An Implementation of Transparent Migration on Standard Scheme

Eijiro Sumii
University of Tokyo
Idea

\[ g_{\text{host}} \cong \text{shift} \left( \text{reval}_{\text{host}} \circ \text{tdpe}(\text{ }) \rightarrow (\text{ }) \right) \]
Idea

\[ g_{\text{host}} \cong \text{shift} \left( \text{reval}_{\text{host}} \circ \text{tdpe}() \rightarrow() \right) \]

Delimited Continuation
+ Type-Directed Partial Evaluation
+ Remote Evaluation
→ Transparent Migration
Outline

- What is transparent migration?
- What are
  - Delimited continuation
  - Type-directed partial evaluation

and how do they enable transparent migration?
Transparent Migration (or "Strong Mobility")

A program moves from one host to another, keeping its execution state (cf. Telescript [White 95])
Transparent Migration (or "Strong Mobility")

A program moves from one host to another, keeping its execution state (cf. Telescript [White 95])

```lisp
> (begin (system "hostname")
  (go "remotehost")
  (system "hostname")

localhost remotehost
```
Naive Approach

(define (go rhost)
  (call/cc (λk. somehow send \texttt{k} to \texttt{rhost})\texttt{)))
Problem: Unnecessary Continuation

(let ([v (make-vector 100000)])
  (go "remotehost")
  (display "hello")
  (go "localhost")
  (display v))
Problem: Unnecessary Continuation

(let ([v (make-vector 100000)])
  (go "remotehost")
  (display "hello")
  (go "localhost")
  (display v))
Problem: Unnecessary Continuation

(let ([v (make-vector 100000)])
  (go "remotehost")
  (display "hello")
  (go "localhost")
  (display v))
Delimited Continuation
[Danvy & Filinski 89, 90]

The rest of the computation up to some point
Delimited Continuation
[Danvy & Filinski 89, 90]

The rest of the computation up to some point

\[ (+ 1 \ (\text{reset} \ (+ 2 \ (\text{shift} \ (\lambda k. \ (k \ (k \ 3))))))) \]
Delimited Continuation
[Danvy & Filinski 89, 90]

The rest of the computation up to some point

\[ (+ 1 \ (\text{reset} \ (+ 2 \ (\text{shift} \ (\lambda k. \ (k \ (k \ 3))))))) \]

\[ \Rightarrow \ (+ 1 \ (k \ (k \ 3))) \]

where \( k = (+ 2 \ \cdot) \)
Delimited Continuation
[Danvy & Filinski 89, 90]

The rest of the computation up to some point

\[(+ 1 (\text{reset} (+ 2 (\text{shift} (\lambda k. (k (k 3))))))) \mapsto (+ 1 (k (k 3))))\]

where \( k = (+ 2 \cdot) \)

\[(+ 1 (+ 2 (+ 2 3))) \mapsto 8\]
Transparent Migration Using Delimited Continuations

(let ([v (make-vector 100000)])
  (reset (go "remotehost")
    (display "hello"))
  (display v))
Transparent Migration Using Delimited Continuations

(let ([v (make-vector 100000)])
  (reset (go "remotehost")
    (display "hello"))
  (display v))

(define (go rhost)
  (shift (λ k.
      (somehow send k to rhost))))
Transparent Migration Using Delimited Continuations

(let ([v (make-vector 100000)])
  (reset (go "remotehost")
    (display "hello"))
  (display v))

(define (go rhost)
  (shift (l)
    (somehow send k to rhost)))
Type-Directed Partial Evaluation
[Danvy 96, 98]

Given a compiled value and its type, "reconstruct" its source code (in long $\beta\eta$-normal form)
Type-Directed Partial Evaluation
[Danvy 96, 98]

Given a compiled value and its type, "reconstruct" its source code (in long $\beta\eta$-normal form)

```scheme
> (define (f x)
    ((lambda (y) y) x))
> (tdpe $\alpha \rightarrow \alpha$ f)
(llambda (z0) z0)
```
Type-Directed Partial Evaluation
[Danvy 96, 98]

Residualizes "non-trivial" computations by \texttt{set!}-ing primitive operators to code generating functions
Type-Directed Partial Evaluation
[Danvy 96, 98]

Residualizes "non-trivial" computations by \texttt{set!}-ing primitive operators to code generating functions

\begin{verbatim}
> (define (g x)
      (display (+ x 1)))
> (tdpe 'int→() g)
(lambda (z1)
  (display (+ z1 1)))
\end{verbatim}
Transparent Migration
Using TDPE

(define (go rhost)
  (shift (λk.
    (let ([e (tdpe '()⇒() k)])
      (reval rhost e)))))
Limitations

- "go" doesn't terminate if "k" has no normal form (e.g. because of recursion)
  - Workaround: use a special fixed-point operator

- "go" duplicates some data
  - set!, set-car!, set-vector!, eq?, etc. may not work
Conclusion

$$g_{\text{rhost}} \cong \text{shift} \left( \text{reval}_{\text{rhost}} \circ \text{tdpe}(\rightarrow) \right)$$
Conclusion

\[ go_{\text{rhost}} \equiv \text{shift} \left( \text{reval}_{\text{rhost}} \circ \text{tdpe}() \rightarrow () \right) \]

Scheme is so flexible!

- call/cc + set! \Rightarrow \text{shift} & \text{reset}
- dynamic typing + set! \Rightarrow \text{TDPE} with ease