Notes on Three Instruction Machine

A Sanyal

October 31, 2000

Abstract

This is an exposition of the Three Instruction Machine (TIM), an abstract machine for implementing lazy higher order functional programming languages. Starting with a version of the machine which performs normal order reduction, we explain how the machine is augmented to ensure laziness. Of special interest is the way shared partial applications are handled.

1 Introduction

The easiest method of compilation involves translation of functional programs into programs involving a fixed number of combinators.(for example **S** and **K**) This is not an efficient method because even simple programs could result in very large **SK**-programs. This has led to the study of general reduction strategies based on user-defined combinators instead of a fixed number of combinators. Typically, abstract machines are defined so that these user defined combinators can be efficiently implemented. The efficiency of implementation is thus directly related to the proper design of the abstract machine.

A simple abstract machine for lazy functional languages called the **Three Instruction Machine (TIM)** was developed by Fairbairn and Wray[?]. It is simple, efficient and easy to implement. Argo[?] improved **TIM** to include efficient built-in functions, economical representation of partial application, efficient full applications and a method for avoiding identical updates. Finally, Argo redesigned **TIM** and showed how program analysis could lead to a more efficient implementation.

An important step in the compiling process is the use of lambda lifting to convert function definitions into supercombinators. This eliminates globals from the body of the function. The environment at the body of a function thus consists of only the arguments passed to the function.

In the implementation of imperative programming languages, the normal place for arguments is the stack. However, a purely stack based implementation does not work for higher order functional programs for the following reason. Consider a function f which returns another function g in weak head normal form (i.e returned function g is already applied to some but not all arguments). In this case the arguments of g (which should normally appear contiguously on stack) get separated by the arguments of f. For example,consider the following program:

$f \ x = g \ (2 * x)$
$g \ a \ b = a + b$
$f \ 3 \ 4$

Before entering f the stack contains 3 and 4. If we create environments on the stack, then before entering g, the stack will contain $(2 * x)$, 3 and 4. Clearly, the two arguments of g, the $(2 * x)$ on top of the stack and 4 at the bottom have become separated by the middle 3 which is f's argument. Note that f's argument cannot be discarded since this will be used to resolve the value of x in $(2 * x)$. We call this the *split activation record* problem.

To solve this problem in TIM the arguments of a function are moved from the stack to the heap as soon as it is called. Because of this, the arguments of the returned function g will now be contiguous on the stack when g is called. In the above example, the intervening 3 is moved out in to the heap by f when it is called, thus making g's arguments contiguous.

Shifting arguments to heap incurs two overheads.

- The cost of moving the arguments from stack to heap.
- Reclaiming the heap space using garbage collection.

In G-TIM, Argo uses a scheme in which the arguments of the function being executed are kept on the stack in the form of a frame. This technique is applicable to first-order programs only, i.e. programs which do not take functions as arguments and return functions as results.

2 Fairbairn and Wray's TIM

The main feature in TIM is that every object in the machine is represented by a closure $< Code, Env >$, consisting of code pointer and an environment pointer. The machine comprises an argument stack, a frame heap and a code stream. The argument stack holds the arguments to functions as they are prepared prior to a functions call, while the heap frame holds the arguments to function while it is being executed. The registers are

PC (Program Counter): Points to the current instruction
CF (Current Frame) : points to the current frame
SP (Stack Pointer): points to the top of stack.

If $Code$ represents the code stream along with the value of $\mathbf{PC}$, Env represents the state of the heap with the value of $\mathbf{CF}$ and $Stack$ represents the stack, then the state of TIM can be represented as: $[< Code,\ Env >,\ Stack]$.

2.1 Take, Push and Enter

The three instruction of TIM are `Take`, `Push` and `Enter`. A function on entry will create its environment by moving its arguments from the stack to a heap frame. The `Take` instruction of TIM achieves this. The effect of `Take` is illustrated through a state transition rule such as:

$[< \texttt{Take}\ n;\ c,\ \ E >,\quad a_1 : ...... : a_n : S] \quad \Longrightarrow \quad [< c,\ \ E' >,\ \ S]$
where E' is $[a_1, ..., a_n]$

Only the relevant part of a state appears in a transition rule. Read this rule as follows. If the machine is in a state where the current instruction is `Take` n and the continuation (the rest of the program) is c, and the top of the stack contains a_1 : ...... : a_n, then the next state of the machine is such that the first instruction of c is to be executed next, the entities a_1 : ...... : a_n are popped from the stack and stored in a frame in the heap which now becomes the current frame.

As we have seen before, the `Take` instruction may not consume all the arguments on the stack, because the result of the called function may itself be a function which could consume the remainder. Let D be a function which describes the compilation rule for a function definition and A be the rule for compilation of an expression. Then we have

$D[[f\ \ a_1 \ldots a_n = exp]] \stackrel{def}{=}$ `f` : `Take` n; $A[[exp]]$
where `f:`, in the TIM code, represents a label.

To prepare arguments to subsequent functions, the `Push` instruction is used.

$$[< \texttt{Push}\ item;\ c,\ \ E >,\ \ S] \quad \Longrightarrow \quad [< c, E >,\ \ < c', E' >: S]$$

where $< c', E' > =$ case $item$ of

$$\begin{array}{lll} \texttt{Arg}\ n & \rightarrow & E[n] \\ \texttt{Comb}\ C & \rightarrow & < C, \phi > \\ \texttt{Label}\ j & \rightarrow & < j, E > \\ \texttt{Const}\ k & \rightarrow & < \texttt{Self}, k > \end{array}$$

And, finally, a function is called using the `Enter` instruction.

$$[< \texttt{Enter}\ \ item;\ c,\ \ E >,\ \ a : S >] \quad \Longrightarrow \quad state$$

where $state =$ case $item$ of

$$\begin{array}{lll} \texttt{Comb}\ C & \rightarrow & [< \texttt{C} : c', \phi >,\ \ a : S] \\ \texttt{Label}\ j & \rightarrow & = \ [< \texttt{j} : c', E >,\ \ a : S] \\ \texttt{Arg}\ n & \rightarrow & \text{case } E[n] \text{ of} \\ & & < C, \phi > \ \rightarrow \ [< \texttt{C} : c', \phi >,\ \ a : S] \\ & & < j, E' > \ \rightarrow \ [< \texttt{j} : c', E >,\ \ a : S] \\ & & < \texttt{Self}, k > \ \rightarrow \ [a,\ \ < \texttt{Self}, k >: S] \end{array}$$

Note the following

1. Executing `Push Comb` C involves pushing a closure consisting of the code pointer pointing to C and a null frame pointer ϕ. The frame pointer is null because C builds its own environment on entry.

2. `Label j` refers to the code starting at label j within the current environment. Therefore, during `Push Label` j, the closure pushed contains the current frame pointer.

3. `Const k` refers to a base (non-functional) value `k`. The closure consists of a special combinator `Self` paired with constant `k`. The use of the combinator `Self` is explained later.

These are similar to addressing modes of instructions on a real machine. Also notice that the environment part of a closure can be used in two different ways - either as a pointer to a frame or a constant.

3 Continuations

An interesting aspect of TIM's execution is the way continuations are handled. Consider a call $f\ \ a_1\ \ a_2 \ldots a_n$ to a function which will in general have a definition:

$f\ \ a_1\ \ a_2 \ldots a_n = g\ \ e_1\ \ e_2 \ldots e_m,$

where a_i and e_j are some expressions. In what follows, it should be noted that the arguments of a function are always evaluated right to left. Because of this, when they are pushed on the stack, the argument on the top of the stack is the first argument which is as it should be.

There are three cases to consider:

1. f is strict in none of its arguments: In this case the call to f would push labels for $a_1, \ldots, a_n$ on the stack and enter f. since the body of f solely consists of a call to g, the value returned by g is the same as the value returned by f. Hence there is no need for continuation to return to f from g. Note that in an equivalent imperative program definitions,

   $f(x_1, \ldots, x_n)$
   begin
   return $g(e_1, \ldots, e_m)$;
   end;

   control would come back to the return statement in f.

2. f is strict in all its arguments: In this case, the arguments $a_1, \ldots, a_n$ are first evaluated before f is entered. Evaluation of a_i means entering the main combinator in it. However, we have to force the order of evaluation from $a_1, \ldots, a_n$ by introducing continuations. Before a_i is executed, a continuation corresponding to a_{i-1} is pushed on the stack. The value returned on evaluating a_i is swapped with this continuation, and control passes to the evaluation of a_{i-1}. Thus $f\ a_1 \ldots a_n$ is compiled as:

```
l_f a1 a2...an:       Push l_f a1 a2...a(n-1)
                      ... code for a_n ...
l_f a1 a2...a(n-1)    Push l_f a1 a2...a(n-2)
                      ... code for a_(n-1) ...
                      ⋮
l_f a1           :    Push f
                      ... code for a_1 ...
```

   Note that when f is entered, the values of its arguments are on top of the stack.

3. f is strict in some, but not all, of its arguments:
   In this case, continuations corresponding to the strict arguments are put on the stack.

Clearly, we need a mechanism by which the constant value returned by an evaluation can be swapped with the continuation on the stack. This is achieved in TIM by representing constants as a pair $<$ `Self`, $const >$, where `Self` is a special combinator which performs the above operations.

4 Examples

1. Consider the code for the expression $S\ K\ K\ 3$, where $S\ f\ g\ x\ =\ f\ x\ (g\ x)$ and $K\ x\ y\ =\ x$. Here both S and K are not strict in any of their arguments.

```
 K:   Take 2
      Enter Arg 1

S:    Take 3
      Push Label l_(g x)
      Push Arg 3
      Enter Arg 1
```

```
l_(g x):    Push Arg 3
            Enter Arg 2

   main:    Push Comb halt
            Push Const 3
            Push Comb K
            Push Comb K
            Enter Comb S
```

2. Consider $(+\ (+\ 2\ 3)(+\ 4\ 5))$. Here $+$ is strict in both its arguments.

```
   main:    Push Comb halt
            Push Label l_(+4 5))
            Push Label l_(+2 3))
            Enter Comb +

l_(+4 5):   Push Const 5
            Push Const 4
            Enter omb +

l_(+2 3):   Push Const 3
            Push Const 2
            Enter Comb +
```

3. Consider $if\text{-}then\text{-}else\ (==\ \ 2\ 3)\ \ (+\ 5\ 1)\ 7$. Here $if\text{-}then\text{-}else$ is strict only in its first argument.

```
    main:   Push Comb Halt
            Push Const 7
            Push label l_(+ 5 1)
            Push Comb if-then-else
            Enter Label l_(2 == 3)

l_(== 2 3): Push Const 3
            Push Const 2
            Enter Comb ==

l_(+ 5 1):  Push Const 1
            Push Const 5
            Enter Comb +
```

4. Consider $f\ x\ =\ (+\ x\ 2)$. The body of f is compiled as:

```
      f:    Take 1
            Push Const 2
            Push Comb +
            Enter Arg 1
```

5 Reduction strategies and laziness

TIM, as described above, described above performs reductions in the normal order. In practice, normal order evaluation is much too inefficient since it reduces shared arguments more than once. To improve efficiency TIM needs a mechanism to save the result of the evaluation of a shared expression. Since we are dealing with supercombinators, the only possible sharing that can take place is due to several occurrences of the same argument in the body. Therefore, when a shared argument is evaluated for the first time, the result must be stored.

However, there is a problem with doing this. Consider for example, a call $f(+\ 3\ 5)$ for the function definition

$$f\ \ x\ \ y = if\text{-}then\text{-}else\ \ (y == 0)\ \ (+\ \ x\ \ x)\ \ (*\ \ x\ \ x).$$

It is not possible to statically determine the occurrence of x which will get evaluated first, and insert code to the updation. Hence, the first evaluation of x at run-time has to somehow update the same slot of f. However, the problem is that while evaluating x (in this case $(+\ 3\ 5)$), we have access only to the frame of the combinator $+$, and have no access to the frame of f. On the other hand, the continuation of $(+\ 3\ 5)$ cannot be easily determined. Hence just before evaluating x, it is necessary to explicitly push something called $Marker$, on the stack which contains a pointer to f's frame, and the slot within this frame where the update is to be carried out. After x is evaluated, the top of the stack is checked to see if it is a marker, and if so, the appropriate frame slot is updated with the value, and the marker is erased from the top of the stack.

However, this will also not work when the argument x is passed on lazily to some other function, for example in $f\ \ x = g\ \ x\ \ x$. The problem here is to figure out who is going to push the marker on the stack. It cannot be f, since f is not going to evaluate x. On the other hand, if g's definition is $g\ \ a\ \ b = (+\ a\ b)$, then g has no sharing, and so will not insert any code for updates. Thus, while preparing each argument to g, f has to push a label to some code which when entered will first push a marker on the stack and then actually enter the argument. Thus the body of f is compiled as:

```
        Take 1
        Push Label SI_1
        Push Label SI_1
        Enter g
 SI_1 : Push Marker 1
        Enter Arg 1
```

We update the semantics of `Push` and `Enter` to reflect this change:

$[< \texttt{Push}\ item;\ c,\ \ E >,\ \ S] \quad \Longrightarrow \quad [< c, E >,\ \ < c', E' >: S]$
where $< c', E' > =$ case $item$ of
$\quad\quad \texttt{Arg}\ n \ \ \rightarrow \ \ E[n]$
$\quad\quad \texttt{Comb}\ C \ \ \rightarrow \ < C, \phi >$
$\quad\quad \texttt{Label}\ j \ \ \rightarrow \ < j, E >$
$\quad\quad \texttt{Const}\ k \ \ \rightarrow \ < \texttt{Self}, k >$
$\quad\quad \texttt{Marker}\ i \ \ \rightarrow \ < Marker_i, E >$

$[< \texttt{Enter}\ \ item;\ c,\ \ E >,\ \ a : S >] \quad \Longrightarrow \quad state$
where $state =$ case $item$ of
$\quad\quad \texttt{Comb}\ C \ \ \rightarrow \ [< \texttt{C} : c', \phi >,\ \ a : S]$

$$
\begin{array}{l}
\texttt{Label}\ j \ \rightarrow\ =\ [<\texttt{j}:c', E>,\ \ a:S] \\
\texttt{Arg}\ n \ \rightarrow\ \ \ \text{case } E[n] \text{ of} \\
\qquad <C,\phi> \ \rightarrow\ [<\texttt{C}:c',\phi>,\ \ a:S] \\
\qquad <j,E'> \ \rightarrow\ [<\texttt{j}:c',E>,\ \ a:S] \\
\qquad <\texttt{Self},k> \ \rightarrow\ \text{case } a \text{ of} \\
\qquad\qquad\qquad\qquad <\texttt{Marker}_i, E> \ \rightarrow\ [<\texttt{Self},k>,\ \ S] \\
\qquad\qquad\qquad\qquad\qquad\qquad \text{where } E[i] = <\texttt{Self},k> \\
\qquad\qquad\qquad\qquad \_\_ \quad \rightarrow\ [a,\ \ <\texttt{Self},k>:S]
\end{array}
$$

6 Sharing Partial Applications

This solution however gives rise to a problem with the Take instruction in the case when a function returns a partially applied function. Recollect that executing SI_1 causes a marker to be pushed on the stack. If the result of entering an argument is a partially applied function, then some of its arguments will be above the marker while the rest, which have been prepared before, will be below the marker. Thus the marker splits the activation record of the function returned by the argument. We demonstrate this through a series of examples which will also serve to illustrate the different ways in which a partial application can be shared.

1. As the simplest example, consider

   $f\ \ x = x\ \ 1 + x\ \ 2$
   $g\ \ a\ \ b\ \ c = a + b + c$
   $f\ \ (g\ \ (1+2)\ \ (3+4))$

   Here, the shared argument of f is bound to $g\ \ (1+2)\ \ (3+4)$. When x is evaluated, the arguments of g get separated by a marker.

2. $plus\ \ a\ \ b = a + b$
   $f\ \ x = g\ \ x\ \ x$
   $g\ \ y\ \ z = h\ \ y$
   $h\ \ w = p\ \ w\ \ w$
   $p\ \ a\ \ b = a\ \ 3$
   $f\ \ (plus\ \ (1+2))$

   would introduce two contiguous markers, one to update f's frame with the value of $(plus\ (1+2))$ and another to update h's frame with the same value. These markers would separate the arguments of $plus$ which are $1+2$ and 3.

3. As another example, consider:

   $f\ \ x = g\ \ x\ \ x$,
   $g\ \ a\ \ b = h\ \ (a\ \ 3)$,
   $h\ \ y = p\ \ y\ \ y$, and
   $p\ \ u\ \ v = (u\ \ 4)$.
   $f\ (three\ 2)$

   Here $three$ is a ternary function. In this example, two separate portions of the partial application $(three\ \ 2\ \ 3)$ are being shared. First, $(three\ \ 2)$ is being shared by the two occurrences of the argument x of f. Second, $(three\ \ 2\ \ 3)$ is being shared by the the two occurrences of the argument y of h. Consequently, on entering $three$, there will be two markers, one to update f's frame with the result of $(three\ \ 2)$ and another to update h's frame with the value of $(three\ \ 2\ \ 3)$.

To deal with this situation, the `Take` instruction has to identify and remove the markers within the activation record. While doing this, it will have to perform updates of shared arguments as required by the markers. Assuming that the function being partially applied is f, the specific steps are:

1. Check for the occurence of the first marker within the arguments of the function.
2. Collect the arguments (say m of them) on the top of the marker and store them in a new frame E'.
3. Update the frame indicated by the marker with a special instruction indicating the fact that the evaluated value is a partial application of the function f on m arguments, which are stored in E'.
4. Remove the marker from the stack. Replace the m arguments with SI code refering to the frame E'.
5. Repeat the above steps till there are no intervening markers between the arguments of f.

Instead of overloading $Take$, TIM uses a separate instruction, `CheckMarkers`, to do this. This instruction is executed just before `Take`. which can detect a shortage of arguments above the first marker. This instruction is executed just before the `Take` instruction. The transition rule for this instruction is fairly complex:

$$
\begin{array}{l}
[<\texttt{CheckMarkers}\ f\ n;\ c, \_\!\_ >,\ \ a_1 : \ldots : a_m :< Markers_i, E >: S] \implies \\
\quad \text{if } m < n \text{ then } [<\texttt{CheckMarkers}\ f\ n;\ c, \_\!\_ >,\ \ I_1, \ldots, I_m : S], \\
\qquad \text{where } E[i] = <\texttt{Partial}\ f\ m, E' > \\
\qquad E' = [a_1, \ldots, a_m] \\
\qquad I_1 = <\texttt{SI\_1}, E' > \\
\qquad \vdots \\
\qquad I_m = <\texttt{SI\_}m, E' > \\
\quad \text{otherwise } [< c, \_\!\_ >,\ \ a_1 : \ldots : a_m :< Markers_i, E >: S]
\end{array}
$$

`CheckMarkers` loops until all shared partial applications have been stored in the location specified by their respective markers. The shared arguments are pushed into a new frame, E' , and their place on the stack is overwritten by indirections to preserve sharing.

What happens when the instruction `Partial` $f\ i$ is encountered. This indicates that a value consisting of f partially applied to i arguments, where the arguments are contained in E'. These arguments have to be pushed on the stack. However since the arguments contained in E' might not have been evaluated and might get evaluated during this call to f, we once again push i `SI` codes, each refering to the frame E'. Thus the code part of the closure describing a partial application of i arguments, `Partial` $f\ i$, is a label into a table of `Push Label SI` instructions preceding the code for the combinator. So the code for a combinator f of arity n looks like:

`Partial  f` $n-1$:
    `Push Label SI_`$n-1$

```
...
Partial  f  2:
    Push Label SI_2
Partial  f  1:
    Push Label SI_1
f:
    CheckMarkers f
    Take n
...
```

Ensure that you understand the entire scheme by generating and executing code for the following example:

$$
\begin{aligned}
&f \;\; x = x \;\; 1 + x \;\; 2\\
&h \;\; a = g \;\; (a + a)\\
&g \;\; p \;\; q = p + q\\
&f \;\; (h \;\; (1 + 3))
\end{aligned}
$$

In the absence of absence of program analysis, we have to treat everything as shared. However, if analysis is performed, then the information from such analysis can be used to annotate the variables that will be evaluated first and consequently will perform the updation. We indicate such variables by, for instance, putting a dash against it.

$$
\begin{aligned}
&fact \;\; n = if\text{-}then\text{-}else \;\; (n' == 0) \;\; 1 \;\; (n \;\; * \;\; (fact \;\; (n - 1)))\\
&program = (fact \;\; 10)
\end{aligned}
$$

In the example above, it is clear that the variable n' will be evaluated first and will therefore do the updation. The other two occurrences of n will find their binding already evaluated.

```
fact:
    CheckMarkers fact 1
    Take 1
    Push Label l_(n*(fact(n-1)))
    Push Const 1
    Push Comb if-then-else
    Enter Label l_(n'==0)

l_(n'==0):
    Push Const 0
    Push Comb ==
    Enter Label SI_1

l_n*(fact(n-1)):
    Push label l_(n*}
    Enter Label l_(fact(n-1))

l_(n*):
    Push Arg 1    /* n is already evaluated*/
    Enter Comb *
```

```
l_(fact(n-1)):
    Push Label l_(n-1)
    Enter Comb fact

l_(n-1):
    Push Const 1
    Push  Arg 1
    Enter Comb -

Program:
    Push Comb Halt
    Push Const 10
    Enter Comb fact
```