A Study in Higher-Order Programming Languages

MS Thesis

Morten Rhiger

Abstract: This thesis describes some interplays between the specification and the implementation of higher-order programming languages. We first investigate a traditional implementation of Scheme (compiler, run-time machine). We then turn to Action Semantics. And finally we cross-fertilize these two approaches using both syntax-directed and type-directed partial evaluation and introducing an alternative representation of data structures.

December 1997

To my late father

Acknowledgments

First of all I thank my supervisor, Olivier Danvy, from whom I learn every day. I appreciate his technical expertise and his knowledge of how to work scientifically. His insight in computer science and his will to share this insight with his students exceeds what I have otherwise been exposed to.

Peter D. Mosses's course at DAIMI in the spring of 1997 was an inspiring tour of Action Semantics.

René Vestergaard gave useful comments on a draft of this thesis of both syntactic and semantic nature. He pointed out several weaknesses of the written material.

Oscar Waddell provided useful comments on my Isis Scheme system when he visited DAIMI in the fall of 1997. He kindly directed me to an implementation of the macro system of Chez Scheme.

Thanks to Inge and Margrethe for proofreading an early draft. I really appreciated their late-night job although it exposed my poor English.

Two implementations of higher-order languages have always been a source of inspiration, R. Kent Dybvig's Chez Scheme system and Mark. P. Jones's Gofer system. Kristoffer Rose's XY-package for the LaTeX documentation system was used to construct the diagrams in the thesis.

Finally, thanks to my family for being there even when I am not.

Aarhus, December 1997
Morten Rhiger

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Studying programming languages is reminiscent of the chicken-and-egg problem. Should one start by studying practical implementations of programming languages, i.e., *how* programs run, or should one start by studying formal semantics of programming languages, i.e., *what* programs compute? For every new programming language, a computer scientist faces this choice as well. Opinions vary about which way is the best one. Through our education at DAIMI we are exposed to both.

The first step towards this MS thesis was bottom-up. It amounted to implementing a complete compiler and runtime system for Scheme, which is a higher-order programming language (this work is documented in Chapter 2). This first step clarified how to implement fundamental entities in programming languages, such as environments, state, and control.

The second step was top-down. It amounted to studying Action Semantics, a semantic framework. As a first application Chapter 3 includes a semantic description of a subset of Scheme. This second step clarified how to give a semantics to fundamental entities of programming languages.

The third step was synthetic. It amounted to explore a connection between the intensional and the extensional representations of programming languages. The link between the two, in this setting, is partial evaluation (presented in Chapter 4), a program transformation technique that can be used to achieve semantics-directed compilation of Action Semantics into Scheme (Chapter 5).

This tour through programming languages turned out to be somewhat fruitful. Chapter 6 presents an alternative representation of data structures and a technique that, given an application of the data structure, translates the traditional data structure into the alternative higher-order representation. Using this technique on an interpreter and the abstract syntax of a programming language, the inherent overhead of the interpreter is removed. Returning to partial evaluation, this technique made it possible to implement type-directed partial evaluation in a statically typed language. Returning to the Action Semantics framework, this technique made it possible to obtain Action Semantics directed compilation in a simpler, albeit less efficient, way than by partial evaluation (Chapter 7).

1.1 Structure of the thesis

The concept of compilation is pervasive throughout the present work. Compilation, as a translation from source programs to target programs, and static and dynamic features of programming languages are introduced below. Chapter 2 describes the Scheme system that was designed and implemented during the bottom-up exploration. Chapter 3 presents the Action Semantics framework developed by Peter Mosses. Emphasis is put on the properties that makes Action Semantics different from the operational and denotational frameworks. An Action Semantics description of a subset of Scheme is included in this chapter too. Chapter 4 presents partial evaluation, a program transformation technique that removes the static computations from source programs. Two techniques, syntax-directed and type-directed partial evaluation, are discussed. The differences and similarities of the two are given. Chapter 5 links the two previous chapters and addresses compiling actions by partial evaluation. This chapter repeats experiments conducted earlier by Anders Bondorf and Jens Palsberg and provides more experiments in order to compare several independent directions, including how syntax-directed and type-directed specialization compare on large examples. Chapter 6 presents and formalizes the alternative representation of data types. One application of the alternative representation consists of deriving a statically typed implementation of type-directed partial evaluation from the dynamically typed implementation presented in Chapter 4. Another application consists of removing the inherent interpretive overhead from an interpreter. This is explored further in Chapter 7 which presents an implementation of action notation, without the overhead inherent to interpreters.

1.2 Prerequisities: An overview of Compiling

This section provides a view on compilation as a process that translates source programs to target programs. Some of the computations in the source program may not be present in the target program, but we require the two to be semantically equal.

A program is a description of a *computation*. The *semantics* of the program is an entity that represents the program. For example, the semantics of the program

$$P = (\lambda x.\ x + 2 * 3)\ 5$$

which is written in λ-notation, may be the integer 11. An *interpreter* for a programming language performs the actual computations described by a program in a way that conforms to the semantics. Thus, applying an interpreter for the λ-notation to the program P above should give a value that represents the semantic entity 11.

In the example program P there are computations that are not necessary to perform in order to obtain a value that represents the semantics of the program:[1] the multiplication and the application of the lambda-abstraction. An interpreter may choose not to perform these computations as long as the outcome conforms to the semantics.

[1]This is not true. A semantics may kep track of the number of primitive operation performed. We shall not consider such semantics.

$$\begin{tikzcd}
\text{Source language} \arrow[r, "\text{Compile}"] \arrow[dr, "\text{Interpreter for source Language}"'] & \text{Target language} \arrow[d, "\text{Interpreter for target language}"] \\
 & \text{Result}
\end{tikzcd}$$

Figure 1.1: Interpretation and compilation

A *compiler* is a translator for a programming language, the *source* language. Applied to a *source program*, the compiler returns a *target program*, written in the *target* language, which may be different from the source language. There are two requirements on a compiler. First, a compiler must preserve the semantics of programs, which means that the semantics of the source program and the target program must be the same, and second, a compiler may not diverge.

A compiler may choose to remove those computation from the source program that does not affect the semantics. For example, a compiler from the λ-notation to the λ-notation may reduce the multiplication in the program P. The target program is

$$Q = (\lambda x.\ x + 6)\ 5$$

To decide whether this can be done, it may be necessary first to *analyse* the program. The result of the analysis may say that both 2 and 3 are values and that $*$ can be applied to these values right away. Because the compiler must terminate, so must the analysis. Thus, the analysis must be approximate since we cannot solve the halting problem.

In some programming languages there are features that can always be computed by a compiler without the need for an analysis. Such features are called *static*. Features that cannot in general be computed by a compiler are called *dynamic*.

Applying an interpreter for the target language to the program Q above is more efficient than applying an interpreter for the source language to the original program P. Still, the semantics of the two programs are the same and thus we obtain the same result. (The situation is depicted in Figure 1.1).

Thus, it often pays off to compile a program before interpreting it. This strategy is possible because the compiler must terminate and the source and target programs must have the same semantics. Compilers for programming languages with static features are able to perform the static computations before the target program is generated. Thus the target program is more efficient than the source program.

All programming languages of today have static features but the set of static features differs from language to language. Below I list three features that may be static or dynamic.

Control flow. The flow of control between functions in a program. In most impertive language (e.g., Pascal) the control flow is static. At any point in the program where a function is called the compiler is able to determined which function is called. C is also an imperative language but allows functions to be referenced via pointers.

Higher-order languages (e.g., Scheme, Haskell, ML, and Gofer) all have dynamic control flow.

Scope. The place where a variable is declared. Most programming languages of today are statically (or lexically) scoped (e.g., C, Pascal, Scheme, ML, Gofer, and Java). One exception is Lisp. Static scope allows efficient access to variables since the location of a variable can be determined by a compiler.

Types. The type of the expressions in a program. Most programming languages of today have many different types of values that may exist at run-time. In statically typed languages the compiler can determine the type of all expressions in a program. There are two kinds of statically typed languages, namely (1) languages in which declarations of functions and variables also gives the type of the entities (e.g., C, Pascal, and Java), and (2) languages where no such information is present in the program (e.g., ML and Gofer). In the latter case, the compiler should be able to infer the types of expressions. This happens automatically but nevertheless statically. Scheme and Lisp are examples of dynamically typed languages.

It may be that a compiler for a language with dynamic features, like control flow, scope, or type, is applied to a program that uses the features in a static way. Whether the compiler is able to effectively compile source programs into target programs or whether it does generate dynamic computations in the target program depends on the compiler, the source and target language, and the nature of the dynamic features.

Chapter 2

Design and Implementation of a Scheme System

This chapter presents a complete Scheme system which conforms to the R^{4}RS standard [6]. The primary goal of the system — called the Isis Scheme system[1] — is to support an implementation that may be easily extended with new experimental features. A few experiments have been conducted successfully and have been adopted in the implementation. Isis is implemented in C and Scheme. This has made it easy to install the system on different architectures.

In this section a *system* refers to an implementation of Scheme including all its components. Different implementation strategies implement systems with different components (e.g., runtime system, compiler, interpreter).

Section 2.1 contains the background and motivation for the Isis Scheme system. Section 2.2 discusses problems on how to compile function efficiently in different classes of languages. Section 2.3 describes the implementation of Isis, in particular the virtual machine, the runtime system, and the compiler. Section 2.4 gives some example sessions, which show how to use some of the more advanced features of Isis. Section 2.5 compares Isis with other Scheme implementations, and includes a measures of performance. Section 2.6 concludes.

2.1 An Experimental Scheme System

There exists at least a dozen free Scheme systems and a couple of industrial-quality Scheme systems. There are systems designed to handle specific tasks (shells, graphics) and systems that aim at being comparable (e.g., with respect to efficiency) to implementations of other programming languages (e.g., C). Furthermore, many systems are portable and thus run on the most widely spread types of computers and operating systems. In other words, there are Scheme systems for almost any combination of price, task, efficiency, computer, and operating system that one may be interested in.

The Isis system is not meant to be yet another implementation of the Scheme programming

[1]Originally an acronym: Implementation of Scheme In Scheme. This strategy has been relaxed but the name stuck.

language. The primary goal of Isis is to serve as an *experimental* Scheme environment: a system that does not aim at being easy to *use* but easy to *change*. The secondary goal was the knowledge I gained in the *process* of implementing a complete Scheme system. However, this is of purely educational value and is not accounted for in the implementation.

The source programs of many free Scheme systems are available but there are reasons not to adopt the sources of an existing Scheme system for Isis:

- By implementing the system from scratch, I have obtained a system that I am completely fluent with. This makes extensions easier to implement and errors easier to correct (for me, that is. It is my hope that a publically available version will be easy to work with for others.)
- I have obtained a system with those properties that I find important for an experimental system. Other systems might be designed for other reasons than being the subject of experimental extensions (like efficiency, size, portability, ease of use, price, etc.).
- I wanted to get true appreciation by building my own system from scratch rather than by merely studying other's (which were not designed with pedagogy in mind).
- Already in the initial stage of the implementation of the Scheme system, there were techniques that I wanted to try to apply.

Being an experimental system does not mean that Isis will never be useful to the Scheme community. Through the experiments it is my hope that I can acquire a mastery of full Scheme implementations and thereby being able to improve the design or implementation of Scheme. Extensions and changes that increases the efficiency or generality of the system might lead to publically available versions.

What makes the system of interest is that in less than two month I succeeded in obtaining a quite stable core by the techniques described in Section 2.3.

2.2 Compiling Higher-Order Programs

When comparing to block-structured[2] languages like C or Pascal, two of the most difficult parts of compiling Scheme programs are that functions may leave the scope in which they are defined and that *continuations* may be reified into manipulable objects. A continuation is an (internal, implicit) representation of the rest of the computation (at a given program point, in a given state). Conversion back and forth between implicit and explicit representations exert a cost. The invariant that the system is able to maintain in the implicit representation may not be obeyed by the applications use of the explicit representation.

Consider a program written in a statically scoped language with functions. During the execution of the program, the flow of control may *activate* a scope by entering a local block or by calling a function. Inside this scope another scope may be activated. The execution of the outermost scope does not resume until the execution of the innermost scope has completed.

[2]Here, block-structured means a language without higher-order functions.

The information local to the outermost scope is retained during the performance of the innermost scope. This behaviour enables us to depict the flow of control as an *activation tree* [1]: the nodes are scopes that have nested blocks or that call functions, and the leaves are scopes without nested blocks and that do not call functions. There is an edge between two nodes when control flows from one to the other.

The execution of the program is a depth-first traversal of the activation tree. A depth-first traversal is implemented by the use of a *control stack*: a node is pushed onto the stack when control reaches it from its ancestor and it is popped when control returns to its ancestor.

For a given point in the activation tree the term continuation denotes the path from the root to that point. This path contains enough information to resume the performance of the program from that point. The control stack will contain exactly this path when control reaches this point.

Some programming languages allow continuations to be *captured* at one point and reinstated later. This amounts to save the path from the root to the point in the activation tree where the continuation is captured. When continuations are captured there will exist several stacks. They correspond to the paths from the root to the points in the activation tree where they were captured. If the control stack is represented by a contiguous block of cells (a *vectorial* stack) then the whole continuation must be copied when captured. If the control stack is represented by explicitly linked element this is not necessary: during the execution of a program the control stack is an incomplete copy of the activation tree. The whole tree will be constructed gradually. Capturing a continuation amounts to remember the node in the activation tree where it is captured. This is represented by the topmost element on the control stack at the time the continuation is captured.

In the following we consider a sequence of lexically scoped languages of increasing generality (but also increasingly difficult to implement) and how to implement them efficiently using a stack. For the moment variables may not be *assigned* and no *tail-call*s are optimized. These topics shall be explained later.

2.2.1 A Block-Structured language

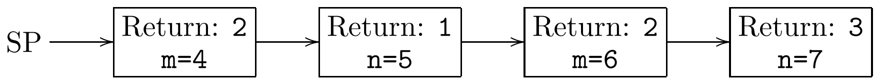
In the first language, call it L_0, functions are not allowed as a result from application, nested function definitions are not allowed, and functions may not be passed as arguments. This is C without pointers to functions [13]. (C may have nested definitions of variables but not functions. Nested definitions of variables are easy to handle since a variable, when referenced, does not need to provide information on the scope in which it is declared. It should be mentioned that the GNU C compiler *does* allow nested function definitions but with undefined results if used improperly.)

The arguments to, and results of functions are never functions, and all variables referenced inside a function are either global variables or formal arguments. Furthermore the compiler is able to identify the function in each function application.

This language can be compiled efficiently by the calling discipline described next: When a function is applied a *frame* is allocated on top of the stack for the actual arguments. These are then evaluated in turn and the results are stored in the frame. The *return address* is saved

```
(define (odd? m)
  (if (zero? m)
      #f
      1:(even? (- m 1))))
(define (even? n)
  (if (zero? n)
      #t
      2:(odd? (- n 1))))
3:(even? 7)
```

Example L_0 program with two functions `odd?` and `even?`. They are mutually recursive. The labels `1:`, `2:`, and `3:` are used to denote the return addresses. They are not considered a part of the program.



Snapshot of execution after four calls. Control is in the function `odd?`. The stack-frames alternates between `odd?`-frames and `even?`-frames. In the snapshot `odd?` is about to read is formal argument `m`, which is found in the topmost stack-frame pointed to by the stack pointer.

Figure 2.1: Snapshot of execution of an L_0 program

in the frame just before control is transfered to the address of the compiled function-body. When control returns the top-most frame is deallocated and execution continues.

A function that references formal arguments reads the value at the appropriate position relative to the stack top. This may be in a different frame than the topmost, if the variable is used as part of an argument to a call. When a function returns it places the result in a dedicated place, either in the frame or in a global register, and jumps to the return address.

Figure 2.1 contains an example L_0 program and a snapshot of the stack while executing the program[3]. Both a vectorial and a linked stack may be used to implement this method. However, since variables may be referenced in frames different from the topmost, the vectorial stack model is more efficient than the linked stack model. The link between two frames (implicitly represented in the former case, explicitly represented by the link in the latter case) is called the *dynamic link* (or *control link* [1]). In the figure this link is represented by an arrow.

2.2.2 Functions as Arguments

Let L_1 be the extension of L_0 that allows an argument to a function to be a function, which may be applied in the body. This corresponds to the C programming language.

[3]To stay in the same notation throughout the discussion of these language I use a Scheme-like syntax. Scheme is the ultimate goal of this sequence of language.

The calling discipline is the same as in L_0 except for the fact that the address of the compiled function-body cannot be known until runtime. Functions are values and are represented by the address of the compiled function-body. This language can be implemented as efficiently as the previous one. (We do not consider optimizations such as inlining of function calls. This can only be done when the function is statically known as in L_0).

2.2.3 Nested Function Definitions

Extending L_0 with nested function definitions gives us language L_2. Unlike L_1, functions may not be passed as arguments. This is Pascal-like languages.

The increased difficulty arises when variables are referenced. The calling discipline is essentially the same as for L_1.

Consider the example L_2 program in Figure 2.2. Function `pow` is defined inside function `g`. A variable referenced inside `pow` may be either a global variable (e.g., `+`), a formal argument to `pow` (e.g., `x`) or a formal argument to `g` (e.g., `v`). Formal arguments to `pow` are found as before: on top of the stack. Variables that are formal arguments to `g` (we refer to them as *free* variables, a notion adopted from the lambda-calculus [2]) are found in the topmost `g`-frame (a frame allocated by a call to the function `g`). But there may be several instances of `pow`-frames on top of the stack, since `pow` is recursive. The topmost `g`-frame does not have a definite offset from the stack-top. It must be possible to access the topmost `g`-frame from the body of `pow`. This can be done in two ways

- All frames contains a link, the so called *static links* (or *access link* [1]), to the topmost frame of the immediately syntactically enclosing function. In this case `pow`-frames would contain a pointer to the topmost `g`-frame. To access one of `g`'s formal arguments from the body of `pow` amounts to referencing the value in the frame pointed to by the static link.

  However, if `pow` was nested deeper than immediately inside `g`, accessing a formal argument of `g` would require a traversal of static links until the correct frame is obtained. The length of this traversal can be computed by the compiler (the difference of the two functions nesting depth) and is bounded by the maximum nesting depth of any function in the program. But traversing static links is inefficient and therefore the following strategy is introduced.

- A global array of pointers to frames is maintained. It has size equal to the maximum nesting depth of any function in a given program. When a function at nesting depth n is called a pointer to the frame is inserted into the array at position n. Thus, the topmost frame of any enclosing function can be accessed in constant time. The array is called a *display* [1]. Frames must still be statically linked in order to keep the display correct on function call and return.

A more complicated calling discipline arises if functions in calls are not statically determinable and if functions may be applied outside their scope. The method described here may be modified to handle these extensions. Functions must be paired with the display when they

```
(define (g xs v)
  (define (map-pow xs)
    (if (null? xs)
        '()
        (cons (pow (car xs))
              (map-pow (cdr xs)))))
  (define (pow x)
    (if (zero? x)
        1
        (* v (pow (- x 1)))))

  (map-pow xs))

(g '(1 2 3) 3)
```

Example L_1 program, which computes 3^1, 3^2, and 3^3. The function `g` has nested definitions of functions `pow` and `map-pow`.

Figure 2.2: Snapshot of execution of an L_2 program

are defined inside other functions. This display must be installed as the current display when the function is called. Next, I will describe a method that implements this. Furthermore, this method allows functions to be returned from function.

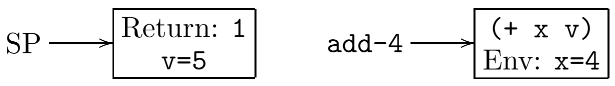
2.2.4 Functions as Results

Finally, in L_3 we allow functions to be passed as arguments, to be applied outside their scope, and to be returned from functions. This is Scheme. When a function is applied it may require the values of free variables that are no longer on the stack, in contrast to the previous languages. A function may outlive its scope: Stack-frames that contain values of free variables of a function may have been removed from the stack at the point where the function is applied. Consider the example L_3 program in Figure 2.3. The function `add` has a local definition of a function `add-x`. The local function is *returned* and is applied after the `add`-frame is removed from the stack. Therefore the free variable `x` cannot be found on the stack.

Different strategies are applicable for the two different kinds of stack representations. Functions are first-class objects and are represented by *closures*. A closure is represented by the compiled code of the function and the *environment* in which the function is defined. How environments are represented differs, but in all cases an environment saves the values of the free variables of a function when the function is defined.

Linked stack. A closure consists of the compiled function and a pointer to the frame that was topmost when the closure was generated. Frames are never overwritten only removed

```
(define (add x)
  (define (add-x v)
    (+ x v))

  add-x)

(define add-4 (add 4))
1:(add-4 5)
```



This is a snapshot of the stack when control is in the body of the function `add-x`. The (global) variable `add-4` is bound to a functional object (closure) created in an environment where `x` was `4`.

Figure 2.3: Snapshot of execution of an L_3 program

from the stack. Therefore a closure remembers the entire stack at the point it was generated. This stack can be reinstated when the closure is applied [10, Chapter 3].

Vectorial stack. With a vectorial stack, frames may be overwritten after they are removed from the stack. It is not possible just to save a pointer to the topmost frame in closures when they are generated. Instead, parts of the stack must be copied into the closure along with the compiled function body. When a function is applied, it needs just the free variables in its body (and of course the arguments). These are stored in a global register; the environment (which corresponds to the display). This kind of closure is referred to as a *display closure* [10, Chapter 4].

In the example program in Figure 2.3 `add-4` will denote a closure created in the environment where `x` is `4`. When the closure is applied this environment and the actual arguments are made accessible to the body of `add-x`.

2.2.5 Mutable Variables

The languages L_0, L_1, and L_2 may be extended with assignments. No modifications are required to the calling discipline. Stack-allocated variables (i.e., variables that are declared as formal arguments of procedures) are stored only once, namely in the appropriate stack-frame. This is not the case in language L_3 when display closures are created since these contains copies of the value of the free variable. An assignment to a variable kept in one display closure does not affect the same variables value in other display closures. To remedy this shortcoming variables can be *boxed*: a variable is represented by the address of a cell containing the real value. This cell is shared among all copies of the variable in display closures. Assignment to a variable takes place in the cell.

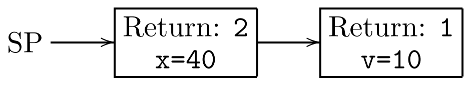
Since, in languages like Scheme, assignments are rare, boxing all variables may not be necessary. The compiler can determine the set of variables that *may* be assigned and only box those.

2.2.6 Tail-call Optimization

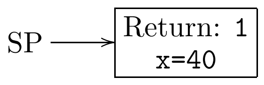
Consider the following program that defines two functions, the second containing a tail-call

```
(define (g x)
  (+ x 2))

(define (f v)
  2:(g (* v 4)))

1:(f 10)
```

Immediately after the function g has been activated the stack looks like



When g returns its result (42) f does nothing but returning this value to *its* caller. The formal argument v to f is not needed after g has been activated. Thus the f-frame may be overwritten by the g-frame so that stack contains only one frame immediately after g is activated:



Note that the return address of the call to f is kept in the frame. When returning from g control is transfered to the caller of f in one step. Thus, not only is one stack frame free to be reused but the return is done in one step.

How this optimization scheme is implemented depends on the kind of stack. Linked stack frames may simply be discarded and assumed recycled by a garbage collector. On a vectorial stack the topmost frame can be *shifted* down the stack [10, Chapter 4.5] immediately before the call.

Care must be taken if nested function definitions are allowed. Consider the program

```
(define (g x)
  (define (f v)
    (* x v))
  (f 21))

(g 2)
```

The call inside `g` is a tail-call. But removing the topmost `g`-frame is not possible since `f` needs the value of `x`. This problem is not present if functions are represented by display closures since the closure representing `f` will contain the value of the free variables, including `x`.

2.2.7 First-Class Continuations

First-class continuations are captured by saving the control stack as an object. Saving a linked stack amounts to saving a pointer to its topmost frame. This may later be reinstated by assigning the saved value to the stack-pointer register.

Saving a vectorial stack in a continuation is done most easily by copying the entire stack into memory. To reinstate a saved continuation yet another copy must be made. Copying stacks back and forth between the stack-area and memory is inefficient and requires more memory than with the explicitly linked stack-model. This may be alleviated by more complex strategies [11].

2.3 Implementation of Isis

The main component of Isis is the *runtime system*. It contains a memory management system and a *virtual machine*, which executes *virtual machine code*. The runtime machine is particularly well suited for running Scheme code; the memory management system contains a garbage collector, there is support for all Scheme data-structures, it is possible to generate *closures*, and there is support for first-class continuations. Furthermore, it is not required that the executable code is compiled Scheme programs. Exactly how the executable code is obtained is not important to the runtime system.

The Isis system contains a compiler from Scheme to virtual machine code. By default it is loaded into the system. This enables Scheme programs to be executed interactively and Scheme source files to be compiled without subsequent execution. The details of the runtime machine and the compiler are explained below.

2.3.1 Source Language

The source language of the Isis compiler is a subset of Scheme (see Figure 2.4). It has quoted literals, local and global variable references, lambda abstractions of fixed and variable arity, applications, local and global assignments, global definitions, conditionals, and sequencing of expressions.

Since the introduction of a macro system (to be presented in Section 2.4.4) the compiler is able to make the following assumptions on the syntax of source programs

- All data, even self-evaluating constants, are quoted.
- The body of a lambda is a single expression. A sequence of expressions must be wrapped into `(begin ...)`.
- Conditional expressions have exactly three sub-expressions, i.e., the alternative branch is always present.

T	::=	`(define` V E`)`	Global Definition
	|	E	
E	::=	V	Variable Reference
	|	`(quote` D`)`	Literal
	|	`(lambda (`V ...`)` E`)`	Lambda Abstraction
	|	`(lambda (`V ... . V`)` E`)`	Variable-arity Lambda Abstraction
	|	`(`E E ...`)`	Application
	|	`(set!` V E`)`	Assignment
	|	`(if` E E E`)`	Conditional
	|	`(begin` E ...`)`	Sequence

The syntactic category T denotes top-level forms. They may be either global definitions or expressions and corresponds to one compilation unit.

Figure 2.4: Source language of the Isis compiler

- A sequence of expressions has at least one sub-expression.
- The macro system rejects erroneous expressions. Thus, no syntax-checks are required by the compiler proper.

These assumptions have greatly simplified the compiler.

2.3.2 Target Language

Three representations of executable code were considered for Isis:

Expressions. The evaluation mechanism is an interpreter for Scheme expressions. Expressions are not compiled into an intermediate representation (or native code) prior to evaluation. However, no advantages are made of the static properties of Scheme programs, for example lexical scope, and there is an overhead of interpreting expression.

Native code. The evaluation mechanism is the CPU of the underlying hardware. Scheme expressions are translated into native code by a compiler. There might be a relatively long compilation phase but there is no interpretive overhead at runtime beyond instruction fetching and decoding in the CPU.

Byte code. The evaluation mechanism is an interpreter for byte-code. Expressions are translated into byte-code by a compiler that does make use of the statical properties of Scheme program. There is an interpretive overhead of running the byte-code programs.

The most efficient representation is by far the native code. However, different computer architectures have different instruction sets so porting a native-code compiler often requires the entire compiler, or at least its code-generation phase, to be rewritten.

Even though an interpreter does not make use of static properties of a program it may be faster than an interpreter for byte-code for the following reasons: An interpreter is directed by Scheme expressions. It fetches *expressions* to be executed, presumably from the memory. An interpreter for byte-code is directed by the byte-code language. It fetches *instructions*, again from memory. However, each Scheme expression is translated into into several byte-code instructions so an interpreter for byte-code must access memory more frequently than an interpreter. On modern hardware this might be costly, especially when running on a processor with cache.

However, an interpreter for byte-code and a byte-code compiler seem to be a more realistic approach to an experimental system. After all, analysis and optimization of Scheme source programs take place most naturally in a compilation or preprocessing phase. Therefore, the Isis system has an interpreter for byte-code, also referred to as the *virtual machine*, and a byte-code compiler.

The design of the target code of the Isis compiler (the instruction set of the virtual machine) has been directed by the source language (a core of Scheme). Each type of expression accounts for a subset of the instructions. There is no overlap between these sets. This has lead to a rather large instruction set (see Figure 2.5, where instructions are grouped together in order to match the corresponding expressions in Figure 2.4). But it has reduced the size of compiled programs. In this way I have attempted to minimize the interpretive overhead of the interpreter for byte-code.

There are instructions to load literal data, to access local and global variables, to construct closures, to apply closures (and primitive functions), to assign local and global variables, to define global variables, and to branch conditionally.

In traditional native machine code, instructions are usually implicitly linked by the order in which they occur in memory. In an interactive system where the amount of compiled code cannot be known a priori, it is important that compiled code can be garbage collected. In the Isis system the virtual machine code instructions are linked explicitly through *continuation-pointers*. Thus, each instruction carries its immediate successor if any. This representation makes it easier to allocate and reclaim instructions stored in the garbage collected heap. Alternatively, only basic blocks of instructions[4] could be linked through continuation pointers. This would reduce the size of compiled code and remove the overhead of reading the address of the next instruction from memory (instead it would be computed by adding the size of the current instruction to its current value).

2.3.3 Runtime System

The runtime system plays a more important role in Scheme implementations than implementations of lower-level languages like C. The runtime system of Scheme must provide a memory management system, a set of predefined procedures and an evaluation mechanism. The runtime system of Isis is implemented in C. C is available on virtually every modern computer system. Implementations of C are also fast enough to implement an efficient backbone of Isis. Furthermore, C has the right division between efficiency and abstraction for implementing a runtime system.

[4]Basic blocks are the largest sequences of instructions such that control flow only *enters* the block at the first instruction and only *leaves* the block at the last expression.

Instruction	Continuation	Arguments	Meaning
`ref_loc_imm`	next	index	access local variable
`ref_loc_mut`	next	index	access and unbox
`ref_fre_imm`	next	index	access free variable
`ref_fre_mut`	next	index	access and unbox
`ref_glb`	next	raw string	access global variable
`lda`	next	object	load object
`env`	next	size	allocate new environment
`sav_loc`	next	index, index	store in new environment
`sav_fre`	next	index, index	
`cls`	next	arity, code	create a closure
`ret`			return from call
`frm`	next	size	allocate stack frame
`arg`	next	index	store in frame
`app`	return		save return and call
`app_tlc`			tail call
`set_fre`	next	index	set free variable
`set_loc`	next	index	set local variable
`set_glb`	next	raw string	set global variable
`box`	next	index	box local variable
`def_glb`	next	raw string	define global variable
`tst`	then, else		branch

Figure 2.5: Target language of the Isis compiler

The memory management system provides an allocation operation and a garbage collector, which is automatically invoked when memory is exhausted. In Isis, the garbage collector is a *stop-and-copy* garbage collector. Stop-and-copy garbage collectors have two advantages over the mark-and-sweep and mark-and-scan garbage collectors: (1) They only traverse the live objects, and (2) they allow blocks of arbitrary size to be allocated. However, it also has a disadvantage: objects are moved back an forth between two *semi-spaces* (or more in the presence of a generational garbage collector). Consequently, an object does not have a definite address. Pointers in the heap are maintained by the memory management system. They are updated whenever the referenced object is moved. But pointers outside the heap (e.g., C variables referring to Scheme objects during execution) are out of the reach of the garbage collector. Therefore, a disciplined style of programming is required in the runtime system in such a way that only C-variables known by the memory management system point into the heap whenever a garbage collection may occur. This has no influence on the use of the Scheme system.

2.3.4 Virtual Machine

The virtual machine has several registers. Most important are the accumulator (`ac`), which holds the most recently computed value; the program counter (`pc`), which points to the instruction to execute next and the stack pointer (`sp`), which points to the topmost stack frame. There are also more ad hoc registers that hold the current environment, an argument vector, and temporary values.

2.3.5 Compiler

The Isis compiler is written in Scheme and it compiles itself. Originally, the demand of a disciplined style of programming prevented a tractable compiler from being implemented as part of the runtime system (as a primitive procedure). The compiler is most easily implemented as a recursive procedure (in C as well as Scheme). The memory management system has no way of accessing local variables in the runtime machine, so trying to implement the compiler in C would be a challenge.

The compiler is abstracted with code-generating functions. Two versions are derived: one that compiles into a file and one that compiles into the heap. The former is used to compile files, in particular library files, whereas the latter is used in interactive sessions.

Isis uses a linked stack of frames. This enables efficient operations on continuations. Functions are represented by display closures that pair the address of the compiled function body with an environment. Environments are vectors of values corresponding to the free values in the functions. This makes variable reference efficient. However, free variables must be saved in each generated closure. Functions like the following, where `x` is free in all the nested functions, impose an overhead from copying free variables into closures.

```
(lambda (x)
  (lambda ()
    (lambda ()
       ...
       (lambda ()
          x))))
```

2.4 Experiments

A few experiments have been successfully conducted. They have all lead to useful extensions and have been integrated into the Isis system.

2.4.1 Error Recovery

Interactive systems demand more from the error handling than so-called batch systems. In batch systems, source programs are written before the system is started. If an error is discovered while the system is running, the systems stops and the source file can be edited to correct the error. In an interactive system, the situation is different: If an error occurs during a session between the system and its user and the system terminates, all previous work is lost. Thus, an interactive system should recover from the error and continue the session. In Isis, recovering from an error amounts to restoring a continuation bound to a top-level variable. The continuation is captured immediately before the initial read-eval-print loop is started. When an error occurs and the continuation is invoked, all pending instructions of the evaluated expression are skipped (and possibly reclaimed by the memory management system later). Errors may be discovered by the runtime system or by the Scheme programs running on top of it. In the following example the errors reported by `car` and `cons` are detected by the runtime system:

```
% iscm
> (car 89)
Error in car: not a pair: 89
> (car (cons 1 2 3))
Error in cons: expected 2 argument(s) but applied to 3
> (+ 37 (error #f "execution stopped") 21)
Error: execution stopped
>
```

2.4.2 Separate Compilation

Isis compiles expressions before they are evaluated. Loading a source file containing a Scheme program may take a while: The contents of the file is compiled before it can be evaluated and the compile times of Isis are relatively long since the compiler itself runs on top of the runtime system. Moreover, the hygienic macro-system is applied prior to compilation (by default). This is a phase consuming even more time than the compilation proper. This inefficiency may be alleviated by compiling the file into an *object file* once and for all. The object file may then be loaded without the overhead of compiling its sources. In the following example the file “`foo.is`” contains a definition `(define x 200)`

```
% iscm
> (load "foo.is")
> x
= 200
> (define x 37)
> (compile-file "foo")
> x
= 37
> (load "foo") ; or (load-object-file "foo.ob")
> x
= 200
>
```

The procedure `load` can load both source files and compiled object files. The presence of the extension "`.is`" in the filename indicates that it is a source file, which should be compiled when it is loaded. Otherwise `load` will attempt to read a file with extension "`.ob`" as if it is an object file. It does not matter whether a newer version of the source file exists, though.

2.4.3 Opaque Objects

In Scheme programs it is often necessary to define new objects that represent structured data. New objects are usually defined in terms of either lists or vectors with a distinguished *tag* (usually a symbol) as the first element. However, with this representation it is impossible to distinguish new objects from lists or vectors that happen to have that particular tag as the first element. This is not a problem if all values that an expression may evaluate to are tagged appropriately. But if the new object may be interchanged with lists or vectors, something else is needed: In Isis new objects can be created dynamically. A constructor and a predicate is associated with each new type of object. Consider

```
% scm
> (call-with-new-object 2
    (lambda (constructor predicate)
      (define-top-level-value 'mypair constructor)
      (define-top-level-value 'mypair? predicate)))
> (define p (mypair 42 'foo))
> (or (vector? p) (pair? p) (number? p))
= #f
> (mypair? p)
= #t
>
```

A new object of size 2 is created. Its constructor and predicate is bound to top-level variables. New objects created by the constructor are neither lists nor vectors so the fields of an object cannot be accessed by list or vector operations. The fields of a structural object must be extracted by the procedure `call-with-contents-of` as in the following example. This procedure is similar to `apply` on lists. We continue the session by adding a procedure that prints objects of the newly constructed type to a list that is accessed by the output procedures (the standard procedures `display` and `write` and the non-standard procedures `printf` and `fprintf`)

```
> (define (print-mypair x port)
    (call-with-contents-of x
      (lambda (fst snd)
        (fprintf port "{~s, ~s}" fst snd))))
> (add-printer mypair? print-mypair)
> p
= {42, foo}
```

```
>
```

Opaque objects are used to represent internal information, for example multiple values:

```
% iscm
> (values 1 #t 'goof)
= 1, #t, goof
>
```

2.4.4 Hygienic Macro System

Writing large applications in the core of Scheme described in section 2.3.1 is a tedious task. A *macro system* facility that allows derived expressions to be described in terms of the core language is required. Isis has adopted the hygienic macro system of the Chez Scheme system.

```
% iscm
> (define-syntax ten (lambda (x) (syntax-case x () [_ #'10])))
> ten
= 10
> (set! ten 20)
Error: invalid syntax (set! ten 20)
> (define-syntax define-constant
    (lambda (x)
      (syntax-case x ()
        [(_ c v) #'(define-syntax c
                     (with-syntax ([value v])
                       (lambda (x)
                         (syntax-case x () [_ #'value]))))])))
> (define f +)
> (define-constant fourtytwo (f 10 32))
> fourtytwo
= 42
> (define f *)
> fourtytwo
= 42
>
```

The goals of the macro-system are (1) to allow essential syntax described by the R^{4}RS standard to be compiled, and (2) to provide a system for adding new syntactic extensions (macros). The former is the primary goal; the latter is secondary though a very useful feature of a Scheme system. The macro-system adopted supports both goals in a general way.

2.5 Other Scheme systems

It is instructive to compare the implementation of Isis with that of other Scheme systems.

2.5.1 Daimi Scheme

Daimi Scheme is a subset of Scheme implemented as part of the compiler course at the University of Aarhus. Daimi Scheme was my first introduction to functional programming and the implementation was my first experience on writing compilers and implementing large systems. The course was aimed at being pedagogical, a necessity when undergraduate students are asked to implement their first compiler. And indeed, I dare to say that more than 95 percent of the three-person groups manage to implement a Scheme system in six weeks.

For me, the experience with Daimi Scheme was an appetiser. Ever since, I have been writing Scheme programs and I have always wanted to start from scratch and implement a complete Scheme system. Now was the time; Isis is the result.

Daimi Scheme is the ancestor of Isis. They have almost the same structure with a runtime system implemented in C and a staged compiler. The Daimi Scheme compiler has more stages than the Isis compiler. Each stage reads a Scheme program from a file an writes a simpler Scheme program to a file. There are (1) a *desugarer*, which corresponds to the macro system of Isis; (2) a *linearizer*, which binds intermediate results to variables, and (3) a code-generation phase. Each stage was one project to be delivered separately.

In Isis, the apparent inefficiency of having several stages that reads from and writes to files has been removed. Isis still has a macro system, but it is itself a Scheme program that is invoked on Scheme data. The linearizer and code-generator are integrated parts of the Isis compiler.

Both Daimi Scheme and Isis have an explicitly linked control stack. Thus, they both support efficient representation of control.

The Daimi Scheme runtime system reads one file of byte-code, executes its contents, and exits after execution. Thus, there are no interactive sessions in Daimi Scheme.

2.5.2 Chez Scheme

Chez Scheme is an industrial-quality Scheme implementation. Chez Scheme has an optimizing compiler that generates native code. The underlying model is a vectorial stack but when continuations are caught the stack is broken into pieces maintained in a linked list of *stack segments*[11].

Several people have been working on the development of Chez Scheme since its first implementation, by its inventor R. Kent Dybvig [10]. This has resulted in a general, efficient implementation of Scheme.

Chez Scheme is the major source of inspiration for Isis. Several features from Chez Scheme have been adopted in Isis: the macro system, which is *the same* macro system as in Chez Scheme; separate compilation; and parameters, which is a variant of dynamic scoped variables.

2.5.3 SCM

The SCM Scheme system of Aubrey Jaffer is a public Scheme implementation. The system basically consists of a runtime machine written in C. It has a mark-and-sweep garbage collector that garbage collects off the C stack(!). This prevents the problems like those of the

Isis runtime machine (where, as mentioned, a disciplined style of programming is required). Reading values directly off the C-stack is hardly standard and may cause problems on some architectures. SCM has a vectorial stack model. In fact, the C-stack is used for recursion. Continuations are caught by copying the entire C-stack to memory. This inefficiency shall be apparent later.

The mark-and-sweep garbage collector prevents anything but cells from being allocated in the heap. Therefore vectors, strings, continuations, and other objects that does not fit into a cell are allocated in the system memory using the operation `malloc`.

Apart from the core of Scheme syntax accepted by the Isis compiler, the interpreter of SCM accepts also a set of derived expressions. This is an advantage over Isis since specialized C-code handles the derived expressions more efficiently than expanded Scheme-code.

2.5.4 Scheme48

The Scheme48 system is an implementation of Scheme that aims at being *tractable* [21]. It is structured into modules each of which is written in a subset of Scheme. It has a virtual machine written in a subset of Scheme. This subset is particularly easy to translate to C. An implemented in C can be is derived automatically from the Scheme sources.

The control stack is a vectorial stack. Environments are represented by a linked list of vectors (the static chain). This is managed in parallel to the call stack. Closures pairs code and environments. To access free variables Scheme48 must traverse the static chain. According to the designers of Scheme48 this is only a small overhead and it frees the compiler from doing a free-variable analysis.

2.5.5 Performance Comparison

The following measurements compares the efficiency of four of the Scheme systems just presented. The programs are written to measure the performance of the central concepts of Scheme discussed in this section: generation of closures, application of procedures, variable references, memory management, etc. It was not the intension to measure how the implementations compares with respect to integer or floating point arithmetic and similar operations.

Some systems might benefit from a larger memory, others may have options controlling the amount of optimization done by the compiler. The tests have been conducted with the default settings for each system.

A short description of each program and why it is of interest follows below.

1. A simple tail-recursive loop that performs 100000 iterations. Applications, and in particular tail-calls, are perhaps the most important operation (with respect to efficiency) in Scheme systems since Scheme programs generally contains many applications.

2. This is also a simple loop, but not a tail-recursive one. After the recursive call 1 is added to the returned value (which is initially 0).

3. This program is essentially the same as the previous one. But in this program a continuation is captured in each iteration. The continuation is never reinstated; in fact, the

System	Pgm. 1	Pgm. 2	Pgm. 3	Pgm. 4	Pgm. 5
Chez	1 420	80	520	10 990	160
Scheme48	24 830	420	1 700	132 600	2 810
SCM	24 520	800	*1 712 080	230 050	2 480
Isis	64 500	1 230	3 680	392 540	6 570
Isis (g.c.)	16 240	510	1 830	104 750	1 660

Numbers in this tables are measured in ms (= 1/1000 sec.). They are the CPU times reported by the timing devices of the different system.

System	Pgm. 1	Pgm. 2	Pgm. 3	Pgm. 4	Pgm. 5
Chez	1.00	1.00	1.00	1.00	1.00
Scheme48	17.49	5.25	3.27	12.07	17.56
SCM	17.26	10.00	*3292.26	20.93	15.50
Isis	45.42	15.38	7.08	35.71	40.44

Numbers in this table are the running times relative to the Chez Scheme system. The numbers for Isis are including the time spend in the garbage collector.

The measures are conducted on a Sun SPARC Classic running SunOS 5.5. The systems are: Chez Scheme (version 4.1u), Scheme48 (version 0.36), SCM (version 4e1), and Isis (not released).

(∗) I interrupted SCM on program 3 after about half an hour.

Table 2.1: Scheme systems benchmarks

continuation may be recycled by the memory management system immediately after it is captured. Systems with a contiguous stack and a naive mechanism for capturing continuations (i.e., copy the entire stack) may have poor performance on this program.

4. This program is another tail-recursive loop. But in each iteration ten closures (with no formal arguments) are allocated and applied.

5. A tail-recursive loop in which a free variable is referenced. The definition and the use of the variables are at different nesting depth corresponding to ten stack frames. This program measures the two methods to fetch the value of a free variable: traversing the static link as in Scheme48, or a constant-time lookup in an environment vector (as in Isis).

The measures are shown in Table 2.1. The last row in the table is the time spend in the garbage-collector on the Isis system. These numbers are included in the total running time of the Isis system.

The important observation is that each system performs well on some programs and not so well on others (or even *remarkably bad*, for example SCM on program 3). Chez Scheme performs best in all tests. In the table all measures are indexed relatively to the Chez Scheme timings. Scheme48 and SCM have almost the same performance, with Scheme48 being slightly better. Isis is three to five times slower than Scheme48 and SCM. The most interesting result is perhaps that SCM has a very poor performance on the continuation program (program 3), even compared to Isis. The reason is that SCM copies the entire (vectorial) execution stack when a continuation as captured.

In program 1, the termination of the iteration is controlled by a check, `(zero? n)`. If this is replaced by the equivalent `(= n 0)` all systems but Isis have the same behaviour. In Isis the procedure `zero?` is a predefined procedure (and is implemented in C) whereas the procedure `=` is written in Scheme and is derived from a binary predefined procedure `integer=?`. As a result, the test `(= n 0)` is much slower than the test `(zero? n)`. Using the former program 1 becomes twenty times slower(!)

It is interesting to compare the first and the final program. They both have a tail-recursive loop (with a different number of recursions, though). But in the final program a free variable is fetched during each iteration. Isis, which saves free variables in display closures, performs slightly better on the final program compared to Scheme48, which traverses the static link to obtain the value.

2.6 Summary and Conclusion

Existing Scheme systems satisfy almost any needs. In particular, much work has been devoted to making Scheme systems more general and more efficient. This required much studying in the area of compilers and runtime systems. State-of-the-art Scheme systems have been under development for decades now, and several people have been involved in their development.

The goals of the Isis Scheme system are to support a small experimental system that is still general and reasonable efficient. In this chapter we have discussed how Scheme programs can be compiled efficiently. In particular, a brief survey of how to represent control was given. With a runtime system, including a virtual machine, implemented in C and a compiler implemented Scheme, Isis is easy to port, and the implementation is small enough than one person could (and has) implemented it in a few months. Yet, Isis is only about 3 to 5 times slower than the other interpreter-based systems, Scheme48 and SCM.

The ancestor of Isis is an implementation of Daimi Scheme. The implementation of Daimi Scheme motivated further studies in the area of compilers. The experience gained by implementing Isis has fueled the rest of this study.

Chapter 3

Action Semantics

This section presents Action Semantics, a general semantic framework for describing the semantics of programming languages.

Section 3.1 gives some uses of semantic descriptions. Two frameworks, denotational and operational semantics, are described briefly. Then Action Semantics and the structure of Action Semantics Descriptions is described. In Section 3.2 actions are introduced. Different aspects of actions are considered and some frequently used actions are explained. In Section 3.3 an example Action Semantics description is given. Section 3.4 concludes.

3.1 Semantic Descriptions

A *semantic description* is a completely formal, unambiguous description of a programming language. The semantic description gives the semantics of programs written in the programming language but this is not its only purpose. The following list contains three uses of semantic description.

Reasoning: It may be possible to prove certain properties of programs written in the programming language or of parts of programs. For example, two programs or two language constructs may be proven equivalent (or not) within the semantic framework.

Behaviour: Programmers may use the semantic description of the programming language as a reference manual. Such a reference may be read either to get an overview of the language or a language construct, or to get a detailed view of a programming language construct.

Implementation: Implementors of the language may use the semantic description to provide a correct implementation. Sometimes, language implementors can even prove that the implementation is correct with respect to the semantic description. Implementations of the language can be written by hand. Somtimes they may be obtained automatically from the semantic description.

"Reasoning" and "Implementation" will be dealt with in examples later in this section. How a semantic description is used as reference manual shall not be discussed in this thesis.

$$\begin{tikzcd}
\text{Source language} \arrow[r, "\text{expander}"] \arrow[dr, dashed] & \text{Action notation} \arrow[d, "\text{Action compiler}"] \\
 & \text{Target language}
\end{tikzcd}$$

Figure 3.1: Semantics directed compiling

We will later consider the automatic translation of programs written in one language, which has a semantic description, into a program written in another language, which has efficient implementations, via a semantics framework. The situation is summarized in Figure 3.1. Translating a source program into an executable target program amounts to the following: From a source program and the semantic description we first obtain the semantics of the program. This is then translated into a target program written in the target language that has an implementation.

Two of the most common semantic frameworks are

Operational Semantics. This framework is widely used to describe the semantics of programming languages. An operational semantics of a program describes *how* the program is executed and thus it closely corresponds to an implementation of the program. Operations semantics are not required to be compositional: The semantics of one programming construct may be described in terms of other programming constructs (or itself. A familiar example is loop-constructs). Operational semantics are seldom complete; there may be constructs that are left out because everybody agrees upon them.

Denotational Semantics. This framework is also widely used. A denotational semantics is not addressing how a program executes but rather what semantic entity a program denotes. These entities live in so-called *domains*. A denotational semantics consists of semantic equations that are compositionally defined.

There is a overlap between denotational semantics and functional programming languages: the λ-calculus. It provides an implementation of a subset of denotational semantics.

Peter Mosses has invented an alternative to these, namely *Action Semantic* [14]. His demands from formal specifications were not satisfied by the operational or denotational frameworks. Action Semantics is a completely formal, general framework for giving semantic descriptions of programming languages. It has successfully been applied to imperative [14, 15], functional

[22] and object-oriented languages [17] not to mention the usual toy languages that computer scientists often provide as testing material [4, 5, 9, 19].

Action Semantics is derived from denotational semantics. It has been developed partly because of the lack of pragmatic features in denotational semantics. In denotational semantics, extending the semantics description of a language with new features that where not originally thought of may require the complete semantics description to be rewritten. In Action Semantics this is seldom necessary: It only requires that new semantic equations, which handles the new features, are added. One reason for this is that, at least to some extent, the semantic equations in Action Semantic implicitly incorporates new features. For example, bindings and storage may be introduced in semantic equations that were not explicitly prepared for this. In contrast, in a denotational semantics, where the semantics of programs are given in terms of mathematical functions, bindings or storage would need to be passed.

As already mentioned, people can read semantic descriptions for two different reasons: to get an overall understanding of a programming language or to get a detailed understanding of a programming language construct. Reading a denotational semantic description for the first purpose is difficult since denotational semantic descriptions use few different symbols. The lambda-operator (λ) is overloaded. It is used to describe both environments, continuations, store, etc. Therefore one must consider the *semantics* of the denotational semantic equations in order to understand the semantics of the programming language. This shortcoming has been alleviated in Action Semantics by replacing combinations of symbols by English words. Thus, one needs to consider the *syntax* of the Action Semantic in order to understand the semantics of the programming language. For example, reading a semantic description that contains terms like bind Y_0 to Y_1 or store Y_0 in Y_1 immediately suggest that bindings are produced or the store is updated.

An Action Semantic description has three parts (or modules). Each describes one aspect of the language under consideration.

Abstract Syntax: This module contains a grammar of the abstract syntax of the language. The abstract syntax may not specify how programs are parsed unambiguously. Instead it is expected that a parser reads source programs and constructs the corresponding piece of abstract syntax. Semantic equations are described in terms of abstract syntax. Requiring an unambiguous grammar would in some cases lead to clumsy semantic equations.

Semantic Functions: This module describes how a piece of abstract syntax is mapped into its semantics. Usually, there is one semantic equation for each non-terminal in the grammar. The semantic equations are given compositionally so mapping a program into its semantics is a matter of expanding the semantic equations with respect to the abstract syntax of the program.

Semantic Entities: This module describes the data processed by actions.

Section 3.3 contains an example Action Semantic description.

3.2 Action Notation

In denotational semantics the semantics of a program is described in terms of *functions*. In contrast, in Action Semantics the semantics of a program is described in terms of *actions*. Actions are described in a style called action notation. In this report we will view action notation as a programming language. This is generally an oversimplification but for our purpose, namely implementing action notation, this is the only point of view.

When action notation is viewed as a programming language actions may be viewed as *statements* of that language. Similar to statements in imperative languages, actions are constructed from primitive actions (concerning primitive operations) and action combinators (concerning control- and data-flow). There is also a notion of *expressions* in action notation called *yielders*. The correspondence between traditional programming languages and action notation exists also here: both actions and statements update and change information, whereas yielders and expressions only access information and give values.

3.2.1 Facets of Actions

Where statements (in most imperative languages) only operates on a *state*, actions operates on several entities. These are addressed by the different *facets* of actions.

Basic facet: Controls *flow*.

Functional facet: Directs *transient information* to and from actions. This information has short lifetime: Actions must explicitly address how this information flows to and from nested actions.

Declarative facet: Produces *bindings of tokens* and addresses how these bindings are combined. Bindings are available in nested actions unless explicitly overwritten.

Imperative facet: Updates and reads a *store*. Changes to the store are available in all actions performed after the time where the update is done. These changes cannot be undone.

Communicative facet: Provides *communication* between agents.

The primitive actions and action combinators are concerned with one at most one facet (except a set of *hybrid* action) but have default behaviours on the information from the other facets.

3.2.2 Performing Actions

Where statements in a programming language are said to be *executed*, actions are said to be *performed*. The performance of an action may terminate or diverge. Terminating actions give information to the next action to be performed.

For a good reason, a diverging action gives no information. It diverges, and any surrounding actions diverge too.

A action may terminate in the following ways:

Complete. This corresponds to normal termination. When completing, an action gives transient information and produces bindings.

Escape. This corresponds to exceptional termination. An escaping action gives transient information but does not produce bindings. One action may *trap* another action. If a trapped action escapes then the closest surrounding action that traps it will be performed. Transient information is transfered between the two.

Fail. This corresponds to abnormal termination. There is a notion of *nondeterminism* is action notation. An action may choose nondeterministically between two branches of performance. If one branch fails then the other branch will be performed (and if the other branch also fails then the entire nondeterministic choice fails). A failing action does not transfer any information: Two branches of a nondeterministic choice are performed with the same information. Consequently, if one branch has made updates to the store then it may not fail since it is impossible to undo these updates.

3.2.3 Frequently used Actions and Yielders

The following explains some examples actions and action combinators. The list describes the facets of each action as well as how it operates on the current information. In this description, we will assume that there is no *interleaving* between actions. That is, the performance of an action is a sequential operation.

complete. A basic, primitive action that terminates normally. No transients are given and no bindings are produced.

escape. A basic, primitive action that escapes with the current transients.

fail. A basic, primitive action that fails.

unfolding A. A basic action combinator that performs A except that inside A, all occurrences of the action unfold are replaced by unfolding A. The action unfold has no meaning outside of unfolding.

A_1 or A_2. Basic action combinator corresponding to a nondeterministic choice. One of the two sub-actions are performed with the current information. If it completes or escapes then the combined action completes or escapes. If it fails then the other sub-action is performed with the current information. If this action fails then the combined action fails. Otherwise, the combined action has the same performance as this action.

A_1 and A_2. Basic action combinator corresponding to implementation-dependent order of evaluation. The two sub-actions are performed in some unspecified order. Each receives the current transients and the current bindings. If a sub-action escapes or fails then the combined action escapes or fails. Otherwise the combined action completes with a tuple of the transients from the two sub-actions and all bindings from the sub-actions provided they are disjoint (if not then the action fails). Handling transients this way is called *functionally conducting* and handling bindings this way is called *declaratively conducting*.

A_1 and then A_2. Basic action combinator corresponding to left-to-right performance. First A_1 is performed, then A_2. The action combinator is functionally and declaratively conducting

A_1 trap A_2. Basic action combinator that acts as A_1 unless A_1 escaped. In this case A_2 is performed. A_2 receives the transients given by A_1 when it escapes but receives the same bindings as A_1.

give Y Functional, primitive action that gives the data yielded by Y. It produces no bindings. If Y yields to nothing then the action fails.

check Y Functional, primitive action that completes with no transients and no bindings if Y yields to true. It fails if Y does not yield true.

A_1 then A_2 Declaratively conducting, functional action combinator. Performs A_1 first and then A_2. Any transients given by A_1 are given to A_2. The transients given by A_2 are given by the combined action. This behaviour is called *functional composing*.

bind Y_1 to Y_2 Declarative, primitive action that produces a binding of a token to a value. None of the received bindings are reproduced.

A_1 moreover A_2 Functionally conducting, declarative action combinator. The bindings received by the combined action are received by both sub-actions. The bindings produced by the combined action are those produced by A_2 and those produce by A_1 that are not also produces by A_2. (The bindings of A_1 are *overlaid* with those produced by A_2).

A_1 hence A_2 Functionally conducting, declarative action combinator. Performs first A_1 then A_2. Any bindings produced by A_1 are received by A_2. The bindings produced by A_2 are produced by the combined action (*declaratively composing*).

A_1 before A_2 Functionally conducting, declarative action combinator. A_1 receives the bindings received by the combined action. A_2 also receives these bindings but overlaid by those produced by A_1. The combined action produces the bindings produced by A_1 overlaid with those produced by A_2.

store Y_1 in Y_2 Imperative, primitive action. Stores the value yielded by Y_1 in the cell yielded by Y_2.

There are also some frequently used yielders. If the conditions in the descriptions are not met, then the yielder yields nothing, which is a special value.

given S. Gives the transients received by the yielder, provided they are of sort S.

given $S\#I$. Gives the Ith component of the tuple of transients received by the yielder, provided it is of sort S.

the D yielded by Y. Gives the data yielded by Y, provided it is of sort D.

the D bound to Y. Gives the value bound, in the received bindings, to the token yielded by Y, provided the value is of sort D.

the D stored in Y. Gives the value contained in the cell yielded by Y, provided the value is of sort D.

Furthermore, a large set of data-operations, which operate on semantic entities, exists: sum, product, difference, not, both, either, is, etc.. Data operations may be applied to yielders to produce new yielders.

3.3 Scheme in Action Notation

To exemplify the material presented in this section, an example of an Action Semantic description of a subset of Scheme is given here. To illustrate the fact that Action Semantic descriptions scales up smoothly, we first consider a description of a pure, higher-order order subset. Afterwards, assignments are added.

The semantic entities defines a sort that represents the possible kinds of values that expressions may evaluate to. These are integer, booleans and functional abstractions.

Semantic Entities .

- entity = integer | truth-value | abstraction .

The syntax of programs are given by the following grammar. Scheme is an expression-oriented language so there is only one syntactic category, Expression. The symbol □ is intended as a placeholder that will allow extension to be added to the grammar later.

Abstract Syntax .

grammar:

- Expression = □ | Identifier | Numeral | ⟦ "#t" ⟧ | ⟦ "#f" ⟧ |
  ⟦ "(" Expression Expression ")" ⟧ |
  ⟦ "(" "lambda" "(" Identifier ")" Expression ")" ⟧ |
  ⟦ "(" "if" Expression Expression Expression ")" |
  ⟦ "(" "let" "(" "(" Identifier Expression ")" ")" Expression ")" ⟧ .

The semantics of programs (Expression) are given by the semantic function evaluate. It maps an Expression into an action. Constant expressions are mapped into actions that gives the corresponding value. The semantics of an identifier is an action that gives the value that the identifier is bound to. In a conditional first the test is evaluated. If it is false, the else-branch is evaluated, otherwise the then-branch is evaluated. Note that the two checks in each of the two sub-action of or are disjoint. Thus exactly one check is true and it does not matter in which order the sub-actions to the basic action combinator or are performed. Lambda abstractions are conveniently represented by abstractions. Scheme is lexically scoped so when an abstraction is made the current bindings are saved.

Semantic Functions .

introduces: evaluate _ .

- evaluate _ :: Expression → Action .

(1) evaluate N:numeral = give decimal of N .

(2) evaluate “#t” = give true .

(3) evaluate “#f” = give false .

(4) evaluate I:Identifier = give the entity bound to I .

(5) evaluate ⟦ “(” “if” E_1:Expression E_2:Expression E_3:Expression “)” ⟧ =
evaluate E_1 then
| | check it is true and then evaluate E_2
| or check not it is true and then evaluate E_3 .

(6) evaluate ⟦ “(” “lambda” “(” I:Identifier “)” E:Expression “)” ⟧ =
give the closure of abstraction of
| | furthermore bind I to the given entity
| hence evaluate E .

(7) evaluate ⟦ “(” E_1:Expression E_2:Expression “)” ⟧ =
| evaluate E_1 and evaluate E_2
then
| enact the application of the given abstraction#1 to the given entity#2 .

(8) evaluate ⟦ “(let ((I:Identifier E:Expression)) E':Expression)” ⟧ =
evaluate E then
| | furthermore bind I to the given entity
| hence evaluate E' .

Using the semantic description we show that for any identifier I and expressions E, E' the following two expressions have the same performance

- (let ((I E)) E')
- ((lambda (I) E') E)

The semantics of the let-construct is taken directly from the semantic description

evaluate ⟦ “(let ((I E)) E')” ⟧ =
evaluate E then
| | furthermore bind I to the given entity
| hence evaluate E' .

The semantics of the application is given by expansion

```
evaluate ⟦ "((lambda (I) E') E)" ⟧ =
| | give the closure of abstraction of
| | | | furthermore bind I to the given entity
| | | hence evaluate E'
| and evaluate E
then
| enact the application of the given abstraction#1 to the given entity#2 .
```

The semantics of the application construct equals

```
evaluate ⟦ "((lambda (I) E') E)" ⟧ =
evaluate E then
| enact the application of the closure of abstraction of A to the given entity
```

where A is the action

$A =$ (furthermore bind I to the given entity) hence evaluate E'

According to the action enact the performance of the application construct equals

```
evaluate ⟦ "((lambda (I) E') E)" ⟧ =
| evaluate E
then
| | furthermore bind I to the given entity
| hence evaluate E' .
```

as required.

To extend the language with assignments requires no changes to the semantic entities. New syntax is added: assignment expressions and sequencing of expressions.

Abstract Syntax . (***continued***)

grammar:

- Expression = ⟦ "(" "set!" I E ")" ⟧ |
  ⟦ "(" "begin" Expression$^+$ ")" ⟧ .

Only the equations that deal directly with variables need to be changed. A variable will be bound to a cell that contains the value denoted by the variable. The cell is allocated when the variable is bound, in lambda abstractions.

Semantic Functions . (***continued***)

introduces: evaluate-sequence _ .

(1) evaluate I:Identifier = give the entity stored in the cell bound to I .

(2) evaluate ⟦ "(" "set!" I E ")" ⟧ =
evaluate E then store it in the cell bound to I .

```
(3) evaluate ⟦ "(" "begin" $E$:Expression$^+$ ")" ⟧ =
    evaluate-sequence $E$ .

(4) evaluate ⟦ "(" "lambda" "(" $I$:Identifier ")" $E$:Expression ")" ⟧ =
    give the closure of abstraction of
    | | allocate a cell and regive
    | then
    | | | store the given entity#2 in the given cell#1
    | | and then
    | | | | furthermore bind $I$ to the given cell#1
    | | | hence evaluate $E$ .

• evaluate-sequence _ :: Expression$^*$ → Action .

(5) evaluate-sequence ⟨ ⟩ = complete .

(6) evaluate-sequence ⟨ $E_1$:Expression $E_2$:Expression$^*$ ⟩ =
    evaluate $E_1$ and then evaluate-sequence $E_2$ .
```

3.4 Summary and Conclusion

Semantic frameworks may be used in reasoning about programs, to give the behaviour of programs, or to provide an implementation of a programming language.

The Action Semantics framework, a form of denotational semantics, has been introduced. Semantics of programs are given in action notation, which may be viewed as a programming language. A complete Action Semantics description of a subset of Scheme has been given. It was shown how to use the semantic description for reasoning. Chapter 7 presents an implementation of Action Semantics. This shall be used to run Scheme programs via the Action Semantic description developed here. The implementation of Action Semantics in Chapter 7 is written in Scheme.

Chapter 4

Partial Evaluation

Partial evaluation is a program transformation technique that specializes programs with respect to part of their input [12]. It is the hope that through this transformation programs become smaller or more efficient or both. This is indeed often the case. This section explains two different approaches to partial evaluation, *syntax directed* and *type directed* partial evaluation.

Section 4.1 introduces some notation and terminology. Section 4.2 introduced the background and motivation for partial evaluation. Section 4.3 discusses the different binding times of expressions. In Section 4.4 partial evaluators are classified according to how they obtain the binding time of expressions and in Section 4.5 partial evaluators are classified according to how flexible they are when a function is used twice with different binding-time patterns.

In Section 4.6 our main use of partial evaluation is explained. Section 4.7 and 4.8 introduces two different classes of partial evaluation. Both Sections also present a tiny partial evaluator. The two classes are compared in Section 4.9. Conclusions are given in Section 4.10.

4.1 Programs and Functions

In order not to confuse programs and the functions they compute we shall introduce the following notation. When a program is executed it takes *input* and produces *output*. Let L be a programming language. By Pgm_L, Inp_L, and Out_L we denote the sets of program texts, input texts, and output texts, respectively, from the language L. The function

$$\mathrm{Run}_L \;::\; \mathrm{Pgm}_L \times \mathrm{Inp}_L^* \rightarrow \mathrm{Out}_L$$

takes the text of a program and a sequence of the texts of input and returns the text of the output obtained by executing the program on the input. The function is partial since a program may diverge when executed.

Different programs takes a different number of input. The operation $\langle\cdot\rangle$ forms a sequence of (input) values. For example

$$\mathrm{Run}_L(P, \langle i_1, \ldots, i_m \rangle) = r$$

indicates that we obtain output r when running program P on input $i_1, \ldots, i_m$.

We may represent the functionality of a program P (written in L) by a function ϕ_P that takes the text of the input to P and returns the text of the output. Define

$$\begin{aligned} \phi_P &:: \mathrm{Inp}_L^* \to \mathrm{Out}_L \\ \phi_P \langle i_1, \ldots, i_m \rangle &= \mathrm{Run}_L(P, \langle i_1, \ldots, i_m \rangle) \end{aligned}$$

4.2 Specializing Programs

From now we consider one programming language L. (The subscripts L from Pgm_L, Inp_L, Out_L and Run_L may be omitted).

Let $P, Q \in \mathrm{Pgm}_L$ be programs written in a language L. Assume that P takes $m + n$ values as input and that Q takes n values as input. If for any input $\langle i_1, \ldots, i_{m+n} \rangle \in \mathrm{Inp}_L^*$ and $r \in \mathrm{Out}_L$ we have

$$\mathrm{Run}_L(P, \langle i_1, \ldots, i_{m+n} \rangle) = r \Leftrightarrow \mathrm{Run}_L(Q, \langle i_{m+1}, \ldots, i_{m+n} \rangle) = r \qquad (4.1)$$

then we say that the program Q is a *specialized* version of the program P with respect to the input $\langle i_1, \ldots, i_m \rangle$. In P we call the formal arguments corresponding to the input $\langle i_1, \ldots, i_m \rangle$ the *static* arguments and the formal arguments corresponding to the input $\langle i_{m+1}, \ldots, i_{m+n} \rangle$ the *dynamic* arguments.

Kleene's S_n^m theorem states that the text of the specialized program Q can be obtained automatically, i.e., there is a computable function that maps the text of the program P and the text of the input $\langle i_1, \ldots, i_m \rangle$ into Q such that the relation in equation (4.1) holds.

The following shows how such a function can be obtained. Let L be a subset of the Scheme programming language. A program that accepts n input is on form `(lambda (i`$_1$ … `i`$_n$`)` E`)` where E may be any Scheme expression. We require that the input to this Scheme-function is the text denoting base values (i.e., Run does not process the input text). The program PE_n^m is implemented in Scheme

```
(lambda (p i_1 ... i_m)
  (let ([i_{m+1} (gensym)] ... [i_{m+n} (gensym)])
     `(lambda (,i_{m+1} ... ,i_{m+n})
        (,p ,i_1 ... ,i_m ,i_{m+1} ... ,i_{m+n}))))
```

Note the use of backquote (`) and comma (,) in this program. Fresh identifiers are generated by `gensym`. We define

$$S_n^m \langle P, i_1, \ldots, i_m \rangle = \mathrm{Run}(\mathrm{PE}_n^m, \langle P, i_1, \ldots, i_m \rangle)$$

$$\begin{tikzcd}
P \arrow[r, "{S_n^m\ \langle P,i_1,...,i_m\rangle}"] \arrow[dr, "{\mathrm{Run}(P,\langle i_1,...,i_{m+n}\rangle)}"'] & Q \arrow[d, "{\mathrm{Run}(Q,\langle i_{m+1},...,i_{m+n}\rangle}"] \\
& r
\end{tikzcd}$$

Figure 4.1: Specialization

Then S_n^m maps a program P and m arguments $\langle i_1, \ldots, i_m \rangle$ into the text of a program Q that takes n input (*the rest of the input*) and executes P on the total sequence of input. The program PE_n^m specializes Scheme programs. It is called a *program specializer* or *partial evaluator*. A similar program must be written to specialize programs written in other languages (but the specializer may still be written in Scheme). Applying Q to the rest of the input gives exactly the same result as applying P to the total sequence of input. This situation is depicted in Figure 4.1.

4.3 Static and Dynamic Expressions

The classification of static and dynamic input extends to expressions and other syntactic categories: In a program P, expressions that depend on dynamic input are called dynamic and expressions that does not, are called dynamic. The execution of a specialized program proceeds in two steps (see Figure 4.1). *Specialize time* where the text of the program P is transformed into the text of the specialized program Q and *run time* where the *residual* program Q is executed.

The values of static expressions can be computed by the specializer whereas the values of dynamic expressions cannot (they depend on input not available at specialize time). It is the hope that a specializer exploits this distinction and reduces static computations at specialize-time so that the residual program Q is smaller or more efficient (or both) than the original program P. The extra overhead from running a program in two steps may be eliminated if the program is often applied to the same set of static input. These are the main reasons for using specializers as program transformators.

The specializer PE_n^m given above does not reduce static computations at specialization time. Section 4.7 and Section 4.8 presents two specializers that (in many cases) gives smaller and faster specialized programs.

4.4 Online and Offline Partial Evaluation

The *binding time* of an expression is the phase in which the value of the expression is determined. This happens at either specialize-time or run-time. One class of program specializers, *online* specializers, determines the binding time of expressions on the fly. They evaluate as much of a program as possible leaving the dynamic expressions in the residual program.

A different class, *offline* partial evaluators, first do a *binding-time analysis* of the program and annotate each expression in the program as either static or dynamic. Then in a second phase they specialize the program. Since the binding time analysis is required to terminate it must necessarily be approximate. Thus, online partial evaluators are often more precise than offline ones. However, offline specializers are often more efficient than online ones.

4.5 Monovariance and Polyvariance

A program may contain several applications of the same function. At each application there may be a different *binding-time pattern*, that is, a different assignment of binding-time to each arguments to the function. An offline partial evaluator may generate just one residual version of this function. This is called *monovariant* binding-time analysis. But then the most conservative binding-time must be associated with the function. This will lead to a residual function that is too dynamic. Instead, the binding-time analysis may distinguish the different points where a function is applied and analyse the function several times with each binding-time pattern. This is called *polyvariant* binding-time analysis. Naturally, a polyvariant binding-time analysis is less conservative than a monovariant one. Specializers with polyvariant binding-time analysis are able to generate more efficient residual programs at the cost of increased size of the residual programs.

In a program, the same function with the same binding-time pattern may be applied at several points to different static values. Residual versions of the function must be generated unless function applications are unfolded. A specializer may choose to generate just one residual function. This is called *monovariant* specialization. Or it may generate several residual functions, one for each set of static arguments. This is called *polyvariant* specialization. Again, polyvariant specialization generates more efficient residual programs at the cost of size.

4.6 Futamura Projections

Partial evaluation has several uses but one of the classical applications is to generate a compiler for a programming language given an interpreter for that language.

An interpreter for a programming language is a program I that takes the text of a source program S and the text of input i and produces output r obtained from interpreting S with input i

$$\text{Run}(I, \langle S, i\rangle) = r$$

Specializing the interpreter with respect to the source program gives, from equation (4.1)), a new program $I_s = \text{Run}(\text{PE}, \langle I, S\rangle)$ that satisfies (see Figure 4.2)

$$\text{Run}(I_S, i) = \text{Run}(I, \langle S, i\rangle) = r$$

$$\begin{tikzcd}
S \arrow[r, "\mathrm{Run}(\mathrm{PE},\langle I,S\rangle)"] \arrow[dr, "\mathrm{Run}(I,\langle S,i\rangle)"'] & I_S \arrow[d, "\mathrm{Run}(I_S,i)"] \\
 & r
\end{tikzcd}$$

Figure 4.2: Compiling by specializing

Thus, I_S is a standalone program, which is independent of the interpreter, or in other words, I_S the compiled version of S. With regard to speed and size it is our hope that in I_S, all computations that dependsonly on S have been performed.

To obtain the compiled version of another program S' from the same language we need to specialize the interpreter once more, this time with static input S'. Now partial evaluation proves useful since we are about to resume the previous application of PE to the same interpreter I (but with a different source program). Instead we can specialize the *specializer* with static input I. We obtain a program $\mathrm{PE}_I = \mathrm{Run}(\mathrm{PE}, \langle \mathrm{PE}, I\rangle)$ that satisfies (from equation (4.1)

$$\mathrm{Run}(\mathrm{PE}_I, S) = \mathrm{Run}(\mathrm{PE}, \langle I, S\rangle) = I_S$$

From the previous paragraph we have $\mathrm{Run}(\mathrm{PE}_I, S) = I_S$, the compiled version of S. Thus, PE_I is a compiler for the language accepted by the interpreter I.

In a similar manner we obtain a compiler from a different interpreter I' (possibly for a completely different language than the on accepted by the interpreter I) by specializing the specializer PE with respect to I'. Again, instead of generating a compiler by the same steps as before we may obtain a *compiler generator* that takes an interpreter as input and produces the corresponding compiler. We obtain the compiler generator $\mathrm{PE}_{\mathrm{PE}} = \mathrm{Run}(\mathrm{PE}, \langle \mathrm{PE}, \mathrm{PE}\rangle)$. It satisfies

$$\mathrm{Run}(\mathrm{PE}_{\mathrm{PE}}, I) = \mathrm{Run}(\mathrm{PE}, \langle \mathrm{PE}, I\rangle) = \mathrm{PE}_I$$

The three operations described in the previous three steps are called the *Futamura projections*. The following equations summarizes the three projections.

$$\begin{array}{lllll}
 & \mathrm{Run}(I, \langle S, i\rangle) & = r & & \\
 & & & & \\
1: & \mathrm{Run}(\mathrm{PE}, \langle I, S\rangle) & = I_s & \text{with } \mathrm{Run}(I_s, i) & = r \\
2: & \mathrm{Run}(\mathrm{PE}, \langle \mathrm{PE}, I\rangle) & = \mathrm{PE}_I & \text{with } \mathrm{Run}(\mathrm{PE}_I, S) & = I_s \\
3: & \mathrm{Run}(\mathrm{PE}, \langle \mathrm{PE}, \mathrm{PE}\rangle) & = \mathrm{PE}_{\mathrm{PE}} & \text{with } \mathrm{Run}(\mathrm{PE}_{\mathrm{PE}}, I) & = \mathrm{PE}_I
\end{array}$$

In each of the latter three steps we obtain programs that may be used to generate whatever was generated in the step immediately before (output or another program).

P	::=	(PGM V V E)	Program
E	::=	(VAR V)	Variable Reference
	|	(LIT C)	Constant Expression
	|	(LAM V E)	Lambda Abstraction
	|	(APP E E)	Function Application
	|	(CND E E E)	Conditional
	|	(OPR O (E ... E))	Primitive Operation
V	::=	a | b | ···	Symbols
C	::=	#t | #f | ··· | -1 | 0 | 1 | ···	Constants
O	::=	+ | * | ···	Primitive Operators

Figure 4.3: Abstract syntax of higher-order language

Note that $\mathrm{PE}_{\mathrm{PE}}$ is selfgenerating: $\mathrm{PE}_{\mathrm{PE}}(\mathrm{PE}) = \mathrm{PE}(\mathrm{PE}, \mathrm{PE}) = \mathrm{PE}_{\mathrm{PE}}$. This is the reason for not having a fourth Futamura projection.

4.7 Syntax-Directed Partial Evaluation

Kleene's S^m_n theorem states that the function that specializes programs is computable. That is, automatic specializers exist. Above it was shown how one could implement such a specializer in a higher-order language, in this case Scheme. The specializer given simply postpones the execution of the program to specialize. Thus the residual program is neither smaller nor faster than the original one.

How can a partial evaluator remove the computations that only depend on the static input? To remove a completely static computation the specializers must be able to evaluate the corresponding parts of the program. On the other hand, dynamic computation must occur in the residual program so the specializer must also be able to generate programs.

One class of partial evaluators, which I call syntax-directed partial evaluators, has been around for years[12]. They map the text of a program and the text of the static input into the text of the residual program. They consist of an interpreter, which evaluates the completely static expression, and a compiler, which constructs the residual program, which may contain already evaluated constants.

4.7.1 Example Higher-Order Language

To illustrate the concepts of syntax-directed partial evaluation I present an example online program specializer for a pure higher-order subset of Scheme[1]. The abstract syntax is given by the grammar in Figure 4.3.

The two variable arguments of a program are the static and dynamic arguments respectively. Thus, there must be exactly one static and one dynamic input to a program. Primitive

[1]Here, *pure* means a language without computational effects other than divergence

```
(define (ev abs)
  (match abs
    [('VAR x)        (error 'ev "Unbound variable: " x)]
    [('LIT c)        abs]
    [('LAM v e0)     abs]
    [('CND e0 e1 e2) (let ([v0 (ev e0)])
                       (if (val v0) (ev e1) (ev e2)))]
    [('APP e0 e1)    (let ([v0 (ev e0)]
                           [v1 (ev e1)])
                       (match v0
                         [('LAM v b)
                          (ev (subst b v v1))]
                         [_
                          (error 'ev "Not a function: " v0)]))]
    [('OPR op es)    `(LIT ,(apply (eval op)
                                   (map (lambda (e)
                                          (val (ev e))) es)))]))
```

The definitional interpreter for the higher-order language. The rest of the implementation, inlcuding the parser and the substitution function is left out. The function `val` extracts the Scheme value c from a literal `(LIT` c`)`.

Figure 4.4: Interpreter for higher-order language

operator may have any arity but lambda-abstraction are unary. These restrictions are only a matter of simplifying the grammar.

The language has a *call-by-value* semantics, that is, arguments to functions (and primitive operators) must be evaluated before the call takes place. Arguments to the primitive operators may not be functions. The result of applying a primitive operator is never a function. The core of a definitional interpreter for the language is shown in Figure 4.4.

The interpreter reduces expressions by the β_V-reduction rule [2][20, Chapter 12].

$$\texttt{(APP (LAM}\ \ x\ e\texttt{)}\ a\texttt{)} \longrightarrow e'$$

where a is a value and $e' = e[x := a]$ is e with all free occurrences of x replaced by a. This evaluation strategy is inefficient but is simple to change to a specializer later in Section 4.7.2

The interpreter leaves constants and lambdas as they are (i.e., it does not evaluate under lambdas). In conditionals, first the test is reduced to a primitive (boolean) value. Depending on this value either the then- or the else-branch is reduced (and returned). An application is evaluated by reducing the function and the argument. If the function does not reduce to a lambda an error is signalled. Otherwise all free occurrences of the formal arguments in the body are replaced by the value of the actual argument. Finally, evaluating an application of a primitive operator amounts to reducing each argument and then applying the corresponding

```
(define power
  `(program (n x)
     (let ([fix (lambda (f)
                  (let ([g (lambda (h)
                             (f (lambda (y) ((h h) y))))])
                    (g g)))])
       ((fix (lambda (loop)
               (lambda (n)
                 (if (= n 0)
                     1
                     (if (odd? n)
                         (* x (loop (- n 1)))
                         (sqr (loop (quotient n 2))))))))
        n))))
```

Figure 4.5: Example higher-order program (I)

operator, which is fetched from Scheme by the non-standard `eval` procedure.

The reduction function is applied to the body of a program whose formal parameters have been replaced by the actual values. The parser reads Scheme-like programs and produces abstract syntax trees described by Figure 4.3. It replaces `let`-constructs by equivalent applications of a lambda. These preliminary tasks are done by the procedure `sdev`, which is applied to the syntax of a program and two arguments. The example program in Figure 4.5 computes x^n and can be evaluated on actual arguments by

```
> (define (sqr x) (* x x))
> (sdev power 3 4)
= 64
> (sdev power 7 4)
= 16384
>
```

4.7.2 Small Syntax-Directed Specializer

The example specializer has a very simple strategy: (1) never specialize under lambdas, and (2) only evaluate an application of a function or operator if all sub-expressions (including the function itself) are static. It is an online partial evaluator so determining the binding-time of arguments is done during specialization. Static values are either constant or lambdas and dynamic values are all expressions other than these two. This distinction is not completely correct since a lambda-abstraction may contain dynamic expression and still be classified as static. Thus, even partially static lambdas are classified as static. As a result of the specializing strategy and the binding-time classification the specializer has two flaws:

- The specializer may loop in various ways: if the source program has no dynamic input the specializer acts as an interpreter and loops whenever the evaluater loops.

  Consider a partially static recursive function whose body is a conditional with a dynamic test, a then-branch that stops the recursion, and an else-branch that recurse with a "smaller" argument. Applying this function to a static argument means that the conditional will be residualized because it has a dynamic test. But then the function will be applied in the else-branch once again and the specializer will diverge.

  A solution would be only to apply completely static lambdas, but then some of the power of specialization is lost.

- Bodies of residualized lambda-abstractions are not specialized. Thus, if a lambda-abstraction occurs in a dynamic context it will not be specialized even if it is applied to a static argument later on. For example, specialing the program

  ```
  (program (s d)
    ((if d
         (lambda (x) (+ s 10))
         (lambda (x) (+ s x)))
     20))
  ```

  with respect to the static value `30` gives a residual program where the body of the first lambda-abstraction has not been specialized:

  ```
  (program (d)
    ((if d
         (lambda (_x0) (+ 30 10))
         (lambda (_x1) (+ 30 _x1)))
     20))
  ```

  To remove this shortcoming the partial evaluator could specialize under lambdas. But then the termination properties would be even worse: specializing a function that is never applied, but with a body that loops, will diverge.

The core of the specializer is shown in Figure 4.6. (it uses the same parser, unparser, and substitution functions as the interpreter). It extends the evaluator by allowing reduced values to be all kinds of expressions and not only constants and lambdas. Any expression other than these two are considered to be dynamic. Variables, constants, and lambdas specialize to themselves. A conditional specializes to one of the two branches if the test is static and to a conditional with specialized test, then-branch, and else-branch otherwise. An application of a function or an operator specializes to the reduced value if all arguments are static and to a specialized application otherwise.

The reason why I chose the β_V-reduction strategy in the interpreter (and hence the specializer) has to do with *lifting*: During specialization, static values may need to be inserted

```
(define (pe abs)
  (match abs
    [('VAR x)        abs]
    [('LIT c)        abs]
    [('LAM v e0)     abs]
    [('CND e0 e1 e2) (let ([v0 (pe e0)])
                       (if (stat? v0)
                           (if (val v0) (pe e1) (pe e2))
                           `(CND ,v0 ,(pe e1) ,(pe e2))))]
    [('APP e0 e1)    (let ([v0 (pe e0)]
                           [v1 (pe e1)])
                       (if (and (stat? v0) (stat? v1))
                           (match v0
                             [('LAM v b)
                              (pe (subst b v v1))]
                             [_
                              (error 'ev "Not a function: " v0)])
                           `(APP ,v0 ,v1)))]
    [('OPR op es)    (let ([vs (map pe es)])
                       (if (all stat? vs)
                           `(LIT ,(apply (eval op) (map val vs)))
                           `(OPR ,op ,vs)))]))
```

The control-flow of the specializer is determined by the *syntax* of the source program. Therefore this class of specializers are called syntax-directed partial evaluator.

Figure 4.6: Syntax-directed specialization in Scheme

(lifted) into dynamic contexts (i.e., branches of residual conditional expressions). Inserting non-functional values is easy: These are base constants, which evaluate to themselves. Thus, static base values may be inserted into the program without changes. Inserting a static higher-order values into dynamic contexts is different. Using the β_V-reduction strategy functions are represented by syntax. Thus, inserting (lifting) functions is easy: they are already represented by syntax, which may be inserted into dynamic contexts.

The main procedure of the specializer is called `sdpe`. It takes a program and the static argument and returns a specialized program (unless specialization diverges). As an example we may specialize the power program from Figure 4.5 with a static exponent

```
> (sdpe power 3)
= (program (x) (* x (sqr (* x 1))))
> (sdpe power 7)
= (program (x) (* x (sqr (* x (sqr (* x 1))))))
>
```

The specialized programs are more efficient than the general program and they are certainly smaller. Thus, in this case partial evaluation is shown at its best. However, if we change the power-program such that the exponent is dynamic and we specialize with a static quotient we obtain a 40-line residual program.

4.7.3 The Similix Partial Evaluator

Similix is a self-applicable, offline partial evaluator implemented by Anders Bondorf, Jesper Jørgensen and Olivier Danvy [3]. Self-applicable means that Similix can specialize itself. Thus, using Similix one can generate target programs (by specializing an interpreter with respect to a source program), generate compilers (by specializing Similix with respect to an interpreter), and generate compiler-generators (by specializing Similix with respect to Similix).

Similix has a monovariant binding-time anaysis and polyvariant specialization. Thus, the most general binding-time pattern is used for a function. Similix have binding-time–monovariant constructors that handles partially static structures.

Similix specializes a large subset of Scheme including functions with computational effects. Applications of effect-full functions should neither be removed, duplicated, nor reordered as this would change the semantics of the specialized program.

Similix is structured into phases. A *parser* reads concrete syntax and passes abstract syntax to a *preprocessor*. This consists of several *analyses* including

- the binding time analysis;

- an evaluation-order dependency analysis. This finds effect-full expressions, which should occur exactly once in the residual program;

- an occurrence-counting analysis, which finds `let`-expressions that has bound variables that occur more than once in the body. These may not be unfolded as it may cause duplication of code; and

- a redundant `let`-elimination anaysis, which removes redundant `let`-expressions inserted by the front-end.

The output from the preprocessor is passed to the specializer proper. After specialization residual programs are reduced further by a *postprocessing phase*. This phase improves the residual program by some "last-minute optimization". For example, it unfolds some calls and some `let`-expressions.

As a result Similix is an effective specializer. Chapter 5 contains our main use of partial evaluation. There we shall see a non-trivial application of Similix where the residual programs are two orders of a magnitude faster than the original programs.

4.8 Type-Directed Partial Evaluation

Until recently the technique described in the previous section was the only known approach to program specialization. In the foreword of the first complete book on partial evaluation, which dates from 1993, it is put that "partial evaluation ... works with *program texts* rather than mathematical functions" [12]. This section presents a new approach to program specialization that does work with mathematical functions (or rather functions implemented in higher-order languages) and *not* program texts.

The new approach, called type-directed partial evaluation, is based on a normalization of the lambda calculus. It is most conveniently described in a two-level lambda calculus [16]. This is an extension of the lambda calculus in which each operator (abstraction and application) has two variants, a *static* (overlined) and a *dynamic* (underlined). The syntax of two-level lambda terms is as follows

$$\begin{array}{lcll}
\text{TLT} & ::= & \text{C} & \text{Constants} \\
 & | & \text{V} & \text{Variables} \\
 & | & \overline{\lambda}\text{V. TLT} & \text{Static Abstraction} \\
 & | & \underline{\lambda}\text{V. TLT} & \text{Dynamic Abstraction} \\
 & | & \text{TLT } \overline{@} \text{ TLT} & \text{Static Application} \\
 & | & \text{TLT } \underline{@} \text{ TLT} & \text{Dynamic Application}
\end{array}$$

The infix operator @ is used to denote applications. Variables are not explicitly overlined or underlined since their annotations can be determined by the lambda abstraction that binds them (λ-terms must be closed). A term with only statically(dynamically) marked operators is refered to as *completely static*(*completely dynamic* respectively). *Static reduction* is β_V-reduction on static β-redeces:

$$(\overline{\lambda}x.\ e)\ \overline{@}\ a \longrightarrow e'$$

where $e' = e[x := a]$ is e with each free occurrence of x replaced by a while not capturing the free variables of a. (We will actually consider static reduction also to include δ-reduction, i.e., reduction with respect to some predefined operators). For example, we may have the following static reduction

$$\underline{\lambda}z.\ ((\overline{\lambda}x.\ x\ \underline{@}\ z)\ \overline{@}\ \underline{\lambda}y.\ y) \longrightarrow \underline{\lambda}z.\ ((\underline{\lambda}y.\ y)\ \underline{@}\ z)$$

The normalization is described in terms of two operations, reify ($\downarrow$) and reflect ($\uparrow$) presented in Figure 4.7. They are applied to a two-level term and are directed by the type of the term. (Hence the term *type-directed*). Reifying a completely static term with respect to its type yields a two-level term with the same type. For example

$$\downarrow^{b_0 \to (t_0 \to y_1) \to t_1} \overline{\lambda}x.\ \overline{\lambda}f.\ f\ \overline{@}\ x = \underline{\lambda}g_0.\ \underline{\lambda}g_1.\ ((\overline{\lambda}x.\ \overline{\lambda}f.\ f\ \overline{@}\ x)\ \overline{@}\ g_0)\ \overline{@}\ (\overline{\lambda}g_2.\ g_1\ \underline{@}\ g_2)$$

Statically reducing this two-level term gives a completely dynamic term

$$\begin{array}{lllll}
 & \downarrow^{\mathrm{b}} v & = & v & \text{where } b \text{ is a base type} \\
(1) & \downarrow^{t_0 \to t_1} v & = & \underline{\lambda} x.\ \downarrow^{t_1} (v\ \overline{@}\ (\uparrow_{t_0} x)) & \text{where } x \text{ is fresh} \\
 & & & & \\
 & \uparrow_{\mathrm{b}} e & = & e & \text{where } b \text{ is a base type} \\
(2) & \uparrow_{t_0 \to t_1} e & = & \overline{\lambda} x.\ \uparrow_{t_1} (e\ \underline{@}\ (\downarrow^{t_0} x)) & \text{where } x \text{ is fresh}
\end{array}$$

Types are either base types or unary function types. However, in Section 4.8.2 we shall discuss how to add other types, most notably functions of any constant arity and products.

$$\begin{array}{llll}
\text{Type} & ::= & \text{Base} & \text{Base Type} \\
 & \mid & \text{Type} \to \text{Type} & \text{Function Type}
\end{array}$$

Figure 4.7: Type-directed residualization

$$\underline{\lambda} g_0.\ \underline{\lambda} g_1.\ g_1\ \underline{@}\ g_0$$

This term does not only have the same type as the original (completely static) term, but it also reduces (via β_V-reduction, after erasing the dynamic annotations) to a term that is extensionally equal to the original term $\overline{\lambda} x.\ \overline{\lambda} f.\ f\ \overline{@}\ x$. This is where partial evaluation enters the scene. Assume that the original completely static term contains a β redex. Then the static reduction will process this redex leaving only the result in the residual term. Yet the two functions are extensionally equal.

To stick to the notation from Section 4.1 we represent a program by (the text of) a lambda abstraction of two arguments, the static and dynamic input (a curried abstraction since until now all functions are unary): $P = \overline{\lambda} s.\ \overline{\lambda} d.\ e$. To specialize a program with respect to some static input i we statically apply the function to the input and reify the application into a two-level term $t = \downarrow^{d \to e} (\overline{\lambda} s.\ \overline{\lambda} d.\ e)\ \overline{@}\ i$, which we reduce statically. The result is a (completely dynamic) term, that is, the text of the residual program.

Note that while residualizing, reify constructs dynamic abstraction and deconstructs static abstractions. Conversely, reflect constructs static abstractions and deconstruct dynamic abstractions. This pattern is repeated when the language is extended in Section 4.8.2: reify (reflect) construct dynamic (static) values and deconstructs static (dynamic) values. Furthermore, we can annotate each part of the type with a *polarity*, either + or −, by

$$\begin{array}{rcl}
\text{Pol}, \text{Pos}, \text{Neg} & :: & \text{Type} \to \text{Type}^{\text{P}} \\
\text{Pol}(T) & = & \text{Pos}(T) \\
 & & \\
\text{Pos}(\text{B}) & = & B^{+} \\
\text{Pos}(T_0 \to T_1) & = & \text{Neg}(T_0)\ \to^{+}\ \text{Pos}(T_1)
\end{array}$$

$$\begin{array}{rcl} \text{Neg}(\text{B}) & = & B^- \\ \text{Neg}(T_0 \to T_1) & = & \text{Pos}(T_0) \ \to^- \ \text{Neg}(T_1) \end{array}$$

Pos and Neg have the same recursive descent of the type as reify and reflect. Thus, reify is applied to types with positive annotation and reflect is applied to types with negative annotations.

Note also that the *type* of the arguments v and e in $\downarrow^t v$ and $\uparrow_t e$ depends on the type-argument t. Thus, reify and reflect as presented here (and therefore type-directed residualization as presented here) cannot be implemented in a statically typed language. However, later in this report (Section 6.5.5) I derive a statically typed implementation from the implementation given here. The notion of polarity plays a central role when this implementation is derived.

4.8.1 Small Type-Directed Specializer

Below, an implementation of a type-directed partial evaluator in Scheme is presented. It is based on the normalization of the lambda-calculus presented above. One approach is to represent both static and dynamic terms as a data type. Then reify and reflect constructs two-level terms in this representation. A static reducer is applied afterwards to yield the dynamic residual term. But we will use a more efficient approach where static reduction is done on the fly *by the Scheme evaluator*. Thus, there is no intermediate representation of static expressions. We represent the static (overlined) terms as higher-order values (and not as values of an abstract data type).

The type-directed partial evaluator is shown in Figure 4.8. It closely corresponds to Figure 4.7 but instead of static values of a data type it represents static values by higher-order Scheme values. These are reduced immediately by the Scheme evaluator. Since Scheme variables are bound lexically there is no need to generate a fresh symbol when static lambdas (i.e., higher-order functions) are generated.

When representing static terms by compiled (possibly higher-order) values one must remember that free variables in the term are evaluated at specialization time. In function positions in applications they may be used to denote primitive operators evaluated during static δ-reduction. But if a variable is intended to occur in the residual program then it must be abstracted in the original term and it must be reflected by the type.

In order to specialize the power program from section 4.7.1 we must abstract it with the free (residual) variables. The result is shown in Figure 4.9 along with the type of the term to residualize. Note that we can use any expressions we like as long as they have a type (or are reduced statically). In the example we use both named `let` and `cond`.

We obtain specialized versions of the power program as in the following session.

```
> (tdpe (power 3) type)
= (lambda (x0)    ; sqr
    (lambda (x1)  ; mult
      (lambda (x2)
(define (reify t v)
  (match t
    ['base
     v]
    [(t0 '-> t1)
     (let ([x (tdpe-gensym)])
       `(lambda (,x)
          ,(reify t1 (v (reflect t0 x)))))]))

(define (reflect t e)
  (match t
    ['base
     e]
    [(t0 '-> t1)
     (lambda (x)
       (reflect t1 `(,e ,(reify t0 x))))]))

(define (tdpe v t)
  (reify t v))
```

Types are represented by Scheme data Type. We consider only a set of base types (like integers, booleans, etc.) and function types

Type	::=	`base`	Base Type
	|	(Type -> Type)	Function Type

Figure 4.8: Type-directed partial evaluation in Scheme

```
        ((x1 x2) (x0 ((x1 x2) 1))))))
> (tdpe (power 7) type)
= (lambda (x0)    ; sqr
    (lambda (x1)  ; mult
      (lambda (x2)
        ((x1 x2) (x0 ((x1 x2) (x0 ((x1 x2) 1))))))))
>
```

The residual terms obtained here corresponds to the residual programs obtained with the syntax-directed specializer from Section 4.6. The only differences are the representations of programs. The fresh names, which both kinds of specializers generate, might also differ.

4.8.2 The TDPE System

The TDPE system is a type-directed partial evaluator along the lines described above. It is implemented by Olivier Danvy [7, 8]. It extends the example specializer we have presented

```
(define power
  (lambda (n)
    (lambda (sqr)
      (lambda (mult)
        (lambda (x)
          (let loop ([n n])
            (cond [(= n 0)
                   1]
                  [(odd? n)
                   ((mult x) (loop (- n 1)))]
                  [else
                   (sqr (loop (quotient n 2)))]))))))))
(define type
  `((base -> base)                 ; sqr
     -> ((base -> (base -> base))  ; mult
     -> (base -> base))))
```

Figure 4.9: Example higher-order program (II)

with n-ary function types, product types, preservation of computational effects, boolean type, and sum types. These extensions may be added to the example specializer given here. (I will only describe how to add some of them).

Functions of n arguments. To extend the example specializer with n-ary functions, types are extended with n-ary function types

Type ::= ((Type ...) => Type) n-ary Function Type

and one clause is added to each of `reify` and `reflect`

```
(define (reify t v)
  (match t
    ...
    [(ts '=> t1)
     (let ([xs (map (lambda (_) (tdpe-gensym)) ts)])
       `(lambda ,xs
          ,(reify t1 (apply v (map reflect ts xs)))))]))

(define (reflect t e)
  (match t
    ...
    [(ts '=> t1)
```

```
      (lambda xs
        (reflect t1 `(,e ,@(map reify ts xs))))]))
```

Product types. Extending the example specializer with product types is also simple. Products are represented by Scheme pairs. The constructor is the procedure `cons` and the deconstructors are `car` and `cdr`. We add a type to the abstract syntax of types

$$\text{Type} \quad ::= \quad (\text{Type} * \text{Type}) \quad \text{Product Type}$$

and one clause is added to each of `reify` and `reflect`. To reify upon a pair amounts to construct an expression denoting a value that is structural equal to the pair. Calling `reify` recursively on the contents of the pair gives the expressions for the elements in the pair. These are then inserted in an expression that constructs the pair.

To reflect upon a pair is dual to reifying it: A pair is constructed from an expression by calling `reflect` recursively on expressions that evaluate to the first and second element of the pair.

```
(define (reify t v)
  (match t
    ...
    [(t0 '* t1)
     `(cons ,(reify t0 (car v)) ,(reify t1 (cdr v)))]))

(define (reflect t e)
  (match t
    ...
    [(t0 '* t1)
     (cons (reflect t0 `(car ,e)) (reflect t1 `(cdr ,e)))]))
```

Computational effects. Applications of function with computational effects (including functions that diverge) should neither be removed, reordered, nor duplicated in the residual program as this might change the semantics. The TDPE system handles functions with computational effect if they are accounted for in the type. These functions are recognized by annotated function types (`-!>`). Reifying an effectful function is similar to reifying a pure function. To reflect upon an effectful function a static function is generated. When applied (statically) it generates a dynamic application of the effectful function. Special effort must be taken to ensure that the application occurs precisely once in the residual program. Therefore a dynamic variable is bound to the result of the application by a residual `let`-expression.

Booleans. To reify a boolean value amounts to test its value and construct either `#t` or `#f`. (The boolean constructors are `#t` and `#f` and the deconstructor is the conditional expression).

```
(define (reify t v)
  (match t
    ...
    ['bool
     (if v #t #f)]))
```

Perhaps counter-intuitively, reflecting a boolean expression is not dual (although it can be made by changing reify appropriately). Reflecting a boolean value should yield a dynamic conditional expression. Consider the static value `(lambda (b) (if b 0 1))` and assume we reify it with respect to its type, Bool → Int. In reify, this function shall be applied to a boolean value. To construct the correct dynamic term the function must be applied to both `#t` and `#f` in order to obtain both branches. Thus, this restricted case can be handled by

```
(define (reflect t v)
  (match t
    ...
    [('bool '-> 'base)
     (let ([x (tdpe-gensym)])
       `(lambda (,x)
          (if ,x ,(v #t) ,(v #f))))]))
```

Reifying `(lambda (b) (if b 0 1))` gives correctly `(lambda (x0) (if x0 0 1))`. When reflecting upon a boolean, a dynamic conditional expression must be generated. (In the example, the dynamic conditional stems from a specialized instance of reflect). But the branches of the conditional cannot be generated from the information available in reflect. (The only arguments to reflect are the type and the dynamic boolean expression). Instead, the required information is available in the most recent instance of reify. In the implementations of the TDPE system this information is accessed by delimiting and abstracting contexts (shift and reset).

Sum types. Sum types are handled by mechanisms similar to those for booleans.

4.8.3 Online Primitive Operations

An extension to the TDPE system have arisen during my work in the field. It is based on the following observations

- When specializing by type-directed partial evaluation one is often forced to abstract variables that corresponds to usual Scheme primitives that have a top-level definition. They are not abstracted in normal Scheme programs. A similar notion of top-level definition in type-directed partial evaluation will simplify terms using top-level variables.
- Often one operator has two different binding times at two different places in the program. Assume that `s` is static input and `d` is dynamic input. Then in the following term the leftmost occurrence of `+` should be residualized and the rightmost should be reduced at specialization time.

```
(lambda (s) (lambda (d) (+ (+ s 42) d)))
```

To obtain the effect of polyvariant binding-time analysis we introduce distinct symbols at the two application points of +. We replace the leftmost use of + by an abstracted function before specializing:

```
(lambda (s) (lambda (add) (lambda (d) (add (+ s 42) d))))
```

In this term there are two versions of the addition procedure, one completely static and one dynamic. This trick will inevitably lead to confusions and errors when large program are specialized.

Both shortcomings have a solution hinted in earlier work [7, Section 4.5]. The symbol add is bound (at top-level) to an function that *probes* its arguments for static-reduction opportunities. It either expands to a dynamic application or it performs the actual addition depending on the binding time of the arguments. This is a binding-time–polyvariant primitive.

```
(define (add x y)
  (if (and (number? x) (number? y))
      (+ x y)
      `(add ,x ,y)))
```

We do not have to abstract this procedure in source programs, still, it will occur in the residual program. Furthermore, the static addition is performed at specialization time and the dynamic addition is residualized *using the same procedure*.

```
> (tdpe ((lambda (s)
           (lambda (d)
             (add (add s 42) d))) 100)
        '(base -> base))
= (lambda (x0) (add 142 x0))
>
```

We can write a function add that in some cases reduces to a static value even though one of its arguments is dynamic. Again, this is a binding-time–polyvariant primitive. This function reflects the equalities $x + 0 = x$ and $0 + y = y$.

```
(define (add x y)
  (cond [(and (number? x) (number? y))
         (+ x y)]
        [(and (number? x) (zero? x))
         y]
        [(and (number? y) (zero? y))
         x]
        [else
         `(add ,x ,y)]))
```

Using this, even partially static application of the addition function may be reduced

```
> (tdpe ((lambda (s)
           (lambda (d)
             (add (add s 42) d))) -42)
         '(base -> base))
= (lambda (x0) x0)
>
```

In a similar way we can define online primitive that multiplies and squares

```
(define (mult x y)
  (cond [(and (number? x) (number? y))
         (* x y)]
        [(and (number? x) (= x 0))
         0]
        [(and (number? x) (= x 1))
         y]
        [(and (number? y) (= y 0))
         0]
        [(and (number? y) (= y 1))
         x]
        [else
         `(mult ,x ,y)]))

(define (sqr x)
  (if (number? x)
      (* x x)
      `(sqr ,x)))
```

We can specialize the power program without abstracting multiplication and squaring operations in an initial environment. Furthermore, we change the `mult` to a binary operator

```
(define power
  (lambda (n)
    (lambda (x)
      (let loop ([n n])
        (cond [(= n 0)
               1]
              [(odd? n)
               (mult x (loop (- n 1)))]
              [else
               (sqr (loop (quotient n 2)))])))))
```

We obtain a slightly better result when specializing than before. The multiplication `(mult x 1)` is reduced to the equivalent `x`. Note that `-`, `=`, `quotient`, and `odd?` are applied to static arguments. Therefore it is not necessary to implement them as online primitives. We obtain a more readable residual program since the online primitives have the names of their top-level definitions.

```
> (tdpe (power 3) '(base -> base))
= (lambda (x0) (mult x0 (sqr x0)))
> (tdpe (power 7) '(base -> base))
= (lambda (x0) (mult x0 (sqr (mult x0 (sqr x0)))))
>
```

Defining procedures that probe their arguments as above is cumbersome: Four clauses are required in the procedure `add` to reflect only two equations. Furthermore, the representation of static and dynamic arguments should be transparent to the user who defines the online primitives. Therefore I have suggested a specification of online primitives using *pattern matching*. The TDPE system allows online primitives to be defined by

`(define-primitive` $x\ t\ s\ d$`)`

where x is the name of the primitive, t is its type, s is the function to use when the primitive is applied to all static arguments, and d, which is optional, is the function to use when some arguments are dynamic. A primitive without the dynamic function always expands into an application when applied to dynamic arguments.

The extension is independent of the definition of online primitives and fits naturally into the dynamic version. It allows (static or dynamic) arguments to be *matched* against a sequence of *patterns*. When the arguments matches a pattern a set of bindings of *pattern variables* is used to expand a *template*. The expanded template goes into the residual program. The new construct is closely related to the pattern matching of syntax in the macro system of Chez Scheme.

The new construct is an expression. Most often it will be used in the body of the dynamic version of the primitive. It has form

`(case-syntax (`x `...)` *clause* `...)`

The x's are variables denoting either static or dynamic values. These are most often the actual arguments of the dynamic version of the online primitive. *clauses* is a list of clauses each containing one pattern for each variable, an optional fender, and a template. The fender is a boolean expression describing extra requirements to be satisfied, apart from matching the pattern. The syntax of patterns follow the grammar given in Figure 4.10. The syntax of templates is also given in Figure 4.10.

Assume that the body of the dynamic version of a online primitive is a `case-syntax` expression. When the primitive is applied the following steps are taken. If all arguments are

Pattern	::=	`(P ...)`	Pattern List
Template	::=	`(with ([V E] ...) Template)`	Bindings
	|	P	Form
P	::=	V	Pattern Variable
	|	C	Self evaluating constant
	|	`(quote L)`	Quoted Constant
	|	`(F P ... P)`	Function Application
F	::=	`a` | `b` | $\cdots$	Function Names

The same syntax P is used for patterns and templates. E denotes any Scheme expression.

Figure 4.10: Syntax of patterns and templates

static then the static version is applied. It yields a static value, which appears in the residual program.

If one or more arguments are dynamic then the dynamic version is applied. The steps taken when evaluating the `case-syntax` expression are as follows. The clauses are processed in turn. If all patterns in the list of patterns matches the actual values (x's) then bindings of pattern variables are obtained. The template is processed with these bindings unless the clause has a fender that evaluates to false. (Fenders may use the bindings of pattern variables). The template is processed by substituting the values bound to the pattern variables. Prior to this expansion, some pattern variables may be bound to the result of evaluating a Scheme expression.

The `case-syntax` construct provides a way of implementing online primitives in a way that is independent of the representation of dynamic expressions. It also allows primitives to be described in a convenient notation. It has, though, less computational power than if dynamic expressions where deconstructed manually. This stems from the restricted syntax of patterns. For example, no pattern may occur in function position in an application-pattern.

As an example, the following implements `add` as above

```
(define-primitive add ((I I) => I)
  +
  (lambda (x y)
    (case-syntax (x y)
      [(0 y) y]
      [(x 0) x]
      [(x y) (p x y)])))
```

This may be used in a session with the type-directed specializer

```
> (tdpe ((lambda (s) (lambda (d) (p (p s 42) d))) 100) '(a -> a))
(lambda (a0) (p 142 a0))
```

```
> (tdpe ((lambda (s) (lambda (d) (p (p s 42) d))) -42) '(a -> a))
(lambda (a0) a0)
>
```

More complex example can be written. It is possible to match on nested function applications. And arbitrary function applications can be constructed. However, this is an area for further investigations. For example, it is currently not possible to construct other dynamic expressions (for example conditionals). Furthermore, the `case-syntax` construct introduces a *rewriting system* on dynamic expressions. The influence that this has on partial evaluation has not been investigated.

4.9 Syntax- and Type-Directed Partial Evaluation

The two different approaches to partial evaluation have similar goals: to evaluate completely static expression and to residualize dynamic expression. But their means are different. Syntax-directed specializers are non-standard interpreters and processes program text. Type-directed specializers normalizes static higher-order values. Both kinds of specializers have advantages and disadvantages.

Source Language. Syntax-directed specializers have a fixed grammar for the *syntax* of accepted programs. Type-directed specializers have a fixed grammar for the *type* of accepted programs. The user needs only recognise a 10 line grammar of types. (In contrast, the grammar that describes the legal input to Similix takes up four pages and more than 100 lines).

Furthermore, the user may add new syntactic extensions without conflicting with the source language of the specializer. Adding new syntax to the source language is not possible for syntax-directed specializers unless they contain macro systems. (Alternatively, programs may be expanded to a core language by the macro system of the underlying Scheme implementation, if it has one).

In contrast to type-directed specializers, there are syntax-directed specializers for both imperative and logic programming languages[12].

Implementation language. Type-directed specializers that perform static reduction on the fly can only be implemented in higher-order languages. Furthermore, some restrictions are imposed by type-directed specializers implemented in static typed higher-order languages. (This will be discussed when in Section 6.5.5 where a statically typed type-directed partial evaluator is derived).

Size of Specializer. Syntax-directed specializers must have one case for each kind of *syntax*. Type-directed specializers must have one case for each kind of *type* (actually two, one in reify and one in reflect), Consequently, type-directed specializers are smaller than syntax-directed ones. (Figure 4.7 defines a specializer in four lines).

More importantly, there is essentially only one way to implement type-directed specializers because there is only one way to handle the different types. As a consequence, the specialization strategy of type-directed partial evaluators cannot be controlled. In contrast, syntax-directed specializers can (and has been) implemented in a variety of ways and with different specialization strategies.

Efficiency. Syntax-directed specializers traverse the *syntax* of the source program. Type-directed specializers traverse the *type* of the source program (applying the higher-order functions in the source program to the appropriate values). Consequently, type-directed specializers appear to be more efficient than syntax-directed ones.

However, the interpretive overhead of specialization may be removed by the second Futamura projection: $\text{PE}_P = \text{Run}(\text{PE}, \langle \text{PE}, P\rangle)$. By this we obtain a specialized specializer for program P. For any input I to P we have namely that $\text{Run}(\text{PE}_P, I) = \text{Run}(\text{PE}, \langle P, I\rangle)$. But there is no interpretive overhead involved when running PE_P. In Chapter 5 we investigate this in practical experiments.

Effectivity. Both kinds of specializers has the same goals: to reduce static computations and to residualize dynamic computations. In most cases the residual programs obtained by the two kind of specializers are essentially the same (textually. They are extensionally equal). However, in contrast to type-directed partial evaluators, Similix has polyvariant specialization and is therefore more powerful.

Conditional expression and boolean types are handled differently in Similix and type-directed partial evaluators. Similix handles conditional expressions as in the example specializer of Section 4.7.2: If the test is static then the expression is reduced to the appropriate branch. Otherwise a residual conditional expression is generated. The type-directed specializer moves the entire context of the dynamic conditional into each branch. Thus, computations that would otherwise take place "across" conditional expressions may be reduced by type-directed specializers but not by syntax-directed specializers. For example, Similix specializes the program

```
(define (f s d)
  (+ 100 (if d s 42)))
```

where `s` is the static value `5` and `d` is dynamic, to

```
(define (f-0 d_0)
  (+ 100 (if d_0 5 42)))
```

The type-directed specializer moves the context of the conditional (that is, the context `(lambda (z) (+ 100 z))`) into each branch. A better result is obtained

```
> (define-base-type Int)
> (residualize ((lambda (s)
                  (lambda (d)
                    (+ 100 (if d s 42)))) 5)
              '(Bool -> Int))
= (lambda (b0) (if b0 105 142))
>
```

However, duplicating a context may lead to an exponential growth of the residual program. Consider the program

```
(define (f g b0 b1)
  (g (if b0 10 20) (if b1 30 40)))
```

If all arguments are dynamic then Similix performs no static computations and leaves the program as it is. The type-directed specializer duplicates a context twice (there are two boolean arguments) and generates the residual program

```
(lambda (x0 b0 b1)
  (if b0
      (if b1
          (x0 10 30)
          (x0 10 40))
      (if b1
          (x0 20 30)
          (x0 20 40))))
```

One way to avoid code duplication is to use an alternative boolean type and implement special purpose conditionals. Scheme booleans are used to represent special purpose boolean values but they must be declared as base types. Conditionals expressions are transliterated into calls to a conditional procedure. It takes a boolean value and two thunks (procedures of no arguments) and applies the first or the second thunk depending on the boolean.

The following example is implemented as an online primitive. It reduces static branches if the test is dynamic.

```
(define (test b t0 t1)
  (if (boolean? b)
      (if b
          ,(if (procedure? t0) (t0) `(,t0))
          ,(if (procedure? t1) (t1) `(,t1))
      `(if ,b
           ,(if (procedure? t0) (t0) `(,t0))
           ,(if (procedure? t1) (t1) `(,t1)))))
```

We can repeat the previous exercise with this new representation of the conditional

```
> (define (f g b0 b1)
    (g (test b0 (lambda () 10) (lambda () 20))
       (test b1 (lambda () 30) (lambda () 40))))
> (tdpe f '((((base base) => base) base base) => base))
= (lambda (x0 x1 x2) (x0 (if x1 10 20) (if x2 30 40)))
>
```

Improving Source Programs. *Binding-time improvements* are modifications performed by hand on the original program before specialization. Their purpose is to yield "better" binding-time analysis making offline specialization yield better results. A simple binding-time improvement is to rewrite `(+ s (+ 10 d))` to `(+ (+ s 10) d)` when `s` is static and `d` is dynamic. The former has two dynamic application where the latter has one dynamic and one static application. Yet, the two programs are extensionally equal (provided `+` is bound to an associative binary procedure).

Binding-time improvements are classical optimization in the field of syntax-directed partial evaluation. They have less use in type-directed partial evaluation since, most often, a binding time improvement coerces dynamic computations to static computations without changing the functionality of the program: They are identity functions. But insertion of identity functions does not matter to type-directed specialization.

Partially Static Structures. During specialization some (compound) values may have a static structure with dynamic contents. For example, a list may have static length but dynamic contents. Operations on such values may only require knowledge of the static structure. For example, appending to a partially static list does not require knowledge of the elements in the list.

Let us first examine how Similix allows partially static structures to be used. Similix has special constructors to define partially static types. We may define a partially static list by

```
(defconstr (ps_nil)
           (ps_cons ps_car ps_cdr))
```

This defines a nullary constructor `ps_nil` and a binary constructor `ps_cons`. It also defines accessor functions (or deconstructors) `ps_car` and `ps_cdr` for values constructed by `ps_cons`. The function `ps_append` appends partially static lists given in this representation. (Actually the second argument need not be partially static). The main function `f` appends a completely dynamic value onto a partially static list.

```
(define (f a b c xs)
  (ps_append (ps_cons a (ps_cons b (ps_cons c (ps_nil))))
             xs))

(define (ps_append xs ys)
  (if (ps_nil? xs)
      ys
      (ps_cons (ps_car xs)
               (ps_append (ps_cdr xs) ys))))
```

The main function `f` is specialized with all dynamic arguments. The first argument is not completely static, an still the specializer is able to generate a residual program that contains unfolded calls to `ps_append`

```
(define (f-0 a_0 b_1 c_2 xs_3)
  (ps_cons a_0 (ps_cons b_1 (ps_cons c_2 xs_3))))
```

Compare this to a program that does the same but with the traditional list operations (i.e., `cons`, `car`, `cdr`, and `null?` instead of `ps_cons`, `ps_car`, `ps_cdr`, and `ps_nil?`). These operations requires completely static arguments in order to be reduced. Therefore we obtain the residual program

```
(define (f-0 a_0 b_1 c_2 xs_3)
  (define (mappend-0-1 xs_0 ys_1)
    (if (null? xs_0)
        ys_1
        (cons (car xs_0)
              (mappend-0-1 (cdr xs_0) ys_1))))
  (mappend-0-1 (list a_0 b_1 c_2) xs_3)))
```

In the residual program no computations has been reduced.

In the TDPE implementation of a type-directed specializer the user can define *records* that represents structural entities. The record-constructors are binding-time–polyvariant. Records can be used to represent partially static structures. For example, partially static lists can be defined by

```
(define-record (ps_nil))
(define-record (ps_cons x y))
```

This corresponds to the definition of partially static lists in Similix except that the constructors are named `make-ps_nil` and `make-ps_cons` and that deconstructors and predicates are merged into a case-like construct. Thus we define

```
(define (ps_append xs ys)
  (case-record xs
    [(ps_nil)      ys]
    [(ps_cons x y) (make-ps_cons x (ps_append y ys))]))

(define (f a b c xs)
  (ps_append (make-ps_cons a
(define-primitive ps_cons ((V Vs) => Vs)
  cons)

(define-primitive ps_nil (() => Vs)
  (lambda () '()))

(define-primitive ps_null? (Vs -> V)
  null?
  (lambda (xs)
    (case-syntax (xs)
      [((ps_nil))            #t]
      [((ps_cons y ys))      #f]
      [(xs)                  (ps_null? xs)])))

(define-primitive ps_car (Vs -> V)
  car
  (lambda (xs)
    (case-syntax (xs)
      [((ps_cons y ys))      y]
      [(xs)                  (ps_car xs)])))

(define-primitive ps_cdr (Vs -> Vs)
  cdr
  (lambda (xs)
    (case-syntax (xs)
      [((ps_cons y ys))      ys]
      [(xs)                  (ps_cdr xs)])))
```

Figure 4.11: Partially static list using online primitives

```
                (make-ps_cons b
                   (make-ps_cons c (make-ps_nil))))
             xs))
```

And then we may specialize the same program as with Similix by

```
> (tdpe f '((V V V Vs) => Vs))
(lambda (v0 v1 v2 v3)
  (make-ps_cons v0 (make-ps_cons v1 (make-ps_cons v2 v3))))
>
```

Partially static structures may be represented differently in a type-directed setting. I present here a technique that implements partially static lists by online primitives, which are binding-time-polyvariant. The idea is to represent a partially static structure by *dynamic applications*. Partially static lists may be defined by the primitives in Figure 4.11.

For example, a one element list with dynamic contents x is represented by the dynamic application `(ps_cons` x `(ps_nil))`. If we define `f` and `ps_append` as above we can repeat the exercise.

```
> (tdpe f '((V V V Vs) => Vs))
= (lambda (v0 v1 v2 vs3)
    (ps_cons v0 (ps_cons v1 (ps_cons v2 vs3))))
```

The primitives that deconstructs partially static lists, `ps_car` and `ps_cdr`, matches on the dynamic *applications* (as an expression) of `ps_cons`. They extract the *arguments* (as expressions) from the application. These are either values (the static parts of the structure) or expressions (the dynamic parts of the structure).

4.10 Summary and Conclusion

We have seen how partial evaluation may be used to compile a source program into a target program given an interpreter for the source language written in the target language. Two different approaches to monovariant program specialization, a syntax-directed and a type-directed specializer, have been described, including two example specializers. We notice that type-directed partial evaluators appear to have a *canonical* implementation whereas syntax-directed partial evaluators are implemented by more *ad hoc* methods. This means that syntax-directed partial evaluation is more flexible when it comes to extensions and specialization strategies.

Finally, we have seen how to represent partially static structures in existing implementations of both syntax-directed and type-directed specializers.

Chapter 5

Compiling Actions by Partial Evaluation

This section reports on the experiments I have conducted on "the compilation of action notation by partial evaluation". Originally, the first part of these experiments were done a few years ago by Anders Bondorf and Jens Palsberg [5]. They used a syntax-directed partial evaluator on an interpreter for a subset of action notation. Their experiments are interesting since (1) it was not a priori known how well partial evaluation applied to interpreters for action notation, (2) the compiler they obtain is efficient, and (3) the compiler is effective: it produces efficient target programs.

I have repeated their exercise with both syntax-directed specialization and the new technique, type-directed specialization. These experiments are interesting as they provide a good base for comparing syntax-directed and type-directed partial evaluation.

Section 5.1 motivates and describes the static subset of action notation implemented by the interpreters. The differences and similarities of interpreters are described in Section 5.2. In Section 5.3 I present measures of the compile- and running-times involved when compiling by specialization. Conclusions are given in Section 5.4.

5.1 Static Subset of Action Notation

Action notation, viewed as a programming language, has several dynamic features. By definition, a compiler cannot process dynamic features statically. Therefore, target programs must contain dynamic computations each time a dynamic feature is used in the source programs. (This observation holds for all compilers, not only action compilers). In action notation the following concepts are classified as dynamic

Control Flow. Actions may be abstracted into manipulatable objects, *abstractions*. These may later be enacted. Abstractions may be given as transients, bound to tokens, and stored in cells. In other words, abstractions are first-class objects. An action compiler is generally not able to determine which abstractions are enacted where.

Scope. A binding is produced when a token is bound to data. An abstraction may, when it is generated, incorporate the current bindings, thus producing a closure. This gives static scope of tokens. But it is also possible to incorporate the bindings when the abstraction is enacted. This gives dynamic scope. A compiler for a dynamically scoped language is generally not able to determine which tokens are bound where.

Furthermore, bindings may be reified into manipulable first-class objects, *maps*. A map may be reinstated as the current bindings. Thus, tokens may be bound by data-operations, which are not conventional binding operators.

Types. The sort of data cannot be determined statically. Transients, bindings, and abstractions do not have static types.

The interpreters (and hence the compilers derived from them) accept only a subset of action notation. This subset is essentially a *static subset* of action notation that prevents source programs from containing dynamic features. These kinds of restrictions makes it possible for a compiler (handwritten as well as automatically generated) to determine the value of dynamic computations at compile time. Alternatively, compilers for the unrestricted action notation must generate target programs that contain the dynamic computation.

Before I describe the static subset of action notation let us discuss the validity of this approach. How can we call the interpreters and compilers *action interpreters* and *action compilers* when they do not accept the full action notation? The answer is that we can't. The interpreters and compilers described in this section are unequal competitors to interpreters and compilers for the full action notation. The assumptions they make enable them to compile more efficiently and more effectively than general action compilers.

Though they do not implement the full action notation these compilers are interesting, exactly because they makes it possible to compare the restricted and the general subset. For example, the cost of the dynamic features in action notation can be measured.

The subset is restricted on the following features

Scope. The subset is statically scoped. The compiler knows at which points in the source action which variables are bound.

Types. The subset is statically typed. The compiler can detect if the source action is well typed. The abstract syntax of a source action is annotated with information that indicates whether actions are correctly typed. This does not imply that type errors cannot occur at run-time. When detecting a type error the compiler will generate code that corresponds to an error condition.

Tuples. The lengths of tuples of transients can be determined statically. Actually, no tuples have length more than three. Furthermore, the types of elements in tuples are known statically.

Combinators. Only the most important action combinators are implemented. Some of them may be used only in certain places: The dummy action unfold must occur in tail-position and the action combinator furthermore must occur as the first branch of one of hence or thence.

Bindings. Bindings may not be produced everywhere. For example, no bindings may be produced in the first subaction of the action then.

Data. Only numbers, booleans, (non-recursive) abstractions, and lists are supported. Lists may not contain abstractions.

The purposes of most of the restrictions are to remove the dynamic features. This subset has been adopted from the earlier work done by Anders Bondorf and Jens Palsberg in order to compare with their work. In particular it was interesting to measure the effect of their extensive binding-time improvements (discussed later).

5.2 Action Interpreters

Adopting the static subset of action notation makes it possible to adopt the interpreter also. For the syntax-directed specialization the interpreter from the article is used without modifications (except for two errors, which prevented the interpreter from running at all). For the type-directed specialization there are different requirements on the interpreter and a different way of representing runtime values and partially static structures.

Before the interpreter can be applied (an hence, before the source action can be compiled) the static properties of the source action must be made explicit. A type checker is applied to the source action. It checks whether the action is correctly typed or not and inserts this information into the action. This applies to both types, scope, and length of tuples of transients.

5.2.1 Syntax-Directed Partial Evaluation

The interpreters used in the experiments with the syntax-directed specializer are applied to the abstract syntax of an action, the current transients, the current bindings, a flag telling whether of the current nondeterministic branch has committed, and three continuations to apply should the action complete, escape, or fail, respectively.

The binding times of the arguments are as follows. The action is static. The current transients is a partially static list. The list has a predetermined length; the contents of the list is dynamic. The current bindings is also a partially static structure. It associates static tokens with dynamic values. This binding time allows environments to be processed at compile-time. A reference to a token in the source program goes via the environment to a (dynamic) value. Thus, there is no need for environments at run time. The commitment argument is dynamic; updates to the store, which triggers a commitment, are dynamic operations.

The escape-, and fail-continuations are dynamic and the complete-continuations is static. The following considerations have lead to this division. Actions resulting from straightline code in a program will be combined by action combinators like and-then, then, and before. Unless a subaction has an exceptional behaviour these actions are performed in sequence.

When one action completes, it transfers information to the next action. This action can and should be determined statically. These sequences of actions occur frequently and should be handled efficiently. The transfer of the current information from one action in the sequence to another is more efficient if it does not involve the generation and application of a complete-continuation. For the same reasons, keeping the complete-continuation static also results in smaller residual programs. The escape-, and fail-continuations are applied less frequently. Experiments made by Bondord and Palsberg showed that static escape-, and fail-continuations results in large target programs [5].

Binding-Time Improvements. Extensive *binding-time improvements* are required to keep the complete-continuation static. Handling introduction and application of abstractions causes problems. An abstraction is applied to (amongst other arguments) a complete-continuation that receives the value when the abstraction is applied later. This continuation is necessarily dynamic since the places where an abstraction is applied are unknown to the compiler. Therefore, when the interpreter is applied to the body of the abstraction the continuation must be coerced to a static value. Similarly, when an abstraction is applied it received (among other arguments) the current complete-continuation. The abstraction is dynamic so the binding-time analysis makes all arguments, including the continuation, dynamic. This causes all complete continuation to be considered dynamic an should be avoided. Both cases are handled by inserting an η-redex around the continuation. When introducing an abstractions, the continuation stays dynamic while the η-redex becomes static. Conversely when applying an abstraction. The continuation stays static while the η-redex becomes dynamic. These kind of binding-time improvements correspond exactly to the insertion of static and dynamic η-redeces done by the type-directed specializer. See Figure 4.7 in Chapter 4. In equation (1) static values are coerced to dynamic and in equation (2) dynamic values are coerced to static.

We note that one interpreter with and one interpreter without these improvements are *extensionally* equal: if f is a function then η-expanding f gives $\lambda x.\ f\ x$ that is extensionally equal to f: $(\lambda x.\ f\ x)\ a = f\ a$ for all a.

Binding-time improvements are also required to keep transients partially static when passed to static continuations and dynamic when passed to dynamic continuations. Actions are annotated with the length of tuples. This is static information and is used to coerce between partially static and dynamic transients. Again, these coercions corresponds type-directed specialization of tuples. The coercion is implemented by reconstructing the transient, directed by the (statically known) length. Again, these improvements do not change the extensional behaviour of the interpreter.

Finally, to prevent code duplication and loop unfolding when a static continuation is used twice (or indefinitely, is in a loop) the continuation it is coerced to a dynamic value. This is not a binding-time improvements but still it does not change the extensional behaviour of the interpreter.

To conclude, much work has been done to obtain good binding times for the various parts of the interpreter. However, the functionality of the interpreter is left unchanged by these improvements.

Representation of Transients and Environments. Transients are represented by partially static tuples of length 0, 1, 2, or 3. The length is hard-coded into the representation:

```
(defconstr (0dats)
           (1dats 1-1st-dats)
           (2dats 2-1st-dats 2-2nd-dats)
           (3dats 3-1st-dats 3-2nd-dats 3-3rd-dats))
```

Environments are represented by partially static lists of partially static pairs. The dynamic parts of these structures are the second elements in all pairs, the value to which a token (the first element of the pair) is bound. Representing partially static structures was discussed in Section 4.9.

All data are represented by Scheme values: Numbers are represented by Scheme numbers, booleans by Scheme booleans, lists by Scheme lists, and abstractions by lambda abstractions.

Specialization. The target programs are obtained by specializing the interpreter using the partial evaluator Similix. (Results are shown in Section 5.3). The final stage of Similix is a post-processing phase that performs constant folding. But even more constant folding (and hence more efficient target programs) are obtained by running the whole partial evaluator on the target program once more. However, this second specialization pass takes considerably more time than the first pass.

5.2.2 Type-Directed Partial Evaluation

The interpreter used for type-directed specialization is similar to the one described above except on a few points.

Binding Time Improvements. All binding time improvements applied to the interpreter above are essentially insertions of *identity functions*, which changes the binding time of parts of the interpreter. Such changes have no influence when switching to type-directed specialization (at least in its pure form. See below). Inserting η-redeces into a well-typed program neither changes the type nor the residual program obtained by specialization:

```
> (define (eta-expand f) (lambda (x) (f x)))
> (tdpe car '((a * a) -> a))
= (lambda (x0) (car x0))
> (tdpe (eta-expand car) '((a * a) -> a))
= (lambda (x0) (car x0))
>
```

However, in order to keep the residual programs small two techniques have been applied. All dynamic conditionals have been replaced by a function call by the technique described in

Section 4.9. This prevents code-duplication, which will otherwise result in exponentially sized programs (in the number of conditionals). Early experiments have shown that without this technique target programs would grow exponentially. The target programs of the examples presented later became tens of thousands of lines long (compare with Table 5.8 in Section 5.3).

The control flow of actions is represented by continuation exactly as in the interpreter specialized by syntax-directed partial evaluation. Thus, we must make similar efforts to prevent duplications of context. This is done by an online primitive operation

```
(define-primitive freeze-continuation (K -!> K)
  (lambda (abs) abs))
```

It is used whenever a continuation is passed to two instances of the interpreter. Instead of using a continuation `c` twice a new variable `cf` is bound to `(freeze-continuation c)`. This value is then passed twice. This works because the operation `freeze-continuation` is declared to have a side-effect (indicated by the `!` in the arrow). The specializer prevents side-effecting operations from being removed, duplicated, or rearranged by inserting a residual `let`-expression. In this case, the residual program will contain

```
(lambda (ev-act ... c)
  ...
  (let ([cf (freeze-continuation c)])
    (ev-act ... cf) ... (ev-act ... cf)))
```

As in the syntax-directed setting this is not a binding time improvement.

Representation of Transients and Environments. In Section 4.9 I described two techniques to represent partially static structures in the TDPE implementation. I use online probing primitive to implement partially static tuples of transients and partially static environments.

Specialization When using type-directed partial evaluation it does not make sense to run a second pass of specialization. This is because type-directed specialization gives a residual program that is extensional equal to the source program. And specializing two extensionally equal source programs gives the same residual program

```
> (tdpe car '((a * a) -> a))
(lambda (x0) (car x0))
> (tdpe (lambda (x0) (car x0)) '((a * a) -> a))
(lambda (x0) (car x0))
>
```

The use of online primitives makes constant folding appear on the fly. It might be possible to obtain more static reduction by using `case-syntax` to specify algebraic laws. But the improvements appear to be negligible.

Program	Source	Action	Input	Iterations
`bubble.hpl`	72	4 726	30	10
`bubble.ad`	74	5 895	30	10
`sieve.ad`	44	4 412		2
`euclid.ad`	40	2 686	77 64 50	1

Sizes of the source programs are the length of the programs in lines. Sizes of actions are the number of pairs in the Scheme representation of the (annotated) action.

Table 5.1: Sizes of intermediate representations

5.3 Performance Evaluation

I have conducted experiments that makes it possible to present a broader perspective on partial evaluation of action interpreters. There are four independent directions in these experiments. First, I wanted to compare syntax-directed and type-directed partial evaluation. The non-trivial action interpreters provide a good framework for this. Second, I wanted to measure the cost and gain from binding-time improvements. Third, I wanted to measure the gain of specializing an interpreter. And finally, I wanted to test the hypothesis suggested in Section 4.9 that compiling by using a compiler is more efficient than compiling by specializing an interpreter.

For their experiments, Anders Bondorf and Jens Palsberg used a version of the SCM Scheme system on a computer architecture not available at the time our measured were conducted. Therefore we have not been able to reproduce their numbers.

Programs were run on a Silicon Graphics Iris4d running Chez Scheme version 5.0a. The Similix system, version 5.0, was used in the syntax-directed experiments and the TDPE system was used in the type-directed experiments. All times are measured in ms (1/1000 of a second). The space usages of running the interpreter, the specializer, or the residual programs are measured in mega-bytes (Mbyte).

The experiments have been conducted using both the binding-time improved interpreter and the original, unimproved interpreter. In the tables, column named "+ Imprv" contains the numbers from using the binding-time improved interpreter and the colum names "÷ Imprv" contains the numbers from using the original, unimproved interpreter.

To compare the two different approaches to compilation, Similix was self-applied with respect to both the improved and the unimproved interpreters (i.e., the second Futamura projection). In the tables, columns have a "(PE)" or "(CO)" appended. They indicate whether programs were compiled by applying Similix ("(PE)") to the interpreter and an action or by applying a compiler ("(CO)") to the action. The times used to generate a compiler for the unimproved interpreter is 6 390 ms and for the improved one 6 530 ms.

As experimental material we use the same actions as those considered by Bondorf and Palsberg. They were originally used by Palsberg in connection with his provably correct

Program	Syntax-Directed		Type-
	÷ Imprv	+ Imprv	Directed
`bubble.hpl`	80 940	88 240	75 380
`bubble.ad`	120 050	130 550	117 420
`sieve.ad`	34 210	34 080	26 520
`euclid.ad`	6 240	6 940	6 140

Table 5.2: Interpretation times (ms)

Program	Syntax-Directed		Type-
	÷ Imprv	+ Imprv	Directed
`bubble.hpl`	297.0	307.4	298.1
`bubble.ad`	414.1	429.8	415.3
`sieve.ad`	103.3	107.0	103.1
`euclid.ad`	23.7	24.8	23.7

Table 5.3: Space usage during interpretation (Mbytes)

compiler generator Cantor [18, 19]. The actions are `bubble.hpl`, a bubble-sort program written in HypoPL; `bubble.ad`, the same bubble-sort program written in Mini-Ada, a subset of Ada; `sieve.ad`, Eratosthenes sieve, which finds prime numbers, written in Mini-Ada; and `euclid.ad`, Euclid's method, also written in Mini-Ada. The sizes of these programs are shown in Table 5.1 along with the sizes of the corresponding actions (in their Scheme representation).

The table also gives the input to each program as well as the number of times the program is executed. To give more reliable numbers programs are run several times. The input to the bubble sort programs are the number of integers to sort. There is no input to `sieve.ad`. The input to `euclid.ad` is the two numbers whose greatest common divisor should be computed and an iteration count.

Interpreting actions. To measure the effect of the compilation phase, I have measured the performance of running the interpreters on the annotated actions. Times are shown in Table 5.2 and the space usage is shown in Table 5.3.

The interpreters used in the syntax-directed settings and the interpreters used in the type-directed setting are almost equal. They differ in the way they represent runtime values and the binding-time improvements. I have measured all examples in the syntax-directed setting both using the binding-time improved interpreter and the original, unimproved interpreter.

Time. The binding-time improvements have a cost when interpreters are not specialized. This is about 10 percent of the time of running the original, unimproved interpreter.

Program	Syntax-Directed (PE)		Syntax-Directed (CO)		Type-Directed
	÷ Imprv	+ Imprv	÷ Imprv	+ Imprv	
`bubble.hpl`	2 700	4 050	3 710	4 540	110
`bubble.ad`	3 150	4 680	4 310	4 610	150
`sieve.ad`	2 450	3 500	3 720	4 010	110
`euclid.ad`	1 880	2 400	3 360	4 030	70

Table 5.4: Compile times (ms)

Program	Syntax-Directed (PE)		Syntax-Directed (CO)		Type-Directed
	÷ Imprv	+ Imprv	÷ Imprv	+ Imprv	
`bubble.hpl`	7.9	10.3	11.6	13.2	0.7
`bubble.ad`	8.2	12.1	11.9	14.1	0.8
`sieve.ad`	6.4	9.5	11.3	13.1	0.6
`euclid.ad`	4.4	5.9	10.3	11.6	0.4

Table 5.5: Space usage during compilation (Mbytes)

The interpreter from the type-directed setting is slightly faster than the interpreters from the syntax-directed setting.

Space. The different representations of values required by Similix and the TDPE system makes no difference at this stage but η-expansion done while running the binding-time improved interpreter has a cost. This is about 3 percent of the space used when running the unimproved version.

Compiling actions. In Section 4.9 it was suggested that specialization by a compiler (i.e., Similix specialized with respect to an interpreter) is more efficient than merely specializing an interpreter. In the former the interpretive overhead is removed. In order to test this hypothesis I have compiled actions by both means. All numbers are in Table 5.4 and Table 5.5. Obviously from the tables the hypothesis was wrong. Considering the improved versions, the compiler ("(CO)") is about 10 percent slower and uses about 25 percent more memory than by specialization of the interpreter. Considering the unimproved version the performance of the compiler even worse.

Specializing the original, unimproved interpreter is more efficient than specializing the binding-time improved one, no matter how compilation is achieved. The speedup is between 1.2 and 1.5. This is the cost of having extensive binding-time improvements in the interpreter. We are willing to pay for this slowdown if the target programs becomes faster (this is measured below).

Program	Syntax-Directed		Type-
	÷ Imprv	+ Imprv	Directed
`bubble.hpl`	1 180	550	590
`bubble.ad`	2 850	2 170	2 010
`sieve.ad`	550	290	350
`euclid.ad`	80	40	60

Table 5.6: Run times of residual programs (ms)

Program	Syntax-Directed		Type-
	÷ Imprv	+ Imprv	Directed
`bubble.hpl`	15.5	7.2	10.2
`bubble.ad`	22.1	13.2	19.1
`sieve.ad`	6.2	4.1	4.5
`euclid.ad`	1.1	0.7	1.2

Table 5.7: Space usage while running residual programs (Mbytes)

Specializing the interpreter using type-directed partial evaluation is more efficient than specializing the interpreter (binding-time improved or not) using syntax-directed partial evaluation. The speedup is as much as 20 to 25 compared to specializing the original, unimproved interpreter and 30 to 35 for the binding-time improved interpreter. The speedup is a result of the different means that syntax- and type-directed specializers use. Again, to justify these numbers the running times of the residual programs should be comparable. The running times are shown in Table 5.6.

Running target programs. In the syntax-directed setting, compiling actions by using a compiler (generated by self-application of Similix) or compiling actions by specializing the action interpreter yields the same residual programs. This follows from the second Futamura projection.

The running times from the type-directed setting and the syntax-directed setting with binding-time improvements are almost the same. The latter performs slightly better by about 7 to 10 percent. (The running times of the `euclid.ad` program are so small that I consider them to be unreliable.)

The residual programs of the syntax-directed settings without improvements are about 1.5 to 2 times slower than the residual programs from the syntax-directed setting with improvements. This is the gain caused by the binding-time improvements. If we consider only the syntax-directed setting we conclude that the gain (at run time) is about the same as the

Program	Syntax-Directed		Type-
	÷ Imprv	+ Imprv	Directed
`bubble.hpl`	4 562	2 801	2 853
`bubble.ad`	5 387	3 465	3 578
`sieve.ad`	4 091	2 664	2 619
`euclid.ad`	2 581	1 601	1 681

Table 5.8: Size of target programs (pairs)

cost (at compile time). If we compare the type-directed setting to the syntax-directed setting with binding-time improvements we conclude that the gain (at compile-time) of type-directed specialization greatly outweighs the cost (at run time). Furthermore, it should be noted that the interpreter in the type-directed setting has not been improved by hand to the same extent as the interpreter in the syntax-directed setting.

It is instructive to compare the space usage of interpreting an action (Table 5.3) with the space usage of compiling the action and running the residual program afterwards (Table 5.5 and Table 5.7). During interpretation about 10 times as much memory is needed as during compilation and while running the target program, together. This relationship is independent of the kind of specializer and of the use of binding-time improvements. This is a result of having a static complete-continuation. Only in the interpreter need the complete-continuations be represented explicitly.

Residual programs. Finally, in Table 5.8 I show the sizes of the residual programs. The target programs from the unimproved setting are bigger than those from the improved setting. The difference measures the gain (in size) of specializing a binding-time improved interpreter. There is almost no difference between the residual programs from the type-directed experiments and those from the syntax-directed experiments with improvements.

Second pass of specialization. I have not been able to apply a second pass of specialization to the target programs because of lack of memory[1]. Measures conducted by Bondorf and Palsberg show that the second pass performs constant folding that makes the final residual program twice as fast. However, the time for the second pass is about ten times larger than the first pass of specialization. The second pass is interesting as it is beyond the reach of the type-directed specializer. Thus, it would pin down the weakness of type-directed not being flexible with respect to specialization strategy.

[1]This problem will be further investigated after the delivery of the thesis.

5.4 Summary and Conclusion

We have repeated the experiments that Anders Bondorf and Jens Palsberg conducted a few years ago. They have specialized an action interpreter with extensive binding-time improvements using Similix, a syntax-directed partial evaluator. The results they obtained were remarkable: the compile time is reduced by one order of a magnitude compared to Palsberg's Cantor system and yet the run times of the compiled programs are as fast as the compiled programs of Cantor.

In the meantime, the technique of type-directed specialization appeared. Thus, it made sense to repeat Bondorf and Palsberg's experiments by using a type-directed specializer. Furthermore, the number of experiments they have conducted is rather limited. They only consider one binding-time improved interpreter

Besides repeating Bondorf and Palsberg's experiments, we have provided more material. Similix has been used both on a binding-time improved interpreter (Bondorf and Palsberg's) and on the unimproved one. This makes it possible to measure both the cost and gain of binding-time improvements. As for type-directed partial evaluation, we have applied it to the original interpreter, which has no binding-time improvements and only one annotation to prevent code duplication.

The type-directed specializer compiles programs faster than the syntax-directed specializer Similix, and yet the residual programs of the two approaches are comparable both in speed and size. So the facts that Similix has a polyvariant specialization and that Bondorf and Palsberg's interpreter contains binding-time improvements do not make a difference here.

Chapter 6

Specializing Data Structures

In this section I introduce a technique that can be used to remove the interpretive overhead of applying a function to elements of an abstract data type. Here, a data type is the disjoint union of a two sets (or more; the degenerated case of only one set is not interesting). Any function that applies to elements of a data type must check from which of the disjoint sets its arguments originates. This is the overhead that the technique removes. The technique is simple enough to be automated but this is not done in this report.

Throughout this section programs and examples are written in Gofer[1] to emphasise the simplicity and immediate usability of the technique. Gofer is a lazy, functional language with a convenient notation to describe and represent types, data types, and functions with data types as domain and codomain.

Section 6.1 presents a motivating example of the technique. Section 6.2 explains traditional data types and how they are understood mathematically. Section 6.3 repeats this explanation for the alternative representation of data types. This representation imposes some restrictions on the use of data types. Apart from these restriction the two representations are equal. This is proven in Section 6.4. Some classical λ-representations of booleans and integers are derived in Section 6.5. This section also contains two interesting examples, namely how to derive a statically typed type-directed specializer from a dynamically typed specializer and how the technique applies to an interpreter for a programming language. Section 6.6 concludes.

6.1 Motivation

Let us start with an example to motivate the technique, which I call *specializing an abstract data type*, and explain how it is applied. We consider a simple data type, binary trees of type α. A binary tree is either a leaf of type α or a node that contains two binary trees. In Gofer we can define binary trees by

```
data Tree a  =  Leaf a
             |  Node (Tree a) (Tree a)
```

[1]Few, if any, modification are required to run the programs in Haskell or Miranda

Gofer automatically generates one functions for each set in the disjoint union. These function, the *constructors*, inject into the corresponding set of the disjoint union. In this case we have functions that construct leaves and nodes of binary trees

```
Leaf          :: a -> Tree a
Node          :: Tree a -> Tree a -> Tree a
```

For example, we can make a binary tree over integers by applying these functions appropriately

```
t             :: Tree Int
t             =  Node (Node (Leaf 1) (Leaf 2)) (Leaf 3)
```

One common operation on binary trees is *folding*. This is an operation that recursively maps a binary tree into another type β. We parameterize it by functions that map leaves and nodes into β

```
type LeafFun a b         = a -> b
type NodeFun a b         = b -> b -> b

fold :: Tree a -> (LeafFun a b, NodeFun a b) -> b
fold (Leaf i)   (fl, fn) =  fl i
fold (Node s t) (fl, fn) =  fn (fold s (fl, fn)) (fold t (fl, fn))
```

The definition of the function contains two *clauses*, one for each kind of binary trees namely leafs and nodes. Gofer allows us to *match* the actual argument against patterns, which contain the symbols of the constructor functions.

Using this, we can compute the sum of the leaves in the binary tree `t` by evaluating

```
? fold t (id, (+))
6
```

As expected, the result is 6. During the computation the function `fold` has matched each leaf and node to determine which operation to apply. This requires testing whether the tree is a node or a leaf and in each case bind the variables to the appropriate components. Performing the tests and producing the bindings is an interpretive overhead, which is removed by specializing the data type.

Specializing the function `fold` with respect to the data type amounts to removing the intermediate representation of the data type. We declare a type that will replace the original data type. A tree in the new representation is a function of the same type as the codomain of `fold`

```
type Tree' a b    =  (LeafFun a b, NodeFun a b) -> b
```

The main idea is that instead of representing a binary tree `t` by a data type we represent it by a function `t'` that satisfies `fold t (fl, fn)` = `t' (fl, fn)`.

We define functions that correspond to the behaviour of `fold` on leaves and nodes respectively. Their types match the type of the original constructors but with the new representation `Tree'`

```
leaf              :: a -> Tree' a b
node              :: Tree' a b -> Tree' a b -> Tree' a b

leaf i   (fl, fn) =  fl i
node s t (fl, fn) =  fn (s (fl, fn)) (t (fl, fn))
```

These functions replace the original constructors.

We can construct specialized binary trees in this untraditional representation in the same way as with the traditional representation: by applying the constructor functions appropriately

```
t'                :: Tree' Int Int
t'                =  node (node (leaf 1) (leaf 2)) (leaf 3)
```

To compute to sum of the leaves in this representation we evaluate

```
? t' (id, (+))
6
```

We obtain the same result as with the traditional representation. However, we have not constructed the binary tree (at least not as a data type) and therefore there is no interpretive overhead of matching leaves and nodes to determine which operation to apply.

6.2 Traditional Data Types

An *inductively defined data type* D is a disjoint union of two sets, of which at most one refers to the data type itself. Such a structure is defined by the triple

$$(\mathrm{D}, \langle T_1, \ldots, T_k\rangle, \langle R_1, \ldots, R_l, \mathrm{D}_{l+1}, \ldots, \mathrm{D}_{l+m}\rangle)$$

where the T_i's and R_i's are types that does not refer to the data type D itself, i.e., either base-types or a type-constructor $C(\cdot)$ applied to a type that does not refer to D. All D_i are the symbol D; the indices are kept merely for counting. For example, binary trees of type α are defined by $(\mathrm{Tree}_\alpha, \langle\alpha\rangle, \langle\mathrm{Tree}_\alpha, \mathrm{Tree}_\alpha\rangle)$. There are no R_i's in this example.

The semantics of the data type D is a set, which is defined by the following. The operator $|\cdot|_n$ defines a sequence of set. Each set is defined in terms of sets with lower index. The first set in the sequence represents elements from the data type that does not refer to the data type itself

$$|\mathrm{D}|_0 = \{(1, \langle t_1, \ldots, t_k\rangle) \ | \ t_i \in |T_i|, \ i = 1, \ldots, k\}$$

Each subsequent set represents elements that may be constructed inductively with elements from sets with lower indices. For $n > 0$

$$|\mathrm{D}|_n = \{(2, \langle r_1, \ldots, r_l, d_{l+1}, \ldots, d_{l+m}\rangle) \mid r_i \in |R_i|, d_j \in |\mathrm{D}|_{n-1}\}$$

The 1 and the 2 are used to keep the two sets disjoint. The semantics of D is the set

$$|\mathrm{D}| = \{x \mid x \in |\mathrm{D}|_i \text{ for finite } i\}$$

For example, we have $(1, \langle a\rangle) \in |\mathrm{Tree}_\alpha|_0$ for each $a \in \alpha$. The example tree denoted by `t` above is represented by

$$(2, \langle(2, \langle(1, \langle 1\rangle), (1, \langle 2\rangle)\rangle), (1, \langle 3\rangle)\rangle) \in |\mathrm{Tree}_{\mathbf{N}}|$$

To the data type is associated two *injection functions*

$$\begin{array}{lcl}
\mathrm{in}_1 & :: & \langle |T_1|, \ldots, |T_k|\rangle \rightarrow |\mathrm{D}| \\
\mathrm{in}_1\ \langle t_1, \ldots, t_k\rangle & = & (1, \langle t_1, \ldots, t_k\rangle) \\
 & & \\
\mathrm{in}_2 & :: & \langle |R_1|, \ldots, |R_l|, |\mathrm{D}|, \ldots, |\mathrm{D}|\rangle \rightarrow |\mathrm{D}| \\
\mathrm{in}_2\ \langle r_1, \ldots, r_l, d_{l+1}, \ldots, d_{l+m}\rangle & = & (2, \langle r_1, \ldots, r_l, d_{l+1}, \ldots, d_{l+m}\rangle)
\end{array}$$

They are used to construct elements in $|\mathrm{D}|$. (In Gofer the injection functions are curried).

The technique applies only to situations where there is only one *use* of the data type D. That is, there must be only one function f where elements of type D are deconstructed. Furthermore, f must be compositionally defined on the data type. We require that f is defined by something similar to

$$\begin{array}{lcl}
f & :: & |\mathrm{D}| \rightarrow \mathrm{V} \\
f\ (1, \langle t_1, \ldots, t_k\rangle) & = & g_1\ \langle t_1, \ldots, t_k\rangle \\
f\ (2, \langle r_1, \ldots, r_l, d_{l+1}, \ldots, d_{l+m}\rangle) & = & g_2\ \langle r_1, \ldots, r_l, (f\ d_{l+1}), \ldots, (f\ d_{l+m})\rangle
\end{array}$$

where neither g_1 nor g_2 refers to f. The d's are elements from the data type and the r's are not. f may maps the elements of the data type into the alternative representation. In other words f must be a homomorphism. For example, the fold operation on binary trees has type

$$\mathrm{fold} :: \mathrm{Tree}_\alpha \rightarrow (\alpha \rightarrow \beta) \times (\beta \rightarrow \beta \rightarrow \beta) \rightarrow \beta$$

It has the shape of f. Thus $\mathrm{V} = (\alpha \rightarrow \beta) \times (\beta \rightarrow \beta \rightarrow \beta) \rightarrow \beta$. The functions g_1 and g_2 are given by

$$\begin{array}{lcl}
g_1\ \langle a\rangle\ \ \ \ \ \ (f_\mathrm{l}, f_\mathrm{n}) & = & f_\mathrm{l}\ a \\
g_2\ \langle d_1, d_2\rangle\ (f_\mathrm{l}, f_\mathrm{n}) & = & f_\mathrm{n}\ (d_1\ (f_\mathrm{l}, f_\mathrm{n}))\ (d_2\ (f_\mathrm{l}, f_\mathrm{n}))
\end{array}$$

Of course, the functions g_1 and g_2 need not be defined explicitly. They are used here only to restrict the shape of f. I will later present other useful data types and functions that obey these restrictions and also some (useful) data types and functions that do not.

6.3 Specialized Data Types

The specialized version of the data types D is represented by the set $\mathrm{D}' = \mathrm{V}$, the codomain of the function that deconstructs the original data type.

From the function f we obtain the specialized injection functions (whose types match the types of the original injection functions). They equal the functions g_1 and g_2.

$$\begin{array}{lcl}
\mathrm{in}_1' & :: & \langle |T_1|, \ldots, |T_k| \rangle \rightarrow \mathrm{D}' \\
\mathrm{in}_1\ \langle t_1, \ldots, t_k \rangle & = & g_1\ \langle t_1, \ldots, t_k \rangle \\
\\
\mathrm{in}_2' & :: & \langle |R_1|, \ldots, |R_l|, \mathrm{D}', \ldots, \mathrm{D}' \rangle \rightarrow \mathrm{D}' \\
\mathrm{in}_2\ \langle r_1, \ldots, r_l, d_{l+1}, \ldots, d_{l+m} \rangle & = & g_2\ \langle r_1, \ldots, r_l, d_{l+1}, \ldots, d_{l+m} \rangle
\end{array}$$

For all practical purposes (for efficiency, that is) the applications of g_1 and g_2 are expanded by hand as they may also be in the definition of f.

Furthermore we replace $f :: \mathrm{D} \rightarrow \mathrm{V}$ by the identity function $\mathrm{id} :: \mathrm{D}' \rightarrow \mathrm{V}$. Thus, we have removed the interpretive overhead originally imposed by f. All application of f can then be reduced.

Finally, all elements in D are embedded homomorphically into D′ by using the specialized injection functions:

$$\begin{array}{lcl}
\mathrm{H} & :: & \mathrm{D} \rightarrow \mathrm{D}' \\
\mathrm{H}\ (1, \langle t_1, \ldots, t_k \rangle) & = & \mathrm{in}_1'\ \langle t_1, \ldots, t_k \rangle \\
\mathrm{H}\ (2, \langle r_1, \ldots, r_l, d_{l+1}, \ldots, d_{l+m} \rangle) & = & \mathrm{in}_2'\ \langle r_1, \ldots, r_l, \mathrm{H}\ d_{l+1}, \ldots, \mathrm{H}\ d_{l+m} \rangle
\end{array}$$

6.4 Equivalence

The connection between the original representation of the data type and the specialized representation is expressed as follows

Lemma. $f\ x = \mathrm{H}\ x$ for any $x \in |\mathrm{D}|$.

Proof. The lemma is proven by structural induction on the two cases of x.

- If $x = (1, \langle t_1, \ldots, t_k \rangle)$ then

  $$\begin{array}{rcl}
  f\ x & = & f\ (1, \langle t_1, \ldots, t_k \rangle) \\
  & = & g_1\ \langle t_1, \ldots, t_k \rangle \\
  \\
  \mathrm{H}\ x & = & \mathrm{H}\ (1, \langle t_1, \ldots, t_k \rangle) \\
  & = & \mathrm{in}_1'\ \langle t_1, \ldots, t_k \rangle \\
  & = & g_1\ \langle t_1, \ldots, t_k \rangle
  \end{array}$$

  as required.

- If $x = (2, \langle r_1, \ldots, r_l, d_{l+1}, \ldots, d_{l+m} \rangle)$ then, by induction hypothesis, the lemma holds for each of the d_i's: $f\ d_i = \mathrm{H}\ d_i$ for each $i = l+1, \ldots, l+m$. Therefore

$$\begin{array}{rcll} f\ x & = & f\ (2, \langle r_1, \ldots, r_l, d_{l+1}, \ldots, d_{l+m}\rangle) & \\ & = & g_2\ (2, \langle r_1, \ldots, r_l, f\ d_{l+1}, \ldots, f\ d_{l+m}\rangle) & \\ & & & \\ \mathrm{H}\ x & = & \mathrm{H}\ (2, \langle r_1, \ldots, r_l, d_{l+1}, \ldots, d_{l+m}\rangle) & \\ & = & \mathrm{in}'_2\ \langle r_1, \ldots, r_l, \mathrm{H}\ d_{l+1}, \ldots, \mathrm{H}\ d_{l+m}\rangle) & \text{(by homomorphism)} \\ & = & \mathrm{in}'_2\ \langle r_1, \ldots, r_l, f\ d_{l+1}, \ldots, f\ d_{l+m}\rangle) & \text{(by hypothesis)} \\ & = & g_2\ \langle r_1, \ldots, r_l, f\ d_{l+1}, \ldots, f\ d_{l+m}\rangle) & \end{array}$$

as required.

The lemma says that the following two approaches are equivalent:

1. Construct an element x from D by using the original injection functions in_1 and in_2. Then apply the function f to this element to obtain result r.
2. Construct the element r from D′ by using the specialized injection functions in'_1 and in'_2.

Since the latter does not involve applications of f there is no interpretive overhead in using this method.

6.5 Examples

There are many situations in which it may be worth to use specialized data types. Some of these situations lead to already known representations. However, situations where the technique does not apply also exist. Let us examine such a situation first.

6.5.1 Counter Example

Consider binary trees again but now with a function `flip`.

```
flip (Leaf a)     = (Leaf a)
flip (Node s t)   = Node (flip t) (flip s)
```

We assume that this is the only function that deconstructs lists. To bring this function into the appropriate shape we define g_1 and g_2 explicitly

```
flip (Leaf a)     = g1 a
flip (Node s t)   = g2 (flip s) (flip t)


g1 a              = Leaf a
g2 s t            = Node t s
```

But here the functions g_1 and g_2 construct binary trees in the original representation of the data type. The specialization technique does not change g_1 or g_2 so there will still be applications of the original injection functions in g_1 and g_2. Therefore the technique does not apply in this case.

6.5.2 Church Booleans

Consider the data type for booleans and a conditional operator

```
data Bool     = True | False

if            :: Bool -> a -> a -> a
if True  x _ =  x
if False _ x =  x
```

We can specialize this data type by writing

```
type Bool' a =  a -> a -> a

true, false  :: Bool'

true  x _    =  x
false _ x    =  x
```

This is the standard λ-representation of booleans in the classical lambda-calculus [2, Chapter 6].

6.5.3 Church Numerals

We can define a data type of natural numbers as

```
data Nat = Zero
         | Succ Nat
```

and function `iter`

```
iter :: Nat -> (a -> a) -> a -> a
iter (Zero)    f x =  x
iter (Succ n)  f x =  f ((iter n) f x)
```

which iteratively applies the argument `f` n times to the argument `x`, where n is the value denoted by the natural number. Specializing this function with respect to the data type `Nat` gives

```
type Nat' = (a -> a) -> a -> a

zero :: Nat'
succ :: Nat' -> Nat'

zero    f x = x
succ n  f x = f (n f x)
```

This is one way of representing natural numbers in the classical lambda-calculus. The representation is called *Church numerals* [2, Page 136]. Another representation, called *the standard numeral system* does not appear to be derivable, for the same reasons as with the counter example presented previously.

6.5.4 Data Types

The operator `|`, which joins two types into their disjoint union (as a data type), may itself be declared as a data type.

```
data Union x y = Fst x | Snd y
```

The only reasonable operation on elements of this type is the `caseof` operator defined below. It takes an element of the data type and two functions and applies one of the function depending on the value of the element of the data type.

```
caseof :: Union x y -> ((x -> a), (y -> a)) -> a
caseof (Fst x) (f1, f2) = f1 x
caseof (Snd y) (f1, f2) = f2 y
```

This is a built-in operation of Gofer. It is applied automatically in all functions that do a pattern-match on the arguments, if the arguments are data types.

If we specialize the data type of disjoint unions with respect to the `caseof` operator we obtain

```
type Union' x y =  ((x -> a), (y -> a)) -> a

fst             :: x -> Union' x y
snd             :: y -> Union' x y

fst x (f1, f2)  =  f1 x
snd y (f1, f2)  =  f2 y
```

6.5.5 Type-Directed Specialization

Previously in this report I promised an implementation of a type-directed specializer in a statically typed language. With the technique introduced in this section, we can derive such an implementation from that given in Section 4.8.1. The syntax of residual programs are given by

```
type Name      =  [Char]
data Exp       =  Num  Int
               |  Var  Name
               |  Pair Exp Exp
               |  Fst  Exp
```

```
                |  Snd  Exp
                |  Lam  Name Exp
                |  App  Exp Exp
```

Types, as passed to reify and reflect, are defined by an abstract data structure. Unfortunately there are two functions that deconstruct this datatype, reify and reflect. We have already mentioned that reify (reflect) is applied to types that occurs positively (negatively) in the type of the source term. Therefore we can divide types into two cases, those that occur positively and those that occur negatively, and replace reify and reflect by just one function, residualize.

```
data Type       =  Pbase
                |  Nbase
                |  Type `Pcross` Type
                |  Type `Ncross` Type
                |  Type `Pto` Type
                |  Type `Nto` Type

res (Pbase)          v = v
res (t0 `Pcross` t1) v = Pair (res t0 (fst v)) (res t1 (snd v))
res (t0 `Pto` t1)    v = Lam x (res t1 (v (res t0 (Var x))))
                         where x = fresh "x"
res (Nbase)          e = e
res (t0 `Ncross` t1) e = (res t0 (Fst e), res t1 (Snd e))
res (t0 `Nto` t1)    e = \x -> res t1 (App e (res t0 x))
```

The second argument to `res` is a pair in the second clause, a function in the third clause and syntax in the last three clauses. Clearly, this program is not statically typable. (We have postponed the problem of finding a fresh identifier in the third clause). However, when we specialize with respect to the abstract data-structure `Type` we obtain one function for each clause. These functions are statically typable.

```
pBASE             v = v
(t0 `pCROSS` t1)  v = Pair (t0 (fst v)) (t1 (snd v))
(t0 `pTO` t1)     v = Lam x (t1 (v (t0 (Var x))))
                  where x = fresh "x"
nBASE             e = e
(t0 `nCROSS` t1)  e = (t0 (Fst e), t1 (Snd e))
(t0 `nTO` t1)     e = \x -> t1 (App e (t0 x))
```

We pass a string that represents the next unique identifier to generate when constructing a dynamic lambda in the second clause. These fresh names correspond to de Bruijn levels [2].

```
pBASE             i v = v
(t0 `pCROSS` t1)  i v = Pair (t0 i (fst v)) (t1 i (snd v))
```

```
(t0 `pTO` t1)     i v = Lam x (t1 i' (v (t0 i' (Var x))))
                        where x = "x" ++ (show i)
                              i' = i + 1
nBASE             i e = e
(t0 `nCROSS` t1) i e = (t0 i (Fst e), t1 i (Snd e))
(t0 `nTO` t1)     i e = \x -> t1 i (App e (t0 i x))

tdpe x t             = t 0 x
```

We can use it like the type-directed specializer implemented in Scheme except that (1) we must specify the “sign” of types (positive or negative occurrence) and (2) elements of ground types in the source term must be coerced to abstract syntax. For example, we can specialize a function that applies its first argument to its second.

```
? tdpe (\f x -> f x) ((pBASE `nTO` nBASE) `pTO` nBASE `pTO` pBASE)
Lam "x0" (Lam "x1" (App (Var "x0") (Var "x1")))
```

Here, the type of `f` occurs negatively and is annotated as such.

In the next example a value of ground type (integer) occurs in the source term. It must be coerced to an expression (type `Exp`) to yield a correctly typed term.

```
? tdpe (\x -> (x, (Num 98))) (nBASE `pTO` pBASE `pCROSS` pBASE)
Lam "x0" (Pair (Var "x0") (Num 98))
```

Neither of the two shortcomings exist if we translate this implementation of type-directed specialization into Scheme. (1) is removed by parsing an un-annotated type into an annotated representation where the sign changes in function domains. This is not possible to do in a static typed language because the type of the parser depends on the syntax of the type it parses. (2) is removed simply by realising that in Scheme constants may be viewed as syntax. Thus, what I have derived here is an implementation in Gofer that is equivalent to type-directed partial evaluation in Scheme.

6.5.6 An Interpreter

The abstract syntax of a programming language may be defined by a data type. In this example we consider the lexically scoped, untyped, call-by-value lambda calculus

```
type  Name          =  [Char]

data  Exp           =  Num Int
                    |  Var Name
                    |  Lam Name Exp
                    |  App Exp Exp
```

The interpreter must manage environment correctly. It may (1) reduce beta-redeces and substitute values in terms or (2) thread an environment which is used to generate closures. Both options are legal but the former requires that terms are constructed by the substitution mechanism. This imposes the same problems as the counter example presented previously. Thus, we define an interpreter that threads an environment explicitly. To ensure a call-by-value semantics the interpreter is written in continuation-passing style.

```
data  Res               =  Cns Int
                        |  Cls (Res -> K -> Res)
                        |  Err String

type  Env               =  Name -> Res
type  K                 =  Res -> Res

interp                  :: Exp -> Env -> K -> Res
interp  (Num i)   r k   =  k (Cns i)
interp  (Var x)   r k   =  k (r x)
interp  (Lam x e) r k   =  k (Cls (\a k'-> interp e (extend r x a) k'))
interp  (App t u) r k   =  interp t r (\f ->
                              interp u r (\a -> apply f a k))

extend                  :: Env -> Name -> Res -> Env
extend r x v y          =  if x == y then v else r y

env0                    :: Env
env0 x                  =  Err "Unbound variable"

apply                   :: Res -> Res -> K -> Res
apply  (Cls f) a k      =  f a k
apply  f a k            =  k (Err "Applying non-closure")
```

We can define terms of the lambda calculus by applying the constructors appropriately. We define the common known combinators

```
s = (Lam "f" (Lam "g" (Lam "x" (App (App (Var "f") (Var "x"))
                                    (App (Var "g") (Var "x"))))))
k = (Lam "x" (Lam "y" (Var "x")))
```

Using this, we can

```
? interp (App (App (App s k) k) (Num 42)) env0 id
Cns 42
```

Specializing the data type with respect to the function `interp` we obtain alternative constructors

```
num i   r k     =  k (Cns i)
var x   r k     =  k (r x)
lam x e r k     =  k (Cls (\a k'-> e (extend r x a) k'))
app t u r k     =  t r (\f -> u r (\a -> apply f a k))

s' = (lam "f" (lam "g" (lam "x" (app (app (var "f") (var "x"))
                                     (app (var "g") (var "x"))))))
k' = (lam "x" (lam "y" (var "x")))
```

And then we can

```
? (app (app (app s' k') k') (num 42)) env0 id
Cns 42
```

This computation is equivalent to interpreting the term, represented by an abstract syntax tree. But *there is no interpretive overhead* of testing which expressions are interpreted. Thus, I have implemented an interpreter that has no interpretive overhead. The term "interpreter" might be misleading but we shall adopt this term instead of the term "compiler" since the alternative representation does not perform static computations. (For example, in the example above, environments in are not removed from the alternative representation.)

The interpretive overhead can also be removed by using partial evaluation to specialize the original interpreter with respect to the source program. Partial evaluation is more powerful than the technique I have introduced. For example, partial evaluation may unfold constant expressions. The technique I have presented does not unfold constants since this amounts to change the represented value (as an abstract syntax tree).

6.6 Summary and Conclusion

Partial evaluation is a technique that removes the interpretive overhead from interpreters. This chapter has described another technique with the same effect. The technique gives an alternative representation for values of an abstract data type. We have presented a formalization of the use of this technique and proven the equivalence between the alternative representation and the traditional representation.

The alternative representation is applicable in several situations. First, to derive a statically typed type-directed partial evaluator from a dynamically typed one. This is achived by specializing the dynamically typed, type-directed specializer with respect to the abstract syntax of types. Second, to remove the interpretive overhead from an interpreter for a programming language. It is however limited because it is only possible to specialize one application of the data type and this application must be a homomorphism.

The next section containts an implementation of action notation that is derived by using the technique described here.

Chapter 7

Embedding Actions into Scheme

This section presents an alternative implementation of action notation to the one found in Chapter 5. It is realized by specializing an action interpreter with respect to the abstract syntax of actions. This technique, which was described in the previous section, does not unfold calls, in contrast to partial evaluation, and might therefore not be as efficient. However, features not available in a compiled system can be added. I present Seas[1], which is an embedding of a deterministic, sequential subset of action notation into Scheme. The embedding is realised by the injection functions obtained by specializing an action interpreter.

Section 7.1 presents the outline of the action interpreter. Its structure is similar to that of the interpreter specialized previously in Chapter 5 but it was never necessary to implement the interpreter. Instead the technique from the previous section was used "on the fly" to implement an interpreter without interpretive overhead. The implemented language is discussed and compared with other implementations in Section 7.2. Section 7.3 gives some details on the implementation. Examples are presented in Section 7.4 — including an implementation of a subset of Scheme in action notation. In Section 7.5 the performance of the system is compared to the automatically generated action compiler derived previously in this report.

7.1 The Type of the Interpreter

The performance of an action may terminate, either by completing, escaping, or failing, or it may diverge. In addition, we will consider an action that *unfolds* — without this being caught by a surrounding unfolding action — to terminate too. This characterization is natural when action interpreters are implemented: the interpreter must handle the each of the outcomes "complete", "escape", "fail", and "unfold" in the same manner. Why is it so? Each of the outcomes represent the passing of information from the current action to another action. Including actions that unfold: they pass all of the current information (transients, bindings) to the innermost inclosing unfolding action from where performance continues.

With these thoughts in mind, we note that

1. the performance of an action may use the current information: current transients, cur-

[1]A recursive acronym: Seas Embeds Actions into Scheme

rent bindings, and the store;

2. the performance of an action may complete, escape, fail, or unfold. When this happens information is transfered to the next action. Which action is the next depends on the behaviour of the current action. (We are considering the situation where the performance of action is sequential, not interleaved);

3. the information an action passes depends on the behaviour of the action: An action that completes gives transients and bindings. An action that escapes gives only transients. An action that fails gives no information. An action that unfolds gives both transients and bindings.

I will describe the outline of an interpreter accounting for these thoughts. From this interpreter I have derived functions that embeds action notation into Scheme. The technique I have applied is described in Chapter 6. The action interpreter is similar to the one used by Bondorf and Palsberg in their experiments on action compilation. It is described in Chapter 5. The interpreter presented in this section is more simple since it is not subject of compilation (by specialization). No attempts have been made to alter the results of binding-time analysis and no effort is made to ensure static features like flow of control, scope, and type.

The control flow in the interpreter is determined by the three items above. To reflect this

1. the interpreter threads the current information: current transients and current bindings. It is neither necessary nor efficient to thread the store since it may be represented by a global updatable vector of values;

2. the interpreter threads one *continuation* for each kind of performance: complete, escape, fail, and unfold;

3. the type of the continuations reflects the kind of information transfered to the following action.

Continuations provide an excellent way of representing control flow. They were also used by Bondorf and Palsberg in their experiments but their interpreter did not have an unfold continuation.

In the case of actions interpreters, continuation are particularly important since the control may be transfered to several actions. The interpreter has one type of continuation for each kind of behaviour of actions and one continuation for yielders:

$\mathrm{K_C}$	$=$	$\mathrm{Data} \times \mathrm{Map} \to ()$	Complete Continuation
$\mathrm{K_E}$	$=$	$\mathrm{Data} \to ()$	Escape Continuation
$\mathrm{K_F}$	$=$	$() \to ()$	Fail Continuation
$\mathrm{K_U}$	$=$	$\mathrm{Data} \times \mathrm{Map} \times \mathrm{K_C} \to ()$	Unfold Continuation
$\mathrm{K_G}$	$=$	$\mathrm{Data} \to ()$	Give Continuation

The codomains are (), which accounts for the fact that once applied to its arguments a continuation never returns.

Note that the unfold continuations are applied to complete continuations. The occurrence of the dummy action unfold (inside the body of an unfolding action) need not be in tail position. When control reaches the unfold action the entire body is performed again. After the performance of the body, control returns to the unfold action. This is represented by the complete continuation.

Bondorf and Palsberg's interpreter does not have an unfold continuation since it only allows the unfold action to occur in tail position. The three other continuation of their interpreter, the complete, escape, and fail continuations, are all applied to a commitment argument as well [5]. This is in order to detect whether the action in one branch of a non-deterministic choice has committed (as this would prevent the branch from failing). In the present implementation we obtain the same control by a global state, which mimics possible update to the store. Most actions do not modify or check the commitment argument so the approach taken here is likely to be more efficient.

The interpretation function for actions is denoted by I_A and has type

$$\mathrm{I_A} :: \mathrm{Action} \to \mathrm{Data} \times \mathrm{Map} \times \mathrm{K_C} \times \mathrm{K_E} \times \mathrm{K_F} \times \mathrm{K_U} \to ()$$

It is applied to the abstract syntax of an action, the current transients, the current bindings, and the four continuation. I will not present the interpreter here but the reader should be able to reconstruct most of the interpreter from the type of the continuations and the type of the interpreter. In fact, the standard reference of Action Semantics [14] indicates how such an interpreter can be implemented. It is suggested that the action A_1 then A_2 is represented in λ-notation by

$$\lambda \epsilon_1 \rho \kappa.\ A_1 \epsilon_1 \rho (\lambda \epsilon_2.\ A_2 \epsilon_2 \rho \kappa)$$

under a simplifying assumption of determinism. Lifting this assumptions we end up with

$$\lambda \epsilon \rho \kappa_e \kappa_f \kappa_u \kappa_c.\ A_1 \epsilon \rho \kappa_e \kappa_f \kappa_u (\lambda \epsilon_1 \rho_1.\ A_2 \epsilon_1 \rho \kappa_e \kappa_f \kappa_u (\lambda \epsilon_2 \rho_2.\ \kappa_c \epsilon_2 (\rho_1 \uplus \rho_2)))$$

where $\uplus$ denotes disjoint union. This is exactly what is reflected by the interpreter and moreover, the λ-term above is exactly the higher-order value that represents the embedded action A_1 then A_2.

There is also an interpretation function for yielders. It has type

$$\mathrm{I_Y} :: \mathrm{Yielder} \to \mathrm{Data} \times \mathrm{Map} \times \mathrm{K_G} \to ()$$

where $\mathrm{K_G}$ is the type of the give continuation. Yielders provides values in one step: they can neither fail, escape, nor diverge. Thus, they only have one continuation.

7.2 Implemented Language

The implementation imposes some restrictions on action notation.

- The implementation is deterministic. When performing an action A_0 or A_1 the action A_0 is always performed first. Most often, non-deterministic choice does most often have exhaustive guards (i.e., exactly one of the two branches will fail). Deterministic performance has no influence in these cases. Non-determinism matters when the specified language has non-deterministic features, however.

  This restriction may be lifted by an appropriate use of *random* choice.

- The implementation is sequential. When performing an action A_0 and A_1 the action A_0 is always performed first. Only when the performance of A_0 completes will the performance of the action A_1 take place.

- There is no support for multiple agents and communication between these.

- There is a limited set of sorts, a subset of those from action. These may be combined into new sorts by a union operation. Otherwise, no new sorts may be introduced.

- There is a limited set of data operations. Each of these are implemented both as a data operation and as a yielder. In action notation a data-operation may always be used as a yielder. However, the implementations given here require data operations to be distinguished from yielders because they have different type.

  New data operations can relatively easily be implemented by Scheme procedures. The corresponding yielders can be derived automatically from the data operations.

On the positive side, this implementation allows a larger subset of action notation than other implementations[2]

- The sorts of data are dynamic.

- The tuples of transients passed to and from actions may have any length and the transients may have any sort.

- The action unfold may occur anywhere, not only in tail position.

- Bindings may be produced in both branches of binary action combinators.

- Recursive bindings (i.e., self-referential objects) may be produced.

- Non-deterministic branches that have committed may not fail. The implementation detects if an action that has committed fails. It it happens, an error is produced.

7.3 Implementation

By specializing the interpreter with respect to the abstract syntax of actions (and yielders) I have derived one injection function for each kind of action (and yielder). Embedded actions and yielders are represented by higher-order function of types

[2]Both Cantor and the work presented in Chapter 5 considers a restricted language

```
(define complete
  (lambda (trn bnd Ke Kf Ku Kc)
    (Kc '() '())))

(define escape
  (lambda (trn bnd Ke Kf Ku Kc)
    (Ke trn)))

(define fail
  (lambda (trn bnd Ke Kf Ku Kc)
    (Kf)))

(define unfold
  (lambda (trn bnd Ke Kf Ku Kc)
    (Ku trn bnd Kc)))

(define (give Y0)
  (lambda (trn bnd Ke Kf Ku Kc)
    (Y0 trn bnd
        (lambda (t0)
          (if (nothing? t0)
              (Kf)
              (Kc t0 '()))))))
```

Figure 7.1: Representation of some primitive actions

$$\begin{array}{lcl} \text{Action} & = & \text{Data} \times \text{Map} \times \text{K}_\text{C} \times \text{K}_\text{E} \times \text{K}_\text{F} \times \text{K}_\text{U} \to () \\ \text{Yielder} & = & \text{Data} \times \text{Map} \times \text{K}_\text{G} \to () \end{array}$$

The implementations of some primitive actions are shown in Figure 7.1. Embedded action combinators combines objects of type Action and Yielder in a way similar to action notation. There is a homomorphic embedding from the action notation action to the embedded representation Action in Scheme as higher-order functions. For example, the binary action combinators are represented in Scheme by higher-order functions of type Action × Action → Action. A few examples are show in figure 7.2.

Note the similarity between the four primitive action `complete`, `escape`, `fail`, and `unfold`. Each of these call the corresponding continuation.

Data is represented by Scheme values. Integers and booleans are represented by Scheme integers and booleans. The special value nothing is represented by the string `"nothing"`. Tokens are represented by Scheme symbols (not strings). Tuples of transients are represented by lists of values except that one-tuples are represented by the only component in the tuple. Lists, cells, abstractions, and bindings are tagged values. A tagged value is a vector with a

```
(define (unfolding A0)
  (lambda (trn bnd Ke Kf Ku Kc)
    (fix (lambda (t0 b0 Ka Kc)
           (A0 t0 b0 Ke Kf Ka Kc))
         trn bnd Kc)))

(define (trap A0 A1)
  (lambda (trn bnd Ke Kf Ku Kc)
    (A0 trn bnd
        (lambda (t0)
          (A1 t0 bnd Ke Kf Ku Kc))
        Kf Ku Kc)))

(define (then A0 A1)
  (lambda (trn bnd Ke Kf Ku Kc)
    (A0 trn bnd Ke Kf Ku
        (lambda (t0 b0)
          (A1 t0 bnd Ke Kf Ku
              (lambda (t1 b1)
                (if (disjoint? b0 b1)
                    (Kc t1 (disjoint-union b0 b1))
                    (Kf)))))))))
```

Figure 7.2: Representation of some action combinators

tag (a string) as the first element and the value as the second.

Abstractions are a tagged actions. They are generated by the yielder

```
(define (abstraction-of A0)
  (lambda (trn bnd Kg)
    (Kg (tag "abstraction" A0))))
```

The definition of the yielders application _ to _ and closure _ follows closely from the description given by Mosses [14, Page 117]. An abstraction is enacted by

```
(define (enact Y0)
  (lambda (trn bnd Ke Kf Ku Kc)
    (Y0 trn bnd
        (lambda (t0)
          ((untag t0) '() '() Ke Kf Ku Kc)))))
```

7.4 Experiments

Here follows some simple examples of the use of the system. The most simple action is probably complete. It completes with no transients, the empty bindings and it makes no changes to the store. We can run this action by

```
> (load "seas.ss")
> (run complete)
the action completes with ()
  bindings   = none
  store      = empty
>
```

As expected, the performance of the action completes. The empty Scheme list `()` corresponds to the empty tuple $\langle\rangle$ so no transients are given. No bindings were produced and no changes were made to the store.

Consider the following action. It contains a binary action combinator then and a data operations sum used as a yielder

give 100 then give the sum of (the given integer#1, 98)

This is performed by

```
> (run (then (give (num 100))
             (give (sum (given# "integer" 1) (num 98)))))
the action completes with 198
  bindings   = none
  store      = empty
>
```

As in action notation the type of `give` is Yielder → Action. Therefore the integer value 100 must be coerced to a yielder. This is done by the procedure `num` :: Int → Yielder.

The Scheme procedure `sum` represents a yielder, not a data-operation. This is in contrast to action notation where `sum` is a data-operation, which may be used as a yielder. The corresponding data-operation as denoted by `D_sum`. It may be applied to Scheme integers:

```
> (D_sum 10 20)
30
>
```

All data-operations are *strict* with respect to the special value `"nothing"`. Thus

```
> (D_sum "nothing" 19)
"nothing"
>
```

7.4.1 Scheme in Action Semantics

As a more complete example we can implement Scheme on top of action notation by using the Action Semantic Description from Section 3.3. The semantic function evaluate is translated into Scheme. It simply expands a Scheme expression into an action, which is a higher-order function. The performance of this may then be obtained as above.

```
> (run (evaluate '42))
the action completes with 42
  bindings   = none
  store      = empty
> (run (evaluate '((lambda x x) 42)))
the action completes with 42
  bindings   = none
  store      =
    00000000: 42
    00000001: unreserved
>
```

The first example simply evaluates a constant. The second has an application of an abstraction. The formal parameter is bound to a cell, which contains the value of the actual argument. In this case the argument is stored in cell number 0.

The following example illustrates the static scope of Scheme. It creates an abstraction in one environment, binds it to the token f and applies it in another environment. The token x is bind to different data in the two environments. The following session shows that the implementation is static scoped.

```
> (run (evaluate '((lambda x
                     ((lambda f
                        ((lambda x
                           (f 2))
                         1))
                      (lambda dummy x)))
                   0)))
the action completes with 0
  bindings   = none
  store      =
    00000000: 0
    00000001: #(abstraction #<procedure>)
    00000002: 1
    00000003: 2
    00000004: unreserved
>
```

7.4.2 Features Beyond Action Notation

Seas provides some features beyond pure action notation. Similar features may be found in implementations of more traditional programming languages. And indeed, most of them are due to the underlying implementation in Scheme. I believe that these features make the Seas-system useful to newcomers to Action Semantics.

Connection with other tools. Seas may be used in connection with the ASD-SDF tool to avoid the abstract syntax of actions required by Scheme. No such attempts have been made at present, though. It requires that the syntax of the output from the ASD-SDF tool is changed to match the Scheme syntax.

Interactive performance. Actions are performed in an interactive session between the user and the system. No compilation is required, but there is no interpretive overhead involved in performing actions (the efficiency of the system is measured in Section 7.5). If the performance of an action terminates then the system shows the information given on termination.

```
> (run (then (give true) escape))
the action escapes with #t
  store      = empty
> (run (then (give true) fail))
the action fails because fail was forced
  store      = empty
> (run (then (give true) unfold))
the action unfolds with #t
  bindings   = none
  store      = empty
>
```

Error messages. When the performance of the action fails the system reports the reason for this behaviour.

```
> (run (then (give (num 10)) (give (given "truth-value"))))
the action fails because value to "give" was "nothing"
  store      = empty
>
```

Currently, when a yielder gives `"nothing"` no reason is passed to the action. If this were the case the quality of the "error messages" could be improved.

Tracing. During the implementation of the system I found it useful to be able to trace the performance of particular actions. This was a help both to debug the system and the actions

A new unary action combinator action

$$\texttt{watch} :: \text{Action} \rightarrow \text{Action}$$

has been introduced. The performance of `(watch A)` is the same as the performance of `A` but certain information is reported during the performance

```
> (run (then (give true) (watch (bind (token 'x) it))))
action [*] entered with #t
  bindings   = none
  store      = empty
action [*] completes with ()
  bindings   =
    [x = #t]
  store      = empty
the action completes with ()
  bindings   =
    [x = #t]
  store      = empty
```

The `[*]` represents the action being watched. If more than one action is traced it may be necessary to identify them (in order not to confuse the output). This is done by giving an extra argument to `watch`. A similar construct, `watchy`, exists for yielders.

The amount of information written during performance can be controlled by the user. In the following example I have modified the semantic description of Scheme such that it reports whenever a lambda-abstraction is applied.

```
> (set-output-flags #t #f #f #f)
> (run (evaluate '((lambda x
      ((lambda f
((lambda x
   (f 2))
 1))
       (lambda dummy x)))
   0)))
action [apply] entered with 0
action [apply] entered with #(abstraction #<procedure>)
action [apply] entered with 1
action [apply] entered with 2
action [apply] completes with 0
action [apply] completes with 0
action [apply] completes with 0
action [apply] completes with 0
the action completes with 0
>
```

Incremental Construction of Action. Action may be constructed gradually. An action may be constructed from actions already tested for their performance

```
> (define add10 (give (sum (given "integer") (num 10))))
> (run-with add10 (tuple 5))
the action completes with 15
  bindings   = none
  store      = empty
> (run-with (then add10 add10) (tuple 5))
the action completes with 25
  bindings   = none
  store      = empty
>
```

The Scheme procedure `run-with` is used instead of `run`. It supplies an initial tuple of transients to the action being performed.

7.5 Performance Evaluation

To measure the performance of Seas I have used the programs from Chapter 5. The measures are presented in Table 7.1. For comparison, this table also repeats the time to interpret and the time to run the compiled programs from the type-directed experiments. As might be expected, the compiled programs are more efficient than the embedded actions in Seas (the speedup is between 7 and 33). Seas does not restrict the action notation to a static subset so dynamic computations must be performed. Furthermore, in the compiled programs the complete continuation has been removed. The complete continuation is applied often, perhaps several times during the performance of a single action. I believe that passing values through the complete continuation has a cost which is part of the reason for the slower performance of Seas.

However, if we compare to the interpreter used in the type-directed setting we see that the non-presence of the interpretive overhead in Seas makes it between 2.2 and 7.7 times faster than the interpreter used in the type-directed setting in Chapter 5, which was the most efficient when it came to interpretation (See Table 5.2).

7.6 Summary and Conclusion

This chapter presents an implementation of action semantics that is a homomorphic embedding of actions into Scheme. It may be derived from an action interpreter (similar to those used in the partial evaluation experiments) by the technique of specializing data structures presented in the previous section.

The implementation accepts a larger subset of action notation than most existing implementations of action notation. Furthermore, the implementation has some features, which

Program	Seas	*Interpreted	*Compiled
`bubble.hpl`	13 210	75 380	590
`bubble.ad`	15 750	117 420	2 010
`sieve.ad`	11 810	26 520	350
`euclid.ad`	790	6 140	60

Times measured in ms (1/1000 of a second). Programs in this table were run on a Silicon Graphics Iris4d running Chez Scheme version 5.0a.

(∗) The numbers in the column name "Interpreted" are the time taken to interpret the programs using the interpreter from the type-directed experiments in Chapter 5. The numbers in the column named "Compiled" are the times to run the compiled programs of the same experiments.

Table 7.1: Run times in Seas

may be found in more traditional programming languages, but that are normally not connected with action notation.

It remains to compare the present implementation technique with that of partial evaluation. This amounts to implementing an embedding of the same static subset as considered in Chapter 5.

Chapter 8

Summary and Conclusion

This thesis contains an exploration of several interrelated aspects of higher-order programming languages. The first two chapters contain a bottom-up and a top-down approach to the understanding of programming languages. They are linked by partial evaluation that makes it simple to achieve semantics-directed compilation.

Compiling actions by partial evaluation. A few years back, Anders Bondorf and Jens Palsberg used syntax-directed partial evaluation to achieve semantics-directed compilation of actions into Scheme. Their work was significant in that they improved the compile time of action compilation by one order of a magnitude compared to Jens Palsberg's Cantor system. With the invention of type-directed specializers it was interesting to repeat their experiments. We have applied a type-directed specializer to essentially the same interpreter. This thesis presents an improvement of their compile times by a factor 30. The compiled programs are as efficient as those obtained by Bondorf and Palsberg which in turn are as efficient as the target programs of Cantor.

Embedding of actions into Scheme. This embedding is an alternative implementation to that achieved by compiling actions by specialization presented in Chapter 5. Whereas the interpreters used for specialization requires a static subset of action notation, the embedding provides an implementation of a larger subset of action notation. Furthermore, it has some pragmatic features, which are adopted from traditional programming languages, that are not found in other implementations of action notation. It will be interesting to implement an embedding of actions into a statically typed language like ML (for an implementation in a call-by-value language) or Gofer (for an implementation in a call-by-name language). The static types of these languages may result in more efficient implementations that that in Chapter 6.

Alternative representation of data types. The representation of data types as higher-order functions is described in Chapter 6 and emerged from formalizing the difference between the action interpreters in Chapter 5 and the embedding of actions in Chapter 7. It provides a representation of abstract data structures in which there is no overhead of deconstructing elements. In particular it removes the inherent interpretive overhead from an interpreter. This overhead is also removed by partial evaluation. Whether there is a formal connction between alternative representation of data types and partial evaluation remains to be explored. An interesting experiment would be to apply type-directed specialization to elements of the alternative representation. These are higher-order functions, and type-directed specialization would yield their normalized text.

Type-directed partial evaluation. The technique for deriving the alternative data type from the traditional one also suggest one way to obtain a statically typed implementation of type-directed partial evaluation.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[2] Henk Barendregt. *The Lambda Calculus — Its Syntax and Semantics*. North-Holland, 1984.

[3] Anders Bondorf. *Similix 5.0 Manual*, May 1993.

[4] Anders Bondorf and Jens Palsberg. Compiling actions by partial evaluation. In *FPCA'93*, pages 308–317, 1993.

[5] Anders Bondorf and Jens Palsberg. Compiling actions by partial evaluation. *Journal of Functional Programming*, 6(2):269–298, 1996.

[6] William Clinger and Jonathan Rees, editors. Revised^4 report on the algorithmic language Scheme. *LISP Pointers*, IV(3):1–55, July-September 1991.

[7] Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

[8] Olivier Danvy. A user's guide to type-directed partial evaluation in Scheme. Technical report, Computer Science Department, Aarhus University, October 1996. Version 0.3 – third draft of a BRICS lecture note (to appear).

[9] Kyung-Goo Doh. Action semantics: A tool for developing programming languages. Technical Report 93-1-005, The University of Aizu, 1993. Accepted for the proceedings of InfoScience'93, International Conference on Information Science and Technology, Seoul, Korea, Oct 21-22, 1993.

[10] R. Kent Dybvig. *Three Implementation Models for Scheme*. PhD thesis, University of North Carolina at Chapel Hill, April 87.

[11] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In Bernard Lang, editor, *Proceedings of the ACM SIGPLAN'90 Conference on Programming Languages Design and Implementation*, SIGPLAN Notices, Vol. 25, No 6, pages 66–77, White Plains, New York, June 1990. ACM Press.

[12] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International Series in Computer Science. Prentice-Hall, 1993.

[13] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming language.* Prentice Hall Software Series, 1988. Second edition.

[14] Peter D. Mosses. *Action Semantics*, volume 26 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1992.

[15] Peter D. Mosses and David A. Watt. Pascal action semantics, version 0.6. Available by FTP as ftp://ftp.brics.dk/pub/BRICS/Projects/AS/Papers/MossesWatt93DRAFT/-pas-0.6.ps.Z, March 1993.

[16] Flemming Nielson and Hanne Riis Nielson. *Two-Level Functional Languages*, volume 34 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1992.

[17] Jens Palsberg. An action semantics for inheritance. Master's thesis, Aarhus University, 1988.

[18] Jens Palsberg. An automatically generated and provably correct compiler for a subset of Ada. In *ICCL'92, Proc. Fourth IEEE Int. Conf. on Computer Languages, Oakland*, pages 117–126. IEEE, 1992.

[19] Jens Palsberg. A provably correct compiler generator. In *ESOP'92, Proc. European Symposium on Programming, Rennes*, volume 582 of *Lecture Notes in Computer Science*, pages 418–434. Springer-Verlag, 1992.

[20] Chris Reade. *Elements of Functional Programming.* Addison-Wesley, 1989.

[21] Jonathan Rees and Richard A. Kelsey. A tractable Scheme implementation. *SYMBOL*, 1994.

[22] David A. Watt. An action semantics of Standard ML. In *Proc. Third Workshop on Math. Foundations of Programming Language Semantics, New Orleans*, volume 298 of *Lecture Notes in Computer Science*, pages 572–598. Springer-Verlag, 1988.