An Implementation of Transparent Migration on Standard Scheme

Eijiro Sumii*
sumii@saul.cis.upenn.edu
University of Tokyo

Abstract

I present a handy (though somewhat restrictive) way to implement mobile computation à la Telescript on top of standard Scheme.

Background. Mobile computation is an efficient and effective approach to distributed programming where a program works by migrating from one host to another. The migration is called transparent if the execution state of the program is preserved before and after the migration. Transparent migration is preferable to non-transparent, because it is easier to use for application programmers. At the same time, however, it is harder to implement for language developers: all existing implementations (to my knowledge) of transparent migration need either a custom runtime system (e.g. [13]) or global source code transformation (e.g. [12]).

Our Method. In this presentation, I describe a library to enable transparent migration in standard Scheme. Unlike existing implementations, it requires neither modification of the runtime system nor transformation of the source code. It works by (i) extracting a delimited continuation [5, 6] with control operators (shift and reset), (ii) modifying the delimited continuation—i.e., reconstructing its source code—with type-directed partial evaluation (TDPE) [2, 4], and (iii) evaluating the source code at the remote host (e.g. by invoking ssh). Assuming the function shift for delimited-continuation extraction and the operator ↓ for TDPE, the main part of the library looks like:

\begin{verbatim}
(begin (reset ; delimits the continuation
  (display "hello from ")
  (system "hostname")
  (go "remotethost") ; migrates to remotethost
  (display "another hello from ")
  (system "hostname")
  (display "yet another hello from ")
  (system "hostname")))
\end{verbatim}

It (i) executes the first display and system at the local host, (ii) extracts, refines, and remotely evaluates the delimited continuation \( \lambda_\_ (\text{display("another hello from "); system("hostname")}) \), and (iii) performs the third display and system at the local host. As a result, the program works as follows.

\begin{verbatim}
> (load "go.scm"); load the library
> (hellos)
hello from localhost
another hello from remotethost
yet another hello from localhost
\end{verbatim}

Example 2. In the following program,

\begin{verbatim}
(reset (for-each (lambda (host) (go host)
  (display "hello from ")
  (system "hostname")
  (("host1" "host2")))

the delimited continuation of the first migration (to host\(_1\)) is

\( \text{\lambda}_\_ (\text{display("hello from "); system("hostname")}; \text{go("host2")};
\text{display("hello from "); system("hostname")}) \)

while that of the second migration (to host\(_2\)) is \( \text{\lambda}_\_ (\text{display
("hello from "); system("hostname"})) \). It therefore yields the following output.

hello from host1
hello from host2

Our method thus unfolds all static recursions, as TDPE does. (In general, this may lead to code duplication or non-termination. See the limitations below for details.)

Example 3. In the program below, the delimited continuations of the first and second migrations are \( \text{\lambda}_\_ (\text{go("host2")}; \text{display}(3 + 4)) \) and \( \text{\lambda}_\_ \text{display}(3 + 4) \), respectively, so the output is 7.
(reset
  (let ((add (lambda (x) (lambda (y) (+ x y))))
        (go "host1")
        (let ((add-three (add 3))
              (go "host2")
              (display (add-three 4)))))
  (let ((add (lambda (x) (lambda (y) (+ x y))))
        (go "host1")
        (let ((add-three (add 3))
              (go "host2")
              (display (add-three 4))))))

As this example suggests, our method inlines all static functions as TDPE does. (This may also lead to code duplication.)

Limitations. Because of the limitations of TDPE, our method also has many limitations. To name some,

- The reification may not terminate if the delimited continuation contains a dynamic recursion, because TDPE does not terminate if the value has no normal form. Furthermore, the source code of the delimited continuation may become large, because TDPE unfolds all static recursions and inlines all static functions. We can mitigate these problems, however, by letting programmers use a special fixed-point operator that does not recur during reification.

- Operators that involve pointers (such as set!, set-car!, and eq?) may not work after migration, because pointers have no textual representation and cannot be reified.

- Higher-order operands of primitive operators (such as apply and call/cc) must have ML-like types, so that the operands can be reified by TDPE. Moreover, in my current implementation, the types need to be given by the programmer.

- Programmers must not take the value of a primitive operator itself (like (define add +)), because it must be substituted by set! in TDPE.

In general, it does not work when TDPE does not.

Conclusion. Despite the limitations above, I believe that our method may be useful for some applications because of its simplicity. In fact, it is so simple that my current prototype\footnote{available at http://www.yl.is.s.u-tokyo.ac.jp/~sumii/pub/} is only a few hundred lines long, including (a variant of) Filinski’s implementation ofayed TDPE [3], (a variant of) Danvy’s implementation of TDPE [4], comments, blank lines, and debugging code. This can be regarded as an achievement of Scheme’s features such as dynamic typing, set!, and call/cc, which are sometimes blamed as a source of its inefficiency.

Acknowledgements

I thank Olivier Danvy, Tatsuru Sekiguchi, and the anonymous reviewer for giving me valuable comments and Benjamin Pierce for proofreading my poor English.

Appendix A: Delimited Continuations

As an ordinary continuation stands for “the rest of the computation,” a delimited continuation (a.k.a. partial continuation) [5, 6] represents “the rest of the computation up to somewhere.” It is typically manipulated by the operators shift and reset: reset delimits a continuation, and shift extracts the continuation delimited by the “last” reset (in the sense of dynamic scope). For example, consider the following program.

\[
\begin{align*}
&(+ 1 (\text{reset} (\text{shift} (\lambda k. (\lambda x (\lambda y (\lambda z z)))) 5)))
\end{align*}
\]

The variable \( k \) is bound to the delimited continuation \( \lambda x. 2 \times x \times 5 \), so the whole program is evaluated to \( 1 + (k 3 \cdot k 4) = 1 + (30 \cdot 40) = -9 \). It is known that shift and reset can be implemented by means of call/cc and a mutable reference cell [7].

Layered continuations [8] are delimited continuations where only one set of continuations is implemented on top of another. They can be used for “multiple-level” control, e.g., as follows.

\[
\begin{align*}
&((\text{cons} 1 (\text{reset1} ((\text{cons} 2 (\text{reset0} (\text{cons} 3 (\text{shift0} (\lambda k. (\lambda x (\lambda y (\lambda z z)))))))))) 1 2 3 3 4) )
\end{align*}
\]

In this program, \( k_0 \) is bound to the continuation \( \lambda y. (3, y) \) delimited by \( \text{reset0} \), while \( k_1 \) is bound to the continuation \( \lambda z. (2, (3, z)) \) delimited by \( \text{reset1} \). It is known that such layered continuation operators can also be implemented by means of call/cc and mutable reference cells [8].

Appendix B: Type-Directed Partial Evaluation

Basics. Type-directed partial evaluation (TDPE) [2, 4] is a way to reify (i.e., unevaluate) a value to its source code (in a certain kind of normal form) by using its (ML-like) type. Consider, for example, a function \( f = \lambda x. (\lambda y. x) \) of type \( \alpha \to \alpha \).

\[
\begin{align*}
&((\text{define} f (\lambda x. (\lambda y. x))) f)
\end{align*}
\]

Since this value has the function type \( \alpha \to \alpha \), we can apply it, say, to a fresh symbol \( z \). Here, the underline denotes dynamic (i.e., quoted) expressions [10].

\[
\begin{align*}
&(f z)
\end{align*}
\]

Given the fresh symbol \( z \), the function returns the same symbol \( z \). Thus, we find it (extensionally) equivalent to the identity function \( \lambda z. z \).

When the results of the function to be reified have a compound type (such as a function type or a pair type), the reification is repeated according to that compound type until the function is fully applied. On the other hand, if the arguments of the function to be reified have a compound type, the fresh symbol (to which the function is applied) is re-expanded according to that compound type [2] so that no type error will be caused. For example, consider a function \( g = \lambda h. h \cdot (1 + 2) \cdot (\text{type function}) \) of type \( \text{int} \to \beta \to \beta \).

\[
\begin{align*}
&((\text{define} g (\lambda h. h \cdot (1 + 2))) g)
\end{align*}
\]

We apply it to the function \( \lambda x. x \cdot \text{int} \cdot w \), rather than the fresh symbol \( \underline{w} \), as follows. (The operator \( \underline{w} \) denotes dynamic function application.)

\[
\begin{align*}
&(g (\lambda x. x \cdot \text{int} \cdot w)) \underline{(3)}
\end{align*}
\]
Given the fresh symbol $\eta$-expanded according to the type \texttt{int $\rightarrow \beta$}, the function $g$ returned the dynamic expression $\eta \, \xi \, \gamma$. Therefore, it is equivalent to the function $\lambda x. \, v \, 3$.

Note that TDPE uses only the extension (semantics and type) of a program, so the intension of the program does not matter. For instance, it is straightforward to reify the "impute" function \texttt{(lambda (x) (set! x 123) x)} of type $\alpha \rightarrow \texttt{int}$ to the pure function $\lambda x. \, x \, 123$, because they have the same semantics.

**Disjoint Sums.** The first challenge in TDPE is disjoint sums (such as booleans) as function arguments. For example, consider a function $f = \lambda x. \, (1 + (if \, x \, \text{then} \, 2 \, \text{else} \, 3))$ of type $\texttt{bool} \rightarrow \texttt{int}$.

> (define f (lambda (x) (+ 1 (if x 2 3))));

We can reify this function by applying it to both \texttt{true} and \texttt{false}.

> (f #t)

3

> (f #f)

4

Since the function returned 3 and 4 for \texttt{true} and \texttt{false}, respectively, it is equivalent to $\lambda x. \, (\text{if} \, x \, 3 \, \text{else} \, 4)$. In general, a function whose arguments have a disjoint sum type can be reified by applying it to both "left" and "right" values of that disjoint sum type; this can be implemented by means of \texttt{if} constructs.

**Side Effects.** The second issue is side effects such as I/O. By default, TDPE treats all side effects as "static" (rather than dynamic). For example, consider a function $\lambda x. \, \text{display} \, (1 + 2)$ of type $(\texttt{int} \rightarrow \texttt{int})$. Applying the function to a fresh symbol causes the output to be performed.

> (define f (lambda (x) (display (+ 1 2))));

> (f #t)

3

If we want to defer the side effects (as we actually do in the case of mobile computation, since we want to migrate them to the remote host rather than executing them in the local host), we need to substitute the "effectful" operators with their dynamic counterparts, e.g. as follows.

> (define static-display display)

> (define dynamic-display (lambda (x) (display x)));

> (set! display dynamic-display)

> (f #t)

(display 3)

Furthermore, in order to avoid eliminating, duplicating, or reordering side effects, we actually have to sequentialize all dynamic function applications by binding them to variables by let-insertion [1]. The resulting (type-directed) partial evaluator is sound with respect to any monadic, dynamic effects [9, 11].

**Primitives.** The next problem is primitives (such as integers) as function arguments. Consider, for example, a function $f = \lambda x. \, (1 + x) \, x$ of type $\texttt{int} \rightarrow \texttt{int}$.

> (define f (lambda (x) (++ (12) x)));

Since there exists no value of type $\texttt{int}$ that represents all integers, we cannot "$\eta$-expand" a fresh symbol to an integer as we did for functions and pairs. Instead, we extend the operator $+$ (in general, every operator on integers) so that it can deal with symbols in addition to numbers [3].

> (define add+ +)

> (define (new+ x y)

> (if (and (number? x) (number? y))

> (+ x y))

> (set! + new+);

Then, it is safe to apply the function to a fresh symbol instead of an integer.

> (f #t)

(+ 3 #f)

Thus, the function is found to be equivalent to $\lambda x. \, (1 + 3 + x)$. Alternatively, we can residualize (rather than reduce) the addition $1 + 2$, if we want, as follows.

> (define (add+ x y) (+ x y));

> (set! + add+)

> (f #t)

(+ (+ 1 2) #f)

Here, the function is reified to $\lambda x. \, (1 + 2 + x)$ instead of $\lambda x. \, 3 + x$.

**Recursion.** The last (but not least) difficulty that we consider here is recursion, which can lead to non-termination of TDPE. For example, the reification of the power function $\texttt{fix}(\lambda \alpha x n. \, (\text{if} \, n = 0 \, \text{then} \, 1 \, \text{else} \, 2 \times x \, (n - 1)))$ does not terminate, because its "normal form" $\lambda n. \, (\text{if} \, n = 0 \, \text{then} \, 1 \, \text{else} \, (\text{if} \, n - 1 = 0 \, \text{then} \, 2 \, \text{else} \, (\text{if} \, n - 2 = 0 \, \text{then} \, 4 \, \text{else} \, \ldots)))))$ is infinite. A workaround for this problem is to use a special fixed-point operator which does not recur during reification [3], though this spoils a merit of TDPE—that it does not need the source code of the program.

**References**


