MinCaml: A Simple and Efficient Compiler for a Minimal Functional Language *

Eijiro Sumii
Tohoku University
sumii@ecei.tohoku.ac.jp

Abstract

We present a simple compiler, consisting of only 2000 lines of ML, for a strict, impure, monomorphic, and higher-order functional language. Although this language is minimal, our compiler generates as fast code as standard compilers like Objective Caml and GCC for several applications including ray tracing, written in the optimal style of each language implementation. Our primary purpose is education at undergraduate level to convince students—as well as average programmers—that functional languages are simple and efficient.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers; D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages

General Terms Languages, Design

Keywords ML, Objective Caml, Education, Teaching

1. Introduction

The Meta Language, or ML, is a great programming language. It is one of the very few languages that achieve rather challenging and often conflicting demands—such as efficiency, simplicity, expressivity, and safety—at the same time. ML is the only language ranked within the top three *both* for efficiency (runtime speed) and for simplicity (code lines) at an informal benchmark site [2] that compares various programming language implementations. ML is also *the* language most used by the winners of the ICFP programming contests [3].

Unfortunately, however, it is also an undeniable fact that ML is a "Minor Language" in the sense that it is not as widespread as C or Perl, even though the situation is getting better thanks to mature implementations such as Objective Caml.

Why is ML not so popular? The shortest answer is: because it is not well-known! However, looking more carefully into this obvious tautology, I find that one of the reasons (among others [24]) for this "negative spiral of social inertia" is misconceptions about implementations. Nowadays, there are a number of programmers who learn ML, but they often think "I will not use it since I do not understand how it works." Or, even worse, many of them make incorrect assumptions based on arbitrary misunderstanding about implementation methods. To give a few real examples:

- "Higher-order functions can be implemented only by interpreters" (reason: they do not know function closures).
- "Garbage collection is possible only in byte code" (reason: they only know Java and its virtual machine).
- "Functional programs consume memory because they cannot reuse variables, and therefore require garbage collection" (reason: ???).

Obviously, these statements must be corrected, in particular when they are uttered from the mouths of our students.

But how? It does not suffice to give short lessons like "higher-order functions are implemented by closures," because they often lead to another myth such as "ML functions are inefficient because they are implemented by closures." (In fact, thanks to known function call optimization, ML functions are just as efficient as C functions if they can be written in C at all—except that function *pointers* can sometimes be used in more efficient ways than function closures.) In order to get rid of the essential prejudice that leads to such ill-informed utterances as above, we end up in giving a full course on how to implement an efficient compiler of a functional language. (Throughout this paper, an efficient compiler means a compiler that generates fast code, not a compiler which itself is fast.) To this goal, we need a simple but efficient compiler which can be understood even by undergraduate students or average programmers.

The MinCaml Compiler was developed for this purpose. It is a compiler from a strict, impure, monomorphic, and higher-order functional language—which itself is also called MinCaml and whose syntax is a subset of Objective Caml—to SPARC Assembly. Although it is implemented in only 2000 lines of Objective Caml, its efficiency is comparable to that of OCamlOpt (the optimizing compiler of Objective Caml) or GCC for several applications written in the optimal style of each language implementation.

Curricular Background. MinCaml has been used since year 2001 in a third-year undergraduate course at the Department of Information Science in the University of Tokyo. The course is just called Compiler Experiments (in general, we do not call courses by numbers in Japan), where students are required to implement their own compiler of the MinCaml language from scratch[1], given both high-level and medium-level descriptions in a natural language and mathematical pseudo-code (as in Section 4.3 and 4.4). Although the course schedule varies every year, a typical pattern looks like Table 1.

FDPE'05 September 25, 2005, Tallinn, Estonia.

* The present work was supported by the Information-Technology Promotion Agency, Japan as an Exploratory Software Project.

[1] The source code of MinCaml was not publicly available until March 2005.

Week	Topics
1	Introduction, lexical analysis, parsing
2	K-normalization
3	α-conversion, β-reduction, reduction of nested `let`-expressions
4	Inline expansion, constant folding, elimination of unnecessary definitions
5	Closure conversion, known function call optimization
6	Virtual machine code generation
7	Function calling conventions
8	Register allocation
9	Register spilling
10	Assembly generation
11	Tail call optimization, continuation passing style
12	Type inference, floating-point number operations
13	Garbage collection [no implementation required]
14	Type-based analyses (case study: escape analysis) [no implementation required]

Table 1. Course Schedule

Compiler Experiments is associated with another course named Processor Experiments, where groups of students design and implement their own CPUs by using programmable LSI called FPGA (field programmable gate arrays). Then, they develop compilers for those CPUs, execute ray tracing, and compete on the speed of execution.[2] The goal of these courses is to understand how computer hardware and software work without treating them as black boxes (which leads to misconceptions).

Since students in Tokyo learn only liberal arts for the first year and half, these courses are in fact scheduled in the third *semester* of the information science major curriculum. By then, the students have learned Scheme, ML, and Prolog in addition to C/C++ and SPARC Assembly (during courses on operating systems and computer architecture) as well as Java (in the liberal arts courses). In particular, they have already learned how to write a simple interpreter for a small subset of ML.

Furthermore, they have already taken lectures on compilers of imperative languages (including standard algorithms for lexical analysis, parsing, and register allocation) for one semester. The purpose of our course is to teach efficient compilation of functional languages, rather than giving a *general* compiler course *using* functional languages.

Design Policy. Given these situations, MinCaml was designed with three requirements in mind: (1) It must be understood in every detail by undergraduate students (through 14 hours of lectures and 42 hours of programming). (2) It must be able to execute at least one non-trivial application: ray tracing. (3) It must be as efficient as standard compilers for this application and other typical small programs. Thus, it makes no sense to try to implement the full functionality of ML. To achieve our first goal, MinCaml only supports a *minimal* subset of ML sufficient to meet the other goals. In particular, we have dropped polymorphism and data types as well as modules and garbage collection, though basic implementation techniques for these features are still covered in class.

To make the compiler easier to understand, every design decision is clearly motivated, as described in the following sections.

Paper Overview. Section 2 presents the source language, MinCaml, and Section 3 discusses the design of our compiler. Section 4 elaborates on its details. Section 5 gives the results of our experiments to show the efficiency of the compiler. Section 6 compares our approach with related work and Section 7 concludes with future directions.

[2] This competition started in 1995 and its official record was held by the author's group since they took the course in 1998 until the FPGA was upgraded in 2003.

$M, N, e ::=$	expressions
c	constants
$op(M_1, \ldots, M_n)$	arithmetic operations
`if` M `then` N_1 `else` N_2	conditional branches
`let` $x = M$ `in` N	variable definitions
x	variables
`let rec` $x\ y_1\ \ldots\ y_n = M$ `and` $\ldots$ `in` N	function definitions
$M\ N_1\ \ldots\ N_n$	function applications
$(M_1, \ldots, M_n)$	tuple creations
`let` $(x_1, \ldots, x_n) = M$ `in` N	reading from tuples
`Array.create` $M\ N$	array creations
$M_1.(M_2)$	reading from arrays
$M_1.(M_2) \leftarrow M_3$	writing to arrays
$\rho, \sigma, \tau ::=$	types
π	primitive types
$\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$	function types
$\tau_1 \times \ldots \times \tau_n$	tuple types
τ `array`	array types
α	type variables

Figure 1. Syntax of MinCaml

The implementation and documentations of MinCaml are publicly available at `http://min-caml.sf.net/index-e.html`. Readers are invited (though not required) to consult them when the present paper refers to implementation details.

2. The Language

The source language of MinCaml is a minimal subset of Objective Caml, whose abstract syntax and types are given in Figure 1. This abstract syntax is designed for the following reasons. First of all, as any practical functional language does, we have basic values such as integers, floating-point numbers, booleans, tuples, and functions. Each of them requires at least one constructor (such as constants, function definitions, and tuple creations) and one destructor (such as arithmetic operations, conditional branches, function applications, and reading from tuples).

Conditional branches must be a special form since we do not have more general data types and pattern matching. Tuples are destructed using a simple form of pattern matching, instead of projections like $\#_i(M)$. This avoids the flex record problem: functions such as $f(x) = \#_1(x)$ do not have principal types in the standard type system of ML without record polymorphism [19, 22].

Higher-order functions are supported since functions can be referred to just as variables, and since nested function definitions with free variables are allowed. For simplicity, however, partial function applications are not automatically recognized by the compiler and must be explicitly written by hand, for example like `let rec` $f_3\ y = f\ 3\ y$ `in` f_3 instead of just $f\ 3$ if f is defined to take two arguments. In this respect, our language is more similar to Scheme than to ML.

Since our primary application is ray tracing, we also need arrays for efficient representation of vectors and matrices. Array construction must be a special syntactic form because it is parametric in the element type. (Objective Caml can express this by using a polymorphic library function, but MinCaml cannot.) Once we have arrays, reference cells can also be implemented as arrays with just one ele-

ment. This implementation does not affect efficiency as we anyway have no boundary checks for array accesses.

The types are standard monomorphic types except for n-ary function types, which reflect the lack of partial function applications as mentioned above, and type variables, which will be used for type inference.

These abstract syntax and types are literally implemented as ML data types `Syntax.t` and `Type.t`, except that bound variables in `Syntax.t` are annotated with elements of `Type.t` to keep the type information for later use. Also, for readability, a function definition is represented by a record with three fields `name`, `args` and `body` instead of their triple.

Why Objective Caml? We have chosen Objective Caml as the meta language of MinCaml, as well as using its subset as the object language. This is just because Objective Caml is the only statically typed functional language that is taught well in our curriculum (and because static typing—in particular, exhaustiveness check of pattern matching—helps much when students write a compiler). Otherwise, either Standard ML or even Haskell would be fine as well (though laziness might affect efficiency and require more optimizations such as strictness analysis and linearity analysis).

Why no data types and no pattern matching? As already stated above, we have omitted data types (including lists) and pattern matching. This may sound disappointing, as almost all real programs in ML use them. However, we find that pattern matching is by far more complex than other features when compiling core ML. In addition, it is still possible to write interesting programs without using any data types (or patter matching), as shown in Section 5. Besides, the students are already busy enough in implementing other, more basic features such as higher-order functions. A possible alternative would be to supply a pattern matcher implemented by the instructor, but we rejected this approach because the whole point of our course was to avoid such a "black box" in the compiler.

Why no polymorphism? We have omitted polymorphism as well. Since we do not have data types at all (let alone polymorphic data types), there is much less use for polymorphic functions. In addition, polymorphic functions may affect even the efficiency of monomorphic functions (if implemented by boxing floating-point numbers, for example). On the other hand, it would not be too hard to implement polymorphic functions by code duplication (as in MLton [4]) without sacrificing efficiency. Polymorphic type inference would not be a big problem, either, because it is only a little more complex than the monomorphic version.

In the actual course, we offer brief explanation of the two basic methods above (boxing and code duplication) of implementing polymorphism. Type inference with `let`-polymorphism is taught (and implemented in a simple interpreter) in a previous course on ML programming.

Why no garbage collection? Once we have decided to drop data types, many of the interesting programs can be written in natural ways without allocating too many heap objects. As a result, they can run with no garbage collection at all.

Of course, however, garbage collection is a fundamental feature of most modern programming languages. Thus, we offer a lecture on garbage collection for 4 hours (with no programming tasks), covering basic algorithms such as reference counting, copying GC, and mark-and-sweep GC as well as more advanced topics (without too much detail) including incremental GC, generational GC, and conservative GC.

Why no array boundary checks? While it is easy to implement array boundary checks in MinCaml, we omitted them by default for fairer comparison with C (and OCamlOpt `-unsafe`) as in Section 5. Optimizing away redundant checks would be much harder, as it requires the compiler to solve integer constraints in general.

Module	LoC
Lexical analysis (in OCamlLex)	102
Parsing (in OCamlYacc)	175
Type inference	174
K-normalization	195
α-conversion	52
β-reduction	43
Reduction of nested `let`-expressions	22
Inline expansion	47
Constant folding	50
Elimination of unnecessary definitions	39
Closure conversion	136
Virtual machine code generation	208
13-Bit immediate optimization	42
Register allocation	262
Assembly generation	256

Table 2. Main Modules of The MinCaml Compiler

External functions and arrays. Unlike in ordinary ML, free variables in MinCaml programs are automatically treated as external—either external functions or external arrays—so their declarations can be omitted. This is just for simplicity: since MinCaml is simply typed, their types can easily be inferred from the programs [15].

3. The Compiler

The main modules of The MinCaml Compiler are listed in Table 2, along with their lines of code. This section discusses major choices that we have made in their design and implementation. Further details about the internal structure of MinCaml are described in Section 4.

Lexical analysis and parsing. Although syntax used to be a central issue in conventional compiler courses, we spend as little time as possible on it in our course: that is, we just give students our lexer and parser written in OCamlLex and OCamlYacc. The reason is that lexical analysis and parsing have already been taught in another compiler course (for imperative languages) in our curriculum. Possible alternatives to Lex and Yacc would be parser combinators or packrat parsing [12], but we did not adopt them as syntax is anyway out of scope.

K-normalization. After parsing (and type inference), we use K-normal forms [7] as the central intermediate language in MinCaml. K-normal forms are useful as it makes intermediate computations and their results explicit, simplifying many optimizations including inline expansion and constant folding.

We did not choose A-normal forms [11] because we did not need them: that is, A-normalizing *all* K-normal forms would help little in our compiler. On the contrary, *requiring* the intermediate code to be A-normal forms (i.e., forbidding nested `let`-expressions) complicates inline expansion.

In addition, A-normalization in the strict sense [11] eliminates all conditional branches in non-tail positions by duplicating their evaluation contexts, which can cause code explosion if applied literally. But if we allow conditional branches in non-tail positions, we lose the merit of A-normal forms that e_1 in `let` $x = e_1$ `in` e_2 is always an atomic expression (because e_1 may be a conditional branch `if` e_{11} `then` e_{12} `else` e_{13}, where e_{12} and e_{13} can themselves be `let`-expressions).

We did not choose CPS [5], either, for a similar reason: it does not allow conditional branches in non-tail positions and requires

extra creation of the continuation closure (or inline expansion of it).

Inline expansion. Our algorithm for inlining is rather simple: it expands all calls to functions whose size (number of syntactic nodes in K-normal form) is less than a constant threshold given by the user. Since this does not always terminate when repeated (e.g., consider `let rec` $f\ x = f\ x$ `in` $f\ 3$), the number of iterations is also bounded by a user-given constant. Although this may seem *too* simple, it just works well for our programs—including recursive functions as well as loops implemented by tail recursion (achieving the effect of loop unrolling)—with reasonable increase of code size. By contrast, other inlining algorithms (see [20] for example) are much more complex than ours. Note that our inlining algorithm is implemented in only 47 lines, including the "size" function.

Closure conversion. MinCaml supports higher-order functions by closure conversion. It optimizes calls to known functions with no free variables. Again, this known function optimization is simple and effective enough for our purpose. Indeed, it optimizes *all* function calls in the critical parts of our benchmark applications.

In addition to K-normal form expressions, we introduce three special constructs `make_closure`, `apply_closure`, and `apply_direct` (which means known function calls) in the intermediate language after closure conversion. This enables us to keep the simple types without more advanced mechanisms such as existential types [17].

Register Allocation. The most sophisticated process in MinCaml (or perhaps in any modern compiler) is register allocation. Although many standard algorithms exist for imperative languages (e.g., [8, 21]), we find them unnecessarily complicated for MinCaml because its variables are never destructively updated, obviating the standard notion of "def-use chains" [18]. In addition, it is always better to spill a variable *as early as possible*, if at all. Thus, we have adopted a simpler greedy algorithm with backtracking (for early spilling) and look-ahead (for register targeting [5]). We need not worry about coverage, because standard algorithms have already been taught in the other compiler course (for imperative languages).

4. Inside The MinCaml Compiler

The architecture of MinCaml adheres to the following principle. A compiler, by definition, is a program that transforms high-level programs to lower-level code. For example, the ML function

```
let rec gcd m n =
  if m = 0 then n else
  if m <= n then gcd m (n - m) else
  gcd n (m - n)
```

is compiled into the SPARC Assembly code

```
gcd.7:          cmp     %i2, 0
                bne     be_else.18
                nop
                mov     %i3, %i2
                retl
                nop
be_else.18:     cmp     %i2, %i3
                bg      ble_else.19
                nop
                sub     %i3, %i2, %i3
                b       gcd.7
                nop
ble_else.19:    sub     %i2, %i3, %o5
                mov     %i3, %i2
                mov     %o5, %i3
                b       gcd.7
                nop
```

which, at a first glance, looks totally different. The MinCaml Compiler, like many other modern compiles, bridges this huge gap by defining appropriate intermediate languages and applying simple program transformations one by one. The major five gaps between MinCaml and SPARC Assembly are:

1. Types. MinCaml has a type discipline; assembly does not.
2. Nested expressions. MinCaml code is tree-structured with compound instructions; assembly code is a linear sequence of atomic instructions.
3. Nested function definitions. In MinCaml, we can define a function inside another function like
   ```
   let rec make_adder x =
     let rec adder y = x + y in
     adder in
   (make_adder 3) 7
   ```
   but assembly has only top-level "labels."
4. MinCaml has data structures such as tuples and arrays, while assembly does not.
5. In MinCaml, we can use as many variables as we want, but only a limited number of registers are available in assembly (and therefore they sometimes must be spilled to memory).

To bridge these gaps, MinCaml applies translations such as type inference, K-normalization, closure conversion, virtual machine code generation, register allocation (in this order). In what follows, we will explain the compilation processes of MinCaml including these translations and other optimizations.

4.1 Lexical Analysis and Parsing (102 + 175 Lines)

```
type Id.t (* variable names *)
type 'a M.t (* finite maps from Id.t to 'a *)
type S.t (* finite sets of Id.t *)

type Type.t (* types *)
type Syntax.t (* expressions *)
```

The lexical analysis and parsing of MinCaml are implemented with standard tools, OCamlLex and OCamlYacc. As usual, they translate a string of characters to a sequence of tokens and then to an abstract syntax tree, which is necessary for any complex program manipulation. There is nothing special to be noted: indeed, in our course, the files `lexer.mll` and `parser.mly` are just given to students in order to avoid the overhead of learning the tools themselves. The only non-trivial point—if any—is function arguments: to parse `x-3` as integer subtraction rather than function application `x(-3)`, the parser distinguishes "simple expressions" (expressions that can be function arguments with no extra parentheses) from other expressions like `-3`, just as Objective Caml does.

4.2 Type Inference (174 Lines)

```
val Typing.f : Syntax.t -> Syntax.t
val Typing.extenv : Type.t M.t ref

(* "private" function to
   destructively substitute type variables *)
val Typing.g : Type.t M.t -> Syntax.t -> Type.t
```

Since MinCaml is an implicitly typed language but our compiler relies on the code being annotated with types, we first carry out a monomorphic version of Hindley-Milner type inference. It is also implemented in a standard way, representing type variables as `Type.t option ref` and substituting `None` with `Some` τ during unification. The only deviation is our treatment of external variables: when the type checker sees a free variable not found in the

type environment, this variable is assumed as "external" and added to a special type environment `Typing.extenv` for external variables. Thus, they need not be declared in advance: their types are inferred just as ordinary variables. This principal typing functionality is peculiar to MinCaml and not applicable to full ML with `let`-polymorphism [15]. After the type inference, instantiated type variables (references to `Some` τ) are replaced with their contents (type τ). Any uninstantiated type variables is defaulted (arbitrarily) to `int`.

4.3 K-Normalization (195 Lines)

```
type KNormal.t (* K-normalized expressions *)

val KNormal.f : Syntax.t -> KNormal.t
val KNormal.fv : KNormal.t -> S.t
```

We stated that compilation is about bridging the gaps between high-level programs and low-level code. One of the gaps is nested expressions: usually, a sequence of several instructions (like `add`, `add`, `add`, and `sub`) are needed to computer the value of a compound expression (like `a+b+c-d`).

This gap is bridged by a translation called *K-normalization*, which defines every intermediate result of computation as a variable. For example, the previous expression can be translated to:

```
let tmp1 = a + b in
let tmp2 = tmp1 + c in
tmp2 - d
```

In general, the process of K-normalization can be described as follows. First, we define the abstract syntax of the intermediate language, K-normal forms, which is implemented as ML data type `KNormal.t`.

$$
\begin{array}{rcl}
M, N, e & ::= & c \\
& | & op(x_1, \ldots, x_n) \\
& | & \texttt{if } x = y \texttt{ then } M \texttt{ else } N \\
& | & \texttt{if } x \leq y \texttt{ then } M \texttt{ else } N \\
& | & \texttt{let } x = M \texttt{ in } N \\
& | & x \\
& | & \texttt{let rec } x\ y_1\ \ldots\ y_n = M \texttt{ and } \ldots \texttt{ in } N \\
& | & x\ y_1\ \ldots\ y_n \\
& \vdots &
\end{array}
$$

The main point is that every target of basic operations—such as arithmetic operations, function applications, tuple creations, and reading from tuples—is now a variable, not a nested expression, because nested expressions are converted into sequences of `let`-expressions as in the example above. This conversion can be described by the following function $\mathcal{K}$. For every equation in this definition, all variables not appearing on the left-hand side are freshly generated. Although this function is straightforward, it is presented here for the purpose of showing how the mathematical pseudo-code given to students (and required to be implemented in Objective Caml, as mentioned in Section 1) looks like in general.

$$
\begin{array}{l}
\mathcal{K}(c) = c \\
\mathcal{K}(op(M_1, \ldots, M_n)) = \\
\quad \texttt{let } x_1 = \mathcal{K}(M_1) \texttt{ in} \\
\quad\quad \ldots \\
\quad \texttt{let } x_n = \mathcal{K}(M_n) \texttt{ in} \\
\quad op(x_1, \ldots, x_n) \\
\mathcal{K}(\texttt{if } M_1 = M_2 \texttt{ then } N_1 \texttt{ else } N_2) = \\
\quad \texttt{let } x = \mathcal{K}(M_1) \texttt{ in let } y = \mathcal{K}(M_2) \texttt{ in} \\
\quad \texttt{if } x = y \texttt{ then } \mathcal{K}(N_1) \texttt{ else } \mathcal{K}(N_2) \\
\mathcal{K}(\texttt{if } M_1 \neq M_2 \texttt{ then } N_1 \texttt{ else } N_2) = \\
\quad \mathcal{K}(\texttt{if } M_1 = M_2 \texttt{ then } N_2 \texttt{ else } N_1) \\
\mathcal{K}(\texttt{if } M_1 \leq M_2 \texttt{ then } N_1 \texttt{ else } N_2) = \\
\quad \texttt{let } x = \mathcal{K}(M_1) \texttt{ in let } y = \mathcal{K}(M_2) \texttt{ in} \\
\quad \texttt{if } x \leq y \texttt{ then } \mathcal{K}(N_1) \texttt{ else } \mathcal{K}(N_2) \\
\mathcal{K}(\texttt{if } M_1 \geq M_2 \texttt{ then } N_1 \texttt{ else } N_2 = \\
\quad \mathcal{K}(\texttt{if } M_2 \leq M_1 \texttt{ then } N_1 \texttt{ else } N_2) \\
\mathcal{K}(\texttt{if } M_1 > M_2 \texttt{ then } N_1 \texttt{ else } N_2) = \\
\quad \mathcal{K}(\texttt{if } M_1 \leq M_2 \texttt{ then } N_2 \texttt{ else } N_1) \\
\mathcal{K}(\texttt{if } M_1 < M_2 \texttt{ then } N_1 \texttt{ else } N_2) = \\
\quad \mathcal{K}(\texttt{if } M_2 \leq M_1 \texttt{ then } N_2 \texttt{ else } N_1) \\
\mathcal{K}(\texttt{if } M \texttt{ then } N_1 \texttt{ else } N_2) = \\
\quad \mathcal{K}(\texttt{if } M = \texttt{false then } N_2 \texttt{ else } N_1) \\
\quad\quad \text{(if } M \text{ is not a comparison)} \\
\mathcal{K}(\texttt{let } x = M \texttt{ in } N) = \\
\quad \texttt{let } x = \mathcal{K}(M) \texttt{ in } \mathcal{K}(N) \\
\mathcal{K}(x) = x \\
\mathcal{K}(\texttt{let rec } f\ x_1\ \ldots\ x_n = M \texttt{ and } \ldots \texttt{ in } N) = \\
\quad \texttt{let rec } f\ x_1\ \ldots\ x_n = \mathcal{K}(M) \texttt{ and } \ldots \texttt{ in } \mathcal{K}(N) \\
\mathcal{K}(M\ N_1\ \ldots\ N_n) = \\
\quad \texttt{let } x = \mathcal{K}(M) \texttt{ in} \\
\quad \texttt{let } y_1 = \mathcal{K}(N_1) \texttt{ in} \\
\quad\quad \ldots \\
\quad \texttt{let } y_n = \mathcal{K}(N_n) \texttt{ in} \\
\quad x\ y_1\ \ldots\ y_n \\
\quad \vdots
\end{array}
$$

As apparent from the definitions above, we also translate conditional branches into two special forms combining comparisons and branches. This translation bridges another gap between MinCaml and assembly where branch instructions must follow compare instructions. Although unrelated to K-normalization, it is implemented here to avoid introducing yet another intermediate language.

In addition, as an optional optimization, our actual implementation avoids inserting a `let`-expression if the term is already a variable. This small improvement is implemented in auxiliary function `insert_let`.

```
let insert_let (e, t) k =
  match e with
  | Var(x) -> k x
  | _ ->
      let x = Id.gentmp t in
      let e', t' = k x in
      Let((x, t), e, e'), t'
```

It takes an expression `e` (with its type `t`) and a continuation `k`, generates a variable `x` if `e` is not already a variable, applies `k` to `x` to obtain the body `e'` (with its type `t'`), inserts a `let`-expression to bind `x` to `e`, and returns it (with `t'`). The types are passed around just because they are necessary for type annotations of bound variables, and are not essential to K-normalization itself.

This trick not only improves the result of K-normalization but also simplifies its implementation. (This would be yet another evidence that continuations are relevant to `let`-insertion [16] in general.) For example, the case for integer addition can be coded as

```
(* in pattern matching over Syntax.t *)
  | Syntax.Add(e1, e2) ->
      insert_let (g env e1)
```

```
          (fun x -> insert_let (g env e2)
              (fun y -> Add(x, y), Type.Int))
```

and reading from arrays as:

```
  | Syntax.Get(e1, e2) ->
      (match g env e1 with
      | (_, Type.Array(t)) as g_e1 ->
          insert_let g_e1
            (fun x -> insert_let (g env e2)
                (fun y -> Get(x, y), t))
      | _ -> assert false)
```

The false assertion in the last line could be removed if K-normalization were fused with type inference, but we rejected this alternative in favor of modularity.

4.4 α-Conversion (52 Lines)

```
val Alpha.f : KNormal.t -> KNormal.t

(* also public for reuse by Inline.g *)
val Alpha.g : Id.t M.t -> KNormal.t -> KNormal.t
```

Following K-normalization, MinCaml renames all bound variables of a program to fresh names, which is necessary for the correctness of transformations such as inlining. It can be specified by the following function α, where variables not appearing on the left-hand side are freshly generated and $\varepsilon(x)$ is defined to be x when x is not in the domain of ε.

$$
\begin{array}{l}
\alpha_\varepsilon(c) \;=\; c \\
\alpha_\varepsilon(op(x_1, \ldots, x_n)) \;= \\
\quad op(\varepsilon(x_1), \ldots, \varepsilon(x_n)) \\
\alpha_\varepsilon(\texttt{if}\ x = y\ \texttt{then}\ M_1\ \texttt{else}\ M_2) \;= \\
\quad \texttt{if}\ \varepsilon(x) = \varepsilon(y)\ \texttt{then}\ \alpha_\varepsilon(M_1)\ \texttt{else}\ \alpha_\varepsilon(M_2) \\
\alpha_\varepsilon(\texttt{if}\ x \le y\ \texttt{then}\ M_1\ \texttt{else}\ M_2) \;= \\
\quad \texttt{if}\ \varepsilon(x) \le \varepsilon(y)\ \texttt{then}\ \alpha_\varepsilon(M_1)\ \texttt{else}\ \alpha_\varepsilon(M_2) \\
\alpha_\varepsilon(\texttt{let}\ x = M\ \texttt{in}\ N) \;= \\
\quad \texttt{let}\ x' = \alpha_\varepsilon(M)\ \texttt{in}\ \alpha_{\varepsilon, x \mapsto x'}(N) \\
\alpha_\varepsilon(x) \;=\; \varepsilon(x) \\
\alpha_\varepsilon(\texttt{let rec}\ f\ x_1\ \ldots\ x_m = M_1 \\
\quad \texttt{and}\ g\ y_1\ \ldots\ y_n = M_2 \\
\quad\ \ldots \\
\quad \texttt{in}\ N) \;= \\
\quad \texttt{let rec}\ f'\ x'_1\ \ldots\ x'_m = \alpha_{\varepsilon, \sigma, x_1 \mapsto x'_1, \ldots, x_m \mapsto x'_m}(M_1) \\
\quad \texttt{and}\ g'\ y'_1\ \ldots\ y'_n = \alpha_{\varepsilon, \sigma, y_1 \mapsto y'_1, \ldots, y_n \mapsto y'_n}(M_2) \\
\quad\ \ldots \\
\quad \texttt{in}\ \alpha_{\varepsilon, \sigma}(N) \qquad (\text{where } \sigma = f \mapsto f', g \mapsto g', \ldots) \\
\alpha_\varepsilon(x\ y_1\ \ldots\ y_n) \;= \\
\quad \varepsilon(x)\ \varepsilon(y_1)\ \ldots\ \varepsilon(y_n) \\
\quad \vdots
\end{array}
$$

It is implemented by a recursive function `Alpha.g`, which takes a (sub-)expression with a mapping ε from old names to new names and returns an α-converted expression. If a variable is not found in the mapping, it is considered external and left unchanged. This behavior is implemented by auxiliary function `Alpha.find`, which is used everywhere in `Alpha.g` since variables are ubiquitous in K-normal forms.

Naturally, as long as we are just α-converting a whole program, we only need to export the interface function `Alpha.f` which calls `Alpha.g` with an empty mapping. Nevertheless, the internal function `Alpha.g` is also exported because it is useful for inlining as explained later.

4.5 β-Reduction (43 Lines)

```
val Beta.f : KNormal.t -> KNormal.t

(* private *)
val Beta.g : Id.t M.t -> KNormal.t -> KNormal.t
```

It is often useful—both for clarify and for efficiency—to reduce expressions such as `let` $x = y$ `in` $x + y$ to $y + y$, expanding the aliasing of variables. We call the expansion *β-reduction* of K-normal forms. (Of course, this name originates from β-reduction in λ-calculus, of which ours is a special case if `let`-expressions are represented by applications of λ-abstractions, like $(\lambda x.\, x + y)y$ for example.) It is not always necessary in ordinary programs, but is sometimes effective after other transformations.

β-reduction in MinCaml is implemented by function `Beta.g`, which takes an expression with a mapping from variables to equal variables and returns the β-reduced expression. Specifically, when we see an expression of the form `let` $x = e_1$ `in` e_2, we first β-reduce e_1. If the result is a variable y, we add the mapping from x to y and then continue by β-reducing e_2. Again, since variables appear everywhere in K-normal forms, auxiliary function `Beta.find` is defined and used for brevity (as in α-conversion) to substitute variables if and only if they are found in the mapping.

4.6 Reduction of Nested `let`-Expressions (22 Lines)

```
val Assoc.f : KNormal.t -> KNormal.t
```

Next, in order to expose the values of nested `let`-expressions for subsequent transformations, we flatten nested `let`-expressions such as `let` $x =$ (`let` $y = e_1$ `in` e_2) `in` $e3$ to `let` $y = e_1$ `in let` $x = e_2$ `in` $e3$. This "reduction" by itself does not affect the efficiency of programs compiled by MinCaml, but it helps other optimizations (e.g., constant folding of e_2) as well as simplifying the intermediate code.

This transformation is implemented by function `Assoc.f`. Upon seeing an expression of the form `let` $x = e_1$ `in` e_2, we first reduce e_1 to e'_1 and e_2 to e'_2 by recursion. Then, if e'_1 is of the form `let ... in` e, we return the expression `let ... in let` $x = e$ `in` e'_2. This verbal explanation may sound tricky but the actual implementation is simple:

```
(* in pattern matching over KNormal.t *)
  | Let(xt, e1, e2) ->
      let rec insert = function
        | Let(yt, e3, e4) ->
            Let(yt, e3, insert e4)
        | LetRec(fundefs, e) ->
            LetRec(fundefs, insert e)
        | LetTuple(yts, z, e) ->
            LetTuple(yts, z, insert e)
        | e -> Let(xt, e, f e2) in
      insert (f e1)
```

Indeed, `assoc.ml` consists of only 22 lines as noted above.

4.7 Inline Expansion (47 Lines)

```
val Inline.threshold : int ref
val Inline.f : KNormal.t -> KNormal.t

(* private *)
val Inline.size : KNormal.t -> int
val Inline.g : ((Id.t * Type.t) list * KNormal.t) M.t ->
               KNormal.t -> KNormal.t
```

The next optimization is the most effective one: inline expansion. It replaces calls to small functions with their bodies. MinCaml implements it in module `Inline` as follows.

Upon seeing a function definition `let rec` $f\ x_1\ \ldots\ x_n =$ e `in` $\ldots$, we compute the size of e by `Inline.size`. If this size is less than the value of integer reference `Inline.threshold` set

by the user, we add the mapping from function name f to the pair of formal arguments $x_1, \ldots, x_n$ and body e. Then, upon seeing a function call $f\ y_1\ \ldots\ y_n$, we look up the formal arguments $x_1, \ldots, x_n$ of f and its body e, and return e with $x_1, \ldots, x_n$ substituted by $y_1, \ldots, y_n$.

However, since inlined expressions are copies of function bodies, their variables may be duplicated and therefore must be α-converted again. Fortunately, the previous process of substituting formal arguments with actual arguments can be carried out by `Alpha.g` together with α-conversion, just by using the correspondence from $x_1, \ldots, x_n$ to $y_1, \ldots, y_n$ (instead of an empty mapping) as the initial mapping. Thus, the inline expansion can be implemented just as

```
(* pattern matching over KNormal.t *)
  | App(x, ys) when M.mem x env ->
      let (zs, e) = M.find x env in
      let env' =
        List.fold_left2
          (fun env' (z, t) y -> M.add z y env')
          M.empty zs ys in
      Alpha.g env' e
```

where `M` is a module for mappings.

4.8 Constant Folding (50 Lines)

```
val ConstFold.f : KNormal.t -> KNormal.t

(* private *)
val ConstFold.g : KNormal.t M.t -> KNormal.t -> KNormal.t
```

Once functions are inlined, many operations have arguments whose values are already known, as $x+y$ in `let` $x = 3$ `in let` $y = 7$ `in` $x+y$. Constant folding carries out such operations at compile-time and replaces them with constants like 10. MinCaml implements it in function `ConstFold.g`. It takes an expression with a mapping from variables to their definitions, and returns the expression after constant folding. For example, given an integer addition $x + y$, it examines whether the definitions of x and y are integer constants. If so, it calculates the result and returns it right away. Conversely, given a variable definition `let` $x = e$ `in` $\ldots$, it adds the mapping from x to e. This is applied to floating-point numbers and tuples as well.

4.9 Elimination of Unnecessary Definitions (39 Lines)

```
val Elim.f : KNormal.t -> KNormal.t

(* private *)
val Elim.effect : KNormal.t -> bool
```

After constant folding, we often find unused variable definitions (and unused function definitions) as in `let` $x = 3$ `in let` $y = 7$ `in` 10. MinCaml removes them in module `Elim`.

In general, if e_1 has no side effect and x does not appear free in e_2, we can replace `let` $x = e_1$ `in` e_2 just with e_2. The presence of side effects is checked by `Elim.effect` and the appearance of variables are examined by `KNormal.fv`. Since it is undecidable whether an expression has a real side effect, we treat any write to an array and any call to a function as side-effecting.

Mutually recursive functions defined by a single `let rec` are eliminated only when none of the functions is used in the continuation. If any of the functions are used after the definition, then all of them are kept.

4.10 Closure Conversion (136 Lines)

```
type Id.l (* label names *)
type Closure.t (* closure-converted expressions *)
type Closure.fundef =
  { name : Id.l * Type.t;
    args : (Id.t * Type.t) list;
    formal_fv : (Id.t * Type.t) list;
    body : Closure.t }
type Closure.prog =
    Prog of Closure.fundef list * Closure.t

val Closure.f : KNormal.t -> Closure.prog
val Closure.fv : Closure.t -> S.t

(* private *)
val Closure.toplevel : Closure.fundef list ref
val Closure.g : Type.t M.t (* typenv for fv *) ->
                S.t (* known functions *) ->
                KNormal.t -> Closure.t
```

Another gap still remaining between MinCaml and assembly is nested function definitions, which are flattened by closure conversion. It is the second most complicated process in our compiler. (The first is register allocation, which is described later.) What follows is how we explain closure conversion to students.

The flattening of nested function definitions includes easy cases and hard cases. For example,

```
let rec quad x =
  let rec dbl y = y + y in
  dbl (dbl x) in
quad 123
```

can be flattened like

```
let rec dbl y = y + y ;;
let rec quad x = dbl (dbl x) ;;
quad 123
```

just by moving the function definition. However, a similar manipulation would convert

```
let rec make_adder x =
  let rec adder y = x + y in
  adder in
(make_adder 3) 7
```

into

```
let rec adder y = x + y ;;
let rec make_adder x = adder ;;
(make_adder 3) 7
```

which makes no sense at all. This is because the function `dbl` has no free variable while `adder` has a free variable `x`.

Thus, in order to flatten function definitions with free variables, we have to treat not only the bodies of functions such as `adder`, but also the values of their free variables such as `x` together. In ML-like pseudo code, this treatment can be described as:

```
let rec adder x y = x + y ;;
let rec make_adder x = (adder, x) ;;
let (f, fv) = make_adder 3 in
f fv 7
```

First, function `adder` takes the value of its free variable `x` as an argument. Then, when the function is returned as a value, its body is paired with the value of its free variable. This pair is called a *function closure*. In general, when a function is called, its body and the values of its free variables are extracted from the closure and supplied as arguments.

The simple-minded approach of generating a closure for every function is too inefficient. Closure conversion gets more interesting when we try to separate the functions that require closures from those that can be called in more conventional ways. Thus, the closure conversion routine `Closure.g` of MinCaml takes the set `known` of functions that are statically known to have no free variables (and therefore can be called directly), and converts a given expression by using this information.

The results of closure conversion are represented in data type `Closure.t` that represents the following abstract syntax:

$P ::=$	
$\{D_1, \ldots, D_n\}, M$	whole program
$D ::=$	
$\ell(y_1, \ldots, y_m)(z_1, \ldots, z_n) = N$	top-level function definition
$M, N, e ::=$	
c	constants
$op(x_1, \ldots, x_n)$	arithmetic operations
`if` $x = y$ `then` M `else` N	conditional branches
`if` $x \leq y$ `then` M `else` N	conditional branches
`let` $x = M$ `in` N	variable definitions
x	variables
`make_closure` $x = (\ell, (z_1, \ldots, z_n))$ `and` $\ldots$ `in` M	closure creation
`apply_closure`$(x, y_1, \ldots, y_n)$	closure-based function call
`apply_direct`$(\ell, y_1, \ldots, y_n)$	direct function call
$\vdots$	

It is similar to `KNormal.t`, but includes closure creation `make_closure` and top-level functions $D_1, \ldots, D_n$ instead of nested function definitions. In addition, instead of general function calls, it has closure-based function calls `apply_closure` and direct function calls `apply_direct` that do not use closures. Furthermore, in the processes that follow, we distinguish the type of top-level function names (labels) from the type of ordinary variable names in order to avoid confusions. Note that `apply_closure` uses variables while `apply_direct` uses labels. This is because closures are bound to variables (by `make_closure`) while top-level functions are called through labels.

Upon seeing a general function call $x\ y_1\ \ldots\ y_n$, `Closure.g` checks if the function x belongs to the set `known`. If so, it returns `apply_direct`. If not, it returns `apply_closure`.

```
| KNormal.App(x, ys) when S.mem x known ->
    AppDir(Id.L(x), ys)
| KNormal.App(f, xs) ->
    AppCls(f, xs)
```

Here, `AppDir` and `AppCls` are constructors in the `Closure` module that correspond to `apply_direct` and `apply_closure`, `S` is a module for sets, and `Id.L` is the constructor for labels.

Function definitions `let rec` $x\ y_1\ \ldots\ y_n = e_1$ `in` e_2 are processed as follows. First, we assume that the function x has no free variable, add it to `known`, and convert its body e_1. Then, if x indeed has no free variable, we continue the process and convert e_2. Otherwise, we rewind the values of `known` and `toplevel` (a reference cell holding top-level functions), and redo the conversion of e_1. (This may take exponential time with respect to the depth of nested function definitions, which is small in practice.) Finally, if x never appears as a proper variable (rather than a top-level label) in e_2, we omit the closure creation `make_closure` for function x.

This last optimization needs some elaboration. Even if x has no free variable, it may still need a representation as a closure, provided that it is returned as a value (consider, for example, `let rec` $x\ y = \ldots$ `in` x). This is because a user who receives x as a value does not know in general if it has a free variable or not, and therefore must anyway use `apply_closure` to call the function through its closure. In this case, we do not eliminate `make_closure` since x appears as a variable in e_2. However, if x is just called as a function, for example like `let rec` $x\ y = \ldots$ `in` $x\ 123$, then we eliminate the closure creation for x because it appears only as a label (not a variable) in `apply_direct`.

The closure conversion of mutually recursive functions is a little more complicated. In general, mutually recursive functions can share closures [5], but MinCaml does not implement this sharing. This simplifies the virtual machine code generation as discussed later. The drawback is that mutually recursive calls to functions with free variables get slower. However, we do *not* lose the efficiency of mutually recursive calls to functions with *no* free variables, because they are anyway converted to `apply_direct`.

4.11 Virtual Machine Code Generation (208 Lines)

```
type SparcAsm.t (* instruction sequences *)
type SparcAsm.exp (* atomic expressions *)
type SparcAsm.fundef =
  { name : Id.l;
    args : Id.t list; (* int arguments *)
    fargs : Id.t list; (* float arguments *)
    body : SparcAsm.t;
    ret : Type.t (* return type *)}
type SparcAsm.prog =
    Prog of (Id.l * float) list * (* float table *)
            SparcAsm.fundef list *
            SparcAsm.t

val SparcAsm.fv : SparcAsm.t -> Id.t list (* use order *)
val Virtual.f : Closure.prog -> SparcAsm.prog

(* private *)
val Virtual.data : (Id.l * float) list ref (* float table *)
val Virtual.h : Closure.fundef -> SparcAsm.fundef
val Virtual.g : Type.t M.t -> Closure.t -> SparcAsm.t
```

After closure conversion, we generate SPARC Assembly. Since it is too hard to output real assembly, we first generate *virtual* machine code similar to SPARC Assembly. Its main "virtual" aspects are:

- Infinite number of variables (instead of finite number of registers)
- `if-then-else` expressions and function calls (instead of comparisons, branches, and jumps)

This virtual assembly is defined in module `SparcAsm`. The ML data type `SparcAsm.exp` almost corresponds to each instruction of SPARC (except `If` and `Call`). Instruction sequences `SparcAsm.t` are either `Ans`, which returns a value at the end of a function, or a variable definition `Let`. The other instructions `Forget`, `Save`, and `Restore` will be explained later.

```
(* C(i) represents 13-bit immediates of SPARC *)
type id_or_imm = V of Id.t | C of int

type t =
  | Ans of exp
  | Let of (Id.t * Type.t) * exp * t
  | Forget of Id.t * t
and exp = (* excerpt *)
  | Set of int
  | SetL of Id.l
  | Add of Id.t * id_or_imm
  | Ld of Id.t * id_or_imm
  | St of Id.t * Id.t * id_or_imm
  | FAddD of Id.t * Id.t
  | LdDF of Id.t * id_or_imm
  | StDF of Id.t * Id.t * id_or_imm
  | IfEq of Id.t * id_or_imm * t * t
  | IfFEq of Id.t * Id.t * t * t
  | CallCls of Id.t * Id.t list * Id.t list
  | CallDir of Id.l * Id.t list * Id.t list
  | Save of Id.t * Id.t
  | Restore of Id.t
```

`Virtual.f`, `Virtual.h`, and `Virtual.g` are the three functions that translate closure-converted programs to virtual machine code. `Virtual.f` translates the whole program (the list of top-level functions and the expression of a main routine), `Virtual.h` translates

each top-level function, and `Virtual.g` translates an expression. The point of these translations is to make explicit the memory accesses for creating, reading from, and writing to closures, tuples, and arrays. Data structures such as closures, tuples, and arrays are allocated in the heap, whose address is remembered in special register `SparcAsm.reg_hp`.

For example, to read from an array, we shift its offset according to the size of the element to be loaded.

```
| Closure.Get(x, y) ->
    let offset = Id.genid "o" in
    (match M.find x env with
    | Type.Array(Type.Unit) -> Ans(Nop)
    | Type.Array(Type.Float) ->
        Let((offset, Type.Int), SLL(y, C(3)),
            Ans(LdDF(x, V(offset))))
    | Type.Array(_) ->
        Let((offset, Type.Int), SLL(y, C(2)),
            Ans(Ld(x, V(offset))))
    | _ -> assert false)
```

In tuple creation `Closure.Tuple`, each element is stored with floating-point numbers aligned (in 8 bytes), and the starting address is used as the tuple's value. Closure creation `Closure.MakeCls` stores the address (label) of the function's body with the values of its free variables—also taking care of alignment—and uses the starting address as the closure's value. As mentioned in the previous section, this is easy because we generate separate closures with no sharing at all even for mutually recursive functions. Accordingly, at the beginning of each top-level function, we load the values of free variables from the closure, where every closure-based function application (`AppCls`) is assumed to set the closure's address to register `SparcAsm.reg_cl`.

In addition, since SPARC Assembly does not support floating-point immediates, we need to create a constant table in memory. For this purpose, `Virtual.g` records floating-point constants to global variable `Virtual.data`.

4.12 13-Bit Immediate Optimization (42 Lines)

```
val Simm13.f : SparcAsm.prog -> SparcAsm.prog
```

In SPARC Assembly, most integer operations can take an immediate within 13 bits (no less than -4096 and less than 4096) as the second operand. An optimization using this feature is implemented in module `Simm13`. It is almost the same as constant folding and elimination of unnecessary definitions, except that the object language is virtual assembly and the constants are limited to 13-bit integers.

4.13 Register Allocation (262 Lines)

[Update on September 17, 2008: The register allocator now uses a simpler algorithm. It omits the backtracking (ToSpill and NoSpill) explained below.]

```
val RegAlloc.f : SparcAsm.prog -> SparcAsm.prog

(* private *)
type g_result =
   NoSpill of SparcAsm.t * Id.t M.t
 | ToSpill of SparcAsm.t * Id.t list
val RegAlloc.h : SparcAsm.fundef -> SparcAsm.fundef
val RegAlloc.g : Id.t * Type.t (* dest *) ->
                 SparcAsm.t (* cont *) ->
                 Id.t M.t (* regenv *) ->
                 SparcAsm.t -> g_result
val RegAlloc.g' : Id.t * Type.t (* dest *) ->
                  SparcAsm.t (* cont *) ->
                  Id.t M.t (* regenv *) ->
                  SparcAsm.exp -> g_result
```

The most complex process in The MinCaml Compiler is register allocation, which implements infinite number of variables by finite number of registers. As discussed in Section 3, our register allocator adopts a greedy algorithm with backtracking for early spilling and look-ahead for register targeting.

4.13.1 Basics

First of all, as a function calling convention, we will assign arguments from the first register toward the last register. (Our compiler does not support too many arguments that do not fit in registers. They must be handled by programmers, for example by using tuples.) We set return values to the first register. These are processed in `RegAlloc.h`, which allocates registers in each top-level function.

After that, we allocate registers in function bodies and the main routine. `RegAlloc.g` takes an instruction sequence with a mapping `regenv` from variables to registers that represents the current register assignment, and returns the instruction sequence after register allocation. The basic policy of register allocation is to avoid registers already assigned to live variables. The set of live variables are calculated by `SparcAsm.fv`.

However, when allocating registers in the instruction sequence e_1 of `let` $x = e_1$ `in` e_2, not only e_1 but also its "continuation" e_2 must be taken into account for the calculation of live variables. For this reason, `RegAlloc.g` and `RegAlloc.g'`, which allocates registers in individual instructions, also take the continuation instruction sequence `cont` and use it in the calculation of live variables.

4.13.2 Spilling

We sometimes cannot allocate any register that is not live, since the number of variables is infinite while that of registers is not. In this case, we have to save the value of some register to memory. This process is called register spilling. Unlike in imperative languages, the value of a variable in functional languages does not change after its definition. Therefore, it is better to save the value of a variable *as early as possible*, if at all, in order to make the room.

Whenever a variable x needs to be saved, `RegAlloc.g` returns a value `ToSpill`, and returns to the definition of x to insert a virtual instruction `Save`. In addition, since we want to remove x from the set of live variables at the point where x is spilled, we insert another virtual instruction `Forget` to exclude x from the set of free variables. For this purpose, value `ToSpill` carries not only the list `xs` of spilled variables, but also the instruction sequence e in which `Forget` has been inserted. After saving x, we redo the register allocation against e.

Saving is necessary not only when registers are spilled, but also when functions are called. MinCaml adopts the caller-save convention, so every function call is assumed to destroy the values of all registers. Therefore, we need to save the values of all registers that are live at that point, as implemented in an auxiliary function `RegAlloc.g'_call`. This is why `ToSpill` holds the *list* of spilled variables.

When saving is unnecessary, we return the register-allocated instruction sequence e' (with the new `regenv`) in another value `NoSpill`.

To put it altogether, the data type for the returned values of these functions is defined as follows:

```
type g_result =
    NoSpill of
        SparcAsm.t (* instruction sequence
                      with registers allocated *)
      * Id.t M.t   (* new regenv *)
  | ToSpill of
```

```
    SparcAsm.t (* instruction sequence
                  with Forget inserted *)
  * Id.t list  (* spilled variables *)
```

4.13.3 Unspilling

A spilled variable will be used sooner or later, in which case `RegAlloc.g'` (the function that allocates registers in individual instructions) raises an exception as it cannot find the variable in `regenv`. This exception is handled in an auxiliary function `RegAlloc.g'_and_unspill`, where virtual instruction `Restore` is inserted to restore the value of the variable from memory to a register.

However, this insertion of `Restore` pseudo-instructions breaks a fundamental property of our virtual assembly that every variable is assigned just one register. In particular, it leads to a discrepancy when two flows of a program join after conditional branches. For example, in the `then`-clause of expression $(\texttt{if}\ f()\ \texttt{then}\ x - y\ \texttt{else}\ y - x) + x + y$, variable x may be restored into register r_0 and y may be restored into r_1, while they may be restored in the other order in the `else`-clause. (A similar discrepancy also arises concerning whether a variable is spilled or not.)

In imperative languages, such "discrepancies" are so common that a more sophisticated notion of *def-use chains* is introduced and used as the unit of register allocation (instead of individual variables). In MinCaml, fortunately, those cases are less common and can be treated in a simpler manner: whenever a variable is not in the same register after conditional branches, it is just assumed as spilled (and needs to be restored before being used again), as implemented in an auxiliary function `RegAlloc.g'_if`.

4.13.4 Targeting

When allocating registers, we not only avoid live registers, but also try to reduce unnecessary moves in the future. This is called register targeting [5], itself an instance of register coalescing [18]. For example, if a variable being defined will be the second argument of a function call, we try to allocate it on the second register. For another example, we try to allocate a variable on the first register if it will be returned as the result of a function. These are implemented in `RegAlloc.target`. For this purpose, `RegAlloc.g` and `RegAlloc.g'` also takes register `dest` as an argument, where the result of computation will be stored.

4.13.5 Summary

All in all, the main functions in module `RegAlloc` can be described as follows.

`RegAlloc.g dest cont regenv e` allocates registers in instruction sequence `e`. It takes into account the continuation instruction sequence `cont` when calculating live variables. Already allocated variables in `e` are substituted with registers according to the mapping `regenv`. The value computed by `e` is stored to `dest`.

`RegAlloc.g'` is similar to `RegAlloc.g` but takes individual instructions (`SparcAsm.exp`) instead of instruction sequences (`SparcAsm.t`). However, it still *returns* instruction sequences—not individual instructions—so that spilling and unspilling can be inserted. It uses auxiliary functions `RegAlloc.g'_call` and `RegAlloc.g'_if` to deal with spilling due to function calls and conditional branches, while unspilling is treated by another auxiliary function `RegAlloc.g'_and_unspill`.

All of the functions above return either `NoSpill(e', regenv2)` or `ToSpill(e, xs)`. The former means that register allocation has succeeded: `regenv2` is the new mapping from variables to registers, and `e'` is the instruction sequence where all variables have been substituted with the allocated registers. The latter means that register spilling is required: `xs` is the list of spilled variables, and `e` is the instruction sequence where `Forget` pseudo-instructions have been inserted. Both results must be treated by every caller of `RegAlloc.g` or `RegAlloc.g'`.

Finally, `RegAlloc.h` takes a top-level function definition and allocates registers. `RegAlloc.f` takes a whole program and allocates registers. Actually, it is the only function exported by module `RegAlloc`.

4.14 Assembly Generation (256 Lines)

```
val Emit.f : ochan -> SparcAsm.prog -> unit

(* private *)
type dest = Tail | NonTail of Id.t
val Emit.h : ochan -> SparcAsm.fundef -> unit
val Emit.g : ochan -> dest * SparcAsm.t -> unit
val Emit.g' : ochan -> dest * SparcAsm.exp -> unit
```

At last, we reach the final phase: assembly generation. Having done most of the hard work (register allocation, in particular), it is easy to output `SparcAsm.t` as real SPARC Assembly by replacing virtual instructions with real ones. Conditional expressions are implemented by comparisons and branches. `Save` and `Restore` are implemented with stores and loads by calculating the set `stackset` of already saved variables (to avoid redundant saves) and the list `stackmap` of their locations in the stack. Function calls are a little trickier: `Emit.shuffle` is used to potentially re-arrange arguments in register order.

```
(* given a list (xys) of parallel moves,
   implements it by sequential moves
   using a temporary register (tmp) *)
let rec shuffle tmp xys =
  (* remove identical moves *)
  let _, xys =
    List.partition (fun (x, y) -> x = y) xys in
  (* find acyclic moves *)
  match (List.partition
           (fun (_, y) -> List.mem_assoc y xys)
           xys) with
  | [], [] -> []
  | (x, y) :: xys, [] ->
      (* no acyclic moves; resolve a cyclic move *)
      (y, tmp) :: (x, y) ::
      shuffle tmp
        (List.map
           (function
             | (x', y') when x' = y -> (tmp, y')
             | xy -> xy)
           xys)
  | xys, acyc -> acyc @ shuffle tmp xys
```

Tail calls are detected and optimized in this module. For this purpose, function `Emit.g` (which generates assembly for instruction sequences) as well as function `Emit.g'` (which generates assembly for individual instructions) takes a value of data type `Emit.dest` that represents whether we are in a tail position:

```
type dest = Tail | NonTail of Id.t
```

If this value is `Tail`, we tail-call another function by a jump instruction, or set the result of computation to the first register and return by the `ret` instruction of SPARC. If it is `NonTail(x)`, the result of computation is stored in `x`.

4.15 Main Routine, Auxiliary Modules, and Runtime Library (45 + 228 + 197 Lines)

After parsing command-line arguments, the main routine of MinCaml applies all the processes above. It also repeats the five optimizations from β-reduction to elimination of unnecessary definitions until their result reaches a fixed point (or the number of iterations reaches the maximum specified by a user).

Finally, we provide a few auxiliary modules, write the runtime routine `stub.c` which allocates the heap and stack of MinCaml, implement external functions `libmincaml.s` in SPARC Assembly for I/O and math, and obtain The MinCaml Compiler.

5. Efficiency

The main point of MinCaml was to let students understand how functional programs can be compiled into efficient code. So we had to demonstrate the efficiency of the code generated by MinCaml. For this purpose, we implemented several applications and compiled them with MinCaml, Objective Caml, and GCC. Each program was written in the optimal style of each language implementation, so that the compiler produces as fast code as possible (to the best of our knowledge) without changing the essential algorithms. These comparisons are never meant to be "fair," in the sense that MinCaml supports only a tiny language—in fact, it is *intended* to be minimal—while other compilers support real languages. Rather, they must be understood as informal references.

First, as small benchmarks, we chose three typical functional programs: Ackermann, Fibonacci, and Takeuchi (also known as Tak) functions. The first two of them test recursion on integers, and the last on floating-point numbers. The results are shown in Table 3. All the numbers are user-level execution times in seconds, measured by `/usr/bin/time`.

The machine is Sun Fire V880 (4 Ultra SPARC III 1.2GHz, 8GB main memory, Solaris 9). MinCaml is given the option `-inline 100`, meaning to inline functions whose size (the number of syntactic nodes in K-normal forms) is less than 100. OCamlOpt is version 3.08.3 and given the options `-unsafe -inline 100`. GCC `-m32` and GCC `-m64` are version 4.0.0 20050319 and given the option `-O3`. GCC `-m32 -mflat` is version 3.4.3 (since more recent versions do not support `-mflat`) and given the same option `-O3`. Note that GCC4 (and, to a lesser degree, GCC3) often produces faster code than older versions such as GCC2.

Although small benchmarks typically suffer from subtle effects of low-level mechanisms in a particular processor—such caches, alignments, and pipelines—our programs did not: indeed, looking at the assembly generated by each compiler, we found more obvious reasons for our results:

- Objective Caml and GCC3 do not inline recursive functions, while MinCaml and GCC4 do.
- Objective Caml boxes—i.e., allocates in the heap—floating-point numbers passed as arguments (or returned as results) of functions in order to support polymorphism, though it does support unboxed *arrays* (and records) of floating-point numbers.
- GCC without `-mflat` (both `-m32` and `-m64`) uses the register window mechanism of SPARC, which is almost always less efficient than other function calling conventions because it saves (and restores) *all* registers including unused ones.
- GCC with `-mflat` uses a callee-save convention instead of register windows, which is still suboptimal since it only saves registers in the prologues of functions (and restores them in their epilogues), not in the middle of them.
- GCC4 reduces arithmetic expressions such as $(n-1)-2$, which appears after the inlining of Fibonacci, to $n-3$.
- GCC `-m32` (with or without `-mflat`) passes floating-point number function arguments through integer registers, which incurs an overhead.

Second, we tested larger applications: ray tracing, a harmonic function, the Mandelbrot set, and Huffman encoding. All of them are first written in C and then ported to ML. In Objective Caml, we adopted an imperative style with references and `for`-statements whenever it is faster than a function style. However, we always used tail recursion in MinCaml, since it does not have any other loop construct. The results are also in Table 3. Again, Objective Caml tends to be slower than other compiles because of boxing when floating-point numbers are used as arguments of functions or elements of tuples (which cannot be replaced with arrays because they contain other types of elements as well). MinCaml also tends to be a little slower than GCC because loops are implemented by tail recursive functions, and entering to (or leaving from) them requires extra saves (or restores) of variables not used within the loops. In addition, GCC implements instruction scheduling for floating-point operations in order to hide their latencies, while MinCaml does not.

	Min-Caml	OCamlOpt `-unsafe`	GCC4 `-m32`	GCC4 `-m64`	GCC3 `-m32 -mflat`
Ackermann	0.3	0.3	1.3	1.8	1.0
Fibonacci	2.5	3.9	1.5	1.4	6.1
Takeuchi	1.6	3.8	3.7	1.6	5.5
Ray Tracing	3.4	7.5	2.3	2.9	2.6
Harmonic	2.6	2.6	2.0	2.0	2.0
Mandelbrot	1.8	4.6	1.7	1.7	1.5
Huffman	4.5	6.6	2.8	3.0	2.9

Table 3. Execution Time of Benchmark Programs

To summarize, for these modest benchmarks that can be written in our minimal language, the efficiency of MinCaml is comparable to major compilers such as Objective Caml and GCC with the speed ratio varying from "6 times faster" at best to "twice slower" at worst.

6. Related Work

There exist many compilers for ML and its variants: Comp.Lang.ML FAQ [1] gives a comprehensive list. However, I am not aware of any publicly available compiler that is as simple and efficient as MinCaml. There also exist various textbooks and tutorials on compilation of functional languages, but most of them present compilers into byte code or other medium-level languages—not native assembly—which do not satisfy our requirement for efficiency. The only exception that I am aware of is a well-known book by Appel [5], which uses CPS as the intermediate language and is distinct from MinCaml as argued below.

Hilsdale et al. [14] presented a compiler for a subset of Scheme, implemented in Scheme, that generates native assembly. However, efficiency of the generated code is not discussed at all, perhaps because it was not a goal in their compiler.

Sarkar et al. [23] developed a compiler course (using Scheme) based on the *nanopass* framework, where the compiler consists of many small translation (or verification) processes written in a domain specific language developed for this purpose. Unlike nanopass, we chose to use ordinary ML as the meta language in order to avoid the overhead of understanding such a domain specific language itself, and to utilize the type system of ML for statically checking the syntactic validity of intermediate code even *before* running the compiler.

Feeley [10] presented a Scheme-to-C compiler which is supposed to be explained in "90 minutes" and implemented in less than 800 lines of Scheme. Its main focuses are on CPS conversion and closure conversion for first-class continuations and higher-order functions. Optimizations are out of scope: indeed, the compiler is reported to produce 6 times slower code than Gambit-C

does. By contrast, our compiler is a little more complex but much more efficient.

Dijkstra and Swierstra [9] are developing a compiler for Haskell based on attribute grammar. It is presented as a sequence of implementations with increasing complexities. So far, their main focus seems to be on typing. To the best of my knowledge, little code or no documentation is available for compilation at this moment. In addition, the most complex version of their compiler is already about 10,000 lines long, excluding an implementation of their domain specific language based on attribute grammar.[3]

One [6] of Appel's series of textbooks implements a compiler *of* an imperative language (called Tiger) *in* ML. This language is not primarily functional and is fundamentally different from ML. For instance, higher-order functions and type inference are only optional [6, Chapters 15 and 16]. With those options, the compiler is much more complex than ours.

MinCaml adopts a variant of K-normal forms [7] as an intermediate language, which itself is a variant of A-normal forms [11]. Another major intermediate language of functional language compilers is continuation passing style (CPS) [5]. The crucial difference between K-normal forms and CPS, which lead us to choose the former, is conditional branches in non-tail positions: since all conditional branches must be in tail positions in CPS, non-tail branches are converted to tail branches with closure creations and function applications, which incur overheads and require optimizations (such as the so-called "callee-save" registers or inter-procedural register allocation).

On the other hand, however, CPS compiles function calls in a very elegant way without *a priori* assuming the notion of call stacks. Besides, K-normal forms have their own complication—which in essence stems from the same root—with non-tail branches (cf. the second and third paragraphs of Section 4.13.3) and, to a lesser degree, `let`-expressions (cf. the last paragraph of Section 4.13.1). Thus, it would also be interesting to see how simple and efficient compiler for education can be developed by using CPS instead of `let`-based intermediate languages.

As we saw in Section 4.13, the most complex process in The MinCaml Compiler was register allocation. Although there exist more standard methods than ours such as graph coloring [8] and linear scan [21], we find them less clear (though much faster at compile-time) in the context of functional languages, in particular concerning where and how to insert spilling and unspilling.

7. Conclusion

We presented an educational compiler, written in 2000 lines of ML, for a minimal functional language. For several applications that can be written in this language, we showed that our compiler produces assembly code of comparable efficiency to Objective Caml and GCC.

The use of MinCaml in Tokyo has been successful. Most of the groups accomplished the implementation of compilers and ran ray tracing on their CPUs. Some students liked ML so much that they started a portal site (`http://www.ocaml.jp/`) and a mailing list as well as a translation of the manual of Objective Caml, all in Japanese.

Like many program transformations in functional languages, most processes in our compiler are implemented by tree traversal over abstract syntax and have many similarities to one another. For instance, functions `KNormal.fv` and `Closure.fv` are almost identical except for the necessary differences such as `let rec` and `make_closure`. This kind of similarities could perhaps be exploited to simplify the compiler even more through subtyping (by means of polymorphic variants [13], for example) or generic programming in the style of Generic Haskell (`http://www.generic-haskell.org/`).

Although our language was designed to be minimal, its extensions would be useful for more advanced—maybe graduate—courses, and perhaps as a vehicle for research prototypes. Features required for these purposes include polymorphism, data types, pattern matching, garbage collection, and modules. We are looking into the tradeoff between simplicity and efficiency of various methods for implementing them.

We chose SPARC Assembly as our target code because of its simplicity and availability in Tokyo, but re-targeting to IA-32 would also be interesting from the viewpoint of popularization in spite of the more complex instruction set architecture. We are also looking into this direction—in particular, how to adapt our code generator to 2-operand instructions (which are destructive by definition) in a "functional" way.

[3] Of course, line numbers are not always an exact measure of software complexity—in particular for different languages—but they often approximate it with a certain precision.

Acknowledgments

I would like to thank Dr. Yutaka Oiwa for porting the ray tracer to MinCaml. Prof. Kenichi Asai, Prof. Kazuhiko Kato, Prof. Kenjiro Taura, and Dr. Yoshihiro Oyama gave useful comments and suggestions, encouraging this project.

References

[1] Comp.lang.ml FAQ. `http://www.faqs.org/faqs/meta-lang-faq/`.

[2] The computer language shootout benchmarks. `http://shootout.alioth.debian.org/`.

[3] ICFP programming contest. `http://icfpcontest.org/`.

[4] MLton Standard ML compiler. `http://mlton.org/`.

[5] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[6] A. W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[7] L. Birkedal, M. Tofte, and M. Vejlstrup. From region inference to von neumann machines via region representation inference. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, 1996.

[8] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, pages 98–101, 1982.

[9] A. Dijkstra and S. D. Swierstra. Essential Haskell compiler. `http://catamaran.labs.cs.uu.nl/twiki/st/bin/view/Ehc/WebHome`.

[10] M. Feeley. The 90 minute Scheme to C compiler. `http://www.iro.umontreal.ca/~boucherd/mslug/meetings/20041020/`.

[11] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 237–247, 1993. In *ACM SIGPLAN Notices*, 28(6), June 1993.

[12] B. Ford. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*, pages 36–47, 2002.

[13] J. Garrigue. Programming with polymorphic variants. In *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, 1998.

[14] E. Hilsdale, J. M. Ashley, R. K. Dybvig, and D. P. Friedman. Compiler construction using Scheme. In *Functional Programming Languages in Education*, volume 1022 of *Lecture Notes in Computer Science*, pages 251–267. Springer-Verlag, 1995.

[15] T. Jim. What are principal typings and what are they good for? In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 42–53, 1996.

[16] J. L. Lawall and O. Danvy. Continuation-based partial evaluation. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, volume VII of *ACM SIGPLAN Lisp Pointers*, pages 227–238, 1994.

[17] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–283, 1996.

[18] S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[19] A. Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, 1995.

[20] S. Peyton Jones and S. Marlow. Secrets of the Glasgow Haskell Compiler inliner. *Journal of Functional Programming*, 12(4):393–434, 2002.

[21] M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[22] D. Rémy. Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, 1994.

[23] D. Sarkar, O. Waddell, and R. K. Dybvig. A nanopass infrastructure for compiler education. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming*, pages 201 – 212, 2004.

[24] P. Wadler. Why no one uses functional languages. *SIGPLAN Notices*, 33(8):23–27, 1998.