Parametric Polymorphism through run-time sealing

(Theorems for low, low prices!)

Ben Greenman
September 22, 2014
Is there a "Best" Programming Language?
Of course not!

We live in a multi-language world
Interoperability?
Interoperability?
Scheme  

ML
Question

• If we mix Scheme code and ML code, what static guarantees still apply?

• Can we prove parametricity after mixing typed and untyped code?
Parametricity

• Parametricity is about invariances.

• A function is parametric if it has a "uniformly given algorithm on all types".
Parametricity

\[ \text{fst} = \lambda(x,y) . x \]
Parametricity

\[ \forall \alpha . \alpha \times \alpha \to \alpha \]

\[ \text{fst} = \lambda (x,y) . x \]
Parametricity

\[
\forall \alpha . \alpha * \alpha \rightarrow \alpha
\]

\[
fst = \lambda (x,y) . x
\]

\[
snd = \lambda (x,y) . y
\]
Parametricity

(1) Strachey [1967]: parametric vs. ad-hoc
(2) Reynolds [1983]: abstraction theorem
(3) Bainbridge, Freyd et. al [1988]: "parametricity"
(4) Wadler [1989]: free theorems

(1) "Fundamental Concepts in Programming Languages"
(2) "Types, Abstraction, and Parametric Polymorphism"
(3) "Functorial Polymorphism" & "Semantic Parametricity in Polymorphic Lambda Calculus"
(4) "Theorems for Free!"
Free Theorems

For all functions $f : \forall \alpha . \alpha \text{ list} \to \alpha \text{ list}$

And any function $g : \beta \to \gamma$

$$(\text{map } g) \circ f = f \circ (\text{map } g)$$
Free Theorems

• Why does this work?
• All values of type \( \alpha \) are black boxes to \( f \).
Free Theorems

• In System F, Haskell, ML, etc., parametricity is guaranteed statically.

• The type of a polymorphic function expresses the invariants it preserves.

What about Scheme?
Theorems for Scheme?

;;; fst : A * A -> A
(define (fst (a,b))
a)

;;; snd : A * A -> A
(define (snd (a,b))
b)
Theorems for Scheme?

;; fst2 : A * A -> A
(define (fst2 (a,b))
  (if (and (int? a)
           (= a 17))
      b
      a))

"Almost always" well-behaved is NOT good enough!
Theorems for Scheme?

;;; min : A * A -> A
(define (min (a,b))
  (cond [(and (int? a) (int? b))
         (if (< a b) a b)]
        [(and (str? a) (str? b))
         (if (<-str a b) a b)]
        ...
        [else (error)]))
Theorems for Scheme?

;; min : A × A → A
(define (min a b))
  (cond
    [(and (int? a) (int? b))
      (if (< a b) a b)]
    [(and (str? a) (str? b))
      (if (<-str a b) a b)]
    ...]
  [else (error)])
What happened?

;;; fst2 : A * A -> A
(define (fst2 (a,b))
  (if (and (int? a)
           (= a 42))
      b
      a))

Programs should NOT be able to inspect a value with an abstract type.
Dynamic Seals

- Morris [1973]: "Types are not Sets".
- Protect exported values with secret keys.
- Crash if a sealed value is used.

\[
\text{(define (seal v s1)} \text{ (λs2 . (if (eq? s1 s2) v (error)))})
\]
Can we combine static type checking and dynamic seals?
Dynamic Seals

Polymorphism

YES!
Paper Outline

I. Define the languages + conversion rules
II. Prove type safety + parametricity
III. Demonstrate applications
The Languages

| v = (\lambda x. e) | n | nil |
| | (cons v v) | fst | rst |
| e = v | (e e) | x | (op e e) |
| | (if0 e e e) | (cons e e) |
| | (pd e) |

| v = \lambda x: \tau. e | n | nil |
| | cons v v | fst | rst |
| e = v | (e e) | x | op e e |
| | if0 e e e | cons e e |

\[ \tau = \text{Nat} | \tau \rightarrow \tau | \tau^* \]
The Languages

\[ v = (\lambda x. e) | n | \text{nil} \]
\[ | (\text{cons } v \ v) | \text{fst} | \text{rst} \]

\[ e = v | (e \ e) | x | (op \ e \ e) \]
\[ | (\text{if}0 \ e \ e \ e) | (\text{cons } e \ e) \]
\[ | (pd \ e) | (\text{SM}^\tau e) \]

\[ \tau = \text{Nat} | \tau \rightarrow \tau | \tau^* \]
<table>
<thead>
<tr>
<th>The Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>((\text{SM}^\tau e))</td>
</tr>
<tr>
<td>Convert the ML expression (e) with type (\tau) into a Scheme expression</td>
</tr>
</tbody>
</table>
### The Languages

<table>
<thead>
<tr>
<th>Types</th>
<th>Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v = (\lambda x. e) \mid n \mid nil$</td>
<td>$v = \lambda x: \tau. e \mid n \mid nil$</td>
</tr>
<tr>
<td></td>
<td>$\mid (\text{cons } v ; v) \mid \text{fst} \mid \text{rst}$</td>
</tr>
<tr>
<td>$e = v \mid (e ; e) \mid x \mid (op ; e ; e)$</td>
<td>$e = v \mid (e ; e) \mid x \mid op ; e ; e$</td>
</tr>
<tr>
<td></td>
<td>$\mid (\text{if0 } e ; e ; e ; e) \mid (\text{cons } e ; e)$</td>
</tr>
<tr>
<td></td>
<td>$\mid (p!d!e) \mid (S.M^\tau e)$</td>
</tr>
<tr>
<td>$\tau = \text{Nat} \mid \tau \rightarrow \tau \mid \tau^*$</td>
<td></td>
</tr>
</tbody>
</table>
Conversion Example

\[
\tau \rightarrow \tau' \quad \text{MS} \ (\lambda x. \ e) \\
\downarrow \\
\lambda x: \tau. \ \text{MS}^{\tau'}((\lambda x. \ e) \ \text{SM}^{\tau} x)
\]
The Languages

| $v = (\lambda x. e) | n | nil$ | $v = \lambda x: \tau. e | n | nil$ |
| --- | --- | --- |
| $(\text{cons } v v) | \text{fst} | \text{rst}$ | $\text{cons } v v | \text{fst} | \text{rst}$ |

| $e = v | (\text{e e}) | x | (\text{op } e e)$ | $e = v | (\text{e e}) | x | \text{op } e e$ |
| --- | --- | --- | --- |
| $(\text{if0 } e e e) | (\text{cons } e e)$ | $(\text{if0 } e e e) | \text{cons } e e$ |
| $(\text{pd } e) | (\text{SM}^\tau e)$ | $\tau ≈ \text{Nat} | \tau \rightarrow \tau | \tau^*$ |
### The Languages

<table>
<thead>
<tr>
<th>v = (\lambda x. e)</th>
<th>n</th>
<th>nil</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(cons v v)</td>
<td>fst</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>e = v</th>
<th>(e e)</th>
<th>x</th>
<th>(op e e)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(if0 e e e)</td>
<td>(cons e e)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(pd e)</td>
<td>(SM^\tau e)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>v = \lambda x:\tau. e</th>
<th>n</th>
<th>nil</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cons v v</td>
<td>fst</td>
</tr>
<tr>
<td></td>
<td>^(1)MS v</td>
<td>\Lambda \tau. e</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>e = v</th>
<th>(e e)</th>
<th>x</th>
<th>op e e</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>if0 e e e</td>
<td>cons e e</td>
<td></td>
</tr>
<tr>
<td></td>
<td>^\tau MS e</td>
<td>e^\langle \tau \rangle</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>\tau = Nat</th>
<th>\tau \rightarrow \tau</th>
<th>\tau^*</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\forall \alpha. \tau</td>
<td>\alpha</td>
</tr>
</tbody>
</table>
Conversion Summary

- Seals are introduced at Scheme/ML boundaries, to protect type variables.
- \( \langle \alpha, \tau \rangle \) is a seal on the variable \( \alpha \), which should have type \( \tau \) back in ML.
- Conversion strategies \( \kappa \) are types that might contain seals.
- Type substitutions \( \eta \) map type variables to closed types.
Parametricity / Fundamental Theorem

For all seal-free terms $e$ and $e'$, type environments $\Delta$ and value environments $\Gamma$:

1. If $\Delta;\Gamma \vdash e : \tau$ then $\Delta;\Gamma \vdash e \approx e : \tau$
2. If $\Delta;\Gamma \vdash e : L$ then $\Delta;\Gamma \vdash e \approx e : L$

"If type checking proves that $e$ has type $\tau$, then our logical relation will prove that $e$ is parametric at type $\tau."
Proof Strategy: Logical Relations

- Syntactically relate terms in each language.
- $e \preceq e' : \tau$ means that a machine running $e$ will behave no differently from a machine running $e'$.
- $(\preceq)$ only relates terms that are parametric at type $\tau$. 
Defining $\equiv$

- The paper defines two logical relations, one for Scheme and one for ML.
- The relation for ML is straightforward.
  - Identical values are related.
  - Related functions map related inputs to related outputs.
- Scheme is trickier...
Nontermination

(define omega
  ((λx . x x) (λx . x x)))

"You can't do that in your typed languages!" -RBF
Solution: Step-Indexing

• Although all ML programs terminate, Scheme programs may fail or loop forever.

• Step-indexed logical relations guarantee some number of computational steps.

• \( \leq^k \) relates terms for up to \( k \) steps.

Ahmed [2006] "Step-indexed syntactic logical relations for recursive and quantified types".
Identical natural numbers are related for $k$ steps. No problem!
Defining \( \equiv \)

\[
\begin{align*}
\delta \vdash n & \equiv^k n : \text{Nat} \\
\delta \vdash v & \equiv^k v' : \alpha \\
Unconditionally & (k, v, v') \in \delta(\alpha)
\end{align*}
\]

Two values are related at type \( \alpha \) if the type relation \( \delta \) says so.
Defining \( \leq \)

\[
\begin{align*}
\delta \vdash n \leq^k n : \text{Nat} & \quad \text{Unconditionally} \\
\delta \vdash v \leq^k v' : \alpha & \quad (k, v, v') \in \delta(\alpha) \\
\delta \vdash [...] v_n \leq^k [...] v_n' : \tau^* & \quad \forall j<k. \forall i \leq n. \\
& \quad \delta \vdash v_i \leq^i v_i' : \tau
\end{align*}
\]

Two lists are related for \( k \) steps if you can't tell apart any pair of elements within \( j<k \) steps.
Two functions are related for $k$ steps if, given arguments related for $j < k$ steps, the outputs are related for $j$ steps.
Defining \( \leq \):

\[
\delta \vdash e \leq^k e' : \tau
\]

\[
\forall j < k. \quad\quad (e \rightarrow^j \text{error} \Rightarrow e' \rightarrow^{*\text{error}}) \quad\quad \land
\]

\[
(\forall v. e \rightarrow^j v \Rightarrow \exists v'. e' \rightarrow^{*v'} \land \delta \vdash v \leq^{k-j} v' : \tau)
\]

Two expressions are related for \( k \) steps if they both explode or both step to related values.
Bridge Lemma

For all $k \geq 0$ and type relations $\delta$:

1. If $\vdash e \preceq^k e' : \tau$ then $\vdash SM e \preceq^{\tau/\delta}^k SM e' : L$

2. If $\vdash e \preceq^k e' : L$ then $\vdash MS e \preceq^{\tau/\delta}^k MS e' : \tau$

"Sealing respects the logical relation."
Parametricity!

- If a term has type $\forall \alpha . \alpha \ast \alpha \rightarrow \alpha$
- Then it is $\text{fst} = \lambda(x,y) . x$
  or it is $\text{snd} = \lambda(x,y) . y$
- (Or it always raises an error)
- (Or it always diverges)
Application: Contracts

• Contracts are like types, but stronger.
• Dynamically check invariants.
• Using seals, we can give an ML type to a Scheme term as its behavioral specification.
• Bridge Lemma gives a simple implementation of higher-order, polymorphic contracts.

Bottom Line

• Proved parametricity in a (simple) multi-language setting.

• Keep the clean abstractions of ML by adding a little enforcement to Scheme.

• Step-indexing handles non-termination (and recursive types).

• One step towards language interoperability.
The End