Realistic Compilation by Program Transformation

— Detailed Summary —

Richard Kelsey
Paul Hudak

Yale University
Department of Computer Science

Abstract

Using concepts from denotational semantics, we have produced a very simple compiler that can be used to compile standard programming languages and produces object code as efficient as that of production compilers. The compiler is based entirely on source-to-source transformations 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, BASIC, and Scheme and the compiler produces code for the MC68020 microprocessor.

1 Introduction

Denotational semantics has been used as a tool to prove compilers correct and to write compiler generating programs. However it has not generally been used to understand the nature of the compilation process itself. For example, most semantics-directed compiler generators begin with a standard denotational semantics description of the source language in which stores, environments, etc. have been made explicit. They then attempt to map the result onto the target architecture, which is the crux of the compilation process, and not such an easy task. Thus denotational semantics only takes care of the easy part: translating the program into an intermediate "meta-language" (the lambda calculus with constants) which still requires significant compilation.

In contrast, our approach has been somewhat more pragmatic, but still sound from the perspective of the formal semantics. Our intermediate language is the call-by-value lambda calculus with data and procedure constants (as in [Plotkin 75]), but with the addition of an implicit store. The target language is a generic register transfer language whose is a subset of the intermediate language's syntax but with a completely different semantics. Thus compilation first consists of transforming the source program into the intermediate language, and then performing source-to-source transformations on the intermediate program until it has the same meaning when considered as either an intermediate language program or a machine language program.

Although this process sounds simple enough, the reasons behind it are more significant. Our motivation stems from the observation that the compilation process is in fact a transformation process, and thus the best way to understand it is simply to look at the source language and compare it feature for feature with the target language. From this perspective it makes sense that common features should be left unaltered. Then by concentrating on the *differences*, the very essence of the compilation (i.e. transformation) process unfolds.

This research was supported in part by the National Science Foundation under Grant CCR-8451415, and the Department of Energy under Grant DE-FG02-86ER25012. The authors' current addresses are:

Richard Kelsey	Paul Hudak
NEC Research Institute	Yale University
4 Independence Way	Box 2158 Yale Station
Princeton, NJ 08540	New Haven, CT 06520
kelsey@research.nj.nec.com	hudak@cs.yale.edu

The compiler described here is still based on denotational semantic descriptions of its intermediate and target languages, and thus its correctness is no harder (we feel easier) to prove than that of a "traditional" semantics-directed compiler. Formally, the correctness of the heart of the compiler is captured as follows:

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

where: $\mathcal{S}_i$ = *intermediate language semantics*
$\mathcal{S}_m$ = *machine language semantics*
$\mathcal{P}$ = *output of a language specific front-end*
$\mathcal{C}$ = *compilation transformations*

Using this methodology we have developed a compiler with the following features:

1. The intermediate language is general enough and powerful enough that many common programming languages can be compiled easily.
2. The output runs as fast as that produced by a production compiler.
3. It is simple and easy to show correct.

We believe this is the first compiler to possess all of these features.

2 The Intermediate Language

The intermediate language used in the compiler is the basis of the entire compilation strategy. Its syntax is given in figure 1. As mentioned above, it is essentially the call-by-value lambda calculus with data and procedure constants and an implicit store. As only the primitive procedures have access to the store (in particular, variables are not set, as will be explained later), it is largely invisible in the semantics as well as in the language itself. However, its presence does rely on the use of continuations in the semantics to specify the order in which applications of primitive procedures use and modify the store.

The intermediate language is also simple in that it has only a few types of expressions, each having a simple semantics. This makes the compiler much less complex and easier to understand. Yet the intermediate language is quite expressive in that it allows first-class procedures, which can be made recursive through the use of the store. Having first-class procedures, along with the implicit store, make it easy to write front-ends for many modern programming languages.

As can be seen from figure 1 the intermediate language is the lambda calculus with some additions:

K ∈ Con	constants
I ∈ Ide	identifiers
P ∈ Pri	primitives (procedure constants)
L ∈ Pro	procedures
	⟶ `(proc I (I*) E)`
C ∈ Sab	simple abstractions
	⟶ `(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)`

Figure 1: The syntax of the intermediate language

The `block` form is a sequencing construct that evaluates its expressions in order and returns the result of the last.

There are three types of 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. Only `lambda` abstractions are used in initial input to the compiler. During the compilation process these are replaced with either `proc` or `cont` as a method of annotating how the abstraction is used in the program: `proc` expressions are passed an explicit continuation argument and `cont` expressions are not.

Calls to primitive procedures represent machine operations. Associated with each primitive procedure is all of the information needed to generate a particular sequence of machine instructions. This information includes the type(s) of values the primitive returns, how the primitive uses the store, and which registers may be used to hold its arguments and return values. Calls to primitives have zero or more continuation arguments – a call with no arguments returns a value, otherwise one of the continuation arguments is called instead of returning.

Note that the syntax of the intermediate language does not allow calls to arbitrary expressions; thus there is no default calling convention. Instead, primitive operations are used to specify call and return conventions for procedures. This simplifies the compiler as all calls may be treated identically. Returns are done using the `(return P E*)` form in which the primitive P specifies the way in which the values are to be returned from the lexically innermost abstraction.

3 The Machine Language

The machine language is an abstract assembly language written in a subset of the syntax of the intermediate language, but with a completely different semantics. Only identifiers, constants, `lambda` expressions, and calls to primitive procedures are allowed, where

- The identifiers of the machine language represent the registers of the machine. Thus there are not very many of them and they are not lexically scoped, but rather are locations whose values are in the store.
- `lambda` expressions that are not continuations to calls to primitive procedures represent code pointers, and their identifiers are ignored as arguments are passed in the store. Calls must place the arguments into the registers in which the procedure expects to find them.
- The primitive procedures are the machine's instructions, and 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 syntax of the intermediate and machine languages and in a conventional assembler syntax:

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

In the intermediate language this adds the values of `r1` and `r2` and calls the continuation argument on the result. In the machine language this adds the contents of registers `r1` and `r2` and places the result in `r2`. A much larger example giving an intermediate code program and the corresponding assembly language program can be found in figure 11.

4 The Transformation Process

As mentioned earlier, we can gain some insight into the nature of the transformation process by comparing the source or intermediate language feature for feature with the machine language, as shown in figure 2. The entire problem of compilation lies in using properties of the second to implement the first.

Note that the store and call-by-value semantics are essentially the *same* in both languages – thus there is nothing to be changed, and the transformation process can ignore them. This both reduces the work that the compiler must do and provides a useful tool, the store, to be used in implementing the rest of the intermediate language.

On the other hand, the first three items are *different*. Based on this simple observation, we can summarize what is required of the compiler as follows:

4.1 Implementing returns as calls

Call and return must be implemented in terms of goto. This is done in two steps: 1) the program is first made *linear* and every temporary value is explicitly bound to an identifier; 2) the `lambda` expressions that bind the temporary values are converted to *explicit continuations* to the calls that produce the values. The explicit continuations are identical to the continuations used in the denotational semantics of the intermediate language, and will eventually be manifested as gotos in the machine code. The resulting program contains only calls – there are no returns – and thus procedure calls no longer need to save a return point.

Intermediate	Machine
Call and return.	Goto.
Nested lexical scoping.	Flat scoping.
Large set of identifiers.	Small set of identifiers.
A store.	A store.
Call by value.	Call by value.

Figure 2: Properties of the intermediate and machine languages

4.2 Transform nested lexical scoping into flat scoping

Just as continuations from the denotational semantics were added to implement transfer of control, now *explicit environments* from the denotational semantics are added to implement lexical scoping. As in the denotational semantics these environments act as an indirection to the store; but whereas the store is normally explicit, in our case it is implicit, corresponding to the realities of the target machine.

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, and procedure calls become gotos that pass arguments.

4.3 Restrict the use of identifiers

Finally, the program must be transformed so that it only uses the small set of identifiers of the machine language, which correspond to the finite resources of the machine. Registers must be allocated to hold the values of the identifiers in the program, and any necessary calls added to move values between registers or to temporarily save values in the store. At this point procedure calls have been reduced to nothing but simple gotos, since the arguments are passed in the store. Again, this agrees with the reality of the machine.

The program now has the same meaning whether it is interpreted as an intermediate language program or as a machine language program and can be viewed as an assembly language program (with a somewhat unusual syntax) for the target machine.

5 The Compilation Process in Detail

To summarize, the compilation process is performed in six steps:

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

Each step restricts the form of the code, and subsequent steps must preserve previous 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 code improvement transformations work to reduce the size of the code, as each expansion of the code typically provides more opportunities to improve it.

Much of the design of the compiler is oriented towards reducing the cost of saving and accessing lexical environments. As the intermediate language is lexically scoped environments may need to be preserved for later use. The compiler uses both a heap and a stack for allocating environments. Stack environments are accessed either through the current stack pointer or through an explicit environment pointer; heap environments always use an explicit environment pointer. As heap environments and explicit environment pointers are less efficient than stack environments and implicit environment pointers the goal of the compiler is to use as few heap environments and explicit environment pointers as possible.

In the remainder of this section we describe each of these steps detail. For even more detail, see [Kelsey 89].

5.1 Translating Source Code into Intermediate Language

Initially, a front-end specific to the source language translates the program into the compiler's intermediate language. Different front-ends are needed for different programming languages. Each consists of a translator from the source language into the intermediate language, a set of primitive procedures describing all of the machine operations that the source language requires, and a mapping from source language constants to machine data. The translator is normally a simple syntax-directed translator very much like the denotational semantics for the source language. Every type of expression in the syntax of the source language has a template that gives an equivalent expression in the intermediate language in terms of the translations of the expression's subexpressions. The primitive procedures and the description of constants are not normally specified in denotational semantics but are necessary to compile programs efficiently.

Standard techniques from denotational semantics are used in translating programs into the intermediate language. The values of variables are kept in locations in the store, allowing the variables to be set to new values. Conditional expressions are implemented using primitive procedures that take more than one continuation argument, only one of which is actually called. Control constructs such as loops require the use of recursive procedures. In denotational semantics recursion is normally done using a fixed-point operator. The intermediate language contains no such operator. Instead, the procedure is stored in a location that is lexically visible within the procedure itself. The procedure can then dereference the location to obtain itself as a value.

Currently there are complete front-ends for Pascal, BASIC, and Scheme (although the latter does not yet have all of the Scheme primitives and run-time system defined). The Pascal and Scheme front-ends were developed along with the compiler. To give some idea of source language portability, writing the BASIC front-end and its primitive operations took less than two days.

5.2 Making the program linear

This transformation gives an explicit order to the applications in the program, introduces identifiers for all temporary values, and removes the `block` expressions. In the resulting code the arguments are never applications (except for arguments to applications of `lambda` expressions with only one argument) and thus the code is linear in that the calls are explicitly ordered.

As an example of the transformation to linear code, in the expression

```
(lambda (x y)
  (return $return ($+ () x ($* () y 2))))
```

the result of the call to `$*` is an anonymous temporary value (primitive procedures begin with a `$` and `$return` is a primitive used to return a single value; remember that procedure call and return conventions must be specified with primitive operations). The transformation converts this expression into

```
(lambda (x y)
  ((lambda (v1)
     ((lambda (v2) (return $return v2))
      ($+ () x v1)))
   ($* () y 2)))
```

where `v1` is the identifier introduced for the result of the call to `$*` and similarly for `v2` and the call to `$+`.

5.3 Adding Explicit Continuations

Here the compiler moves the `lambda` expressions introduced by the previous transformation into the applications themselves as continuations. The transformation is given in [Plotkin 75], but made slightly more complex here due to a more complicated syntax and a desire to limit the size of the resulting program. [1]

All `lambda` expressions are replaced with either `cont` or `proc` expressions. Those that are in call position or are continuations to primitive calls become `cont` expressions. The remaining `lambda`s become `proc` expressions with an additional identifier for the continuation to be called when the procedure has finished. The `return` calls in the procedure are replaced with calls to the continuation identifier.

As mentioned above, all three types of abstraction expressions have the same semantics; the only difference is a syntactic one in that `proc` expressions have a distinguished identifier that is bound to the procedure's continuation. An important distinction between `cont` and `proc` expressions is that the environments for `cont` expressions can always be allocated on the stack and are always at a known offset from the current stack pointer. The environments for `proc` expressions may be in the heap or the stack but must be accessed through an explicit environment pointer. This distinction is a natural one in that as `cont` expressions do not get passed a continuation, the current continuation (and thus the current stack value) must be known at compile time. For a `cont` expression to be always called with the same current continuation all calls to the value of the expression must be in the body of the same `proc` expression as the `cont` itself. If any calls were in the body of a second `proc` expression that call would occur with the stack environment created by the second `proc` expression on the top of the stack, instead of that created by the `proc` expression containing the `cont`.

After continuations have been added to the program the only `cont` expressions are either continuation arguments to primitives or were introduced during the transformation to avoid the duplication of continuation arguments to primitives. In either case the calls to the `cont` expression meet the above criterion.

To continue the example used above, the transformed program would be:

```
(proc c (x y)
  ($* ((cont (v1)
         ($+ ((cont (v2)
                ($return () c v2)))
             x
             v1)))
       y
       2))
```

The resultant program is in *continuation passing style* (CPS) and is now more structured than before:

1. Arguments to calls may no longer be calls.
2. The bodies of abstractions are now always calls.
3. There are no longer any returns.

The parts of the compiler that follow must preserve the continuation passing nature of the transformed program.

5.4 Simplifying the Program

Conversion to continuation passing style is followed by a number of code improving transformations, many of which are well known [Steele 78,Brooks 82,Kranz

[1] [Steele 78, Kranz 86] also combine this with the previous transformation. It seems somewhat simpler as two seperate transformations.

86,Standish 76]. These include both local transformations such as beta-reduction, and two global transformations, one of which is based on flow analysis. The transformations are simpler when done after conversion to continuation passing style as the code is more structured. For example, beta substitution may be done without reference to side-effects as arguments to applications are never applications themselves.

The first global transformation substitutes known values for identifiers bound by abstractions that are used in more than one application. For example, if the `(proc (c x) ...)` is called in two places and in both cases the value of `x` is `y` and `y` is lexically visible in the `proc` expression, then `y` will be substituted for `x`. If the value being substituted is a continuation, then the `proc` expression becomes a `cont` as long as the scoping restrictions detailed above are met. This allows many procedures, including recursive procedures such as those introduced to compile iterative loops, to become `cont` expressions and calls to these procedures to become simple jumps. Thus the requirement that all control constructs in the source language be implemented using procedures and procedure calls does not prevent the compiler from producing simple and efficient code for those control constructs.

The second 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 `cont` expression to `cont` expression instead of implicitly in the store. This is equivalent to classical definition-use flow analysis with the results of the analysis expressed in the program itself, allowing the other transformations, such as beta-substitution, to implement copy propogation, constant folding, and other flow analysis based optimizations. Each `proc` expression is transformed separately and thus only some uses of some locations may be removed by this transformation.

5.5 Adding Explicit Environments

As stated in section 4.2 the environments of the intermediate language's semantics are added to the program. This results in a program where the only abstractions that may have free identifiers are continuations to calls to primitive procedures. The register allocator described in the next section 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.

As an example, here is a procedure that takes an argument `x` and returns a procedure that returns `x` when called (see section 6 for a expansion of `let*` into the internal language):

```
(proc c1 (x)
  ($return () c1 <PROC>))

<PROC> =
(proc c2 ()
  ($return () c2 x))
```

In the transformed code shown below, both procedures are passed their environment as an additional argument and construct environments for their lexically inferior procedures (only one will be shown as the inner procedure's environment is not used):

```
(proc c1 (e1 x)
  (let* ((e3 ($make-environment)))
    ($return () c1 <PROC>)))

<PROC> =
(proc c2 (e2)
  ($return () c2 x))
```

The value of `x` is added to the environment in the outer procedure and obtained from it in the inner:

```
(proc c1 (e1 x)
  (let* ((e3 ($make-environment))
         (i1 ($set-environment e3 'x x)))
    ($return () c1 <PROC>)))

<PROC> =
(proc c2 (e2)
  (let ((x1 (get-environment e2 'x)))
    ($return () c2 x1)))
```

The call to `$set-environment` does not return a meaningful value, but it does modify the store. The final change is to add the code for the inner procedure, which is now a constant as it contains no free variables, to the environment so that the two may be returned as a single value:

```
(proc c1 (e1 x)
  (let* ((e3 ($make-environment))
         (i1 ($set-environment e3 'x x)))
         (i2 ($set-environment e3 'p '<PROC>))
         (p  ($make-procedure e2 'p)))
    ($return () c1 p)))

<PROC> =
(proc c2 (e2)
  (let ((x1 (get-environment e2 'x)))
    ($return () c2 x1)))))
```

The only analysis required for the addition of environments is a determination of which procedures require heap environments and which may use stack environments. This is done either by fiat, as in Pascal where the language design ensures that procedures are never returned upwards, or program analysis, in the case of a language such as Scheme. There are two cases in which procedures require heap environments: the procedure has a use that is not a call to that procedure, such as being passed to another procedure; or the procedure has a calling point within a procedure that both requires a heap environment and is not lexically superior to the called procedure.

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

5.6 Identifier Renaming / Register Allocation

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 registers and functional units can be done in any manner but the allocation 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.

The current implementation does one form of instruction selection in that it 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. The current implementation uses a very simple register allocation algorithm that allocates registers for each basic block separately. 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.

6 Factorial Example

By far the best way to understand the transformations and their effects is to follow the compilation of a simple program. As an example of the compiler in action, the steps in compiling a very simple Pascal program will be presented here. The sample program, shown in 3, reads in an integer x and prints out the value of $\mathtt{x}! = 1 * 2 * \ldots * \mathtt{x}$.

```
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.
```

Figure 3: Sample Pascal program

While CPS code is easy for programs to analyze it is very hard to read and some syntactic sugaring makes the code much more comprehensible. The syntax that will be used here is a variation on Scheme's `let*` syntax.

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

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

```
((id1 id2 ...) ($p arg1 arg2 ...)) {rest}
    ⇕
($p (cont (id1 id2 ...) {rest}) arg1 arg2 ...)
```

In the `let*` notation each basic block of the program becomes a single `let*` ending in a primitive call with either more than one continuation argument or none at all.

Figure 4 show the factorial program after the compiler has converted the code into CPS and done some simplification, including substituting the body of the factorial procedure at its one calling point. The first block is the body of the program, which reads a value for x, introduces locations for the variables i and r and the recursive procedure needed for the loop, calls the looping procedure, and writes out the value of r. The location introduced for x has been removed as its contents was set only once and the value could be substituted at all other uses of the location. A location is needed for the recursive procedure implementing the loop as that is the most efficient way of expressing recursion in the intermediate language.

The body of the loop tests the value of r and either

```
(proc p.39 ()
  (let* (((t.14) ($read input))
         (()     ($read-line input))
         ((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) ($contents p.19))
         (()     ($write t.23 output))
         (()     ($write-line output)))
    ($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)))
```

Figure 4: Factorial in CPS

returns or does the multiply, adds one to i, and then calls itself recursively.

The names of the introduced identifiers reflect the runtime values they represent: p for pointers and t for other values. $push is a primitive for producing new locations that can be allocated on the stack; its argument is the size of the location in bits. $contents and $set-contents read and write the contents of locations.

In figure 5 the program has been simplified by changing the loop procedure from a cont to a proc and its continuation has been substituted into the body of the procedure. The two calls to the loop now use $jump as there is no longer any continuation argument.

The second global simplifying transformations modifies the program to pass the contents of the locations p.18 and p.19 (which hold the values of i and r) explicitly as t.40 and t.41 as shown in figure 6. This is the code at the end of the code improvement phase of

```
(proc p.39 ()
  (let* (((t.14) ($read input))
         (()     ($read-line input))
         ((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) ($contents p.19))
         (()     ($write t.23 output))
         (()     ($write-line output)))
    ($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)))
```

Figure 5: The loop becomes a jump

the compiler.

Figure 7 shows the program after the introduction of environments. This example does not require much in the way of simplifications other than removing unused calls. Only p.45 (the global environment passed to the program) and t.14 (the value of x) are kept in an environment. Once the <LOOP> procedure (now a constant as it has no free variables) 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.

Finally, in figure 8 all of the identifiers have been renamed with the registers that will contain their values. Two calls to $move16 are used to move constants into registers. Even this small program shows up the lack of sophistication in the current register allocator. The

```
(proc p.39 ()
  (let* (((t.14) ($read input))
         (()     ($read-line input))
         ((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* ((()     ($write t.41 output))
         (()     ($write-line output)))
    ($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)))
```

Figure 6: Locations removed

lack of a global register allocation scheme is shown in that the value of `x` is loaded from the stack environment every time around the loop instead of remaining in a register. This load could also be avoided by using an indirect operand to the `cmp` instruction that is emitted for the `$equal16` primitive.

7 Results

Our compiler is written in T [Rees 84], a dialect of Scheme, and generates code for the Motorola MC68020 microprocessor. As mentioned above, two front-ends have been written, one for Pascal and one for Basic, along with a front-end for Scheme that lacks all of the necessary primitive operations and an appropriate run-time system. The Pascal and Scheme front-ends were developed along with the compiler. Writing the Basic front-end and primitive operations took less than two days.

Several Pascal benchmarks have been used to compare the output of the implementation with that of a more traditional production compiler.[2] The timings are shown here along with the times for the same programs

[2] Except for `palindrome` the benchmark programs were gathered by John Hennessy and modified by Peter Nye.

```
(proc p.39 (p.45)
  (let* (((p.42) ($push-stack-environment p.39))
         (()     ($set-environment p.42 'p.45 p.45))
         ((p.46) ($contents p.45 'input))
         ((t.14) ($read p.46))
         (()     ($set-environment p.42 't.14 t.14))
         ((p.47) ($contents p.45 '(si)))
         (()     ($read-line p.47)))
    ($jump <LOOP> '1 '1)))

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

<TRUE> =
(cont ()
  (let* (((p.43) ($get-environment p.42 'p.45))
         ((p.48) ($contents p.43 'output))
         (()     ($write t.48 p.11))
         ((p.49) ($contents p.43 'output))
         (()     ($write-line p.49)))
    ($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)))
```

Figure 7: Environments added

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, and is used extensively by Apollo in production software (for example, their entire operating system is written predominantly in Pascal). The main difference between the two compilers is that the Apollo compiler uses a less efficient procedure call mechanism, but does some non-local register allocation and loop invariant code hoisting that we do not do.

Figure 9 lists several Pascal programs that have been compiled and run using the compiler. Figure 10 gives the running times for these programs. "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.

Note that the transformational compiler's output runs somewhat slower for four of the programs, and in the other two, `Fib` and `Towers`, it runs somewhat faster (both of which perform a large number of procedure calls). We consider these results to be quite good, and

```
(lambda (sp a0)
  (let* (((sp) ($push-stack-environment sp))
         (()   ($set-environment sp 'p.45 a0))
         ((a0) ($contents a0 'input))
         ((d0) ($read a0))
         (()   ($set-environment sp 'v4 d0))
         ((a0) ($get-environment sp 'p.45))
         ((a0) ($contents a0 '<si>))
         (()   ($read-line a0))
         ((d0) ($move16 '1))
         ((d1) ($move16 '1)))
    ($jump <loop> d0 d1)))

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

<true> =
(lambda ()
  (let* (((a0) ($get-environment sp 'p.45))
         ((a0) ($contents a0 'output))
         (()   ($write d1 a0))
         ((a0) ($get-environment sp 'p.45))
         ((a0) ($contents a0 'output))
         (()   ($write-line a0))
         ((sp) ($pop-stack-environment sp)))
    ($simple-return () sp)))

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

Figure 8: register allocation

perhaps surprising, given the simple structure of our compiler. We also do not consider our compiler to be complete, in that many more standard optimizations could be implemented without undue effort.

8 Related work

There have only been a few *realistic* compiler generators, such as those of Paulson [Paulson 82] and Lee [Lee 87], written using denotational semantics or attribute grammar descriptions of the source languages as input. However, we do not feel they have been realistic enough, nor could they be called simple. The crucial distinction between the "pure" semantics-directed compiler generators and the compilation method described here is that although both may translate the source program into a form of the lambda calculus, here the identifier bindings, continuations, and the store of the source program are actually *implemented* using the bindings and continuations of the lambda calculus along with an implicit store. This restricts the ways in which the bindings, continuations, and store can be used by the source program, but allows the compiler to implement them *efficiently*. By viewing the compilation process as a transformational one, we are able to concentrate on the differences between the intermediate "meta-language" and machine language, yielding a *more direct path* between intermediate and machine code. As a result, much of the utility and generality of using the lambda calculus to describe programming languages can be obtained without paying the performance cost of compiling and running general lambda calculus programs.

Fib	integer fibonacci
Bubble	bubble sort on 1000 integers
Quick	quicksort on 5000 integers
Palindrome	integer arithmetic operations
Perm	recursive array permutations
Towers	towers of hanoi

Figure 9: Benchmark Programs

	Us	Them	Us / Them
Fib	3.40	4.47	0.76
Bubble	0.72	0.64	1.09
Quick	0.38	0.26	1.46
Palindrome	5.06	4.79	1.06
Perm	0.91	0.81	1.12
Towers	3.50	3.92	0.89

Figure 10: Benchmark Results

On the negative side, our methodology is somewhat less general than the pure approaches in that our intermediate language is "biased" somewhat in the direction of sequential, register-based, uni-processors. On the other hand, the *compilation strategies* of the pure approaches are also certainly biased in that direction. To retarget our approach to a radically different architecture would require changing our intermediate language as well as some of the transformations, whereas retargeting a pure approach would require considerable changes to the compilation strategy itself.

Some parts of our compilation by transformation methodology are related to other approaches to compiler design. For example, passing continuations as explicit arguments has been used in [Steele 78, Kranz 86]. Pass-

```
                              (lambda (sp a0)
   lea     -6(sp),sp            (let* (((sp) ($push-stack-environment sp))
   move.l  a0,(sp)                     (()   ($set-environment sp 'p.45 a0))
   move.l  (a0),a0                     ((a0) ($contents a0 'input))
   jsr     read                        ((d0) ($read a0))
   move.w  d0,4(sp)                    (()   ($set-environment sp 'v4 d0))
   move.l  (sp),a0                     ((a0) ($get-environment sp 'p.45))
   move.l  (a0),a0                     ((a0) ($contents a0 'input))
   jsr     read_line                   (()   ($read-line a0))
   moveq   #1,d0                       ((d0) ($move16 '1))
   moveq   #1,d1                       ((d1) ($move16 '1)))
   bra     loop                   ($jump <loop> d0 d1)))

false:                        (lambda ()
   muls.w  d1,d0                (let* (((d1) ($multiply16 d0 d1))
   addq.l  #1,d1                       ((d0) ($add16 d0 '1)))
                                  ($jump <loop> d0 d1)))

loop:                         (lambda (d0 d1)
   move.w  4(sp),d2             (let* (((d2) ($get-environment sp 'v4)))
   cmp.l   d0,d2                  ($equal16 (<true> <false>) d0 d2)))
   ble     false

true:                         (lambda ()
   move.l  (sp),a0              (let* (((a0) ($get-environment sp 'p.45))
   move.l  4(a0),a0                    ((a0) ($contents a0 'output))
   jsr     write                       (()   ($write d1 a0))
   move.l  (sp),a0                     ((a0) ($get-environment sp 'p.45))
   move.l  4(a0),a0                    ((a0) ($contents a0 'output))
   jsr     write_line                  (()   ($write-line a0))
   lea     6(sp),sp                    ((sp) ($pop-stack-environment sp)))
   rts                            ($simple-return () sp)))
```

Figure 11: The machine code produced for the factorial example

ing environments as explicit arguments is one of our key innovations, but is related to "lambda-lifting" as used in functional language compilers [Johnsson 87] and to the method described in [Feeley]. The use of an intermediate language to allow compilation of more than one language is fairly 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, together with a few key innovations, these various techniques have been put together into one common framework. Transformations are used exclusively, and only on programs in one intermediate language. The semantics of the intermediate and target languages are specified denotationally, and the compilation transformations are based directly on the differences and similarities of the two languages. The result is a very simple compiler that generates surprisingly good code.

References

[Boyle 84]
James M. Boyle and Monagur N. Muralidharan.
Program reusability through program transformation.
in *IEEE Transactions on Software Engineering* SE-10(5):574-588, 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.

[Feeley]
M. Feeley and G. Lapalme.
Closure generation based on viewing LAMBDA as EPSILON 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).

[Johnsson 87]
Thomas Johnsson.
Lambda lifting: Transforming programs into recursive equations.
In *Compiling Lazy Functional Languages.*
PhD thesis, Chalmers University of Technology, 1987

[Kelsey 89]
Richard Kelsey.
Compilation by Program Transformation.
PhD thesis, Yale University, 1989.

[Kranz 86]
Kranz D.A., Kelsey, R., Rees J.A., Hudak P., Philbin, J. and Adams, N.I.
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, 1987.

[Paulson 82]
Lawrence Paulson.
A semantics-directed compiler generator.
in *Converence 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.

[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.