

An Implementation of Transparent Migration on Standard Scheme

Eijiro Sumii

University of Tokyo

Idea

$$\text{go}_{\text{rhost}} \cong \text{shift} (\text{reval}_{\text{rhost}} \circ \text{tdpe}_{() \rightarrow ()})$$

Idea

$$\text{go}_{\text{rhost}} \cong \text{shift} (\text{reval}_{\text{rhost}} \circ \text{tdpe}_{() \rightarrow ()})$$

Delimited Continuation

- + Type-Directed Partial Evaluation
- + Remote Evaluation
- Transparent Migration

Outline

- What is transparent migration?
- What are
 - Delimited continuation
 - Type-directed partial evaluationand 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])

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

localhost

remotehost

Naive Approach

```
(define (go rhost)
  (call/cc (lk.
    somehow send k to rhost)))
```

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 (reset (+ 2 (shift (lk.  
                        (k (k 3)))))))
```

Delimited Continuation

[Danvy & Filinski 89, 90]

The rest of the computation *up to* some point

```
(+ 1 (reset (+ 2 (shift (lk.  
                    (k (k 3)))))))
```

```
P (+ 1 (k (k 3)))
```

where $k = (+ 2 \bullet)$

Delimited Continuation

[Danvy & Filinski 89, 90]

The rest of the computation *up to* some point

```
(+ 1 (reset (+ 2 (shift (lk.  
                                (k (k 3))))))))
```

```
↳ (+ 1 (k (k 3)))
```

where $k = (+ 2 \bullet)$

```
↳ (+ 1 (+ 2 (+ 2 3)))
```

```
↳ 8
```

Transparent Migration Using Delimited Continuations

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

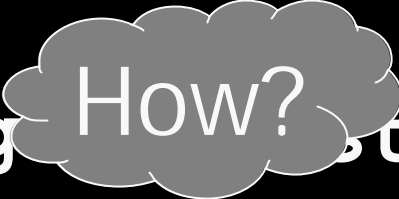
Transparent Migration Using Delimited Continuations

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

```
(define (go rhost)  
  (shift (lk.  
         somehow send k to rhost)))
```


Transparent Migration Using Delimited Continuations

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

```
(define (go  k st)  
  (shift (k.  
         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)

```
> (define (f x)
      ((lambda (y) y) x))
> (tdpe 'aⓂa f)
(lambda (z0) z0)
```

Type-Directed Partial Evaluation

[Danvy 96, 98]

Residualizes "non-trivial" computations
by **set!**-ing primitive operators
to code generating functions

Type-Directed Partial Evaluation

[Danvy 96, 98]

Residualizes "non-trivial" computations
by **set!**-ing primitive operators
to code generating functions

```
> (define (g x)
      (display (+ x 1)))
> (tdpe 'int® () g)
(lambda (z1)
  (display (+ z1 1)))
```

Transparent Migration Using TDPE

```
(define (go rhost)
  (shift (lk.
    (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

$$\text{go}_{\text{rhost}} \cong \text{shift} (\text{reval}_{\text{rhost}} \circ \text{tdpe}_{() \rightarrow ()})$$

Conclusion

$$\text{go}_{\text{rhost}} \cong \text{shift} (\text{reval}_{\text{rhost}} \circ \text{tdpe}_{() \rightarrow ()})$$

Scheme is so flexible!

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