Lambda Encoding, Types and Confluence

Peng Fu

Advisor: Prof. Aaron Stump
Dept. of Computer Science

Prelude: Programs and Data

- Program as data

Prelude: Programs and Data

- Program as data
- Data as Program?

Prelude: Programs and Data

- Program as data
- Data as Program?
- $2\,f\;a \;= f\;(f\;a)$.

Prelude: Programs and Data

- Program as data
- Data as Program?
- $2\,f\;a \;= f\;(f\;a)$.
- Lambda Calculus

Outline

- Typed Lambda Calculus and Lambda Encoding
- Type Preservation and Confluence
- Limits of Dependent type
- Selfstar
- Summary

Lambda Calculus

How to describe the *plus one* function.

- $x + 1$
- $\xi + 1$ (Frege, Grundgesetze der Arithmetik)
- $\hat{x} + 1$ (Principia Mathematica)
- $\hat{x}.x + 1$ (Church)[1]
- $\wedge x.x + 1$ (Typewriter)
- $\lambda x.x + 1$ (Modern)
- $x + 1$ v.s. $\lambda x.x + 1$

[1] Come from Barendregt's Impact of Lambda Calculus

Lambda Calculus

Higher order expression reduction system

- Terms(expression) $t \ ::= \ x \mid \lambda x.t \mid t\ t'$
- Reduction $(\lambda x.t)t' \to_\beta [t'/x]t$
- In what sense it is higher order?
- Bind and free variable are primitives.
  free variable of $\lambda x.x\ y$ is y, x is called bind variable.
- Variable binding in $\lambda x.\lambda y.x\ y \equiv \lambda z.\lambda x.z\ x$
  $\lambda x.x \equiv \lambda y.y$
- Reduction examples:
  $(\lambda x.x)\ y \to_\beta y$
  $(\lambda x.x\ x)\ y \to_\beta y\ y$.

Lambda Calculus

- To express higher order function
- Semantically, for an $a \in A$, $g : (A \to A) \to A$, where $f \mapsto f(a)$.
- With Lambda Calculus:
  $g \; := \lambda y.y \; a$
  $g \, f \equiv (\lambda y.y \; a) \, f \to_\beta f \; a$
- Function g is emulated at the syntactic level.

Types

- Originated from Russell works, later used by Curry, Church and many others.
- Now common in programming language.
- E.g. types in C, java. Central in Ocaml, Haskell.
- Used to express certain assumptions, e.g. String $\rightarrow$ int
- The notion of types can be generalized to some sophisticated types, leads to theorem provers like Coq, Agda.

Typed Lambda Calculus

Type is expression to govern the form of lambda term: $\Gamma \vdash t : T$

- Simple type: $T \ ::= X \mid T \to T'$
- Context(Environment) $\Gamma \ ::= \ \cdot \mid \Gamma, x : T$
- Typing

$$\frac{(x : T) \in \Gamma}{\Gamma \vdash x : T} \ \textit{Var} \qquad \frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x.t : T_1 \to T_2} \ \textit{Abs}$$

$$\frac{\Gamma \vdash t : T_1 \to T_2 \quad \Gamma \vdash t' : T_1}{\Gamma \vdash t\ t' : T_2} \ \textit{App}$$

Typed Lambda Calculus

Extend simple type:

- Polymorphic type: $T \ ::= \ ... \mid \forall X : \kappa . T$
- Kind: $\kappa \ ::= \ *$
- Context: $\Gamma \ ::= \ ... \mid X : \kappa$
- Typing

$$\frac{\Gamma, X : \kappa \vdash t : T}{\Gamma \vdash t : \forall X : \kappa . T} \textit{ Gen} \qquad \frac{\Gamma \vdash t : \forall X : \kappa . T \quad \Gamma \vdash T' : \kappa}{\Gamma \vdash t : [T'/X]T} \textit{ Inst}$$

Typed Lambda Calculus

Extends Polymorphic type(System F):

- Product type(dependent type): $T \ ::= \ ... \mid \Pi x : T.T' \mid T \ t$
- Kind: $\kappa \ ::= \ * \mid \xi x : T.\kappa$.
- Typing

$$\frac{\Gamma, x : T' \vdash t : T \quad x \in FV(T)}{\Gamma \vdash \lambda x.t : \Pi x : T'.T} \ \textit{Pi}$$

$$\frac{\Gamma \vdash t : \Pi x : T'.T \quad \Gamma \vdash t' : T'}{\Gamma \vdash t \ t' : [t'/x]T} \ \textit{Elim}$$

$$\frac{\Gamma \vdash t : [t_1/x]T \quad t_1 =_\beta t_2}{\Gamma \vdash t : [t_2/x]T} \ \textit{Conv}$$

Typed Lambda Calculus

Why Dependent Types?

- Curry-Howard correspondent for $\Gamma \vdash t : T$
  $<$ Environment $> \vdash <$ program $>:<$ type $>$
  $<$ Assumptions $> \vdash <$ proof $>:<$ formula $>$
- Type Preservation:
  If $\Gamma \vdash t : T$ and $t \rightarrow_\beta t'$, then $\Gamma \vdash t' : T$.
- Strong Normalization:
  If $\Gamma \vdash t : T$, then there is no infinite beta reduction sequence.

Algebraic Data types

An example in Haskell:

```haskell
data Nat = Zero
          | Succ Nat

add :: Nat-> Nat -> Nat
add n m = case n of
          Zero -> m
         | Succ p -> add p (Succ m)

data List a = Nil | Cons a (List a)
data Tree = Empty
          | Leaf Int
          | Node Tree Tree
```

Core Language Design

To support (algebraic) data type, one way is to add data type and pattern matching(to core) as primitive.

- expression for data type declaration:
  data $T\ (a_i : A_i)_{i \in I} : A$ where $\{C_i : A_i\}_{i \in I}$
- expression for pattern matching :
  case a of $\{C_i(x_1, ...x_u)\ \Rightarrow\ a\}_{i \in I}$

Core Language Design

To support (algebraic) data type, one way is to add data type and pattern matching(to core) as primitive.

- expression for data type declaration:
  data $T\ (a_i : A_i)_{i \in I} : A$ where $\{C_i : A_i\}_{i \in I}$
- expression for pattern matching :
  case a of $\{C_i(x_1, ...x_u)\ \Rightarrow\ a\}_{i \in I}$
- Now typing rule become:

$$\dfrac{\begin{array}{l} r = n + m \\ x_1 \ldots x_n \not\in \mathsf{FV}\,(|t''|) \\ \mathsf{getArgs}(t') = [w_1, \ldots, w_m] \\ \mathsf{buildCtx}(\Delta_2(\mathsf{getHC}(t'))) = [y_1 : t''_1, \ldots, y_n : t''_m] \\ \mathsf{cut}([y_1 : t''_1, \ldots, y_n : t''_m], \mathsf{buildCtx}(\mathsf{getCType}(t', \mathsf{C}, \Delta))) = [x_1 : t'_1, \ldots, x_n : t'_n] \\ \Delta, \Gamma \vdash^{TB} H\, t_1\, t'\, y\, (l - \{\mathsf{C} : getCType(t', C, \Delta)\}) : t'' \\ \Delta, \Gamma, x_1 : [w_1/y_1]t'_1, \ldots, x_n : [w_m/y_m]t'_n, y : t_1 = (\mathsf{C}\, w'_{1\,\varepsilon_1} \ldots w'_{r\,\varepsilon_n}) \vdash t_2 : t'' \end{array}}{\Delta, \Gamma \vdash^{TB} (\mathsf{C}\, x_{1\,\varepsilon'_1} \ldots x_{n\,\varepsilon'_n} \Rightarrow t_2 \mid H)\, t_1\, t'\, y\, l : t''}\ \textsc{TB\_Branch}$$

Core Language Design

Complicate the analysis for core language.

- Type Preservation Proof for Standard ML
- *The machine-assisted proof of programming language properties*, 1996, Ph.D thesis by M. VanInwegen.
- Machine assisted proof by HOL.
- Quoted from abstract:
  "We were not able to complete the proof of type preservation because it is not true: we found counterexamples."
- In Haskell Core Language [2]:
  "Case expressions are the most complicated bit of Core. "

[2] http://hackage.haskell.org/trac/ghc/wiki/Commentary/Compiler/CoreSynType

Scott Numerals

- Scott encoding with Recursive Definition:

```
Zero = lam s . lam z . z
Succ n =  lam s . lam z . s n
add n m = case n of
              Zero -> m
            | Succ p -> add p (Succ m)
pred n  = case n of
              Zero -> Zero
            | Succ p -> p
```

Scott Numerals

- Scott encoding with Recursive Definition:

```
Zero = lam s . lam z . z
Succ n =  lam s . lam z . s n
add n m = case n of
             Zero -> m
           | Succ p -> add p (Succ m)
pred n  = case n of
             Zero -> Zero
           | Succ p -> p
```

- Translate to lambda calculus with recursive definition
  Zero $:= \lambda s.\lambda z.z$
  Succ $:= \lambda n.\lambda s.\lambda z.s\ n$
  add $:= \lambda n.\lambda m.n\ (\lambda p.\text{add}\ p\ (\text{Succ}\ m))\ m$
  pred $:= \lambda n.n\ (\lambda p.p)$ Zero

Scott Numerals

- Isn't it great? No primitive data type and pattern matching needed!
- Beta reduction and definition unfolding are enough.
- the catch(or is it?): need recursive type and operation on Scott encoding data can not be typed in polymorphic type system.

$$\frac{\Gamma \vdash t : [\mu X.T/X]T}{\Gamma \vdash t : \mu X.T}\ \textit{Fold} \qquad \frac{\Gamma \vdash t : \mu X.T}{\Gamma \vdash t : [\mu X.T/X]T}\ \textit{unFold}$$

Nat $:= \mu X.(X \to U) \to U \to U$ for any type U.
$\cdot \vdash$ add : Nat $\to$ Nat $\to$ Nat.
$\cdot \vdash$ Zero : Nat.
$\cdot \vdash$ Succ : Nat $\to$ Nat.

Church Numerals

Idea, $((2\,f)\;a) \;= f\;(f\;a)$, so $2 := \lambda f.\lambda a.f\;(f\;a)$.

- Church encoding

```
Zero = lam s . lam z . z
Succ n =  lam s . lam z . s (n s z)
iterator n f a = n f a
add n m = iterator n Succ m
```

- One can see
  $1\;$ = Succ Zero $\rightarrow^*_\beta \lambda s.\lambda z.s\;z$
  $2\;$ = Succ $1 \rightarrow^*_\beta \lambda s.\lambda z.s\;(s\;z)$
- Easily translated to lambda calculus.

Church Numerals

- This is also great.
- Even better, it can be typed in system F(Polymorphic type).
  Nat $:= \forall X : *.(X \to X) \to X \to X$
  $\cdot \vdash \text{Succ} : \text{Nat} \to \text{Nat}$
  $\cdot \vdash \text{iterator} : \forall X : *.\text{Nat} \to (X \to X) \to X \to X$
  $\cdot \vdash \text{add} : \text{Nat} \to \text{Nat} \to \text{Nat}$
- The catch, inefficient predecessor.
- The predecessor function takes linear time(beta reductions) to compute, while with Scott encoding it only takes constant time.

Outline

- Typed Lambda Calculus and Lambda Encoding
- Type Preservation and Confluence
- Limits of Dependent type
- Selfstar
- Summary

Abstract Reduction System

- Abstract Reduction System(ARS): $(\mathcal{A}, \{\rightarrow_i\}_{i\in\mathcal{I}})$
- Basic concepts: $a \overset{*}{\rightarrow}_i b$(or $a \twoheadrightarrow_i b$), $a =_i b$, *reducible*, *normal form*.
- Confluence and Church-Rosser property:
  $\rightarrow := \bigcup_{i\in\mathcal{I}} \rightarrow_i$.

$$
\begin{tikzcd}
 & a \arrow[dl] \arrow[dr] & \\
b \arrow[dr, dashed] & & c \arrow[dl, dashed] \\
 & d &
\end{tikzcd}
\quad \text{iff} \quad
\begin{tikzcd}
a \arrow[r, phantom, "="] \arrow[dr, dashed] & & b \arrow[dl, dashed] \\
 & c &
\end{tikzcd}
$$

Confluence and Type Preservation

- Recall type preservation:
  If $\Gamma \vdash t : T$ and $t \to t'$, then $\Gamma \vdash t' : T$.
  Usually prove by induction on derivation of $\Gamma \vdash t : T$.

Confluence and Type Preservation

- Recall type preservation:
  If $\Gamma \vdash t : T$ and $t \to t'$, then $\Gamma \vdash t' : T$.
  Usually prove by induction on derivation of $\Gamma \vdash t : T$.
- Need to analyze the case:
  $$\frac{\Gamma \vdash \lambda x.t : T' \to T \quad \Gamma \vdash t' : T'}{\Gamma \vdash (\lambda x.t)t' : T}\ \textit{App}$$

Confluence and Type Preservation

- Recall type preservation:
  If $\Gamma \vdash t : T$ and $t \to t'$, then $\Gamma \vdash t' : T$.
  Usually prove by induction on derivation of $\Gamma \vdash t : T$.
- Need to analyze the case:
  $$\frac{\Gamma \vdash \lambda x.t : T' \to T \quad \Gamma \vdash t' : T'}{\Gamma \vdash (\lambda x.t)t' : T} \; \textit{App}$$
- Present of non-syntax directed rule:
  $$\frac{\Gamma \vdash t : T' \quad T = T'}{\Gamma \vdash t : T} \; \textit{Conv}$$
- Inversion on $\Gamma \vdash \lambda x.t : T' \to T$ gives us only:
  $\Gamma, x : T_1 \vdash t : T_2$ where $T_1 \to T_2 = T' \to T$.

Confluence and Type Preservation

- Recall type preservation:
  If $\Gamma \vdash t : T$ and $t \to t'$, then $\Gamma \vdash t' : T$.
  Usually prove by induction on derivation of $\Gamma \vdash t : T$.
- Need to analyze the case:
  $$\frac{\Gamma \vdash \lambda x.t : T' \to T \quad \Gamma \vdash t' : T'}{\Gamma \vdash (\lambda x.t)t' : T} \textit{ App}$$
- Present of non-syntax directed rule:
  $$\frac{\Gamma \vdash t : T' \quad T = T'}{\Gamma \vdash t : T} \textit{ Conv}$$
- Inversion on $\Gamma \vdash \lambda x.t : T' \to T$ gives us only:
  $\Gamma, x : T_1 \vdash t : T_2$ where $T_1 \to T_2 = T' \to T$.
- Want *inverse structure congruence*:
  If $T_1 \to T_2 = T_1' \to T_2'$, then $T_1 = T_1'$ and $T_2 = T_2'$.

Confluence and Type Preservation

- Recall type preservation:
  If $\Gamma \vdash t : T$ and $t \to t'$, then $\Gamma \vdash t' : T$.
  Usually prove by induction on derivation of $\Gamma \vdash t : T$.
- Need to analyze the case:
  $$\frac{\Gamma \vdash \lambda x.t : T' \to T \quad \Gamma \vdash t' : T'}{\Gamma \vdash (\lambda x.t)t' : T}\ \textit{App}$$
- Present of non-syntax directed rule:
  $$\frac{\Gamma \vdash t : T' \quad T = T'}{\Gamma \vdash t : T}\ \textit{Conv}$$
- Inversion on $\Gamma \vdash \lambda x.t : T' \to T$ gives us only:
  $\Gamma, x : T_1 \vdash t : T_2$ where $T_1 \to T_2 = T' \to T$.
- Want *inverse structure congruence*:
  If $T_1 \to T_2 = T_1' \to T_2'$, then $T_1 = T_1'$ and $T_2 = T_2'$.
- By induction on derivation of $T_1 \to T_2 = T_1' \to T_2'$, case:
  $$\frac{T_1 \to T_2 = T_3 \quad T_3 = T_1' \to T_2'}{T_1 \to T_2 = T_1' \to T_2'}\ \textit{Trans}$$

Confluence and Type Preservation

- Recall type preservation:
  If $\Gamma \vdash t : T$ and $t \to t'$, then $\Gamma \vdash t' : T$.
  Usually prove by induction on derivation of $\Gamma \vdash t : T$.
- Need to analyze the case:
  $$\frac{\Gamma \vdash \lambda x.t : T' \to T \quad \Gamma \vdash t' : T'}{\Gamma \vdash (\lambda x.t)t' : T} \textit{ App}$$
- Present of non-syntax directed rule:
  $$\frac{\Gamma \vdash t : T' \quad T = T'}{\Gamma \vdash t : T} \textit{ Conv}$$
- Inversion on $\Gamma \vdash \lambda x.t : T' \to T$ gives us only:
  $\Gamma, x : T_1 \vdash t : T_2$ where $T_1 \to T_2 = T' \to T$.
- Want *inverse structure congruence*:
  If $T_1 \to T_2 = T_1' \to T_2'$, then $T_1 = T_1'$ and $T_2 = T_2'$.
- By induction on derivation of $T_1 \to T_2 = T_1' \to T_2'$, case:
  $$\frac{T_1 \to T_2 = T_3 \quad T_3 = T_1' \to T_2'}{T_1 \to T_2 = T_1' \to T_2'} \textit{ Trans}$$
- Can't apply induction hypothesis!

Confluence and Type Preservation

Inverse structure congruence:
If $T_1 \to T_2 = T_1' \to T_2'$, then $T_1 = T_1'$ and $T_2 = T_2'$.

- We can define $=$ to be the reflexive, symmetric, transitive closure of $\rightarrowtail$.
- We show that $\rightarrowtail$ is confluent, thus Church-Rosser.
- So $T_1 \to T_2 = T_1' \to T_2'$ implies there is a T_3 such that $T_1 \to T_2 \overset{*}{\rightarrowtail} T_3$ and $T_1' \to T_2' \overset{*}{\rightarrowtail} T_3$.
- We design $\rightarrowtail$ in such a way that $T_a \to T_b \rightarrowtail T$ iff $T \equiv T_a' \to T_b$ or $T \equiv T_a \to T_b'$, where $T_a \rightarrowtail T_a'$, $T_b \rightarrowtail T_b'$.
- Thus we conclude inverse structure congruence, conquered one problem for proving type preservation.

Confluence: Tait-Martin Löf's Method

Lambda calculus $(\Lambda, \rightarrow_\beta)$ as ARS is confluent.

- Diamond property:

$$
\begin{tikzcd}
 & a \arrow[dl] \arrow[dr] & \\
b \arrow[dr, dashed] & & c \arrow[dl, dashed] \\
 & d &
\end{tikzcd}
\quad \text{implies} \quad
\begin{tikzcd}
 & & a \arrow[dl] \arrow[dr] & & \\
 & c_1 \arrow[dl] \arrow[dr, dashed] & & c_2 \arrow[dl, dashed] \arrow[dr] & \\
e \arrow[dr, dashed] & & d \arrow[dl, dashed] \arrow[dr, dashed] & & b \arrow[dl, dashed] \\
 & c_1 \arrow[dr, dashed] & & c_2 \arrow[dl, dashed] & \\
 & & c & &
\end{tikzcd}
$$

Confluence: Tait-Martin Löf's method

$\to_\beta$ reduction does not have diamond property.

$$
\begin{tikzcd}
& (\lambda x.f\ x\ x)\ ((\lambda y.y\ z)\ u) \arrow[dl] \arrow[dr] & \\
f\ ((\lambda y.yz)u)((\lambda y.yz)u) \arrow[dr, dashed] & & (\lambda x.f\ x\ x)(u\ z) \arrow[dl, dashed] \\
& ? &
\end{tikzcd}
$$

Not joinable in one step, but joinable in “many” steps.

Confluence: Tait-Martin Löf's method

- Parallel Reduction(a notion of "many" steps).
  $$\frac{t_1 \Rightarrow_\beta t_1' \quad t_2 \Rightarrow_\beta t_2'}{(\lambda x.t_1)t_2 \Rightarrow_\beta [t_2'/x]t_1'}$$
- It has diamond property.
- $\rightarrow_\beta \subseteq \Rightarrow_\beta \subseteq \stackrel{*}{\rightarrow}_\beta$, which implies $(\stackrel{*}{\rightarrow}_\beta) = (\stackrel{*}{\Rightarrow}_\beta)$
- Confluence of $\Rightarrow_\beta$ implies confluence of $\rightarrow_\beta$

Confluence: Takahashi's method

- A stronger property, triangle property:

$$\begin{tikzcd} t \arrow[dr, Rightarrow] \arrow[dd, mapsto] & \\ & t' \arrow[dl, Rightarrow, dashed] \\ t^* & \end{tikzcd} \quad \text{implies} \quad \begin{tikzcd} & t \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] \arrow[dd, mapsto] & \\ t_1 \arrow[dr, Rightarrow, dashed] & & t_2 \arrow[dl, Rightarrow, dashed] \\ & t^* & \end{tikzcd}$$

- Parallel contractions:

  $x^* \ := \ x$.
  $(\lambda x.t)^* \ := \ \lambda x.t^*$.
  $(t_1\ t_2)^* \ := \ t_1^*\ t_2^*$ if $t_1\ t_2$ is not a beta redex.
  $((\lambda x.t_1)\ t_2)^* \ := \ [t_2^*/x]t_1^*$.

Confluence: Barendregt's method

- Strip property:

$$\begin{tikzcd}
 & a \arrow[dl] \arrow[dr] & \\
b \arrow[dr, dashed] & & c \arrow[dl, dashed] \\
 & d &
\end{tikzcd}
\quad \text{implies} \quad
\begin{tikzcd}
 & & a \arrow[dl] \arrow[dr] & \\
 & c_1 \arrow[dl] \arrow[dr, dashed] & & b \arrow[dl, dashed] \\
e \arrow[dr, dashed] & & d \arrow[dl, dashed] & \\
 & c_1 & &
\end{tikzcd}$$

Confluence: Hardin's Interpretation method

- Assumption I: $\to := \to_1 \cup \to_2$ and $\to_1$ is strongly normalizing and confluent.
- Assumption II:$\to_i \subseteq \to$ is defined on $\to_1$ normal form $\nu(t)$.
- Assumption III: If $a \to_2 b$, then $\nu(a) \to_i \nu(b)$.
- Then: Confluence of $\to_i$ implies confluence of $\to$.
-

$$
\begin{tikzcd}
& & a \arrow[dll] \arrow[d, dashed, "1"'] \arrow[drr] & & \\
a' \arrow[dr, dashed, "1"] & & \nu(a) \arrow[dl, dashed, "i"'] \arrow[dr, dashed, "i"] & & a'' \arrow[dl, dashed, "1"'] \\
& \nu(a') \arrow[dr, dashed, "i"] & & \nu(a'') \arrow[dl, dashed, "i"'] & \\
& & b & &
\end{tikzcd}
$$

Hardin's Interpretation method: Applications

- Confluence of $(\Lambda_\mu, \to_\beta, \to_\mu)$, originated from Selfstar.
- Λ_μ denotes terms $t ::= x \mid \lambda x.t \mid tt' \mid \mu t$.
- Closure: $\mu ::= \{x_i \mapsto t_i\}_{i \in \mathcal{I}}$. Locality.
- New reductions:

$$\frac{(x_i \mapsto t_i) \in \mu}{\mu x_i \to_\beta \mu t_i} \qquad \frac{dom(\mu) \# \mathsf{FV}(t)}{\mu t \to_\mu t}$$

$$\frac{}{\mu(\lambda x.t) \to_\mu \lambda x.\mu t} \qquad \frac{}{\mu(t_1 t_2) \to_\mu (\mu t_1)(\mu t_2)}$$

Outline

- Typed Lambda Calculus and Lambda Encoding
- Type Preservation and Confluence
- Limits of Dependent type
- Selfstar
- Summary

Limits of Dependent Type

How to show some formula is unprovable.

- Formalized the notion of proof.
- If there is a proof of $\perp$, then draw a contradiction at meta-level.
- Under Curry Howard correspondent, showing some type is uninhabited.

Limits of Dependent Type

Recall second order dependent type:

- Type as expression:
  $T \ := X \mid \forall X : \kappa.T \mid T_1 \to T_2 \mid \Pi x : T_1.T_2 \mid T\ t$
  Kind as expression $\kappa \ := * \mid \xi x : T.\kappa$

[3] Coquand's Metamathematical investigations of a calculus of constructions
[4] Werner's A Normalization Proof for an Impredicative Type System with Large Elimination over Integers

Limits of Dependent Type

Recall second order dependent type:

- Type as expression:
  $T \ := X \mid \forall X : \kappa . T \mid T_1 \to T_2 \mid \Pi x : T_1 . T_2 \mid T \ t$
  Kind as expression $\kappa \ := * \mid \xi x : T . \kappa$
- Induction principle:
  $\forall P : (\xi x : \mathsf{Nat}.*). P \ 0 \to (\Pi y : \mathsf{Nat}.(P \ y) \to (P \ (\mathsf{S} \ y))) \to \Pi x : \mathsf{Nat}.P \ x$

[3] Coquand's Metamathematical investigations of a calculus of constructions

[4] Werner's A Normalization Proof for an Impredicative Type System with Large Elimination over Integers

Limits of Dependent Type

Recall second order dependent type:

- Type as expression:
  $T \ := X \mid \forall X : \kappa.T \mid T_1 \to T_2 \mid \Pi x : T_1.T_2 \mid T\ t$
  Kind as expression $\kappa \ := * \mid \xi x : T.\kappa$
- Induction principle:
  $\forall P : (\xi x : \mathsf{Nat}.*).P\ 0 \to (\Pi y : \mathsf{Nat}.(P\ y) \to (P\ (\mathsf{S}\ y))) \to \Pi x : \mathsf{Nat}.P\ x$
- $P : (\xi x : \mathsf{Nat}.*), a : P\ 0, b : \Pi y : \mathsf{Nat}.(P\ y) \to (P\ (\mathsf{S}\ y)), x : \mathsf{Nat} \vdash$
  $(b\ 1\ (b\ 0\ a)) : P\ (\mathsf{S}\ 1)$
  Induction is not derivable(provable) within Dependent type system[3]

---

[3] Coquand's Metamathematical investigations of a calculus of constructions

[4] Werner's A Normalization Proof for an Impredicative Type System with Large Elimination over Integers

Limits of Dependent Type

Recall second order dependent type:

- Type as expression:
  $T \ := X \mid \forall X : \kappa.T \mid T_1 \to T_2 \mid \Pi x : T_1.T_2 \mid T\ t$
  Kind as expression $\kappa \ := * \mid \xi x : T.\kappa$
- Induction principle:
  $\forall P : (\xi x : \mathsf{Nat}.*).P\ 0 \to (\Pi y : \mathsf{Nat}.(P\ y) \to (P\ (\mathsf{S}\ y))) \to \Pi x : \mathsf{Nat}.P\ x$
- $P : (\xi x : \mathsf{Nat}.*), a : P\ 0, b : \Pi y : \mathsf{Nat}.(P\ y) \to (P\ (\mathsf{S}\ y)), x : \mathsf{Nat} \vdash$
  $(b\ 1\ (b\ 0\ a)) : P\ (\mathsf{S}\ 1)$
  Induction is not derivable(provable) within Dependent type system[3]
- Since $\perp$ is uninhabit, can not inhabit types like $T \to \perp$, where T is inhabited. Thus can not prove $0 \neq 1$.[4]

---

[3] Coquand's Metamathematical investigations of a calculus of constructions

[4] Werner's A Normalization Proof for an Impredicative Type System with Large Elimination over Integers

Limits of Dependent Type

Recall second order dependent type:

- Type as expression:
  $T ::= X \mid \forall X : \kappa.T \mid T_1 \to T_2 \mid \Pi x : T_1.T_2 \mid T\ t$
  Kind as expression $\kappa ::= * \mid \xi x : T.\kappa$
- Induction principle:
  $\forall P : (\xi x : \mathsf{Nat}.*).P\ 0 \to (\Pi y : \mathsf{Nat}.(P\ y) \to (P\ (\mathsf{S}\ y))) \to \Pi x : \mathsf{Nat}.P\ x$
- $P : (\xi x : \mathsf{Nat}.*), a : P\ 0, b : \Pi y : \mathsf{Nat}.(P\ y) \to (P\ (\mathsf{S}\ y)), x : \mathsf{Nat} \vdash (b\ 1\ (b\ 0\ a)) : P\ (\mathsf{S}\ 1)$
  Induction is not derivable(provable) within Dependent type system[3]
- Since $\perp$ is uninhabit, can not inhabit types like $T \to \perp$, where T is inhabited. Thus can not prove $0 \neq 1$.[4]
- These compromise the usage of Church encoding in dependent type system.

---

[3] Coquand's Metamathematical investigations of a calculus of constructions

[4] Werner's A Normalization Proof for an Impredicative Type System with Large Elimination over Integers

Outline

- Typed Lambda Calculus and Lambda Encoding
- Type Preservation and Confluence
- Limits of Dependent type
- Selfstar
- Summary

Selfstar: I

- Recall that:
  $x : \mathsf{Nat} \vdash \bar{n} : \forall P : (\xi x : \mathsf{Nat}.*).P\ 0 \to (\Pi y : \mathsf{Nat}.(P\ y) \to (P\ (\mathsf{S}\ y))) \to P\ \bar{n}$,
  for any Church numerals $\bar{n}$
- Dependent type system, namely, Π construct can not grasp this level of quantification.
- One way to try to capture(with the help of recursion) this is:
  $x : \mathsf{Nat} \vdash \bar{n} : \iota x.\forall P : (\xi x : \mathsf{Nat}.*).P\ 0 \to (\Pi y : \mathsf{Nat}.(P\ y) \to (P\ (\mathsf{S}\ y))) \to P\ x$
  $\mathsf{Nat} := \iota x.\forall P : (\xi x : \mathsf{Nat}.*).P\ 0 \to (\Pi y : \mathsf{Nat}.(P\ y) \to (P\ (\mathsf{S}\ y))) \to P\ x$
- add two typing rules:
  $$\frac{\Gamma \vdash t : \iota x.T}{\Gamma \vdash t : [t/x]T}\ \textit{SelfInst} \qquad \frac{\Gamma \vdash t : [t/x]T}{\Gamma \vdash t : \iota x.T}\ \textit{SelfGen}$$
- Introduce closure $\mu := \{x_i \mapsto t_i\}_{i \in \mathcal{I}} \cup \{X_i \mapsto T_i\}_{i \in \mathcal{N}}$
- Wrap around term and type, $\mu t, \mu T$, $\tilde{\mu} \in \Gamma$.

Selfstar: II

Church encoding and Scott encoding in Selfstar

- Church encoding($\tilde{\mu}_c$):
  $(\mathsf{Nat} : *) \mapsto$
  $\iota x.\Pi C : \mathsf{Nat} \to *.(\Pi n : \mathsf{Nat}.(C\ n) \to (C\ (\mathsf{S}\ n))) \to (C\ 0) \to (C\ x)$
  $(\mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}) \mapsto \lambda n.\lambda C.\lambda s.\lambda z.s\ n\ (n\ C\ s\ z)$
  $(0 : \mathsf{Nat}) \mapsto \lambda C.\lambda s.\lambda z.z.$
- Scott encoding($\tilde{\mu}_s$):
  $(\mathsf{Nat} : *) \mapsto \iota x.\Pi C : \mathsf{Nat} \to *.(\Pi n : \mathsf{Nat}.C\ (\mathsf{S}\ n)) \to (C\ 0) \to (C\ x)$
  $(\mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}) \mapsto \lambda n.\lambda C.\lambda s.\lambda z.s\ n$
  $(0 : \mathsf{Nat}) \mapsto \lambda C.\lambda s.\lambda z.z$

Selfstar: II

- Induction principle for Church encoding:
  $\tilde{\mu}_c \vdash (\mathsf{Ind} \ := \lambda C.\lambda s.\lambda z.\lambda n.n\ C\ s\ z) :$
  $\Pi C : \mathsf{Nat} \to *.\Pi n : \mathsf{Nat}.((C\ n) \to (C\ (\mathsf{S}\ n))) \to C\ 0 \to \Pi n : \mathsf{Nat}.C\ n$
- Addition function:
  $\tilde{\mu}_c \vdash (\mathsf{add} := \lambda n.\lambda m.\mathsf{Ind}\ (\lambda y.\mathsf{Nat})\ (\lambda x.\mathsf{S})\ m\ n) : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}.$
- Case analysis principle for Scott encoding:
  $\tilde{\mu}_s \vdash (\mathsf{Case} \ := \lambda C.\lambda s.\lambda z.\lambda n.n\ C\ s\ z) :$
  $\Pi C : \mathsf{Nat} \to *.\Pi n : \mathsf{Nat}.(C\ (\mathsf{S}\ n)) \to C\ 0 \to \Pi n : \mathsf{Nat}.C\ n$
- addition function:
  $(\mathsf{add} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}) \mapsto$
  $\lambda n.\lambda m.\mathsf{Case}\ (\lambda n.\mathsf{Nat})\ \ (\lambda p.(\mathsf{S}\ (\mathsf{add}\ p\ m)))\ m\ n$

Selfstar: III

- Due to $* : *$ and recursive defintion, term does not correspond to proof, type does not correspond to formula.
- Future work: show a fragment of Selfstar that can be erased to F^{ω}.

Outline

- Typed Lambda Calculus and Lambda Encoding
- Type Preservation and Confluence
- Limits of Dependent type
- Selfstar
- Summary

Summary

- We seen Church and Scott encoding data and as alternatives to implement algebraic data type.
- The use of confluence in preservation proof.
- Several methods to prove confluence.
- Limits of dependent type system give rise to Selfstar.
- Introduced Selfstar.

Last But Not Least

- Thank my advisor Prof. Aaron Stump.
- Thank my exam committee:
  Prof. Cesare Tinelli, Prof. Kasturi Varadarajan, Prof. Ted Herman, Prof. Douglas Jones.
- For all the audiences.