Lambda-lifting as an optimization for compiling Scheme to C

Tanel Tammet

Department of Computer Sciences,
Chalmers University of Technology
S-41296 Göteborg, Sweden
e-mail: tammet@cs.chalmers.se
fax: +46 31 165655
phone: +46 31 7721053

Keywords: Scheme, compilation, global optimisations, lambda-lifting.

Abstract. We describe an optimizing Scheme-to-C compiler Hobbit. The innovation implemented in Hobbit is the usage of the technique of lambda-lifting (see [Hughes 84], [Johnsson 85], [Peyton-Jones 87]) combined with standard closures to compile lambda expressions and higher-order functions in the context of an impure functional language like Scheme and the object language like C. Lambda-lifting avoids the need for accessing the variables through an environment.

Lambda-lifting is not always applicable to lambda-terms in Scheme. It is applicable only in case the code surrounding the lambda-term satisfies certain criteria, which is checked by Hobbit using global analysis. In all the other cases the compiler reverts to using closures.

Lambda-lifting as an optimization for compiling Scheme to C

Tanel Tammet
Department of Computer Sciences,
Chalmers University of Technology,
41296 Göteborg, Sweden
email: tammet@cs.chalmers.se

Abstract

We describe an optimizing Scheme-to-C compiler Hobbit. The innovation implemented in Hobbit is the usage of the technique of lambda-lifting (see [Hughes 84], [Johnsson 85], [Peyton-Jones 87]) combined with standard closures to compile lambda expressions and higher-order functions in the context of an impure functional language like Scheme and the object language like C. Lambda-lifting avoids the need for accessing the variables through an environment.

Lambda-lifting is not always applicable to lambda-terms in Scheme. It is applicable only in case the code surrounding the lambda-term satisfies certain criteria, which is checked by Hobbit using global analysis. In all the other cases the compiler reverts to using closures.

1 Introduction

C has been proposed and used ([Page, Muller 92], [TAL 91], [Bartlett 89]) as a portable target language for implementing functional languages. C is relatively close to assembly language, yet mostly machine independent. C compilers are available on most machines, and ordinarily include a carefully developed optimizer. Using an optimizing C compiler as a last stage of compilation means that the Scheme-to-C compiler does not have to take care of a variety of possible optimizations handled by the C compiler instead.

On the other hand, using assembly directly as a target language often enables the compiler to generate faster code than is achievable by using C as a middle level. Thus the choice of C is an alternative which has both its pros and cons. One of the aims behind the design of Hobbit is to keep most of the pros of using C by producing C code which is structurally as similar to the compiled Scheme code as practically possible.

In order to enable the optimizing C compiler use most of its optimization power, it is recommendable that the structure of the output C program were similar to the input Scheme program. Preserving the structural similarity of the output program to the input program is also important in case of using a mixed programming environment, where some procedures are written in Scheme and some directly in C, or the user modifies the C output of the compiler manually, possibly in order to introduce special optimizations.

Partly due to our aim to keep structural similarity we prefer to avoid the continuation-passing style used by several Scheme compilers. As a next step, we are minimizing the usage of closures (pairs of procedure code and environment), preferring to use lambda-lifting whenever possible. Scheme procedures are called directly as C functions whenever

possible, without a mid-layer of calling `apply`. These techniques often translate Scheme variables directly into C variables, thus removing the dereference levels of accessing variables which are necessary in the general case.

Hobbit is intended to be used together with the portable Scheme interpreter Scm, written and maintained by Aubrey Jaffer. Scm is used as a basis for the GNU extension language Guile, currently in the development phase. Hobbit treats Scm files as a C library and provides integration of compiled functions with the Scm interpreter. It should be easy to modify Hobbit for other scheme-function libraries in C, different from Scm, in case such a library supports garbage collection off C-stack. Hobbit is written in Scheme, has a size of approximately 200 Kbytes and compiles full Scheme as defined in Revised[4] Scheme Report (R4SR). Since we have preferred to avoid continuation-passing style, Hobbit necessarily deviates from R4SR in the following aspect: it does not fully conform to the requirement of being properly tail-recursive (i.e. reusing stack space for all tailrecursive calls). Non-mutual tailrecursion (`foo` calling `foo` tailrecursively) is detected by Hobbit, but mutual tailrecursion (`foo` calling `bar` and `bar` calling `foo` tailrecursively) is not.

2 Obstacles at compiling Scheme to efficient C

One of the crucial questions for the speed and the structural similarity of the output C program is the mapping of the Scheme variables to the C objects. Scheme variables do not have associated property lists like Lisp variables and there is no analogue to the Lisp `set` function (Scheme `set!` is analogous to the Lisp `setq`). Thus there remain two main obstacles in mapping a Scheme variable directly to a C variable, necessary for avoiding an extra overhead each time the variable is accessed:

- The garbage collector must have an access to all variables allocated in the call stack, in order to avoid removing objects which may be needed in the future computation.

  The standard solution in earlier C implementations of Scheme has been using a special data structure containing all local variables, so that these could be always accessed by the garbage collector. Maintaining this data structure, however, imposes an overhead and creates a certain structural difference for a C procedure as opposed to the corresponding Scheme procedure.

  The innovation of the Scheme interpreters Siod (by George Carrette) and Scm (by Aubrey Jaffer) stands in using the call stack of C instead of a special explicit data structure. The garbage collectors of Siod and Scm mark objects on the calling stack of C, assuming them to be pointers to Scheme objects. The calling stack, including the contents of registers, is accessible in C by using the `setjmp` procedure. Such a scheme (conservative garbage collector) is guaranteed to never remove accessible objects, though it may occassionally fail to remove some actually unaccessible objects. The probability of the latter happening is low.

- Since Scheme contains lambda expressions, higher-order procedures and continuations, the value of a variable is generally dependent on the environment. I.e, a value of a variable is accessed via a special table. C variables, in contrast, have a fixed location and are accessible directly without dereference.

For the full Scheme the need for variables to be accessed via an environment appears to be unavoidable. However, for procedures satisfying certain criteria an environment can be avoided altogether. For such procedures the Scheme variables can be accessed directly, and thus can be mapped one-to-one to C variables, avoiding any overhead. We use a special modification of the well-known lambda-lifting technique (instead of using environment-containing closures) to achieve the one-to-one mapping of variables.

3 Terminology

We will assume that locally bound variables are renamed so that no name clashes occur in a program file. Hobbit performs such renaming before starting to analyse a source file.

A scheme variable x is either predefined as a primitive, defined on the top level of a Scheme source file or is local to some procedure. The variables of first two kinds will be called *top-level variables*, all the other will be called *local variables*.

We will say that a variable x is an *assigned variable* iff x either occurs in a construction `(set!` x t) where t is an arbitrary term or x occurs in two different constructions of the form `(define` x t). The second case can occur only if x is a top-level variable. Assignment analysis is possible in case the source code is assumed to be closed, see the section 4.2.

We will say that a non-assigned variable is *lambda-defined* iff it gets a lambda-term as a value upon creation.

We will say that a term t occurs in a *function position* iff t occurs as a first element of an application term: $(t\ a_1\ \ldots\ a_n)$.

An *SCM object* is a datastructure in C which represents a Scheme object, that is, either an unboxed integer, Boolean, character or a C pointer to a boxed object (eg. cons cell). The SCM type is a type of the SCM object.

4 Analysis and Compilation

4.1 Immediate Transformations

1. All the macros and quasiquotes are expanded and transformed into equivalent form without macros and quasiquotes. For example, `'(a ,x)` will be converted to `(cons 'a (cons x '()))`.

2. `Define`-constructions with the nonessential syntax like `(define (foo x) ...)` are converted to `define`-constructions with the essential syntax, in our case `(define foo (lambda(x)...))`. Non-top-level `define`-constructions are converted into equivalent `letrec`-constructions.

3. Variables are renamed to avoid name clashes, so that any local variable may have a whole procedure as its scope. The renaming algorithm converts `let`-s to `let*`-s. Variables which do not introduce potential name clashes are not renamed. For example,

```
(define (foo x y)            |    (define foo (lambda (x y)
   (let ((x y)               |       (let* ((x_1 y)
```

```
    (z x))                |=>            (z x)
(let* ((x (+ z x)))       |              (x_2 (+ z x_1)))
  x)))                    |          x_2))
```

4. The case of mutually recursive definitions in `letrec`-constructions needs special treatment – all the free variables in mutually recursive procedures have, in general, to be passed to each of those procedures. For example, in

```
(define (foo x y z i)
  (letrec ((f1 (lambda (u) (if x (+ (f2 u) 1) 0)))
           (f2 (lambda (v) (if (zero? v) 1 (f1 z)))) )
    (f2 i) ))
```

   the procedure `f1` contains a free local variable `x` and the procedure `f2` contains a free local variable `z`. Both `f1` and `f2` must have access to both `x` and `z`. The lambda-lifted form of `f1` and `f2`:

```
(define foo-f1
  (lambda (x z u) (if x (+ (foo-f2 x z u) 1))) )

(define foo-f2
  (lambda (x z v) (if (zero? v) 1 (foo-f1 x z z))) )
```

   In case the set of functions defined in one `letrec`-construction is actually not wholly mutually recursive (e.g, `f1` calls `f2`, but `f2` does not call `f1`, or there are three functions, `f1, f2, f3` such that `f1` and `f2` are mutually recursive but `f3` is not called from `f1` or `f2` and it does not call them), it is possible to minimize the number of additional variables passed to functions ([Peyton-Jones 87]).

   The `letrec`-constructions are therefore split into ordered chunks using dependency analysis and topological sorting, to reduce the number of mutually passed variables. Wherever possible, `letrecs`-s are replaced by `let*`-s inside these chunks.

5. The majority of Scheme control operators like `cond, case, or, and` are transformed into equivalent expressions using a small set of primitives, mainly just `if` and `let*`, possibly introducing new local variables.

   Local type inference is performed for Booleans. In case a primitive procedure like `or` or `and` occurs in the place where its value is treated as a Boolean (e.g. a first argument of `if`), we introduce an optimization by converting it into an analogous Boolean-returning procedure which will finally be open-coded using a C operator, in our case `||` or `&&`. Primitive associative procedures are converted into structures of corresponding nonassociative procedures. Primitive higher-order procedures `map` and `for-each` with more than two arguments are open-coded in Scheme as an equivalent `do`-cycle. `map` and `for-each` with two arguments are treated as if they were defined in the compiled file – the extra definitions for `map1` and `for-each1` are automatically included. This is necessary for the general scheme of handling clonable higher-order procedures in Hobbit, to be explained later.

4.2 Global Analysis

1. Source file is analysed for determining which top-level variables are nonassigned. Such analysis is always possible in case the source program is assumed to be *closed*: it does not contain `load` or `eval` and the resulting program is not going to be used under an interactive top-level. Since the last fact cannot be determined by Hobbit, a user can give Hobbit a set of parameters, saying which top-level variables, if any, are allowed to be assigned outside the program source.

2. Clonability of higher-order procedures (HOP-s for short) is analysed: Hobbit will build a list of clonable HOP-s with associated information about higher-order arguments.

   A HOP is defined as a procedure with some of its parameters occuring in the procedure body in a function position. Such an argument is called a "higher-order argument".

   A HOP *bar* given as a value to a variable `bar` is called *clonable* iff it satisfies the following four conditions:

   (a) `bar` is a nonassigned top-level variable.

   (b) The variable `bar` occurs inside the body of *bar* only in a function position and not inside an internal lambda-term.

   (c) Let `f` be a higher-order parameter of *bar*. Any occurrence of `f` in *bar* must have one of the following two forms:

   - `f` occurs in a function position,
   - `f` is passed as an argument to *bar* and in this application term it occurs in the same position as it occurs in the parameter list.

   (d) Let `f` be a higher-order parameter of `bar`. Then `f` must not occur inside a lambda-term occurring in *bar*.

   Example If `member-if` is a nonassigned top-level variable, then `member-if` is a clonable HOP:

```
(define (member-if fn lst)
   (if (fn (car lst))
       lst
       (member-if fn (cdr lst)) ))
```

   `member-if-not` is not a clonable HOP, since `fn` occurs in a lambdaterm:

```
(define (member-if-not fn lst)
  (member (lambda(x)(not (fn x))) lst) )
```

   `show-f` is not a clonable HOP, since `fn` occurs in a non-function position:

```
(define (show-f fn x)
    (set! x (fn x))
    (display fn)
    x)
```

3. Analysis of liftability: Hobbit will determine which lambda-terms have to be represented as real closures (implemented as a vector where the first element is a pointer to a function and the rest contains values of environment variables or environment blocks of surrounding code) and which lambda-terms are liftable.

We will say that a lambda-term l is *liftable* iff it satisfies the following five criterias (observe that this is a recursive definition and notice that t in `(define` $(f \ldots)$ $t)$ occurs inside a lambda-term):

1. l occurs either as an argument to a clonable HOP, primitives `map` or `for-each` or it is given as a value to a non-assigned variable v, for example in `(define` v $l)$ or in `(let` $((v\ l)) \ldots)$.

2. if l occurs inside a lambda-term, then l does not contain a non-liftable lambda-term.

3. if l is given as a value to a non-assigned variable v, then v occurs only in the function position. We note that under certain restrictions it is possible to allow v also occur as an argument to a clonable HOP or primitives `map` or `for-each`. We will not go into exact details here.

4. if l is given as a value to a non-assigned variable v, then v does not occur in a non-liftable lambda-term t such that v itself is created outside t.

5. if l is given as a value to a non-assigned variable v in a `letrec`-construction

$$\texttt{(letrec } ((v_1\ l_1) \ \ldots \ (v\ l) \ \ldots \ (v_n\ l_n))\ b)$$

   then each l_i must be a liftable lambda-term and v must not occur in a non-liftable lambda-term in any l_i or b. Observe that Hobbit performs dependency analysis and reorganization of the `letrec`-construction before it starts to analyse lambda-liftability. This is important in order to increase the amount of lambda-liftable lambda-terms.

Example The lambda-term `(lambda(u)(lambda(v)(+ u v)))` in the following example violates the case 2 above, since it occurs inside a lambda-term and contains a non-liftable lambda-term `(lambda(y)(+ x y))` itself:

```
(define (bar x y)
  (let ((f (lambda(u)(lambda(v)(+ u v)))))
    ((f x) y) ))
```

The lambda-term in the following example violates the case 3 above, since `f` occurs in a non-function position in `(set! w f)`, thus the lambda-term `(lambda(x)(+ x y))` is not liftable:

```
(define (bar y z w)
  (let ((f (lambda(x)(+ x y))))
     (set! w f)
     (cons (f (car z)) (map f z)) ))
```

If we remove `(set! w f)` from the above definition, then `(lambda(x)(+ x y))` becomes liftable.

The lambda-term in the third example violates the case 4 above, since `f` occurs in the non-liftable lambda-term `(lambda(u)(f (+ 1 u))` and `f` is created outside this term:

```
(define (bar y z w)
  (let ((f (lambda(x)(+ x y))))
     (lambda(u)(f (+ 1 w))) ))
```

If we remove `f` from `(lambda(u)(f (+ 1 u))`, then `(lambda(x)(+ x y))` becomes liftable.

4.3 Building Closures

Hobbit builds closures before lambda-lifting. In principle it is also possible to lambda-lift before (or simultaneously to) closure-building. In the latter two cases it is possible to change the liftability criteria somewhat. The presented liftability criteria is correct for the case where closure-building code is created before lambda-lifting starts.

All the lambda-terms which are not marked as being liftable by the previous liftability analysis will be transformed to closure-building code.

Top-level variables (eg. `z`) are generally translated as pointers (locations) to SCM objects and used via a C fetch, eg. `*z` . Functional occurrences of lambda-defined top-level variables are translated directly to C function names.

While producing closures Hobbit tries to minimize the dereference levels necessary. In the general case a local variable x may have to be translated to an element of a vector of local variables built in the procedure creating x. If x occurs in a non-liftable closure, the whole vector of local variables is passed to a closure.

Such a translation using a local vector will only take place if either x is assigned a value by `set!` inside a non-liftable lambda-term or x is a name of a recursively defined non-liftable function, and the definition of x is irregular. The definition of x is *irregular* if x is given the non-liftable recursive value t by extra computation, eg as `(set! x (let ((u 1)) (lambda(y)(display u)(x (+ u 1)))))`
and not as a simple lambdaterm: `(set! x (lambda(y)(display x)(x (+ y 1))))`.

In all the other cases a local scheme variable x is translated directly to a local C variable x having the type of a C pointer. If such an x occurs in a non-liftable closure, then only its value is passed to a closure via the closure-vector. In case the directly-translated variable x is passed to a liftable lambda-term where it is assigned by `set!`, then x is passed indirectly by using its address `&x`. In the lifted lambda-term it is then accessed using C fetch operator `*`.

If all the variables $x_1, \ldots, x_n$ created in a procedure can be translated directly as C variables, then the procedure does not create a special vector for (a subset of) local variables.

An application term `(foo ...)` is generally translated to C by an internal apply of the scm interpreter: `apply(*foo,...)`. Using an internal apply is much slower than using a direct C function call, since:

- there is an extra fetch done by `*foo`,
- internal apply performs some computations,

- the arguments of foo are sometimes passed as a list constructed during application: that is, there is a lot of expensive consing every time `foo` is applied via an internal apply.

However, in case `foo` is either a name of a non-assigned primitive or is lambda-defined, then the application is translated to C directly without the extra layer of calling apply: `foo(...)`.

Sometimes lambda-lifting generates the case that some local variable `x` is accessed not directly, but by a fetch as `*x`. See the next section.

Undefined procedures are assumed to be defined via interpreter and are called using an internal apply.

Example The following simple procedure contains a non-liftable lambda-term. None of the local free variables are assigned.

```
(define plus
  (lambda (x)
     (lambda (y) (+ y x)) ))
```

is transformed into two procedures

```
(define plus
  (lambda (x)
    (let* ((newclosure
             (**make-cclo** plus_cl1_clproc0 2) ))
      (vector-set! newclosure 1 x)
      newclosure)))

(define plus_cl1
  (lambda (closurearg)
    (let* ((closurearg_car (car closurearg))
           (x (vector-ref closurearg_car 1))
           (y (begin
                (set! closurearg (cdr closurearg))
                (car closurearg))))
      (+ y x))))
```

where `**make-cclo**` represents the C procedure which builds a compiled closure, that is, a vector having the pointer to the procedure as its first element. `plus_cl1_clproc0` will be instantiated to a new Scheme primitive implemented as `plus_cl1`. The inside procedure `plus_cl1` takes a list as an argument with the closure-vector as its first element. We are planning to improve Hobbit by passing arguments to compiled closures directly, without building an argument list.

The procedures `plus` and `plus_cl1` will be finally compiled to C as

```
SCM plus(x)
SCM x;
{
  SCM newclosure;
```

```
  newclosure=makcclo(plus_cl1_clproc0,2);
  VECTOR_SET(newclosure,MAKINUM(1),x);
  return newclosure;
}

SCM plus_cl1(closurearg)
SCM closurearg;
{
  SCM closurearg_car,x,y;

  closurearg_car=CAR(closurearg);
  x=VECTOR_REF(closurearg_car,MAKINUM(1));
  closurearg=CDR(closurearg);
  y=CAR(closurearg);
  return MAKINUM(INUM(y)+INUM(x));
}
```

where `MAKINUM` and `INUM` do conversions between C integers and unboxed Scheme integers.

4.4 Lambda-lifting

When this pass starts, all the non-liftable closures have been translated to closure-creating code. The remaining lambda-terms are all liftable. Lambda-lifting means that all the procedures defined locally inside some other procedure (e.g. in `letrec`) and unnamed lambda expressions are made top-level procedure definitions. Any n variables not bound in these procedures which were bound in the surrounding procedure are added as extra n first parameters to the procedure, and whenever the procedure is called, the values of these variables are given as extra n first arguments. For example:

```
(define foo
  (lambda (x y)
     (letrec ((bar (lambda (u) (+ u x))))
       (bar y) )))
```

is converted to two top-level definitions

```
(define foo                  |     (define foo-bar
  (lambda (x y)              |        (lambda (x u)
     (foo-bar x y) ))        |           (+ u x) ))
```

4.4.1 Modification of lambda-lifting for variable assignment

The standard procedures of lambda-lifting ([Hughes 84], [Johnsson 85], [Peyton-Jones 87]) are used in the context of languages without the variable assignment operator. Since Scheme has an assignment operator `set!`, we have to introduce an additional indirection for passing local free variables which are assigned with the `set!`.

Whenever some free variable is assigned a value by `set!` in the procedure, this variable is passed by reference instead. This is not directly possible in Scheme, but it is possible in C, due to the address-taking and fetching operators `&` and `*`.

Example The procedure

```
(define foo
  (lambda (x y z)
    (letrec ((bar (lambda (u) (set! z (cons u (cons x z))))))
      (bar y)
      z)))
```

is first converted to incorrect (pseudo-C) Scheme containing type operators in the parameter list:

```
(define foo                        | (define foo-bar
  (lambda (x y z)                  |   (lambda (x (**c-adr** z) u)
    (foo-bar x (**c-adr** z) y)    |     (set! (**c-fetch** z)
    z))                            |       (cons u
                                   |         (cons x
                                   |           (**c-fetch** z) )))))
```

The last two will finally be compiled into correct C as:

```
SCM foo(x,y,z)
SCM x,y,z;
{
  foo_bar(x,&z,y);
  return z;
}

SCM foo_bar(x,z,u)
SCM x,u;
SCM *z;
{
  return (*z = CONS(u,CONS(x,*z)));
}
```

In case an already `*`-prefixed variable `x` has to be passed to a procedure by address for further assignments, the address-taking operator `&` cancels the fetch operator `*`. I.e, any construction `&*x` is replaced by `x`, since `x` is an address (a C pointer) already.

4.5 Statement-lifting

Hobbit compiles the Scheme `do`-loop into the C `for`-loop, since most of the optimising compilers for C are able to use the presence of `for` for various loop optimisations. However, `for` cannot occur everywhere in the C code, since `for` is a statement. Thus, if the `do`-construction in a Scheme procedure occurs in a place which will not be a statement in C, the whole `do`-construction is lifted out into a new top-level procedure analogously to lambda-lifting, making the free variables in the lifted `do`-construction to be the parameters of the newly created procedure.

The special pseudo-C function `**return**` is pushed into a Scheme term as far as possible to extend the scope of statements in the resulting C program.

Example `**return**` is pushed into the procedure `foo`:

```
(define foo                    |       (define foo
  (lambda (x y)                |         (lambda (x y)
      (if x                    | ==>       (if x
         (cons x y)            |               (**return** (cons x y))
         (cons y x) )))        |               (**return** (cons y x)) )))
```

and the result will be compiled to the following C function:

```
SCM foo(x,y)
SCM x,y;
{
  if(BOOL(x))
    return(CONS(x,y));
  else
    return(CONS(y,x));
}
```

where `BOOL` is a macro converting a Scheme object to a C integer following the conventions for Booleans. Notice that we can use the C statement `if ... ... else ...` here instead of the `... ? ... : ...` operator only due to extending the scope of statements by pushing the `return` operator inside.

Immediate tailrecursion (`foo` calling `foo` tailrecursively) is recognized during statement-lifting and it is then converted into an assignment of new values to arguments of `foo` and a jump to the beginning of the C function body. Unfortunately, Hobbit is unable to recognize mutual tailrecursion (`foo` calling `bar` and `bar` calling `foo` tailrecursively), which is required by R4RS.

4.6 Cloning Higher-Order Procedures

In parallel to cloning the higher-order procedures Hobbit handles procedures with list arguments. All procedures taking a list argument are converted to ordinary non-list taking procedures and they are called with the list-making calls inserted. For example,

```
(define foo (lambda (x . y) (cons x (reverse y))) )
```

is converted to

```
(define foo (lambda (x y) (cons x (reverse y))) )
```

and any call to `foo` will make a list for a variable `y`. For example, a call `(foo 1 2 3)` is converted to a call `(foo 1 (cons 2 (cons 3 '())))`.

The principal problem arising in compiling clonable HOP-s in the context of lambda-lifting and avoiding closures stands in the need to carry free variables in the argument term into the body of the higher-order procedure. Since one higher-order procedure is generally called at several places in the program and each call may provide an argument term with a different number of free variables, we need several different instances of the HOP, one for each number of free variables to be passed inside.

To be more exact: consider a clonable HOP with functional parameters $x_1, \ldots, x_r$. Each call to the procedure specifies a pair $(n_i.m_i)$ for each parameter x_i, where n_i is the number of non-assigned free variables and m_i is the number of assigned free variables in the corrresponding argument term. Thus each call to and each copy of the HOP has a characteristic tuple of integer pairs $(n_1.m_1), \ldots, (n_r.m_r)$. Whenever the characteristic tuples of two distinct calls to the procedure coincide, one corresponding instance of the procedure can serve both.

Example Consider the following clonable HOP:

```
(define (member-if fn lst)
   (if (fn (car lst))
       lst
       (member-if fn (cdr lst)) ))
```

and a call `(member-if (lambda (x) (eq?  x y)) lst)` where `y` is a free local variable. Hobbit creates a new instance of `member-if` (if an analogous one has not been created before) along with the new top-level procedure `foo`:

```
(define (member-if-inst1 npar-1 fn lst)          |  (define (foo y x)
   (if (fn tmp (car lst))                        |    (eq? x y))
       lst                                       |
       (member-if-inst1 npar-1 fn (cdr lst)) ))  |
```

The above instance of `member-if` is compiled to C as:

```
SCM member_if_inst1(npar_1,fn,lst)
SCM (*fn)();
SCM npar_1,lst;
{
tailrecursion:
  if(BOOL((*fn)(npar_1,CAR(lst))))
    return lst;
  else {
    lst=CDR(lst);
    goto tailrecursion;
  }
}
```

The call is converted to `(member-if-inst1 y foo lst)` and finally compiled to C as `member_if_inst1(y,foo,lst)`.

In order to provide callability of a compiled higher-order procedure with closures as arguments, an additional instance is created which uses an internal apply to call all argument procedures.

Example Consider a call `(member-if f lst)` where `f` is neither a nonassigned Scheme primitive nor a top-level lambda-defined variable. Hobbit creates a generic instance of `member-if`, in case an analogous one has not been created before, and this instance is compiled to C as

```
SCM member_if_closurecase(fn,lst)
SCM fn,lst;
{
tailrecursion:
  if(NFALSEP(apply(fn,CAR(lst),listofnull)))
    return lst;
  else {
    lst=CDR(lst);
    goto tailrecursion;
  }
}
```

where `apply(fn,CAR(lst),listofnull)` is a call to an internal `apply`. `(member-if f lst)` is compiled to `member_if_closurecase(f,lst)`.

4.7 Type Conversion

The next pass introduces specific Scheme↔C conversions for unboxed objects: integers, booleans and characters. Internal apply is introduced for undefined procedures. Some optimizations are performed to decrease the amount of C↔Scheme object conversions.

All vector, pair and string constants are replaced by new variables. These variables are instantiated to the right values in the special initialization procedure produced by Hobbit. A special list is generated for all those variables to protect them from garbage collection.

4.8 C Code Generation

Straightforward. Some built-in Scheme procedure names are replaced by the corresponding C function names defined in the Scm interpreter. Scheme variable names are converted to correct C names, adding special suffixes and numbers in case it is necessary to avoid name clashes. C control operators like `if, for, return, ||, &&, &, *` are used for corresponding Scheme operators and compiler-introduced pseudo-C operators. Integer arithmetics is open-coded by C operators plus Scheme↔C object conversion macros (performing bit operations adding or removing type tags). The result of the compiled and linked program has the form of a Scheme interpreter with the compiled procedures available as primitives.

5 Gain in Speed: Some Examples

The author has so far used Hobbit for compiling several large automated theorem provers which have been written with special care to make the Hobbit-compiled version run fast. The speedup for the provers was between 25 and 50 times for various provable formulas.

According to experiments made by A.Jaffer, the compiled form of his pi-computing speed benchmark `pi.scm` was approximately 11 times faster than the interpreted form, whereas his handed-coded (in C) version of the same algorithm was just 12 times faster than the interpreted form.

Selfcompilation speeds Hobbit up ca 10 times.

However, there are examples where the code compiled by hobbit runs actually slower than the same code running under interpreter: this may happen in case the speed of the code relies on non-liftable closures and proper mutual tailrecursion. See for example the closure-intensive benchmark CPSTAK in the following table.

We will present a table with the performance of three scheme systems on Gabriel benchmarks originally written for Lisp and modified for Scheme by Will Clinger.

Hobbit performs well on most of the benchmarks except CPSTAK and CTAK: CPSTAK is a closure-intensive tailrecursive benchmark and CTAK is a continuations-intensive benchmark.

SCM and Hobbit benchmarks were run giving ca 8 MB of free heap space before each test. All times are in seconds. The number in parentheses, if present, shows the part of time spent in garbage collection. Notice that the tested versions of Scm and Hobbit are not optimized for continuations. Hobbit is not optimized for closures and proper mutual tailrecursion.

Columns:

- *scm-dec:* Scm interpreter (version 4c0) run by Matthias Blume on a DecStation 5000 (Ultrix).
- *vscm-dec:* VSCM byte-code compiler run by Matthias Blume on a DecStation 5000 (Ultrix).
- *scm-sun:* Scm interpreter (version 4e2) run by Tanel Tammet on a Sun SS-10 (Solaris 2.4), correctness checks prohibited, integer-only version for all the benchmarks except fft.
- *hobbit-sun:* Hobbit-compiled by Tanel Tammet on the same Sun SS-10 (Solaris 2.4) as Scm interpreter.

benchmark	*scm-dec*	*vscm-dec*	*scm-sun*	*hobbit-sun*
Deriv	3.40	3.86	2.9	0.18
Div-iter	3.45	2.12	2.6	0.083
Div-rec	3.45	2.55	3.5	0.42
TAK	1.81	1.71	1.4	0.018
TAKL	14.50	11.32	13.8(1.8)	0.13
TAKR	2.20	1.64	1.5	0.018
Destruct	?	?	7.4(1.8)	0.18
Boyer	?	?	27.(3.8)	1.9
CPSTAK	2.72	2.64	1.92	3.46(2.83)
CTAK	31.0	4.11	memory	memory
CTAK(7 6 1)	?	?	0.83	0.74
FFT	12.45	15.7	10.8	1.0
Puzzle	0.28	0.41	0.26	0.03

References

[Bartlett 89] J.F. Bartlett. SCHEME→C: A Portable Scheme-to-C Compiler. Technical report, DEC Western Research Laboratory, 100 Hamilton Avenue, Palo Alto, CA 94301 USA, Jan 1989.

[Hughes 84] R.J.M. Hughes. The design and implementation of programming languages. PhD thesis, PRG-40, Programming Research Group, Oxford. September 1984.

[Johnsson 85] T. Johnsson. Lambda-lifting: transforming programs to recursive equations. In *Conference on Functional Programming Languages and Computer Architecture, Nancy.* Jouannaud (ed). LNCS 201. Springer-Verlag 1985.

[Page, Muller 92] J.R.Page, H.Muller. Integrating the Scheme and C languages. In *Proc. of the 1992 ACM conf. on LISP and Functional Programming*, pp 247-259. ACM Press, 1992.

[Peyton-Jones 87] S.L.Peyton-Jones. The Implementation of Functional Programming Languages. Prentice-Hall 1987.

[TAL 91] D. Tarditi, A.Acharya, P.Lee. No Assembly Required: Compiling Standard ML to C.