The Static Pattern Calculus

Term Reduction

The Book

- Barry Jay “Pattern Calculus: Computing with Functions and Structures”, Springer, 2009.
- Part 1: Terms
- Part 2: Types
- Part 3: Bondi programming language

Static Pattern Calculus Syntax

<u>patterns</u>

p ::=

x	(matchable symbol)
c	(constructor)
p p	(application)

<u>terms</u>

t ::=

x	(variable)
c	(constructor)
t t	(application)
$p \to t$	(case)

Static Pattern Calculus Examples

- (x → x) A ⇛ A
- (Pair x y → x) (Pair A B) ⇛ A
- (Pair x y → Pair y x) (Pair A B) ⇛ Pair B A
- (Pair x y → x) A ⇛ NoMatch
- (x y → x) (A B) ⇛ A
- (x y → y x) (A B) ⇛ B A
- (x y z → y) (A B C) ⇛ B

Static Pattern Calculus Examples

- $(x\ y \rightarrow y)\ (A\ B\ C) \quad \Rrightarrow \quad C$
- $(x\ y \rightarrow y)\ ((x \rightarrow x)\ (A\ B)) \quad \Rrightarrow \quad B$
- $(x\ y \rightarrow y)\ ((x \rightarrow x)\ A) \quad \Rrightarrow \quad \text{NoMatch}$
- $(x\ x \rightarrow x)\ (A\ A) \quad \Rrightarrow \quad \text{NoMatch}$

Linear patterns

- What happens if a pattern has more than one occurrence of the same matchable symbol? e.g.

  (Pair x x → x) (Pair U U)

- Hard to determine equality if U is a case (a term denoting a function).

- “nonlinearity can break confluence of reduction” (page 35) (how so?)

- For data structures, equality can be defined in other ways.

Linear patterns

- For the reasons on previous slide, non-linear patterns are deemed to always fail.

- Why not just make them a syntax error (as they are in Haskell)?

    - In the *Dynamic* Pattern Calculus, patterns can be computed.

    - In that context, syntactic checks for linearity will (unjustly) prohibit programs where non-linear patterns reduce to linear ones.

    - So the approach of the Static Calculus is motivated by future issues in the Dynamic Calculus.

Substitutions

- $\sigma = \{ u_1 / x_1 \dots u_n / x_n \}$
- Partial function from symbols to terms.
- $\{ u / x \}$ reads as "replace u for x"
- Usual rules apply regarding avoiding variable capture, e.g.

$$(x \rightarrow (y \rightarrow x\ y))\ y \quad \Rrightarrow \quad y' \rightarrow y\ y'$$

Static matching

$\langle\!\langle t, p \rangle\!\rangle$ reads "match term t with pattern p" (book uses different brackets)

$\langle\!\langle u, x \rangle\!\rangle$ = some $\{u / x\}$

$\langle\!\langle c, c \rangle\!\rangle$ = some { }

$\langle\!\langle u\ v, p\ q \rangle\!\rangle$ = $\{u / p\} \uplus \{v / q\}$, if $u\ v$ is a compound

$\langle\!\langle u, p \rangle\!\rangle$ = none, otherwise, if u is matchable

$\langle\!\langle u, p \rangle\!\rangle$ = undefined, otherwise

Static matching

disjoint union
ensures linearity

$\langle\!\langle u, x \rangle\!\rangle$ = some $\{u / x\}$

$\langle\!\langle c, c \rangle\!\rangle$ = some $\{\ \}$

$\langle\!\langle u\ v, p\ q \rangle\!\rangle$ = $\{u / p\} \uplus \{v / q\}$, if $u\ v$ is a compound

$\langle\!\langle u, p \rangle\!\rangle$ = none, otherwise, if u is matchable

$\langle\!\langle u, p \rangle\!\rangle$ = undefined, otherwise

Static matching

ensure term is not a redex

$\langle\!\langle u, x \rangle\!\rangle$ = some $\{u / x\}$

$\langle\!\langle c, c \rangle\!\rangle$ = some $\{\ \}$

$\langle\!\langle u\ v, p\ q \rangle\!\rangle$ = $\{u / p\} \uplus \{v / q\}$, if $u\ v$ is a compound

$\langle\!\langle u, p \rangle\!\rangle$ = none, otherwise, if u is matchable

$\langle\!\langle u, p \rangle\!\rangle$ = undefined, otherwise

Matchable, data structure, compound

<u>compound</u>

k ::= d t (data structure applied to term)

<u>data structure</u>

d ::=
- c (constructor)
- k (compound)

<u>matchable</u>

m ::=
- p → t (case)
- d (data structure)

Reduction

```
reduce :: Term -> Reduce Term

reduce (Application t1 t2) = do
   reduct1 <- reduce t1
   case reduct1 of
      Case pattern body ->
         case match pattern t2 of
            None -> return noMatch
            Some subst -> do
               newBody <- applySubst subst body
               reduce newBody
            Undefined -> do
               reduct2 <- reduce t2
               reduce $ Application reduct1 reduct2
      other -> return $ Application reduct1 t2

reduce other = return other
```

Reduction

```
reduce :: Term -> Reduce Term

reduce (Application t1 t2) = do
    reduct1 <- reduce t1
    case reduct1 of
        Case pattern body ->
            case match pattern t2 of
                None -> return noMatch
                Some subst -> do
                    newBody <- applySubst subst body
                    reduce newBody
                Undefined -> do
                    reduct2 <- reduce t2
                    reduce $ Application reduct1 reduct2
        other -> return $ Application reduct1 t2

reduce other = return other
```

apply the static matching rule from before

Reduction

matching failed

```
reduce :: Term -> Reduce Term

reduce (Application t1 t2) = do
    reduct1 <- reduce t1
    case reduct1 of
        Case pattern body ->
            case match pattern t2 of
                None -> return noMatch
                Some subst -> do
                    newBody <- applySubst subst body
                    reduce newBody
                Undefined -> do
                    reduct2 <- reduce t2
                    reduce $ Application reduct1 reduct2
        other -> return $ Application reduct1 t2

reduce other = return other
```

Reduction

```
reduce :: Term -> Reduce Term

reduce (Application t1 t2) = do
    reduct1 <- reduce t1
    case reduct1 of
        Case pattern body ->
            case match pattern t2 of
                None -> return noMatch
                Some subst -> do
                    newBody <- applySubst subst body
                    reduce newBody
                Undefined -> do
                    reduct2 <- reduce t2
                    reduce $ Application reduct1 reduct2
        other -> return $ Application reduct1 t2

reduce other = return other
```

matching succeeded

Reduction

```
reduce :: Term -> Reduce Term

reduce (Application t1 t2) = do
    reduct1 <- reduce t1
    case reduct1 of
        Case pattern body ->
            case match pattern t2 of
                None -> return noMatch
                Some subst -> do
                    newBody <- applySubst subst body
                    reduce newBody
                Undefined -> do
                    reduct2 <- reduce t2
                    reduce $ Application reduct1 reduct2
        other -> return $ Application reduct1 t2

reduce other = return other
```

argument needed to be reduced, try matching again

Theorems

- Reduction is confluent (if a term has a normal form, then it is unique).
- Reduction cannot get stuck (every term of the form ‘(p → t) u’ is reducible).

Fixed points

Ω = Rec x → x (Rec x)

FIX = f → Ω (Rec (x → f (Ω x)))

(you could also define FIX as it is done in the Lambda Calculus, but on page 37 there is promise of a type for Rec in part 2 of the book).

Extensions

- Sequence of cases:

$$p_1 \to t_1 \mid p_2 \to t_2 \mid \ldots \mid p_n \to t_n$$

- For example:

$$\text{Pair } x\ y \to x \mid \text{Triple } x\ y\ z \to x \mid \text{Quad } w\ x\ y\ z \to w$$

- Reduction:

$$(\, p \to s \mid r\,)\ u \quad\longrightarrow\quad \sigma\, s \qquad \text{if } \langle\!\langle u, p \rangle\!\rangle = \text{some } \sigma$$

$$(\, p \to s \mid r\,)\ u \quad\longrightarrow\quad r\ u \qquad \text{if } \langle\!\langle u, p \rangle\!\rangle = \text{none}$$

Generic programming

fold f g = x y → f (fold f g x) (fold f g y) | x → g x

size = fold plus (x → 1)

(assuming some sugar for named (possibly recursive) definitions).