Dependent Lambda Encodings with Self Types

Aaron Stump, Peng Fu

Dept. of Computer Science
The University of Iowa
Iowa City, Iowa, USA

Lambda Calculus and Lambda Numerals

- Lambda Calculus:
  $t ::= x \mid \lambda x.t \mid t\ t'$
- Beta-Reduction:
  $(\lambda x.t)t' \rightsquigarrow_\beta [t'/x]t$
- Church encoding:
  $$\begin{array}{lcl} 0 & = & \lambda s.\lambda z.z \\ n & = & \lambda s.\lambda z.(\underbrace{s \ldots (\ s}_{n} z)) \end{array}$$
- Scott encoding:
  $$\begin{array}{lcl} 0 & = & \lambda s.\lambda z.z \\ n & = & \lambda s.\lambda z.s\,(n-1) \end{array}$$
- "Encoding" usually means to encode a variety of datatypes (i.e. Lists, Vectors, Trees ...), in this talk we only focus on numerals.

Church Encoding v.s. Scott Encoding

- Scott encoding :

  $pred := \lambda n.n\ (\lambda m.m)\ 0$

  E.g. $pred\ 1 = (\lambda n.n\ (\lambda m.m)\ 0)1 \leadsto_\beta 1(\lambda m.m)0 = (\lambda s.\lambda z.s\ 0)\ (\lambda m.m)\ 0 \leadsto_\beta (\lambda m.m)\ 0 \leadsto_\beta 0$

- Church encoding :

  $pred := \lambda n.snd\ (n\ (\lambda p.pair\ (Succ\ (fst\ p))\ (fst\ p))\ (pair\ 00))$, where $snd, pair, Succ, fst$ need to be further defined. This is by Kleene.

  Decompose $S...S0$, leave out the last S.

Church Encoding v.s. Scott Encoding

Typing in in Girard-Reynolds' System F:

- Church encoding :

  $nat := \Pi A.(A \to A) \to A \to A$

  $0 := \Lambda A.\lambda s : A \to A.\lambda z : A.z$

  $1 := \Lambda A.\lambda s : A \to A.\lambda z : A.s\ z$

- Scott encoding :

  No known way to type Scott numerals in system F or Calculus of Constructions.

Church Encoding in Calculus of Constructions(CC)

- Induction principle is not derivable. [H. Geuvers 2001]
- Can not derive $0 \neq 1$. [B. Werner 1992]
- Inefficient predecessor operation.
- Leads to Calculus of Inductive Constructions(CIC).
  - Adding datatype as primitive
  - CIC is much more complex than CC.
  - A glimpse of the complexity?

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

Self-Types: Motivations

Can we take a step back from CIC?

- For Scott-encoded data, want:

  $n : \Pi C : Nat \to \star.(\Pi y : Nat.C\ (S\ y)) \to C\ 0 \to C\ \boxed{n}$

  where $n \equiv \Lambda C.\lambda s.\lambda z.s(n-1)$
- The type of n literally mentions n.

Self-Types: Motivations

- How about this:

  $n : \boxed{\text{self x}} .\Pi C : Nat \rightarrow \star.(\Pi y : Nat.C\ (S\ y)) \rightarrow C\ 0 \rightarrow C\ \boxed{\text{x}}$

- Together with these:

$$\frac{}{\Gamma \vdash \texttt{self}\ x.t' \stackrel{t}{=} [t/x]t'} \qquad \frac{\Gamma, x : \texttt{self}\ x.t \vdash t : t'}{\Gamma \vdash \texttt{self}\ x.t : t'}$$

$$\frac{\Gamma \vdash t : t' \quad \Gamma \vdash t' \stackrel{t}{=} t''}{\Gamma \vdash t : t''}$$

- The language:

  $terms\ \ t ::= x \mid \star \mid t\ t' \mid \lambda x : t.t' \mid \Pi x : t.t' \mid \texttt{self}\ x.t$
  $\mid \mu\ x_1 = t_1, \cdots, x_n = t_n.\ \ t$

Couples of Points

- Predicativity:
  - $\star : \star$, that is, the $\star$ that categorizes all types is also categorized by $\star$.
  - Will be inconsistency as logic, but no problem for programs.
  - In $\star : \star$, lambda encoded data can be used in both terms and types levels.
  - While in a predicative type system, different levels will have their own data.
  - Admittedly, the mutual recursive operator is causing us a lot of problems when we try to prove the language is type safe.
- New type construct:
  - We propose self types:
    - $\mathtt{self}\ x.t$ is a type.
    - t can use x to refer to the subject of the type.

Implementation

- In Google Trellys repo: *https://trellys.googlecode.com/svn/trunk/trellys/lib/subcore/src/*.
- Around 1000 source lines OCaml.(Written by Prof. Stump.)

Example: Scott Encoding in Self-type

μ *nat* $=$ *self n*. Π *C* : (*nat* $\rightarrow$ $*$) .
(Π *n* : *nat* . (*C* (*succ n*))) $\rightarrow$ (*C zero*) $\rightarrow$ (*C* (*conv n to nat*)) ,

zero $=$ *conv*
(Π*C* : *nat* $\rightarrow$ $*$.
λ *s* : (Π *n* : *nat* . (*C* (*succ n*))). λ *z* : (*C zero*). *z*)
to nat ,

succ $=$ λ *n* : *nat*. *conv*
(Π *C* : *nat* $\rightarrow$ $*$. λ *s* : (Π *n* : *nat*.(*C* (*succ n*))). λ *z* : (*C zero*).(*s n*)
to nat .

body...

Summary

- Self types allows lambda-encoding data:
  - $\star : \star$, so no problem with having data at different levels as predicative system.
  - $\texttt{self}\ x.t$ allows us actully inhabit dependent type likes $\Pi n : Nat.P(n)$.
  - Church- and Scott-encodings work well with self type.
- Future works:
  - Study more about the metatheoretic properties of self types.
  - Moving toward a logical fragment.
  - Thank you!