Compilation By Program Transformation

Richard Andrews Kelsey

YALEU/CSD/RR #702
May, 1989

Compilation By Program Transformation

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by

Richard Andrews Kelsey

May, 1989

Abstract

Compilation By Program Transformation

Richard Andrews Kelsey
Yale University
1989

This disseration describes a simple compiler, based on concepts from denotational semantics, that can be used to compile standard programming languages and produces object code as efficient as that of production compilers. The compiler uses only source-to-source transformations. The transformations are performed on programs that have been translated into an intermediate language resembling the lambda calculus. The output of the compiler, while still in the intermediate language, can be trivially translated into machine code for the target machine. The compilation by transformation strategy is simple: the goal is to remove any dependencies on the intermediate language semantics that the target machine cannot implement directly. Front-ends have been written for Pascal and BASIC and the compiler produces code for the MC68020 microprocessor.

Preface to the "2012 edition"

This is a version of Richard Kelsey's 1989 Yale doctoral dissertation, lightly hacked so that it will compile in a 2012 LaTeX. This version also has mildly improved typography: non-ugly fonts, single-spaced text, useful PDF metadata, and active hypertext links.

Around 2010, I discovered that Richard Kelsey's doctoral dissertation from Yale had vanished from the net. Yale's Computer Science Department no longer had it on their web page. Worse, Richard himself no longer had a PostScript or PDF copy.

This is a shame: Kelsey's dissertation is a fine piece of work. It's one of about 5–6 items I recommend to students who want to learn the technology of functional-language / lambda-calculus compilers. Among other things, it was extremely influential on the design of the rather well-known SML/NJ compiler described in Appel's *Compiling with Continuations* book. I was reduced to lending out my personal hardcopy to my graduate students with the strict injunction not to lose it or put damp coffee mugs on it. Or you could pay for a large, low-quality, scanned version from University Microfilms International.

Then, in November of 2011, Richard stumbled over the 1989 LaTeX source for his dissertation and sent it to me, where it sat on my hard-drive until April the following spring, when I have taken a Saturday afternoon and gotten it to compile in a modern LaTeX.

Here's what I did to Kelsey's original source:

- Ported it from 1980's-era LaTeX to LaTeX 2e. Besides having to port the obsolete `yalethesis.sty` style package (see below), there was almost nothing to do here.

- Killed the archaic `yalethesis.sty` style file, retaining just enough of its API (mostly the tech-report titlepage machinery) to compile the document.

- Switched from Knuth's awkward Computer Modern font to a more reason-

able Times Roman and moved the document to LaTeX's `book` class. This tightened the document up from 120 pages of excessively spaced lines to 107 pages of easier-on-the-eyes single-spaced text.

- Fixed some bad line breaks that shoved material out into right margin.
- Set up the document for two-sided page layout.
- Fixed a typesetting bug in the original document that prevented the bibliography from being typeset: the original document had `\samepage` declarations in chapters 8 and 9 that weren't bounded inside a `{...}` scope. This caused the entire bibliography to get typeset onto a single, very long page.
- Added LaTeX's `hyperref` package, to get intra-document links (*e.g.*, active table of contents, section refs and bib cites) and to set PDF metadata properties (author, title, subject, and keywords), which should help search engines index the document.
- Undid three ugly hacks Kelsey used that are no longer needed with modern LaTeX:
  - In two places, Kelsey had been forced to insert the actual text for multicites that LaTeX 2.09 wouldn't split across a line.
  - He `\input` an empty file for bug-workaround reasons that I don't understand. In any event, they do not appear to pertain in 2012.

I've taken the trouble to list out each of these issues to make the point that the text is otherwise *completely unaltered* from the LaTeX used to produce the original Yale Tech Report YALEU/CSD/RR #702 of Kelsey's dissertation.

However, if you intend to use this document to produce citations to Kelsey's work, be aware that the re-typesetting I've done *has* altered the pagination of the text—so if your cites include page numbers, you'll need to make clear that you are referring to this particular "second edition" document, *not* the original Tech Report from Yale.

Olin Shivers
Northeastern University
April 14, 2012

Contents

Acknowledgements

I would like to thank my advisor, Paul Hudak, for years of support, encouragement, advice and all the rest, but especially for allowing me to do what I wanted to do. My readers, Marina Chen, Alan Perlis, and William Clinger, provided an amazingly broad range of insight into this dissertation.

The work presented here is inextricably intertwined with that of the other members of the T project: Norman Adams, David Kranz, Rick Mohr, Jim Philbin, and Jonathan Rees. I can only hope that I have been as helpful to them as they were to me. Many of the ideas in this dissertation are a direct result of conversations I have had with Jonathan Rees. Norman Adams wrote the assembler used in the current implementation of the compiler.

I would also like to thank Dana Angluin, who twice awakened my interest in computer science, and David and Deirdre Byrne who provided a lot of love and silliness in a series of homes, notably at 374 Crown Street.

This work was supported the National Science Foundation under grant number DCR-8451415 and the Department of Energy under grant number DOE FG02-86ER25012.

Chapter 1

Introduction

1.1 The problem

There are many problems with compilers:

1. Different ones are needed for different languages.
2. Different ones are needed for different machines.
3. Many do not implement the source language correctly.
4. Their output is often inefficient.
5. They run slowly.

An ideal compiler would compile any language for any machine, produce very efficient code very quickly, and never make a mistake. This dissertation is not about such a compiler. The compilation method described here can be used to write a compiler that will compile many languages for some machines, produce efficient code, and seldom make a mistake. The main issues being addressed are compiling different source languages, correctness, and producing efficient output.

Compilation is done here using source-to-source transformations on programs in an intermediate language based on the lambda calculus. The compiler takes as input a program in the intermediate language and produces an equivalent program, also in the intermediate language, that can be run on the target machine. The transformations implement the parts of the intermediate code that the machine does not.

How does this solve the problems? The intermediate language has the variable binding and control flow of the lambda calculus and also gives direct access to the target machine's instructions and memory. This makes it easy to write translators for different source languages. The correctness comes from using only simple source-to-source transformations that can be shown to be correct as well as being easy to implement correctly. Efficient output results from the use of many transformations that simplify the program during the compilation process.

In this thesis the ability to compile multiple source languages is demonstrated by actually compiling programs written in several languages and by appeal to the known generality of the lambda calculus. The efficiency of the output is shown by comparing the speed of the compiled code with that produced by a hand-coded compiler. While speed of compilation is not one of the issues being addressed here, a compiler must run reasonably quickly to be usable, and measurements of compilation speed are given. The compilation method is not tied to any particular machine but it has only been used to compile code for the Motorola 68020 microprocessor [Motorola 85] and I make no claims as to the ease of porting it to a different architecture.

Demonstrating the correctness of an actual compiler (as opposed to just a compilation strategy or algorithm) is difficult. The current implementation of the compiler consists of thousands of lines of Scheme code. It is hard to imagine how such a large program could be proven correct. The transformations that comprise the compiler's algorithm are specified exactly and can be seen to be correct. Showing the absolute correctness of the implementation is more difficult. The correctness of the current implementation of the compiler is shown by running the output of the implementation. Compiling test files tests the implementation fairly well as the simplicity and independence of the transformations allows relatively simple test cases to exercise every part of the compiler.

1.2 Compilation by program transformation

The compiler works entirely by source-to-source transformations. It is unusual in that its output and input are in the same language. The program to be compiled is first translated into the compiler's intermediate language and then transformed into an equivalent program, still in the same intermediate language. This method of compilation is possible because, although the two languages have completely different semantics, the syntax of the target machine's language is a subset of that of the intermediate language. The program produced by the compiler has the

same meaning as the initial program when interpreted either as an intermediate language program or as a machine language program.

$\mathcal{S}_i$ = *intermediate language semantics*
$\mathcal{S}_m$ = *machine semantics*
$\mathcal{P}$ = *initial program*
$\mathcal{C}$ = *compiler*

$$\mathcal{S}_i(\mathcal{P}) = \mathcal{S}_i(\mathcal{C}(\mathcal{P})) = \mathcal{S}_m(\mathcal{C}(\mathcal{P}))$$

Compilation consists of transforming the program in such a way that it has the same meaning in the intermediate language and the machine language. The compilation transformations are based on the similarities and differences of the semantics of the two languages as expressed in denotational descriptions. The intermediate language is the call-by-value lambda calculus with procedure and data constants [Plotkin 75] and the addition of an implicit store. The machine language is an assembly language with a syntax made to look like the lambda calculus. The machine is assumed to be a Von Neumann machine with a store and register-to-register instructions. The identifiers in the machine language represent the machine's registers. Denotational semantics descriptions [Stoy 77, Gordon 79, Schmidt 86] for both languages are presented in Chapter 2.

In motivating the transformation methodology, it is helpful to consider the basic properties of the source and target languages.

Properties of the Intermediate Language

1. Call and return
2. Lexically nested scoping
3. Very large set of homogeneous identifiers
4. A store
5. Call by value

Properties of the Machine Language

1. Goto
2. Flat scoping

3. Very small set of nonhomogeneous identifiers
4. A store
5. Call by value

As can be seen the first three properties are very different for the two languages. The compilation transformations are based on these differences.

Compilation consists of translating the program into the intermediate language and then performing four global transformations. In addition, numerous transformations are done to improve the efficiency of the final code. Most of these are applied after the introduction of explicit continuations and before environments are introduced.

Compilation is performed in five steps:

1. Translating to intermediate code
2. Making the program linear
3. Adding explicit continuations
4. Adding explicit environments
5. Identifier renaming / register allocation

The translation into intermediate code is a necessary first step. Making the program linear and adding explicit continuations implement call and return in terms of gotos that pass arguments. Adding environments uses the store to implement the lexical scoping of the intermediate language. Finally, register allocation and identifier renaming restrict the program to using only the identifiers of the target language. The resulting program depends only on the properties of the store, the call-by-value semantics of the source and target languages, and the overlap in their identifier scoping rules. Thus its meaning is the same in both languages.

Each step in the compilation process restricts the form of the code, and the steps that follow must preserve these restrictions. The code expands as the compiler moves more and more of the work of the intermediate language's semantics into the program. At the same time the code improvement transformations work to reduce the size of the code as every expansion of the code provides more opportunities to improve it.

1.2.1 Source code into intermediate language

Initially, a front-end specific to the source language translates the program into the compiler's intermediate language. As the intermediate language is similar to the lambda calculus, a front-end is similar to a denotational semantics for the source language. This makes it easy to write correct front-ends.

1.2.2 Making the program linear

This transformation gives an explicit order to the applications in the program and introduces identifiers for all temporary values. In the linear code arguments in applications are never applications except for arguments to applications of `lambda` expressions with only one argument.

1.2.3 Adding continuations

The previous transformation introduced `lambda` expressions to bind the results of applications. Here these expressions are moved into the applications themselves as continuations. The transformation is given in [Plotkin 75], made slightly more complex due to a more complicated syntax and a desire to limit the size of the resulting program. Every procedure takes an additional argument that is the continuation to be called when the procedure has finished. Returns in the procedure are replaced with calls to the continuation. The resulting program is in 'continuation passing style' (CPS).

After conversion to continuation passing style the code is more structured: 1) arguments to calls are never calls, 2) the bodies of lambda expressions are always calls, 3) there are no longer any returns, just calls. The parts of the compiler that follow must preserve the continuation passing nature of the transformed program.

1.2.4 Code improvements

Conversion to continuation passing style is followed by a number of code improving transformations. These include both local transformations such as beta-substitution and two global transformations, one of which is based on flow analysis. These transformations could have been done before the transformation of the program into CPS, but they are simpler when done afterwards due to the increased regularity of the code. For example, beta substitution may be done with-

out reference to side-effects as arguments to applications are never applications themselves.

1.2.5 Adding environments

This transformation adds explicit environments to the program, much as the conversion to CPS added continuations. Calls are added to the program to construct the environments and to write and read the values they contain. Procedures have their lexical environments passed to them as arguments. After this transformation the only abstractions that may have free identifiers are continuations to calls to primitive procedures. As procedures no longer return or have free identifiers, procedure calls have become gotos that pass arguments.

The code could now run on a machine that had a register for every different identifier used in the program. The final phase, register allocation, replaces the identifiers with the names of the registers on the target machine.

After the environments have been added, more code improving transformations are applied. Examples of these include removing unused environments and removing calls that write into locations that the program never reads.

1.2.6 Register allocation

Register allocation is no different than in any other compiler. Once registers have been allocated to hold values, calls are introduced into the code to move the values to and from registers and temporary locations in the stack. At this point every identifier's value resides in a single register throughout its lifetime, and every identifier is renamed to be the register that contains its value. The program can now be viewed as an assembly language program (with a somewhat unusual syntax) for the target machine.

1.2.7 Extensions

Some programming language features require direct manipulation of the structures the compiler introduces. For example, many languages have some form of non-local return from procedures or loops that is often implemented by summarily removing continuations from the stack. The compiler can accomodate such nonstandard use of continuations (or lexical environments or stores) through the use of special primitive procedures and possibly additional compilation transformations.

1.3 Other solutions

A great deal of effort has been expended on improving compilers and solving the problems listed earlier. In this section I compare the methods described in this thesis with some other solutions to the same problems.

1.3.1 Multiple source languages

The standard way of producing a compiler that can compile more than one language is to write a compiler for a general intermediate language and translators that translate the different source languages into the intermediate language. This is exactly what is done here. The usefulness of this approach depends entirely on the generality of the intermediate language and the ease of writing translators for different programming languages. The intermediate language I use is extremely powerful as it allows lexical closures and recursive procedures.

1.3.2 Correctness

Another method for solving the problem of needing different compilers for different languages, and simultaneously addressing the need for compiler correctness, is to generate compilers automatically from a description of the input language. As long as the compiler generating program is correct all of the generated compilers will be correct (or at least as correct as the input to the compiler generator).

There have been several such compiler generators, such as those in [Paulson 82] and [Lee 87], written using denotational semantic [Stoy 77, Gordon 79, Schmidt 86] or attribute grammar [Knuth 68] descriptions of the source languages as input. The generated compilers tend to be very slow and their output is inefficient. The problem appears to be that the semantic specifications provide a description of the programming language that is not particularly appropriate for compilation to machine code. The generated compilers typically do not produce native machine code and thus their output needs to be either interpreted, with a resulting loss in efficiency, or compiled further, which requires another compiler.

The crucial distinction between the semantics based compiler generators and the compilation method described here is that the transformational compiler is a general compiler, not a compiler generator. While both approaches translate the source program into a form of the lambda calculus, here the identifier bindings, continuations, and the store of the source program are implemented using the

bindings and continuations of the lambda calculus along with an implicit store. This puts restrictions on the ways in which the bindings, continuations, and store can be used by the source program and thus allows the compiler to implement them efficiently. In this way much of the utility and generality of using the lambda calculus to describe programming languages can be obtained without paying the performance cost of running general lambda calculus programs.

Another approach to compiler correctness is to treat a compiler as any other program and attempt to prove formally that the algorithms or the program itself are correct as in [Clinger 84]. Again, denotational semantics can be used to specify the desired behavior of the compiler. This approach has all of the general problems of proving program correctness as described in [DeMillo 78].

1.3.3 Efficient output

There are many different techniques used to improve the efficiency or reduce the size of the output of compilers [Aho 86, Barrett 79]. Here many of these same optimizations have been recast in the form of program transformations. Some traditional code improvements, such as boolean short-circuiting, are not done as single transformations but come about throughout the interaction of several simpler transformations.

Program transformations have been used in several compilers to improve the quality of the code produced. Lists of useful transformations have been published [Standish 76]. The usefulness of program transformations comes from the ease of proving individual transformations correct and of adding new transformations to a transformation based system. Rabbit [Steele 78], S1 [Brooks 82], and Orbit [Kranz 86], as well as others, use many of the local code improvement transformations used here.

1.3.4 Conclusion

The compilation method presented here encompasses parts of several other approaches to compiler design. Passing continuations as explicit arguments has been used in other compilers, as in [Steele 78, Kranz 86]. Passing environments as explicit arguments, either in parts or as an aggregate object, has also been done before in [Johnsson] and [Feeley]. The use of an intermediate language to allow compilation of more than one language is quite common, as is the use of program transformations. Indeed, there is at least one other compiler based solely on program transformations [Boyle 84, Boyle 86].

Here all of these techniques have been put together in a common framework. Only transformations are used and only on programs in one intermediate language. The semantics of the intermediate language and of the target machine language are specified and the compilation transformations are based directly on the differences and similarities of the two languages. The result is a simple, correct, and efficient approach to compilation.

Chapter 2

Semantics

2.1 The compiler's intermediate language

The compiler's intermediate language is essentially the call-by-value lambda calculus with procedure and data constants and with the addition of a store. It also has three distinguished types of abstractions, all having the same semantics. The different abstraction forms allow the compiler to distinguish between continuations and other abstractions and make the compiler's transformations simpler.

In addition, identifiers have associated run-time types that specify the size of the data the identifier represents. The compiler uses these types when generating code to move values from place to place in the machine and requires that any identifier be bound to values of only one size. Primitives must also specify the sizes of any values that they return. In the current implementation a size is just the number of bits in the machine representation of a value.

For simplicity, the semantics given here does not include any error checking. Possible errors include referencing unbound identifiers, binding an identifier to a value that is not of the correct size, or calling a procedure with the wrong number of arguments. Primitives must be called with the right number of both continuations and arguments.

2.1.1 The store

The main difference between the compiler's internal language and the lambda calculus is the former's implicit store. As only the primitive procedures have access to the store, and the behavior of the primitive procedures is not specified in

the denotational description of the internal language, the store is largely invisible in the semantics as well as in the language itself. However, the presence of the store does force the use of continuations in the semantics to specify the order in which applications of primitive procedures use and modify the store.

2.1.2 Notation

$t*$	a sequence of zero or more ts
t^+	a sequence of one or more ts
$s\S t$	sequence concatenation
$\langle \ldots \rangle$	sequence formation
$s \downarrow n$	the nth element of sequence s
$s \dagger n$	s with the first n elements removed
$\#s$	the length of sequence s
$x\ in\ D$	injection of x into domain D
$x\|D$	projection of x to domain D
$\rho[x/i]$	environment update "ρ with x for i"

2.1.3 Syntax

K ∈ Con		constants
I ∈ Ide		identifiers
P ∈ Pri		primitives
L ∈ Pro		procedures
	⟶	`(proc I (I*) E)`
C ∈ Eco		continuations
	⟶	`(lambda (I*) E)` | `(cont (I*) E)`
A ∈ App		applications
	⟶	`(C E*)` | `(P (C*) E*)` | `(return P E*)`
E ∈ Exp		expressions
	⟶	K | I | L | C | A |`(block E* E`$_0$`)`

In code examples primitives will be written with a `$` before their names, as in `$add`. Constants will be preceded with a `'`, such as `'100`.

2.1.4 Domain equations

$\nu \in$	N	natural numbers
$\alpha \in$	L	locations
	M	miscellaneous
	F = E* $\to$ K $\to$ C	procedure values
$\epsilon \in$	E = L + M + F	expressed values
$\sigma \in$	S = L $\to$ E	stores
$\rho \in$	U = Ide $\to$ E	environments
$\theta \in$	C = S $\to$ S	command continuations
$\kappa \in$	K = E $\to$ C	expression continuations
$\phi \in$	G = E* $\to$ N $\to$ C	primitive continuations

Primitives both return a list of results and specify which actual continuation to call (as well as accessing and updating the store).

There is no answer domain as programs do not return answers but instead return a modified store.

2.1.5 Semantic functions

$\mathcal{K}_i$: Con $\to$ E
$\mathcal{P}_i$: Pri $\to$ E* $\to$ G $\to$ C
$\mathcal{E}_i$: Exp $\to$ U $\to$ K $\to$ C
$\mathcal{E}_i$* : Exp* $\to$ U $\to$ (Exp* $\to$ C) $\to$ C

$\mathcal{K}_i$ translates data constants into values. It will not be specified further. $\mathcal{P}_i$ specifies the behavior of primitives, which are the procedure constants. Examples of primitives are given in the next chapter. $\mathcal{E}_i$ gives meanings to expressions as follows:

$$\mathcal{E}_i[\![\mathrm{K}]\!] = \lambda\rho\kappa \,.\, \kappa\,(\mathcal{K}_i[\![\mathrm{K}]\!])$$

$$\mathcal{E}_i[\![\mathrm{I}]\!] = \lambda\rho\kappa \,.\, \kappa\,(\rho\,[\![\mathrm{I}]\!])$$

Constants are handed to $\mathcal{K}_i$ and identifiers to the environment. The result is passed to the continuation.

$$\mathcal{E}_i[\![\texttt{(lambda (I*) E)}]\!] =$$
$$\lambda\rho\kappa \,.\, \kappa((\lambda\epsilon^*\kappa' \,.\, \mathcal{E}_i[\![\mathrm{E}]\!]\ (\mathit{extends}\ \rho\ \mathrm{I}^*\ \epsilon^*)\ \kappa')\ \mathit{in}\ \mathsf{E})$$

$\mathcal{E}_i[\![$`(cont (I*) E)`$]\!] =$
$\lambda\rho\kappa\,.\,\kappa((\lambda\epsilon^*\kappa'\,.\,\mathcal{E}_i[\![\mathsf{E}]\!]\ (\textit{extends}\ \rho\ \mathrm{I}^*\ \epsilon^*)\ \kappa')\ in\ \mathsf{E})$

$\mathcal{E}_i[\![$`(proc I (I*) E)`$]\!] =$
$\lambda\rho\kappa\,.\,\kappa((\lambda\epsilon^*\kappa'\,.\,\mathcal{E}_i[\![\mathsf{E}]\!]\ (\textit{extends}\ \rho\ (\langle \mathrm{I} \rangle\ \S\ \mathrm{I}^*)\ \epsilon^*)\ \kappa')\ in\ \mathsf{E})$

The continuation is passed a λ-expression that makes a new environment by extending the old one with the actual parameters bound to the formal parameters and then evaluates the expression in the new environment. *extends* is an auxiliary function for building environments.

These are the three abstraction expressions in the intermediate language. The only difference between them is syntactic in that one identifier in the `proc` expressions is distinguished; indeed `lambda` and `cont` have identical semantics. The different abstraction forms are used to distinguish between abstractions that get passed an explicit continuation and those that are continuations. Initially the program contains only `(lambda (I*) E)` abstractions. The conversion to CPS changes these to `(proc I (I*) E)` and introduces explicit continuations `(cont (I*) E)`. The distinction between `(lambda (I*) E)` and `(cont (I*) E)` is made only for the purposes of exposition.

$\mathcal{E}_i[\![$`(C E*)`$]\!] =$
$\lambda\rho\kappa\,.\,\mathcal{E}_i[\![\mathrm{C}]\!]\rho(\lambda\epsilon\,.\,\mathcal{E}_i{}^*(\textit{permute}([\![\mathrm{E}^*]\!]))$
ρ
$(\lambda\epsilon^*\,.\,((\lambda\epsilon^*\,.\,(\epsilon \mid \mathsf{F})\ \epsilon^*\ \kappa)$
$(\textit{unpermute}\ \epsilon^*))))$

The C is evaluated followed by the E* in the order specified by *permute*. *unpermute* puts the resulting values back into the original order. The value of C is called on the values and the continuation. *permute* and *unpermute* are used to keep from specifying the order in which the arguments are evaluated and are defined in the section on auxiliary functions. The evaluation of C does not affect the store so it need not be included in the call to *permute*.

The semantics would allow any expression to be applied, but as the compiler has no default calling convention, the syntax allows only continuations here. When the procedure is a continuation no calling convention is needed, as code for the continuation can be generated in-line; there need be no actual procedure call at all.

$$\mathcal{E}_i[\![\texttt{(P ()\ E*)}]\!] =$$
$$\lambda\rho\kappa\,.\,\mathcal{E}_i\text{*}(\textit{permute}\ [\![\text{E*}]\!])$$
$$\rho$$
$$(\lambda\epsilon\text{*}\,.\,((\lambda\epsilon\text{*}\,.\mathcal{P}_i[\![\text{P}]\!]\ \epsilon\text{*}\ (\lambda\epsilon\text{*}\nu\,.\,\kappa(\epsilon\text{*}\downarrow 1)))$$
$$(\textit{unpermute}\ \epsilon\text{*})))$$

Here a primitive is applied with no continuation arguments. The value arguments are evaluated and passed to $\mathcal{P}_i$ with the primitive and a continuation. The continuation ignores the number it gets and calls the actual continuation with the first value (not getting exactly one value here is actually an error). A continuation argument will be added to these applications when the program is converted to CPS.

$$\mathcal{E}_i[\![\texttt{(P\ (C*)\ E*)}]\!] =$$
$$\lambda\rho\kappa\,.\,\mathcal{E}_i\text{*}(\textit{permute}\ [\![\text{E*}]\!])$$
$$\rho$$
$$(\lambda\epsilon\text{*}\,.\,((\lambda\epsilon\text{*}\,.\mathcal{P}_i[\![\text{P}]\!]\ \epsilon\text{*}\ \lambda\epsilon\text{*}\nu\,.\,\mathcal{E}_i([\![\text{C*}]\!]\downarrow\nu)\ \rho\ \lambda\epsilon\,.\,(\epsilon\mid\mathsf{F})\ \epsilon\text{*}\ \kappa)$$
$$(\textit{unpermute}\ \epsilon\text{*})))$$

Here the continuations to the application are specified. The number returned by $\mathcal{P}_i$ is used to pick out one of the continuation arguments. This argument is evaluated using $\mathcal{E}_i$ and the result is called on the values $\mathcal{P}_i$ returns. The continuation arguments are not passed to $\mathcal{P}_i$, in order to restrict the way in which they may be used. With the semantics as given here the continuations have known extent and a known use.

$$\mathcal{E}_i[\![\texttt{(return\ P\ E*)}]\!] =$$
$$\lambda\rho\kappa\,.\,\mathcal{E}_i\text{*}(\textit{permute}\ [\![\text{E*}]\!])$$
$$\rho$$
$$(\lambda\epsilon\text{*}\,.\,((\lambda\epsilon\text{*}\,.\mathcal{P}_i[\![\text{P}]\!]\ \epsilon\text{*}\ \lambda\epsilon\text{*}\nu\,.\,\kappa(\epsilon\text{*}\downarrow 1))$$
$$(\textit{unpermute}\ \epsilon\text{*})))$$

This semantics is identical to `(P () E*)`. As will be described later, there are requirements and restrictions on the use of `return` expressions.

$$\mathcal{E}_i[\![\texttt{(block\ E}_0\texttt{)}]\!] = \mathcal{E}_i[\![\text{E}_0]\!]$$

$$\mathcal{E}_i[\![\texttt{(block\ E}_1\text{E}^+\texttt{)}]\!] =$$
$$\lambda\rho\kappa\,.\,\mathcal{E}_i[\![\text{E}_1]\!]\ \rho\ \lambda\epsilon\,.\,\mathcal{E}_i[\![\texttt{(block\ E}^+\texttt{)}]\!]\ \rho\ \kappa$$

`block` is a sequencing construct that evaluates its expressions in order and returns the result of the last, ignoring any intermediate results.

$\mathcal{E}_i$*$[\![\ ]\!] = \lambda\rho\kappa\ .\ \kappa\langle\ \rangle$

$\mathcal{E}_i$*$[\![\mathrm{E}_0\ \mathrm{E}^*]\!] =$
$\lambda\rho\kappa\ .\ \mathcal{E}_i[\![\mathrm{E}_0]\!]\ \rho\ \lambda\epsilon_0\ .\ \mathcal{E}_i^*[\![\mathrm{E}^*]\!]\ \rho\ (\lambda\epsilon^*\ .\ \kappa\ (\langle\epsilon_0\rangle\ \S\ \epsilon^*))$

This is much like `block` except that the results are gathered into a sequence and passed to the continuation.

2.1.6 Auxiliary functions

extends : U $\to$ Ide* $\to$ E* $\to$ U
extends $=$
$\lambda\rho\mathrm{I}^*\epsilon^*\ .\ \#\mathrm{I}^* = 0 \to \rho,$
extends $(\rho[(\epsilon^* \downarrow 1)/(\mathrm{I}^* \downarrow 1)])\ (\mathrm{I}^* \dagger 1)\ (\epsilon^* \dagger 1)$

extends creates a new environment by adding the bindings of the Ide* to the E* to the given environment.

permute : E* $\to$ E* [language dependent]

unpermute : E* $\to$ E* [inverse of *permute*]

permute and *unpermute* are borrowed from the Scheme semantics in [Rees 86]. *permute* determines the order in which the arguments to a call are evaluated. Given the way these functions are used, this order may depend only on the number of arguments as that is the only information shared by *permute* and *unpermute*. Ideally this order would depend on the argument expressions themselves and the surrounding code. The simplest solution, as this is not an important point, is to assume that *unpermute* nondeterministically does the right thing in reordering the values of the arguments. In any case, when compiling a language with a specified order of evaluation for arguments, such as left-to-right, *permute* and *unpermute* are modified to follow that order.

2.2 The machine language

The machine language is an assembly language written in the syntax of the intermediate language and has a much simpler semantics. The machine is assumed to be a Von Neumann machine with a store and register-to-register instructions. Identifiers represent the machine's registers and primitive procedures are the machine's instructions. The semantics of `lambda` expressions depends upon their context. `lambda` expressions that are not continuations to calls to primitive procedures represent code pointers and their identifiers are ignored. The identifiers in a continuation to a call to a primitive procedure represent the registers in which the results of the instruction appear. As an example, here is the interpretation of a call to a primitive procedure for a two-address add instruction both in the machine language's syntax and in a conventional assembler syntax:

```
($add ((lambda (r1) ...)) r1 r2)  ⟺  add r1,r2
```

2.2.1 Abstract syntax

K ∈ Con		constants
I ∈ Ide		identifiers
P ∈ Pri		primitives
C ∈ Eco		continuations
	⟶	`(lambda (I*) A)`
A ∈ App		applications
	⟶	`(P (C*) E*)`
E ∈ Exp		expressions
	⟶	K | I | C

The machine language syntax is a restriction of the intermediate language: `(block E* E`$_0$`)`, `(proc I (I*) E)`, and `(P (C*) E*)` are not allowed. The body of a `lambda` expression must be an application. Calls to `lambda` expressions are not needed. This is a consequence of the machine language's lack of lexical scoping. In the intermediate language calls to `lambda` expressions are needed to bind the values of `lambda` expressions to identifiers as the expressions cannot be duplicated without increasing the size of the program. In the machine language the value of a `lambda` expression depends only on the expression itself and not on its lexical context. The `lambda` expressions represent pointers to code and are thus a type of constant and can be duplicated freely.

2.2.2 Domain equations

$\nu \in \mathsf{N}$	natural numbers
$\alpha \in \mathsf{L}$	locations
M	miscellaneous
$\mathsf{F} = \mathsf{C}$	procedure values
$\epsilon \in \mathsf{E} = \mathsf{L} + \mathsf{M} + \mathsf{F}$	expressed values
$\sigma \in \mathsf{S} = (\mathsf{L} + \mathrm{Ide}) \to \mathsf{E}$	stores
$\theta \in \mathsf{C} = \mathsf{S} \to \mathsf{S}$	command continuations
$\phi \in \mathsf{G} = \mathsf{E}^* \to \mathsf{N} \to \mathsf{C}$	primitive continuations

There are no environments and no expression continuations. Procedures no longer take actual parameters or continuations. Values in the machine are passed to procedures by side-effecting the store. The store now includes values for identifiers.

2.2.3 Semantic functions

$\mathcal{K}_m : \mathrm{Con} \to \mathsf{E}$
$\mathcal{P}_m : \mathrm{Pri} \to \mathsf{E}^* \to \mathsf{G} \to \mathsf{C}$
$\mathcal{A}_m : \mathrm{App} \to \mathsf{C}$
$\mathcal{L}_m : \mathrm{Eco} \to \mathsf{E}^* \to \mathsf{C}$
$\mathcal{E}_m : \mathrm{Exp} \to \mathsf{S} \to \mathsf{E}$
$\mathcal{E}_m{}^* : \mathrm{Exp}^* \to \mathsf{S} \to \mathsf{E}^*$

Unlike their intermediate language counterparts $\mathcal{K}_m$, $\mathcal{P}_m$, $\mathcal{E}_m$, and $\mathcal{E}_m{}^*$ do not take a continuation as an argument and simply return values. $\mathcal{E}_m$ and $\mathcal{E}_m{}^*$ do not have environment arguments but do require the store in which to look up the values of identifiers. $\mathcal{A}_m$ has been split off from $\mathcal{E}_m$ as applications are no longer expressions. $\mathcal{L}_m$ is called on the continuations to calls to primitives.

The definition of $\mathcal{K}_m$ is again deliberately omitted.

$$\mathcal{E}_m[\![\mathrm{K}]\!] = \lambda\sigma \,.\, \mathcal{K}_m[\![\mathrm{K}]\!]$$

$$\mathcal{E}_m[\![\mathrm{I}]\!] = \lambda\sigma \,.\, \sigma\ \mathrm{I}$$

Identifiers get their values from the store as they now represent side-effectable registers.

$\mathcal{E}_m[\![\texttt{(lambda (I*) A)}]\!] = \lambda\sigma\ .\ (\mathcal{A}_m[\![\text{A}]\!] in \text{E})$

A procedure simply calls $\mathcal{A}_m$ on its body. The identifiers and the store are ignored.

$\mathcal{A}_m[\![\text{(P () E*)}]\!] =$
$\quad \lambda\sigma\ .\ \mathcal{P}_m[\![\text{P}]\!]\ (\mathcal{E}_m\text{*}[\![\text{E*}]\!]\ \sigma)(\ \lambda\epsilon\text{*}\nu\sigma\ .\ \sigma)\ \sigma$

Permute and *unpermute* are not needed here or in the following as the arguments to procedures cannot be applications and so cannot affect the store; thus the order of evaluation of arguments has no effect on the meaning of the program. The values the primitive returns are ignored.

$\mathcal{A}_m[\![\text{(P (C}^+\text{) E*)}]\!] =$
$\quad \lambda\sigma\ .\ \mathcal{P}_m[\![\text{P}]\!]\ (\mathcal{E}_m\text{*}[\![\text{E*}]\!]\ \sigma)(\ \lambda\epsilon\text{*}\nu\sigma\ .\ \mathcal{L}_m([\![\text{C}^+]\!] \downarrow \nu)\ \epsilon\text{*}\ \sigma)\ \sigma$

The returned values and the selected continuation are passed to $\mathcal{L}_m$. In the intermediate language's semantics the continuation is evaluated using $\mathcal{E}_m$ and then called on the values.

$\mathcal{L}_m[\![\texttt{(lambda (I*) A)}]\!] =$
$\quad \lambda\epsilon\text{*}\ .\ \lambda\sigma\ .\ \mathcal{A}_m[\![\text{A}]\!]\ (\textit{extends}\ \text{I*}\ \epsilon\text{*}\ \sigma)$

A new store is made with the identifiers I* having the values ϵ*. Note that this is different from the value of a `lambda` as an expression; in that case the identifiers are ignored as shown in the definition of $\mathcal{E}_m$. The body of the `lambda` expression is evaluated with the new store.

$\mathcal{E}_m\text{*}[\![\ ]\!] = \lambda\sigma\ .\ \langle\ \rangle$

$\mathcal{E}_m\text{*}[\![\text{E}_0\ \text{E*}]\!] =$
$\quad \lambda\sigma\ .\ \langle \mathcal{E}_m[\![\text{E}_0]\!]\ \sigma\ \rangle\ \S\ \mathcal{E}_m\text{*}[\![\text{E*}]\!]\ \sigma$

The store is used here instead of the environment. There are no continuations as the values can just be returned.

2.2.4 Auxiliary functions

$\textit{extends} : \mathsf{S} \to \text{Ide*} \to \text{E*} \to \mathsf{S}$
$\textit{extends} =$
$\quad \lambda\sigma\text{I*}\epsilon\text{*}\ .\ \#\text{I*} = 0 \to \sigma,$
$\qquad\qquad \textit{extends}\ (\sigma[(\epsilon\text{*} \downarrow 1)/(\text{I*} \downarrow 1)])\ (\text{I*} \dagger 1)\ (\epsilon\text{*} \dagger 1)$

extends is exactly the same as before except that it now works on stores instead of environments.

Chapter 3

Front Ends

In order to use the transformational compiler a front-end is needed to translate programs into the compiler's intermediate language. Different front-ends are needed for different programming languages. Translations must be provided for three different aspects of the source language:

1. Control constructs
2. Primitive data structures
3. Primitive operations

The ease of producing a front-end for a particular language is dependent on the simplicity of its primitive data structures and operations (ignoring any difficulties that lexical or syntactic analysis may present). The control constructs can be translated using a simple syntax-directed translator very like a denotational semantics for the source language. This part of the translation could be derived directly from a denotational semantics for the source language, much as is done by semantics based compiler generators. The translator consists of a template, either written by hand or automatically generated, for every type of expression in the syntax of the source language that gives an equivalent expression in the intermediate language in terms of the translations of the expression's subexpressions. The only requirement is that the control flow be easily implemented in the lambda calculus without direct manipulation of continuations; writing a translator for a logic programming language, or an object-oriented one, would be quite difficult.

The translation of primitive data structures and operations is more difficult as it requires knowledge of the target machine architecture and is typically not dealt

with in denotational semantics. As an obvious example, while most programming languages contain integers as a data type and most machines implement operations on integers similar to those used in the programming languages, the matchup is often not exact. The machine's integers may not be large enough, or some operations may be missing. Translating more complex domains, such as string manipulations, or dealing with run-time requirements, such as tagged data types, can be quite complex. Producing a translator for a language and machine pair requires designing data structures and instruction sequences appropriate to both. Argument passing and value returning mechanisms are included here as they are dependent both on the language being compiled and the target machine.

This is not to say that writing a front-end is necessarily a lot of work. Many languages (and machines) have similarities that may be exploited. For example, the same set of numerical primitive operations and data structures could be used for FORTRAN and Pascal. The same calling convention could be used on many different machines for many different languages. Modifications to existing front-ends, such as adding new control constructs or new primitive data types and operations is quite simple.

The rest of this chapter covers in more detail the initial translation into the intermediate language. Language specific optimizations are discussed in subsection 6.1.2 and compiler modifications in Chapter 9.

3.1 Identifiers and syntax

There are two constraints on identifiers in the output of the front-end: no identifier may be bound more than once in a program and every identifier must have an associated size indicating the size of the values to which it will be bound when the program is run. The sizes used in examples will be either 1) `ptr`, indicating that that the identifier represents a machine pointer, 2) a number of bits, or 3) `state`, indicating that the identifier stands for the current contents of the registers and any arguments pushed on the stack. When needed, the size of an identifier will be written after the identifier, separated by a colon, such as `x:16` for a sixteen bit value.

The front-end may not use either `(proc I (I*) ...)` or `(cont I (I*) ...)`. These are introduced (and `(lambda (I*) ...)` removed) during the conversion to CPS as detailed in Chapter 5.

3.2 Primitive procedures

As the compiler has no built-in data or procedure constants the front-end must include a mapping from source language constants to machine data and a set of primitive procedures that includes all of the machine operations that the source language requires. The primitive procedures and the descriptions of constants are not normally specified in denotational semantics but are necessary to compile programs. For example, the denotational description of Scheme in [Rees 86] does not specify the the semantics of `17` or `+` but a Scheme compiler must somehow translate them into machine data and instructions, and thus a front-end for Scheme must include these translations. This makes the front-ends machine dependent, in that an operation such as `+` may mean different things on different machines as well as in different languages.

Primitive procedures must provide all of the information that the compiler needs to generate code to execute the instructions. This information includes:

1. The set of registers the operation requires
2. A description of the operation's interaction with the store
3. Code improving transformations specific to the operation
4. Code to generate machine instructions
5. The size of the value returned by the primitive, if the primitive is called without any explicit continuations

In the current implementation primitive operations are created using an object-oriented programming extension to Scheme provided in T [Rees 84]. As an example, presented only to show the amount of code needed to specify a simple primitive, here is the code for the sixteen bit add instruction on the Motorola 68020 as used by the Pascal front-end.

```
(define-primop add16
  ((primop.arg-specs self)
   '((reg 2 1) (any 3 #f)))
  ((primop.generate self call block)
   (destructure (((#f x y) (call-arguments call)))
     (emit block
      (add 'w (->ea y) (ea/r x)))))
  ((primop.simplify self node)
   (simplify-integer-add node))
  ((primop.value-size self) '(16)))
```

The `add16` primitive does not use the store (which is the default). It requires one register which must contain its second argument (the first is its continuation). The third argument can be in any register or in memory. The first (and only) result appears in the register that contained the second argument. There is code for emitting an add-word instruction using the register of the second argument and the location of the third. Calls may be simplified using the procedure `simplify-integer-add` which performs constant folding. Finally, the result of the `add16` primitive is a sixteen bit value. Primitives that do not return an interesting value specify a return size of zero.

Conditional expressions are implemented using primitives that have more than one continuation. For example, `IF` in Pascal can be implemented using a primitive `$if` (in the examples primitive procedure names will begin with a '`$`') that calls its first continuation if its argument is true and its second otherwise.

```
IF test THEN do-true-thing ELSE do-false-thing
  ⟹
($if ((cont () do-true-thing)
      (cont () do-false-thing))
     test)
```

A comparison operation such as `<` could be implemented either as primitive returning a boolean value or as a conditional primitive. Conditional primitives may be used to return boolean values by using value-returning continuations.

```
x < y
  ⟹
($less-than ((lambda () 'true) (lambda () 'false)) x y)
```

3.3 Locations

Locations are pointers into the store. New ones are created using either `$push` or `$allocate`. Both of these take one argument, which is the amount of memory needed. Locations created using `$allocate` have indefinite lifetimes; those created using `$push` may not be referenced after the program returns from the `lambda` expression immediately surrounding the call to `$push`.

Values are stored in locations using `$set-contents` which takes four arguments: a location, a size, an offset within the location, and a value. The value, which must be of the specified size, is stored in the location at the offset. `$set-contents` returns no value (actually, it returns a value of size zero as returning no value would require the front-end to supply an explicit continuation). Values are retrieved from locations using `$contents` which takes the same arguments as `$set-contents` but without the value. `$contents` returns the value found at the offset in the location.

3.4 Global environment

Free identifiers are not allowed in the intermediate language. Programs receive values from and provide values to the global (external) environment through a location that is passed as an argument to the program. The loader constructs this location at runtime using data the compiler has included in the object file. The front-end needs to supply the compiler with a description of the global environment as used by the program. A `(lambda (globals:ptr) ...)` is wrapped around the program to receive the global environment. The program treats the global environment as a location, storing and retrieving the values of global variables at fixed offsets within it.

3.5 Procedure call and return

The semantics of the intermediate language allows only calls to primitives and `lambda` (or the identical `cont`) expressions. Other values cannot be called directly as there is no default calling convention and the compiler would not know how to generate code for the call. Calls to values are done using primitives that specify the calling and return conventions. These primitives specify which of their arguments is being called. They also have simplifying transformations that are applied

if the compiler determines that the value called will be the value of a particular abstraction expression and that all calls to that value call only that value. The compiler can then use whatever calling convention is most appropriate for both the expression whose value is called and the calls to that value. In the (Scheme) example below the calls to g can be easily identified and only g is called at those calls; thus the compiler can use a special calling sequence for g. The only expression that calls f can also be identified, but f is not the only procedure called at that point so f must use a calling convention specified by the Scheme implementation.

```
(lambda (a)
  (let ((f (lambda () 1))
        (g (lambda (x) (x))))
    (g a)
    (g f)))
```

The flip side of not having a default calling convention is having no default return convention. Every `lambda` expression that is not a continuation to a primitive call must return using `(return P E*)` expressions. That is, every execution path through a procedure must end with a `return`. The primitive in the `return` expression gives the protocol to be used in returning from the enclosing (non-continuation) `lambda` expression.

In the examples used in this dissertation all primitives that call a value will have `call` in their names if they have a continuation argument and `return` if they do not. Each calls its first non-continuation argument.

The compiler uses three primitive operations to indicate calls to known abstractions. These call either the values of `proc` or `cont` expressions as they are introduced after CPS conversion when all `lambda` expressions have been replaced. `$call` is used for calls to the values of known `proc` expressions, `$return` is used for calls to continuations to `$call` calls, and `$jump` (an exception to the above naming convention) for calls to the values of other `cont` expressions.

3.6 Recursion

In the intermediate language recursive procedures can be implemented using locations. A new location is created and bound to an identifier which is lexically apparent to the procedure. The procedure is then stored in the location, after which the procedure can reference itself by dereferencing the location. This method is used as it is both general enough to implement `goto` (although it cannot be used

to implement non-local returns) and it is natural to implement in the machine language.

```
((lambda (proc:ptr)
   ($set-contents () proc:ptr 'ptr '0 (lambda ...))
   ...)
 ($allocate () 'ptr))
```

Since every procedure must have its own explicit `return` statements these procedure calls cannot be tail recursive. The best that can be accomplished is to return immediately after the procedure call.

```
(block ($proc-call () some-procedure)
       (return $proc-return))
```

There is a potential problem when iteration is being expressed using a recursive procedure where the recursive call is not tail recursive. This may cause a continuation stack overflow at runtime as each iteration adds another continuation to the stack. There are two separate transformations in the compiler that address this problem. The substitution of known values for identifiers may remove the troublesome continuation; alternatively, the call may be made properly tail recursive if the continuation turns out to be unnecessary. These transformations will cause any simple iteration within the body of a single source procedure to be converted into iterative machine code, but will not work for all tail-recursive calls. Thus to implement Scheme, where calls that are tail-recursive in the source must be implemented without using any finite resource, additional work must be done (see Chapter 9).

3.7 Pascal examples

To start off, I will translate a small Pascal program into the intermediate language.

```
PROGRAM Small;
  VAR x : integer;
  BEGIN
    Read(x);
    x := x + 1;
    Write(x);
  END.
```

First, a location is needed to hold the value of `x`. Pascal was designed in such a way that locations do not escape upwards and can be allocated on a stack. This

allows $push to be used to allocate the location for x.

```
((lambda (x:ptr)
   (block ...))
 ($push () '16))
```

$push is passed the number of bits that the location needs to contain and returns the location, which is a pointer. Pascal integers are assumed here to be sixteen bits long. Note that x is of size ptr and not size 16 as x represents the location of the value and not the value itself.

Assuming that the primitive $read reads and returns a sixteen bit integer the first statement can be translated as follows:

```
Read(x);     ⟹   ($set-contents () x '16 '0 ($read ()))
```

The arguments to $SET-CONTENTS are a location, a size in bits, an offset, and a value. The value, which should be of the specified size, is stored in the location starting at the offset. The next statement dereferences x, adds one to the value, and stores it back into the location for x:

```
x := x + 1;  ⟹ ($set-contents () x '16 '0
                ($add16 () ($contents () x '16 '0) '1))
```

$contents takes the same size and offset information as $set-contents. $add16 is a primitive that adds two sixteen bit numbers and returns the result. The last statement writes out the value of x using a primitive $write:

```
Write(x);    ⟹ ($write () ($contents () x))))
```

Putting it all together gives:

```
((lambda (x:ptr)
   (block ($set-contents () x '16 '0 ($read ()))
          ($set-contents () x '16 '0
            ($add16 () ($contents () x '16 '0) '1))
          ($write () ($contents () x))))
 ($push () '16))
```

The Pascal read procedure actually has an implicit file variable for the standard input. The $read primitive needs the actual value of standard input, which is not defined in the file and so must be obtained from the global environment. In the final version of the program $read and $write are passed the appropriate input and output streams obtained from the global environment indexed by the constants std-in and std-out.

```
(lambda (globals:ptr)
  ((lambda (x:ptr)
     (block ($set-contents () x '16 '0
              ($read () ($contents () globals 'ptr 'std-in)))
            ($set-contents () x '16 '0
              ($add16 () ($contents () x '16 '0) '1))
            ($write () ($contents () x)
                       ($contents () globals 'ptr 'std-out))))
   ($push () '16)))
```

3.7.1 Data structures

Compound data structures can be implemented using the primitives already introduced. Records and arrays are created using `$push` or `$allocate`, set using `$set-contents` and accessed using `$contents`.

```
a : array [1..10] of integer;
  ⟹
($push () '160)
```

Accessing a field of a record is simply a matter of specifying the appropriate offset. Accessing an array requires adjusting the index by the array's lower bound (one in this case) and converting to bytes for `$contents` and `$set-contents`.

```
a[i] := 50
  ⟹
($set-contents () a '16
  ($multiply16 () '2
                ($add16 () '-1
                         ($contents () i '16 '0)))
  '50)
```

3.7.2 Compound statements

`IF` can be implemented using a primitive `$if` that has one argument and two continuations. The first continuation is called if the argument is true and the second if it is false.

```
IF exp THEN statement1 ELSE statement2;
   ⟹
($if ((lambda () statement1) (lambda () statement2)) exp)
```

with `statement1` and `statement2` translated and `exp` translated and dereferenced.

Loops are implemented using recursive procedures. Calling the procedure and returning from it require special primitives as described above. `$simple-call` and `$simple-return` are primitives for calling and returning from procedures that take no arguments and return no values.

```
WHILE exp DO statement;
⟹
((lambda (proc:ptr)
   (block ($set-contents () proc 'ptr '0
            (lambda ()
              ($if ((lambda ()
                      (block statement
                             ($simple-call ()
                              ($contents () proc 'ptr '0))
                             (return $simple-return)))
                    (lambda () (return $simple-return)))
                   exp)))
          ($call () ($contents () proc 'ptr '0))))
 ($push () 'ptr))
```

3.7.3 Procedures and functions

Pascal procedure and function calls are handled by primitives that package up the arguments to be passed as a single value. Within the procedure another primitive breaks the argument up into the original values. In Pascal arguments may be passed either by value or by reference. A simple implementation method is to pass all arguments by reference and have the called procedure copy the values that should have been passed by value. A primitive `$copy` can be used to copy data from one location to another.

```
PROCEDURE Frog (x : integer; VAR y : integer); ...
⟹
(lambda (all:state)
  ($unpack-call ((lambda (x:ptr y:ptr)
                   ((lambda (x1:ptr)
                      (block
                        ($copy () x x1 '16)
                        ...))
                    ($push () '16))))
                 '<ptr ptr>
                 all))
```

<ptr ptr> is a constant specifying the particular argument passing protocol to be used. In this case two pointer values are being passed as arguments. Giving the parameter types as an argument to the primitive call allows the front-end to use only one procedure-call primitive for many different procedures. The primitive uses the type argument to determine exactly how the parameters are to be passed to the procedure. For example, x and y may be passed in two specific registers, or in the first two stack locations, or whatever. The procedure returns using $simple-return as Pascal procedures return no values. If Frog were a Pascal function returning an integer it could use

```
($return-with-values () '<16> z:16)
```

to return one sixteen bit result. Calling the function and getting the return value is done as follows:

```
($call-with-values-and-return () proc:ptr
                                 '<<ptr ptr> <16>>
                                 a:ptr
                                 b:ptr)
```

where $call-with-values-return handles both calling the procedure and getting the returned value. <<ptr ptr> <16>> is a constant containing both the types of arguments passed and the values returned.

The scoping of identifiers for variables, procedures, and functions within the code for a Pascal block follows their scoping in the language definition. Procedures and functions in Pascal may be recursive, so locations are introduced for all procedures and functions before any of the values themselves appear in the program. The translation of a block would look something like this:

```
((lambda (var1 var2 ...)
   ((lambda (proc1 proc2 ...)
      (block ($set-contents () proc1 'ptr '0
                             first-procedure)
             ($set-contents () proc2 'ptr '0
                             second-procedure)
             ...
             {the body of the program}
             ))
    ($push () 'ptr) ($push () 'ptr) ...))
 ($push () size1) ($push () size2) ...)
```

3.8 Factorial in Pascal

Here is a simple Pascal program that will be used to demonstrate the entire process of compilation. It reads in a positive integer x and prints out x! $= 1 * 2 * \ldots * x$.

```
PROGRAM Fact;
  VAR x, r : integer;
  PROCEDURE Fact(n : integer; VAR res : integer);
    VAR i, r : integer;
    BEGIN
      r := 1;
      FOR i := 1 TO n DO
          r := r * i;
      res := r
    END;
BEGIN
  Readln(x);
  Fact(x, r);
  Writeln(r)
END.
```

This becomes a very large program in the intermediate language, shown here in three parts. The first is the body of the program which introduces locations for the variables x and y and the procedure Fact, reads a value for x, calls Fact, and writes out the value of r.

The second part is the procedure Fact (which is represented in the first part by

<FACT>). It gets the values of its arguments, copies the value of n, which is passed by value, makes locations for r and i, sets up and calls a recursive procedure for the FOR loop, sets the value of r, and returns. Note that some new identifiers have been introduced so that each identifier is bound only once.

The recursive procedure is shown in the third part. It compares i and n1, and if they are equal the procedure returns, otherwise i and r are set to their new values and the procedure calls itself.

For clarity, the size and offset arguments to $contents and $set-contents are not shown here.

```
(lambda (global:ptr)
  ((lambda (x:ptr r:ptr)
     ((lambda (fact:ptr)
        (block
          ($set-contents () fact <FACT>)
          ($set-contents () x ($read () ($contents () global '<si>)))
          ($read-line () ($contents () global '<si>))
          ($proc-call () ($contents () fact) x r '<ptr ptr>)
          ($write () ($contents () r) ($contents () global '<so>))
          ($write-line () ($contents () global '<so>))
          (return $simple-return)))
      ($push () 'ptr)))
   ($push () '16)
   ($push () '16)))
```

```
<FACT> =
(lambda (args:all)
  ($get-args
   ((lambda (n:ptr res:ptr)
      ((lambda (n1:ptr)
         (block
           ($copy () n n1 '16)
           ((lambda (i:ptr r1:ptr)
              (block
                ($set-contents () r1 '1)
                 ((lambda (loop:ptr)
                    (block ($set-contents () loop <LOOP>)
                           ($set-contents () i '1)
                           ($simple-call () ($contents () loop))))
                  ($push () 'ptr))
                ($set-contents () res ($contents () r1))
                (return $simple-return)))
            ($push () '16)
            ($push () '16))))
       ($push () '16))))
   '<ptr ptr>
   args))

<LOOP> =
(lambda ()
  ($equal16 ((lambda ()
               (return $simple-return))
             (lambda ()
               (block
                 ($set-contents () r1
                   ($multiply16 () ($contents () r1)
                                   ($contents () i)))
                 ($set-contents () i
                   ($add16 () ($contents () i) '1))
                 ($simple-call () ($contents () loop))
                 (return $simple-return))))
            ($contents () i)
            ($contents () n1)))
```

Chapter 4

Ordering Calls

4.1 Making code linear

The first step of the actual compilation is to give an explicit order to the applications in the program. At the same time identifiers are introduced for all temporary values and all `block` expressions are removed. A global transformation does all of this, producing a transformed program in which the arguments to applications are, with one exception, never applications themselves, but either constants, identifiers, or `lambda` expressions. The one exception is calls of the form `((lambda (x) ...) A)`; calls to `lambda` expressions with only one argument may have an application as the argument.

As an example of the transformation to linear code, in the expression `($call () f ($call g x))` the result of the call to g is an anonymous temporary value. The transformation converts this expression into `((lambda (v) ($call () f v)) ($call g x))` where `v` is the identifier introduced for the result of the call to g. If more than one temporary is needed they are bound by separate `lambda` expressions. This has the effect of specifying the order in which the calls that produce the temporary values are executed.

```
($call () f ($call () g x) ($call () h y))
  ⟹
((lambda (v1)
   ((lambda (v2) ($call () f v1 v2))
    ($call () h y)))
 ($call () g x))
```

Here the call to g is done before the call to `f`.

`Block` expressions are removed by introducing temporary identifiers to hold the values returned by the non-final expressions in the block. These introduced identifiers are not referenced and so the values that the intermediate expressions return are ignored.

```
(block ($call () f) ($call () g))
  ⟹
((lambda (v1) ($call () g)) ($call () f))
```

The `lambda` expressions the transformation introduces correspond exactly to the expression continuations used by the semantic function $\mathcal{E}_i$* and by $\mathcal{E}_i$ for `block` expressions. In interpreting the linear code these expression continuations are used only ephemerally (or not at all) as `block` no longer appears in the program and arguments to applications can be evaluated trivially, as in the machine language.

4.2 The transformation

lin : Exp $\rightarrow$ Exp

lin $[\![\mathrm{K}]\!] = [\![\mathrm{K}]\!]$

lin $[\![\mathrm{I}]\!] = [\![\mathrm{I}]\!]$

Constants and identifiers remain unchanged.

lin $[\![$`(lambda (I*) E)`$]\!] = [\![$`(lambda (I*) E′)`$]\!]$
where: $\mathrm{E}' =$ *lin*$([\![\mathrm{E}]\!])$

`lambda` expressions have their bodies transformed.

lin $[\![$`(P (`$\mathrm{C}_0 \ldots \mathrm{C}_n$`) `$\mathrm{E}_0 \ldots \mathrm{E}_m$`)`$]\!] =$

```
[[((lambda (x0)
    ((lambda (x1)
       ...
       ((lambda (xm)
          (P (C′0 ...C′n) y0 ...ym))
        E′m)
       ...)
     E′1)
```

$\mathrm{E}'_0)[\![$
where: $\langle \mathrm{E}'_0 \ldots \mathrm{E}'_m \rangle = \mathit{permute}(\mathit{lin}(\mathrm{E}_0) \ldots \mathit{lin}(\mathrm{E}_m))$
$\mathrm{C}'_i = \mathit{lin}(\mathrm{C}_i)$
$\mathtt{x}_i$ is a new identifier with size $\mathit{size}(\mathrm{E}'_i)$
$\langle \mathtt{y}_0 \ldots \mathtt{y}_m \rangle = \mathit{unpermute}(\mathtt{x}_0 \ldots \mathtt{x}_m)$

New `lambda` expressions are introduced to bind the arguments to applications. The arguments are evaluated in the order specified by the procedure *permute* from the semantics. *size*, which is defined below, determines the size of the value returned by an expression. The current implementation of the compiler does not introduce `lambda` expressions to bind the values of arguments that are not applications as the new expressions would be immediately removed again during the code improvement phase.

`(return P E*)` and `((lambda (...) E) ...)` are transformed in exactly the same fashion as calls to primitives. The procedure in `((lambda (...) E) ...)` is transformed but no additional binding is introduced for it.

lin $[\![$`(block E)`$]\!] = \mathit{lin}([\![\mathrm{E}]\!])$

A `block` with one expression is the same as the expression.

lin $[\![$`(block E` E^+`)`$]\!] =$ `((lambda (x)` E''`)` E'`)`
where: $[\![\mathrm{E}']\!] = \mathit{lin}([\![\mathrm{E}]\!])$
$[\![\mathrm{E}'']\!] = \mathit{lin}([\![$`(block` E^+`)`$]\!])$
`x` is a new identifier with size $\mathit{size}(\mathrm{E}')$

A `lambda` expression is introduced to bind the value of the first expression in the block to an identifier that is afterwards ignored.

The function *size* is used above to determine the size of value returned by expressions. *size* is defined as follows:

size $[\![\mathrm{I}]\!] = \mathit{identifiersize}([\![\mathrm{I}]\!])$
size $[\![\mathrm{K}]\!] = \mathit{constantsize}([\![\mathrm{K}]\!])$
size $[\![$`(lambda ...)`$]\!] =$ `ptr`
size $[\![$`(block E* E)`$]\!] = \mathit{size}([\![\mathrm{E}]\!])$
size $[\![$`(P (...) E*)`$]\!] = \mathit{primitivevaluesize}([\![\mathrm{P}]\!], [\![\mathrm{E}^*]\!])$
size $[\![$`((lambda (...) E) ...)`$]\!] = \mathit{size}([\![\mathrm{E}]\!])$

identifiersize and *constantsize* return the sizes of identifiers and constants. `lambda` expressions are all pointers. *primitivevaluesize* returns the size of the value returned by a call to a particular primitive. It is passed the arguments to the call for the benefit of primitives such as `$contents` that return different sizes of values depending on their arguments.

size should never find a `return` statement as it is originally called only on expressions that are arguments to calls or non-final expressions in blocks, which thus cannot end with a return from a procedure.

4.3 Factorial example

Here is result of calling *lin* on the `Fact` example of the previous chapter.

The `LET*` syntax used here differs slightly from Scheme's to allow for calls to return more than one value. This will be needed at the end of the next section. Each clause has a list of identifiers and a list of expressions instead of only one of each. In the linear code if there is more than one expression in a clause, none of the expressions may be calls. If there is only one expression and it is a call then the identifiers are bound to the values returned by the call. Otherwise, each identifier is bound to the value of the corresponding expression.

The names of the introduced identifiers correspond to the size of the value they will be bound to at run-time: `p` for `ptr`, `t` for `16`, and `x` for `0`.

```
(lambda (global)
  (let* (((p.0) ($push () '16))
         ((p.1) ($push () '16))
         ((x r) p.0 p.1)
         ((p.2) ($push () 'ptr))
         ((fact) p.2)
         ((x.3) ($set-contents () fact <FACT>))
         ((x.4) (let* (((t.14)
                        (let* (((t.15) ($contents () global '(si))))
                          ($read () t.15))))
                  ($set-contents () x t.14)))
         ((x.5) (let* (((p.13) ($contents () global '(si))))
                  ($read-line () p.13)))
         ((x.6) (let* (((p.12) ($contents () fact)))
                  ($proc-call () p.12 x r '(ptr ptr))))
         ((x.7) (let* (((t.10) ($contents () r))
                       ((p.11) ($contents () global '(so))))
                  ($write () t.10 p.11)))
         ((x.8) (let* (((t.9) ($contents () global '(so))))
                  ($write-line () t.9))))
    (return $simple-return)))

<FACT> =
(lambda (args)
  ($get-args (<FACT-BODY>) '(ptr ptr) args))
```

```
<FACT-BODY>=
(lambda (n res)
  (let* (((p.16) ($push () '16))
         ((n1) p.16)
         ((x.17) ($copy () n n1 '16))
         ((p.18) ($push () '16))
         ((p.19) ($push () '16))
         ((i r1) p.18 p.19)
         ((x.20) ($set-contents () r1 '1))
         ((x.21) (let* (((p.24) ($push () 'ptr))
                        ((loop) p.24)
                        ((x.25) ($set-contents () loop <LOOP>))
                        ((x.26) ($set-contents () i '1))
                        ((p.27) ($contents () loop)))
                   ($simple-call () p.27)))
         ((x.22) (let* (((t.23) ($contents () r1)))
                   ($set-contents () res t.23))))
    (return $simple-return)))

<LOOP> =
(lambda ()
  (let* (((t.28) ($contents () i))
         ((t.29) ($contents () n1)))
    ($equal16 (<TRUE> <FALSE>) t.28 t.29)))

<TRUE> =
(lambda ()
  (return $simple-return))

<FALSE> =
(lambda ()
  (let* (((x.30) (let* (((t.36) (let* (((t.37) ($contents () r1))
                                       ((t.38) ($contents () i)))
                                  ($multiply16 () t.37 t.38))))
                   ($set-contents () r1 t.36)))
         ((x.31) (let* (((t.34) (let* (((t.35) ($contents () i)))
                                  ($add16 () t.35 '1))))
                   ($set-contents () i t.34)))
         ((x.32) (let* (((p.33) ($contents () loop)))
                   ($simple-call () p.33))))
    (return $simple-return)))
```

Chapter 5

Continuation Passing Style

5.1 Converting code into CPS

The transformation to linear code moved many of the continuations of the semantics into the program but did not make them actual continuations. As an example, given the source expression `($p1 () ($p2 () x y) z)` (and considering only the application of `$p2`) the linear code is:

```
((lambda (t) ($p1 () t z)) ($p2 () x y))
```

The transformation to CPS moves the `lambda` expression inside the application as a continuation and replaces `$p2` with a similar primitive that calls the new continuation; the result is:

```
($p2' ((lambda (t) ($p1 () t z))) x y)
```

At the same time identifiers are introduced for the currently anonymous continuations passed to `lambda` expressions. This allows the `return` expressions to be replaced with calls to the new continuation identifiers.

In the linear program every primitive application either has one or more continuation arguments or there is a corresponding `lambda` expression that binds the result of the application. This is because the transformation to linear code introduced identifiers for the values of all arguments to applications and non-final expressions in blocks, which is in fact all primitive applications, as in the source program only `return` expressions may return values from `lambda` expressions. To convert the program into continuation passing style the `lambda` expression that binds the result of an application is made a continuation argument to the application. The primitive procedure being called is replaced with one that is identical except that it calls its continuation argument rather than returning a value.

`lambda` expressions that are continuation arguments or procedures in applications are changed to `cont` expressions with the same identifiers and body. All other `lambda` expressions are changed to `proc` expressions with an additional identifier added for the continuations. `return` expressions are replaced with applications that call the continuation identifier of the enclosing `proc` expression. Primitive procedures used to perform procedure calls are replaced with versions that pass a continuation to the procedure being called instead of having it return a value.

The result of the transformation is a program that is in continuation passing style: expressions no longer return values and every application finishes by calling a continuation. All of the expression continuations of the semantics have been translated into the program. The expression continuations are no longer needed in that the program could be interpreted as being in a language which had the environments of the intermediate language but not its continuations. As the transformation into CPS has not changed the meaning of the program in the intermediate language there is no need to actually define this third language.

5.2 The transformation

Conversion to continuation passing style is a global transformation. Unlike *lin* it is correct only as a global transformation in that the entire program must be transformed at the same time. In addition to an expression, the transformation requires a continuation, which is either an identifier bound to a `cont` expression or a `cont` expression itself, and a boolean value that specifies whether the continuation is a known `cont` expression. Continuation identifiers of `proc` expressions are unknown continuations, as there is no way in general to identify the continuation that they will represent at run-time, and thus they can only be called using a primitive supplied by a `return` expression. All other continuations are considered known. For any known continuation it is easy to identify the applications that may call it and to determine that only that continuation will be called at those applications. These applications use the `$return` primitive mentioned in Section 3.5 to call the continuation.

The transformation uses two auxiliary functions, *value* to convert expressions that are known not to be applications and *convert-cont* to convert continuation arguments.

$$\begin{aligned} \textit{convert} &: \text{Exp} \to \text{Exp} \to \{\textit{true}, \textit{false}\} \to \text{Exp} \\ \textit{value} &: \text{Exp} \to \text{Exp} \\ \textit{convert-cont} &: \text{Exp} \to \text{Exp} \to \{\textit{true}, \textit{false}\} \to \text{Exp} \end{aligned}$$

value $[\![\text{K}]\!] = [\![\text{K}]\!]$

value $[\![\text{I}]\!] = [\![\text{I}]\!]$

Constants and identifiers are unchanged.

value $[\![$`(lambda (I*) E)`$]\!] = [\![$`(proc c (I*) E`$')]\!]$
where: `c` is a new identifier with size `ptr`
$\text{E}' = \textit{convert}([\![\text{E}]\!], [\![\texttt{c}]\!], \textit{false})$

`lambda` expressions are converted to `proc` expressions with the body converted using the new identifier as the continuation. The continuation is unknown as it is not bound to a particular `cont` expression.

convert-cont $[\![$`(lambda (I*) E)`$]\!]\ [\![\text{E}_c]\!]$ *known?* =
$[\![$`(cont (I*) E`$')]\!]$
where: $\text{E}' = \textit{convert}([\![\text{E}]\!], [\![\text{E}_c]\!], \textit{known?})$

convert-cont propagates the continuation down into the body of the `lambda` expression. It is only used on continuation arguments to primitive applications and these are always `lambda` expressions.

convert $[\![\text{K}]\!]\ [\![\text{E}_c]\!]$ *true* =
$[\![$`($return () E`$_c$ `E`$')]\!]$
where: $\text{E}' = \textit{value}([\![\text{K}]\!])$

Constants are passed to the continuation using the primitive `$return`. The continuation must be known. This transformation is also used for identifiers and `lambda` expressions.

convert $[\![$`((lambda (I) E) A)`$]\!]\ [\![\text{E}_c]\!]$ *known?* =
$\textit{convert}([\![\text{A}]\!], [\![$`(cont (I) E`$')]\!], \textit{true})$
where: $\text{E}' = \textit{convert}([\![\text{E}]\!], [\![\text{E}_c]\!], \textit{known?})$

For an application that binds the result of a call (the one place that applications may be arguments in the linear code) the `lambda` expression is changed to a `cont` and is used as the continuation to the argument. The continuation itself is propagated in to the body of the procedure (now a continuation).

convert $[\![$`((lambda (I*) E) E`$_0$ `...E`$_n$`)`$]\!]$ $[\![$`E`$_c$$]\!]$ *known?* =
$[\![$`((cont (I*) E')`$\ $`E'`$_0$ `...E'`$_n$`)`$]\!]$
where: `E'` $=$ *convert*$([\![$`E`$]\!], [\![$`E`$_c$$]\!],$ *known?*$)$
`E'`$_i$ $=$ *value*$([\![$`E`$_i$$]\!])$

In all other applications of `lambda` expressions the procedure is again changed into a `cont` expression with the continuation propagated into the body, but here it is left as the procedure being called. *value* is called on all of the arguments; in the linear code none of these can be applications.

convert $[\![$`(return P E`$_0$ `...E`$_n$`)`$]\!]$ $[\![$`E`$_c$$]\!]$ *known?* =
$[\![$`(P () E`$_c$ `E'`$_0$ `...E'`$_n$`)`$]\!]$
where: `E'`$_i$ $=$ *value*$($`E`$_i$$)$

`return` expressions are the only place calls are made to unknown continuations. The primitive becomes the procedure and the continuation is added as a value argument, not as a continuation argument.

convert $[\![$`(P () E`$_0$ `...E`$_n$`)`$]\!]$ $[\![$`(cont ...)`$]\!]$ *true* =
$[\![$`(P' ((cont ...)) E'`$_0$ `...E'`$_n$`)`$]\!]$
where: `P'` $=$ *convertprimitive*$([\![$`P`$]\!])$
`E'`$_i$ $=$ *value*$([\![$`E`$_i$$]\!])$

convert $[\![$`(P () E`$_0$ `...E`$_n$`)`$]\!]$ $[\![$`I`$]\!]$ *true* =
$[\![$`(P' ((cont (x) ($return () I x))) E'`$_0$ `...E'`$_n$`)`$]\!]$
where: `P'` $=$ *convertprimitive*$([\![$`P`$]\!])$
`x` is a new identifier with size *primitivevaluesize*$([\![$`P`$]\!], [\![$`E`$_0$$]\!], \ldots, [\![$`E`$_n$$]\!])$
`E'`$_i$ $=$ *value*$([\![$`E`$_i$$]\!])$

If a primitive application has no continuation arguments the continuation is added as one. If the continuation is an identifier, a `cont` form must be wrapped around it as continuation arguments cannot be identifiers. The primitive itself must be modified as it now calls a continuation argument instead of returning a value. Actually, the first case above is unnecessary as the second is always correct. The first is included only to improve the speed of compilation and has no effect on the output of the compiler.

convert $[\![$(P (C$_0$ ... C$_n$) E$_0$ ... E$_m$) $]\!][\![$E$_c]\!]$ *known?* =
$[\![$(P (C$'_0$ ... C$'_n$) E$'_0$ ... E$'_n$)$]\!]$
if $n = 1$ or *cont* is an identifier, otherwise
$[\![$((cont (j) (P (C$''_0$ ... C$''_n$) E$'_0$ ... E$'_n$)) E$_c$)$]\!]$
where: j is a new identifier of size ptr
C$'_i$ = *convert-cont*($[\![C_i]\!]$, $[\![$E$_c]\!]$, *known?*)
C$''_i$ = *convert-cont*($[\![C_i]\!]$, $[\![$j$]\!]$, *known?*)
E$'_i$ = *value*($[\![$E$_i]\!]$)

The final case is that of a primitive application with one or more continuation arguments. If there is only one, and thus the continuation will not need to be copied, or the continuation is an identifier, in which case the copying will not increase the size of the program, the continuation is propagated into the continuation arguments. Otherwise the continuation is bound to a new identifier and the new identifier is propagated into the continuation arguments. Again, the second transformation is correct in all cases and the first is used only for compiler efficiency.

5.3 Notation and basic blocks

While CPS code is easy for programs to analyze it is very hard to read. A little syntactic sugaring makes the code much more comprehensible. The syntax that is used here is another variation on Scheme's `let*` syntax. It looks much like that used for the linear code, but the syntax has a somewhat different interpretation. The identifiers are not bound by `lambda` expressions that are called on values, instead they are bound by `cont` expressions that are continuation arguments to the application. Calls to `cont` expressions are written exactly as calls to `lambda` expressions were before.

`Let*` as used in the linear examples:

```
(let* ((v) ($p () x y)) ...)
```

$\Longleftrightarrow$

```
((lambda (v) ...) ($p () x y))
```

`Let*` as used in the CPS examples:

```
(let* ((v) ($p x y)) ...)
```

$\Longleftrightarrow$

```
($p ((lambda (v) ...)) x y)
```

The meaning of the binding clauses in the `let*` is as follows:

```
(vars ($p arg ...)) {rest}
  ⟺
($p ((lambda vars {rest})) arg ...)

((v1 v2 ...) x1 x2 ...) {rest}
  ⟺
((cont (v1 v2 ...) {rest}) x1 x2 ...)
```

Unreferenced identifiers with a size of zero will not be shown.

The `let*` ends whenever a call to a primitive does not have exactly one continuation. Thus each `let*` consists of a sequence of calls through which there is only one possible execution path. In other words, each `let*` is a basic block of the program.

The abstraction that begins a basic block may be a `proc` expression, in which case it is called a 'procedure', or a `cont` expression that is either a continuation argument to a call that has more than one continuation, called a 'split', or a `cont` expression that is a non-continuation argument, called a 'join'. Every expression is contained in one procedure that is lexically inferior to all other procedures containing the expression. This procedure is called the expression's 'enclosing procedure'.

Some of the compiler's algorithms operate on the basic blocks and the call graph of the program directly. Each basic block has the following attributes that are determined and used by the compiler:

start	- the starting lambda expression
end	- the final call
known?	- are all of the calling points known
join?	- is the start of this block a join
split?	- is the start of this block a split
procedure?	- is the start of this block a procedure
lexical-parent	- the lexically superior block
lexical-children	- the immediate lexical inferiors of this block
parents	- the blocks known to precede this one in the call graph
children	- the blocks known to follow this one in the call graph

5.4 Restrictions

After conversion to CPS there are two properties of the use of continuations and joins in the code that the rest of the compiler must preserve. Continuation identifiers and the continuations bound to them are used in a first-in-first-out fashion. This is equivalent to ensuring that the `proc` expression binding the continuation identifier is the enclosing procedure of all references to the identifier. The second property is that the enclosing procedure of all calls to joins must be the enclosing procedure of the join itself. These properties will allow the compiler to allocate and deallocate environments for continuations and joins on a stack.

5.5 Factorial example

The program has now been transformed into CPS. There are five basic blocks: the main program, the `Fact` procedure, and three blocks that make up the recursive procedure from the `FOR` loop.

```
(proc p.39 (global)
  (let* (((p.0)  ($push '16))
         ((p.1)  ($push '16))
         ((x r)  p.0 p.1)
         ((p.2)  ($push 'ptr))
         ((fact) p.2)
         (()     ($set-contents fact <FACT>))
         ((p.15) ($contents global '(si)))
         ((t.14) ($read p.15))
         (()     ($set-contents x t.14))
         ((p.13) ($contents global '(si)))
         (()     ($read-line p.13))
         ((p.12) ($contents fact))
         (()     ($proc-call p.12 x r '(ptr ptr)))
         ((t.10) ($contents r))
         ((p.11) ($contents global '(so)))
         (()     ($write t.10 p.11))
         ((p.9)  ($contents global '(so)))
         (()     ($write-line p.9)))
    ($simple-return () p.39)))
```

```
<FACT> =
(proc p.40 (args)
  (let* (((n res) ($get-args '(ptr ptr) args))
         ((p.16)  ($push '16))
         ((n1)    p.16)
         (()      ($copy n n1 '16))
         ((p.18)  ($push '16))
         ((p.19)  ($push '16))
         ((i r1)  p.18 p.19)
         (()      ($set-contents r1 '1))
         ((p.24)  ($push 'ptr))
         ((loop)  p.24)
         (()      ($set-contents loop <LOOP>))
         (()      ($set-contents i '1))
         ((p.27)  ($contents loop))
         (()      ($simple-call p.27))
         ((t.23)  ($contents r1))
         (()      ($set-contents res t.23)))
    ($simple-return () p.40)))

<LOOP> =
(proc p.41 ()
  (let* (((t.28) ($contents i))
         ((t.29) ($contents n1)))
    ($equal16 (<TRUE> <FALSE>) t.28 t.29)))

<TRUE> =
(cont ()
  ($simple-return () p.41))
```

```
<FALSE> =
(cont ()
  (let* (((t.37) ($contents r1))
         ((t.38) ($contents i))
         ((t.36) ($multiply16 t.37 t.38))
         (()     ($set-contents r1 t.36))
         ((t.35) ($contents i))
         ((t.34) ($add16 t.35 '1))
         (()     ($set-contents i t.34))
         ((p.33) ($contents loop))
         (()     ($simple-call p.33)))
    ($simple-return () p.41)))
```

Chapter 6

Code Improvements

6.1 Local code transformations

After CPS conversion the compiler uses a variety of local code transformations and two global transformations, one of which is based on flow analysis, in order to improve efficiency of the code. The compiler applies the local transformations both before and after the global transformations. Doing them beforehand makes the global transformations more effective by simplifying the code and doing them afterwards allows them to take advantage of the changes made by the global transformations. The compiler continues to apply local transformations until none are applicable to any point in the code.

Since there is no specified order in which the local transformations are applied the question of termination arises. There are two ways in which the code improvement transformations might fail to terminate: either the transformed program grows without bound or the transformations loop, with some undoing the work of others. The only place where code (other than identifiers and constants) is duplicated is in the beta-substitution of abstractions. The compiler only duplicates abstractions below a certain size and if the program grows too large (relative to the size of the original program) duplication stops for the rest of the compilation. It is important to verify that none of the transformations undo the work of others so that the improvement process will eventually terminate.

Many of the local transformations used are well known [Steele 78, Brooks 82, Kranz 86, Standish 76]. For clarity most of the transformations are shown in non-CPS code. It is important to remember that in CPS code the arguments to calls cannot be calls and so can be evaluated without consulting or modifying the store.

Each separate transformation is meant to be as simple as possible. The efficiency of the final code comes from the interaction of the various transformations. Many of the transformations shown here could be improved and many more could be added.

6.1.1 Beta substitution

The simplest code improvement done is beta substitution; that is, substituting values for identifiers. $[value/identifier]$*expression* is defined to be *expression* with *value* substituted everywhere for *identifier*. Since identifiers must be unique shadowing is not a problem.

`((cont (x)` *body*`)` *value*`)`

$\Longrightarrow$

`((cont () [` $value/$ *value*`]` *body*`))`

After CPS conversion *value* can only be a constant, an identifier, or an abstraction and can be substituted in *body* without worrying about side-effects. The only case in which the value is not substituted is when `x` is referenced more than once and *value* is too large an expression to be duplicated cheaply. The current definition is of 'too large' is weighted towards keeping the program small. Copying larger values might speed up the compiled programs; I have not made any measurements on how the maximum size of duplicated expressions affects the final program's size and speed.

A call to a `cont` expression with no arguments is replaced with the body of the expression.

```
((cont () body))
```

$\Longrightarrow$

```
body
```

If a primitive does not use the store and its continuation does not use any values that the primitive returns, the call to the primitive is replaced with the body of the continuation.

```
(primitive ((cont (x) body-not-referencing-x)) arguments)
```

$\Longrightarrow$

```
body-not-referencing-x
```

6.1.2 Operation specific transformations

Some primitives have associated code improvement transformations. Examples of this are constant folding and reduction in strength. Different languages require different primitive operations and thus may have different operation specific transformations.

The current implementation performs constant folding for arithmetic primitives and conditional expressions. The following examples show the method of substituting a value for the result of a primitive call.

```
($add ((cont (x) body)) '2 '3)
  ⟹
((cont (x) body) '5)

($if ((cont () true-body) (cont () false-body)) 'false)
  ⟹
((cont () false-body))
```

Reduction in strength means replacing an expensive instruction with a cheaper one. The one example of this that the compiler does is replacing multiplication by integer constants with sequences of shifts and adds. On the Motorola 68020 multiplication is a slow instruction and the corresponding shifts and adds are noticeably faster.

As an example of a transformation that is both language and machine specific the `$copy` primitive used in the factorial example is implemented in one of three different ways depending on the size of the location being copied. If the location is not larger than one machine word the call to `$copy` is replaced with calls to `$contents` and `$set-contents`. If the location is larger than one machine word the `$copy` primitive is replaced with one of two different primitives, one of which generates a sequence of move instructions, the other generates a simple loop. The size at which the loop version is specified by the front-end and depends on the relative importance of program size and speed, and the size and speed of move instructions on the target machine.

```
($copy () n n1 '16)
  ⟹
(let* ((x) ($contents n '16))
  ($set-contents n1 '16 x))
```

6.1.3 Simplifying procedure calls

Certain primitives call one of their values. If the value that is being called can be identified then the calls may be simplified. If all calls to a particular abstraction expression are known, all use the same primitive, and only that expression is called at those calls, then a transformation specific to the calling primitive is applied. This usually results in replacing the primitive with one of the standard call primitives, `$call` or `$return`, along with removing any argument destructuring from the body of the abstraction. If the procedure returns values a primitive call may need to be introduced to destructure the return values.

```
(let* (((p) (proc c:ptr (all:state))
            ($unpack-call ((cont (x:ptr y:ptr) ...))
                          '<ptr ptr>
                          all))
       ((z:16) ($func-call p a b)))
  ...)
  ⟹
(let* (((p)       (proc c:ptr (x:ptr y:ptr) ...))
       ((ret:all) ($call p a b))
       ((z:16)    ($unpack-return ret '<16>)))
  ...)
```

In this example the primitive `$func-call` packages up the call arguments, calls the procedure, and unpacks the return value. Once the procedure has been identified the arguments no longer need to be packaged and unpackaged and the compiler does not need to use any particular calling convention when calling the identified procedure. The packaging of return values can be removed if the continuation passed to the procedure can be identified in the same manner.

Some languages require a slightly different process. In Scheme some arguments may need to be consed into a list. In the example below, `f` has two formal parameters: `x`, which is bound to the first argument, and `y`, which is bound to a list of all the remaining arguments. The call to `f` passes it three arguments. In the transformation a new primitive, `$list`, is used to make a list of the last two actual arguments. `$call` can only be used when there is a one-to-one correspondence between the expected and actual arguments.

```
(let ((f (lambda (x . y) ...)))
  (f a b c)
  ...)
  ⟹ {Translation and CPS conversion}
(let* (((f) (proc c:ptr (all:state))
            ($unpack-call ((cont (x:ptr y:ptr) ...))
                          '<ptr rest-list>
                          all))
      ((z:ptr) ($func-call f a b c)))
  ...)
  ⟹
(let* (((f)        (proc c:ptr (x:ptr y:ptr) ...))
       ((temp:ptr) ($list b c))
       ((ret:all)  ($call f a temp))
       ((z:ptr)    ($unpack-return ret '<ptr>)))
  ...)
```

There is also a transformation associated with the three standard calling primitives `$call`, `$return`, and `$jump`. If the expression being called is an abstraction the call is changed to call the expression directly. In the case of `$call` the continuation identifier of the `proc` expression is added to the list of other identifiers and the `proc` is changed to a `cont`.

```
($call ((cont (...) ...)) (proc c (x y z) ...) a b c)
  ⟹
((cont (c x y z) ...) (cont (...) ...) a b c)
```

6.1.4 Evaluation for control

Two local transformations implement evaluation for control (boolean short circuiting). The first involves testing the result of a conditional expression, the second is the propagation of conditional results. These are shown in non-CPS code for clarity.

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

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

A third transformation that moves conditionals down into blocks increases the applicability of the nested conditionals transformation.

```
(if (block <1> (if a b c)) d e)
  ⟹
(block <1> (if (if a b c) d e))
```

The following (Scheme) example shows how these transformations implement evaluation for control. In the example boolean negation is defined to be the procedure `NOT`. This definition is substituted into a piece of code which is then reduced to a single conditional by the transformations described above. `AND` and `OR` can be done similarly.

```
NOT = (lambda (x) (if x false true))

(if (not x) 0 1)
    ⟹ (if (if x false true) 0 1)
    ⟹ (if x (if false 0 1) (if true 0 1))
    ⟹ (if x 1 0)
```

6.1.5 Local location removal

A variety of transformations may be applied to calls that allocate locations.

```
(let ((loc ($make-location 'size)))
  ...)
```

If `loc` is unused, the call to `$make-location` is removed from the code as it is unnecessary.

If all uses of `loc` are known, that is `loc` only appears as an argument to `$contents` and `$set-contents` with a single, constant offset, then several transformations may be possible:

If `loc` only appears in calls to `$set-contents` then the contents of the location are never accessed and all of the `$set-contents` calls are removed.

If `loc` only appears in calls to `$contents` then the contents of the location are never set and some constant value, currently zero, is substituted for the values of the `$contents` calls.

If there is exactly one call to `$set-contents` then the value being put in the cell is substituted for all of the `$contents` calls which are in its scope. If the value is an abstraction that is too large to duplicate, the call to `$set-contents` is moved as far down in its containing basic block as possible without moving it past any use of the location. This increases the possibility that other values may be substituted into the body of the abstraction.

6.2 Flow analysis

This is a global transformation that substitutes known values for identifiers. The algorithm uses the basic block structure of the program and the call graph.

6.2.1 The algorithm

1. Collect all basic blocks that have all of their calling points identified and are not continuation arguments. These are all of the blocks called using `$call`, `$return`, or `$jump`.

2. Collect all identifiers bound by the top abstractions of the selected blocks. These are the known identifiers.

3. For every known identifier collect all possible arguments and divide the arguments into known identifiers and other values. For every known identifier determine the known identifiers to which it is passed as a value.

4. Calculate the transitive closure of the values received by the known identifiers, each of which passes its values to all known identifiers which got it as a value in step 3.

5. For every known identifier with a single possible unknown value, if that value is lexically in scope wherever the identifier is referenced, introduce a new `cont` expression binding the identifier to the value at the lowest common ancestor of all of the uses. Remove the identifier from its original `cont` or `proc` expression and remove the corresponding argument at all calls to the expression. If the value is an abstraction the transformation may be done only if there are no other uses of the expression as duplication of abstractions may cause the program to grow overly large.

6.2.2 An example

```
(let ((f (lambda (x) body)))
  (let ((g (lambda (y) (f y)))
        (h (lambda (z) (f z))))
    ((g a) (h a))))
```

Following the five steps listed above we get:

1. The calling points of procedures `f`, `g`, and `h` are known.
2. The known identifiers are `x`, `y`, and `z`.
3. The only possible value of `y` and `z` is the value `a`. `x` gets the known identifiers `y` and `z` as values.
4. The transitive closure gives all three known identifiers just the value `a`.
5. The (unknown) identifier `a` may be substituted for all three identifiers.

```
(let ((f (lambda () ((lambda (x) body) a))))
  (let ((g (lambda () ((lambda (y) (f)) a)))
        (h (lambda () ((lambda (z) (f)) a))))
    ((g) (h))))
  ⟹ {local transformations}
(let ((f (lambda () [a/x] body)))
  (let ((g (lambda () (f)))
        (h (lambda () (f))))
    ((g) (h))))
```

If the value to be substituted is a continuation, either a `cont` expression or the continuation identifier of a `proc` expression, the algorithm must check that the substitution of the value does not move a calling point of a continuation from one procedure into another. This is to preserve the first-in-first-out usage of continuations. When substituting a value for the continuation identifier of a `proc` expression the `proc` expression must be changed to a `cont` and the primitives used to call the procedure must be changed from `$call` to `$jump` as the procedure no longer takes a continuation argument.

To allow the substitution of loop continuations, continuations of the form `(cont ({identifiers}) ($return c {identifiers}))` are considered to

be the same as the identifier `c`. The `cont` expression is present only because continuation arguments to primitives cannot be identifiers. The continuations to simple iterative loops are always substituted, as `$set-contents` calls where the value is an abstraction are moved as far down in the code as possible as explained in Section 6.2 above. In the case of a simple iteration, this puts the `$set-contents` just before the initial call to the recursive procedure, so that all identifiers in the continuation to this call are in scope in the body of the recursive procedure and thus the continuation may be substituted into the body. This situation arises in the factorial example presented at the end of this chapter.

6.3 Removing locations

This global transformation attempts to reduce the use of the store and thus increase the effectiveness of the other transformations by allowing them to manipulate values that would otherwise be hidden in the store. The contents of particular locations in the store are passed explicitly from procedure to procedure instead of implicitly in the store. Only some uses of some locations may be removed in this fashion.

The transformation will be presented here in terms of a single location. The compiler actually does all locations in a single pass over the code. For the purposes of this transformation a location is identified with the identifier to which it is originally bound. Due to the wonders of CPS there will always be such an identifier.

The transformation is applied globally only if all references to the location are either setting or retrieving the contents of the location, and always with the same size of data and an offset of zero. This transformation can be used for locations that do not fit the listed criterion, such as locations that are passed as an argument to a procedure, if the transformation is begun only at the call that creates the location and stops when it reaches any call to a procedure or continuation.

6.3.1 Examples

Here is an example of this transformation, applied to a location `loc`. The contents and use of the location can be determined within the basic block and calls to `$set-contents` and `$contents` are removed.

```
(proc c ()
  (let* (((w) ($contents loc))
         ((x) ($+ w '3))
         (()  ($set-contents loc x))
         ((y) ($contents loc))
         ((z) ($/ y '2))
         (()  ($set-contents loc z)))
    ($return c)))
  ⟹
(proc c ()
  (let* (((w) ($contents loc))
         ((x) ($+ w '3))
         ((z) ($/ x '2))
         (()  ($set-contents loc z)))
    ($return c)))
```

For a `cont` expression, where all of the calling points are known and within a single procedure, the transformation is more effective as the contents of the location are passed as an argument and not inside the location.

```
(cont ()
  (let* (((w) ($contents loc))
         ((x) ($+ w '3))
         (()  ($set-contents loc x))
         ((y) ($contents loc))
         ((z) ($/ y '2))
         (()  ($set-contents loc z)))
    ($jump a)))
  ⟹
(cont (w)
  (let* (((x) ($+ w '3))
         ((z) ($/ x '2)))
    ($jump a z)))
```

6.3.2 The transformation

The information required for the transformation is the call graph of the program and a list of the `proc` expressions that contain references to the location. Each `proc` expression is transformed separately. For this transformation a `proc` expres-

sion is considered to include all `cont` expressions that are lexically inferior to it and not inferior to any lexically inferior `proc` expression. Remember that all calls to each such `cont` expression will be within the same `proc` expression as the expression itself. Only `proc` expressions that contain references to the location are transformed.

The transformation `flow` takes an application, two boolean flags, and an expression and returns the transformed application. The first flag, *in*, indicates whether the value that should be in the location at the application is actually stored in the location. The second flag, *known?*, indicates whether the expression argument is known to be the value that should be in the location. The expression is always either an identifier or a constant.

In the presentation of the transformation `loc` is the identifier bound to the location and `size` is the size of the value it contains.

The bodies of all `proc` expressions that reference the store are transformed using *flow* with the value initially in the location and not known to the compiler.

`(proc I (I*) A)`
$\Longrightarrow$
`(proc I (I*)` *flow*$(\mathrm{A}, \mathit{true}, \mathit{false}, '0))$

Two utility transformations are used for `cont` expressions. The first is applied to continuation arguments to primitive calls and `cont` expressions that are being called directly. It just calls `flow` on the body of the expression.

flow-cont $[\![$`(cont (I*) A)` $]\!]$ *in? known?* $[\![\mathrm{E}_v]\!] =$
$[\![$`(cont (I*) A`$')]\!]$
where: $\mathrm{A}' =$ *flow*$(\mathrm{A}, \mathit{in?}, \mathit{known?}, [\![\mathrm{E}_v]\!])$

The second, *flow-value*, may be called on any type of expression but only changes `cont` expressions. These get a new identifier for the contents of the location which is in turn propagated into the body of the expression.

flow-value $[\![$`(cont (I*) A)`$]\!] =$
$[\![$`(cont (I I*) A`$')]\!]$
where: $\mathrm{A}' =$ *flow*$([\![\mathrm{A}]\!], \mathit{False}, \mathit{True}, [\![\mathrm{I}]\!])$
I is a new identifier with size `size`

Now for `flow` itself. The call that creates the location is not changed. At that point, as the contents of the location has not yet been set, the current value can be considered to be stored in the location as well as known. The transformation is the same if the location is created with `$allocate`.

flow $[\![$`($push (C) size)`$]\!]$ *in? known?* $[\![\mathrm{E}_v]\!] =$
$[\![$`($push (C′) size)`$]\!]$
where: $\mathrm{C}' =$ *flow-cont*$([\![\mathrm{C}]\!], \mathit{True}, \mathit{True}, [\![$`'0`$]\!])$

Calls that set the contents of the location are removed. The current value is the new value, which has not been stored in the location.

flow $[\![$`($set-contents ((cont (I) A)) loc size '0 E)`$]\!]$ *in? known?* $[\![\mathrm{E}_v]\!] =$
flow$([\![\mathrm{A}]\!], \mathit{False}, \mathit{True}, [\![\mathrm{E}]\!])$

Calls that dereference the location just pass the known value, if any, to the continuation. If there is not a known value, the call is not removed and the continuation's identifier becomes the new value.

flow $[\![$`($contents ((cont (I) A)) loc size '0)`$]\!]$ *in? known?* $[\![\mathrm{E}_v]\!] =$
$[\![$`((cont (I) A′) `$$\mathrm{E}_v$`)`$]\!]$
where: $\mathrm{A}' =$ *flow*$([\![\mathrm{A}]\!], \mathit{in?}, \mathit{True}, [\![\mathrm{E}_v]\!])$
if *known?*, otherwise
$[\![$`($contents ((cont (I) A′)) loc size '0)`$]\!]$
where: $\mathrm{A}' =$ *flow*$([\![\mathrm{A}]\!], \mathit{True}, \mathit{True}, [\![\mathrm{I}]\!])$

When control is transferred to a known `cont` expression the current value is passed along as an argument.

flow $[\![$`($jump () I E*)`$]\!]$ *in? known?* $[\![\mathrm{E}_v]\!] =$
$[\![$`($jump () I `$$\mathrm{E}_v$`  E*)`$]\!]$
if *known?*, otherwise
$[\![$`($contents ((cont (I′) ($jump () I I′ E*))) loc size '0)`$]\!]$
where: I' is a new identifier with size `size`

Calls to `cont` expressions are not changed.

flow $[\![$`(C `E_0 $\ldots$ E_m`)`$]\!]$ *in? known?* $[\![\mathrm{E}_v]\!] =$
$[\![$`(C′ `E'_0 $\ldots$ E'_m`)`$]\!]$
where: $\mathrm{C}' =$ *flow-cont*$([\![\mathrm{C}]\!], \mathit{in?}, \mathit{known?}, [\![\mathrm{E}_v]\!])$
$\mathrm{E}'_i =$ *flow-value*$([\![\mathrm{E}_i]\!])$

At all other calls, if the call is a return or might use the location the current value must be stored in the location if it is not already there. A return is a call with no continuations that does not call `$jump`. An expression might use the location if either 1) it actually contains a reference the location; 2) it is a call to a known `proc` expression that references the location; or 3) it is a call to an unknown procedure and there is a procedure that might use the location and not all of whose calling points are known to the compiler.

flow $[\![$`(P (C*) E*)` $]\!]$ *in? known?* $[\![\mathrm{E}_v]\!] =$
$[\![$`($set-contents ((cont (I) A′)) loc size '0` $[\![\mathrm{E}_v]\!])$ $]\!]$
where: $\mathrm{A}' =$ *flow-call*$([\![$`(P (C*) E*)` $]\!], \mathit{True}, \mathit{False}, [\![\mathrm{E}_v]\!])$
I is a new identifier with size zero
if not *in?* and `loc` is used in $[\![$`(P (C*) E*)` $]\!]$, otherwise
$[\![\mathrm{A}']\!]$
where: $\mathrm{A}' =$ *flow-call*$([\![$`(P (C*) E*)` $]\!], \mathit{in?}, \mathit{known?}, [\![\mathrm{E}_v]\!])$

flow-call $[\![$`(P (`$\mathrm{C}_0 \ldots \mathrm{C}_n$`) `$\mathrm{E}_0 \ldots \mathrm{E}_m$`)` $]\!]$ *in? known?* $[\![\mathrm{E}_v]\!] =$
$[\![$`(P (`$\mathrm{C}'_0 \ldots \mathrm{C}'_n$`) `$\mathrm{E}'_0 \ldots \mathrm{E}'_m$`)` $]\!]$
where: $\mathrm{C}'_i =$ *flow-cont*$([\![\mathrm{C}_i]\!], \mathit{in?}, \mathit{known?}, [\![\mathrm{E}_v]\!])$,
$\mathrm{E}'_i =$ *flow-value*$([\![\mathrm{E}_i]\!])$

6.4 Factorial example

6.4.1 Local transformations

The local transformations do some beta substitution, remove several locations that are set only once, and substitute the value of the factorial procedure and its continuation to their respective calling points. The calling points of the loop procedure have been identified and `$call` and `$return` are now used to call and return from that procedure.

```
(proc p.39 (global)
  (let* (((p.15) ($contents global '(si)))
         ((t.14) ($read p.15))
         ((p.13) ($contents global '(si)))
         (()     ($read-line p.13))
         ((p.18) ($push '16))
         ((p.19) ($push '16))
         (()     ($set-contents p.19 '1))
         ((p.24) ($push 'ptr))
         (()     ($set-contents p.24 <LOOP>))
         (()     ($set-contents p.18 '1))
         ((p.27) ($contents p.24))
         (()     ($call p.27))
         ((t.23) ($contents p.19))
         ((p.11) ($contents global '(so)))
         (()     ($write t.23 p.11))
         ((p.9)  ($contents global '(so)))
         (()     ($write-line p.9)))
    ($simple-return () p.39)))

<LOOP> =
(proc p.41 ()
  (let* (((t.28) ($contents p.18)))
    ($equal16 (<TRUE> <FALSE>) t.28 t.14)))

<TRUE> =
(cont ()
  ($return () p.41))
```

```
<FALSE> =
(cont ()
  (let* (((t.37) ($contents p.19))
         ((t.38) ($contents p.18))
         ((t.36) ($multiply16 t.37 t.38))
         (()     ($set-contents p.19 t.36))
         ((t.35) ($contents p.18))
         ((t.34) ($add16 t.35 '1))
         (()     ($set-contents p.18 t.34))
         ((t.33) ($contents p.24))
         (()     ($simple-call t.33)))
    ($return () p.41)))
```

6.4.2 Flow analysis

The continuation of the loop is substituted for the identifier `p.41`. The two calls to the loop now use `$jump` instead of `$call` and the loop is changed from a `proc` to a `cont`.

```
(proc p.39 (global)
  (let* (((p.15) ($contents global '(si)))
         ((t.14) ($read p.15))
         ((p.13) ($contents global '(si)))
         (()     ($read-line p.13))
         ((p.18) ($push '16))
         ((p.19) ($push '16))
         (()     ($set-contents p.19 '1))
         ((p.24) ($push 'ptr))
         (()     ($set-contents p.24 <LOOP>))
         (()     ($set-contents p.18 '1))
         ((p.27) ($contents p.24)))
    ($jump p.27)))

<LOOP> =
(cont ()
  (let* (((t.28) ($contents p.18)))
    ($equal16 (<TRUE> <FALSE>) t.28 t.14)))

<TRUE> =
(cont ()
  (let* (((t.23) ($contents p.19))
         ((p.11) ($contents global '(so)))
         (()     ($write t.23 p.11))
         ((p.9)  ($contents global '(so)))
         (()     ($write-line p.9)))
    ($simple-return () p.39)))
```

```
<FALSE> =
(cont ()
  (let* (((t.37) ($contents p.19))
         ((t.38) ($contents p.18))
         ((t.36) ($multiply16 t.37 t.38))
         (()     ($set-contents p.19 t.36))
         ((t.35) ($contents p.18))
         ((t.34) ($add16 t.35 '1))
         (()     ($set-contents p.18 t.34))
         ((t.33) ($contents p.24)))
    ($jump t.33)))
```

6.4.3 Removing locations

The contents of the locations `p.18` and `p.19` are passed explicitly as `t.40` and `t.41`. Various local transformations were applied after the flow transformation. This is the code at the end of the code improvement phase of the compiler.

```
(proc p.39 (global)
  (let* (((p.15) ($contents global '(si)))
         ((t.14) ($read p.15))
         ((p.13) ($contents global '(si)))
         (()     ($read-line p.13))
         ((p.24) ($push 'ptr))
         (()     ($set-contents p.24 <LOOP>))
         ((p.27) ($contents p.24)))
    ($jump p.27 '1 '1)))

<LOOP> =
(cont (t.40 t.41)
  ($equal16 (<TRUE> <FALSE>) t.40 t.14))

<TRUE> =
(cont ()
  (let* (((p.11) ($contents global '(so)))
         (()     ($write t.41 p.11))
         ((p.9)  ($contents global '(so)))
         (()     ($write-line p.9)))
    ($simple-return () p.39)))

<FALSE> =
(cont ()
  (let* (((t.36) ($multiply16 t.41 t.40))
         ((t.34) ($add16 t.40 '1))
         ((t.33) ($contents p.24)))
    ($jump t.33 t.34 t.36)))
```

Chapter 7

Implementing Environments

7.1 Environments

The intermediate and machine languages treat identifiers differently. In the intermediate language's semantics the values of identifiers are stored in environments that are created when procedures are called and the environments are saved for use in evaluating lexically inferior expressions. In the machine language identifiers are treated as locations and their values are kept in the store, that is, the value of an identifier is the value to which it was bound most recently. The compiler uses a global transformation to add the environments and lexical scoping of the intermediate language's semantics to the program itself.

The environments added by the transformation are linear arrays in the store with the values of particular identifiers stored at fixed offsets. The compiler must have an environment for every abstraction that contains free identifiers and must make that environment available whenever the value of the abstraction is applied. The environments may be either on the continuation stack or allocated from a heap. In languages such as FORTRAN, which has no procedure values, or Pascal, which is designed to prohibit procedures from outliving their lexical superior, all procedures may be given stack environments. Scheme procedures may use the stack only if the compiler can prove that they do not escape upwards.

`proc` expressions have their environments passed to them as an additional argument. All `cont` expressions use the current stack environment. The restrictions on the use of continuations ensure that the top stack environment when a `cont` expression is evaluated is also on top of the stack whenever the value of the expression is called. The register allocator takes care of saving and retrieving the

values of identifiers needed by continuation arguments to primitive applications, so these may be ignored for the moment.

The primitives introduced by the compiler to create and manipulate environments are:

`($make-heap-environment)` creates an environment in the heap.

`($push-stack-environment c)` pushes a new stack environment on top of continuation `c`.

`($pop-stack-environment c)` removes environment `c` from the stack and returns the continuation upon which it was pushed.

`($set-environment e name value)` sets the value of `name` in environment `e` to be `value`.

`($get-environment e name)` gets the value of `name` in environment `e`.

`($set-code e name value)` the same as `set-environment` except that `name` is the name of a `proc` expression and `value` is a pointer to the code for a procedure.

`($make-procedure e name)` makes a procedure from environment `e` and the value of `name` in `e`.

`($make-pointer e name)` makes a pointer into the environment at the location containing the value of `e`.

The names used by the compiler are symbolic constants that are replaced with concrete offsets after register allocation.

7.2 The transformation

This transformation is done in four stages:

1. Create heap and stack environments for every `proc` expression.
2. Store all identifiers into the local heap and stack environments.
3. Create a procedure value for every `proc` expression.

4. Obtain the values of identifiers from the environments where necessary.

The first step is to add the calls that create the environments. Every procedure creates two environments, one on the stack and the other in the heap. All procedures that are lexically inferior to it, but not to any other procedure that is inferior to it, have as their environment either the stack environment or the heap environment, depending on whether they escape upwards or not. The stack environment is created on top of the continuation passed to the procedure. References to the continuation are replaced with references to the environment. An environment identifier is added to each procedure so that its own environment may be passed to it. The outer `proc` of the program does not get an identifier added as it is already passed the global environment.

```
(proc c (x y) ...)
  ⟹
(proc c (e x y)
  (let* (((s) ($push-stack-environment c))
         ((h) ($make-heap-environment)))
    ...))
```

The second step is to store the values of all identifiers in the current stack and heap environments.

```
(proc c (e x y)
  (let* (((s) ($push-stack-environment c))
         ((h) ($make-heap-environment)))
    ...
    (cont (z) ...)
    ...))
  ⟹
```

```
(proc c (e x y)
  (let* (((s) ($push-stack-environment c))
         ((h) ($make-heap-environment))
         (()  ($set-environment s 'h h))
         (()  ($set-environment h 's s))
         (()  ($set-environment s 'x x))
         (()  ($set-environment h 'x x))
         {and so forth})
    ...
    (cont (z)
      (let* ((() ($set-environment s 'z z))
             (() ($set-environment h 'z z)))
        ...))
    ...))
```

In the third step every procedure is replaced with two calls, one that adds the procedure's code to the appropriate environment of its superior, and a second that makes a pointer into that environment. In this example `hs` is the heap environment if the procedure escapes upwards and the stack environment otherwise.

```
(proc c (e x y) ...)
  ⟹
(let* ((() ($set-code hs 'l1 (proc c (e x y) ...)))
       ((p) ($make-procedure hs 'l1)))
  p)
```

Finally, every identifier that is not bound within the surrounding procedure or join is replaced with its value from the proper environment.

```
(proc c0 (e0 x y)
  ...
  (proc c1 (e1) ... x ...)
  ...)
  ⟹
(proc c0 (e0 x)
  ...
  (proc c1 (e1) ... ($get-environment e1 'x) ...)
  ...)
```

For nested applications the appropriate environment itself must be obtained before the value can be retrieved.

```
(proc c0 (e0 x y)
  ...
  (proc c1 (e1) ...
    (proc c2 (e2) ... x ...)
    ...)
  ...)
  ⟹
(proc c0 (e0 x y)
  ...
  (proc c1 (e1) ...
    (proc c2 (e2)
      (let* (...
             ((e) ($get-environment e2 'e1))
             ((x) ($get-environment e 'x))
      ...)
  ...)
```

At this point primitive continuations may have any number of free identifiers, joins have one free identifier that is the current stack environment, and procedures have no free identifiers. Abstractions may now be copied without cost as they have become simple pointers to code. Since all of the calling points of joins have been identified, the `cont` expressions themselves have only the current stack as a free identifier, and that identifier has the correct value wherever the `cont` is applied, the value called at each `$jump` and `$return` may be replaced with the corresponding `cont` expression.

```
(let* (((p) (cont (x y) ...))
       ...)
  ($jump p a b))
  ⟹
(let* (...)
  ($jump (cont (x y) ...) a b))
```

7.3 Improvement transformations

After environments have been added to the code it can then be improved using local transformations. As with the previous set of code improving transformations the compiler continues to perform these transformations until none of them apply. Since none of them increase the size of the program and none of them undo the

work of others this process will terminate.

At the same time beta-substitution is performed whenever a transformation makes it possible and any calls that do not access the store and whose results are no longer used are removed.

If the value of an identifier has already been obtained in a lexically superior call that is within the same procedure or join, the second `$get-environment` call is replaced with the value of the first one.

```
(let* (((x0) ($get-environment e 'x))
       ...
       ((x1) ($get-environment e 'x))
       (()   ...x1...))
  ...))
  ⟹
(let* (((x0) ($get-environment e 'x))
       ...
       (()   ...x0...))
  ...))
```

If a value is never obtained from an environment, the call that inserts it is removed.

If the value of a name in an environment is the same as the value of some other name, then one name is used for both. This commonly arises with procedures and continuations, in which case calls to `$get-environment` become calls to `$make-procedure`.

```
(let* (((p1) (proc c1 () ...))
       ((p2) (proc c2 () ... p1 ...)))
  ⟹ {environment conversion}
(let* ((()   ($set-code hs 'L1 (proc c1 (e1) ...)))
       ((p1) ($make-procedure hs 'L1))
       (()   ($set-environment hs 'p1 p1))
       (()   ($set-code hs 'L2
               (proc c2 (e2)
                 ... ($get-environment e2 'p1) ...)))
       ((p2) ($make-procedure hs 'L2)))
```

The call `($get-environment e2 'p1)` is then replaced with `($make-procedure e2 'L1)`.

If a procedure does not use its environment at all, then its `$set-code` call can be removed and all its `$make-procedure` calls can be replaced with the `proc`

expression. If a group of procedures only use their environments to reference one another, then none of them actually needs their environment, and all of their `$make-procedure` calls are replaced with their `proc` expressions.

Heap environments that are not referenced may be removed. Stack environments may not be removed yet as the register allocator may need them for the environments of continuations to primitive calls.

7.4 Locations

Locations may be merged with environments if they share the same extent. All locations allocated using `$push` can be moved into the current stack environment. The `$push` call is replaced with a call to `$make-pointer`. Calls that access the location are replaced with calls that access the environment.

```
(let* (((s) ($push-stack-environment c))
       ((x) ($push '32))
       ((y) ($contents x '16 '16)))
  ...)
  ⟹
(let* (((s) ($push-stack-environment c))
       ((x) ($make-pointer s 'x))
       ((y) ($get-environment s 'x+16)))
  ...)
```

An identical transformation may be made using heap environments and locations returned by calls to `$allocate`.

7.5 Popping the stack

The calls to `$pop-stack-env` that remove stack environments are not added until after register allocation. At that point the calls that create unused stack environments are removed from the code. All remaining stack environments need to be popped off the stack. This is done by replacing the environments in returns with the continuation obtained by popping the environment off of the stack. Remember that the environment transformation replaced all references to continuation identifiers with the stack environment pushed on top of the continuation.

```
(proc c (...)
  (let* (((s) ($push-stack-environment c)))
     ... ($some-return s ...) ...))
  ⟹
(proc c (...)
  (let* (((s) ($push-stack-environment c)))
     ... ($some-return ($pop-stack-environment s) ...) ...))
```

In the case of a nearly tail-recursive call, one in which the continuation just calls a continuation identifier, if none of the arguments (including the called value) reference the stack environment, that stack environment may be popped off before the call. This turns the nearly tail-recursive call into a truly tail recursive call as no code needs to be generated for the continuation.

```
(proc c (...)
  (let* (((s0) ($push-stack-environment c)))
    ($some-call (cont (...) ($some-return s0)) ...)))
  ⟹
(proc c (...)
  (let* (((s0) ($push-stack-environment c))
         ((s1) ($pop-stack-environment s0)))
    ($some-call (cont (...) ($some-return s1)) ...)))
```

7.6 Factorial example

This is the factorial program after environments have been added and simplified. This example does not require much in the way of improvements other than removing unused calls. Only `global` and `t.14` are kept in an environment. Once the `<LOOP>` procedure has been substituted at its two calling points the cell for the recursive reference is no longer used and is removed. The call to pop off the stack environment will be added after register allocation.

```
(proc p.39 (global)
  (let* (((p.42) ($push-stack-environment p.39))
         (()     ($set-environment p.42 'global global))
         ((p.15) ($contents global '(si)))
         ((t.14) ($read p.15))
         (()     ($set-environment p.42 't.14 t.14))
         ((p.13) ($contents global '(si)))
         (()     ($read-line p.13)))
    ($jump <LOOP> '1 '1)))

<LOOP> =
(cont (t.40 t.41)
  (let* (((p.44) ($get-environment p.42 't.14)))
    ($equal16 (<TRUE> <FALSE>) t.40 p.44)))

<TRUE> =
(cont ()
  (let* (((p.43) ($get-environment p.42 'global))
         ((p.11) ($contents p.43 '(so)))
         (()     ($write t.41 p.11))
         ((p.9)  ($contents p.43 '(so)))
         (()     ($write-line p.9)))
    ($simple-return () p.39)))

<FALSE> =
(cont ()
  (let* (((t.36) ($multiply16 t.41 t.40))
         ((t.34) ($add16 t.40 '1)))
    ($jump <LOOP> t.34 t.36)))
```

Chapter 8

Resource Allocation

8.1 Machine resources

The final phase of compilation is the allocation of machine resources, such as registers and functional units, to the different parts of the program. The allocation of resources is expressed through transforming the program. For registers this involves changing the names of identifiers to correspond to the register currently containing the value of the identifier. Functional units are specified by the primitive operations, in that every primitive operation uses particular functional units. Allocating functional units involves replacing primitive operations with others that use the desired functional units.

8.2 Instruction selection and scheduling

The current implementation does one form of instruction selection. The instruction selection transformation attempts to find sets of primitive applications that can be coalesced into a single load or store instruction using the MC68020's indexed addressing mode. In Pascal programs opportunities for this transformation most often arise from array index calculations. On the MC68020 reading from an array of 16 bit values using a 16 bit index value involves several instructions, each of which is a separate primitive. The index must be sign-extended to a 32 bit value, added to the base offset of the array, multiplied by two (to get a byte address), and added to the address of the array to get the final address. Using the indexed addressing mode on the MC68020 the entire calculation can be done using one instruction.

Instructions, or rather the primitive operations that represent instructions, could be reordered in order to improve the register allocation. Currently the compiler does not do this.

8.3 Register allocation

Machine registers must be allocated to hold the values of identifiers. The two constraints that determine possible allocations are that the machine has only a fixed number of registers and primitive procedures have requirements as to the locations of their arguments and results. The correctness of a particular set of register allocations for a program is verified when the program is transformed to reflect the allocations.

Register allocation includes saving the environments of continuation arguments to primitive applications under the guise of ensuring at every application all live values are saved on the stack or in unused registers. The live values are exactly the free identifiers of the continuation arguments in the application. Primitives that call values are specified as modifying every register (except the stack) and thus every free identifier in the continuation to the call must be saved on the stack.

The current implementation uses a very simple register allocation algorithm. Each basic block is done separately, with splits inheriting the final machine state of their predecessor. The register selection algorithm is purely local to basic blocks with the exception that it must look ahead to determine which values need to be preserved for use in later blocks. Arguments in calls to abstractions whose applications are known are passed in registers determined entirely by the size of the values. Pointer values are passed in the MC68020's address registers, all other values are passed in the data registers. Values are targetted to particular registers if the block ends with a call, as calls usually require that each argument be in a particular location. Typically, non-call primitives only require that a value be in a register of a particular type.

It would be easy to improve the current implementation's register allocation algorithm. Global register allocation techniques such as graph coloring [Chaiten 82] or trace scheduling [Fisher 81] could be used with some modification.

8.4 Identifier renaming

Once registers have been selected for the identifiers in the program the names of the identifiers must be changed to those of the registers. At the same time primitive applications are added to the code to move values from register to register and to and from the stack. On the MC68020 the two primitives for moving values between registers are `$move` which moves a value from one register to another and `$exchange` which exchanges the values between two registers. `$set-environment` and `$get-environment` are used to move values to and from the current stack environment.

The renaming is done one call at a time, going down each basic block in turn. The program keeps track of the current machine state, which is the correspondence between identifiers and registers and stack locations and the correspondence between identifiers in the transformed and untransformed programs. For each call it verifies that the arguments are indeed in registers appropriate to the primitive and that the values of the arguments are the same as those in the unmodified call.

8.5 Finishing up

As described in Section 7.5, once the identifiers have been renamed any empty stack environments are removed from the code and primitives are added to pop off the rest.

One last improvement transformation is applied. Trivial basic blocks, those that consist of a single call, are removed from the code to eliminate jumps to jumps.

Finally, all `proc` and `cont` expressions are turned back into `lambda`s, with the continuation identifier of the `proc` expressions added to the front of the identifier list.

Compilation is now complete. To actually run the program it must be assembled into machine code. The current implementation uses the assembler from the Orbit compiler, which accepts hints as to how the blocks should be ordered. The compiler uses this to invert loops by having the exit conditionals in loops be followed by the block that continues around the loop.

8.6 Factorial example

```
(lambda (stack a0)
  (let* (((stack)  ($push-stack-environment stack))
         (()       ($set-environment stack 'global a0))
         ((a0)     ($contents a0 '<si>))
         ((d0)     ($read a0))
         (()       ($set-environment stack 'v4 d0))
         ((a0)     ($get-environment stack 'global))
         ((a0)     ($contents a0 '<si>))
         (()       ($read-line a0))
         ((d0)     ($move16 '1))
         ((d1)     ($move16 '1)))
    ($jump <loop> d0 d1)))

<loop> =
(lambda (d0 d1)
  (let* (((d2) ($get-environment stack 'v4)))
    ($equal16 (<true> <false>) d0 d2)))

<true> =
(lambda ()
  (let* (((a0)    ($get-environment stack 'global))
         ((a0)    ($contents a0 '<so>))
         (()      ($write d1 a1))
         ((a0)    ($get-environment stack 'global))
         ((a0)    ($contents a0 '<so>))
         (()      ($write-line a0))
         ((stack) ($pop-stack-environment stack)))
    ($simple-return () stack)))

<false> =
(lambda ()
  (let* (((d1) ($multiply16 d0 d1))
         ((d0) ($add16 d0 '1)))
    ($jump <loop> d0 d1)))
```

Following is the machine code produced for the factorial example. Each instruction is shown with the corresponding primitive application.

```
      lea     -6(a7),a7  ($push-stack-environment stack)
      move.l  a0,(a7)    ($set-environment stack 'global a0)
      move.l  (a0),a0    ($contents a0 '<si>)
      jsr     read       ($read a0)
      move.w  d0,4(a7)   ($set-environment stack 'v4 d0)
      move.l  (a7),a0    ($get-environment stack 'global)
      move.l  (a0),a0    ($contents a0 '<si>)
      jsr     read_line  ($read-line a0)
      moveq   #1,d0      ($move16 '1)
      moveq   #1,d1      ($move16 '1)
      bra     loop       ($jump <loop> d0 d1)
   false:
      muls.w  d1,d0      ($multiply16 d0 d1)
      addq.l  #1,d1      ($add16 d0 '1)
   loop:
      move.w  4(a7),d2   ($get-environment stack 'v4)
      cmp.w   d0,d2      ($equal16 (<true> <false>) d0 d2)
      bne     false
   true:
      move.l  (a7),a0    ($get-environment stack 'global)
      move.l  4(a0),a0   ($contents a0 '<so>)
      jsr     write      ($write d1 a1)
      move.l  (a7),a0    ($get-environment stack 'global)
      move.l  4(a0),a0   ($contents a0 '<so>)
      jsr     write_line ($write-line a0)
      lea     6(a7),a7   ($pop-stack-environment stack)
      rts                ($simple-return () stack)
```

Chapter 9

Compiler Extensions

The compilation method described in the previous chapters handles the basic programming language constructs used in many programming languages. It is also true that there are useful constructs that it does not implement or does so inefficiently. Typically these require manipulation of the continuation stack or the store in ways that are not provided for in the intermediate language. Efficient implementation can often be achieved by the adding new transformation phases to the compiler, usually to determine where the increased functionality is not needed.

Extensions to the compiler needed for several programming language constructs are presented here. Neither the list of constructs nor the techniques presented are meant to be exhaustive. The goal here is to show the flexibility of the compilation method in that it allows for modifications to the compiler when required for particular languages.

9.1 First-class continuations

In the programming language Scheme continuations are first-class objects. One implementation possibility would be to create a copy of the continuation stack in the heap and encapsulate it in a callable data structure. When the continuation was called, the stack would be copied back into its original location in memory and returned to. The compiler would have to be modified so that it would not move modifiable locations onto the stack.

Another possibility would be to have the Scheme front-end introduce its own continuations into the source. The compiler would then create heap closures for most continuations, although the stack would be used when the compiler found

continuations that did not escape upwards.

9.2 Tail recursion in Scheme

Scheme also requires that tail-recursive calls be implemented without using any finite resource (such as a continuation stack). To ensure this, the compiler could be modified so that procedures called tail-recursively would be considered to escape so that their environment will be created in the heap and not on the stack. It would then always be possible to remove the current stack environment before any nearly tail-recursive call, thus making the call truly tail-recursive.

9.3 Latent types

Another difficulty with Scheme (along with other Lisp dialects, APL [Gilmam 84], and other languages) is that it is not possible to determine the types of all objects at compile time. The data structures of an implementation must include the types of objects. This often introduces run-time overhead of one form or another in that type information must be added and removed from data. The compiler would need additional code improvement transformations to generate efficient code in the presence of typed data structures.

9.4 Lazy evaluation

Using the tranformational compiler to compile a lazy functional language such as ALFL [Hudak 84] would be simple, as such languages are based directly on the lambda calculus. However, the generated code would be quite inefficient without additional code improvement transformations. Specifically, some form of strictness analysis would be needed to remove some of the overhead of lazy evaluation.

9.5 Non-local return

Many languages contain some form of non-local return. This is equivalent to returning to a continuation other than the most recently created one. An example of this is `goto` in Pascal which may transfer control out of a procedure body bypassing the procedure's return. The obvious implementation is to put the code

following a `goto` label into a procedure and effect a `goto` by calling the appropriate procedure without creating a new continuation (this would require a special primitive procedure). This works for local `goto`s in that the compiler will determine that the procedure is always called with a known continuation on top of the stack and will make the `goto` procedure into a `cont` expression and thus the call will be compiled into a simple jump.

The difficulty with implementing non-local returns is the compiler's dependence on the fact that all calls to the value of a `cont` expression will be within the same procedure as the expression itself. Thus the `goto` procedure might have to remain a `proc` expression and calls would not be simple jumps. One solution to this would be to add another type of abstraction having the semantics of `lambda`. These abstractions, `escape` expressions, are like `cont` expressions in that they are not passed a continuation and may access their environment through the stack but, unlike `cont` expressions, may be called from within procedures other than the one containing the `escape` expression. The only restriction would be that the procedure containing the `escape` expression can not have returned (that is, called its continuation) when the expression is called.

I have come up with a variety of ways to translate `goto`s into `escape`, but as none of them are entirely pleasing I am still working on this problem. Adding `escape` procedures requires a few changes to the compilation transformations, usually just deciding whether, for a particular transformation, they are to be treated as `proc` expression or as `cont` expressions. They are called using primitive procedures as are the other abstractions. Additional code improvement transformations are needed to generate the desired machine code, including changing `escape` expressions to `cont` expressions if they turn out to be local returns after all.

Chapter 10

Results

The compilation method described in this thesis has been implemented in T [Rees 84], a dialect of Scheme. Two front-ends have been written, one for Pascal and one for Basic, along with the required primitive operations. In addition, there is a front-end for Scheme but without primitive operations or the necessary run-time system.

The Pascal and Scheme front-ends were developed along with the compiler. Once that was done, writing the Basic front-end and primitive operations took less than two days.

10.1 Timings

Several Pascal benchmarks have been used to compare the output of the implementation with that of a more traditional production compiler. The timings are shown here along with the times for the same programs compiled using the Apollo Pascal compiler. The Apollo Pascal compiler is a hand-coded compiler that does approximately the same optimizations as the compiler presented here. The main differences are that the Apollo compiler uses a less efficient procedure call mechanism but does some non-local register allocation and loop invariant code hoisting.

Except for `palindrome` the benchmark programs were gathered by John Hennessy and modified by Peter Nye. I obtained them from David Kranz.

Running Time

	Us	Them	Us / Them	Description
Fib	3.40	4.47	0.76	integer fibonacci
Bubble	0.72	0.64	1.09	bubble sort on 1000 integers
Quick	0.38	0.26	1.46	quicksort on 5000 integers
Palindrome	5.06	4.79	1.06	integer arithmetic operations
Perm	0.91	0.81	1.12	recursive array permutations
Towers	3.50	3.92	0.89	towers of hanoi

'Us' is the times for the benchmarks as compiled by the transformational compiler, 'Them' is the times when compiled using the Apollo Pascal compiler. All times are in seconds. The third column contains the ratio of the two times.

The transformational compiler's output runs a bit slower except in the case of `Fib` and `Towers`, both of which do a fair number of procedure calls.

Compilation Time

	Us	Them	Us / Them
Fib	3.40	0.45	7.56
Bubble	9.70	1.00	9.70
Quick	12.28	1.50	8.19
Palindrome	16.43	1.96	8.38
Perm	7.01	0.71	9.87
Towers	17.02	1.88	9.05

This table shows the time it takes the two compilers to compile the benchmarks. Not surprisingly, as little effort was made to speed it up, the transformational compiler is much slower. Its compilation speed comes out to between five and seven lines of Pascal per second on these programs.

10.2 Results and future work

This dissertation began with five problems with compilers:

1. Different ones are needed for different languages
2. Different ones are needed for different machines
3. Many do not implement the source language correctly

4. Their output is often inefficient

5. They run slowly

The compilation transformations described in this dissertation are correct, an implementation of them can compile programs written in a variety of programming languages and get output that is as good as that of a hand-coded compiler doing the same optimizations. What remains to be done is to work towards the ideal compiler described at the beginning of Chapter 1. More front-ends need to be written, the compiler should be ported to other machines, the implementation needs more debugging, more code improvement transformations are needed, and the compiler must be made faster.

My current goal is to finish the Scheme primitives and use the compiler as part of a full Scheme implementation. A front-end for a pure, lazy functional language is also planned, along with associated transformations based on strictness and related analyses. The compiler will also be ported to different machines.

There are numerous large and small code improvement transformations that can be added: moving loop-invariant code out of loops, lambda hoisting to reduce consing in Scheme programs, taking advantage of commutative operation during register allocation, and so on.

Speeding up the current implementation is in part just a matter of program engineering.

Bibliography

[Aho 86] Alfred V. Aho, Ravi Sethi, and Jeffery D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, 1986.

[Barrett 79] William A. Barrett and John D. Couch. *Compiler Construction: Theory and Practice.* Science Research Associates, 1979.

[Marateck 75] Samuel L. Marateck. *Basic.* Academic Press, 1975.

[Boyle 84] James M. Boyle and Monagur N. Muralidharan. Program reusability through program transformation. In *IEEE Transactions on Software Engineering* SE-10(5), September 1984.

[Boyle 86] James M. Boyle, Kenneth W. Dritz, M .N. Muralidharan, and Robert J. Taylor. Deriving sequential and parallel programs from pure Lisp Specifications by program transformation. In *IFIP WG2.1 Working Conference on Programme Specifications and Transformations.*

[Brooks 82] Brooks, R.A., Gabriel, R.P. and Steele, G.J. Jr. An optimizing compiler for lexically scoped LISP. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, ACM, SIGPLAN Notices 17(6), June 1982.

[Clinger 84] William Clinger. The Scheme 311 Compiler. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, ACM, 1984.

[Chaiten 82] G. J. Chaiten. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, ACM, SIGPLAN Notices 17(6), June 1982.

[DeMillo 78] Richard A. DeMillo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. Department of Computer Science Research Report 136, Yale University, June 1978.

[Fisher 81] Joseph A. Fisher. Trace Scheduling: a technique for global microcode compaction. In *IEEE Transactions on Computers* c-30(7), July 1981.

[Feeley] M. Feeley and G. Lapalme. Closure generation based on viewing LAMBDA as EPSION plus COMPILE. Département d'informatique et de recherche opérationnelle (I.R.O.), Université de Montréal, P.O.B. 6128, Station A, Montréal, Québec, H3C3J7 (Canada).

[Gilmam 84] Leonard Gilman and Allen J. Rose. *APL: An Interactive Approach.* John Wiley & Sons, 1984.

[Hudak 84] Paul Hudak. ALFL Reference Manual and Programmers Guide. Technical Report YALEU/DCS/TR-322, Department of Computer Science, Yale University, October 1984,

[Gordon 79] Michael J. C. Gordon. *The Denotaional Description of Programming Languages.* Springer-Verlag, 1979.

[Jensen 74] Kathleen Jensen and Niklaus Wirth. *Pascal: User Manual and Report*. Springer-Verlag, 1974.

[Johnsson] Thomas Johnsson. Lambda lifting: Transforming programs into recursive equations. In *Compiling Lazy Functional Languages.* PhD thesis, Chalmers University of Technology, 1987

[Knuth 68] Donald E. Knuth. Semantics of context-free languages. In *Mathematical Systems Theory* 2(2):127-145, February 1968.

[Kranz 86] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams Orbit: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, ACM, SIGPLAN Notices 21(7), June 1986.

[Lee 87] Peter Lee. *The Automatic Generation of Realistic Compilers from High-level Semantics Descriptions*. PhD thesis, University of Michigan, 1977.

[Motorola 85] Motorola Inc. *MC68020: 32-bit Microprocessor User's Manual.* Prentice-Hall, 1985.

[Paulson 82] Lawrence Paulson. A semantics-directed compiler generator. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, ACM, 1982.

[Plotkin 75] G. D. Plotkin. Call-by-name, call-by-value and the λ-calculus. In *Theoretical Computer Science* 1:125-159, 1975.

[Rees 84] Jonathan A. Rees, Norman I. Adams, and James R. Meehan.The T manual, fourth edition. Yale University Computer Science Department, January 1984.

[Rees 86] Jonathan Rees and William Clinger, editors. The revised^3 report on the algorithmic language Scheme. In *SIGPLAN Notices*, 21(12), December 86.

[Schmidt 86] David A. Schmidt. *Denotational Semantics.* Allyn and Bacon, 1986.

[Slade 87] Stephen Slade. *The T Programming Language*. Prentice-Hall, Inc. 1987.

[Standish 76] T. A. Standish, D. C. harriman, D. F. Kibler, and J. M. Neighbors. The Irvine program transformation catalogue Department of Information and Computer Science, University of California at Irvine, 1976.

[Steele 78] Guy L. Steele Jr.Rabbit: a compiler for Scheme. MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.

[Stoy 77] Joseph E. Stoy. *Denotational Semantics: the Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.