A DYNAMIC COMPILER FOR SCHEME

A THESIS
SUBMITTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE
OF
MASTER OF SCIENCE IN COMPUTER SCIENCE
IN THE
UNIVERSITY OF CANTERBURY
by
Mark Alexander Hugh Emberson

University of Canterbury
1995

Abstract

Traditionally, dynamically-typed languages have been difficult to compile efficiently. This thesis explores *dynamic compilation*, a recently developed technique for compiling dynamically-typed languages. A *dynamic compiler* compiles and optimizes programs *as they execute*, using information collected from the running program to perform optimizations that are impossible to perform in a conventional batch compiler.

To explore these techniques we developed SKI, a dynamic compiler for Scheme. Tests on programs compiled by SKI, have shown that dynamic compilation techniques can give a substantial increase in the performance Scheme programs. In some cases they can increase performance by up to 400%.

Acknowledgements

I would like to thank my supervisor Bruce McKenzie for the help he has given me, and for his patience while I was writing this thesis. I would also like to thank my friends and colleagues, especially David Bainbridge and Nigel Waston for putting up with me for the past two years.

Finally, I would like to thank: my friends Mark Cox, Karen Ameye, Simon Lipscombe, Hana Rakena and Mark Dunlop for keeping me sane; my cat Damm Thing for being a cat; and my mother Mavis for housing and feeding me.

Contents

Chapter 1

Introduction

1.1 Introduction

Dynamically-typed languages can not usually be compiled as efficiently as statically-typed languages. This has prevented the widespread adoption of dynamically-typed languages such as Scheme, Lisp and Smalltalk.

There are three reasons why dynamically-typed languages are hard to compile efficiently:

1. There is very little type information available to the compiler.
2. They often have dynamic environments which permit the redefinition of existing procedures.
3. They have a high procedure call frequency which impedes optimization.

These reasons conspire to make it hard to compile even the simplest expressions.

Consider this Scheme expression

```
(define (add x y) (+ x y))
```

which creates a procedure called 'add' that adds two numbers together and returns the result. Or does it? Let us look at it from a compiler's point of view. For one thing, there is nothing to indicate that 'x' and 'y' will be bound to numbers. Furthermore, the expression '(+ x y)' calls the procedure bound to the variable '+', but at compile time we can not be sure that '+' will be bound to a procedure, let alone which procedure. Even if we assume that '+' is the standard addition procedure and that 'x' and 'y' are numbers, we still do not know what type of numbers they are—they could be integers, floating point numbers or complex numbers.

In contrast the equivalent C function is full of information:

```
int add(int a, int b) { return a + b; }
```

A quick glance tells us that 'a' and 'b' are numbers, in fact they are integers. We also know that '+' is a builtin operator, which in this context adds two integers and returns an integer.

The C version will execute in a few cycles. The Scheme version will take much longer. It has to check that '+' is a procedure and call it. Then, assuming that '+' is the standard addition procedure, it will have to check the types of its arguments and select the appropriate method for adding them together.

Traditionally, there have been two approaches to solving these problems: compiler options and type declarations.

Many Scheme compilers have options that allow the user to disable some of the dynamic features of the language. For instance, the user could specify that the standard procedures can not be changed. This would allow the compiler to inline the call to '+' in our example. Another option might specify that all variables that appear in numeric contexts contain small integers. Combining these two flags would allow the compiler to generate code for our example similar to that generated by the C compiler. Unfortunately, compiler options change the semantics of the language and in the case of the second option, compromise the safety of the language.

Common Lisp is dynamically-typed, but it also allows type declarations [Ste84]. Common Lisp programmers can add type declarations to the parts to the program that need to be fast and leave the types unspecified in the rest of the program. This hybrid approach is becoming increasingly popular and has been adopted by two recent object oriented languages Cecil [Cha93] and Dylan [Sha92].

An alternative approach to efficiently compiling dynamically-typed languages is to use a *dynamic compiler.* A *dynamic compiler* is compiler that compiles and optimizes programs *as they execute.* The compiler can collect information from the running program and use it to optimize program as it runs. For instance, the compiler can collect information about the types of the arguments that a procedure is commonly invoked with and create a special version of the procedure optimized for the common types. The compiler can also inline frequently executed procedures. If an inlined procedure is changed, then the compiler can undo the optimization.

The objective of this thesis is to apply the techniques of dynamic compilation to Scheme [CR91]. To this end, we have developed SKI, a dynamic compiler for Scheme. SKI conforms to the "Revised^4 Report on the Algorithmic Language Scheme"[1] (or R^{4}RS) which is the de-facto standard for Scheme.

1.2 Outline

The next chapter discusses previous work in the field of dynamic compilation. Chapter 3 gives an overview of the structure of SKI, which is discussed in detail in Chapters 4, 5 and 6. The performance of SKI is analyzed in Chapter 7 and Chapter 8 concludes this thesis.

1.3 Typographical Conventions

There are many small programs and code fragments embedded in the text of this thesis. To make them easier to read, they are are laid out according to a set of typographical rules:

Bold font is used for program keywords, e.g., **if**, **lambda**.

'Roman font' is used for all other lexical forms, e.g., a-variable, 1234. If necessary these are enclosed in single quotes to distinguish them from the surrounding text, e.g., the variable 'a' is ...

Italic font is used for meta-variables; variables which are used to hold unspecified parts of a program. E.g., (**if** *a b c*).

⟨*Angle brackets*⟩ are used to denote non-terminals in extended BNF style grammars. E.g., (**if** $\langle exp_1\rangle$ $\langle exp_2\rangle$ $\langle exp_3\rangle$).

[1]Revised^4 ≡ Revised Revised Revised Revised.

... are used to denote repetition and as an anonymous meta-variable. E.g., a_1, a_2, ... a_n and (**lambda** (x) ...).

[Square brackets] in Scheme code are equivalent to round brackets, selectively intermixing the round and square brackets makes Scheme code more readable. E.g., (**let** ([x 1]) (+ x 1)).

Chapter 2

Previous Work

In this chapter we discuss previous and current research that is related to the goal of this thesis. This research falls into three categories:

1. Research into using similar methods to achieve similar goals. E.g., using dynamic optimisation to increase the performance of dynamically typed languages. SELF, APL\3000 and Napier88 (Sections 2.1, 2.2 and 2.3) fall into this category.

2. Research into using different methods to achieve the same goals. E.g., using type inference to eliminate type checking, like Soft Scheme (Section 2.4).

3. Techniques that are somehow related, like BitBlt (Section 2.5).

2.1 SELF

SELF is a dynamically typed pure object oriented language under development by Stanford University and Sun Microsystems [US87, US91]. In SELF everything is an object and there are no classes. Each object encapsulates both its state and its behaviour. Unlike C++ [Str91], CLOS [Pae93] and other hybrid object oriented languages, there are no "simple" or "builtin" types. Even the simplest types are objects and the simplest operations are implemented as methods.

Figure 2.1 shows the objects involved in adding two Cartesian points $(3, 5)$

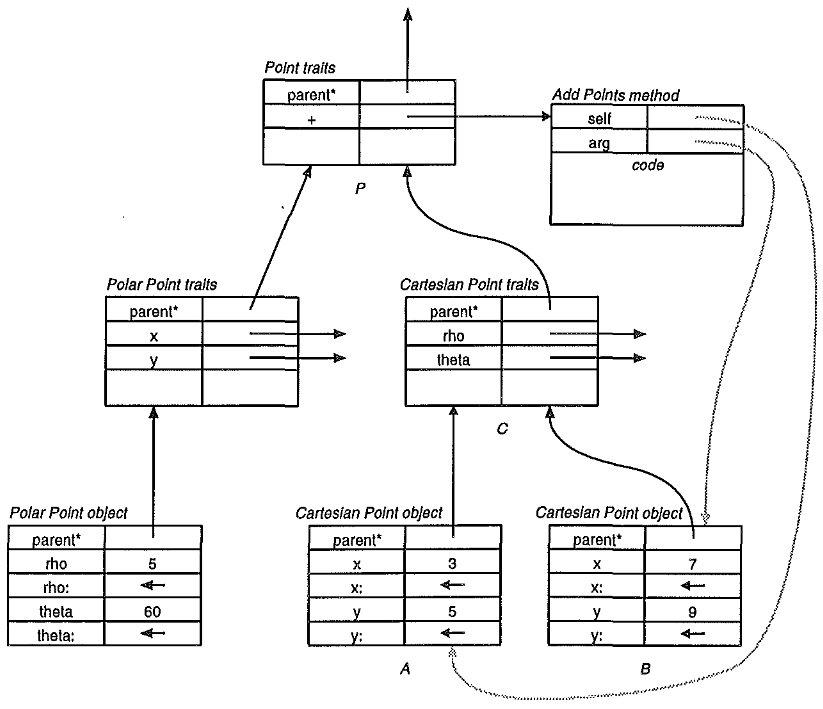


Figure 2.1: The objects involved in adding two Cartesian points in SELF.

and (7, 9) together in SELF.[1] Each object consists of a number of slots and each slot has a name and a value. Sending a message consisting of a slot's name to an object returns the value of that slot or, if the value is a method, the method is executed. E.g., sending 'x' to *A* would return 3. The slots with names ending in a colon hold special methods which set the values of the slots with the same base name. E.g., sending 'x: 2' to object *A* sets the value of *A*'s 'x' slot to 2.

The traits objects hold the attributes shared by all objects of a type and have the same function as class objects in Smalltalk. The Point traits object holds attributes shared by all points such as the Add Points method. The Cartesian Points traits holds all attributes shared by Cartesian Point objects such as the methods for simulating a polar point, 'rho' and 'theta'.

To add the two points *A* and *B*, the message '+' is sent to *A* with *B* as an argument. Since *A* has no slot called '+' it passes the message onto its 'parent*' slot[2] which references the point traits object *C*. *C* doesn't have a slot called '+' either, so it passes the message onto its parent *P*. *P* has a slot called '+' which contains a method, so the method is called with its argument slots,[3] 'self' and 'arg', bound to the message's original receiver, *A*, and argument, *B*, as shown by the gray lines.

The SELF code for the add points method is shown in Appendix A.1. The line which sets the *x* value of the new point is

$\underline{\text{newPoint x: } \underline{\underline{\text{x}} + \underline{\underline{\text{arg x}}}}}$.

This line is composed of a number of message send operations. Each message and its arguments is indicated by an underline. The first message, 'x', is short for 'self x'. It retrieves the *x*-coordinate from 'self', i.e., the integer object 3. The second message, retrieves the *x*-coordinate from 'arg', i.e. 7. The third sends the message '+' is sent to the result of 'x', 3, with the result of 'arg x', 7, as its argument, i.e., '3 + 7'. The final message sets the *x*-coordinate of the new point 'newPoint' to the result of the previous message, 10. So, adding two

[1]This is the "standard" SELF example which appears in many of the papers about SELF. This one is a composite of examples in [US91, CU89].

[2]An object may have one or more parent slots, they are distinguished from normal slots by a trailing asterisk.

[3]In SELF methods are a special kind of object and the method's slots are used for the same purpose as variables in other languages.

integers together and storing the result in an object takes at least four message send operations[4] and as we have seen each message send requires at least one lookup in the object. If SELF programs are to be fast then message sends will have to be fast.

The best way to make a message send fast is to perform the lookup part of the operation statically—at compile time. If this can be done, then slot access messages, like 'x', can be inlined and messages which invoke a method can be transformed into standard procedure calls or the method can be inlined. However, to perform the lookup at compile time, the compiler must know the type of the receiver of the message, but since SELF is dynamically typed, there is very little type information available to the compiler.

To provide type information for the compiler the SELF team have used a number of techniques [CUL89, CU89, CU90, CU91, HCU91, Cha92, Hö94] which are described in the following sections.

2.1.1 Customization

Customization creates multiple versions of methods each of which is *specialized* on a the type of a different receiver.

Specialization is a general technique which makes a special copy of a block of code in which the values of some parameters which were variable in the original are held constant. The parameters that the code is specialized on are generally the types or values of a set of variables. In some cases a runtime check is needed to ensure that the specialized version is executed only when the runtime values of the parameters match the values it was specialized on and if the check fails then the original version of the code is executed. In other cases, the check may be unnecessary or accomplished by other means.

When a method is customized, it is specialized on one particular type of receiver. Within the customized method the type of 'self' slot is known statically and the lookup operations for all message sends to 'self' can be done statically. Since a large percentage of all messages are sent to 'self' this can increase performance substantially.

[4]We say "at least four" because the '+' method for numbers is probably implemented using

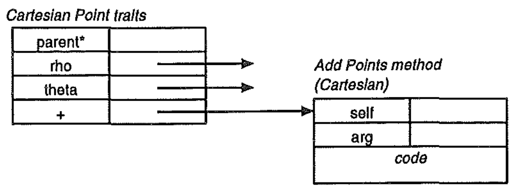


Figure 2.2: Customization

In the example in Figure 2.1 customized versions of the Add Points could be produced for both Cartesian and Polar points. The compiler can then ensure that specialized method is called only ever called for the correct type by adding a slot to that types' traits object as shown in Figure 2.2, to ensure that the method will only be called with the intended type as the receiver.

2.1.2 Type Prediction

Type prediction is used to predict the type of the receiver of a message based on the name of the message.

Early versions of the SELF compiler [CUL89, CU89, CU91] used static rules like "the receiver of '+' is a small integer 90% of the time". Later versions of the compiler [HCU91, H94] uses dynamic type information obtained from the running program to predict the types of the receivers.

When the compiler has predicted the type of the receiver of the message it "splits the message" and generates two copies of the message send, one for the expected case with the lookup done statically, the other for the general case which will perform the lookup at runtime. A runtime check also is generated which branches to the expected case if the prediction is correct and the general case otherwise. E.g., when the compiler encounters the expression

$i + 1$

it will predict that 'i' is an integer and generate the following (in psuedocode):

a technique called *double dispatching* which would require another message send.

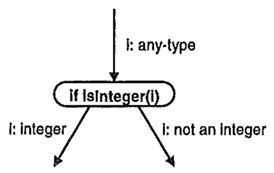
```
if isInteger(i) then
    i + 1               — Static lookup.
else
    i + 1               — Dynamic lookup.
endif.
```

In this case the statically bound message send could be inlined and reduced to a call to a primitive.

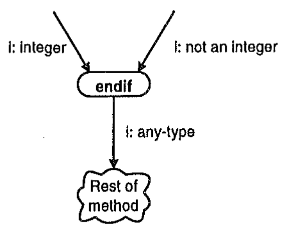
2.1.3 Type Analysis and Splitting

Type analysis [CU90, Cha92] is used to propagate type information though a method so that type checks can be eliminated and message sends can be statically bound or eliminated.

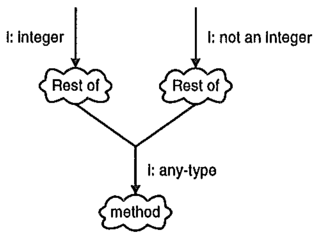
The sources of the type information are literal values, primitive operations, slots with known types and type tests. The type of a literal (e.g., a constant like '1') can be determined trivially and compiler also knows the types returned by all primitive operations. The types of some slots are also known , for instance in a customized method the type of 'self' is known.



(a) Type information created by a type check.



(b) Type information lost after a join.



(c) Splitting preserves type information.

Figure 2.3:

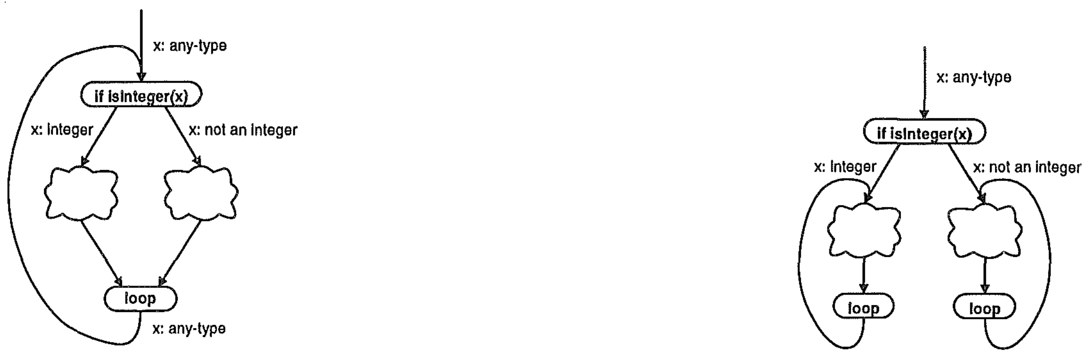
Type tests can also be a source of type information. Figure 2.3a[5] shows part of the *control flow graph* (or CFG, see Chapter 10 of [ASU86]) for the previous example. Before the type check the type of 'i' is unknown, in the true branch the

[5]Figures adapted from [CU90].

type of 'i' is known to be an integer and in the false branch it is anything but an integer.

However, joins in the CFG can destroy type information as shown in Figure 2.3b. In cases like this the compiler can eliminate the join and *split* the code following the join into two copies as shown in Figure 2.3c.

Early versions to the SELF compiler [CUL89, CU89] performed splitting only on the message send immediately following the join. Later versions [CU90] use extended splitting which can split many message sends if it is profitable, although the amount of splitting must be limited to keep the code size in check.



(a) Before. (b) After.

Figure 2.4: Loop splitting.

Type analysis and splitting can also be applied to loops [CU90] to split off loops in which the types of some of the slots are constant for the entire loop. Figure 2.4[6] shows an example of loop splitting.

2.1.4 Inlining

If the lookup part of a message send can be resolved statically and the message invokes a method, then the SELF compiler can choose to inline that method if it is small and not recursive.

[6]Figures adapted from [CU90].

Inlining a method doesn't just eliminate the call overhead, it can also provide opportunities for further optimization. In particular the types of the methods arguments can be propagated through the inlined body and the type of the return value of the method can be determined.

2.2 APL\3000

APL\3000 [Van77] is a dynamic incremental compiler/interpreter for APL which runs on Hewlett-Packard 3000 Series II minicomputers. APL is a dynamically typed language for mathematics with a rich set of data types and operators. Like Scheme, APL is also a dynamic in the sense that the meaning of names can be changed by the user and that the interpretation of an expression depends on the context. Figure 2.5[7] shows a three apparently identical expressions which have different meanings depending on the context in which they appear.

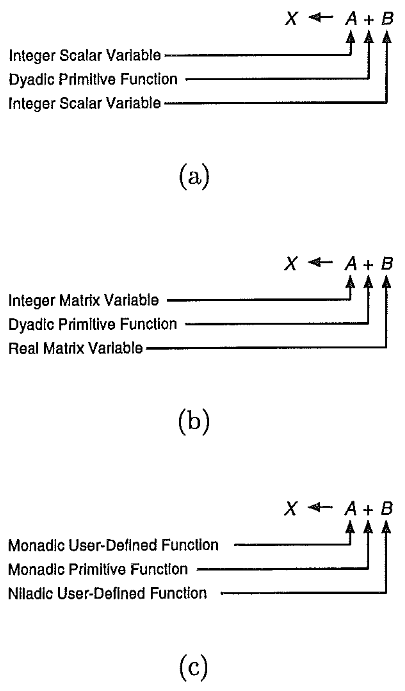

X ← A + B
Integer Scalar Variable
Dyadic Primitive Function
Integer Scalar Variable
(a)
X ← A + B
Integer Matrix Variable
Dyadic Primitive Function
Real Matrix Variable
(b)
X ← A + B
Monadic User-Defined Function
Monadic Primitive Function
Niladic User-Defined Function
(c)

Figure 2.5: The meaning of APL expressions depends on the context.

[7]Figure adapted from [Van77].

Clearly the dynamic properties of APL make it difficult to compile. The solution adopted in APL\3000 is to compile each expression the first time it is evaluated, assuming that the types of the variables and the bindings of the operators remain constant. For instance, the expression in Figure 2.5b would be compiled with the assumptions that A is a variable holding an integer matrix, B is a variable holding a real matrix and $+$ is the primitive addition function. The compiler also generates a prologue, or *signature*, for the compiled expression which checks that the assumptions are still valid. The first time the expression is executed the signature is skipped since the assumptions are guaranteed to be valid, but every subsequent time the signature is executed. If the signature fails the interpreter is signalled and the expression is recompiled.

There are two kinds of code that the compiler can generate. These are known as *hard* and *soft* code. *Hard* code assumes that the size of the values in an expression will remain constant. If the signature fails because the size of values changes, then *soft* code is generated which will operate on any size arguments, but which may be less efficient. For instance, if A and B are bound to 2×2 matrices the first time the expression is executed then the hard code generated will assume that they will always be 2×2 matrices. If the expression is executed with A and B bound to 4×4 then soft code will be generated.

2.3 Napier88

Napier88 [MBCD89] is a polymorphic language coupled with a persistent object store. Napier88 is statically typed, but has polymorphic procedures and data types, similar to generic packages in Ada [DOD83] and templates in C++ [Str91].

```
let sort = proc[ T ]( v : *T; lessThan : proc(T, T → bool))
begin
   ...
end
```

Figure 2.6: A polymorphic sort procedure

Figure 2.6a[8] shows the declaration of a polymorphic sort procedure which can sort a vector of any type. The type is specified by a *type variable* 'T'. The arguments to the procedure are 'v', a vector of objects of type 'T', and 'lessThan', a procedure which takes two arguments of type 'T', compares them and returns a boolean.

There are at least two strategies for compiling polymorphic procedures such as 'sort':

1. The compiler can generate a single polymorphic procedure which works on any type. This is the strategy used by ML [MTH89].

2. The compiler can generate multiple copies of the procedure, each specialized to a particular type. This strategy is used by Ada and C++.

Code generated using the first strategy requires less space, but runs more slowly. The second strategy produces faster code because the compiler has more information to work with. E.g., the compiler knows the size of the objects the procedure manipulates.

The strategy used in Napier88 is a mixture of the two. Initially, a single polymorphic procedure is compiled. This procedure is instrumented with profiling code to count the number of times it is invoked and the types that it is invoked with. When the procedure has been run a number of times the profiling data is examined and if the procedure is called frequently with some type then a specialised version of the procedure is generated for that type and linked into the system.

2.4 Soft Scheme

Soft Scheme [WC93] is a *soft type system* for Scheme. A type system infers and checks the types of variables and expressions in a program. Languages like ML [MTH89] and Haskell [HPJW$^+$92] have static type systems that can infer the types of most expressions with no need for type declarations. However, static type system can occasionally fail and reject a correctly typed program. If this happens

[8]Figures adapted from [CCKM94].

then the programmer must insert a type declaration or rewrite the portion of the program that caused the failure. The type system will also reject incorrectly typed programs.

In contrast, a *soft* type system *attempts* to infer the types of the variables and expressions in a program. Soft type systems represent the types of variables and expressions as sets of simple types. This makes them ideal for dynamically typed languages where variables and expressions can have a range of types. When the a soft type system fails, it does not reject the program but simply assumes that the expression that caused the failure can have any type and continues.

A soft type system can be used for two purposes:

1. An aid for understanding and debugging programs. The type system can report the types it discovers to the programmer so that he/she can check them against the model of the program in his/her head. It can also indicate the places where it failed so that the programmer can check them—if the type system failed then the program may be incorrect.

2. The type system can pass the type information to the compiler so that type checks can be eliminated.

Wright and Cartwright [WC93] modified the Chez Scheme compiler to make use of the type information generated by Soft Scheme. Programs compiled using the type system achieved speedups of up to 70% over programs with full type checking.

Unfortunately, there are a two of major problems with Soft Scheme and soft type systems:

1. They must operate over the entire program. Parts of the program cannot be checked in isolation. This can be impractical for large programs.

2. They are imprecise in the presence of assignment or mutation. Assignment to a variable, particularly a global variable, can result in very large sets of types. This imprecision is propagated to expressions that reference the variable (see [WC93]).

3. The type system depends on the types of the standard procedures. If the standard procedures are changed then the type system will fail.

These magnitude of these problems could be reduced if Scheme had a module system like ML. Modules could be checked in isolation and the effects of mutation might be limited to a module. Unfortunately standard Scheme doesn't yet have a module system.

2.5 Bitblt

Bitmap graphics kernels often make heavy use of the *bit block transfer*, or *bitblt* operation. Bitblt takes each bit from a source rectangle, *src*, and combines it with the corresponding bit in a destination rectangle, *dest*, using an operation $\triangle$.[9]

$$dest_{x,y} = src_{x,y} \triangle dest_{x,y}$$

Where x and y range over the rectangle, and $\triangle$ is a simple logical operator like *and*, *or*, *xor*, *nand*, ..., etc.

Bitblt is typically implemented using two nested loops, the outer loop iterates along the y-axis and the inner loop iterates along the x axis. If bitblt is to be efficient then the inner loop must operate on machine words rather individual bits, but since the source and destination rectangles can have arbitrary bit alignment and may overlap, the inner loop can become quite complex and slow.

One way to speed bitblt up is to specialize the inner loop for all combinations of operation, alignment and overlap. If there are 16 possible operations and 32 possible alignments, then neglecting overlap there are 512 possible combinations. Clearly specialising for all possible cases will be expensive in terms of code size.

Another solution [KEH91] is to generate specialized machine code for the inner loop at runtime. The runtime code generator needn't be very complex, essentially it just has to choose from a small number of templates and fill in the blanks. For instance, there might be a template for non-overlapping rectangles and several templates to handle overlapping rectangles. The blanks would be for the shifts to handle alignment and the instructions to perform the operation.

Simple runtime generated code can outperform a highly optimized static code by a factor of four [KEH91]. If the runtime code generator does some simple

[9]Some bitblts have two sources and two operators. $dest_{x,y} = (src1_{x,y} \triangle_1 src2_{x,y}) \triangle_2 dest_{x,y}$.

optimizations, for instance unrolling the inner loop, the generated code can be as much as ten times faster.

Chapter 3

Overview of the SKI Compiler

This chapter presents an overview of the structure of the SKI compiler as a prelude to the in-depth discussion of the compiler in the following chapters.

3.1 Structure of the Compiler

Figure 3.1 shows the basic structure and *modus operandi* of the compiler:

1. The *front end* (Chapter 4) takes standard Scheme expressions, transforms them into equivalent expressions in an intermediate language called SKI-CPS and then performs some simple optimizations on them.

2. The *back end* (Chapter 5) takes SKI-CPS expressions and transforms them into executable code.

3. The code is executed and information is gathered about the procedures that are executed, how frequently they are executed and what types they are executed with.

4. Based on the information collected, the compiler decides whether the expressions are worth further optimization.

5. Using the information and the SKI-CPS versions of the expressions, the compiler attempts to further optimize the code and produces new SKI-CPS expressions and passes them to the back end to be executed again (see Chapter 6).

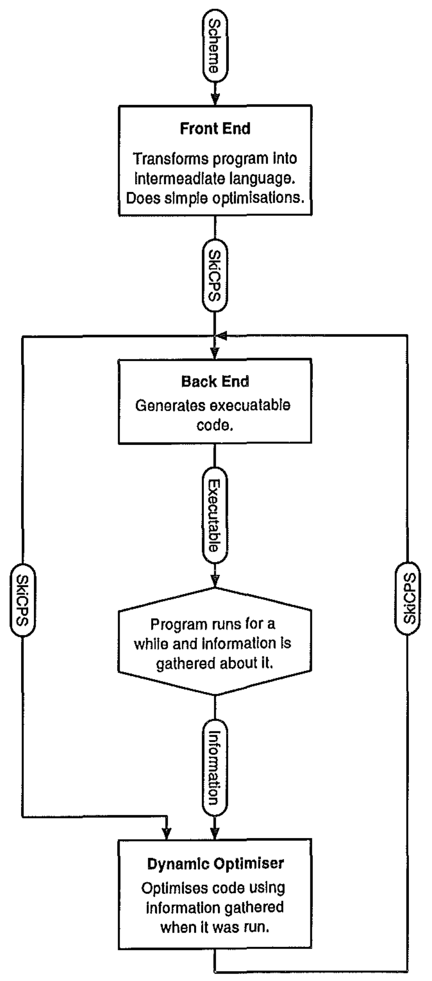

Scheme
Front End
Transforms program into intermeadiate language. Does simple optimisations.
SkiCPS
Back End
Generates execuatable code.
Executable
Program runs for a while and information is gathered about it.
Information
Dynamic Optimiser
Optimises code using information gathered when it was run.
SkiCPS
SkiCPS

Figure 3.1: Overview of SKI

6. This process is repeated until it is no longer profitable to optimize the expressions any further.

Chapter 4

The Front End

SKI's front end performs the initial transformation of Scheme programs into an intermediate language and some simple optimizations on this intermediate form.

The front end is structured as a sequence of passes. As shown in Figure 4.1, each pass takes the output of the previous pass and performs a simple transformation on it. The overall effect is that the program is transformed into successively simpler forms.

The front end can be further divided into two halves. The first half transforms programs into the CPS intermediate language. The second half performs simple optimizations on the CPS representation before passing it to the back end for code generation (Chapter 5). The output of the front end is also stored and, together with dynamically collected information, used as the input for the dynamic optimizations (Chapter 6).

This chapter is structured as follows: Sections 4.1, 4.2 and 4.3 cover the initial passes leading up to the transform into the intermediate language. Sections 4.4 and 4.5 discuss the CPS intermediate language and CPS transformation pass respectively. Sections 4.6, 4.7 and 4.9 cover the optimization passes. Finally, section 4.10 discusses the implementation of the front end and the internal representation of the intermediate language.

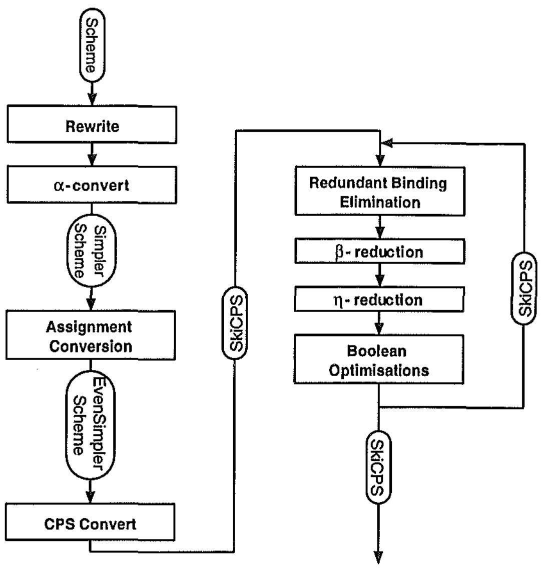


Figure 4.1: Overview of the front end. Passes are in **bold**.

4.1 Rewriting

Scheme expressions can be composed of a number of different types of expression. Some, such as **lambda**, **if** and variable reference, are known as *primitive expressions* because they are essential and it would be impossible to write programs without them. Other types of expression are known as *derived expressions* because they can be written in terms of primitive expressions.

SKI's rewriting pass transforms a Scheme program by rewriting most of the derived expressions as primitive expressions. This simplifies subsequent passes as they have fewer types of expression to deal with.

A good example of a derived expression is the **and** expression. The **and** expression has the syntax:

(**and** $\langle exp_1 \rangle$ $\langle exp_2 \rangle$ ...)

The $\langle exp \rangle$s are evaluated in order and if one of them evaluates to false then false is returned. If all the $\langle exp \rangle$s evaluate to true values then the value of the last $\langle exp \rangle$ is returned[1].

[1]In Scheme any value that is not false, '#f', is true, though in some older versions of Scheme the empty list, '()', is also considered to be false.

An **and** expression can be rewritten by (repeatedly) applying these three rules from Section 7.3 of the R^{4}RS [CR91]:

```
(and)            ≡  #t
(and ⟨exp⟩)      ≡  ⟨exp⟩
(and ⟨exp₁⟩ ⟨exp₂⟩ ...)
  ≡  (let ((x ⟨exp₁⟩)
           (thunk (lambda () (and ⟨exp₂⟩ ...))))
       (if x (thunk) x))
```

In the third rule, the value of ⟨*exp*$_1$⟩ is bound to the variable 'x' since it must be evaluated once only, but its value is used twice in the **if**. The rest of the ⟨*exp*⟩s are placed inside a **lambda** to delay the their evaluation until their values are required, and to prevent them from "capturing" the variable 'x'[2]. Creating and calling the procedure 'thunk' may look inefficient, but the procedure will be inlined by the β-reduction pass (Section 4.7). The following is an example of applying these rules to an expression:

```
   (and (null? left) (null? right))
⇒ (let ((x (null? left))
         (thunk (lambda () (and (null? right)))))
     (if x (thunk) x))
⇒ (let ((x (null? left))
         (thunk (lambda () (null? right))))
     (if x (thunk) x))
```

It is possible to rewrite any Scheme expression in terms of the six primitive expression types: literal, variable reference, procedure call, **lambda**, **if** and **set!**. We choose not to rewrite **letrec** and **let** expressions in this pass, leaving them for subsequent passes to rewrite, but we do rewrite **let*** and **let** loop. **Letrecs** will be rewritten in the assignment conversion pass (section 4.3) and **lets** in the cps-conversion pass (Section 4.4). This pass also recognizes and emits **prim** expressions which are not defined by the R^{4}RS [CR91]. These are used used extensively in the standard library to directly call SKI's primitives.

[2]The initialization expressions of a **let** are evaluated in the scope enclosing the **let** and can't "see" variables bound by the **let**.

The rewriting pass also rewrites quasiquote[3] expressions into expressions using builtin procedures like 'cons' and 'append'. The procedures that do this were borrowed from the Scheme->C compiler [Bar89].

4.2 α-conversion

The α-conversion pass renames all variables so that they have unique names. These are generated by concatenating the original variable name, an underscore and an unique number. Counters are maintained for each variable name encountered, so the first occurrence of 'x' becomes 'x_0', the second 'x_1', and so on.

So that variable names can be used as C identifiers in generated code, all non-alphanumeric characters in identifiers, such as '+', '?' and '!', are replaced by underscore delimited names. For example 'eq?' becomes 'eq_hook_0'. Minus characters, '–', are replaced by underscores except when they are at the beginning or end of an identifier. So 'frobnicate-thing' becomes 'frobnicate_thing_0', but '–' becomes 'minus_0'. These transformations try to maintain the readability of the Scheme identifiers while converting them into legal C identifiers.

Global variables are also detected in this pass and assigned locations in the global table. These are discussed further in the next section.

4.3 Assignment conversion

The assignment conversion pass removes all variable assignment expressions. This is achieved by detecting all the variables that are assigned to and replacing them with memory cells to hold their values.

More formally: for each local variable v which is assigned in a **set!** expression, we introduce a variable v' which has the same scope as v and which is bound to a cell that contains the initial value of v. We then replace all references to v with expressions that return the value of cell v' and replace all assignments (**set!** v x) with expressions that set the value of cell v' to x.

[3]Quasiquote is a way of generating almost constant data structures. For example the quasiquote expression '(1 a ,x) evaluates to (1 a 3) when 'x' is bound to 3.

For variables bound by a **let** we can substitute a cell for the **set!**'d variable:

```
(let ((v ⟨init⟩))          ⇒       (let ((v′ (prim $make-cell ⟨init⟩)))
  ... v ...                           ... (prim $get-cell v′) ...
  (set! v ⟨exp⟩)                      (prim $set-cell! v′ ⟨exp⟩)
  ...)                                ...),
```

but for variables bound by a **lambda** we introduce a new cell variable:

```
(lambda (v)                ⇒        (lambda (v)
  ... v ...                            (let ((v′ (prim $make-cell v)))
  (set! v ⟨exp⟩)                         ... (prim $get-cell v′) ...
  ...)                                   (prim $set-cell! v′ ⟨exp⟩)
                                         ...)).
```

The expression (**prim** \$make-cell x) returns a new cell initialized to x, (**prim** \$get-cell y) returns the current value of cell y and (**prim** \$set-cell! y w) stores w in cell y.

As we mentioned in Section 4.1, the assignment conversion pass also rewrites **letrec** expressions. The reason for this is that **letrec**s are rewritten using a special kind of assignment and the assignment conversion pass has all the information about which variables are assigned. **Letrec**s can be rewritten as follows:

```
(letrec ((v₁ ⟨init₁⟩)          ⇒        (let ((v₁ ⟨undefined⟩)
         ...)                                  ...)
  ⟨body⟩)                                 (set! v₁ ⟨init₁⟩)
                                          ...
                                          ⟨body⟩).
```

However if we use these rules and then use the standard assignment conversion discussed above, the simple data and control flow analysis in the Known call optimization pass (Section 6.6.1) will be impeded because it will have to try to trace values through cells.

Rather than this we introduce the notion of *once-cells*. Once-cells are similar to normal cells except that they are assigned once and only once. The conversion becomes:

$$
\begin{array}{lcl}
(\textbf{letrec}\ ((v_1\ \langle init_1\rangle) & \Rightarrow & (\textbf{let}\ ((v_1'\ (\textbf{prim}\ \text{\$make-once-cell})) \\
\qquad\qquad\ldots) & & \qquad\qquad\ldots) \\
\ \ldots & & \ (\textbf{prim}\ \text{\$set-once-cell!}\ v_1'\ \langle init_1\rangle) \\
\ v_1 & & \ \ldots \\
\ \ldots) & & \ (\textbf{prim}\ \text{\$get-once-cell}\ v_1') \\
 & & \ \ldots)
\end{array}
$$

unless any of the v_i are **set!**'d, in which case the **set!**'d variables are replaced with normal cells. E.g.,

$$
\begin{array}{lcl}
(\textbf{letrec}\ ((a\ \langle init_a\rangle) & \Rightarrow & (\textbf{let}\ ((a'\ (\textbf{prim}\ \text{\$make-cell-undef})) \\
\qquad\qquad (b\ \langle init_b\rangle)) & & \qquad\quad (b'\ (\textbf{prim}\ \text{\$make-once-cell}))) \\
\ \ldots & & \ (\textbf{prim}\ \text{\$set-cell!}\ a'\ \langle init_a\rangle) \\
\ (\textbf{set!}\ a\ \langle exp\rangle) & & \ (\textbf{prim}\ \text{\$set-once-cell!}\ b'\ \langle init_b\rangle) \\
\ \ldots) & & \ \ldots \\
 & & \ (\textbf{prim}\ \text{\$set-cell!}\ a'\ \langle exp\rangle) \\
 & & \ \ldots).
\end{array}
$$

The primitive '$make-cell-undef' makes a cell initialized to a undefined value.

This transformation makes analysis much easier since once-cells will only ever have one value after they have been set—essentially the initialization of the variable is separated from it's declaration.

Global variables are assumed to be **set!**'d, but rather than store them in separate cells we store them in a mutable table called the global table. All references to globals are replaced with calls to the primitive '$get-global' and all assignments to globals, including **define**s, are replaced with a call to '$set-global!':

$$
\begin{array}{lcl}
(\textbf{define}\ \text{x}\ 3) & \Rightarrow & (\textbf{prim}\ \text{\$set-global!}\ n\ 3) \\
 & & \\
(\ldots\ \text{x}\ \ldots) & & (\ldots\ (\textbf{prim}\ \text{\$get-global}\ n)\ \ldots)
\end{array}
$$

where n is the offset of 'x' in the global table. This is the offset allocated by the α-conversion pass.

After assignment conversion the values of variables don't change all mutation happens in memory. This simplifies the following passes because we don't have to keep track of changing variables and we are free to substitute values for variables or equivalent variables for one another. The real advantage of assignment conversion only becomes apparent when we introduce closures, see Section 5.1.

Assignment conversion was introduced by Kranz *et al.* in the Orbit Scheme compiler [KKR+86]. Scheme->C [Bar89] also uses a similar idea.

The SCHEMEXEROX [ACS93] compiler also performs assignment conversion and treats cells which are assigned only once specially. It attempts to collect these cells and transform the assignments into a **letrec**!

4.4 The Intermediate Language

Many compilers transform programs into an *intermediate representation* which is more amenable to analysis and optimization than the source language. This intermediate representation is often considered to be a language in itself and is often referred to as an intermediate language.

Intermediate languages may also be independent of source language and the target architecture making it easy to reuse parts of compilers for new languages and architectures. The Amsterdam Compiler Kit and GNU CC[4] [Sta92] are good examples, supporting several source languages and many target architectures.

There are many different kinds of intermediate languages. A few examples are:

RTL or *Register Transfer Language* is the language used by GNU CC [Sta92] and a number of other GNU compilers. RTL represents a program as a sequence of simple instructions for an abstract register machine. RTL is a kind of *three address code* (see [ASU86]).

SSA or *Static Single Assignment form*. SSA is a restricted form of a three address code, in which each variable or register is assigned only once and never changes.

EM is the intermediate language used by the Amsterdam Compiler Kit. It is a language for a simple abstract stack machine.

STG or *Spineless Tagless G-machine*. STG is a language for an abstract graph reduction machine. STG is used for compiling non-strict functional languages such as Haskell [Jon92].

[4]GNU CC was originally a C compiler, but it now has "front-ends" for Ada 9X, Pascal, C++, Fortran77 and Objective-C, with further languages planned.

CPS or *Continuation Passing Style*. CPS is a style of writing programs in which flow of control and data is represented by *continuations*. Continuations are functions which represent the rest of the program and which take a single argument, the result of executing the current expression. Continuation passing style intermediate languages are used in the Rabbit [Ste78] and Orbit [KKR+86] Scheme compilers and the SML/NJ [App92] SML compiler.

SKI uses CPS. Like SSA and three address codes, CPS isn't a single language, but rather a style of language. Before we introduce SKI's version of CPS, we will demonstrate the ideas behind CPS with an example.

4.4.1 CPS example

The Scheme procedure:[5]

```
(define f
  (lambda (x)
    (+ (* x x) (* 2 x) 1)))
```

can be expressed in continuation passing style Scheme as:[6]

```
(define f
  (lambda (x k)
    (* x x [lambda (u)
              (* 2 x [lambda (v)
                        (+ u v 1 k)])])))
```

Each function ('f', '+', '*') has been given a new parameter, which holds its continuation, and a continuation is supplied to each function call. Specifically, 'k' is the continuation of 'f', and the continuation of the first call to '*' is [**lambda** (u) ...]. Each continuation takes exactly one argument, which is the result of the function which calls it. So 'v' in the continuation [**lambda** (v) (+ u v 1 k)] will be bound to $2 \cdot x$ and the result of 'f' is passed to 'k'.

If we step through the example, expression by expression, we can see exactly what happens at each point.

[5]Note that in Scheme '+' and '*' are procedures of zero or more arguments.
[6]Note that this is still a legal Scheme program.

(**lambda** (x k) . . .)	Entering the function, 'x' is bound to some number, 'k' is the continuation of the whole function.
(∗ x x . . .)	The function '∗' is called, with arguments 'x' and 'x' and a continuation. '∗' multiplies its arguments and calls its continuation with the result.
[**lambda** (u) . . .]	The result of ∗ is bound to 'u'.
(∗ 2 x . . .)	∗ is called again and passes its result to
[**lambda** (v) . . .]	which binds the result to 'v'.
(+ u v 1 k)	'+' is called with arguments 'u', 'v' and 1. It calls 'k', with the result of the 'f'.

It should now be clear why this is called continuation *passing* style. For each procedure call, a continuation is passed to the procedure, which calls the continuation with its result. This is equivalent to the procedure call/return mechanism in imperative languages, but the "return" mechanism, calling the continuation, is made explicit. Kranz *et al.* [KKR+86] say that:

> . . . procedures do not 'return,' but rather 'continue into' the code represented by the continuation.

Two of the advantages of CPS as a intermediate language are :

1. The flow of control is absolutely explicit. At any point, the current continuation represents the rest of the programs execution—it is easy see what happens next. For example, the order of evaluation of function arguments is explicit, and as we will see later, all transfers of control are represented by continuations. All procedure calls in CPS are *tail calls*, that is, they never need to save information on a stack to use when they return, this makes tail recursion optimization trivial.

2. The flow of data is absolutely explicit. Data is propagated by the well understood, and simple, mechanisms of lexical scoping and procedure calls. The closure conversion transformation (Section 5.1) makes this even more explicit by eliminating the lexical scoping.

$$
\begin{array}{lll}
\langle \mathit{exp} \rangle & \rightarrow & (\textbf{jmp}\ \langle \mathit{carg} \rangle\ \langle \mathit{cont} \rangle) \\
& \mid & (\textbf{app}\ \langle \mathit{var} \rangle\ (\langle \mathit{arg} \rangle\ \ldots)\ \langle \mathit{cont} \rangle) \\
& \mid & (\textbf{cif}\ \langle \mathit{arg} \rangle\ \langle \mathit{consequent\text{-}cont} \rangle\ \langle \mathit{alternate\text{-}cont} \rangle\ \langle \mathit{cont} \rangle) \\
\\
\langle \mathit{carg} \rangle & \rightarrow & \langle \mathit{arg} \rangle \\
& \mid & (\textbf{nlambda}\ \langle \mathit{kvar} \rangle\ (\langle \mathit{var} \rangle\ \ldots)\ \langle \mathit{rest\text{-}var} \rangle\ \langle \mathit{exp} \rangle) \\
& \mid & (\textbf{prim}\ \langle \mathit{name} \rangle\ (\langle \mathit{arg} \rangle\ \ldots)) \\
& \mid & \langle \mathit{cont} \rangle \\
\\
\langle \mathit{arg} \rangle & \rightarrow & \langle \mathit{var} \rangle \\
& \mid & \langle \mathit{literal} \rangle \\
\\
\langle \mathit{cont} \rangle & \rightarrow & (\textbf{clambda}\ (\langle \mathit{var} \rangle)\ \langle \mathit{exp} \rangle) \\
& \mid & \langle \mathit{kvar} \rangle
\end{array}
$$

Figure 4.2: The SKI-CPS language.

4.4.2 SKI's CPS intermediate language

SKI uses a CPS based intermediate language, SKI-CPS, which is similar in spirit, but different in appearance, to that used in the Orbit paper [KKR+86]. This is a consequence of our decision to use Orbits CPS-conversion algorithm (Section 4.5).

Orbit uses a dialect of Scheme with continuations, called CPS-Scheme, which is the similar to the language used in the example in the previous section, with the exception that the continuation is always the first argument in a function call.

Figure 4.2 shows the SKI-CPS intermediate language. We will now explain what each form in the language does and show how they fit together. For clarity of explanation, we will start with ⟨*cont*⟩.

A ⟨*cont*⟩ can either be a **clambda** or a ⟨*kvar*⟩. A **clambda** is continuation procedure, it binds the value passed to it to the variable, ⟨*var*⟩, and executes the expression, ⟨*exp*⟩. The scope of the binding is the ⟨*exp*⟩. A ⟨*kvar*⟩ is a variable which holds a continuation.

The top level forms are the expressions, ⟨*exp*⟩. This name is probably a misnomer since ⟨*exp*⟩s don't return a result, they call a continuation.

A **jmp** form jumps to a continuation, ⟨*cont*⟩, passing value ⟨*carg*⟩. A ⟨*carg*⟩ is a continuation argument, it generates a value that can be passed to a continuation.

An **app** form calls the procedure in ⟨*var*⟩ passing it some arguments, (⟨*arg*⟩ ...) and a continuation ⟨*cont*⟩.

A **cif** is the CPS conditional expression. If the test-value ⟨*arg*⟩ is not false then the consequent branch continuation, ⟨*consequent-cont*⟩ is called, otherwise the alternate branch continuation ⟨*alternate-cont*⟩ is called. The value passed to the branch continuations is the continuation of the whole expression ⟨*cont*⟩. The branch continuations call their argument when they are finished. E.g., in the example below 'v' will be bound to "true" if 'x' is true and "false" otherwise.

```
(cif x
     (clambda (k1) (jmp "true" k1))
     (clambda (k2) (jmp "false" k2))
   (clambda (v) ...))
```

This is one of the places where SKI-CPS differs from CPS-Scheme. CPS-Scheme binds the continuation of the whole expression to a variable and the branch continuations call this variable when they are finished. E.g., the previous example would look like this;

```
(jmp (clambda (v) ...)
  (clambda (k)
    (cif x
         (clambda () (jmp "true" k))
         (clambda () (jmp "false" k)))))
```

Each approach has some advantages, the first keeps all of the information in one place, the **cif**, and this makes some of the conditional optimizations easier (Section 4.9). However it can result in one continuation having multiple names which complicates analysis in latter phases.

As we mentioned previously, a ⟨*carg*⟩ is something that has or produces a value which can be passed to a continuation. The reason that we have two kinds of arguments, the other being ⟨*arg*⟩, is to limit the number of places that the more complex kinds argument (**prims**, **nlambdas** and **clambdas**) can appear. This simplifies the rest of the compiler a great deal.

A **nlambda** makes a "normal" procedure which takes three kinds of arguments. The ⟨*kvar*⟩ is bound to the continuation of the procedure, the ⟨*var*⟩s are bound to the fixed[7] arguments and the ⟨*rest-var*⟩ is bound to the rest arguments. If the procedure has no rest arguments then we replace ⟨*rest-var*⟩ with '#f'.

A **prim** calls the primitive ⟨*name*⟩ with some arguments. The value of the **prim** is the value returned by the primitive. Primitives don't take continuations, this allows us to write primitives as normal 'C' functions or macro.

Finally, an ⟨*arg*⟩, is an argument to a procedure, primitive, etc. It can either be a variable or a literal. A ⟨*literal*⟩ can be any Scheme literal, e.g., 1, "hello world", 'a-symbol, '(a list).

4.5 CPS-conversion

As we mentioned in the previous section, SKI uses the same CPS conversion algorithm as Orbit [KKR+86] with a few minor differences. The CPS converter is implemented as a small number of rewrite rules, which we will express using the function $\mathcal{C}$ and the helper function $\mathcal{S}$. These functions take a Scheme form and a continuation, and yield a SKI-CPS expression, e.g., $\mathcal{C}[\![\langle exp \rangle]\!]\, k$.

Variables and literals are simply passed to the continuation:

$$\begin{aligned} \mathcal{C}[\![\langle var \rangle]\!]\, k &= (\textbf{jmp}\ \langle var \rangle\ k) \\ \mathcal{C}[\![\langle literal \rangle]\!]\, k &= (\textbf{jmp}\ \langle literal \rangle\ k) \end{aligned}$$

To convert a procedure call we simply convert all the operand expressions, $\langle arg_i \rangle$, binding their values to temporary variables t_i, then we convert the operator expression, $\langle op \rangle$, binding it to t_0 and finally we call the procedure with its continuation

$$\begin{array}{l} \mathcal{C}[\![(\langle op \rangle\ \langle arg_1 \rangle\ \langle arg_2 \rangle \ldots)]\!]\, k = \\ \quad \mathcal{C}[\![\langle arg_1 \rangle]\!]\, (\textbf{clambda}\ (t_1) \\ \qquad\qquad \mathcal{C}[\![\langle arg_2 \rangle]\!]\, (\textbf{clambda}\ (t_2) \\ \qquad\qquad\qquad\qquad \ddots \\ \qquad\qquad\qquad\qquad \mathcal{C}[\![\langle op \rangle]\!]\, (\textbf{clambda}\ (t_0) \\ \qquad\qquad\qquad\qquad\qquad (\textbf{app}\ t_0\ (t_1 \ldots)\ k)))) \end{array}$$

[7]A Scheme procedures can take a variable number of arguments. For instance '(**lambda** (x y . r) ...)' takes two or more arguments. The first two will are bound to the fixed argument variables, 'x' and 'y', and the rest of the arguments are put in a list which is bound to the rest argument variable, 'r'.

This is slightly different from the way that Orbit does this transformation. Orbit converts $\langle op \rangle$ first, then converts the $\langle arg_i \rangle$. In some cases slightly better code will be generated if we convert the operator last. The reason is that if the operator is a global variable and one of the arguments is a function call, then if we converted the operator first, the value of the global variable will be read into a local variable which will then have to be preserved across the function call (see section 5.1). For instance

```
(+ 1 (f x))
```

becomes

```
((prim $get-global 1) 1 (f x))
```

after assignment conversion. If we convert the operator first, we get[8]

```
(jmp (prim $get-global 1)
  (clambda (t0)
    (app f (x)
      (clambda (t1)
        (app t0 (1 t1))))))
```

and the value of 't_0' will have to be preserved across the call to 'f', but if we convert the operator last then the following code will be generated

```
(app f (x)
  (clambda (t1)
    (jmp (prim $get-global 1)
      (clambda (t0)
        (app t0 (1 t1))))))
```

and there is no need to save any temporary variables.

Converting primitives is similar to converting procedure calls.

```
C[[(prim ⟨prim-name⟩ ⟨arg1⟩ ⟨arg2⟩ ...)]] k =
  C[[⟨arg1⟩]] (clambda (t1)
                C[[⟨arg2⟩]] (clambda (t2)
                              ⋱
                                  (jmp (prim ⟨prim-name⟩ (t1 t2 ...)) k)))
```

[8]For clarity, all redundant bindings have been eliminated from these examples. The SKI-CPS code in these examples is more representative of the output of the redundant binding elimination phase (Section 4.6) rather than the CPS conversion phase. The output of the CPS conversion phase is considerably more verbose.

Converting **lambdas** requires three rules. The first rule converts **lambdas** with fixed arguments, the second converts **lambdas** with a fixed number of arguments and a rest argument, and the last converts **lambdas** with only a rest argument.

$$
\begin{aligned}
&\mathcal{C}[\![(\textbf{lambda}\ (\langle arg_1\rangle\ \langle arg_2\rangle\ \ldots)\ \langle body\rangle)]\!]\,k\ =\\
&\quad(\textbf{jmp}\ (\textbf{nlambda}\ k_1\ (\langle arg_1\rangle\ \langle arg_2\rangle\ \ldots)\ \#\text{f}\ \mathcal{S}[\![\langle body\rangle]\!]\,k_1)\ k)\\
&\mathcal{C}[\![(\textbf{lambda}\ (\langle arg_1\rangle\ \langle arg_2\rangle\ \ldots\ .\ \langle rest\rangle)\ \langle body\rangle)]\!]\,k\ =\\
&\quad(\textbf{jmp}\ (\textbf{nlambda}\ k_1\ (\langle arg_1\rangle\ \langle arg_2\rangle\ \ldots)\ \langle rest\rangle\ \mathcal{S}[\![\langle body\rangle]\!]\,k_1)\ k)\\
&\mathcal{C}[\![(\textbf{lambda}\ \langle rest\rangle\ \langle body\rangle)]\!]\,k\ =\\
&\quad(\textbf{jmp}\ (\textbf{nlambda}\ k_1\ ()\ \langle rest\rangle\ \mathcal{S}[\![\langle body\rangle]\!]\,k_1)\ k)
\end{aligned}
$$

$\mathcal{S}$ converts the bodies of **lambdas** and **lets** which are sequences of expressions.

$$
\begin{aligned}
\mathcal{S}[\![\langle exp_1\rangle\ \langle exp_2\rangle\ \ldots]\!]\,k &= \mathcal{C}[\![\langle exp_1\rangle]\!]\,(\textbf{clambda}\ (ign)\ \mathcal{S}[\![\langle exp_2\rangle\ \ldots]\!]\,k)\\
\mathcal{S}[\![\langle exp\rangle]\!]\,k &= \mathcal{C}[\![\langle exp\rangle]\!]\,k
\end{aligned}
$$

'*ign*' is a temporary variable which will never be used.

It is possible to transform **lets** into **lambdas** in the rewriting phase (Section 4.1), but we chose not to since **lets** are useful in assignment conversion phase. Transforming **lets** directly into CPS also reduces the amount of work for later phases, in particular rewriting **lets** as **lambdas** and then CPS transforming would produce many candidates for β-reduction (Section 4.7).

$$
\begin{aligned}
&\mathcal{C}[\![(\textbf{let}\ ((\langle var_1\rangle\ \langle exp_1\rangle)\\
&\qquad\qquad(\langle var_2\rangle\ \langle exp_2\rangle)\\
&\qquad\qquad\ldots)\\
&\quad\langle body\rangle)]\!]\,k =\\
&\quad\mathcal{C}[\![\langle exp_1\rangle]\!]\,(\textbf{clambda}\ (\langle var_1\rangle)\\
&\qquad\qquad\mathcal{C}[\![\langle exp_2\rangle]\!]\,(\textbf{clambda}\ (\langle var_2\rangle)\\
&\qquad\qquad\qquad\qquad\ddots\\
&\qquad\qquad\qquad\quad\mathcal{S}[\![\langle body\rangle]\!]\,k))
\end{aligned}
$$

The final type of expression that we have to convert is the **if** expression.

$$
\begin{aligned}
&\mathcal{C}[\![(\textbf{if}\ \langle \textit{test-exp}\rangle\ \langle \textit{exp}_1\rangle\ \langle \textit{exp}_2\rangle)]\!]\,k\ =\\
&\quad \mathcal{C}[\![\langle \textit{test-exp}\rangle]\!]\ (\textbf{clambda}\ (t)\\
&\qquad\qquad\qquad (\textbf{cif}\ t\\
&\qquad\qquad\qquad\quad (\textbf{clambda}\ (k_1)\ \mathcal{C}[\![\langle \textit{exp}_1\rangle]\!]\,k_1)\\
&\qquad\qquad\qquad\quad (\textbf{clambda}\ (k_2)\ \mathcal{C}[\![\langle \textit{exp}_2\rangle]\!]\,k_2)\\
&\qquad\qquad\qquad\ k))
\end{aligned}
$$

To complete the CPS conversion, we have to provide the first continuation. To do this we wrap a **clambda** around the top-level continuation using the $\mathcal{T}$ function, which takes a top-level expression and returns a SKI-CPS expression.

$$\mathcal{T}[\![\langle \textit{exp}\rangle]\!] = (\textbf{clambda}\ (k)\ \mathcal{C}[\![\langle \textit{exp}\rangle]\!]\,k)$$

When the SKI-CPS expression is executed, the runtime environment calls this clambda with another continuation which is bound to k, the expression is executed and eventually calls k with its result and the runtime resumes execution.

4.6 Redundant Binding Elimination

The SKI-CPS expressions produced by the CPS-converter in the previous section are rather verbose. In particular there are many sub-expressions of the form:

(jmp *a* **(clambda** (v) ...) ; Binds value of a to v.

which are redundant in since references to the variable v can be replaced with a. In practice it is possible to eliminate all such sub-expressions when:

1. a is a variable or a literal of an immediate type and not a symbol,[9]

2. v is never referenced and a is not a side-effecting primitive, or,

3. a is a **clambda** and v is referenced exactly once.

The condition on the first rule is for two reasons. Firstly, removing the binding may result in duplication of the literal and therefore duplication of the code required to construct the literal, this code can be quite complex (see Section 5.5). Secondly, some aggregate types (e.g. pairs and strings) are mutable and if they were duplicated then mutation might lead to unpredictable results.[10]

Rule two removes *dead code*—code that will never be executed and values that will never be used, but doesn't remove primitives which cause side effects. The last rule moves continuations which are only used once to the point where they are used, but avoids duplicating continuations.

Redundant binding elimination is a simple case of the β-reduction transformation discussed in the next section, but it is desirable for it to be a separate pass since it is used to "tidy up" after other optimization passes. It also simplifies other passes since it ensures that SKI-CPS expressions are in a regular form in which any value is bound to at most one variable.

[9]See section 5.6.1 for definition of immediate types.

[10]The R^{4}RS states that the it is an error to mutate the value of a literal expression, but doesn't require implementations to raise an error if a literal value is mutated. SKI, like many other Scheme implementations, will not raise an error, this would complicate the run-time system (see Section 5.6). Therefore it is desirable that if a constant is mutated then the result should be predictable.

4.7 β-reduction

The β-reduction transformation performs inline expansion [Bak92, App92] of all procedures which are called exactly once and and don't *escape*. Escaping procedures are procedures which are stored in data structures or global variables, passed to other procedures, or returned by the enclosing procedure. For instance in the following example, the procedure 'g' escapes 3 times:

```
(lambda ()
  (let ([g (lambda () ...)])
    (set! global g)         ; Stored in global,
    (h g)                   ; passed to a function, and
    g))                     ; returned.
```

When the β-reduction pass detects a procedure that meets the above conditions, its call is replaced with a copy body of the procedure in which all references to the formals have been replaced with the call's arguments. For simplicity this is actually done in two phases. The first phase replaces the application with the body and binds the arguments to the formals:

```
(jmp (nlambda k (x1 ...) #f      ⇒     (jmp ⟨cont⟩ (clambda (k)
       ⟨body⟩)                            (jmp a1 (clambda (x1)
  (clambda (f)                                   ⋱
    (app f (a1 ...) ⟨cont⟩)))                      ⟨body⟩)))))).
```

The second phase finishes the renaming by running the redundant binding elimination transform to remove the bindings we just introduced. Figure 4.3 shows an example of a β-reduction.

If a procedure has a rest argument it can still be β-reduced, but it is necessary to insert code to construct the rest list. E.g. if a procedure which takes one fixed argument and a rest argument 'r' is called with arguments (1 2 3) then the following code would be generated to construct the rest list:

```
(jmp (prim $cons 3 '())
  (clambda (t)
    (jmp (prim $cons 2 t)
      (clambda (r) ...)))).
```

```
(jmp (nlambda k (x y z) #f
          (app + (x y)
             (clambda (t)
                (app + (t z) k))))
  (clambda (f)
     (app f (1 2 3)
        (clambda (r) ...))))  .
```

(a) The **nlambda** 'f' is called once and doesn't escape—β-reduce it.

```
(jmp (clambda (r) ...)
  (clambda (k)
     (jmp 1
       (clambda (x)
          (jmp 2
            (clambda (y)
               (jmp 3
                 (clambda (z)
                    (app + (x y)
                       (clambda (t)
                          (app + (t z) k)
                          ))))))))))
```

(b) Pass 1: Substitute the body and bind the arguments.

```
(app + (1 2)
   (clambda (t)
      (app + (t 3)
         (clambda (r) ...))))
```

(c) Pass 2: Remove the redundant bindings introduced in pass 1.

Figure 4.3: A example of β-reduction.

β-reduction is different from general inline expansion in that it is guaranteed not to increase the size of the program—it will always result in a simpler program.

4.8 η-reduction

η-reduction is similar to β-reduction. Like β-reduction it is a form of inline expansion, but rather than expanding procedures which are called once, it expands procedures which do nothing but call other procedures. E.g.,

```
(jmp (nlambda k (a1 ...)              ⇒  (app g (x1 ... b1 ...) ⟨cont⟩).
         (app g (a1 ... b1 ...) k))
     (clambda (f)
       (app f (x1 ...))))
```

The b_i above are to indicate that the g may take more arguments than f, but the order of the extra arguments is not important to the transform.

η-reduction is accomplished using the same two phase technique as β-reduction (see previous section), and like β-reduction it always results in a simpler program.

4.9 Conditional Optimizations

The conditional optimization pass consists of three transformations for optimizing conditional, **cif** expressions. These transformations were taken from the Orbit paper [KKR+86].[11]

The first transformation eliminates the unreachable branches in **cif**s where the test value is a literal. E.g.,

```
(cif l                  ⇒    (jmp cont then-branch)    l ≠ #f,
     then-branch
     else-branch             (jmp cont else-branch)    l = #f,
  cont)
```

The **jmp** may then become a candidate for redundant binding elimination.[12]

The second transformation is very similar to the first, it propagates the

[11]This paper also contains a long example showing how effective these optimizations can be, which we will not duplicate here.

[12]Remember that branch continuations of a **cif** take the continuation of the **cif** as an argument.

boolean values of variables into the branches of a conditionals. For example, in

(**cif** *v then-branch else-branch cont*)

the boolean value of the variable *v* is known to be true in *then-branch* and false in *else-branch*. Therefore if a **cif** which tests *v* is encountered in the branches we can eliminate one of the branches.

```
(cif v                ⇒      (jmp cont consequent)   if v true,
      consequent
      alternate              (jmp cont alternate)    if v false.
   cont)
```

The last transformation rearranges nested conditional expressions where the result of one conditional is used as the test value of the other. E.g., in Scheme:[13]

(**if** (**if** *a b c*) *d e*).

Which, by rearranging the order of the **if**s, can be transformed into:

(**if** *a* (**if** *b d e*) (**if** *c d e*)),

or to avoid duplicating *d* and *e*:

```
(let ([x (lambda () d)]
      [y (lambda () e)])
  (if a (if b (x) (y)) (if c (x) (y)))).
```

Unlike the previous optimizations this one actually increases the complexity of the program, but it is worthwhile since it often leads to more optimizations. In particular it is often possible to eliminate one, or sometimes both of the **if**s that were introduced. Kranz *et al.* [KKR+86] gives an example of this.

4.10 Implementation

At the start of this chapter we stated that the front end is structured as a number of passes, each of which performs some transformation on the program and hands it onto the next pass. In practice it is organised slightly differently. Some logically independent passes are combined into a single pass, and some logical

[13]We will use Scheme for the following examples since the CPS versions are rather verbose. The full CPS version of this transform is shown in Section A.2

passes require two passes over the program, one to gather information, the second to perform the transformations. For example, rewriting and α-conversion (Sections 4.1 and 4.2) are done in a single pass. β-reduction and η-reduction (Sections 4.7 and 4.8) are also done in a single pass, but require a separate information gathering pass to discover which procedures escape and how often each procedure is used.

The optimization passes are actually performed several times since some optimizations introduce or uncover opportunities for further optimization, and as we mentioned above redundant binding elimination pass is used to "tidy up" after the transformations.

Chapter 5

The Back End

The back end takes SKI-CPS expressions generated by either the front end (Chapter 4) or the dynamic optimizer (Chapter 6) and generates C code which is compiled by the C compiler and linked into the running system. Figure 5.1 shows an overview of the back end.

In the remainder of this chapter we dicuss the reasons for closure allocation and callee save variables (section 5.1 and 5.2). We then detail the closure allocation algorithm used by SKI (Section 5.3). Sections 5.4 and 5.5 discuss issues invloved in generating C code and finally Section 5.6 discusses the runtime type system.

5.1 Closure conversion

Before we can generate executable code, there are two problems that must be solved:

1. SKI-CPS, like Scheme, has lexically scoped procedures. This means that procedures can be nested inside other procedures, and these nested procedures can refer to variables declared in the procedures enclosing them. Variables which are referenced in one procedure, but declared in an enclosing procedure are said to be *free* in the inner procedure.

2. When a procedure, a, calls another procedure, b, the values of a's variables must be saved somewhere so that they can be restored when b returns, or

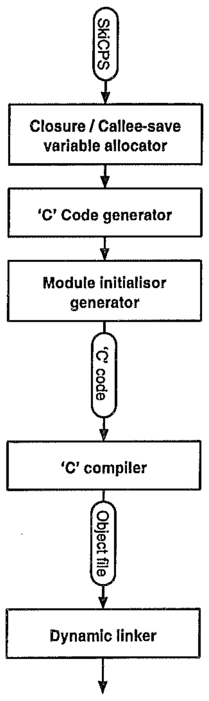


Figure 5.1: Overview of the back end of Ski.

equivalently, when *b* calls *a*'s continuation.

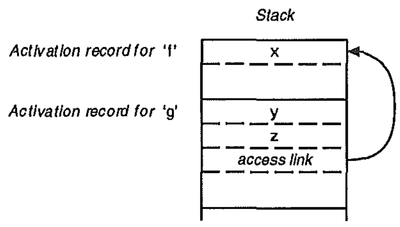
It turns out that these problems are very closely related and that the mechanisms used to solve them are very similar.

Note: In the following discussion we assume that the target architecture of the compiler is register based, rather than stack based. We also assume that each variable is mapped onto a register and that there are sufficient registers to hold all active variables.

Traditionally, languages with lexically scoped procedures, like Algol and Pascal, put free variables into linked activation records on a stack (see Chapter 7 of *The Dragon Book* [ASU86]). If we were to use linked activation records on a stack for Scheme, the program in Figure 5.2a would have a stack like the one in Figure 5.2b when executing procedure 'g'. The variable 'x' which is free in 'g' can be accessed in 'g' by indirection through the *access link*. The stack can also be used to save variables across procedure calls, the variables are pushed onto the stack before the procedure call and restored from the stack after the call returns.

```
(define f
  (lambda (x)
    (let ([g (lambda (y z)
               (+ (* x y) z))])
      (g 2 3))))
```

(a)



(b)

Figure 5.2: A Scheme program with nested procedures and the corresponding stack with access links between activation records. **Note:** The stack grows downwards.

Stack allocation of activation records would work well if nested procedures never escaped from the enclosing procedures. However, Scheme procedures can be passed as arguments to other procedures, returned from their enclosing procedures and stored indefinitely in data structures. Consider what happens if we change the program in Figure 5.2a so that 'f' now returns 'g':

```
(define f
  (lambda (x)
    (let ([g (lambda (y z)
               (+ (* x y) z))])
      g))).
```

When 'f' returns its activation record is popped off the stack and destroyed, so there is nowhere for 'g's access link to point.

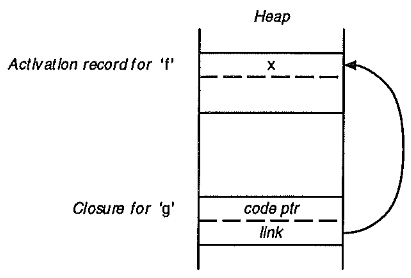


Figure 5.3: A heap allocated activation record and closure.

The obvious solution to this problem is to store the activation records on the heap rather than the stack and when we return a procedure we can return a pointer to the procedure plus a pointer to its enclosing activation record. This pair of pointers is called a *closure*. Since the heap is garbage collected activation records that are no longer necessary will be reclaimed eventually and activation records which are still pointed to by a closure will be retained. Figure 5.3 shows the closure returned by the modified version of 'f'.

There are many ways to represent closures, the two most obvious are called *linked* and *flat*. Linked closures are similar to linked activation records. The closure for each procedure has a pointer to the closure of the enclosing procedure. In contrast, a flat closure holds *copies* of all the variables that are free in its procedure. Figure 5.4a shows the linked closures for the case where procedure 'C' is nested inside procedure 'B' which is nested inside procedure 'A' and Figure 5.4b shows the flat closures for the same case. Note that it is unnecessary to store all of a procedures variables in the closures. In the linked case, 'A's closure only needs to contain those variables which are free in 'B' and 'C', and 'C's closure doesn't need to hold any variables since it encloses no other procedures. In the

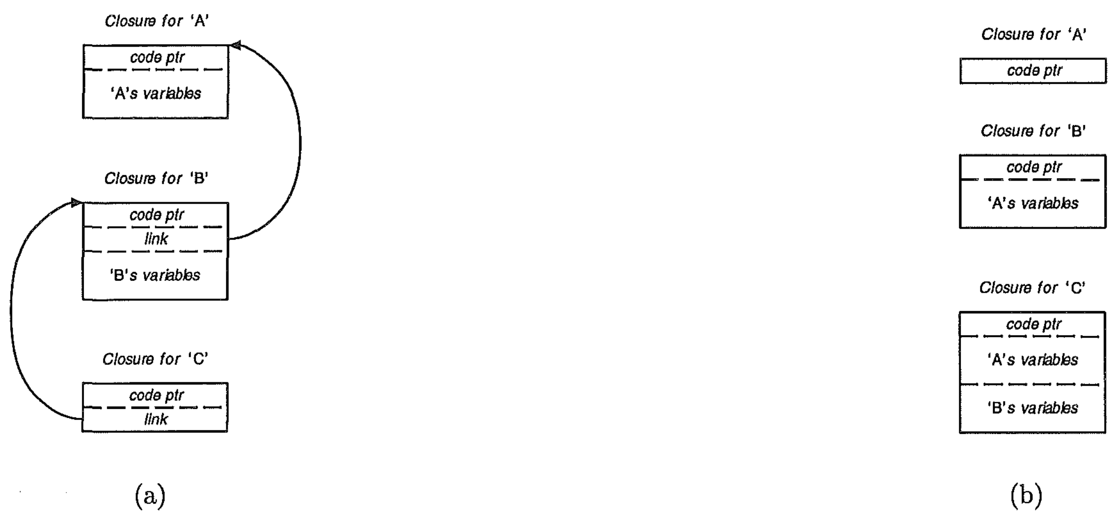


Figure 5.4: Linked closures and flat closures

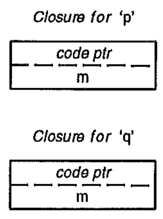
flat case, 'B's closure need only hold copies of those variables which belong to 'A' and are free in 'B', 'A's closure need not hold any variables since 'A' has no free variables.

Flat closures have the advantage that access to the variables they contain takes a small constant time, whereas access to variables in linked closures is linear in the depth of nesting. The disadvantage of flat closures is that they may require more memory than linked closures since variables may be duplicated among many closures, flat closures may also take more time to create than linked closures.

One possible problem with flat closures is that assignment to a variable which is duplicated in two or more closures might lead to inconsistent results. E.g., in the code fragment shown in Figure 5.5a, 'm' is free in 'p' and 'q'. If we make make closures for 'p' and 'q' naïvely, as shown in Figure 5.5b, then the value of 'm' in 'p's closure could be updated independently of the value of 'm' in 'q's closure. Fortunately we will not encounter this problem in Ski since the code will have been assignment converted (Section 4.3) and, as shown in Figure 5.6, 'm' is no longer free in 'p' and 'q' as it has been replaced by a cell, the reference to this cell, 'm-cell' may be duplicated freely. Without assignment conversion we would

```
(define r
  (lambda (m)
    (let ([p (lambda ()
               . . .
               (set! m . . .)
               . . .)]
          [q (lambda ()
               . . .
               (set! m . . .)
               . . .)])
      . . .)))
```

(a)

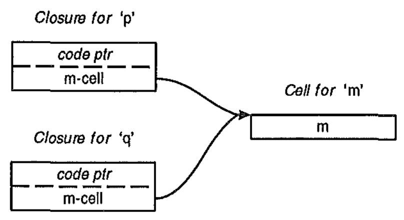


(b)

Figure 5.5: Assignments can lead to problems when variables are copied.

```
(define r
  (lambda (m)
    (let ([m-cell (prim $make-cell m)])
      (let ([p (lambda ()
                 . . .
                 (prim $set-cell! m . . .)
                 . . .)]
            [q (lambda ()
                 . . .
                 (prim $set-cell! m . . .)
                 . . .)])
        . . .))))
```

(a)



(b)

Figure 5.6: Mutable variables are shared with assignment conversion.

have to arrange to do something similar when we introduce closures or discard the idea of flat closures. Of course, linked closures don't have this problem, since 'm' would be in 'r's closure.

The problem of how and where to save variable across procedure calls still remains. It is still possible to save variables on the stack, but this can lead to problems with first class continuations. These problems occur when a continuation "captures" the stack.

```
(define task-queue '())
(define (enqueue-task k) ...)
(define (dequeue-task) ...)
(define (suspend-task)
  (call-with-current-continuation [lambda (k)
                                     (enqueue-task k)
                                     ((dequeue-task) #f)]))
```

Figure 5.7: A simple coroutine package.

Consider the simple coroutine package outlined in Figure 5.7. 'enqueue-task' puts a coroutine, represented by a continuation, in the task queue, 'dequeue-task' gets the next coroutine off the task list and 'suspend-task' suspends the current coroutine and resumes execution of the next coroutine in the task queue. When 'suspend-task' is executed by a coroutine, it calls the builtin procedure 'call-with-current-continuation' (or 'call/cc').

'Call/cc' stores the current state of execution into a continuation object, 'call/cc' then passes the continuation object to its argument which must be a single argument procedure. A continuation object behaves as if it were a procedure of one argument, calling the continuation object resumes execution at the point immediately after the call to 'call/cc' that created the continuation. The return value of 'call/cc' is the argument that was passed to the continuation.[1] Continuations are first-class objects with indefinite extent, they may be stored in variables and called again at anytime in the future, they may also be called any number of times.

[1]E.g., (call/cc [**lambda** (k) (k 1)]) $\Rightarrow$ 1.

So in 'suspend-task', the continuation object is bound to 'k' and stored in the task-queue. The next coroutine is resumed by retrieving its continuation object from the task queue and calling it.

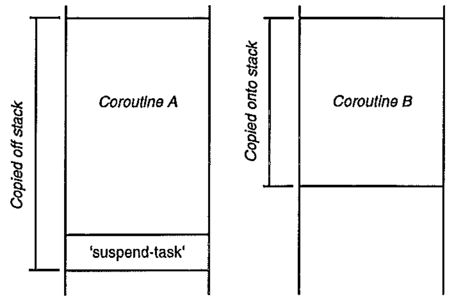


Figure 5.8: Using a stack in the presence of first class continuations leads to excessive copying.

If variables are saved on the stack then the "current state of execution" includes the stack. This means that when a continuation is created the stack must be copied into the continuation object, and when the continuation is called the stack must be copied back again. Figure 5.8 shows what happens when we switch coroutines. When coroutine A invokes 'suspend-task' a copy of the stack is copied into a continuation object by 'call/cc'. The stack is "unwound" and when the next task is resumed, its stack is restored when its continuation is called.

Copying the stack back and forth can be very expensive in both time and memory. There have been several attempts to reduce this cost. Hieb *et al.* [HDB90] proposed a scheme that uses stack segments to limit the amount of copying. In this scheme it is only necessary to copy some part of a stack segment when a continuation is invoked. This is especially useful when continuations are used to implement exceptions which are often created but rarely invoked.

A far simpler, but more contentious scheme is to dispense with the stack entirely and save variables in special records, which we will call *closures*, on the heap. This is much simpler than saving variables on the stack since we do not have to worry about copying anything for continuations—the saved variables part of the "current state of execution" is just a reference to a closure, which may refer to previous closures if they are necessary.

It may seem confusing that we call these new records *closures* when we already have objects called closures that do something entirely different, but it turns out that these records perform exactly the same function as "normal" closures. Consider this expression and its SKI-CPS equivalent:

```
(let ([s ...])            ⇒        (jmp ...
  (+ (* 3 t) s))                     (clambda (s)
                                       (app * (3 t)
                                         (clambda (u)
                                           (app + (u s) ...))))).
```

The variable 's' must be saved across the call to '*'. Interestingly, 's' is also a free variable in the continuation of the call to '*', (**clambda** (u) ...), in fact it turns out that those variables which have to be saved across procedure calls are those variables which are free in the continuation of the call, therefore the record that we save these variables in is a closure for the continuation of the call.

In practice the closures created for continuation procedures are slightly different from the closures created for "normal" procedures. Continuation closures don't contain a pointer to the continuation's code for two reasons.

1. There is no need to represent the continuation and its closure as a single value. The compiler can maintain a mapping between a continuation and its closure.

2. Memory usage can be reduced by sharing closures between continuations.

5.1.1 Stack Allocation vs. Heap Allocation

We mentioned previously that saving variables on the heap is contentious. There is currently some debate about whether saving variables on the heap is as efficient as saving them on a stack, which is summarised in the rest of this section. The following assumes the reader has some knowledge of modern computer architecture and in particular cached memory subsystems.[2]

In *Garbage Collection Can Be Faster that Stack Allocation* [App87], Appel

[2] A good book about computer architecture is *Computer Architecture: a Quantitative Approach* by Hennessy and Patterson [HP90] and a good overview of garbage collection can be found in *Uniprocessor Garbage Collection Techniques* by Wilson [Wil92].

calculated that in terms of instruction counts, allocation of continuation closures[3] on a garbage collected heap is faster than allocation on the a stack. His argument is as follows:

- Copying garbage collectors only ever touch live date and the amount of live data in a heap is (usually) much smaller than the amount of garbage.
- The cost of reclaiming the memory occupied by garbage is proportional to the amount of live data.
- The costs of allocating memory on heaps and stacks is roughly the same. When the heap is managed by a copying, compacting garbage collector allocation is just a matter of an add to the heap's free pointer. On a stack it is just an add to the stack pointer.
- The cost of reclaiming memory from a stack is one instruction per object, an add to adjust the stack pointer.
- So, if the amount of memory occupied by dead closures is sufficiently larger that that occupied by live closures then reclaiming it using garbage collection is faster than reclaiming it from the stack.

Appel calculated that the crossover point is when the number of dead closures is about 7 times the number of live closures, and that when this ratio is exceeded then heap allocation with copying garbage collection is cheaper than stack allocation.

The principle argument against Appel's theory is that it failed to account for the fact that modern computers have cached, hierarchical memory systems and that stacks exhibit good cache locality whereas heaps have poor locality and tend to "thrash" caches [HDB90].

Recently, Tarditi *et al.* [TDM94] and, Appel and Shao [AS94] have published results which indicate that programs using heap allocation do have good locality of reference.

They re-did Appel's original analysis in greater detail and found that the instruction count costs of heap allocation and stack allocation are exactly the

[3]For the purposes of this discussion we are only interested in continuation closures, it is assumed that normal closures an other large objects are heap allocated.

same. They also simulated cache behaviour for a variety of cache sizes and found that while heap allocation and stack allocation have similar read miss rates, heap allocation results in a much higher write miss rate. Depending on cache architecture, the high write miss rate can have significant performance impact or none at all.

There are a number of policies for handling write misses [Jou93]:

Write-allocate: When a write misses, allocate a line in the cache and either:

> ***Fetch-on-write:*** Fetch the contents of the cache line from memory (except for the word that is to be overwritten) and perform the write.
>
> ***Write-validate:*** Mark the words in the cache line as invalid, except for the one written to. If a read hits a word that is marked invalid then read the word (or the rest of the line) from memory.

Write-through: When a write misses, perform the write in memory. Only allocate cache lines on read misses.

If *write-allocate* with *write-validate*[4] is used, then write misses cost nothing, but *fetch-on-write* results in wasted memory traffic as the words fetched to fill the cache line will probably be overwritten. *Write-through* will also cause much memory traffic, since the locations written out to memory will probably be read soon afterward and will have to loaded back into the cache.

Appel and Shao [AS94] concluded that heap allocation will have similar performance to stack allocation if the cache supports *write-allocate* with *write-validate,* or there are cache hint or pre-fetch instructions which can be used to simulate write allocation [App94].

Unfortunately, current implementations of the two architectures Ski currently runs on, the SPARC and the Intel 386, do not have good policies for handling cache write misses. However, this is an implementation issue rather than a architectural issue—maybe future implementations of the architectures will have better cache write miss policies.[5] Despite this Ski uses heap allocation since

[4]Also known as *write-allocate* with *sub-block-placement.*

[5]The UltraSPARC processor, due to be released sometime in 1995 will have a write-allocate with write-validate second level cache [Nor95].

it is much than simpler stack allocation and 'call/cc' can be implemented very efficiently.

5.2 Callee-save variables

The problem with saving variables in closures across function calls is that many closures are created and almost immediately discarded. This is costly for two reasons.

1. Referencing memory is expensive. Creating a closure to store n variables takes $n + 1$ store instructions, 1 for each of the variables and 1 for the tag (see Section 5.6), and loading them back into registers takes n load instructions.

2. Since the memory used by closures must be reclaimed by garbage collection, the more closures we allocate the more frequently we have to garbage collect.

These problems are also faced in more conventional languages which store variables on the stack. There are two policies that can be used for saving registers. The policy we have assumed until now is to have the calling function (the *caller*) save those registers which are live across the call, or equivalently save those registers which are imported by the continuation. This is called the *caller-save* policy. The other policy to have the called function (the *callee*) save those registers which it needs and restore them before it returns.

Both policies have their advantages and disadvantages. The callee-save policy has the advantage that it only saves those registers which it needs use, but it has no way of knowing whether the registers it saves contain useful information—it could save some registers needlessly. Conversely, the advantage of the caller-save policy is that only those registers which need to be saved are saved, but the caller has no way of knowing that the callee will use any of the registers it saves—it too could save some registers needlessly.

In practice a mixture of the two policies appears best. The registers are partitioned into two sets, one set is caller-save and the other is callee-save.

Appel [App92, AJ89] showed how it is possible to use the callee-save policy in CPS. Each continuation is given m[6] extra arguments in addition to the result argument. E.g.,

(**clambda** (v) ...) $\Rightarrow$ (**clambda** $(v\ c_1\ c_2\ \ldots\ c_m)$ ...).

These are used to pass either the values of the variables that the continuation imports, if there are less than m, or the first $m-1$ imported variables and reference to a closure containing the rest. Each procedure is also given m extra arguments in addition to its continuation and its "normal" arguments. E.g.,

(**nlambda** k $(a_1\ a_2 \ldots)$ ...) $\Rightarrow$ (**nlambda** k $(c_1\ c_2\ \ldots\ c_m\ a_1\ a_2\ \ldots)$ ...).

The values of these arguments must be passed to continuation of the procedure.

```
(app f (a b)
  (clambda (v)
    (app g (x y)
      (clambda (w)
        ...        ;; Uses v, y, z.
        ))))
```

(a)

```
(app f (x y z a b)
  (clambda (x1 y1 z1 v)
    (app g (v y1 z1 x1 y1)
      (clambda (v1 y2 z2 w)
        ...
        ))))
```

(b)

Figure 5.9: Adding callee-save variables. The callee-save variables are indicated by sans-serif.

Using these arguments a caller can pass variables to its continuation across a procedure call without having to save them in memory. The callee is, of course, free to save the variables as long as it restores them before passing them to its continuation. We therefore call the variables we pass using these arguments *callee-save variables*.

Figure 5.9a shows a (somewhat idealised) sequence of two procedure calls. The first continuation, (**clambda** (v) ...), imports the variables 'x', 'y' and 'z' from an enclosing scope. The second continuation imports 'v', 'y' and 'z'. Using only closures[7] we would have to save 'x', 'y' and 'z' in closure, and pass the closure to the first continuation. We would then retrieve 'x', 'y' and 'z' from the first

[6]The value of m is constant over the entire program.

[7]I.e., a pure caller-save policy.

closure and save 'v', 'y' and 'z' into a second closure and pass it to the second continuation ... The cost to the caller of this would be $2 \times (3+1) = 8$ stores to create two 3 variable closures and $2 \times 3 = 6$ loads to retrieve the variables. However if we use callee-save variables (as shown in Figure 5.9b) we can reduce the cost of saving the variables to almost zero.[8]

5.3 Ski's closure allocation algorithm

In the previous two sections we have discussed methods for saving variables across procedure calls and for accessing variables declared in lexically enclosing procedures. In this section we present an overview of the algorithm that Ski uses to decide whether a variable needs to be stored and where it should be stored. We call this algorithm the closure allocation algorithm, but it really should be called the closure and callee-save variable allocation algorithm since it has to decide whether to make a variable callee-save or store it in a closure.

Closure allocation is a difficult problem [SA94, App92] and finding a perfect solution to it may be impossible. Ski uses a simple heuristic algorithm which appears to perform quite well.[9] Ski's algorithm is influenced by the algorithm recently developed by Shao and Appel [SA94] for the SML/NJ compiler, but is much simpler.

As mentioned previously there are two kinds of closures: closures for "normal" procedures which hold variables that a procedure imports from enclosing scopes and continuation closures which are used to save variables across procedure calls. We will call these **nlambda** closures and **clambda** closures.

Allocating **nlambda** closures is very easy, we simply have to calculate the *imports* set for each **nlambda** and make a closure that contains the imports set and a pointer to the code for the nlambda.

The *imports* set of a lambda (**nlambda** or **clambda**) is defined as the set-difference[10] of the set of variables referenced in the body of the lambda and the

[8]The extra cost could probably be reduced to one register-to-register move instruction to move 'v' from the result register to the first callee save register in the first continutation. Assuming that all variables are mapped into registers.

[9]All that can be said about it is that it is better than the algorithms that came before it.

[10]The set-difference of two sets A and B is defined as the set of elements in A which are not

set of variables declared by the lambda.

So the transformation for **nlambdas** is:

```
(jmp (nlambda k (a1 a2 ...)
        ...)
  (clambda (f)
    ...))
⇒
(jmp (nlambda k cl (c1 c2 ... cm a1 a2 ...)
        ...)
  (clambda (t)
    (jmp (prim $make-clo t i1 i2 ...)
      (clambda (f)
        ...)))).
```

Where the t is a temporary introduced to hold the pointer to the procedures code, f now references the procedures closure, i_1, i_2, ... are the variables that the procedure imports, *cl* is the procedures closure and c_1, c_2, ... c_m are the procedures callee-save variables. Within the body of the procedure, the callee-save variables are associated with the procedures continuation, k, and wherever the continuation is called the callee-save variables are passed. E.g.,

```
(jmp r k)                ⇒    (jmp c1 c2 ... cm r k)
(app g (p1 p2 ...) k)         (app g (c1 c2 ... cm p1 p2 ...) k)
```

and if k appears as the continuation of a **cif**,

```
(cif v (clambda (k1) ...) (clambda (k2) ...) k)
```

then the callee-save variables are associated with k_1 and k_2 as well.

Allocating **clambda** closures is harder. Firstly, we have to decide which **clambdas** *escape* and therefore need special attention, and secondly, we have to decide whether the escaping **clambda** needs a new closure or whether we can provide all the variables it imports using using callee-save variables and existing closures.

An *escaping* **clambda** is defined as one which is passed as the continuation to a procedure, e.g.,

```
(app f (a1 a2 ...) (clambda (v) ...)),
```

in B.

or one which appears in the continuation position of a **cif**, which has a procedure application in either of its branches, e.g.,

```
(cif v (clambda (k1) ...)
       (clambda (k2)
         ...
         (app f (...)
           ...))
      (clambda (v) ...)).
```

This special case is because we can generate better code for **cif**s which don't have applications in the branches (see Section 5.5).

Before allocating closures we first determine the imports and *uses* sets for each lambda (**clambda** or **nlambda**) and the *range information* on each imported variable. The *uses* set is a subset of the imports set mentioned above, it contains those variables which are imported and used within the lambda before any enclosed **clambda**s. E.g.,

```
(clambda (a)                    ; imports: (c b) uses: (b)
  (jmp (prim $cons b a)
    (clambda (p)                ; imports: (c)   uses: (c)
      (jmp (prim $cons c p)
        (clambda ...)))))
```

The uses set determines which variables need to be in registers for each **clambda**.

The *range information* is the number of escaping **clambda**s that must be crossed before a variable is next used. The number of escaping **clambda**s is an indication of the amount of time until the variable is next used. This is determines whether a variables is made callee-save or whether it is stored in a closure, variables which are to be used soon are made callee-save if possible, variables that are not used for a long time are stored in closures.

Figure 5.10 shows SKI's strategy for saving variables for escaping **clambda**s:

- Every escaping **clambda** gets m callee-save variables. If the **clambda**s imports can be satisfied by the with m or fewer variables then they are passed via the callee-save variables (Figure 5.11a).

- If the **clambda** imports more than m variables then the first $m - 1$ are

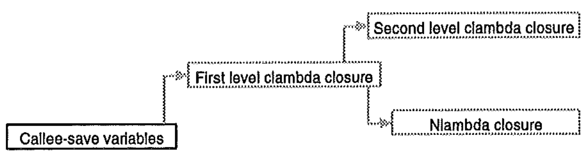
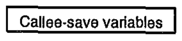


Figure 5.10: SKI's strategy for **clambda** closures. Parts in grey are optional.

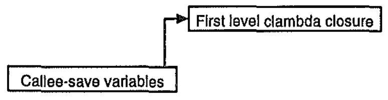
passed by making them callee-save and the last callee-save variable is used to pass a closure containing the rest of the imports (Figure 5.11b). Variables are made callee-save according to their priority. A variable's priority is determined using the range information and by whether the variable is already callee-save or not. Variables which will be used sooner are given priority over those that will be used later and variables that are already callee-save are given priority over those that aren't, these two rules help make closures last longer and help keep the most frequently used variables in registers.

If a first level closure already exists then we check it to see whether it can be reused. This is possible when the set of variables in the existing closure is a superset of the variables imported by the clambda or when the differences between the two sets can be covered by the changing some of the callee-save variables.

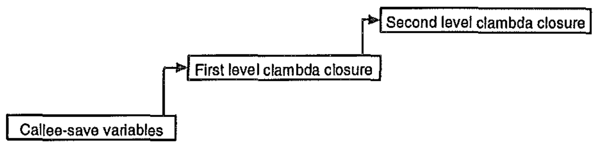
- The second level closure is used to hold variables which are long lived but referenced infrequently and would otherwise be copied from one first level closure to another. As an approximation we fill the second level closure with the enclosing **nlambda**s callee-save variables and it's continuation. As a further approximation, we only make the second level closure if there are two or more escaping **clambda**s between the start of the **nlambda** and the use of the continuation. The second level closure is created at the start of the **nlambda**, if a first level closure is created for any escaping **clambda** then the first level closure is is made to reference the second level closure (Figure 5.11c) otherwise the last callee-save variable references the second level closure (Figure 5.11d).



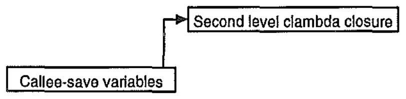
(a) No closures.



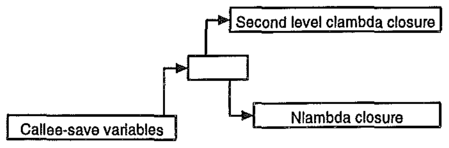
(b) Just the first level closure.



(c) The first level and second level closures.



(d) Just the second level closure.



(e) Second level and **nlambda** closures, empty first-level closure.

Figure 5.11: Some of the permutations possible for SKI's **clambda** closure strategy.

- The **nlambda** closure is the enclosing **nlambdas** closure. A reference to it is kept if it holds more than one variable that is not referenced until after an escaping **clambda**. This also helps to reduce copying from closure to closure. If a first level closure is created then it will reference the **nlambda** closure, if there is no first level closure then the last callee-save variable will reference the last **nlambda** closure, unless there is also a second level closure in which case a otherwise empty first level closure will be created to reference both (Figure 5.11e).

At the start of each **clambda** we check that the variables in the **clambda**'s uses set are in registers. If they are not then code is generated to load them from whichever closure they are in. Loading variables from the second level and **nlambda** closures when there is a first level closure is made as efficient as possible, as we cache the reference to the second level closure in a register. Loading the first variable from a second level closure costs two loads, one to load the reference to the second level closure from the first level closure and a second to load the variable, but loading another variable from the same closure only costs a single load.

5.4 C as a Target

SKI generates C code as its target language. In recent years this has become a popular method of code generation. Several other Scheme compilers generate C including Bartlett's Scheme->C compiler [Bar89], and more recently SCHEMEXEROX [ACS93], Bigloo Scheme compiler and GambitC. Other languages which have been compiled to C include: procedural languages such as Pascal [Gil91], Fortran (the GNU f2c compiler) and Napier88 [CCKM94]; statically typed object-oriented languages such as Eiffel [Mey88] and Sather [Omo93]; strict and non-strict functional languages such as ML [TAL91, Cri92, SW94], Erlang [Hau94], Sisal and Haskell [Jon92]; declarative languages such as Mercury [SHC94] and RML [Pet94]; dialects of Lisp such as Kyoto[YH85] Common Lisp and GNU Common Lisp.

Reasons for choosing C as the target, rather assembly language or raw machine code are:

1. It simplifies the compiler. Generating assembler or raw machine code is difficult, although there are tools such as code generator generators which can help. If a code generator generator had been used it would still be neccessary to write tables to drive it (i.e. to map our primitive operations onto the machines), and have to present our intermediate language to it in a form that it could understand. Even then other problems like register allocation might have to be solved.

   Generating C is *much* easier. Furthermore it is possible to take advantage to the C compiler to do register allocation, instruction scheduling and other optimizations.

2. C is easy to debug. The C code generated by the compiler can be debugged using standard debugging tools. On the other hand, code generators and the code generated by them are notoriously difficult to debug.

3. C is portable. Because it generates C, SKI is easily portable and in fact has already been ported from Sun SPARCstations running SunOS 4.1 to Intel 386 based PCs running Linux. SKI should be portable to any architecture that runs a modern version of Unix.

If we generated assembly language or machine code we would have to write a new code generator for each architecture, or if we used code generator generator then we would have to write new tables to drive it.

However, generating C is not without disadvantages:

1. It is slow and inefficient. To execute some code in our scheme system we have to generate the C code, write it to a file, invoke the C compiler to compile it, link the object file that the C compiler generated into our Scheme system and finally run it. All this takes quite a long time, generally a few seconds on a fast workstation. It is also wastes CPU time, since the code has to be written to a file, read back in, parsed, etc.

   Clearly this kind of performance is not suitable for a production quality interactive environment, but for an experimental system it is quite sufficient.

2. The semantics of Scheme do not map exactly onto the semantics of C. In particular C procedure calls are not properly tail recursive, integer arithmetic in C doesn't not detect overflows and we have to interface the generated code with the garbage collector that manages the Scheme systems heap. Fortunately, it is not too hard to work around these problems, (see Section 5.4.1).

The remainder of this section deals with generating C code. First we discuss the problem of making C procedure calls properly tail recursive. We then describe SKI generates C code from SKI-CPS. Finally, we describe the interface between the generated code and the garbage collector.

5.4.1 Tailcalls

The major difficulty with generating efficient C code is that function calls in C are not properly tail recursive. To be properly tail recursive, the compiler must generate efficient code for *tailcalls*.

A function call is a tailcall if it is the last thing that happens in a function before it returns. For example, the function call 'f(i−1)' in Figure 5.12 is a tailcall, but 'g(i−1)' is not. Since a tailcall is the last thing that happens in a function

```
void f(int i)
{
    ...
    f(i - 1);
}
```

(a) A tailcall.

```
void g(int i)
{
    ...
    g(i - 1);
    ...
}
```

(b) Not a tailcall.

Figure 5.12:

before the function returns, it is not necessary to save the state of the function across the call. A efficient tailcall not should increase the size of the stack and in general should not increase the amount of memory used by the program.

```
void f1(int i)
{
  again:
    ...
    i = i - 1;
    goto again;
}
```

Figure 5.13: Simple tail recursion elimination.

Some C compilers[11] can recognize simple recursive tailcalls and perform the tail recursion elimination, transforming a function like 'f' in Figure 5.12 into something similar to 'f1' in Figure 5.13, which is functionally identical, but will not cause any stack growth.

However the author is not aware of any C compiler that can generate properly tail recursive code when the functions are not immediately recursive, like the functions in Figure 5.14 where 'j' calls 'k' in the tail position and and 'k' calls 'm' in the tail position, all of these calls could be done with no net growth in stack space.

[11]For instance GNU CC [Sta92].

```
void j(int x)
{
    ...
    k(x + 1);
}

void k(int y)
{
    ...
    m(y - 1);
}
```

Figure 5.14: Non recursive tailcalls.

Efficient tailcalls are necessary for two reasons:

1. The R^4RS standard requires all Scheme implementations to be *properly tail recursive.*

2. All calls in CPS are tailcalls. If the compiler doesn't generate efficient code for tailcalls then our programs may run very slowly and use a lot of memory.

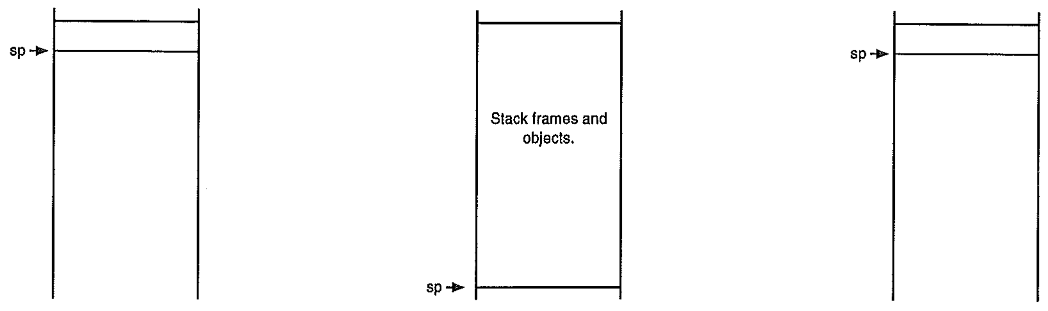
The problem of efficient tailcalls in C and other languages which don't have them has been encountered before and there are a number of possible solutions.

Ignore Tailcalls

One non-solution is to ignore the problem. The Scheme->C compiler [Bar89] performs its own tail recursion elimination for simple tail recursive procedures like the one shown in Figure 5.12, but generates normal C function calls for all other tailcalls.

The advantage of this scheme is that the C code emitted by the compiler is very portable and most of the time the performance will be reasonable compared to some of the other schemes discussed later. The disadvantages are that the compiler no longer conforms to the standard and that sometimes programs will use a large amount of memory or even fail to work owing to limits on the amount of memory available.

The Pushy scheme



(a) Save sp by calling 'setjmp' ...

(b) When the stack reaches a certain size, reset it by calling 'longjmp' ...

(c) Continue execution ...

Figure 5.15: Using the Pushy scheme. sp is the stack pointer.

A similar solution is the *Pushy* scheme which was recently invented by Baker [Bak94]. The Pushy scheme uses normal C function calls in place of tailcalls. When the stack gets larger than a certain limit it fixes up the stack by calling the standard C function 'longjmp', discarding all the old activation records, as shown in Figure 5.15. In addition the stack can also be used for first generation storage in a generational garbage collection system—objects are allocated on the stack and before the stack is fixed up the live objects are copied into the second generation space. This relies on all calls being tail calls and no call may return since it might pop objects off the stack. The Pushy scheme is therefore well suited to use with CPS since in CPS all calls are tailcalls and no call ever returns.

A recent study of the performance of a number of tailcall schemes on various architectures [Pet95] and found that the performance of the Pushy scheme is competitive with other techniques on some architectures and quite bad on others. The reasons for its bad performance appear on other to be:

- The Pushy scheme interacts badly with *Register Windows* on the SPARC [SPA92] and causes many register window flushes which are quite expensive and require intervention from the operating system. This problem can be solved with a small amount of inline assembler similar to that shown on Page 72.

- Many modern architectures (e.g., SPARC, Alpha, Power) allocate large stack frames for function calls. This results in frequent stack fix-ups or poor cache performance.

UUO handler

```
int arg1, arg2, ...;

fptr j()
{
    int x = arg1;
    ...
    arg1 = x + 1;
    return k;
}

fptr k()
{
    int y = arg1;
    ...
    arg2 = y - 1;
    return m;
}
```

Figure 5.16: Using a UUO handler. Arguments are passed in global variables.

Another solution is to use a *UUO handler*, which was invented by Steele for the Rabbit Scheme compiler [Ste78], which compiles Scheme into MacLisp, though this technique has also been used in compilers that emit C [TAL91, Jon92]. A UUO handler is a loop of the form.

```
fptr next;
...
while(next = next());
```

Where 'next' is a pointer to a function which returns a pointer to a function ... Each function in the program then returns the address of the function to call next to the UUO handler. E.g., Figure 5.16 shows the functions in Figure 5.14

rewritten to use a UUO handler, 'j' returns a pointer to 'k' to the UUO handler, the UUO handler then calls 'k' which returns 'm' and so on ...

The UUO handler clearly does tailcalls with no net stack growth—the stack frame of each function is removed when the function returns—but:

1. It is slow. Returning from a function is approximately as costly as calling a function, therefore a tailcall using a UUO handler is approximately as expensive as two function calls.

2. Since there is no way for a C function to return more than one value, arguments to functions can't be passed in registers across tailcalls. The usual way around this is to store the arguments on a global variables across the call. This makes tailcalls even slower.

Big Switch

Another alternative is to compile the entire program as a single **switch** statement, as shown in Figure 5.17. Each source function becomes a **case** in the **switch** and is given a unique integer is its label. A function is called by storing the number of the function in 'next' and falling out of the switch. Arguments are passed in local variables which, hopefully, are mapped onto machine registers.

This scheme is quite straightforward but it has two disadvantages:

1. Calls are indirect—a jump to the beginning of the **switch** and then, depending on how well the compiler compiled the switch, an indirect jump to the body of the function. However, this problem can be eliminated using GNU CC's "Labels as Values" extension [Pet95, Sta92].

2. More seriously, the entire program in the source languages is transformed into a single *very large* C function which would take a long time to compile, especially with optimization.[12]

   This problem can be partially overcome by compiling each source module (or file) into a separate C function and using the **switch** for intra-module calls and a modified UUO handler for intermodule calls. Pettersson [Pet95]

[12]Many optimizations are quadratic (or worse) in the length of the function.

```
int main()
{
    int arg1, arg2, ...;
    int next;
    ...
    while(1)
        switch(next)
        {

        case 42:            /* j(int x) */
        {
            int x = arg1;
            ...
            arg1 = x + 1;
            next = 43;
            break;
        }
        case 43:            /* k(int y) */
        {
            int y = arg1;
            ...
            arg1 = y - 1;
            next = 44;
            break;
        }

        }
}
```

Figure 5.17: The Big function approach.

found that this technique has very good performance since most calls are intramodule calls, but unfortunately it is too coarse grained for SKI which must be able to recompile and replace individual functions. The best we could do in SKI would be to map each escaping SKI-CPS function onto a C function with a switch. This would degenerate into the UUO handler scheme.

Inline Assembler

```
#define TAIL_CALL2(dest, a0, a1)
{
        register SKI __arg0 asm ("%i0");
        register SKI __arg1 asm ("%i1");
        __arg0 = (SKI) a0;
        __arg1 = (SKI) a1;
        asm volatile (
       "jmpl %0, %%g0, %%g0
        restore"
        :
        : "r" (dest), "r" (__arg0), "r" (__arg1)
        : "%i7");

        DO_NOTHING();
}
```

Figure 5.18: A two argument tailcall macro for the SPARC.

The final option is to replace the standard C calling sequence with one of our own which is tail recursive. GNU CC, and many other C compilers, allow the programmer to embed assembly language statements into C functions. We can make use of this facility to define our own tailcall sequence. Note that it is only necessary to replace the calling sequence, we can still make use of the standard function entry sequence[13] with a few restrictions. Figure 5.18 shows a two argument tailcall macro for the SPARC, the macro makes use of several non-standard features of GNU C and is explained in full in Appendix A.3.

[13]Also known as the function prologue.

The inline assembler solution can be as fast as the standard C calling sequence, on the SPARC it uses the same sequence of instructions as a call to a function pointer, and coincidently the same sequence as a return from a function.[14]

The disadvantages of this scheme are twofold:

1. It is non-portable. That said, the amount of code that must be changed is very small, but changing it does require knowledge of the architecture and calling sequence. Appendix A.4 shows a tailcall macro for Intel 386 machines running Linux so that the reader can compare it with the macro for the SPARC in Appendix A.3.

2. If it is necessary to pass more arguments than there are registers in the standard calling sequence then the extra arguments must be passed in global variables. In a "normal" call the extra arguments would be pushed onto the stack and then popped off the stack when the call returns by the caller, but since tailcalls never return this is impossible. This isn't a major handicap on modern architectures which have many registers, the SPARC for instance, uses 6 registers for passing arguments in the standard calling sequence, but on older register poor architectures like the Intel 386 this may lead to a serious drop in performance, the standard calling sequence uses no registers on 386s running Linux 1.0.

   Storing arguments in global variables is slightly slower than storing them on the stack. With RISC architectures like the SPARC the extra cost is in loading the address of the global(s), Pettersson [Pet95] discusses this in more detail.

[14]Since calling a continuation is a tail call, this is a good thing!

5.5 Generating C

Generating C code from SKI-CPS is quite straight forward since each SKI-CPS construct can be relatively easily transformed into a C construct.

The first step in generating C is to isolate all **nlambdas** and *escaping* **clambdas** (see Section 5.3) since each will become a separate C function. The lambdas are replaced by unique labels which become the names of the functions. At the same time we isolate all literals of aggregate types and replace them with references to special global variables which will be initialised to hold the values of the literals. E.g., for **nlambdas**

```
(jmp (nlambda ...)            ⇒      (jmp (label nlambda_x)
  ...)                                  ...),
```

for escaping **clambdas**

```
(app f (...)                  ⇒      (app f (...)
  (clambda ...))                        (label clambda_y))

(cif t (clambda (k1) ...)     ⇒      (cif t (clambda (k1) ...)
       (clambda (k2) ...)                   (clambda (k2) ...)
     (clambda ...))                         (label clambda_z)),
```

and for aggregate immediates and symbols,

```
(jmp ⟨imm⟩ ...)          ⇒          (jmp (prim $get-global q) ...).
```

Where ‘nlambda_x’, ‘clambda_y’ and ‘clambda_z’ are the labels generated for the lambdas, and q is the offset of the slot in the global table that the immediate will be placed in.

Having extracted the functions, we can now generate the code for each. For notational convenience we express code generation as a function $\mathcal{G}$ which takes a single argument, a SKI-CPS form, and yields the C equivalent. The transformation for **clambdas** is quite simple:

```
  G[(clambda (c1 ... cm r) ⟨body⟩)]
⇒ SKI_DEFUNn(clambda_x, SKI c1, ..., SKI cm, SKI r)
    G[⟨body⟩]
  SKI_ENDFUN(clambda_x)
```

‘SKI’ is the C type for all the Scheme objects. ‘SKI_DEFUNn’ is a C macro of n arguments which hides the difference between the C calling sequences on different architectures. Remember that the assembly language tailcall (Section 5.4.1) may have to pass some arguments in registers and some in globals. For example on a architecture with two argument passing registers

```
    SKI_DEFUN4(clambda_42, SKI a, SKI b, SKI c, SKI d)
```

expands to

```
    void clambda_42(SKI a, SKI b)
    {
      SKI c = GlobalArgReg[0];
      SKI d = GlobalArgReg[1];
      {
```

'SKI_ENDFUN' closes the braces opened by 'SKI_DEFUN*n*'.

The transformation for **nlambdas** is somewhat harder because **nlambdas** have a rather complex calling convention. This calling convention is necessary for the following reasons:

- The calling site has no knowledge of the number of arguments that the **nlambda** takes.
- **nlambdas** can take a variable number of arguments.
- The number of arguments passed to the **nlambda** has to be checked against then number expected, and an exception raised if the number is wrong.
- There is a fixed number of registers (including those simulated with globals) for passing arguments in.

Table 5.1 shows the calling convention for a procedure of k arguments when there are n argument passing registers, including those simulated by global variables. The calling convention is quite complex and is designed to be simple and fast for the common case when there are a small number of fixed arguments and no rest argument. In the more complex cases when there are a large number of fixed arguments and/or a rest argument then the code generated for the **nlambda** gets quite complicated.

In the simple case when the **nlambda** takes a fixed number of arguments j and $j \leq r$, then the code generated is also simple:

$\mathcal{G}[\![(\textbf{nlambda}\ \textit{cont}\ \textit{clo}\ (c_1\ \ldots\ c_m\ a_1\ \ldots\ a_j)\ \#\text{f}\ \langle \textit{body} \rangle)]\!]$

Register Number(s)	Use
$0 \ldots m-1$	The callee-save variables. m is the number of callee-save variables.
m	The continuation variable.
$m+1$	The closure variable.
$m+2$	The number of arguments, an unsigned integer.
$m+3 \ldots n-1$	The arguments. If the number of arguments, k, is less than or equal to the number of registers remaining, $r = n-(m+3)$, then all the arguments are passed in registers. If $k \geq r$ then the first $r-1$ arguments are passed in registers and the last register holds a list containing the rest of the arguments.

Table 5.1: The calling convention for n argument passing registers (real and simulated).

$\Rightarrow$ SKI_DEFUNb(nlambda_x, SKI c_1, ..., SKI c_m, SKI *cont*, SKI *clo*,
unsigned *nargs*, SKI a_1, ..., SKI a_j)
    CHECK_NARGS_FIXED(*nargs*, j, nlambda_x);
    $\mathcal{G}[\langle \mathit{body} \rangle]$
  SKI_ENDFUN(nlambda_x)

'CHECK_NARGS_FIXED' checks that the number of arguments passed, *nargs*, is the equal to the number expected, j, and if the check fails raises an exception.[15]

If the **nlambda** takes a more than r fixed arguments then code is generated to extract the rest of the arguments from the list passed in the last argument register. For example, if $m = 3$ and $n = 8$ then for:

```
(nlambda k clo (cs1 cs2 cs3 a b c d) #f
  ...)
```

the last three arguments will be passed in the list and code will be generated to extract them.

[15]The exception is raised by calling a special continuation.

```
SKI_DEFUN8(nlambda_42, SKI cs1, SKI cs2, SKI cs3, SKI k, SKI clo,
              unsigned nargs, SKI a, SKI lis)
  SKI b, c, d;
  CHECK_NARGS_FIXED(nargs, 4, nlambda_42);
  b = SKI_CAR(lis); lis = SKI_CDR(lis);
  c = SKI_CAR(lis); lis = SKI_CDR(lis);
  d = SKI_CAR(lis);
  ...
SKI_ENDFUN(nlambda_42)
```

'SKI_CAR' and 'SKI_CDR' are C macros which perform the 'car' and 'cdr' primitive operations.

If the number of fixed arguments exceeds the number of argument passing registers and the **nlambda** takes a rest argument, then the rest argument is set to the tail of the list after the fixed arguments have been removed. E.g. if we modify the previous example so that it takes a rest argument 'r':

```
(nlambda k clo (cs1 cs2 cs3 a b c d) r
  ...)
```

the code generated is:

```
SKI_DEFUN8(nlambda_43, SKI cs1, SKI cs2, SKI cs3, SKI k, SKI clo,
              unsigned nargs, SKI a, SKI lis)
  SKI b, c, d, r;
  CHECK_NARGS_REST(nargs, 4, nlambda_43);
  b = SKI_CAR(lis); lis = SKI_CDR(lis);
  c = SKI_CAR(lis); lis = SKI_CDR(lis);
  d = SKI_CAR(lis); lis = SKI_CDR(lis);
  r = lis;
  ...
SKI_ENDFUN(nlambda_43)
```

'CHECK_NARGS_REST' is similar to 'CHECK_NARGS_FIXED' except that it checks that at least the expected number of fixed arguments was passed.

When there are fewer r fixed arguments and a rest argument some of the arguments passed in registers should be in the rest list, so code is generated to add them onto the list. E.g., the code generated for:

```
(nlambda k clo (cs1 cs2 cs3) r
  ...)
```

which has no fixed arguments apart from the callee-save variables is:

```
SKI_DEFUN8(nlambda_44, SKI cs1, SKI cs2, SKI cs3, SKI k, SKI clo,
           unsigned nargs, SKI tt1, SKI tt2)
  SKI r;
  CHECK_NARGS_REST(nargs, 0, nlambda_44);
  r = SKI_NULL;
  switch(nargs)
  {
  case 2:
    tt2 = SKI_CONS(tt2, SKI_NULL);
  default:
    r = tt2;
  case 1:
    r = SKI_CONS(tt1, r);
  case 0: ;
  }
  ...
SKI_ENDFUN(nlambda_44)
```

'SKI_CONS' is a macro which implements the 'cons' primitive.

Finally, when there are exactly r fixed arguments and a rest argument code is generated to distinguish between the case when there are zero rest arguments and the last register holds the final fixed argument, and the case when there are one or more rest arguments.

Now that code has been generated for the **nlambdas** and escaping **clambdas**, we are in a position to generate code for the bodies of the lambdas. The code generated of a **jmp** depends on the continuation. If the continuation is a **clambda** then we generate a declaration for the **clambdas** argument and assign the value of the $\langle carg \rangle$ to it:

$$\mathcal{G}[(\textbf{jmp}\ \langle carg\rangle\ (\textbf{clambda}\ (r)\ \ldots))]$$
$$\Rightarrow \{\ \text{SCM}\ r = \mathcal{G}[\langle carg\rangle];$$
$$\ldots$$
$$\}$$

If the continuations argument is never used then it is not necessary to declare it:

$\mathcal{G}$[(**jmp** $\langle carg \rangle$ (**clambda** (($\langle unused-var \rangle$) $\ldots$))]
$\Rightarrow$ $\mathcal{G}[\langle carg \rangle]$;

If the continuation is a continuation variable then we generate a call to the continuation passing the continuation's callee-save variables and the value of the **jmp**:

$\mathcal{G}$[(**jmp** ($cs_1 \ldots cs_m$ $\langle carg \rangle$) k)]
$\Rightarrow$ TAIL_CALLb(k, cs_1, $\ldots$, cs_m, $\mathcal{G}[\langle carg \rangle]$);

The transformation for **app** expressions is similar:

$\mathcal{G}$[(**app** $\langle var_t \rangle$ ($cs_1 \ldots cs_m$ $\langle arg_1 \rangle \ldots \langle arg_j \rangle$) $\langle cont \rangle$)]
$\Rightarrow$ TAIL_CALL_UNKNOWNb($\mathcal{G}[\langle var_t \rangle]$, cs_1, $\ldots$, cs_m, $\mathcal{G}[\langle cont \rangle]$, $\mathcal{G}[\langle var_t \rangle]$,
j, $\mathcal{G}[\langle arg_1 \rangle]$, $\ldots$, $\mathcal{G}[\langle arg_j \rangle]$);

The variable $\langle var_t \rangle$ holds the closure of the target procedure and the macro 'TAIL_CALL_UNKNOWN' checks that its first argument is a closure and calls the procedure in the first slot of the closure.

The only complication arises when the number of arguments, j, exceeds the number of registers available, r. In this case a list is constructed with the last $j - r + 1$ arguments in it and placed in the last argument passing register. E.g., if we call procedure 'f' with four arguments 'a', 'b', 'c' and 'd' then the following code will be generated.

```
{ SKI tt = SKI_NULL;
  tt = SKI_CONS(d, tt);
  tt = SKI_CONS(c, tt);
  tt = SKI_CONS(b, tt);
  TAIL_CALL_UNKNOWN8(f, cs1, cs2, cs3, k, f, 4, a, tt);
}
```

Conditional expressions, **cifs**, are transformed into **if** statements:

```
(cif ⟨var⟩ (clambda (k_1) ⟨body_1⟩)
           (clambda (k_2) ⟨body_2⟩)
  ⟨cont⟩)

⇒ if(𝒢[⟨var⟩] ≠ SKI_FALSE) {
    SKI k_1 = 𝒢[⟨cont⟩];
    𝒢[⟨body_1⟩]
  } else {
    SKI k_2 = 𝒢[⟨cont⟩];
    𝒢[⟨body_2⟩]
  }
```

'SKI_FALSE' is the false value '#f'.

If the continuation of the **cif** is a **clambda** and it doesn't escape then we can generate more efficient code which no tail calls. E.g., the code generated for:

```
(cif x (clambda (k1) (jmp x k1))
       (clambda (k2) (jmp y k1))
     (clambda (r) ...)),
```

which binds 'r' to the value of 'x' if 'x' is true and 'y' otherwise, is:

```
{ SKI r;
  if(x ≠ SKI_FALSE) {
    r = x;
  } else {
    r = y;
  }
  ...
}
```

The transformation for the continuations of **cifs** and **apps**, which can be either continuation variables or **labels**, is:

$\mathcal{G}[k] \Rightarrow k$

$\mathcal{G}[(\textbf{label}\ l)] \Rightarrow (\text{SKI})\ l$

The cast, '(SKI) l', is necessary because the label, l, is a C function pointer. This relies on the fact that function pointers are 4 byte aligned, and therefore appear to be fixnums to the runtime system.[16]

Finally, the transformations for $\langle carg \rangle$s and $\langle arg \rangle$s, which can be calls to primitives, labels, literals and variables, are:

$$\begin{array}{lcl}
\mathcal{G}[(\textbf{prim}\ name\ \langle arg_1 \rangle\ \ldots\ \langle arg_j \rangle)] & \Rightarrow & c\text{-}name(\mathcal{G}[\langle arg_1 \rangle], \ldots, \mathcal{G}[\langle arg_j \rangle]) \\
\mathcal{G}[(\textbf{label}\ l)] & \Rightarrow & (\text{SKI})\ l \\
\mathcal{G}[\langle literal \rangle] & \Rightarrow & makelit(\langle literal \rangle) \\
\mathcal{G}[v] & \Rightarrow & v
\end{array}$$

c-name is the C name for the primitive, e.g., '\$cons' $\Rightarrow$ 'SKI_CONS', '\$car' $\Rightarrow$ 'SKI_CAR', etc., *makelit* generates an expression which makes a value with the value of the immediate, e.g., *makelit*(3) $\Rightarrow$ INT2SKI(3),[17] *makelit*(#f) $\Rightarrow$ SKI_FALSE, etc.

[16]Fixnums have a tag of 00 in the least significant two bits. The least significant two bits of a 4 byte aligned pointer will also be 00, therefore function pointers will appear to be fixnums to the runtime system. See Section 5.6.

[17]'INT2SKI(3)' makes a fixnum with the value 3.

5.6 Run-Time Type System

Language implementations need a way of representing their basic, built-in, types. Some, like C [KR78, KR88], map their basic types onto those data types provided by the machine, so that very little effort is required to support them. Others have more complex basic types requiring more support. Strongly-typed dynamic languages such as Scheme have a large number of built-in types which require that instances of types are tagged to distinguish them from one another. Some statically typed languages which have garbage collection, such as SML [App92], tag objects so that the garbage collector can know their types.

An efficient tagging scheme is necessary since dynamic language implementations can spend a large amount of time on tag handling operations. Steenkiste [SH87, Ste91] has shown that some dialects of Lisp spend approximately one quarter of their runtime in tag handling when type checking is disabled and run 25% slower when checking is enabled.

The cost of tag handling can be reduced in two ways. Firstly, eliminate as many tag handling operations as possible. This is the focus of the optimizations presented in Chapter 6. Secondly, make tag handling as inexpensive as possible by reducing the cost of the basic tag handling operations. These operations are[18]:

- *Tagging* – Converting between the machine representation of a value and the tagged representation. This usually involves adding some tag bits to the value and is sometimes called tag insertion.

- *Un-tagging* – Converting a tagged value back into it's machine representation so that some operation can be performed on it. This is sometimes called tag removal.

- *Tag checking* – Checking that a value is a certain type.

The remainder of this section discusses the implementation of SKI's types and then compares SKI's types with the type schemes used by some other dynamic language implementations.

[18]Steenkiste identifies one other kind of tagging operation which he calls tag extraction – extracting the tag from a tagged value. Since tags are only ever extracted so that they can be checked we consider tag extraction to be part of the tag checking.

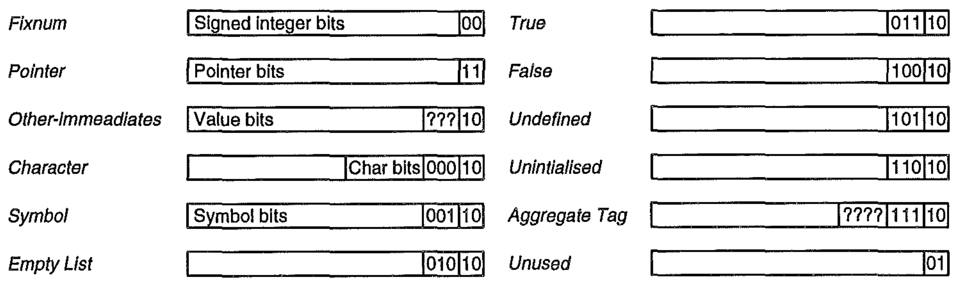


Figure 5.19: SKI's immediate types

5.6.1 SKI's types

SKI has two classes of types, *immediates* and *aggregates*. *Immediate* types are those types which hold a single value and which fit into a machine register, e.g., small integers, characters, symbols and pointers to aggregates. *Aggregate* types hold several values or are too large to fit into a register, e.g., pairs, vectors and floating point numbers. In this section we assume registers are 32 bits wide and the natural word size is 32 bits, but everything we discuss will work equally well on machines with other word sizes, especially 64-bit machines.

Immediate Types

The class of immediate types can be divided into three subclasses, *fixnums*, *pointers* and other-immediates. All immediates contain a tag field in their least significant two bits as shown in figure 5.19.

Fixnums are fixed length integers consist of the least significant 30 bits of a signed integer and are tagged with 00. Fixnums can be converted into machine integers with a single shift instruction. Tagging fixnums with 00 means conversion to machine integers is unnecessary for fixnum addition, subtraction and comparison, but some method of detecting overflow is required. Checking that a value is a fixnum requires only a single bitwise-and. For example the following sequence of SPARC [SPA92] instructions will branch to `not_fixnum` if the value in register `%l1` is not a fixnum:

```
andcc %l1, 11b, %g0 ! And %l1 with 11b setting the condition codes
                    ! (cc) and ignoring the result (%g0 is always zero).
bnz not_fixnum      ! Branch if not zero.
```

Pointers are used to reference aggregate types on the heap. They consist of the most significant 30 bits of a machine pointer and a tag field of 11. Since aggregates are word aligned (the least significant two bits of a pointer are always zero) no information is lost by placing the tags in the least significant two bits. Un-tagging a pointer requires a single bitwise-and or an addition. If an addition is used for un-tagging then it can often be combined with an offset operation. For example taking the car of the pair pointed to by the register %l2, requires an un-tagging operation, an offsetting operation and a load:

```
add %l2, -3, %l3     ! Un-tag.
add %l3, 4, %l3      ! Add the offset of the car field.
ld [%l3], %l3        ! Load the car.
```

The two additions can be combined and since the offset is a constant, the addition can be combined with the load giving:

```
ld [%l2 + 1], %l3    ! Load the car.
```

The other-immediates are types which are small enough to fit into a register, but do not need a special representation. Other-immediates all share a primary tag of 10, and include a secondary tag of three bits and an optional value. Table 5.2 lists all of the other-immediates. Some other-immediate types have many values (e.g., characters and symbols), in these cases the value is stored in the upper 27 bits of the word. Other types only have one possible value, for instance the empty list type. The boolean values true and false are also encoded as separate types. Checking the type of these values requires a bitwise-and and a comparison if the type is multi-valued or a comparison if the type has a single value. Un-tagging a multi-valued type requires a shift.

The tag 01 is never used, this allows a pointer to be checked using a single bitwise-and operation, rather than a bitwise-and operation followed by a compare.

Type	Description
Character	
Symbols	
Null	The empty list '`()`'
True	
False	
Undefined	The value returned by library procedures and special forms whose value is undefined or unspecified by the R^{4}RS [CR91] (e.g., `set!`, `for-each`), also the value that vectors are initialised to if no other value is specified.
Uninitialised	The value of an uninitialised global variable, used internally.
Aggregate tags	The tag word of an aggregate, used internally.

Table 5.2: Other immediates.

Type	Description
Cell	Cells hold the values of variables which are assigned to. See Section 4.3.
Pair	
Vector	
String	
Extern	Externs hold values which aren't SKI types.
Flonum	Flonums are SKI's real numbers, represented as a double precision float. This means flonums must be double-word aligned.
Forward	Forwarding pointers are used by the garbage collector to point to a new copy of an object.
Bignum	Bignums are infinite precision integers. They consist of a word for the size and sign of the number and zero or more unsigned words for the bits.
Closure	Closures hold a pointer to a procedure and the values of all the procedures free variables.

Table 5.3: Aggregate types.

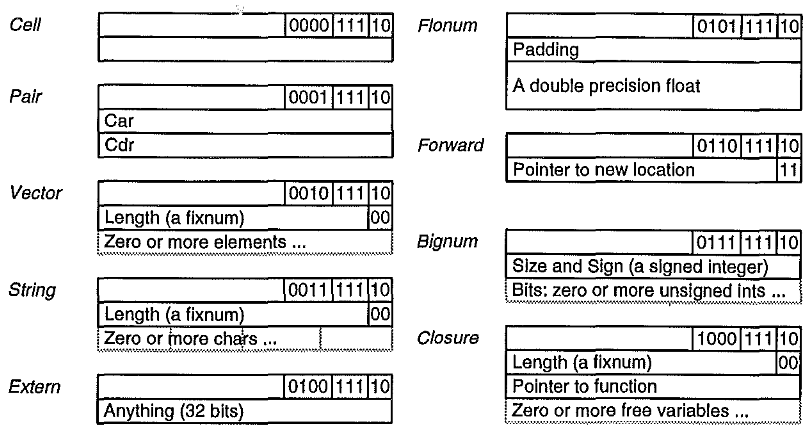


Figure 5.20: SKI's aggregate types

Aggregate Types

Aggregate types are types which have several values, or are too large to fit in a register, or must be stored in memory for some reason. Table 5.3 lists SKI's aggregate types.

All aggregate values are allocated on the heap and share the same basic format, a tag word identifing the type of the object followed by the value(s) of the object in subsequent words. Figure 5.20 shows the format of SKI's aggregates. Additionally all aggregate values must be at least two words long, so that they can be overwritten with a forward node during garbage collection. Variable length values, such as strings and vectors, are stored length first and with zero or more trailing values.

Strings have a special representation. The characters are packed four to a word to reduce the space requirements. This means the primitives that insert and retrieve characters in strings must convert between the packed representation and the immediate representation.

Bignums also have a special representation, the length word includes the sign of the bignum and is stored as a signed integer. This representation is used because bignums are implemented using the GNU MP multi-precision math library.

Aggregates don't need tagging and un-tagging as their values can be accessed directly, but they do need to have their tags checked. Tag checking an aggregate requires a pointer tag check, a load and a comparison. The comparison can be done in a single instruction, but the load is potentially expensive. Section 5.6.2 discusses alternative tagging schemes that may be more efficient in this case.

5.6.2 Other typing schemes

Virtually every different dynamic language implementation has a different run-time type system. SKI's was chosen to be simple to implement and reasonably efficient, but other implementors have chosen different tradeoffs. This section outlines the differences between SKI's run-time type system and the run-time type systems in other language implementations. We will limit the discussion to implementations for general purpose hardware. Implementations on special purpose hardware such as Lisp Machines [THL$^+$86] have a different set of requirements.

The major differences between run-time type systems are the location and number of tag bits in immediates and the types which are considered to be immediates. There are two locations where it is reasonable to store the tag bits in a word, the high (most significant) bits and the low (least significant) bits. The advantage of putting the tags in the high bits is that more bits can usually be used. The Portable Standard Lisp compiler [SH87, Ste91] and the CMU Common Lisp compiler [FM91] both use the high five bits of a word for tags. When this number of tag bits is used, all the tagging information for aggregates can be encoded in the tag of a pointer to an aggregate, making a tag word unnecessary. The benefits of this are that memory usage is reduced and that tag checking doesn't require a potentially expensive load.

One disadvantage of using the high bits for tags is that two tags must be used for fixnums, all zeros for positive fixnums and all ones of negative fixnums, so that addition and subtraction can be done without un-tagging. This means that checking that a value is a fixnum is quite an expensive operation. Another disadvantage is that all pointers have to be un-tagged before they are used, although CMU Common Lisp uses a clever trick to avoid this when running on Mach[19]. Mach can allocate segments of memory at arbitrary addresses in a processes address space, so the tag bits in the pointer can be thought of as segment

[19]Mach is a Unix-like operating system being developed at CMU.

selectors and each segment will contain only one type of object. This trick should be possible on any modern version of Unix which supports the `mmap()` system call. SKI uses a trick like this to avoid fragmentation of the C heap.

Putting the tag in the low bits has the advantages that only one tag is necessary for fixnums, and that un-tagging a pointer can often be combined with using it as we showed in Section 5.6.1.

Combining pointer use and un-tagging can give a speedup of 4% to 9% according to Steenskiste [SH87], but doing this limits the number of tag bits that can be used to two or three. If two bit tags are used then all pointers will word aligned, if three bits are used then pointer will be double-word aligned and all objects will have to start on double-word boundaries, which will sometimes result in a word being wasted between objects. If four tag bits were used, then pointers would have to be quad word aligned and several words could be wasted between objects which would be unnacceptable[20]. So most implementations use only two or three tag bits: SELF, Screme [Ple91, VP89], New Jersey SML [App92] and Scheme->C [Bar89] all use two bits. SCHEMEXEROX and Lucid Common Lisp [SH87] use three bits. If three bits are used then the tags of commonly used aggregates could be encoded in their pointers and only infrequently used aggregates would need a tag word. Scheme->C uses a restricted version of this idea, it uses two tag bits but has two types of pointers, one type for pairs and one for all other aggregates.

The other major difference between run-time type systems is what types are immediate. Most Scheme and Lisp systems have more or less the same set of immediates that SKI has, with the exception of symbols. In SKI symbols are tagged as other-immediates and the top 27 bits are occupied with the hash index value of the symbol in the global symbol table. In some other systems, Screme [Ple91, VP89] and Scheme->C for example, symbols are a special kind of aggregate. They are special in the sense that there can be only one instance of each symbol since the R^4RS [CR91] requires that two symbols that are spelt the same must be equal in the sense of `eq?`. `Eq?` is usually implemented as a word comparison so the pointers to the symbol aggregates must be the same.

[20]If most objects were pairs occupying two words each, then 50% of the occupied memory would be wasted.

SELF has a somewhat different set of immediates. SELF uses a two bit tagging scheme and tags integers and pointers like SKI. The other two tags are used for floats and marks. Floats are stored as standard IEEE 32-bit floating point numbers (see appendix A of [HP90]) except that two bits are "stolen" from the exponent field so that they can be tagged[21]. Marks, like aggregate tags in SKI, are used to mark the start of an object on the heap, but they have a unique tag because SELFS run-time system sometimes needs to scan through the heap to find all object of a certain type.

In summary, SKIS' run-time type system is reasonably efficient and is comparable with other systems. There are several ways it could be made more efficient. The most obvious inefficiency in the run-time typing scheme is that tag-checking an aggregate requires a potentially expensive memory access. The obvious solution to this is to encode more type information in the pointer. We could single out one frequently used type and use the extra immediate tag for pointers to it, but we would no longer be able to tag-check a pointer in a single instruction. By using another bit to tag immediates, we could distinguish between three to five[22] more types without having to access memory. The cost of this scheme would be about the same as that of the current one, we wouldn't be able to tag-check pointers in one instruction, but we wouldn't have to so often.

[21]The exponent field, which is in the middle of the float, is truncated and then the remainder of the exponent field and the mantissa field are shifted left to make room for the tag.

[22]Steenkiste [SH87] suggests that `000` and `100` should be used to tag fixnums, to support fast indexing into word vectors (they wouldn't need un-tagging) and that `011` and `111` be used for pointers to other aggregates. In any case we have eight tag values and we need at least one for fixnums, one for other immediates and one for other pointers, which leaves a maximum of five values.

Chapter 6

The Dynamic Optimizer

The dynamic optimizer optimizes programs while they are executing. It uses information collected by running the program to assist in the optimization and then passes the re-optimized program to the back end (see Chapter 5) for code generation and insertion into the running system.

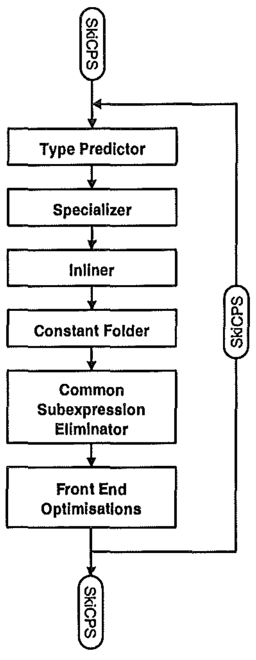


Figure 6.1: Overview of the Dynamic Optimizations

Figure 6.1 shows the basic structure of the dynamic optimizer. The three main strategies employed by the dynamic optimizer are *type prediction*, *specialization* and *inlining*:

Type prediction (Section 6.1) attempts to predict the types of variables during the execution of the program. The type information is used to eliminate type checking and to direct further optimization—effort is concentrated on the areas where the optimizer is most confident about the types since these are where the largest benefit can be expected.

Specialization (Section 6.2) produces versions of procedures in which the types of at least some of the variables are known (see Sections 2.1 and 2.3).

Inlining (Section 6.3) performs inline expansion of procedures[1] to eliminate procedure call overhead and, more importantly, to introduce further optimization opportunities.

Other optimizations performed by the dynamic optimizer are: *constant folding* (Section 6.4) which tries to remove constant expressions, including constant type expressions, and *common subexpression elimination* (Section 6.5) which attempts to eliminate common expressions. The dynamic optimizer also makes use of the optimizations we introduced in Chapter 4, especially redundant binding elimination (Section 4.6) and the conditional optimizations (Section 4.9).

The optimizations tend to reinforce each other, each creating opportunities for the others, so they are applied iteratively until there is nothing further to be gained.

6.1 Type Predictor

The type predictor attempts to predict the types that variables will have when a program is executed. To be more precise, it tries to predict the types of the values contained in the variables, since in scheme a variable is a location which may hold a value of any type.[2] The scope of the type prediction is limited to

[1]Similar to β-reduction but the size of the code may increase (see Section 4.7).

[2]In the remainder of this chapter we will talk about "the types of variables" rather than the "types of values stored in variables" since this extra indirection is confusing and verbose.

single procedures since trying to predict the types of, for instance, all the variables in an entire program is much more difficult and time consuming.

The type predictor employs two strategies, *dynamic type prediction* and *static type prediction*, to determine the types of some variables and then uses *type propagation* to propagate the type information through the procedure in an attempt to infer the types of the rest of the variables.

6.1.1 Dynamic Type Prediction

Dynamic type prediction uses the running program to gather information about the types of variables. The primary method of gathering type information is by instrumenting the program so that it records the types of variables and then letting it run for a while. The information gathered while the program was running can then be used to predict the types of the variables in the future. This method is based on the assumption that the distribution of the types of the variables in the program is relatively constant over time, although the values of the variables may vary.

Another source of dynamic type information is the values stored in the program's memory. The values of global variables, cells introduced by assignment conversion (Section 4.3) and variables stored in closures can be used as additional sources of type information. For example, all the closures for a particular procedure could be examined and type distributions determined for the variables imported by the procedure, likewise the type distribution of a global variable could be found by sampling it occasionally.

SKI performs dynamic type prediction by instrumenting each **nlambda** so that the types of its arguments recorded each time it is invoked. The types are recorded using counters, one counter for each type and one set of counters for each argument. Each counter is initialized to 1 since we can not be certain that a type will never occur. Initialising the counter to 1 ensures that every type will have a non-zero probability of occurring. When a procedure been called a number of times, we can determine $p_t(a)$, the probability that an argument a will be type t, by dividing count for that type $c_t(a)$ by the sum of the counts of all the types $C(a)$.

$$\begin{aligned} p_t(a) &= \frac{c_t(a)}{C(a)} \\ C(a) &= \sum c_i(a) \end{aligned}$$

When a procedure is optimized, the argument variables are annotated with *type sets* which contain the types that the variable may have and the probability of occurrence of each type. The type sets for the argument variables contain all possible types since every type has a probability of at least $\frac{1}{C}$.

Currently SKI doesn't obtain any type information from the program's memory, but the inliner (Section 6.3) does use the values of global variables and closure variables.

6.1.2 Static Type Prediction

Static type prediction determines the types of variables by examining the structure of the program. The sources of type information it uses are calls to primitives, literals and procedure declarations.

The set of types each primitive can return is known by the compiler. Some primitives can return only one type, for instance the '\$cons' primitive always returns a pair. Other primitives can return a small set of types; the '\$fix+' primitive, which two adds fixnums can return another fixnum or false if the addition overflows.[3] Finally, some primitives like '\$car' can return any type.

In the case where a primitive can return a small number of types, it can be useful to assign probabilities to the types returned. These probabilities can be useful for deciding what to optimize. For instance, the '\$fix+' primitive will return a fixnum most of the time and false very occasionally, so the fixnum type is given a high probability and the false type a low probability. Code which depends on the type returned by the '\$fix+' primitive can then be optimized for the common case.

[3]The true and false values are considered to be members of two distinct single-valued types, because it is easier for the type predictor to keep track of two separate types than a single type that has two values, especially since the type predictor isn't concerned with the values of other types.

The problem with using primitives as the sources of type information is that programs written by users don't contain any calls to primitives, the primitives are encapsulated inside the standard library procedures. The static type prediction therefore depends on the inliner to perform inline expansion of the standard library procedures and expose the primitives.

SKI's static type predictor annotates each variable which is bound to the result of a primitive or to a literal or procedure with a type set containing the types that the variable might have and probabilities of each type. Unlike the type sets produced by the dynamic type predictor, these type sets need not contain all possible types.

Static type prediction is a limited form of type inference [WC93]. It is limited in that it is only concerned with the types of variables, whereas type inferencers are also concerned with the types of global data structures.

6.1.3 Type Propagation

Type propagation takes the type annotations which have been attached to some variables by the type predictors and propagates them through the procedure. Every reference to one of the annotated variables is itself annotated with the variables type.

In addition, the type propagator tracks changes in a variables type set due to conditional expressions and type predicates. For example in the expression:

```
(cif x
    (clambda (k1) body1)
    (clambda (k2) body2)
  cont)
```

then it is certain that if 'x' is referenced in *body*$_1$ it cannot be false, so false is removed from the type set given to references to 'x' in *body*$_1$. Conversely, we can be certain that 'x' is false in *body*$_2$ and all references to 'x' within can be given a type set which contains only false. This is similar to the boolean value propagation optimization mentioned in section 4.9.

It is possible to extend this idea further to conditional expressions which branch on the result of a type predicate primitive. For example in the expression:

```
(jmp (prim $fixnum? y)
  (clambda (t)
    (cif t
        (clambda (k3) body3)
        (clambda (k4) body4)
      cont)))
```

the type predicate, '$fixnum?', guarantees that within *body*$_3$ 'y' is a fixnum, so all references to 'y' in *body*$_3$ are annotated with a type set which consists solely of fixnum and the type sets given to references to 'y' within *body*$_4$ do not contain fixnum.

It is also possible to calculate the probability of a type predicate returning true or false. If a type predicate tests for a type t, and the probability that its argument x is of type t is $p_t(x)$ then the probability that it returns true is $p_t(x)$ and the probability the predicate returns false is $1 - p_t(x)$. When the type of x is unknown is it is assumed that the predicate will return true or false with equal probability.

6.2 Specializer

The general idea behind specialization is to make a special copy of a block of code in which some parameters which were variable in the original are held constant. At runtime check is then used to decide whether to use the original block or the specialized block. For specialization to be successful, the parameters which are held constant must be selected so that they represent the common case, so that the cost of specialization (the runtime check and the decrease in code density) is outweighed by decrease in the runtime. To do this the specialization must increase the number of opportunities for optimization in the specialized block.

SKI uses specialization to remove the uncertainty of type prediction, and specializes on the types of arguments to **nlambdas**. It does this by selecting an argument or combination of arguments whose types are particularly predictable, making a second copy of the body of the procedure and generating a conditional with type tests to distinguish between the bodies. Running the type predictors and the type propagator over the procedure will then annotate the specialized version with the restricted type sets. E.g., if we decide to specialize the procedure:

```
(nlambda k (a b c) #f
  body)
```

for the case when 'a' is a fixnum, we get:

```
(nlambda k (a b c) #f
  (jmp (prim $fixnum? a)
    (clambda (t)
      (cif t
          (clambda (k1)
            body_s)
          (clambda (k2)
            body_g)
      k)))).
```

If we run the type predictors and the type propagator over the procedure, it will discover that in specialized copy of the body, *body*$_s$, 'a' is must be a fixnum and, as a side effect, 'a' can not be a fixnum in the unspecialized or *general* copy of the body, *body*$_g$.

When a procedure is specialized on the types of two or more arguments then a special primitive '$type-test' is used to test the types, '(**prim** $type-test \$*pred*$_1$ *x* \$*pred*$_2$ *y*)' yields true iff '(**prim** \$*pred*$_1$ *x*)' and '(**prim** \$*pred*$_2$ *y*)' both yield true. We use '$type-test' because we can generate more efficient code for it than we can generate for two or more nested **cifs**.[4]

6.2.1 When to specialize?

6.3 β-expander or Inliner

β-expansion performs essentially the same transformation as β-reduction (see Section 4.7). The differences between the two are:

[4]The argument to '$type-test' are actually written differently, rather than put the names of type predicates in the list, which would be illegal (see the grammar on page 32), we encode the names as numbers. E.g., '(**prim** $type-test $fixnum? x $null? y)' is really encoded as '(**prim** $type-test 0 x 4 y)' and the names are decoded when code is generated for the **prim**.

- β-reduction performs inline expansion of procedures which are called once and never escape from the enclosing scope. β-expansion can inline any procedure.
- The site where a procedure is expanded by β-reduction is in the same scope as the declaration of the procedure. This is not necessarily true for β-expansion. The β-expander may have to retrieve the values of variables imported by a procedure it expands from the closure of the procedure.
- When a procedure is β-reduced the program is guaranteed to be smaller and faster. No such guarantees can be made for β-expansion. The program will always be larger after a β-expansion, but it should be faster, though excessive β-expansion might increase the size of the program so much that it becomes slower due to increased paging or bad cache utilisation.

The β-expansion transformation is otherwise the same as the β-reduction transformation and the details of the transformation will not be repeated here (see Section 4.7).[5]

When a procedure is β-expanded in a scope other than the one it was declared in, the values of the variables that it imports must be determined. This can be accomplished by locating the closure belonging to the instance of the procedure to be expanded in the programs heap and recovering the values from it. We call this *closure inlining*. If the value of a variable in the closure is of an immediate type and not a symbol then we convert it into a literal and generate code that binds it to the variable. If the value is a aggregate or a symbol then we put it in a global variable and generate code to retrieve it from the table. For example if the closure contains the values of two variables, 'a' and 'b', and the value of 'a' is a string and 'b' is the fixnum 12 then the code generated to inline the closure is:

```
(jmp (prim $get-global n)
  (clambda (a)
    (jmp 12
      (clambda (b)
        body)))).
```

[5]In fact, SKI uses the same code for β-expansion, β-reduction and η-reduction.

Here n is offset of the variable holding the string in the global table and *body* is the body of the procedure.

Closure inlining is possible for two reasons.

1. Closures are immutable and once created they cannot be modified. This means that it is possible to include a value from a closure in a program, since it can't be changed.

2. The value of an aggregate in a closure is actually a reference to the real value in memory and references can be copied freely without copying the real value.

6.3.1 Deciding what β-expand

The mechanics of β-expansion are quite simple. The difficulty is in deciding which procedures to β-expand and when to stop β-expansion. The benefits of β-expansion are twofold.

1. The overhead of the procedure call is eliminated.

2. The expansion "uncovers" information which can be used for further optimization.

The cost of β-expansion is the increase in code size, which may increase paging and decrease cache locality. In an extreme case it could lead to the program running out of memory. Obviously we should β-expand those procedures which are most likely to benefit, but have the least impact on the cost.

The first benefit is essentially a small constant speedup—if a procedure call takes x nano-seconds, then inlining a procedure will make it x ns faster every time it is executed. This leads us to two conclusions.

1. Since the benefit is a small constant, inlining will benefit small procedures proportionally more than larger ones. Inlining small procedures will also cost less (in some cases may it cost nothing[6]) since the code generated for a small procedure could be smaller than the code generated for the procedure call.

[6]Or less than nothing!

2. Since the benefit is small, it must be allowed to accumulate to produce a noticeable speedup. To do this, we can either expand a large number of procedure calls, or expand those procedure calls which are most frequent. Clearly the latter is better than the former, since expanding a large number of calls would be expensive. Conversely, infrequently executed procedure calls should not be expanded since the costs would outweigh the benefits.

The second benefit is harder to quantify. In some cases a β-expansion will generate an opportunity for a very profitable optimization or sequence of optimizations. An obvious example is expansion of a call to a function with constant arguments which is reduced to another constant, e.g., '(* 3 5)' could be reduced to '15'. In other cases β-expansion may lead to no further optimizations.

The best that we can hope to do is to expand calls which appear to offer opportunities for further optimization. As a heuristic, procedures with many calls to primitives are often good candidates for expansion since type prediction and optimisations like constant folding and common subexpression elimination operate on primitives (see Sections 6.1, 6.4 and 6.5). Call sites where the type information on the arguments is good or some of the arguments constant also make good candidates.

One solution is to have the compiler "learn" which procedures and call sites β-expand well. Dean and Chambers [DC93] describe a modification to the SELF compiler which records details of each β-expansion in a database. The details include the name of the procedure, information about the types and values of the arguments, and a metric indicating how successful the β-reduction was. Using this information the compiler can deduce rules like: "Procedure 'f' β-expands well when its third argument is an integer" and use these rules to guide further expansion.

Currently SKI has no rules for deciding what to inline, but instead it interactively asks the user whether it should inline a procedure call.

6.4 Constant Folding

Constants folding is a essentially simple form of partial evaluation [JGS93] which evaluates simple expressions that are constant at compile time and replaces them with constants. In languages like C and Pascal, constant expressions are usually expressions involving arithmetic operators with constant operands, e.g., a C compiler might replace the expression '$1+2$' with the constant 3 (see Section 10.2 of *The Dragon Book* [ASU86]).

In SKI constant expressions take the form of calls to primitives where the arguments are constants or, if the primitive is a type predicate, where the type of the arguments is known. Constants in SKI-CPS are either immediates or variables which are bound to immediates. For example, the **prim** expressions in both

```
(jmp (prim $fix+ 2 3)
  ...)
```

and[7]

```
(jmp 3.14
  (clambda (a)
    (jmp (prim $flo-negate a)
      ...)))
```

are constant expressions which can be eliminated giving

```
(jmp 5
  ...)
```

and

```
(jmp 3.14
  (clambda (a)
    (jmp −3.14
      ...)))
```

respectively. These expressions can then be further simplified by using the redundant binding eliminator (see Section 4.6). In particular if 'a' is never referenced again then its binding will be eliminated.

When a primitive is a type predicate and the type of its argument is known then we may be able to eliminate the predicate according to the following rules.

- If the predicate is true for all types in the type set then it can be replaced with true.
- If the predicate is false for all types in the type set then it can be replaced

[7]'$Flo-negate' negates a flonum

with false.

- Otherwise the predicate remains unchanged.

For example, if the type set of 'i' is {fixnum} in

```
(jmp (prim $fixnum? i)
  (clambda (t)
    (cif t branch_t branch_f cont))),
```

then the predicate (**prim** \$fixnum? i) can be replaced with '#t'. The redundant binding eliminator can then eliminate the binding of 't' giving

(**cif** #t $branch_t$ $branch_f$ *cont*),

and the conditional optimiser (see Section 4.9) can eliminate the **cif** and the unreachable branch $branch_f$.

6.5 Common Subexpression Eliminator

Common subexpression elimination (or CSE) identifies expressions which are identical and computed more than once, it then replaces the duplicate expressions with a variable containing the result of evaluating the first expression. E.g., in the following C fragment:

```
x1 = x + i + 1;
y1 = y + i + 1;
```

the expression 'i + 1' is duplicated and can be eliminated by assigning 'i + 1' to a temporary, 't', and substituting 't' for each occurrence of 'i + 1':

```
t = i + 1;
x1 = x + t;
y1 = y + t;
```

Like constant folding, CSE in SKI operates on primitives. The common subexpression eliminator traverses each **nlambda** procedure and when it encounters a call to a primitive, m:

(**jmp** (**prim** p_m a_{m_1} a_{m_2} $\ldots$ a_{m_k})
  (**clambda** (v_m) ...))

it stores the name and arguments in a table along with the variable that the result will be bound to v_m. If it later encounters a primitive call, n:

(jmp (prim p_n a_{n_1} a_{n_2} $\ldots$ a_{n_k})
  (clambda (v_n) ...))

that is identical to m, i.e., $p_n = p_m$ and $a_{n_i} = a_{m_i}$, then it replaces n with a reference to v_m:

(jmp v_m
  (clambda (v_n) ...)).

Not all primitives can be eliminated. Primitives that rely on, or side-effect, global state can't be eliminated, e.g., '$read-char',[8] '$set-global!', etc. Others which rely on the values of mutable data structures in memory require special precautions. For instance, repeated calls to '$vector-ref' with the same arguments can be eliminated iff there are no intervening calls to '$vector-set!'.[9] If there is a single call to '$vector-set!' between the two identical calls to '$vector-ref', even if it appears to mutate a different vector, then the second '$vector-ref' cannot be eliminated. This last restriction, is a result of *alias problem* which is illustrated in Figure 6.2.

```
(jmp (prim $vector-ref a 10)          ; 1
  (clambda (s)
    (jmp (prim $vector-set! b 10 w) ; 2
      (clambda (ign)
        (jmp (prim $vector-ref a 10) ; 3
          (clambda (t)
            ...))))))
```

Figure 6.2: An example of the alias problem.

If 'a' and 'b' are references to distinct vectors then it is possible to eliminate the the second '$vector-ref' replacing it with a reference to 's'. If, however 'a' and 'b' refer to the same vector, that is they are *aliases* for each other, then the meaning of the program would be changed. So unless it can be proved that 'a' can never alias 'b' it is illegal to eliminate the second '$vector-ref'.[10] In general

[8]'$read-char' gets the next char form and input port, like 'getc' in C.

[9](**prim** $vector-ref v i) returns the value of element 'i' of vector 'v'. (**prim** $vector-set! v i x) sets element 'i' of 'v' to the value of 'x'.

[10]The alias problem occurs with any mutable aggregate type, we use vectors purely as an example.

it is difficult to decide whether two variables are aliases for each other so SKI conservatively assumes that all references of the same type are aliases.[11] See Chapter 10 of *The Dragon Book* [ASU86] for a more complete discussion of the alias problem.

6.6 Other optimizations

In this sections we briefly discuss a number of optimizations which have not been implemented in the current version of SKI, but which could be included in a future version.

6.6.1 Known Procedure Calls, η-splitting and Once-Cell Elimination

If the target of a procedure call is known at compile time then it is possible to optimize the procedure calling sequence in a number of ways:

- The address of the procedure can be included in the generated code as a constant instead of being fetched from the closure.
- Since the target is known to be a procedure, it is not necessary to check it at runtime (see Section 5.5).
- If all the calls to a procedure are known calls then we can eliminate the runtime check on the number of arguments (see Section 5.5).
- If the target procedure is in the same scope as the calling procedure and does not escape, then the closure can be eliminated and the values of the variables the procedure imports can be passed to it as arguments. This can get quite complex in the presence of recursive procedures. For example, if the set of variables imported by procedure a is $I(a)$ and procedure b imports $I(b)$. Then if a calls b, $I(a)$ must be augmented to include the variables in $I(b)$, and if b calls a then $I(b)$ must be augmented to include

[11]In some languages, C for instance, it is impossible to assume even this and compilers must assume that all pointers are aliases.

$I(a)$. Fortunately it is possible to find the fixed points of these sets by iteratively computing

$$\begin{aligned} I_{n+1}(a) &= I_n(a) \cup I_n(b) \\ I_{n+1}(b) &= I_n(b) \cup I_n(a) \end{aligned}$$

until $I_{n+1}(a) = I_n(a)$ and $I_{n+1}(b) = I_n(b)$. In this limited case the equations will converge on their fixed points in a single iteration, but in the general case when there are multiple recursive it may require a number of iterations.

η-splitting is the opposite of η-reduction (see section 4.8). η-reduction eliminates procedures which do nothing but call other procedures, η-splitting introduces them! E.g., the η-splitting transform is (in Scheme):

```
(let ([f (lambda (a) ...)])      ⇒      (let ([f (lambda (a) ...)])
  ...)                                     (let ([f1 (lambda (b) (f a))])
                                             ...)).
```

If we then replace all escaping references to 'f' with 'f1', then the first procedure, 'f', becomes a non-escaping known procedure which doesn't need a closure. The introduced procedure 'f1' becomes a stub which retrieves the variables imported by 'f' from its, 'f1's, closure and passes them to 'f'. η-splitting is especially useful for recursive procedures like 'fibonacci' in Figure 6.3a. Both the internal recursive calls to 'fib' could be transformed into known procedure calls and no closures would be needed except for the escaping version of the procedure.

Unfortunately, the cells introduced by assignment conversion will obstruct this kind of optimization (see Section 4.3). Recall that the assignment conversion pass introduces once-cells to hold the variables bound by a **letrec**, see Figure 6.3b.

However, it is possible to eliminate the once-cells. If a once-cell is set to a constant value or to a variable bound to a constant then the once-cell can be eliminated and all references to its value can be replaced by that constant, if the constant is a procedure then we can substitute a reference to the procedure. E.g., we can replace (**prim** $get-once-cell fib-cell) with a reference to the procedure bound to 'fib'. Calls to this reference then become known calls.

An earlier version of SKI, performed known call optimization, η-splitting and once-cell elimination, but the current version does not.

```
(define fibonacci
  (letrec ([fib
            (lambda (n)
              (if (<= n 1)
                  1
                  (+ (fib (- n 1))
                     (fib (- n 2)))))])
    fib))
```

(a) Fibbonacci.

```
(define fibonacci
  (let ([fib-cell (prim $make-once-cell)])
    (let ([fib
           (lambda (n)
             ...
             (+ ...
                ((prim $get-once-cell fib-cell) (- n 2))))])
      (prim $set-once-cell fib-cell fib)
      (prim $get-once-cell fib-cell))))
```

(b) Fibonacci with cells

Figure 6.3:

A more detailed explanation of known call optimization and η-splitting can be found in [App92]. As far as we are aware, once-cells and once-cell elimination are not used in any other compiler. Though once-cell elimination is similar to the "letrecification" transformation used in the SchemeXerox compiler [ACS93].

6.6.2 Inlining Rest Lists

If a procedure that has a rest argument is inlined either by β-reduction or β-expansion (Sections 4.7 and 6.3) then the rest list for the procedure is constructed and bound to a variable. The inlined body of the procedure can then retrieve the rest arguments from the list. For example the procedure

```
(define (foo . r)
  ...
  (let ([a1 (car r)])
    ...))
```

takes zero or more arguments and binds first value of the first argument to 'a1'. If 'foo' is inlined at

```
(foo x y z)
```

then the following code is generated (in Scheme)

```
(let ([r (prim $cons x (prim $cons y (prim $cons z '())))])
  ...
  (let ([a1 (car r)])
    ...)).
```

Unfortunately, inlining 'foo' hasn't achieved much, because when we store the values of 'x', 'y' and 'z' in the list, all the type information we have on them is lost to the (inlined) body of the procedure. We may, for instance, know that 'x' is a fixnum, but we have to assume that 'a1' can have any type because it is bound to the return value of the procedure 'car'. Even if we inline 'car' and get

```
(let ([a1 (prim $car r)]) ...)
```

we still have to assume that 'a1' can have any type since the '$car' primitive can return any type.

Fortunately this problem can be resolved. During the constant folding we can evaluate the expression that constructs the rest list and create a special list, called a *rlist*, which holds the names of the variables rather than their values.

Then when we encounter a '$car' primitive with the rlist as its argument, we can eliminate the primitive and replace it with a variable.

For example, constant folding

    (**prim** $cons x (**prim** $cons y (**prim** $cons z '())))

will yield the rlist ('x' 'y' 'z'), which will be temporarily bound to 'r'. Then when we encounter

    (**let** ([a1 (**prim** $car r)]) ...)

we can replace it with

    (**let** ([a1 x]) ...).

Similarly when if encounter

    (**let** ([r1 (**prim** $cdr r)]) ...)

we can bind the remainder of the rlist ('y' 'z') to 'r1' and continue constant folding. Hopefully, we can eliminate all the operations on the rest list and if the list is no longer used, we can eliminate the procedures that construct it.

The next Scheme standard, the *Revised^5 Report on the Algorithmic Language Scheme*, will include a new form of rest arguments which will be more efficient and easier to optimise. Each procedure that takes rest arguments will have two extra parameters: the number of rest arguments, and a procedure that returns the n^{th} rest argument. This will allow the use of a more efficient data structure for holding the rest arguments and the procedure used to retrieve the rest arguments could be easily inlined.

Chapter 7

Performance

In this chapter the performance of the code generated by the SKI compiler is examined. In particular, we examine the effect of the dynamic optimizations presented in the previous Chapter.

7.1 Methodology

To investigate the effect of the optimizations a number of small benchmark programs were compiled with varying levels of optimization and the execution times compared.

The benchmark programs are small for three reasons:

1. The compiler is *slow*.

2. Some of the dynamic optimizations require human intervention to guide them. β-expansion and specialization optimizations require the user to interactively tell them when to inline or specialize a procedure.

3. There are still a number of bugs in the compiler, which tend to affect larger programs.

The compiler is slow because it is a prototype and was designed for flexibility rather than performance. Many of the optimizations use two or more passes over the entire procedure that is being compiled, consisting of one or more passes to collect information and final pass to perform the transformations. Many of these passes could be combined but have been kept separate for ease of maintenance. The passes are also wasteful of memory and information with each making a new copy of the entire procedure and discarding all the information previously collected. This makes the compiler easier to maintain and modify, but it also makes it very inefficient. One measure of the inefficiency is that the compiler spends approximately 25% of its CPU time in garbage collection.

The second reason, that the compiler requires guidance from a human, is more serious. The frequency of procedure calls in Scheme is so high that even for relatively small examples, many inlining decisions are required. Specialization also requires human intervention, but this is a single decision for each procedure. The thresholding heuristic currently used by the specializer (see Section 6.2.1) appears to be quite effective in deciding what parameters and types to specialize a procedure on.

The small size of the benchmarks do have some advantages. It is possible to try many combinations of optimizations various optimizations to try to isolate the effect of each optimization. It is also possible to modify the benchmarks to determine the upper limits of the optimizations.

7.1.1 Details

The tests were conducted on a lightly loaded Sun SPARCStation 10/51 with 128MB of RAM running SunOS 4.1.3u. The SPARCStation 10/51 has a single SuperSPARC processor clocked at 50MHz. The SuperSPARC processor is three-way superscalar[1] and has 36 kilobytes of on-chip, level 1, cache. The on-chip cache is split into a 20 kB instruction cache and a 16kB data cache. In addition the processor module includes a 1 MB level 2 cache.

Each benchmark was run 10 times and the times were averaged. To measure the CPU time used by the benchmarks we used the `getrusage(2)` [Sun90] system call. We only recorded the user time, the system time was ignored.[2] The

[1]This means it can issue up to 3 instructions per clock cycle, consisting of two integer/ALU operations and a single "special" instruction. Special instructions include branches and floating point operations. There are other restrictions on the instructions that can be issued, for instance only one of the ALU instructions can be a shift.

[2]`getrusage` returns two times, the user time, which is the time spent by the process in user mode, and the system time, which is the time that the process spends running in kernel mode.

resolution of the clock used by `getrusage` is 1/100th of a second. We report all times in milliseconds.

SKI's heap size was set at 4 MB. SKI's runtime system and the code generated by SKI was compiled using GNU CC version 2.5.8 with the "-O2" flag.

Each of the following sections introduces a single benchmark program and presents the results of running that benchmark with various optimization options.

7.2 Loop

```
(define (loop1 n)
  (if (=:2 n 0)
      n
      (loop1 (−:2 n 1))))
```

(a)

```
(define (looprec n)
  (letrec ([lp (lambda (i)
                 (if (= i 0)
                     i
                     (lp (- i 1))))])
    (lp n)))
```

(b)

Figure 7.1: The loop benchmark.

The first benchmark is a simple loop which counts down from the initial value of its parameter 'n' to zero. Figure 7.1a shows the code for the loop which is written as a tail recursive function call. We choose not to use either of Scheme's looping constructs, the named **let** or the **do** loop, since they are just "syntactic sugar" for recursion. The loop is not implemented using an internal procedure, as shown in Figure 7.1b, since it would be slightly slower because of the extra indirection introduced by assignment conversion (see Section 4.3). If the compiler did known procedure call optimizations (Section 6.6.1) then an internal procedure would be slightly faster.

The procedures '=:2' and '−:2' are simplified versions the standard '=' and '−' procedures. '=:2' and '−:2' take exactly two arguments while the standard procedures take one or more arguments and use rest lists. We use '=:2' and '−:2' to simulate the rest list inlining optimization discussed in Section 6.6.2.

	Optimizations	Times (ms)										Average Time (ms)	Relative Time (%)	
(loop1 1000000)														
0	—	3911	4020	3980	3940	3930	3981	3970	3921	3980	4091	3972.4	—	—
1	Inline ‘=:2’ & ‘−:2’	1110	1090	1090	1080	1111	1100	1090	1080	1100	1100	1095.1	27.6	0
2	Specialize on fixnum ‘n’. Inline ‘=:2’ & ‘−:2’	1000	1000	990	1010	990	1020	1000	990	1000	1010	1001.0	25.2	0
3	Unroll ×1. Inline all ‘=:2’ & ‘−:2’.	710	690	721	710	720	710	720	720	730	710	714.1	18.0	0
4	Specialize on fixnum ‘n’. Unroll ×1. Inline all ‘=:2’ & ‘−:2’.	670	690	690	670	670	780	690	681	700	750	699.1	17.6	0
5	Unroll ×3. Inline all ‘=:2’ & ‘−:2’.	510	500	510	510	510	500	500	600	530	511	518.1	13.0	0
(loop2 1000000)														
6	—	780	780	770	780	810	770	790	770	810	790	785.0	71.6	1
7	Unroll ×1.	480	470	470	460	470	460	570	460	460	460	476.0	66.7	3
8	Unroll ×3.	300	310	310	310	300	300	300	310	300	310	305.0	58.9	5
(loop3 1000000)														
9	—	830	790	800	790	810	800	800	811	800	790	802.1	73.2	1
10	Unroll ×1.	430	450	440	440	470	460	450	430	450	470	449.0	62.9	3
11	Unroll ×3.	260	260	270	260	270	250	270	260	260	250	261.0	49.6	5

Table 7.1: Performance of the loop benchmarks.

Table 7.1 shows results of running the several variations on the loop benchmark with an initial argument of 1000000.

Row 0 of the table gives the times collected when none of the dynamic optimizations were enabled but with the front end optimizations enabled (see Chapter 4).

Row 1 of gives the times collected when 'loopl' was re-optimized and the calls to '=:2' and '−:2' inlined.

The relative time column contains two numbers, the first number is ratio of the average time of the current row to the average time of the row indicated by the second number. So the entry in row 1 says that the version of 'loopl' with both procedure calls inlined executes in 27% of the time taken by the unoptimized version of 'loopl'.

Why is it so much faster? Examining the code that the compiler generated reveals three reasons:

1. Two procedure calls have been eliminated.

2. The un-optimized version creates one first level continuation closure per loop (see Section 5.3). This closure holds the continuation and one of the callee-save variables which allows 'n' to become a callee-save variable. When the two procedures are inlined, no closures are created.

3. The second arguments to both '−:2' and '=:2' are both constants. The constant folding optimization can eliminate the type tests on these arguments. The code for '−:2' and '=:2' is contained in Appendix A.5.

Row 2 of the table shows the results of specializing 'loopl' for the case when 'n' is a fixnum. The calls to '−:2' and '=:2' are inlined in the specialized version of the procedure's body and not inlined in the general version (see Chapter 6.2).

Specialization doesn't produce a very dramatic speedup. The reason is that we are exchanging two type tests for a single type test. When 'loopl' is specialized, a type predicate is inserted at the head of the procedure to test the value of 'n'. If 'n' is a fixnum then the specialized version of the procedure's body is executed, otherwise the general version is executed. When '=:2' is inlined in the specialized version, the constant folder can eliminate all the type tests on the

'=:2's first argument, which is now 'n', since 'n' is known to be a fixnum. The same thing happens when '−:2' is inlined. The net result is that one type test is inserted, and two are removed.

Row 3 shows the results of inlining the recursive call to 'loop1', this is equivalent to unrolling the loop by one iteration. All calls to '−:2' and '=:2' are inlined. The difference is not as large it was when we first inlined '−:2' and '=:2', because there are no closure creations to eliminate.

Row 4 shows the result of unrolling the specialized version of the loop. This is not as effective as we might expect since the specialization has no effect on the unrolled copy of the loop. The reason is that even though 'n' is known to be a fixnum in the first, specialized, copy of the loop, we don't know the type of 'n' in the second, unrolled, copy. This is because when we do the subtraction we have to check the result to make sure that the subtraction didn't overflow.[3] If the subtraction did overflow then we have to convert its operands to bignums and do it again. Testing the result of the subtraction requires a branch and after the branch all type information about the result is lost. So in the unrolled copy of the loop the type of 'n' is unknown and none of the type tests can be eliminated.

Finally, row 5 shows the result of unrolling the loop 3 times with no specialization.

```
(define (loop2 n)
  (if (prim $fix= n 0)
      n
      (loop2 (prim $fix- n 1))))
```

(a) loop2

```
(define (loop3 n)
  (if (prim $fix= n 0)
      n
      (loop3 (prim $fix-no n 1)))))
```

(b) loop3

Figure 7.2:

Rows 6, 7, and 8 show results of performing some of the same tests on 'loop2'. As shown in Figure 7.2a, 'loop2' is the same as 'loop1' except that the calls to '=:2' and '−:2' have been replaced with calls to primitives. The point of this

[3]In ski, a fixnum subtraction overflows when the result less than $-(2^{30})$.

experiment is to show the cost of generic arithmetic. Unlike all the variations on 'loopl' discussed above, 'loop2' will fail silently if it is passed a flonum or a bignum, or if the subtraction causes an overflow; 'loopl' one will always work no matter what kind of number it is passed.

As we can see from the relative time column, 'loop2' is significantly faster than 'loopl' and since 'loop2' implicitly assumes that 'n' is a fixnum, it is not affected by the loss of type information when the loop is unrolled.

Rows 9, 10 and 11 shows the results of performing the same tests on 'loop3', which is a variation on 'loop2'. The difference between the two is that in 'loop3' '$fix−' primitive is replaced with '$fix−no'. '$fix−no' doesn't check the subtraction for overflow.

Strangely, the un-optimized version of 'loop3' (Row 9) is slower than the un-optimized version of 'loop2' (Row 6). However, the unrolled versions of 'loop3' (Rows 10 and 11) are faster the the corresponding versions of 'loop2' (Rows 7 and 8), which is what we would expect.

The SELF compiler uses range analysis on integers to eliminate overflow checking [CU90]. If SKI could do the same, then it might be able to compiler 'loopl' so that it is as fast as 'loop3' without sacrificing safety.

7.3 Fibonacci

```
(define (fib n)
  (if (<=:2 n 1)
      1
      (+:2 (fib (−:2 n 1))
           (fib (−:2 n 2)))))
```

Figure 7.3: fib

Our second benchmark, 'fib', is the "classic" recursive Fibonacci function. For the same reasons that we discussed in the previous section, 'fib' is written as a global procedure and uses simplified, two argument, procedures for arithmetic.[4]

Table 7.2 shows the results of evaluating '(fib 28)'.

[4]The source code for '+:2' and '<=:2' is included in Appendix A.5

	Optimizations	Times (ms)										Average Time (ms)	Relative Time (%)	
(fib 28)														
0	—	4880	4851	4930	4831	4830	4991	4830	4821	4980	4821	4876.5	—	—
1	Inline '+:2', '−:2' and '<=:2'.	1710	1741	1770	1780	1880	1781	1780	1780	1750	1790	1776.2	36.4	0
2	Specialize on 'n' is a fixnum. Inline '+:2', '−:2' and '<=:2'.	1611	1710	1660	1660	1690	1651	1660	1630	1720	1680	1667.2	34.2	0

Table 7.2: Performance of the Fibonacci benchmark.

Row 1 of the table shows that inlining '+:2', '−:2' and '<=:2' has results similar to inlining '−:2' and '=:2' in 'loop1', but the effect is not as large. The reason is that we can't completely eliminate the creation of continuation closures, though the number of closures created is reduced from three in the un-optimized version to one in the optimized version.

Row 2 of the table shows that the effect of specializing 'fib' for the case when 'n' is a fixnum is also quite small. In this case specialization introduces one type test and eliminates three.

Unfortunately, an elusive bug in the compiler prevented us from inlining the recursive calls and unrolling 'fib'.

7.4 Deriv

The final benchmark is the Deriv benchmark shown in Figure 7.4. Deriv is taken from the Gabriel benchmark suite which was created by Richard Gabriel to measure the performance of Lisp implementations [Gab85]. Deriv was written by Vaughan Pratt and was ported to Scheme by William Clinger. The full source for Deriv, including comments, is included in Appendix A.6.

Deriv computes the symbolic derivatives of expressions. Unlike the previous benchmarks Deriv does no arithmetic, operates mostly on symbols and lists. The numbers in expressions are treated like symbols.

Table 7.3 shows the results of calling 'deriv' 20,000 times with the argument '(+ (∗ 3 x x) (∗ a x x) (+ b x) 5)'[5].

Unfortunately, the same bug that affected the previous benchmark also affect 'deriv'. The bug appears to occurs when a procedure that starts with sequence

(**if** *exp then* (*procedurecall*))

is inlined, but it does not occur in all such sequences!

Row 1 of the table gives the results of inlining all small procedures except for those that started with the above sequence. The procedures inlined were 'not', 'pair?', 'eq?' and 'cons'. The procedures that were not inlined because of the bug were 'car' and 'cdr'. 'map', 'cadr' and 'caddr' were not inlined because of their size.

[5] $3x^2 + ax^2 + bx + 5$.

```
(define (deriv-aux a) (list '/ (deriv a) a))

(define (deriv a)
  (cond
    ((not (pair? a))
     (cond ((eq? a 'x) 1) (else 0)))
    ((eq? (car a) '+)
     (cons '+ (map deriv (cdr a))))
    ((eq? (car a) '-)
     (cons '- (map deriv
                   (cdr a))))
    ((eq? (car a) '*)
     (list '*
          a
          (cons '+ (map deriv-aux (cdr a)))))
    ((eq? (car a) '/)
     (list '-
          (list '/
               (deriv (cadr a))
               (caddr a))
          (list '/
               (cadr a)
               (list '*
                    (caddr a)
                    (caddr a)
                    (deriv (caddr a))))))
    (else 'error)))
```

Figure 7.4: The Deriv Benchmark

Optimizations		Times (ms)										Average Time (ms)	Relative Time (%)	
(deriv '(+ (∗ 3 x x) (∗ a x x) (∗ b x) 5))														
0	—	4750	4850	4750	4771	4791	4831	4841	4820	4770	4920	4809.4	—	—
1	Inline all small procedures avoiding bug.	3820	3810	3771	3810	3661	3730	3700	3690	3780	3881	3765.3	78.3	0
2	As above, inline special 'car' and 'cdr' too.	3621	3500	3441	3560	3410	3671	3430	3311	3450	3370	3476.4	72.3	0
3	As 2, inline 'list' too.	3190	3031	3360	3380	3410	3760	3250	3050	3110	3190	3273.1	68.1	0

Table 7.3: Performance of the Deriv benchmark.

Row 2 gives the results of inlining special versions 'car' and 'cdr' which do not provoke the compiler bug (see Appendix A.5) as well as all the procedures inlined above.

Row 3 gives the results of inlining the 'list' procedure as well as all procedures inlined by the example in row 2.

We did not try specializing 'deriv' as its argument 'a' has no obvious most frequent type to specialize it on. 'a' is pair or a symbol with almost equal frequency.

The speedup achieved by inlining is not as great as that achieved in the previous two benchmarks. This is probably due to two factors. Firstly, still makes a large number of procedure calls, and secondly, deriv creates many lists.

Chapter 8

Conclusions

Research is what I'm doing when I don't know what I'm doing.
— Wernher von Braun [Arn86]

8.1 Discussion

As the previous chapter shows, dynamic compilation techniques can yield a substantial increase the performance of Scheme programs. The execution speed of simple programs increased by a factor of three or four, but the increase was less substantial for more complex programs.

Surprisingly, the increase in performance that we observed came primarily from inlining procedures rather than eliminating type checking. We had expected that eliminating type checking would have a greater impact on the performance than we observed. Further work is needed to clarify these issues.

8.2 Future Work

Much work needs to be done on the SKI to increase the stability and useabilty of the compiler so that its performance can be evaluated on larger programs. In particular, a set of heuristics for deciding when to inline procedures is urgently needed.

SKI could also perform more optimizations than it currently does. Known call optimisation and rest list inlining (see Sections 6.6.1 and 6.6.2) could result substantial increases in performance. Another optimization that could make a big difference is splitting, as SKI's type predictor loses valuable type information after conditional expressions (see Section 2.1.3). Finally the specialization optimization could be generalized to the continuations of procedure calls as these are a major source of uncertainty in type prediction.

8.3 Conclusion

SKI has demonstrated that dynamic compilation is an effective way of increasing the performance of Scheme programs, although more work is necessary to unleash its full benefits.

Appendix A

Miscellaneous

A.1 Example SELF code

The following is the SELF code to add two points together copied from [US91].

```
+ arg = (
   | newPoint |
   newPoint: copy.
   newPoint x: x + arg x.
   newPoint y: y + arg y.
   newPoint.
)
```

A.2 CPS version of nested if optimization

If we CPS convert the nested **if** expression:

```
(if (if a b c) d e)
```

we get (assuming for simplicity that a, b, c, d and e are variables or literals):

```
(cif a
    (clambda (k1) (jmp b k1))
    (clambda (k2) (jmp c k2))
  (clambda (v)
    (cif v
        (clambda (k3) (jmp d k3))
        (clambda (k4) (jmp e k4))
      k0))).
```

which can be transformed into:

```
(jmp (nlambda k3 () #f (jmp d k3))
  (clambda (dt)
    (jmp (nlambda k4 () #f (jmp e k4))
      (clambda (et)
        (cif a
            (clambda (j1)
              (cif b
                  (clambda (j2) (app dt () j2))
                  (clambda (j3) (app et () j3))
                j1))
            (clambda (j4)
              (cif c
                  (clambda (j5) (app dt () j5))
                  (clambda (j6) (app et () j6))
                j4))
          k0)))))
```

The real transform is slightly more complicated since it must be able handle cases where a, b, c, d and e are more complex expressions.

A.3 Tail call macro for the SPARC

```
0   #define TAIL_CALL2(dest, a0, a1)
1   {
2           register SKI __arg0 asm ("%i0");
3           register SKI __arg1 asm ("%i1");
4           __arg0 = (SKI) a0;
5           __arg1 = (SKI) a1;
6           asm volatile (
7          "jmpl %0, %%g0, %%g0
8           restore"
9           :
10          : "r" (dest), "r" (__arg0), "r" (__arg1)
11          : "%i7");
12
13          DO_NOTHING();
14  }
```

Figure A.1: A two argument tailcall macro for the SPARC.

Figure A.1 shows a C macro which tailcalls a function 'dest' with two arguments 'a0' and 'a1'. The following is a line-by-line description of how the macro works:

2–3 The variables '__arg0' and '__arg1' are declared to hold the arguments. The **asm** syntax is a GNU C extension which instructs the compiler to map the variables onto specific registers, in this case '%i0' and '%i1', the first two input registers in the current register window.

6 The keyword **asm** begins an inline assembler sequence. **volatile** tells the compiler that it shouldn't try to move or otherwise interfere with the sequence.

7 `jmpl` is the SPARC jump-and-link instruction, it sets the program counter to the sum of its first two operands and stores the old value of the program counter in its third operand. The first operand '%0' is replaced by the register holding the destination address 'dest' (which is mapped into an unspecified register on line **10**), the second and third operands specify the

'%g0' register which is always zero. So this instruction jumps to 'dest + 0' and discards the old value of the programme counter.

8 The `restore` instruction, which is executed in the delay slot of the `jmpl`,[1] switches to the previous register window. This means that the "input" registers '%i0–%i7' become the output registers '%o0–%o7' of the previous register window. When the destination function is reached the first instruction executed in the standard function prologue will be a `save` which switches to the next register window and the output registers become the input registers again.

The `restore` instruction also sets the stack pointer back to the value that it had when the calling function was entered, deallocating anything that was allocated on the stack.

9 This line is used to tell the compiler where to find the result of the sequence, in this case there isn't one.

10 This line tells the compiler where to put the arguments to the sequence. '`"r"` (dest)' tells the compiler to make sure that 'dest' is in a register, the string `%0` in the sequnce is replaced with the name of the register. '`"r"` (__arg0)' and '`"r"` (__arg1)' tell the compiler that the values of '__arg0' and '__arg1' are used to prevent it from eliminating them.

11 This line tells the compiler that the value of '%i7', the stack pointer, is changed.

13 This line, which is never executed, is used to inform the compiler that this macro never "returns." 'DO_NOTHING' is declared:

```
volatile void DO_NOTHING(void)
{
    abort();
}
```

[1]I.e., before the jump is completed.

Using the keyword **volatile** like this indicates that 'DO_NOTHING' never returns and as an aid to debugging a core dump is generated (by 'abort') if 'DO_NOTHING' is ever executed. A better solution would be to declare that the **asm** sequence never returns but this is not currently possible.

A.4 Tail call macro for the i386

```
0   SKI ExtraArgRegs[];
1
2   #define TAIL_CALL2(dest, a0, a1)
3   {
4           SKI *__ear = ExtraArgRegs;
5           __ear[0] = (SKI) a0;
6           __ear[1] = (SKI) a1;
7           asm volatile (
8          "movl %%ebp,%%esp
9           popl %%ebp
10          jmp %0"
11          :
12          : "r" (dest)
13          : "%ebp", "%esp");
14
15          DO_NOTHING();
16  }
```

Figure A.2: Two argument tail call macro for the i386.

Figure A.2 shows a two argument tailcall macro used by Ski on i386 based machines running Linux. It is similar in structure to the tailcall macro shown in Appendix A.3. The following discusses the differences on a line-by-line basis:

4–6 The standard calling convention used on i386 machines running Linux 1.0 doesn't pass any arguments in registers. Therefore we have to pass all arguments in a global array 'ExtraArgRegs', '__ear' is a local pointer to the array which will (hopefully) be allocated in a register.

8–9 These lines restore the stack pointer, `%esp`, and and base pointer, `%ebp` to the values that they had when the calling function was entered, deallocating anything that was allocated on the stack by the calling function.

10 Finally we jump to the destination function, the address of which was placed in an unspecified register by line **12**.

A.5 Library procedures

The following is the Scheme code for some of the library procedures discussed in 7.

```
(define =:2
  (lambda (x y)
    (cond ([prim $fixnum? x]
           (cond ([prim $fixnum? y]
                  (prim $fix= x y))
                 ([prim $bignum? y]
                  (prim $big= (prim $fixnum->bignum x) y))
                 ([prim $flonum? y]
                  (prim $flo= (prim $fixnum->flonum x) y))
                 ))
          ([prim $bignum? x]
           (cond ([prim $fixnum? y]
                  (prim $big= x (prim $fixnum->bignum y)))
                 ([prim $bignum? y]
                  (prim $big= x y))
                 ([prim $flonum? y]
                  (prim $flo= (prim $bignum->flonum x) y))
                 ))
          ([prim $flonum? x]
           (cond ([prim $fixnum? y]
                  (prim $flo= x (prim $fixnum->flonum y)))
                 ([prim $flonum? y]
                  (prim $flo= x y))
                 ([prim $bignum? y]
                  (prim $flo= x (prim $bignum->flonum y)))
                 ))
          )))
```

```
(define -:2
  (lambda (x y)
    (cond ([prim $fixnum? x]
           (cond ([prim $fixnum? y]
                  (or (prim $fix- x y)
                      (prim $big- (prim $fixnum->bignum x)
                             (prim $fixnum->bignum y))))
                 ([prim $bignum? y]
                  (prim $big- (prim $fixnum->bignum x) y))
                 ([prim $flonum? y]
                  (prim $flo- (prim $fixnum->flonum x) y))
                 ))
          ([prim $bignum? x]
           (cond ([prim $fixnum? y]
                  (prim $bignum->fixnum?
                        (prim $big- x (prim $fixnum->bignum y))))
                 ([prim $bignum? y]
                  (prim $big- x y))
                 ([prim $flonum? y]
                  (prim $flo- (prim $bignum->flonum x) y))
                 ))
          ([prim $flonum? x]
           (cond ([prim $fixnum? y]
                  (prim $flo- x (prim $fixnum->flonum y)))
                 ([prim $flonum? y]
                  (prim $flo- x y))
                 ([prim $bignum? y]
                  (prim $flo- x (prim $bignum->flonum y)))
                 ))
          )))
```

```
(define <=:2
  (lambda (x y)
    (cond ([prim $fixnum? x]
           (cond ([prim $fixnum? y]
                  (prim $fix<= x y))
                 ([prim $bignum? y]
                  (prim $big<= (prim $fixnum->bignum x) y))
                 ([prim $flonum? y]
                  (prim $flo<= (prim $fixnum->flonum x) y))
                 ))
          ([prim $bignum? x]
           (cond ([prim $fixnum? y]
                  (prim $big<= x (prim $fixnum->bignum y)))
                 ([prim $bignum? y]
                  (prim $big<= x y))
                 ([prim $flonum? y]
                  (prim $flo<= (prim $bignum->flonum x) y))
                 ))
          ([prim $flonum? x]
           (cond ([prim $fixnum? y]
                  (prim $flo<= x (prim $fixnum->flonum y)))
                 ([prim $flonum? y]
                  (prim $flo<= x y))
                 ([prim $bignum? y]
                  (prim $flo<= x (prim $bignum->flonum y)))
                 ))
          )))
```

```
(define +:2
  (lambda (x y)
    (cond ([prim $fixnum? x]
           (cond ([prim $fixnum? y]
                  (or (prim $fix+ x y)
                      (prim $big+ (prim $fixnum->bignum x)
                             (prim $fixnum->bignum y))))
                 ([prim $bignum? y]
                  (prim $big+ (prim $fixnum->bignum x) y))
                 ([prim $flonum? y]
                  (prim $flo+ (prim $fixnum->flonum x) y))
                 ))
          ([prim $bignum? x]
           (cond ([prim $fixnum? y]
                  (prim $bignum->fixnum?
                        (prim $big+ x (prim $fixnum->bignum y))))
                 ([prim $bignum? y]
                  (prim $big+ x y))
                 ([prim $flonum? y]
                  (prim $flo+ (prim $bignum->flonum x) y))
                 ))
          ([prim $flonum? x]
           (cond ([prim $fixnum? y]
                  (prim $flo+ x (prim $fixnum->flonum y)))
                 ([prim $flonum? y]
                  (prim $flo+ x y))
                 ([prim $bignum? y]
                  (prim $flo+ x (prim $bignum->flonum y)))
                 ))
          )))
```

The following are the standard versions of the 'car' and 'cdr'.

```
(define (car x)
  (if (prim $pair? x)
      (prim $car x)
      (c*r-err 'car)))

(define (cdr x)
  (if (prim $pair? x)
      (prim $cdr x)
      (c*r-err 'cdr)))
```

These are the special versions of 'car' and 'cdr' which do not provoke the bug mentioned in Section 7.4.

```
(define (car x)
  (if (prim $pair? x)
      (prim $car x)
      'error))

(define (cdr x)
  (if (prim $pair? x)
      (prim $cdr x)
      'error))
```

The difference between the two versions is that the special versions do not call the error reporting procedure 'c*r-err' if they are passed arguments of the wrong type.

A.6 The Deriv Benchmark Program

The following is the full Scheme source of the Deriv benchmark from the Gabriel benchmark suite [Gab85].

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; File:         deriv.sch
; Description:  The DERIV benchmark from the Gabriel tests.
; Author:       Vaughan Pratt
; Created:      8-Apr-85
; Modified:     10-Apr-85 14:53:50 (Bob Shaw)
;               23-Jul-87 (Will Clinger)
;               9-Feb-88 (Will Clinger)
; Language:     Scheme
; Status:       Public Domain
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; DERIV -- Symbolic derivative benchmark written by Vaughn Pratt.
;;; It uses a simple subset of Lisp and does a lot of  CONSing.

; Returns the wrong answer for quotients.
; Fortunately these aren't used in the benchmark.

(define (deriv-aux a) (list '/ (deriv a) a))

(define (deriv a)
  (cond
    ((not (pair? a))
     (cond ((eq? a 'x) 1) (else 0)))
    ((eq? (car a) '+)
     (cons '+ (map deriv (cdr a))))
    ((eq? (car a) '-)
     (cons '- (map deriv
                   (cdr a))))
    ((eq? (car a) '*)
     (list '*
           a
           (cons '+ (map deriv-aux (cdr a))))
    ((eq? (car a) '/)
     (list '-
           (list '/
                 (deriv (cadr a))
                 (caddr a))
           (list '/
                 (cadr a)
                 (list '*
                       (caddr a)
                       (caddr a)
                       (deriv (caddr a))))))
    (else 'error)))

(define (run)
  (do ((i 0 (+ i 1)))
      ((= i 1000))
    (deriv '(+ (* 3 x x) (* a x x) (* b x) 5))
    (deriv '(+ (* 3 x x) (* a x x) (* b x) 5))
    (deriv '(+ (* 3 x x) (* a x x) (* b x) 5))
    (deriv '(+ (* 3 x x) (* a x x) (* b x) 5))
    (deriv '(+ (* 3 x x) (* a x x) (* b x) 5))))

;;; call:  (run)

(run-benchmark "Deriv" (lambda () (run)))
```

Bibliography

[ACS93] Norman Adams, Pavel Curtis, and Mike Spreitzer. First-Class Datatype Representations in SchemeXerox. In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 139–146, 1993. Published as *SIGPLAN Notices*, 28(6).

[AJ89] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Sixteenth ACM Symposium on Princples of Programming Languages*, pages 293–302, New York, 1989. ACP Press.

[App87] Andrew W. Appel. Garbage Collection Can Be Faster than Stack Allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[App92] Andrew W. Appel. *Compiling with continuations.* Cambridge University Press, 1992.

[App94] Andrew W. Appel. Emulating Write-Allocate on a No-Write-Allocate Cache. Technical Report CS-TR-459-94, Princeton University, June 1994.

[Arn86] Ken Arnold. `fortune(1)`, 1986.

[AS94] Andrew W. Appel and Zhong Shao. An Empirical and Analytic Study of Stack vs. Heap Cost for Languages with Closures. Technical Report CS-TR-450-94, Princeton University, 1994. Availible for FTP from `ftp.cs.princeton.edu:/reports/1994/`.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques and Tools.* Addison–Wesley, 1986.

[Bak92] Henry G. Baker. Inlining Semantics for Subroutines which are Recursive. *SIGPLAN Notices*, 27(12):39–46, December 1992.

[Bak94] Henry G. Baker. CONS Should Not CONS Its Arguments, Part II: Cheny on the M.T.A. Posted to `comp.lang.scheme.c` newsgroup, Feburary 4, 1994.

[Bar89] Joel F. Bartlett. Scheme->C a Portable Scheme-to-C Compiler. Research Report 89 1, DEC Western Research Laboratory, Palo Alto, California, January 1989.

[CCKM94] Quintin Cutts, Richard Connor, Graham Kirby, and Ron Morrison. An Execution-Driven Approach to Code Generation. In *Proc. of the Seventeenth Annual Computer Science Conference*, pages 83–92. Austrailian Computer Science Communications, 1994.

[Cha92] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages.* PhD thesis, Stanford University, 1992.

[Cha93] Craig Chambers. The Cecil Language: Specification and Rationale. Technical Report TR-93-03-05, Department of Computer Science and Engineering, University of Washington, 1993.

[CR91] William Clinger and Jonathan Rees. Revised^4 Report on the Algorithmic Language Scheme. *ACM Lisp Pointers IV*, 1991. FTPable from `altdorf.ai.mit.edu` and WWWable from `ftp://altdorf.ai.mit.edu/archive/scm/HTML/r4rs_toc.html`.

[Cri92] R. Cridlig. An optimizing ML to C compiler. In *Workshop on ML and its applications.* ACM, 1992.

[CU89] Craig Chambers and David Ungar. Customization: Optimizing Compiler Technology for SELF, a Dynamically-Typed Object-Oriented Programming Language. In *Proceedings of the SIGPLAN*

'89 Conference on Programming Language Design and Implementation, pages 146–160, 1989. Published as *SIGPLAN Notices*, 24(7).

[CU90] Craig Chambers and David Ungar. Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. In *OOPSLA '90 Proceedings*, pages 150–164. Association for Computing Machinery, 1990. Published as *SIGPLAN Notices*, 24(?).

[CU91] Craig Chambers and David Ungar. Making Pure Object-Oriented Languages Practical. In *OOPSLA '91 Proceedings*, pages 1–15. Association for Computing Machinery, 1991. Published as *SIGPLAN Notices*, 25(6).

[CUL89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes. In *OOPSLA '89 Proceedings*, pages 49–70. Association for Computing Machinery, 1989. Published as *SIGPLAN Notices*, 24(10).

[DC93] Jeffery Dean and Craig Chambers. Training Compilers for Better Inlining Decisions. Technical Report 93-05-05, University of Washington, 1993.

[DOD83] U.S. Department of Defence. *Reference Manual for the Ada Programming Language*, 1983.

[FM91] Scott E. Fahlman and David B. McDonald. Design Considerations for CMU Common Lisp. In *Topics in Advanced Language Implementation* [Lee91].

[Gab85] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, 1985.

[Gil91] Dave Gillespie. *p2c - Pascal to C translator, version 1.20*, 1991. Available for FTP from `csvax.cs.caltech.edu`.

[H$\ddot{9}$4] Urs Hölzle. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming.* PhD thesis, Stanford University, August 1994.

[Hau94] B. Hausman. *Turbo Erlang: Approaching the Speed of C*, chapter 0, pages 119–135. Kluwer Academic Publishers, 1994.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *ECOOP '91, European Conference on Object Oriented Programming*, pages 21–38. Springer Verlag, 1991.

[HDB90] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing Control in the Presence of First–Class Continuations. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 66–77, 1990.

[HP90] John L. Hennessy and David A. Patterson. *Computer Architecture: a Quantitative Approach.* Morgan Kaufman, 1990.

[HPJW$^+$92] P. Hudak, S.L. Peyton Jones, P.L. Walder, B. Boutel, J. Fairbain, J. Fasel, M. Guzman, K. Hammond, J. Hughes, T. Johnsson, R. Kieburtz, R.S. Nikhil, W. Partian, and J. Peterson. Report on the functional programming language Haskell, Version 1.2. *SIGPLAN Notices*, 1992.

[JGS93] N. Jones, K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 1993.

[Jon92] Simon L. Peyton Jones. Implementing lazy functional languages on stock hardware: the Spineless Tagless G-machine. *Journal of Functional Programming*, 2(2):127–202, April 1992.

[Jou93] Norman P. Jouppi. Cache write policies and performance. In *Proceeedings of the 20th Annual International Symposium on Computer Architecture*, pages 191–201. ACM Press, May 1993.

[KEH91] David Keppel, Susan J. Eggers, and Robert R. Henry. A Case for Runtime Code Generation. Technical Report 91-11-04, Dept. of Computer Science and Engineering, Seattle, 1991.

[KKR$^+$86] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An Optimising Compiler for Scheme. In *Proceedings of the ACM SIGPLAN'86 Symposium on Compiler Construction*, pages 219–233. ACM Press, June 1986.

[KR78] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, New Jersey, 2nd edition, 1988.

[Lee91] Peter Lee. *Topics in Advanced Language Implementation.* MIT Press, Cambridge, Massachusetts, U.S.A, 1991.

[MBCD89] R. Morrison, A. L. Brown, R. C. H. Connor, and A. Dearle. *The Napier88 Reference Manual.* University of St. Andrews, 1989.

[Mey88] Bertrand Meyer. *Object-oriented software construction.* Prentice-Hall, 1988.

[MTH89] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1989.

[Nor95] Kevin Normoyle. `Re: Cache write miss policies`. Personal email message from `Kevin.Normoyle@Eng.Sun.COM` reference `<9501231929.AA26575@gluon.Eng.Sun.COM>` received 24 January, 1995.

[Omo93] Stephen Omohundro. The Sather 1.0 Specification. Technical report, International Computer Science Institute, Berkeley, 1993.

[Pae93] Andreas Paepcke. *Object Oriented Programming: The CLOS Perspective.* MIT Press, 1993.

[Pet94] Mikael Pettersson. RML – A New Language and Implementation for Natural Semantics. In M. Hermenegildo and J. Penjam, editors, *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming*, volume 844 of *Lecture Notes in Computer Science*, pages 117–131. Springer Verlag, 1994.

[Pet95] Mikael Pettersson. Simulating Tailcalls in C. Submitted to *Functional Programming and Computer Architecture '95*, FTPable from `ftp.ida.liu.se:/pub/labs/pelab/rml/tailcall.ps.gz`, January 1995.

[Ple91] Uwe F. Pleban. Compilation Issues in the Screme Implementation for the 88000. In *Topics in Advanced Language Implementation* [Lee91].

[SA94] Zhong Shao and Andrew W. Appel. Space Efficient Closure Representations. In *Conference Record of the 1994 ACM Symposium on Lisp and Functional Programming.* Association for Computing Machinery, June 1994.

[SH87] Peter A. Steenkiste and John L. Hennessy. Tags and Type Checking in LISP: Hardware and Software Approaches. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 50–59. ACM/IEEE, October 1987.

[Sha92] Andrew Shalit. *Dylan: An object-oriented dynamic language.* Apple Computer Inc., Cupertino, CA, 1992.

[SHC94] Zoltan Somogyi, Fergus James Henderson, and Thomas Charles Conway. The implementation of Mercury, an efficient purely declarative logic programming language. Technical report, Dept. of Computer Science, University of Melbourne, 1994. WWWable from: `http://www.cs.mu.oz.au/~zs/papers/mercury.ps.gz`.

[SPA92] SPARC International. *The SPARC Architecture Manual, Version 8*, 1992.

[Sta92] Richard M. Stallman. *Using and porting Gnu CC, Version 2.0.* Free Software Foundation Inc., 1992.

[Ste78] Guy L. Steele. RABBIT: A Compiler for Scheme. Technical Report AI Memo 474, MIT, 1978.

[Ste84] Guy L. Steele jr. *Common Lisp: The Language.* Digital Press, 1984.

[Ste91] Peter A. Steenkiste. The Implementation of Tags and Run-Time Type Checking. In *Topics in Advanced Language Implementation* [Lee91].

[Str91] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 2 edition, 1991.

[Sun90] Sun Microsystems. `getrusage(2)`, 1990.

[SW94] M. Serrano and P. Weis. $1+1=1$: an optimizing caml compil. In *ACM SIGPLAN Workshop on ML and its Applications*, 1994.

[TAL91] D. Tarditi, A. Acharya, and P. Lee. No assembly require: Compiling Standard ML to C. Technical Report CMU-CS-90-187, Carnegie Mellon University, Pittsburg, Pennsylvania, School of Computer Science, March 1991. FTPable from `dravido.soar.cs.cmu.edu:/usr/nemo/sml2c`.

[TDM94] David Tarditi, Amer Diwan, and Eliot Moss. Memory Subsystem Performance of Programs Using Copying Garbage Collection. In *Conference Record of the 21st Annual ACM Conference on Principles of Programming Languages*, pages 1–14, January 1994.

[THL+86] George S. Taylor, Paul N. Hilfinger, James R. Larus, David A. Patterson, and Benjamin G. Zorn. Evaluation of the SPUR Lisp Architecture. In *The* 13th *Annual International Symposium on Computer Architecture*, pages 444–452. IEEE and the ACM, IEEE Computer

Society Press, June 1986. Published as *Computer Architecture News*, 14(2).

[US87] David Ungar and Randall B. Smith. SELF: The power of simplicity. In *OOPSLA '87 Proceedings*, pages 227–242. Association for Computing Machinery, 1987. Published as *SIGPLAN Notices*, 22(12).

[US91] David Ungar and Randall B. Smith. SELF: The power of simplicity. *Lisp and Symbolic Computation*, 4(3), 1991.

[Van77] Eric J. Van Dyke. A Dynamic Incremental Compiler for an Interpretive Language. *Hewlett-Packard Journal*, 28(11):17–23, July 1977.

[VP89] Steven R. Vegdahl and Uwe F. Pleban. The Runtime Enviroment for Screme, a Scheme Implementation on the 88000. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 172–182. ACM/IEEE, 1989. Published as *SIGPLAN Notices*, 24(Special Issue).

[WC93] Andrew K. Wright and Robert Cartwright. A Practical Soft Type System for Scheme. Technical Report TR93-218, Dept. of Computer Science, Rice University, Houston, December 1993.

[Wil92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In Y. Bekkers and J. Cohen, editors, *Memory Management: International Workshop IWMM 92*, number 637 in Lecture Notes in Computer Science, pages 1–42, Berlin, 1992. Springer Verlag. This paper and an expanded version are FTPable from: `cs.utexas.edu:/pub/garbage`.

[YH85] Taiichi Yuasa and Masami Hagiya. Kyoto common lisp report. Technical report, Research Institute for Mathematical Sciences, Kyoto University, 1985.