follows:

$$\mathcal{G}[\![\Gamma]\!] \longrightarrow T(\mathcal{G}[\![\sigma]\!] \Rightarrow T\mathcal{G}[\![\tau]\!]) \times T\mathcal{G}[\![\sigma]\!] \xrightarrow{\mathsf{dst};\mathsf{ev}^*} T\mathcal{G}[\![\tau]\!]$$



In the above, the questions named q begin the preambular dialogue in the strategy ev, the q' questions mark the additional moves prepended to the start of the input sequences because this strategy is lifted, and the q_1 questions are further queries which play simply as the re-labelled identity. Notice how the copycat behaviour of the ev strategy relegates queries to the input variable in the $\mathcal{G}[\![\sigma]\!]$ arena of $\mathcal{G}[\![M]\!]$ to the arena $T\mathcal{G}[\![\sigma]\!]$ in which the play of $\mathcal{G}[\![N]\!]$, i.e. the strategy corresponding to the argument, occurs.

This completes the description of the model of λ^*_{R} with the exception of the strategies for the conditional, and of $\mathtt{newref}[\tau](M)$. The conditional is encoded in an obvious way: querying the appropriate branch depending on the response to the question asked in the boolean, with lifting and η used appropriately in order to match the types. In order to model `newref`, we require a strategy $\mathcal{G}[\![\Gamma \vdash \mathtt{newref}[\tau](M) : \mathtt{ref}[\tau]]\!]$, i.e. a map of type $\mathcal{G}[\![\Gamma]\!] \to T\mathcal{G}[\![\mathtt{ref}[\tau]]\!]$. In [3], where

locations are not initialised, this is constructed using a map $\mathsf{cell} : \mathtt{unit} \to T\mathcal{G}[\![\mathtt{ref}[\tau]]\!]$, i.e. a strategy over the arena

$$T[\ (\mathcal{G}[\![\tau]\!] \Rightarrow T\mathbf{1})\ \times\ (\mathbf{1} \Rightarrow T\mathcal{G}[\![\tau]\!])\ ]$$

Here, we adapt this definition to initialised locations using a map $\mathsf{icell} : T\mathcal{G}[\![\tau]\!] \to T\mathcal{G}[\![\mathtt{ref}[\tau]]\!]$. Denote the initial question in the "dereferencing" component of $\mathcal{G}[\![\mathtt{ref}[\tau]]\!]$ by read, and the initial question in the i^{th} component of the "assignment" sub-arena by $\mathsf{write}(i)$. The strategy icell is then defined to play as follows:

- When icell is asked its initial question, it queries its input. Upon receiving a response i, it responds to the initial question with the unique answer $\star$.
- If read is played before any write move, then icell responds with the move i played in the input component, and plays copycat between its input and output on subsequent non-initial moves justified by this move.
- icell responds to each $\mathsf{write}(i)$ with its unique answer ok
- If read is played and there is a most recent write move $\mathsf{write}(i)$, then icell responds with i.
- If a non-initial move a is played in the read component, this move must be hereditarily justified by an answer i to read. If this is the first such move, then icell copies this move to the i^{th} write component, justified by $\mathsf{write}(i)$, otherwise it plays copycat between the read component and the write component.

The denotation of $\mathtt{newref}[\tau](M)$ is then given by composing the interpretation of M with icell. An example play of the latter is depicted in figure C.1.

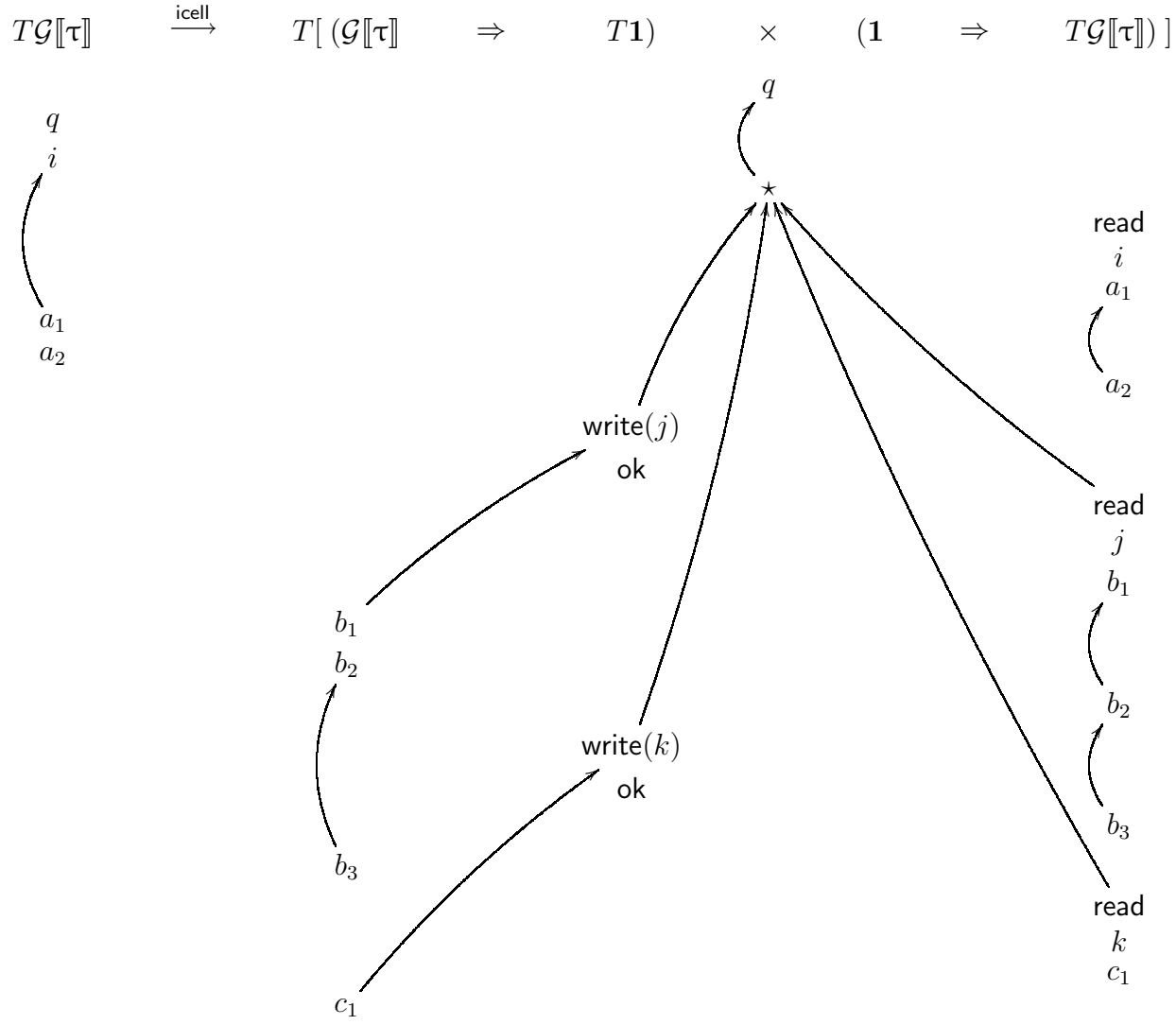


Figure C.1: *A play of the strategy* $\mathcal{G}[\![\Gamma \vdash \texttt{newref}[\tau](M) : \texttt{ref}[\tau]]\!]$. *Note that the play in the* $T\mathcal{G}[\![\texttt{ref}[\tau]]\!]$ *component on the right is identical to a play of the usual strategy for a cell, which plays as* $\mathcal{G}[\![M]\!]$ *if the cell is read before it is written to.*

Definability for $[\![-]\!]^*_{\mathsf{A}}$ and $[\![-]\!]^*_{\mathsf{F}}$

We examine the strategies

$$\begin{aligned}\mathscr{S}_{\mathsf{F}} &\triangleq \mathcal{G}[\![\ \Gamma \vdash [\![\ \mathsf{F}[\![\mathtt{newref}[\tau](M)]\!]\ ]\!]^*_{\mathsf{F}} : \mathtt{ref}[\tau]\ ]\!]\\ \mathscr{S}_{\mathsf{A}} &\triangleq \mathcal{G}[\![\ \Gamma \vdash [\![\ \mathsf{A}[\![\mathtt{newref}[\tau](M)]\!]\ ]\!]^*_{\mathsf{A}} : \mathtt{ref}[\tau]\ ]\!]\end{aligned}$$

By meticulously analysing their plays, we argue that they respond to questions from the environment identically to the strategy for $\mathcal{G}[\![\Gamma \vdash \mathtt{newref}[\tau](M) : \mathtt{ref}[\tau]]\!]$ above. Due to the degree of similarity between the strategies in question, and thus to avoid unnecessary repetition, both of the strategies are analysed simultaneously. The strategy $\mathscr{S}_{\mathsf{F}}$ is described first in each case, and the major differences that may occur between it and $\mathscr{S}_{\mathsf{A}}$ are pointed out in the text. As a result, it is recommended that the reader is fairly comfortable with the model above, as well as both the $[\![-]\!]^*_{\mathsf{F}}$ and $[\![-]\!]^*_{\mathsf{A}}$ before proceeding. Depicting all of the plays diagrammatically proved to be intractable, not to mention unreadable, as they simply comprise instances of the basic building blocks defined in the previous section. Therefore, repeating them each time they are required below was deemed not only impossibly difficult, but in fact detrimental to following the proof. Recall from chapter 5 that the term mapped to $\mathscr{S}_{\mathsf{F}}$ above is observationally equivalent to

$$\begin{aligned}&\mathtt{let\ Init} : \tau = M\\ &\qquad \mathtt{Cell} : \mathtt{unit} \Rightarrow \tau = [\![\mathtt{fun}\ (d : \mathtt{unit})\{\mathtt{Init}\}]\!]^*_{\mathsf{F}}\\ &\mathtt{in}\ \langle \mathtt{assign}_{\mathsf{F}}, \mathtt{deref}_{\mathsf{F}}\rangle\end{aligned}$$

where

$$\begin{aligned}\mathtt{assign}_{\mathsf{F}} &= \lambda \mathtt{Val} : \tau.\pi_1(\mathtt{Cell}) \cdot (\lambda \mathtt{proceed} : \mathtt{unit} \to \tau.\lambda d : \mathtt{unit}.\mathtt{Val})\\ \mathtt{deref}_{\mathsf{F}} &= \lambda d : \mathtt{unit}.\pi_2(\mathtt{Cell}) \cdot \mathtt{skip}\end{aligned}$$

Similarly, the corresponding term for $\mathscr{S}_{\mathsf{A}}$ is

$$\begin{array}{l}\texttt{let Init} : \tau = M \\ \qquad \texttt{Cell} : \texttt{lab}[\tau] = [\![\texttt{newlab}[\tau]]\!]^*_{\mathsf{A}} \\ \texttt{in } \langle \texttt{assign}_{\mathsf{A}}, \texttt{deref}_{\mathsf{A}} \rangle\end{array}$$

where

$$\begin{array}{rcl}\texttt{assign}_{\mathsf{A}} & = & \lambda \texttt{Val} : \tau.\pi_1(\texttt{Cell}) \cdot (\lambda \texttt{d} : \tau.\texttt{Val}) \\ \texttt{deref}_{\mathsf{A}} & = & \lambda d : \texttt{unit}.\pi_2(\texttt{Cell}) \cdot \texttt{Init}\end{array}$$

When expanding the syntactic sugar for `let` what we have are the terms

$$[\, \lambda \texttt{Init} : \tau.(\lambda \texttt{Cell} : \texttt{unit} \Rightarrow \tau.\langle \texttt{assign}_{\mathsf{F}}, \texttt{deref}_{\mathsf{F}} \rangle) \cdot [\![\texttt{fun}\ (d : \texttt{unit})\{\texttt{Init}\}]\!]^*_{\mathsf{F}}\,] \cdot M$$

$$[\, \lambda \texttt{Init} : \tau.(\lambda \texttt{Cell} : \texttt{lab}[\tau].\langle \texttt{assign}_{\mathsf{A}}, \texttt{deref}_{\mathsf{A}} \rangle) \cdot [\![\texttt{newlab}[\tau]]\!]^*_{\mathsf{A}}\,] \cdot M$$

Check that these terms indeed have the type $\texttt{ref}[\tau]$, and therefore their denotations are played over the same arena as $\mathcal{G}[\![\Gamma \vdash \texttt{newref}[\tau](M) : \texttt{ref}[\tau]]\!]$. We now trace these strategies' responses to moves played by the environment for each case enumerated in the definition of icell.

The Initial Question

At the highest level, $\mathscr{S}_{\mathsf{F}}$ is the denotation of an application and has the following form:

$$\langle\, \mathcal{G}[\![\Gamma \vdash \lambda \texttt{Init} : \tau.[\ldots] : \tau \to \texttt{ref}[\tau]]\!]\ ,\ \mathcal{G}[\![\Gamma \vdash M : \tau]\!]\, \rangle; \mathsf{dst}; \mathsf{ev}^*$$

Therefore, in response to the initial question, this strategy asks the initial questions in the first and second components of the pair, receiving responses $\star$ and i respectively. This is immediately followed by asking the initial question in the output component of the λ term denotation, i.e. the curried version of

$$\begin{array}{l}\langle \mathcal{G}[\![\Gamma, \texttt{Init} : \tau \vdash \lambda \texttt{Cell} : \texttt{unit} \Rightarrow \tau.[\ldots] : (\texttt{unit} \Rightarrow \tau) \to \texttt{ref}[\tau]]\!]), \\ \mathcal{G}[\![\Gamma, \texttt{Init} : \tau \vdash [\![\texttt{fun}\ (d : \texttt{unit})\{\texttt{Init}\}]\!]^*_{\mathsf{F}} : \texttt{unit} \Rightarrow \tau]\!] \rangle; \mathsf{dst}; \mathsf{ev}^*\end{array} \tag{C.1}$$

The response made by this strategy to the initial question is the response given by $\mathscr{S}_{\mathsf{F}}$. To calculate it, we again observe that this is an application strategy as above and therefore asks the initial question in the two components of the pair, and after receiving responses (the response in the λ component is the unique one, we shall explore the response from the argument below) plays the initial question in the "body" of the λ term. The latter is the strategy denoting the pair of assignment and dereferencing functions:

$$\langle \mathcal{G}[\![\Gamma, \mathtt{Init} : \tau, \mathtt{Cell} : \mathtt{unit} \Rightarrow \tau \vdash \mathtt{assign}_{\mathsf{F}} : \tau \to \mathtt{unit}]\!],$$
$$\mathcal{G}[\![\Gamma, \mathtt{Init} : \tau, \mathtt{Cell} : \mathtt{unit} \Rightarrow \tau \vdash \mathtt{deref}_{\mathsf{F}} : \mathtt{unit} \to \tau]\!] \rangle ; \mathsf{dst}$$

This strategy's response to its initial question therefore ultimately becomes the response given by $\mathscr{S}_{\mathsf{F}}$. However, since both the components of the pair here are strategies for λ terms, the response to their respective initial questions is again unique, and therefore the response given by dst is their unique pairing. We have therefore shown that when asked the initial question, $\mathscr{S}_{\mathsf{F}}$ responds (correctly) with its unique answer. In fact, to this point, the strategy $\mathscr{S}_{\mathsf{A}}$ has the same structure, and therefore also responds uniquely.

It remains to show that $\mathcal{G}[\![\Gamma, \mathtt{Init} : \tau \vdash [\![\mathtt{fun}\ (d : \mathtt{unit})\{\mathtt{Init}\}]\!]^*_{\mathsf{F}} : \mathtt{unit} \Rightarrow \tau]\!]$ responds to its initial question. Recall that $[\![\mathtt{fun}\ (d : \mathtt{unit})\{\mathtt{Init}\}]\!]^*_{\mathsf{F}}$ was defined to be the term

$$(\lambda \mathtt{Jp} : \mathtt{ref}[\mathtt{unit} \to \tau].\mathscr{F}(\mathtt{Jp})) \cdot \mathtt{newref}[\mathtt{unit} \to \tau](\lambda d : \mathtt{unit}.\mathtt{Init})$$

Where $\mathscr{F}(-)$ is the advisable function emulator which takes a reference cell of function type and produces the corresponding advisable function. The denotation of this term is therefore again an application. After the initial question is played, it asks the initial question in the denotation of the λ term, receiving a unique response. It then asks the initial question in the argument, which by the definition of the icell strategy also responds uniquely after querying the term with which it is initialised, which in this case is a λ term which itself responds uniquely. It then asks

the initial question in the game

$$\mathcal{G}[\![\Gamma, \mathtt{Init} : \tau, \mathtt{Jp} : \mathtt{ref}[\mathtt{unit} \to \tau] \vdash \mathscr{F}(\mathtt{Jp}) : \mathtt{unit} \Rightarrow \tau]\!] \tag{C.2}$$

The answer (if any) supplied is then copied as the answer to the overall strategy. However, this game is simply the denotation of a pair of λ terms, and therefore after querying and receiving unique responses from each component, it responds with their unique pairing as desired. For $\mathscr{S}_{\mathsf{A}}$, the argument is again the same, as the term $[\![\mathtt{newlab}[\tau]]\!]^*_{\mathsf{A}}$ similarly `let` binds a pair into pair of λ-terms emulating a label.

Note that the only part of this analysis which isn't completely determined by the definition of the strategy itself is the response of the strategy $\mathcal{G}[\![\Gamma \vdash M : \tau]\!]$ to its question. Therefore both strategies respond to their initial question if and only if $\mathcal{G}[\![\Gamma \vdash M : \tau]\!]$ does, just as in the $\mathcal{G}[\![\Gamma \vdash M : \tau]\!]$; icell strategy.

Write Moves

Consider an initial question in the i^{th} write component. By definition of the arena $\mathcal{G}[\![\mathtt{ref}[\tau]]\!]$, such a move can only legally be played if the initial question in $\mathscr{S}_{\mathsf{F}}$ has been played an responded to. Therefore, by the reasoning in the previous case, $\mathscr{S}_{\mathsf{F}}$ copies this move down into its subgames until it plays the initial question in the i^{th} component of

$$\mathcal{G}[\![\Gamma, \mathtt{Init} : \tau, \mathtt{Cell} : \mathtt{unit} \Rightarrow \tau \vdash \mathtt{assign}_{\mathsf{F}} : \tau \to \mathtt{unit}]\!]$$

By definition of $\Lambda(-)$, this is the initial move in the following strategy:

$$\mathcal{G}[\![\Gamma, \mathtt{Init} : \tau, \mathtt{Cell} : \mathtt{unit} \Rightarrow \tau, \mathtt{Val} : \tau \vdash \pi_1(\mathtt{Cell}) \cdot (\lambda \mathtt{proceed} : \mathtt{unit} \to \tau.\lambda d : \mathtt{unit}.\mathtt{Val})]\!] \tag{C.3}$$

As the denotation of an application, this strategy proceeds in the now familiar pattern of querying its function and argument components, and receiving unique responses in each case, and

then copying the initial question to the output of the function component $\pi_1(\texttt{Cell})$. By the denotation of projection, this is therefore a query of the first component of `Cell`. Since this is a free variable, this question is therefore copied to the corresponding component in the context. By currying the context we see that this move is now a move in the input component of C.1, and therefore by the definition of ev this is again copied to the "installation" component of C.2. This move is legal because the initial question of the λ term modelling installation was asked and responded to during the interaction in which $\mathscr{S}_{\mathsf{F}}$ responded to its initial question. We must therefore now examine the strategy denoting the installation component of $\mathscr{F}(\texttt{Jp})$. Let

$$\Gamma_0 \triangleq \Gamma, \texttt{Init} : \tau, \texttt{Jp} : \texttt{ref}[\texttt{unit} \to \tau], f : (\texttt{unit} \to \tau) \to (\texttt{unit} \to \tau)$$

The move above is the initial question in

$$\mathcal{G}[\![\Gamma_0 \vdash \pi_1(\texttt{Jp}) \cdot [(\lambda p : \texttt{unit} \to \tau.\lambda x : \texttt{unit}.fpx) \cdot (\pi_2(\texttt{Jp}) \cdot \texttt{skip})] : \texttt{unit}]\!] \tag{C.4}$$

This term is a triply nested application: its argument is itself an application whose argument is in turn a third application. Since the behaviour of the application strategy is to query its function, and then to query its argument after a response has been given, this strategy therefore asks the initial question in the denotations of $\pi_1(\texttt{Jp})$, the $\lambda px.fpx$ subterm, and then $\pi_2(\texttt{Jp})$ in that order. Once each of these strategies respond with their unique answer, the strategy for `skip` is queried and answered, and the nested ev strategies begin to take effect from the inside out:

1. First the initial question is asked in the second component of `Jp` which is copied to the context and results in a read move being played in $\mathcal{G}[\![\texttt{newref}(\lambda d.\texttt{Init})]\!]$. By definition this move is responded to by the appropriate j corresponding to the last write move played in this strategy (or the one from its initial component if no such move exists).

2. This response triggers the next ev strategy to query the output arena of the denotation of $\lambda px.fpx$, which is itself a λ term and therefore responds immediately with its unique answer.

3. Finally, the argument of the outer application has been queried and responded to, and therefore the initial question is copied to the denotation of $\pi_1(\mathtt{Jp})$, i.e. it is copied to the write component of the reference "bound" to Jp. Crucially, observe that the arena $\mathcal{G}[\![\tau]\!]$ is isomorphic to the arena $\mathcal{G}[\![\mathtt{unit} \to \tau]\!]$, and so the write components of the internal reference and the overall strategy $\mathscr{S}_{\mathsf{F}}$ match. This means that the write(i) move played in $\mathscr{S}_{\mathsf{F}}$ is now copied to the i^{th} write component of the internal reference. Put bluntly, a write(i) move in the overall strategy results in a write(i) move in the denotation of Jp. By definition, the latter strategy responds with its unique answer ok. This move is now copied back as the response to the question asked in the denotation of $\mathtt{assign}_{\mathsf{F}}$.

This establishes the desired result that $\mathscr{S}_{\mathsf{F}}$ responds to a write(i) move with its unique answer as desired. Moreover, as in the case of the initial question, the interactions that occurred to arrive at this conclusion crucial in showing the correct read/write behaviour in the remainder of the proof.

The argument for $\mathscr{S}_{\mathsf{A}}$ is very similar, but in its case the subterm $\lambda \mathtt{d} : \tau.\mathtt{Val}$ in the $\mathtt{assign}_{\mathsf{A}}$ component is connected via copycat to the input variable New in the installation component in cell, which corresponds to the strategy

$$\mathcal{G}[\![\lambda \mathtt{New}.[(\lambda \mathtt{Old}.\pi_1(\mathtt{x}) \cdot (\mathtt{New} \circ \mathtt{Old})) \cdot (\pi_2(\mathtt{x}) \cdot \mathtt{skip})]]\!]$$

Here we have omitted the additional let binding of the input variable for clarity, and the x variable corresponds to the internal reference cell generated in the translation of $\mathtt{newlab}[\tau]$. The write move above, once copied to the body component of this strategy, can be traced down to the write component of the cell x. Note that this strategy also results in a read being played in cell via plays in the denotation of the $(\pi_2(\mathtt{x}) \cdot \mathtt{skip})$ subterm. However, as in the case for $\mathscr{S}_{\mathsf{F}}$, it is never queried for further values because the copycat connection between New and $\lambda \mathtt{d} : \tau.\mathtt{Val}$ assures that the variable Old is never queried.

Reads Which Follow Writes

Consider an initial question in the read component of $\mathscr{S}_{\mathsf{F}}$. Since the initial question of $\mathscr{S}_{\mathsf{F}}$ must have been played and responded to to make this move legal, the same reasoning as in the previous case can be used to show that this move is immediately copied to play the initial move in

$$\mathcal{G}[\![\Gamma, \mathtt{Init} : \tau, \mathtt{Cell} : \mathtt{unit} \Rightarrow \tau, d : \mathtt{unit} \vdash \pi_2(\mathtt{Cell}) \cdot \mathtt{skip}]\!]$$

As an application, this strategy therefore plays (and receives unique responses to) the initial questions in its function and argument, then copies the initial read move to the second component of `Cell` (i.e. the `apply` component of $\mathscr{F}(\mathtt{Jp})$. Let

$$\Gamma_0 \triangleq \Gamma, \mathtt{Init} : \tau, \mathtt{Jp} : \mathtt{ref}[\mathtt{unit} \to \tau], x : \mathtt{unit}$$

This move above is therefore the initial question in

$$\mathcal{G}[\![\Gamma_0 \vdash [\lambda \mathtt{Arg} : \mathtt{unit}.((\pi_2(\mathtt{Jp}) \cdot \mathtt{skip}) \cdot \mathtt{Arg})] \cdot x]\!]$$

At this point the usual sequence of initial moves and responses occurs due to the nested applications, in this case they occur in the denotations of $\lambda\mathtt{Arg}.[\cdots]$ and x before proceeding $\pi_2(\mathtt{Jp})$ and `skip`. At this point the initial question is asked in the second component of `Jp` and copied as in the case for writes to the internal reference cell. Since a write(i) move has been played in $\mathscr{S}_{\mathsf{F}}$ by assumption, the reasoning of the previous case tells us that this move is mirrored inside the denotation of `Jp`. Furthermore, the remaining cases of the proof will show that this is the only way a write move can be played, and so the response i to this move (which is made by definition of the strategy icell) corresponds to the i of the last write move played in $\mathscr{S}_{\mathsf{F}}$. This response is then copied back in the usual way as the response of the original read move as required.

For $\mathscr{S}_{\mathsf{A}}$, note that $\mathtt{deref}_{\mathsf{A}}$ is identical to $\mathtt{deref}_{\mathsf{F}}$ and therefore similarly results in a query of

the invocation component in Cell. In this case however, this is simply the strategy

$$\mathcal{G}[\![\lambda \mathtt{d} : \mathtt{unit}.\lambda \mathtt{Base} : \tau.\pi_2(\mathtt{x}) \cdot \mathtt{Base}]\!] \tag{C.5}$$

Which again results in a copying of the read question to x, receiving a response as in the case of $\mathscr{S}_{\mathsf{F}}$ and returning back up the structure of the strategy.

Reading the Initial Value

This case is identical to the previous one until read is played in the encapsulated reference. In this case, the assumption implies that no write moves have been played in the encapsulated reference, therefore this move is copied to the input component of icell, which is composed with the strategy

$$\mathcal{G}[\![\Gamma, \mathtt{Init} : \tau \vdash \lambda d : \mathtt{unit}.\mathtt{Init}]\!]$$

Recall from the interaction that occurred when the initial move was played in $\mathscr{S}_{\mathsf{F}}$, that the initial move in this abstraction has also been played and responded to. Therefore the above read move is the initial question in the body component of this strategy, i.e. the initial question in

$$\mathcal{G}[\![\Gamma, \mathtt{Init} : \tau, d : \mathtt{unit} \vdash \mathtt{Init} : \tau]\!]$$

By definition, any further play in this strategy is copy cat with the initial term $\mathcal{G}[\![\Gamma \vdash M : \tau]\!]$. The two strategies must therefore play in the same fibre, and so the response to the above query must be the same i to which the denotation of M responded. This response is then copied back out as the response to the read question played in $\mathscr{S}_{\mathsf{F}}$, showing that the overall strategy responds as $\mathcal{G}[\![M]\!]$ would if read before it is written to as required.

In the case of $\mathscr{S}_{\mathsf{A}}$, this move triggers the application in the body of the invocation component of Cell (C.5). Resulting in a query of into the internal reference. However, in this case, since no write moves have been played, the reference just plays the identity, and therefore copycats with

Base, which in turn copycats with the variable Init from the $\mathtt{deref}_{\mathsf{A}}$ term, which itself copycats with $\mathcal{G}[\![M]\!]$ as desired.

Non-Initial Moves in the Read Component

This case considers further queries made by the environment in the second component of $\mathscr{S}_{\mathsf{F}}$ (i.e. the a_i, b_i, and c_i in figure C.1). Note that such moves can only occur after at least one read move has been played and responded to, and by definition of the arena in which our strategies are played must be hereditarily justified by some response i to a read. Also, observe that if such a move is played before a write move, the interaction is identical to the one in the previous case: a string of copycats leads to the question being copied to the initial value $\mathcal{G}[\![M]\!]$, and since this strategy can't play in the cell itself its response is just copied back as a response to the query in $\mathscr{S}_{\mathsf{F}}$. We therefore consider the case of such a move a being played after a write has been played.

We want to show that $\mathscr{S}_{\mathsf{F}}$ responds to a by playing a copy of it in the i^{th} write component, then copies any response it receives back to the read component. As usual, the question is copied down the structure of the strategy until it is played in the substrategy

$$\mathcal{G}[\![\ldots \vdash \mathtt{newref}[\tau](\lambda d.\mathtt{Init})]\!]$$

denoting the internal reference encapsulated by the term. By the above cases and the supposition, this move is hereditarily justified by i corresponding to the last write move played in the cell (by the fact that each write move played in $\mathscr{S}_{\mathsf{F}}$ is mirrored in this sub strategy). By its definition, icell, now responds by copying this move to the corresponding write component.

Recall that this substrategy is being played in the context of an application representing the binding of the reference to the variable Jp in the context, therefore this move must be copied to the context and back out to the install component of the advisable function emulator $\mathscr{F}(\mathtt{Jp})$. In other words, this move becomes a query for the input variable in the denotation of $\pi_1(\mathtt{Jp})$ in C.4. By the definition of ev, this question must be copied to the argument, which triggers the evaluation of the body of the $\lambda px.fpx$ term. Being itself an application, the series of usual queries

and responses to each component occur until a query for the output of f is made, which is in turn copied to the context $\mathcal{G}[\![\Gamma_0]\!]$ which by decurrying and projection is copied out the $\pi_1(\texttt{Cell})$ subcomponent in C.3. Again, this must be copied to the argument component, i.e. into the denotation of the $\lambda\texttt{proceed}.\lambda d.\texttt{Val}$ subterm. A series of dummy move now take place between this arena and the fpx sub-arena in C.4 which effectively eliminates the η abstractions present in these moves, and ultimately culminates a being played in the denotation of the free variable `Val`. Crucially, because the variables `proceed` and d do not appear in the argument to $\pi_2(\texttt{Cell})$, the dereferencing subterm – the second projection of the variable `Jp` – in the strategy C.4 is never queried for a value.

The move a played in the denotation of `Val` is now copied into the context of C.3. Observe now that by decurrying and copying this move through the outer applications representing the `let` bindings, this is precisely an instance of a being played in the i^{th} write component of $\mathscr{S}_{\mathsf{F}}$: the first part of the desired result. It is now the onus of the environment to respond to this move with b. Note that before this happens, there may be any number of read and write queries in $\mathscr{S}_{\mathsf{F}}$, but each of these must be played in full and responded to because of the bracketing condition. Once b is played, it is copied back down through the structure of the term as a response to the pending a's until it responds to the a move played in the write component of the internal reference bound to `Jp`. This strategy then copies it back to its read component, triggering the obvious copycats which result in b being played as the response to the original a move played in the read component of $\mathscr{S}_{\mathsf{F}}$, thereby establishing the result.

This series of interactions is essentially emulated in $\mathscr{S}_{\mathsf{A}}$, as this strategy has also played the initial moves in its subcomponents. Therefore the move a is played down into the internal reference, where it is similarly copied to its write component and copied back out to $\texttt{assign}_{\mathsf{A}}$. Now, by the connection between the $\lambda d : \texttt{unit}.\texttt{Val}$ component and the `New` variable in the installation component of `Cell` which we described in the case of a write move, this sequence results in a similar query of `Val` to which the environment is left to respond. Once it does so, the response b is copied back through `New` in to the internal reference, where it is copied to the read component and passed back up the structure.

Bibliography

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer-Verlag, 1996.

[2] Samson Abramsky, Dan R. Ghica, C.-H. Luke Ong, and Andrzej S. Murawski. Applying game semantics to compositional software modelling and verification. In *Proceedings of the 10^{th} International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 421–435. Springer-Verlag, 2004.

[3] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references. In *Proceedings of the 13^{th} Annual IEEE Symposium on Logic in Computer Science (LICS '98)*, pages 334–344. IEEE Computer Society Press, 1998.

[4] Samson Abramsky, Radhakrishnan Jagadeesan, and Pasquale Malacaria. Full abstraction for PCF. *Information and Computation*, 163(2):409–470, December 1997.

[5] Samson Abramsky and Guy McCusker. Linearity, sharing and state: A fully abstract game semantics for Idealised Algol with active expressions. In P.W. O'Hearn and R.D. Tennent, editors, *Algol-like Languages*, volume 2, pages 297–329. Birkhaüser, 1997.

[6] Samson Abramsky and Guy McCusker. Call-by-value games. In M. Nielsen and W. Thomas, editors, *Computer Science Logic: 11th International Workshop Proceedings*, volume 1414 of *Lecture Notes in Computer Science*, pages 1–17. Springer Berlin / Heidelberg, 1998.

[7] Samson Abramsky and Guy McCusker. Full abstraction for Idealised Algol with passive expressions. *Theoretical Computer Science*, 1998.

[8] Samson Abramsky and Guy McCusker. Game semantics. In *Logic and Computation: Proceedings of the 1997 Marktoberdorf Summer School*, 1998.

[9] Jonathan Aldrich. Open modules: Modular reasoning about advice. In *Proceedings of the 19^{th} European Conference on Object-Oriented Programming (ECOOP '05)*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer-Verlag, 2005.

[10] Glenn Bruns, Radhakrishnan Jagadeesan, Alan Jeffrey, and James Riely. μABC: A minimal aspect calculus. In *Proceedings of the 15^{th} International Conference on Concurrency Theory (CONCUR '04)*, volume 3170 of *Lecture Notes in Computer Science*, pages 209–224. Springer-Verlag, September 2004.

[11] Edmund M. Clarke, Ansgar Fehnker, Zhi Han, Bruce H. Krogh, Joël Ouaknine, Olaf Stursberg, and Michael Theobald. Abstraction and counterexample-guided refinement in model checking of hybrid systems. *International Journal of Foundations of Computer Science*, 14(4):583–604, 2003.

[12] Edmund M. Clarke, Orna Grumberg, Sumit Kumar Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[13] Curtis Clifton and Gary T. Leavens. Minimao1: An imperative core language for studying aspect-oriented reasonings. *Science of Computer Programming, special issue on Foundations of Aspect-Oriented Programming*, 63(3):321–374, 2006.

[14] Curtis Clifton, Gary T. Leavens, and Mitchell Wand. Parametrized aspect calculus: A core calculus for the direct study of aspect-oriented languages. Technical Report 03-13, Iowa State University, November 2003.

[15] Daniel S. Dantas and David Walker. Harmless advice. In *Proceedings of the 33rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*, pages 383–396. ACM Press, 2006.

[16] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. Poly AML: A polymorphic aspect-oriented functional programming language. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 2005.

[17] Daniel S. Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. AspectML: A polymorphic aspect-oriented functional programming language. *Transactions on Programming Languages and Systems (TOPLAS)*, To Appear.

[18] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *OOPSLA Workshop on Advanced Separation of Concerns*, Minneapolis, USA, October 2000.

[19] Dan R. Ghica. Regular language semantics for a call-by-value programming language. In *Proceedings of the Conference on Mathematical Foundations of Programming Semantics*, volume 45 of *Electronic Notes in Theoretical Computer Science*, pages 106–118. Elsevier, November 2001.

[20] Dan R. Ghica and Guy McCusker. The regular-language semantics of second-order idealised algol. *Theoretical Computer Science*, 309(1):469–502, 2003.

[21] Jean-Yves Girard. *Proofs and Types*. Cambridge University Press New York, 1989.

[22] Chris Hankin and Pasquale Malacaria. Program analysis games. *ACM Computing Surveys*, 31(3), September 1999.

[23] Robert Harper. A simplified account of polymorphic references. *Information Processing Letters*, 51(4):201–206, 1994.

[24] Dominic Hughes. *Hypergame Semantics: Full Completeness for System F*. PhD thesis, University of Oxford, 1999.

[25] Peter Hui and James Riely. Typing for a minimal aspect language: Preliminary report. In *Proceedings of the Workshop on Foundations of Aspect Oriented Languages (FOAL)*, 2007.

[26] Martin Hyland and C.-H. Luke Ong. On full abstraction for PCF: I, II and III. *Information and Computation*, 163(2):285–408, December 1997.

[27] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, May 2001.

[28] Radhakrishnan Jagadeesan, Alan Jeffrey, and James Riely. A typed calculus for aspect oriented programs. Unpublished manuscript, available at `http//fpl.cs.depaul.edu/ajeffrey/papers/typedABL.pdf`, 2003.

[29] Radhakrishnan Jagadeesan, Alan Jeffrey, and James Riely. An untyped calculus of aspect oriented programs. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *Lecture Notes In Computer Science*, 2003.

[30] Radhakrishnan Jagadeesan, Alan Jeffrey, and James Riely. Typed parametric polymorphism for aspects. *Science of Computer Programming, special issue on Foundations of Aspect-Oriented Programming*, 63(3):267–296, December 2006.

[31] Radhakrishnan Jagadeesan, Corin Pitcher, and James Riely. Open bisimulation for aspects. In *Proceedings of the 6^{th} International Conference on Aspect-Oriented Software Development*, pages 107–120, Vancouver, British Columbia, Canada, 2007. ACM Press New York, NY, USA.

[32] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15^{th} European Conference on Object-Oriented Programming (ECOOP '01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, February 2001.

[33] James Laird. *A Semantic Analysis of Control*. PhD thesis, University of Edinburgh, 1998.

[34] James Laird. A fully abstract games semantics of local exceptions. In *Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science (LICS '01)*, pages 105–114. IEEE Computer Society Press, 2001.

[35] James Laird. A game semantics of linearly used continuations. In *Foundations of Software Science and Computation Structures*, volume 2620 of *Lecture Notes in Computer Science*, pages 313–327. Springer Berlin / Heidelberg, January 2003.

[36] James Laird. A game semantics of local names and good variables. In *Proceedings of the Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*, volume 2987 of *Lecture Notes in Computer Science*, pages 289–303. Springer-Verlag, 2004.

[37] James Laird. Game semantics and linear CPS interpretation. *Theoretical Computer Science*, 333:199–244, 2005.

[38] James Laird. A calculus of coroutines. *Theoretical Computer Science*, 350(2-3):275–291, February 2006.

[39] Xavier Leroy. Polymorphism by name for references and continuations. In *Proceedings of the 20th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '93)*, pages 220–231, Charleston, South Carolina, United States, 1993. ACM Press New York, NY, USA.

[40] Xavier Leroy and Pierre Weis. Polymorphic type inference and assignment. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)*, pages 291–302, Orlando, Florida, 1991. ACM Press New York, NY, USA.

[41] Pasquale Malacaria and Chris Hankin. Generalized flowcharts and games (extended abstract). In *Proceedings of the International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1443 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[42] Pasquale Malacaria and Chris Hankin. A new approach to control flow analysis. In *Proceedings of the Conference on Compiler Construction (CC)*, volume 1383 of *Lecture Notes in Computer Science*, pages 95–108. Springer-Verlag, 1998.

[43] Pasquale Malacaria and Chris Hankin. Non-deterministic games and program analysis: An application to security. In *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*, pages 443–452, 1999.

[44] Ian A. Mason and Carolyn L. Talcott. Equivalence in functional languages with effects. *Journal of Functional Programming*, 1(3):287–327, 1991.

[45] Guy McCusker. Full abstraction by translation. In *Proceedings of the Third Workshop of the Theory and Formal Methods Section*, Department of Computing, Imperial College, London, 1996.

[46] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, 1991.

[47] Andrzej S. Murawski. Functions with local state: Regularity and undecidability. *Theoretical Computer Science*, 338(1-3):315–349, June 2005.

[48] C.-H. Luke Ong. Observational equivalence of 3rd-order idealized algol is decidable. *Proceedings of the IEEE Symposium on Logic in Computer Science (LICS)*, pages 245–256, 2002.

[49] Andrew M. Pitts. Operationally-based theories of program equivalence. *Samantics and Logics of Computation, Publications of the Newton Institute*, pages 241–298, 1997.

[50] Andrew M. Pitts and Ian D. B. Stark. Operational reasoning for functions with local state. In Andrew Gordon and Andrew Pitts, editors, *Higher Order Operational Techniques in Semantics*, pages 227–273. Publications of the Newton Institute, Cambridge University Press, 1998.

[51] John C. Reynolds. The essence of Algol. In J. W. de Bakker and Johannes C. van Vliet, editors, *Algorithmic Languages*, pages 345–372. North-Holland Publishing Company, Amsterdam, 1981.

[52] John G. Riecke. Fully abstract translations between functional languages. In *Proceedings of the 18th Annual ACM Symposium on Principles of Programming Languages (POPL '91)*, pages 245–254, Orlando, Florida, 1991.

[53] Sam B. Sanjabi and C.-H. Luke Ong. Fully abstract semantics of additive aspects by translation. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, pages 135–148, Vancouver, British Columbia, Canada, 2007. ACM Press New York, NY, USA.

[54] Manfred Schmidt-Schauss, Joachim Niehren, Jan Schwinghammer, and David Sabel. Adequacy of compositional translations for observational semantics. In *5th IFIP International Conference on Theoretical Computer Science*, volume 273 of *IFIP International Federation for Information Processing*, pages 521–535. Springer Boston, July 2008.

[55] Jean-Pierre Talpin and Pierre Jouvelot. The type and effect discipline. In *Proceedings of the 7th Annual IEEE Symposium on Logic in Computer Science (LICS '92)*, pages 162–173, Los Alamitos, California, 1992. IEEE Computer Society Press.

[56] Hideaki Tatsuzawa, Hidehiko Masuhara, and Akinori Yonezawa. Aspectual CAML: An aspect-oriented functional language. In *Proceedings of the Workshop on Foundations of Aspect Oriented Languages (FOAL)*, pages 39–50, March 2005.

[57] Mads Tofte. Type inference for polymorphic references. *Information and Computation*, 89(1):1–34, 1990.

[58] Nikos Tzevelekos. Full abstraction for nominal general references. In *Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science (LICS '07)*, pages 339–410, Washington, DC, USA, 2007. IEEE Computer Society Press.

[59] David Walker, Steve Zdancewic, and Jay Ligatti. A theory of aspects. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming (ICFP '03)*, pages 127–139, Uppsala, Sweden, August 2003. ACM Press.

[60] Mitchell Wand, Gregor Kiczales, and Christopher Dutchyn. A semantics for advice and dynamic join points in aspect oriented programming. *Transactions on Programming Languages and Systems (TOPLAS)*, 26(5):890–910, September 2004.

[61] Meng Wang, Kung Chen, and Siau-Cheng Khoo. On the pursuit of static and coherent weaving. *Proceedings of the Workshop on Foundations of Aspect Oriented Languages (FOAL)*, 2006.

[62] Meng Wang, Kung Chen, and Siau-Cheng Khoo. Type-directed weaving of aspects for higher-order functional languages. *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantcs-Based Program Manipulation*, pages 78–87, 2006.