

# 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

---

## patterns

$p ::=$

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

## terms

$t ::=$

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

# Static Pattern Calculus Examples

---

- $(x \rightarrow x) A \Rightarrow A$
- $(\text{Pair } x \ y \rightarrow x) (\text{Pair } A \ B) \Rightarrow A$
- $(\text{Pair } x \ y \rightarrow \text{Pair } y \ x) (\text{Pair } A \ B) \Rightarrow \text{Pair } B \ A$
- $(\text{Pair } x \ y \rightarrow x) A \Rightarrow \text{NoMatch}$
- $(x \ y \rightarrow x) (A \ B) \Rightarrow A$
- $(x \ y \rightarrow y \ x) (A \ B) \Rightarrow B \ A$
- $(x \ y \ z \rightarrow y) (A \ B \ C) \Rightarrow B$

# Static Pattern Calculus Examples

---

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

# Linear patterns

---

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

$(\text{Pair } x \ x \ \rightarrow \ x) (\text{Pair } U \ U)$

- 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.

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



# Static matching

---

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

---

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

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

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

$\langle\langle u, p \rangle\rangle = \text{none},$       otherwise, if  $u$  is matchable

$\langle\langle u, p \rangle\rangle = \text{undefined},$       otherwise

# Static matching

---

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

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

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

$\langle\langle u, p \rangle\rangle = \text{none},$  otherwise, if  $u$  is matchable

$\langle\langle u, p \rangle\rangle = \text{undefined},$  otherwise

disjoint union  
ensures linearity



# Static matching

---

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

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

$\langle\langle u v, p q \rangle\rangle = \{u / p\} \uplus \{v / q\},$

$\langle\langle u, p \rangle\rangle = \text{none},$

$\langle\langle u, p \rangle\rangle = \text{undefined},$

ensure term is not  
a redex

if  $u v$  is a compound

otherwise, if  $u$  is matchable

otherwise

# Matchable, data structure, compound

---

## compound

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

## data structure

$d ::=$

$c$             (constructor)

$k$             (compound)

## matchable

$m ::=$

$p \rightarrow t$     (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

---

```
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 failed



# 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
```

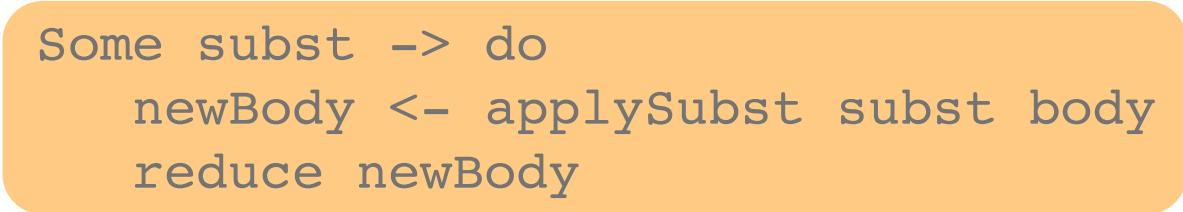
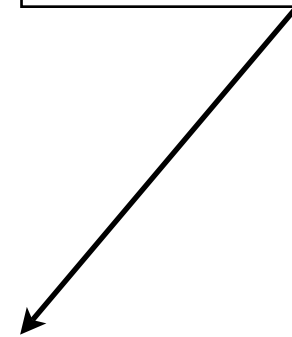
```
          reduct2 <- reduce t2
```

```
          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  $(p \rightarrow t) u$  is reducible).

# Fixed points

---

$$\Omega = \text{Rec } x \rightarrow x (\text{Rec } x)$$

$$\text{FIX} = f \rightarrow \Omega (\text{Rec } (x \rightarrow f (\Omega x)))$$

(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:

$$p_1 \rightarrow t_1 \mid p_2 \rightarrow t_2 \mid \dots \mid p_n \rightarrow t_n$$

- For example:

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

- Reduction:

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

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

# Generic programming

---

$\text{fold } f \ g = x \ y \rightarrow f \ (\text{fold } f \ g \ x) \ (\text{fold } f \ g \ y) \mid x \rightarrow g \ x$

$\text{size} = \text{fold plus } (x \rightarrow 1)$

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