

Down with λ -lifting

Erik Meijer and Ross Paterson

email: erik@cs.kun.nl and rap@doc.imperial.ac.uk

Abstract

Simplifications of the Spineless Tagless G-Machine and TIM are presented, which like the classic SECD machine or the Krivine machine reduce λ -expressions to weak head normal form — no prior λ -lifting is necessary. The machines are at least as efficient as their combinator forebears but more importantly they are simpler due to the elimination of a translation step that obfuscates programs without improving their efficiency.

1 Introduction

In early implementations of functional languages the values of free variables occurring in an expression were accessed via an environment mapping those free variables to their values. Combinator based implementations eliminate free variables by a process of β abstraction usually called *λ -lifting* which turns an expression into a combinator by passing the values of its free variables as explicit arguments. The fixed combinator method eliminates free variables by introducing a fixed set of combinators, i.e. the well-known S, K, I combinators. The super-combinators method yields a specific set of ‘user defined’ combinators for each expression. We claim that eliminating free variables by some form of λ -lifting is unnecessary and even repugnant as it obfuscates programs without improving their efficiency — a reappraisal of conventional compiler technology.

2 Implementing the λ -calculus

As a source language we simply take pure λ -expressions. Adding data types, pattern matching and other sugar poses no fundamental problems, but at the moment would only obscure the presentation needlessly.

$$A, B, E, F \in \text{expr} ::= \text{var} \mid \lambda \text{var}.\text{expr} \mid \text{expr expr}$$

We let x, y, z range over var .

The following stack based semantics serves as the starting point for further developments. The various semantic domains are defined as

$$\begin{aligned} \eta \in \text{env} & == \text{var} \rightarrow \text{susp} \\ \mathbf{a}, \mathbf{b} \in \text{susp} & == \text{stack} \rightarrow \text{whnf} \\ \xi \in \text{stack} & ::= [] \mid \text{susp} \succ \text{stack} \\ \text{whnf} & ::= \text{WHNF susp} \end{aligned}$$

The evaluations functions are straightforward.

$$\begin{aligned} \mathcal{E}[_] & \in \text{expr} \rightarrow \text{env} \rightarrow \text{stack} \rightarrow \text{whnf} \\ \mathcal{E}[x] \eta \xi & = \eta \ x \ \xi \\ \mathcal{E}[F A] \eta \xi & = \mathcal{E}[F] (\mathcal{E}[A] \eta \succ \xi) \\ \mathcal{E}[\lambda x. B] \eta (\mathbf{a} \succ \xi) & = \mathcal{E}[B] \eta [x := \mathbf{a}] \xi \\ \mathcal{E}[\lambda x. B] \eta [] & = \text{WHNF } \mathcal{E}[\lambda x. B] \eta \end{aligned}$$

Evaluation halts when a λ hits an empty stack, the value is thus in WHNF.

2.1 Environment trimming

In the above description environments are ‘shared’; $\mathcal{E}[F A] \eta \xi = \mathcal{E}[F] \eta (\mathcal{E}[A] \eta \succ \xi)$. An alternative is that each suspension has its own private environment consisting of precisely the values of its free variables.

Let $\mathcal{FV} e$ denote the set of free variables occurring in the expression e , and let $\eta \upharpoonright s$ be the restriction of environment η to the variables occurring in s . Then a refinement for the above rule for function application may be expressed as follows

$$\mathcal{E}[F A] \eta \xi = \mathcal{E}[F] (\eta \upharpoonright \mathcal{FV} F) (\mathcal{E}[A] (\eta \upharpoonright \mathcal{FV} A) \succ \xi)$$

If a function body contains no free occurrences of its binding variable, the rule for λ -expressions may be refined to

$$\begin{aligned} & \mathcal{E}[\lambda x. B] \eta (\mathbf{a} \succ \xi) \\ = & \text{evaluate} \\ & \mathcal{E}[B] (\eta [x := \mathbf{a}] \upharpoonright \mathcal{FV} B) \xi \\ = & \quad x \notin \mathcal{FV} B \\ & \mathcal{E}[B] \eta \xi \end{aligned}$$

The net effect being that \mathbf{a} is removed from the stack.

The restricted environments of argument expressions indicate the trimmed suspensions that are created for them, thus each suspension gets a tailor-made environment instead of sharing a single global one. For the function part, trimming merely shows to what extend variable slots in its environment may be reused.

2.2 An optimization for bound variables

When the argument of an application is a variable, an optimization is possible whether or not environments are trimmed or shared.

$$\begin{aligned}
 & \mathcal{E}[\text{F } x] \eta \xi \\
 = & \text{ unfold} \\
 & \mathcal{E}[\text{F}] \eta (\mathcal{E}[x] \eta \varkappa \xi) \\
 = & \text{ unfold} \\
 & \mathcal{E}[\text{F}] \eta (\eta x \varkappa \xi)
 \end{aligned}$$

As all other cases for $\mathcal{E}[_]$ make use of ξ , no other such optimization is possible.

3 Intermediate Code

The semantics function $\mathcal{E}[_]$ can be partially evaluated into a compiler $\mathcal{C}[_] \in \text{expr} \rightarrow \text{I_code}$ and a lower level semantics $\mathcal{M}[_] \in \text{I_code} \rightarrow \text{env} \rightarrow \text{stack} \rightarrow \text{whnf}$ with $\mathcal{E}[_] = \mathcal{M}[_] \circ \mathcal{C}[_]$, if the various alternatives of $\mathcal{E}[_]$ are refined by introducing explicit run-time actions. Those actions should be such that the meaning of expressions can be expressed in terms of them without referring to the environment nor the stack. Environment and stack are not available at compile-time.

$$\begin{aligned}
 \mathcal{C}[_] & \in \text{expr} \rightarrow \text{I_code} \\
 \mathcal{C}[x] & = \text{JUMP } x \\
 \mathcal{C}[\text{F } A] & = \text{PUSH } \mathcal{A}[A]; \mathcal{C}[f] \\
 \mathcal{C}[\lambda x. B] & = \lambda x. \{ \mathcal{C}[B] \} \\
 \mathcal{A}[x] & = \text{ARG } x \\
 \mathcal{A}[A] & = \text{SUSP } [\mathcal{FV } A] \{ \mathcal{C}[A] \}
 \end{aligned}$$

The advantage of using this intermediate code is that it leaves the choice between an implementation based on shared or private environments open, depending on $\mathcal{M}[_]$. Moreover the idea of splitting a semantic function into compile-time and run-time parts may be applied recursively to $\mathcal{M}[_]$ as well. We will refrain from defining $\mathcal{M}[_]$.

3.1 Example

As an example we take the addition of Church numerals

$$\text{add} = \lambda m. \lambda n. \lambda f. \lambda x. m \text{ f } (n \text{ f } x)$$

which will be translated into

$$\begin{aligned}
 \text{add} & = \lambda m. \lambda n. \lambda f. \lambda x. \{ \text{PUSH } (\text{SUSP } nfx); \text{PUSH } (\text{ARG } f); \text{JUMP } m \} \\
 nfx & = [n, f, x] \{ \text{PUSH } (\text{ARG } x); \text{PUSH } (\text{ARG } f); \text{JUMP } (\text{ARG } n) \}
 \end{aligned}$$

Although some may recognize TIM [?] in the above code, the real connoisseur will spot the Krivine [?] machine.

4 Register Machines

If we assign names to intermediate results instead of PUSH-ing them anonymously on the stack, we greatly facilitate code generation and optimization. The translation scheme is easily modified to generate three-address code by introducing a compile-time stack $\sigma \in \text{c_stack} == \text{reg}^*$. At run-time we need an additional *register set* mapping the names given to the intermediate results to their values.

$$\begin{aligned}
 \mathcal{C}[_] &\in \text{expr} \rightarrow \text{c_stack} \rightarrow \text{I_code} \\
 \mathcal{C}[x] \sigma &= \text{JUMP } x \ \sigma \\
 \mathcal{C}[F A] \sigma &= r := \mathcal{A}[A]; \mathcal{C}[F] (r \succ \sigma) \\
 \mathcal{C}[\lambda x. B] (r \succ \sigma) &= [x := r] \{ \mathcal{C}[B] \ \sigma \} \\
 \mathcal{C}[\lambda x. B] [] &= \lambda x. \{ \mathcal{C}[B] [] \} \\
 \mathcal{A}[x] &= \text{ARG } x \\
 \mathcal{A}[A] &= \text{SUSP } [\mathcal{FV} A] \{ \mathcal{C}[A] [] \}
 \end{aligned}$$

Notice that as a side-effect, applications $(\lambda x. B) A$ are now reduced at compile-time; the resulting code becomes $r := \mathcal{A}[A]; [x := r] \{ \mathcal{C}[B] \ \sigma \}$ instead of $\text{PUSH } \mathcal{A}[A]; \lambda x. \{ \mathcal{C}[B] \}$ which causes pointless run-time movement of data to and from the stack.

4.1 Comparing with the STGM

For combinator source code there is virtually no difference between our implementation and combinator based implementations such as the STGM [?]. The contrast arises in programs like

$$Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

where a λ -expression with free variables occurs in argument position; the typical situation to perform λ -lifting. The combinatorized equivalent of Y would be something like

$$\begin{aligned}
 Y f &= \text{c4711 } f \ (\text{c007 } f) \\
 \text{c4711 } f \ x &= f (x x) \\
 \text{c007 } f \ x &= f (x x)
 \end{aligned}$$

which translates into the following T-code.

```

SUPERCOMBINATOR Y [f] {
  r1 := SUSP [f] { r1 := f; JUMP c007 [r] };
  r2 := f;

```

```
JUMP c4711 [r2,r1]
}
```

```
SUPERCOMBINATOR c4711 [f,x]{
  r1 := SUSP [x] { r1 := x ; JUMP x [r] };
  JUMP f [r1]
}
```

```
SUPERCOMBINATOR c007 [f,x]{
  r1 := SUSP [x] { r1 := x ; JUMP x [r] };
  JUMP f [r1]
}
```

The only effect of λ -lifting is the introduction of a superfluous move of f onto the stack and back again when calling the newly introduced redexes `c4711 f` and `c007 f`.

Our translation yields the following code, where the effect of compile-time reduction is nicely visible.

```
Y =  $\lambda f.$ {r1 := SUSP c4711; [x := r1]{r2 := SUSP xx1; JUMP f [r2]}}
c4711 = [f]{ $\lambda x.$ {r1 := SUSP xx2; JUMP f [r1]}}
xx1 = [x]{r1 := ARG x; JUMP x [r1]}
xx2 = [x]{r1 := ARG x; JUMP x [r1]}
```

Or in a more spacy layout

```
Y =  $\lambda f.$ {
  r1 := SUSP [f] {
     $\lambda x.$ {
      r1 := SUSP [x]{r1 := ARG x; JUMP x [r1]};
      JUMP f [r1]
    }
  };
  [x := r1]{
    r2 := SUSP [x]{r1 := ARG x; JUMP x [r1]};
    JUMP f [r2]
  }
}
```

5 Updates

The idea of delaying updates until an expression is in WHNF using stack markers is due to Meijer []. Similar schemes have since been incorporated into TIM and the STGM. In order to formalize sharing, we need to introduce a *store* to accommodate for destructive assignments.

$$\mu \in \text{store} == \text{loc} \rightarrow \text{susp}$$

$$\eta \in \text{env} \quad == \quad \text{var} \rightarrow \text{loc}$$

We let α denote elements of loc .

An expression is in WHNF when it attempts to take another argument from the stack, so this is the time to update the suspension from which this expression originated. To recognize this situation, a list of *update markers* is maintained pointing into the stack and to the location to be updated.

$$\begin{aligned} \alpha \in \text{susp} & == \text{stack} \rightarrow \text{marker}^* \rightarrow \text{store} \rightarrow \text{whnf} \\ \xi \in \text{stack} & == \text{loc}^* \\ \delta \in \text{marker}^* & == (\text{loc}, \text{stack})^* \\ \text{whnf} & ::= \text{WHNF susp store} \end{aligned}$$

The valuation function $\mathcal{E}[_] \in \text{expr} \rightarrow \text{env} \rightarrow \text{stack} \rightarrow \text{store} \rightarrow \text{susp}$ makes the above intuition precise. When a λ hits an empty stack an update occurs, and the λ then retries to fetch its argument.

$$\mathcal{E}[\lambda x.B] \eta \ [] \ ((\alpha, \xi) \succ \delta) \ \mu = \mathcal{E}[\lambda x.B] \eta \ \xi \ \delta \ \mu[\alpha := \mathcal{E}[\lambda x.B] \eta]$$

If the list of update markers is also empty, evaluation terminates.

$$\mathcal{E}[\lambda x.B] \eta \ [] \ [] \ \mu = \text{WHNF} (\mathcal{E}[\lambda x.B] \eta) \ \mu$$

If there are enough arguments on the stack, the topmost is simply fetched and bound.

$$\mathcal{E}[\lambda x.B] \eta \ (\alpha \succ \xi) \ \delta \ \mu = \mathcal{E}[B] \eta[x := \alpha] \ \xi \ \mu$$

When an argument expression is a possibly sharable potential redex, the code for its suspension sets up an update marker.

$$\begin{aligned} \mathcal{E}[F A] \eta \ \xi \ \delta \ \mu & = \mathcal{E}[f] \eta \ (\alpha \succ \xi) \ \delta \ \mu[\alpha := a] \\ & \text{where} \\ & \alpha \ \xi \ \delta \ \mu = \mathcal{E}[A] \eta \ [] \ ((\alpha, \xi) \succ \delta) \ \mu \end{aligned}$$

Finally, calling a variable now goes indirectly via the store.

$$\mathcal{E}[x] \eta \ \xi \ \delta \ \mu = (\mu (\eta x)) \ \xi \ \delta \ \mu$$

5.1 Indirections

In an implementation based on trimmed environments suspensions may have different sizes, in that case updating should go via an indirection.

$$\begin{aligned} \mathcal{E}[\lambda x.B] \eta \ [] \ ((\alpha, \xi) \succ \delta) \ \mu & = \mathcal{E}[\lambda x.B] \eta \ \xi \ \delta \ \mu[\alpha := a] \\ & \text{where } a = \mathcal{E}[x] \eta[x := \mathcal{E}[\lambda x.B] \eta] \end{aligned}$$

6 Lazyness

Local definitions are introduced to make the treatment of lazyness more comprehensible. The compilation scheme for local definitions

$$\text{expr} ::= \mathbf{let} \text{ var} = \text{expr} \mathbf{in} \text{expr}$$

is simply $\mathcal{E}[\mathbf{let} \ x = A \ \mathbf{in} \ B] = \mathcal{E}[(\lambda x. B) \ A]$.

Eventhough **letrec** definitions could be handled similarly using Y as given earlier, it is more efficient to build cyclic environments.

$$\begin{aligned} \mathcal{E}[\mathbf{letrec} \ x = a \ \mathbf{in} \ B] \ \eta \ \xi \ \delta \ \mu &= \mathcal{E}[B] \ \eta' \ \xi \ \delta \ \mu[\alpha := a] \\ &\text{where} \\ a \ \xi \ \delta \ \mu &= \mathcal{E}[A] \ \eta' \ [\] \ ((\alpha, \xi) \multimap \delta) \ \mu \\ \eta' &= \eta[x := \alpha] \end{aligned}$$

Desugaring nonrecursive local definitions yields the following translation scheme:

$$\mathcal{C}[\mathbf{let} \ x = A \ \mathbf{in} \ B] \ \sigma = r := \mathcal{A}[A]; [x := r]\{\mathcal{E}[B] \ \sigma\}$$

However for improved readability and compatibility with **letrec** definitions we extend the intermediate code with two new constructs.

$$\begin{aligned} \mathcal{C}[\mathbf{let} \ x = A \ \mathbf{in} \ B] \ \sigma &= \mathbf{LET} \ x := \mathcal{A}[A] \ \{\mathcal{C}[B] \ \sigma\} \\ \mathcal{C}[\mathbf{letrec} \ x = A \ \mathbf{in} \ B] \ \sigma &= \mathbf{LETREC} \ x := \mathcal{A}[A] \ \{\mathcal{C}[B] \ \sigma\} \end{aligned}$$

assuming that A is taken to be an expression in $\mathcal{A}[A]$ (at least for **LETREC**).

Lambda hoisting

As Peyton Jones and Lester [?] also point out, λ -lifting and (full) lazyness can be decoupled by means of λ -hoisting, a technique originally proposed by Takeichi [?]: the introduction of **let**-definitions for maximal free subexpressions outside the λ 's in which they occur free. The usual caveats apply; e.g. floating **let**(**rec**) definitions outward until outside any lambda abstraction in which they occur free.

Consider for example $f = \lambda x. \lambda y. E[E']$ where $E[E']$ denotes an expression e with a subexpression E' containing no free occurrences of y . Then the λ -hoisted, fully lazy, version of f is:

$$f = \lambda x. \mathbf{let} \ z = E' \ \mathbf{in} \ \lambda y. E[z]$$

which translates into

$$f = \lambda x. \{ \mathbf{LET} \ z := \mathbf{SUSP} \ [FV \ E'] \{ \mathcal{C}[E'] \ [\] \} \{ \lambda y. \{ \mathcal{C}[E] \ [\] \} \} \}$$

Fully lazy λ -lifting yields

$$\begin{aligned} f\ x &= c1984\ E'\ x \\ c1984\ z\ x\ y &= E[z] \end{aligned}$$

Translating these into T-code gives

```
SUPERCOMBINATOR f [x]{
  r1 := x;
  r2 := SUSP [free-vars E'] {code for E'};
  JUMPG g [r2,r1]
}
```

```
SUPERCOMBINATOR c1984 [z,x,y]{
  code for E[z]
}
```

The work done is the same except that the STGM passes z via the stack by means of a run-time function call. In either case a marker between x and y will generate a suspension for E' .

7 Towards a concrete implementation

In this section we will differentiate between implementations based on private environments and those based upon shared environments. In particular the shape and hence the access to and from the environment differs.

7.1 λ -STGM

Implementations based on private environments are quite straightforward since the block structure is trivial. A variable can either be in a register or in the (flat) environment. Variables bound by λ -abstraction and **let(rec)** definitions are assigned to registers. The free variables passed to a suspension are bound in the environment.

7.2 λ -TIM

An implementation based on shared environments must implement full block-structure. To increase efficiency, as many variables as possible have to be bound 'en block'. In the classical implementation of block structure the address of a bound variable has two parts

1. the distance between the nesting depths of the applying and defining occurrences of the variable, specifying the number of *static links* to be traversed, and

2. the relative position i within the environment frame thus reached.

A trick to speed up the access to non-locals, due to Dijkstra, is to maintain a *display* of pointers to all environment frames below the 'current' environment. With the help of a display a variable can be accessed directly via the additional pointer into the respective frame.

Taken (too) strictly, the display is *at any time* a copy of the *complete* static chain. This means that the display has to be set up at every context switch. As suggested by Hoare to Wirth, [?], this means in practice that an implementation based on pure static links is usually the better choice. Keeping in the spirit of lazy evaluation, we propose to restore the display lazily as a side effect of walking the static chain. Though this seems very obvious, we have not seen this idea before.