Lambda-Calculus Confluence

Matthew Doty

Outline

Why Confluence?

Why Confluence? — Stuck Macros I

```
pullMaybe :: Q Exp
pullMaybe = getExpectedType >>= \case
  Arr (Maybe _) _ -> [| id |]
  _               -> [| traverse $pullMaybe |]
```

Why Confluence? — Stuck Macros II

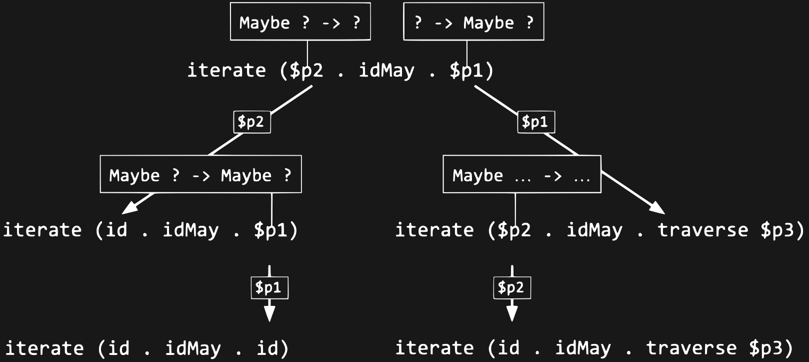
```
pullMaybe0 ::   Maybe a     -> Maybe     a
pullMaybe0 = $pullMaybe

pullMaybe1 ::  [Maybe a]    -> Maybe    [a]
pullMaybe1 = $pullMaybe

pullMaybe2 :: [[Maybe a]]   -> Maybe   [[a]]
pullMaybe2 = $pullMaybe
```

Why Confluence? —
Naïve Typed Macros Aren't Confluent

```
p1 = p2 = p3 = pullMaybe
iterate :: (a -> a) -> (a -> a)
  idMay :: Maybe a -> Maybe a
```



Maybe ? -> ?
? -> Maybe ?
iterate ($p2 . idMay . $p1)
$p2
$p1
Maybe ? -> Maybe ?
Maybe … -> …
iterate (id . idMay . $p1)
iterate ($p2 . idMay . traverse $p3)
$p1
$p2
iterate (id . idMay . id)
iterate (id . idMay . traverse $p3)

Untyped Lambda Calculus

Untyped Lambda Calculus — Syntax I

```haskell
module LambdaCalc where
import Prelude hiding (foldl, foldl')

infixl 7 :.

data Lam
  = V Int
  | Lam :. Lam
  | Abs Int Lam
  deriving Eq
```

Untyped Lambda Calculus — Syntax II

Conventional	Haskell
$\lambda x.\ x$	`Abs 1 (V 1)`
$\lambda z.\ z$	`Abs 2 (V 2)`
$\lambda x.\lambda y.\ x$	`Abs 1 (Abs 2 (V 1))`
$(\lambda x.\lambda x.\ x)\ (\lambda y.\ y)$	`Abs 1 (Abs 1 (V 1)) :. Abs 2 (V 2)`

Untyped Lambda Calculus — Variable Capture I

```
freeVars :: Lam → [Int]

freeVars (V x) = [x]

freeVars (lam1 :. lam2) =
  freeVars lam1 <> freeVars lam2

freeVars (Abs x lam) =
  [y | y ← freeVars lam, y /= x]
```

Untyped Lambda Calculus — Variable Capture II

```haskell
freshVar :: Lam → Int

freshVar lam = 1 + foldr max 0 (freeVars lam)
```

Untyped Lambda Calculus — Substitution I

A single variable substitution:

```
infix 6 :=

data Subst = Int := Lam
```

Substitution operation:

```
infixl 5 !

(!) :: Lam → Subst → Lam
```

Untyped Lambda Calculus — Substitution II

Conventional LaTeX

$$s\,[x \;:=\; t]$$

Our Haskell Implementation

```
s ! x := t
```

Untyped Lambda Calculus — Substitution III

```
V y ! x := t
  | x == y = t
  | otherwise = V y
```

Untyped Lambda Calculus — Substitution IV

```
lam1 :. lam2 ! x := t =
  (lam1 ! x := t) :. (lam2 ! x := t)
```

Untyped Lambda Calculus — Substitution V

```haskell
s@(Abs y lam) ! x := t
  | x == y = Abs y lam
  | x /= y && not (y `elem` freeVars t) =
    Abs y (lam ! x := t)
  | otherwise =
    Abs z (lam ! y := V z ! x := t)
  where
    z = freshVar (s :. t :. V x)
```

Untyped Lambda Calculus — Beta Rule

Conventional LaTeX

$$(\lambda x.\, s)\ t\ \to_\beta\ s\,[x\ :=\ t]$$

Haskell implementation

$$\texttt{(Abs x s) :. t} \to_\beta \texttt{s ! x := t}$$

Untyped Lambda Calculus — Evaluation Strategies I

```
normEval :: Lam → Lam

normEval ((Abs x s) :. t) =
  normEval (s ! x := t)

---
normEval (V x) = V x

normEval (s :. t) =
  (normEval s) :. (normEval t)

normEval (Abs x s) = Abs x (normEval s)
```

Untyped Lambda Calculus — Evaluation Strategies II

```
callByVal :: Lam → Lam

-- normEval ((Abs x s) :. t) =
--   normEval (s ! x := t)

callByVal ((Abs x s) :. t) =
  callByVal (s ! x := callByVal t)

---
callByVal (V x) = V x

callByVal (s :. t) =
  (callByVal s) :. (callByVal t)

callByVal (Abs x s) = Abs x (callByVal s)
```

Untyped Lambda Calculus — Evaluation Strategies III

```
appEval :: Lam → Lam

-- callByVal ((Abs x s) :. t) =
--   callByVal (s ! x := callByVal t)

appEval ((Abs x s) :. t) =
  appEval (appEval s ! x := t)

---
appEval (V x) = V x

appEval (s :. t) =
  (appEval s) :. (appEval t)

appEval (Abs x s) = Abs x (appEval s)
```

Untyped Lambda Calculus — Evaluation Strategies IV

```
lazyEval :: Lam → Lam

-- normEval (Abs x s) = Abs x (normEval s)
lazyEval (Abs x s) = Abs x s

---
lazyEval ((Abs x s) :. t) =
  lazyEval (s ! x := t)

lazyEval (V x) = V x

lazyEval (s :. t) =
  (lazyEval s) :. (lazyEval t)
```

Untyped Lambda Calculus — Digression: Weak Head Normal Form I

```haskell
weakHeadNF :: Lam → Bool

weakHeadNF (V _) = True

weakHeadNF ((Abs _ _) :. _) = False

weakHeadNF (s :. t) =
  weakHeadNF s && weakHeadNF t

weakHeadNF (Abs _ _) = True
```

Untyped Lambda Calculus — Digression: Weak Head Normal Form II

Claim

if

```
weakHeadNF x
```

then

```
lazyEval x == x
```

Untyped Lambda Calculus — Digression: Weak Head Normal Form III

```haskell
foldl :: (a → b → a) → a → [b] → a
foldl _f a [] = a
foldl f a (x:xs) = foldl f (f a x) xs
```

Untyped Lambda Calculus — Digression: Weak Head Normal Form IV

```
foldl (+) 0 [1, 2, 3, 4]
  = foldl (0 + 1) [2, 3, 4]
  = foldl ((0 + 1) + 2) [3, 4]
  = foldl (((0 + 1) + 2) + 3) [4]
  = foldl ((((0 + 1) + 2) + 3) + 4) []
  = ((((0 + 1) + 2) + 3) + 4)
  = (((1 + 2) + 3) + 4)
  = ((3 + 3) + 4)
  = 6 + 4
  = 10
```

Untyped Lambda Calculus — Digression: Weak Head Normal Form V

Haskell has a special primitive **seq** :: a → b → b

When Haskell tries to evaluate

```
x `seq` y
```

Haskell will first evaluate x to weak head normal form, and then evaluate y to weak head normal form

Untyped Lambda Calculus — Digression: Weak Head Normal Form VI

```haskell
foldl' :: (a → b → a) → a → [b] → a
foldl' f a [] = a
foldl' f a (x : xs) =
  let a' = f a x
   in a' `seq` foldl' f a' xs
```

Untyped Lambda Calculus — Digression: Weak Head Normal Form VII

```
foldl' (+) 0 [1, 2, 3, 4]
  = foldl' (+) 0 [1, 2, 3, 4]
  = let a' = 0 + 1
    in a' `seq` foldl' (+) a' [2, 3, 4]
  = 1 `seq` foldl' (+) 1 [2, 3, 4]
  = foldl' (+) 1 [2, 3, 4]
```

Untyped Lambda Calculus — Digression: Weak Head Normal Form VIII

```
foldl' (+) 0 [1, 2, 3, 4]
  = foldl' (+) 1 [2, 3, 4]
  = foldl' (+) 3 [3, 4]
  = foldl' (+) 6 [4]
  = foldl' (+) 10 []
  = 10
```

Untyped Lambda Calculus — Normal Form I

```haskell
nf :: Lam → Bool

-- weakHeadNF (Abs _ _) = True
nf (Abs _ s) = nf s

---
nf (V _) = True

nf ((Abs _ _) :. _) = False

nf (s :. t) = nf s && nf t
```

Untyped Lambda Calculus — Normal Form II

Claim

If `nf x` *then*

- `normEval x == x`
- `callByVal x == x`
- `appEval x == x`

Church’s Arithmetic

Church's Arithmetic — Church Numerals

Number	Church Numeral
0	$\lambda f.\lambda x.\ x$
1	$\lambda f.\lambda x.\ f\,x$
2	$\lambda f.\lambda x.\ f\,(f\,x)$
3	$\lambda f.\lambda x.\ f\,(f\,(f\,x))$
$\vdots$	$\vdots$
n	$\lambda f.\lambda x.\ f^{\circ n}\ x$

Church's Arithmetic — Arithmetical Operations I

$$(+) \quad \equiv \quad \lambda m.\lambda n.\lambda f.\lambda x.\ m\ f\ (n\ f\ x)$$

...since $f^{\circ(m+n)}x = f^{\circ m}\ (f^{\circ n}\ x)$

Church's Arithmetic — Arithmetical Operations II

$$\begin{aligned} \mathit{succ} &\equiv \lambda n.\lambda f.\lambda x.\ f\,(n\ f\ x) \\ &\approx_\beta (+1) \end{aligned}$$

Church's Arithmetic — Arithmetical Operations III

$$(*) \quad \equiv \quad \lambda m.\lambda n.\lambda f.\lambda x.\ m\ (n\ f)\ x$$

...since $f^{\circ(m*n)}\ x = (f^{\circ n})^{\circ m}\ x$

Church's Arithmetic — Is Church's Arithmetic Consistent?

Can we make an arithmetical expression e which evaluates to 1 *using one strategy, but* 2 *using some other strategy?*

(kind of a "philosophy of math" version of Sam G's problem)

Formalization: de Bruijn Notation

Formalization: de Bruijn Notation — Syntax I

Conventional	de Bruijn
$\lambda x.\ x$	$\lambda\ 0$
$\lambda z.\ z$	$\lambda\ 0$
$\lambda x.\lambda y.\ x$	$\lambda\lambda\ 1$
$(\lambda x.\lambda x.\ x)\ (\lambda y.\ y)$	$(\lambda\lambda\ 0)\ (\lambda\ 0)$
$(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$	$(\lambda\lambda\ 0\ 0)\ (\lambda\lambda\ 0\ 0)$
$\lambda x.\lambda y.\lambda s.\lambda z.\ x\ s\ (y\ s\ z)$	$\lambda\lambda\lambda\lambda\ 3\ 1\ (2\ 1\ 0)$

Formalization: de Bruijn Notation — Syntax II

Number	Church Numeral	de Bruijn
0	$\lambda f.\lambda x.\ x$	$\lambda\lambda\ 0$
1	$\lambda f.\lambda x.\ f\,x$	$\lambda\lambda\ 1\ 0$
2	$\lambda f.\lambda x.\ f\,(f\,x)$	$\lambda\lambda\ 1\ (1\ 0)$
3	$\lambda f.\lambda x.\ f\,(f\,(f\,x))$	$\lambda\lambda\ 1\ (1\ (1\ 0))$

Formalization: de Bruijn Notation — Syntax III

```
datatype dB =
    Variable nat ("⟨_⟩" [100] 100)
  | Application dB dB (infixl "·" 200)
  | Abstraction dB ("λ")
```

Formalization: de Bruijn Notation — Lifting

```
primrec
  lift :: "dB ⇒ nat ⇒ dB"
where
    "lift (⟨i⟩) k = (if i < k then ⟨i⟩ else ⟨(i + 1)⟩)"
  | "lift (s · t) k = lift s k · lift t k"
  | "lift (λ s) k = λ (lift s (k + 1))"
```

Formalization: de Bruijn Notation — Substitution

```
primrec
  subst :: "dB ⇒ nat ⇒ dB ⇒ dB"
  ("_[_ '↦ _]" [300, 0, 0] 300)
  where
    subst_Var: "(⟨i⟩)[k ↦ s] =
      (if k < i then ⟨(i - 1)⟩ else if i = k then s else ⟨i⟩)"
  | subst_App: "(t · u)[k ↦ s] = t[k ↦ s] · u[k ↦ s]"
  | subst_Abs: "(λ t)[k ↦ s] = λ (t [k + 1 ↦ lift s 0])"
```

Formalization: de Bruijn Notation — Beta Reduction I

$$\frac{}{\lambda\ \mathtt{s} \cdot \mathtt{t} \rightarrow_\beta \mathtt{s}[\mathtt{0} \mapsto \mathtt{t}]}$$

$$\frac{\mathtt{s} \rightarrow_\beta \mathtt{t}}{\mathtt{s} \cdot \mathtt{u} \rightarrow_\beta \mathtt{t} \cdot \mathtt{u}} \qquad \frac{\mathtt{s} \rightarrow_\beta \mathtt{t}}{\mathtt{u} \cdot \mathtt{s} \rightarrow_\beta \mathtt{u} \cdot \mathtt{t}}$$

$$\frac{\mathtt{s} \rightarrow_\beta \mathtt{t}}{\lambda\ \mathtt{s} \rightarrow_\beta \lambda\ \mathtt{t}}$$

Formalization: de Bruijn Notation — Beta Reduction II

β-rule in Isabelle/HOL

$$\lambda\ \mathtt{s} \cdot \mathtt{t} \rightarrow_\beta \mathtt{s[0} \mapsto \mathtt{t]}$$

β-rule for the Haskell implementation

$$\mathtt{(Abs\ x\ s)\ :.\ t} \rightarrow_\beta \mathtt{s\ !\ x\ :=\ t}$$

Formalization: de Bruijn Notation — Beta equivalence I

$$\frac{}{\mathtt{t} \approx_\beta \mathtt{t}} \qquad \frac{}{\boldsymbol{\lambda}\ \mathtt{s} \cdot \mathtt{t} \approx_\beta \mathtt{s[0} \mapsto \mathtt{t]}}$$

$$\frac{\mathtt{s} \approx_\beta \mathtt{t}}{\boldsymbol{\lambda}\ \mathtt{s} \approx_\beta \boldsymbol{\lambda}\ \mathtt{t}} \qquad \frac{\mathtt{s} \approx_\beta \mathtt{t}}{\mathtt{t} \approx_\beta \mathtt{s}}$$

$$\frac{\mathtt{s} \approx_\beta \mathtt{t} \qquad \mathtt{p} \approx_\beta \mathtt{q}}{\mathtt{s} \cdot \mathtt{p} \approx_\beta \mathtt{t} \cdot \mathtt{q}}$$

Formalization: de Bruijn Notation — Beta equivalence II

Lemma

$$(p \approx_\beta q) = ((\rightarrow_\beta) \sqcup (\rightarrow_\beta)^{-1})^* \; p \; q$$

Formalization: de Bruijn Notation — Beta equivalence III

Chains of reductions like this:

$$\begin{tikzcd}
a \arrow[dr] & & & & \\
& b \arrow[dr] & & & \\
& & c \arrow[dr] & & \\
& & & \ddots \arrow[dr] & \\
& & & & z
\end{tikzcd}$$

Formalization: de Bruijn Notation — Beta equivalence IV

Can be drawn like this:

$$\begin{tikzcd}
a \arrow[dr, dashed] & \\
& z
\end{tikzcd}$$

...and are represented symbolically with $\rightarrow^*$

Formalization: de Bruijn Notation — Beta equivalence V

$$\mathsf{p} \;\approx_\beta\; \mathsf{q}$$

is graphically depicted by

$$\begin{tikzcd}
& \exists a \arrow[dl, dashed] \arrow[dr, dashed] & & \cdots \arrow[dl, dashed] \arrow[dr, dashed] & & \exists z \arrow[dl, dashed] \arrow[dr, dashed] & \\
p & & \exists b & & \exists y & & q
\end{tikzcd}$$

Formalization: de Bruijn Notation — Consistency Redux

Stronger consistency challenge:

Can we show $1 \not\approx_\beta 2$?

Abstract Confluence

Abstract Confluence — Definition of Confluence

A relation $\rightarrow$ is *confluent* when for all x, y and z where $x \rightarrow^* y$ and $x \rightarrow^* z$, there exists a (not necessarily unique) p where $y \rightarrow^* p$ and $z \rightarrow^* p$

$$\begin{tikzcd}
& x \arrow[dl, dashed] \arrow[dr, dashed] & \\
y \arrow[dr, dashed] & & z \arrow[dl, dashed] \\
& \exists p &
\end{tikzcd}$$

Abstract Confluence — The Church-Rosser Property I

Define the predicate

```
Church_Rosser R
```

to mean

$$\forall \texttt{p q. } (\texttt{R} \sqcup \texttt{R}^{-1})^{*} \texttt{ p q = } (\exists \texttt{x. R}^{*} \texttt{ p x} \wedge \texttt{R}^{*} \texttt{ q x})$$

Abstract Confluence — The Church-Rosser Property II

$$\begin{tikzcd}
 & \exists a \arrow[dl, dashed] \arrow[dr, dashed] & & \cdots \arrow[dl, dashed] \arrow[dr, dashed] & & \exists z \arrow[dl, dashed] \arrow[dr, dashed] & \\
p & & \exists b & & \exists y & & q
\end{tikzcd}$$

if and only if

$$\begin{tikzcd}
p \arrow[dr, dashed] & & q \arrow[dl, dashed] \\
 & \exists x &
\end{tikzcd}$$

Abstract Confluence — Alternate Definition of Confluence

Theorem

```
confluent R = Church_Rosser R
```

A relation R is confluent if and only if R has the Church-Rosser property.

Abstract Confluence — Diamond Property I

A relation $\rightarrow$ has *the diamond property* if for all x, y, and z if $x \rightarrow y$ and $x \rightarrow z$ then there is a (not necessarily unique) p such that $y \rightarrow p$ and $z \rightarrow p$

Abstract Confluence — Diamond Property II

$$\begin{tikzcd}
 & x \arrow[dl] \arrow[dr] & \\
y \arrow[dr] & & z \arrow[dl] \\
 & \exists p &
\end{tikzcd}$$

Abstract Confluence — Diamond Property III

Lemma

$$\texttt{confluent R = diamond (}R^*\texttt{)}$$

Abstract Confluence — Diamond Property IV

Theorem

$$\texttt{diamond R} \implies \texttt{confluent R}$$

Abstract Confluence — Diamond Property V

Claim

The untyped λ-calculus does not have the diamond property

To see this, consider:

$$(\lambda y.\ u\ y\ y)\ ((\lambda x.\ x)\ z)$$

Confluence of Beta-Reduction

Confluence of Beta-Reduction — Parallel Reduction

Attributed to William Tait and Per Martin Löf [1]

$$\frac{}{\langle \mathtt{i} \rangle \;\Rrightarrow_\beta\; \langle \mathtt{i} \rangle}$$

$$\frac{\mathtt{s} \;\Rrightarrow_\beta\; \mathtt{t}}{\boldsymbol{\lambda}\ \mathtt{s} \;\Rrightarrow_\beta\; \boldsymbol{\lambda}\ \mathtt{t}} \qquad \frac{\mathtt{s} \;\Rrightarrow_\beta\; \mathtt{s'} \qquad \mathtt{t} \;\Rrightarrow_\beta\; \mathtt{t'}}{\mathtt{s} \cdot \mathtt{t} \;\Rrightarrow_\beta\; \mathtt{s'} \cdot \mathtt{t'}}$$

$$\frac{\mathtt{s} \;\Rrightarrow_\beta\; \mathtt{s'} \qquad \mathtt{t} \;\Rrightarrow_\beta\; \mathtt{t'}}{\boldsymbol{\lambda}\ \mathtt{s} \cdot \mathtt{t} \;\Rrightarrow_\beta\; \mathtt{s'[0} \mapsto \mathtt{t']}}$$

Confluence of Beta-Reduction — Squeeze Theorems

$$\mathtt{s} \rightarrow_\beta \mathtt{t} \implies \mathtt{s} \Rrightarrow_\beta \mathtt{t}$$

$$\mathtt{s} \Rrightarrow_\beta \mathtt{t} \implies \mathtt{s} \rightarrow_\beta{}^* \mathtt{t}$$

Confluence of Beta-Reduction — Confluence I

Lemma

$$\texttt{diamond}\ (\Rrightarrow_\beta)$$

Lemma

$$\texttt{confluent}\ (\Rrightarrow_\beta)$$

Confluence of Beta-Reduction — Confluence II

Theorem

$$\texttt{confluent}\ (\rightarrow_\beta)$$

Confluence of Beta-Reduction — Confluence III

Claim

$$1 \not\approx_\beta 2$$

Bibliography

[1] M. Takahashi, *Parallel Reductions in λ-Calculus*, Information and Computation, 118 (1995), pp. 120–127.