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) *reifying* 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 $\downarrow$ for TDPE, the main part of the library looks like:

```
(define (go rhost)
  (shift (lambda (k) ; extract the delimited continuation
    (let ([e (↓()→() k)]) ; reconstruct its source code
      (reval e rhost))))) ; remotely evaluate the source code
```

Note that the delimited continuation k has the type $() \rightarrow ()$, because the transparent migration operator **go** works as a side effect. Since TDPE itself uses control operators, the library actually uses *layered* control operators [8]. For details of (layered) delimited continuations and TDPE, see Appendix A and B, respectively.

Example 1. Consider the following program. (In the examples, we use the non-standard procedure *system* for the sake of convenience.)

```
(define (hellos)
  (begin (reset ; delimits the continuation
           (display "hello from ")
           (system "hostname")
           (go "remotehost") ; migrates to remotehost
           (display "another hello from ")
           (system "hostname"))
         (display "yet another hello from ")
         (system "hostname")))
```

It (i) executes the first *display* and *system* at the local host, (ii) extracts, reifies, and remotely evaluates the delimited continuation $\lambda\_.$ (**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.

```
> (load "go.scm") ; load the library
> (hellos)
hello from localhost
another hello from remotehost
yet another hello from localhost
```

Example 2. In the following program,

```
(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

$\lambda\_.$ (**display**(`"hello from "`); **system**(`"hostname"`);
  go(`"host2"`);
  display(`"hello from "`); **system**(`"hostname"`))

while that of the second migration (to $host_2$) is $\lambda\_.$ (**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 $\lambda\_.$ (**go**(`"host2"`); **display**$(3 + 4)$) and $\lambda\_.$ **display**$(3 + 4)$, respectively, so the output is 7.

*Visiting scholar at the University of Pennsylvania. Research fellow of the Japan Society for the Promotion of Science.

```
(reset
  (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 recurse 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[1] is only a few hundred lines long, including (a variant of) Filinski's implementation of layered control operators [8], (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, Tatsurou 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.

```
> (+ 1 (reset (* 2
                 (shift (lambda (k) (- (k 3) (k 4))))
                 5)))
-9
```

The variable k is bound to the delimited continuation $\lambda x.\, 2 \times x \times 5$, so the whole program is evaluated to $1 + (k\,3 - k\,4) = 1 + (30 - 40) = -9$. 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 one set of continuations is implemented on top of another. They can be used for "multiple level" control, e.g. as follows.

```
> (cons 1 (reset1 (cons 2 (reset0 (cons 3
                  (shift0 (lambda (k0) (k0 (k0 4)))))))))
(1 2 3 3 . 4)
> (cons 1 (reset1 (cons 2 (reset0 (cons 3
                  (shift1 (lambda (k1) (k1 (k1 4)))))))))
(1 2 3 2 3 . 4)
```

In this program, k_0 is bound to the continuation $\lambda y.\,(3, y)$ delimited by $\mathbf{reset_0}$, while k_1 is bound to the continuation $\lambda z.\,(2, (3, z))$ delimited by $\mathbf{reset_1}$. 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 $f = \lambda x.\,(\lambda y.\,y)\,x$ of type $\alpha \to \alpha$.

```
> (define f (lambda (x) ((lambda (y) y) x)))
> f
#⟨procedure⟩
```

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

```
> (f 'z)
z
```

Given the fresh symbol $\underline{z}$, the function returns the same symbol $\underline{z}$. 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 η-expanded according to that compound type [2] so that no type error will be caused. For example, consider a function $g = \lambda h.\,h\,(1+2)$ of type $(\mathbf{int} \to \beta) \to \beta$.

```
> (define g (lambda (h) (h (+ 1 2))))
> g
#⟨procedure⟩
```

We apply it to the function $\lambda w.\,\underline{v}\,\underline{@}\,w$, rather than the fresh symbol $\underline{v}$, as follows. (The operator $\underline{@}$ denotes dynamic function application.)

```
> (g (lambda (w) `(v ,w)))
(v 3)
```

[1] available at http://www.yl.is.s.u-tokyo.ac.jp/~sumii/pub/

Given the fresh symbol $\underline{v}$ η-expanded according to the type $\mathbf{int} \to \beta$, the function g returned the dynamic expression $\underline{v} \,\underline{@}\, 3$. Therefore, it is equivalent to the function $\lambda v.\, v\, 3$.

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 "impure" function (**lambda** (x) (**set!** x 123) x) of type $\alpha \to$ **int** to the pure function $\lambda x.\, 123$, 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 $f = \lambda x.\, 1 + (\mathbf{if}\ x\ \mathbf{then}\ 2\ \mathbf{else}\ 3)$ of type $\mathbf{bool} \to \mathbf{int}$.

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

We can reify this function by applying it to *both* **true** and **false**.

```
> (f #t)
3
> (f #f)
4
```

Since the function returned 3 and 4 for **true** and **false**, respectively, it is equivalent to $\lambda x.\, \mathbf{if}\ x\ \mathbf{then}\ 3\ \mathbf{else}\ 4$. 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 delimited continuations [2].

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 $\lambda\_.$ **display** $(1 + 2)$ of type $() \to ()$. Applying the function to a fresh symbol causes the output to be performed.

```
> (define f (lambda _ (display (+ 1 2))))
> (f 'z)
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 arg `(display ,@arg)))
> (set! display dynamic-display)
> (f 'z)
(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 $f = \lambda x.\, (1 + 2) + x$ of type $\mathbf{int} \to \mathbf{int}$.

```
> (define f (lambda (x) (+ (+ 1 2) x)))
```

Since there exists no value of type **int** that represents all integers, we cannot "η-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 old-+ +)
> (define (new-+ x y)
    (if (and (number? x) (number? y))
        (old-+ x y)
        `(+ ,x ,y)))
> (set! + new-+)
```

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

```
> (f 'z)
(+ 3 z)
```

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

```
> (define (alt-+ x y) `(+ ,x ,y))
> (set! + alt-+)
> (f 'z)
(+ (+ 1 2) z)
```

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

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 $\mathbf{fix}(\lambda p.\, \lambda n.\, \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ 2 \times p(n - 1))$ does not terminate, because its "normal form" $\lambda n.\, \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ (\mathbf{if}\ n - 1 = 0\ \mathbf{then}\ 2\ \mathbf{else}\ (\mathbf{if}\ n - 2 = 0\ \mathbf{then}\ 4\ \mathbf{else}\ \ldots))$ is infinite. A workaround for this problem is to use a special fixed-point operator which does not recurse during reification [3], though this spoils a merit of TDPE—that it does not need the source code of the program.

References

[1] Olivier Danvy. Pragmatic aspects of type-directed partial evaluation. PE '96, LNCS 1110.

[2] Olivier Danvy. Type-directed partial evaluation. POPL '96.

[3] Olivier Danvy. Online type-directed partial evaluation. FLOPS '98.

[4] Olivier Danvy. Type-directed partial evaluation. PE '98, LNCS 1706.

[5] Olivier Danvy and Andrzej Filinski. A functional abstraction of typed contexts. Technical Report 89/12, DIKU.

[6] Olivier Danvy and Andrzej Filinski. Abstracting control. LFP '90.

[7] Andrzej Filinski. Representing monads. POPL '94.

[8] Andrzej Filinski. Representing layered monads. POPL '99.

[9] John Hatcliff and Olivier Danvy. A computational formalization for partial evaluation. MSCS 7(5), 1997.

[10] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[11] Julia Lawall and Peter Thiemann. Sound specialization in the presence of computational effects. TACS '97, LNCS 1281.

[12] Tatsurou Sekiguchi, Hidehiko Masuhara, and Akinori Yonezawa. A simple extension of Java language for controllable transparent migration and its portable implementation. COORDINATION '99, LNCS 1594.

[13] James E. White. Telescript technology: An introduction to the language. White Paper, General Magic, 1995.