Lambda Encodings in Type Theory

Peng Fu

Advisor: Prof. Aaron Stump
Department of Computer Science

Outline

- Introduction
- An Attempt to Expressiveness through Internalization.
  A Framework for Internalizing Relations into Type Theory. Peng Fu, Aaron Stump, Jeff Vaughan. PSATTT'11
- Lambda Encodings with Dependent Type.
  Self Types for Dependently Typed Lambda Encodings. Peng Fu, Aaron Stump. RTA-TLCA 2014
- Lambda Encoding with Comprehension.
- Implementation and Future Improvements.

Introduction

Common features of functional programming languages:

- Algebraic data type.
- Pattern matching, recursion and functional application.
- Type inference.

Introduction

```
data List A where
   nil :: List A
   cons :: A -> List A -> List A
  deriving Ind

(++) nil l = l
(++) (cons u l') l = cons u (l'++ l)

--inferred
(++) :: forall A . List A -> List A -> List A
```

Introduction

We would like to reason about the program.

```
theorem assoc . forall l1 l2 l3 A . l1 :: List A ->
         Eq ((l1 ++ l2) ++ l3) (l1 ++ (l2 ++ l3))
proof
    ... (19 lines proofs)
qed
```

Introduction

- How to implement the theorem proving feature?
  - Build-in user defined data type.
  - Assume induction principle.
  - Assume other axioms as needed.

Introduction

- How to implement the theorem proving feature?
  - Build-in user defined data type.
  - Assume induction principle.
  - Assume other axioms as needed.
- Why such design decision?
  - Seems intuitive and convenient.
  - Hard to prove certain principles from ground up.

Introduction

- How to implement the theorem proving feature?
  - Build-in user defined data type.
  - Assume induction principle.
  - Assume other axioms as needed.
- Why such design decision?
  - Seems intuitive and convenient.
  - Hard to prove certain principles from ground up.
- What is the cost?
  - Complicated execution model for program.
  - Not obvious to see the proof system is consistent.

Introduction: Thesis

- Basic assumptions.
  - Lambda calculus.
  - Higher order quantified minimal logic $(\rightarrow, \forall)$.
  - Comprehension principle.
  - Extensionality.

Introduction: Thesis

- Basic assumptions.
  - Lambda calculus.
  - Higher order quantified minimal logic $(\to, \forall)$.
  - Comprehension principle.
  - Extensionality.
- Derivations.
  - Algebraic data.
  - $(\vee, \wedge, \exists)$ fragment.
  - Induction principle(including strong induction).
  - Principle of explosion.

Outline

- Introduction
- An Attempt to Expressiveness through Internalization.
  A Framework for Internalizing Relations into Type Theory. Peng Fu, Aaron Stump, Jeff Vaughan. PSATTT'11
- Lambda Encodings with Dependent Type.
  Self Types for Dependently Typed Lambda Encodings. Peng Fu, Aaron Stump. RTA-TLCA 2014
- Lambda Encoding with Comprehension.
- Implementation and Future Improvements.

The Internalization Approach

- Types $T \ ::= X \mid T \to T' \mid \Pi x : T.T' \mid \forall X.T$

The Internalization Approach

- Types $T \ ::= X \mid T \to T' \mid \Pi x : T.T' \mid \forall X.T$
- Equality between terms, subtype and term-type membership.

The Internalization Approach

- Types $T \ ::= X \mid T \to T' \mid \Pi x : T.T' \mid \forall X.T$
- Equality between terms, subtype and term-type membership.
- Extend types
  $F \ ::= \ X \mid F \to F' \mid \Pi x : F.F' \mid \forall X.F \mid t = t' \mid t \epsilon T \mid T <: T'$

The Internalization Approach

Additional rules:

$$\frac{t_1 = t_2 \in D}{\Gamma \vdash \textsf{EqAxiom} : t_1 = t_2}$$

$$\frac{\Gamma \vdash t : [t_1/x](t_3 = t_4) \quad \Gamma \vdash t' : t_1 = t_2}{\Gamma \vdash t : [t_2/x](t_3 = t_4)}$$

$$\frac{t \epsilon T' \in D}{\Gamma \vdash \textsf{MembAxiom} : t \epsilon T'}$$

$$\frac{\Gamma \vdash t : T \quad \Gamma \vdash t' : t \epsilon T'}{\Gamma \vdash t : T'}$$

The Internalization Approach

- Proved weak normalization.

The Internalization Approach

- Proved weak normalization.
- Limitations.
  - Additional typing rules for each new type.
  - Semantics is not modular.
  - Type preservation fails.

The Internalization Approach

- Proved weak normalization.
- Limitations.
  - Additional typing rules for each new type.
  - Semantics is not modular.
  - Type preservation fails.
- What we learned: reify meta-level relation as type.

Outline

- Introduction
- An Attempt to Expressiveness through Internalization.

  A Framework for Internalizing Relations into Type Theory. Peng Fu, Aaron Stump, Jeff Vaughan. PSATTT'11
- Lambda Encodings with Dependent Type.

  Self Types for Dependently Typed Lambda Encodings. Peng Fu, Aaron Stump. RTA-TLCA 2014
- Lambda Encoding with Comprehension.
- Implementation and Future Improvements.

Lambda Encodings with Dependent Type

- Motivations:
  - All dependent type systems include datatype.
  - Surprisingly daunting to formalize datatype system.

Lambda Encodings with Dependent Type

- Motivations:
  - All dependent type systems include datatype.
  - Surprisingly daunting to formalize datatype system.
- On the other hand.
  - Church encoding, Parigot encoding and Scott encoding.
  - Church encoding is available in System $\mathbf{F}$

Lambda Encodings with Dependent Type

Why not use Church encoded data?

- Inefficient to retrieve subdata.
- Can not prove $0 \neq 1$.
- Induction principle is not derivable.

Church Encoding: Inefficiency

- Church numerals: $\bar{0} := \lambda s.\lambda z.z, \mathsf{S} := \lambda n.\lambda s.\lambda z.s\ (n\ s\ z)$
  $\bar{3}\ := \lambda s.\lambda z.s\ (s\ (s\ \ z))$
- Linear time predecessor for Church numerals.
  $\mathsf{pred}\ n := \mathsf{fst}\ (n\ (\lambda p.(\mathsf{snd}\ p, \mathsf{S}\ (\mathsf{snd}\ p)))\ (0, 0))$
- Parigot numerals: $\bar{0} := \lambda s.\lambda z.z, \mathsf{S} := \lambda n.\lambda s.\lambda z.s\ \mathsf{n}\ (n\ s\ z)$
  $\bar{3}\ := \lambda s.\lambda z.s\ \bar{2}(s\ \bar{1}(s\ \bar{0}\ z))$
- Constant time Parigot predessesor.
  $\mathsf{pred}_p\ n = n\ (\lambda x.\lambda y.x)\ 0$

Church Encoding: Underivability of $0 \neq 1$

- Calculus of Construction($\mathbf{CC}$)

$$
\begin{array}{lcl}
x =_A y & := & \Pi C : A \to *.C\ x \to C\ y \\
\bot & := & \Pi X : *.X \\
0 =_{\mathsf{Nat}} 1 \to \bot & := & (\Pi C : \mathsf{Nat} \to *.C\ 0 \to C\ 1) \to \Pi X : *.X
\end{array}
$$

Church Encoding: Underivability of $0 \neq 1$

- Calculus of Construction($\mathbf{CC}$)

$$
\begin{array}{lcl}
x =_A y & := & \Pi C : A \to *. C\, x \to C\, y \\
\bot & := & \Pi X : *. X \\
0 =_{\mathsf{Nat}} 1 \to \bot & := & (\Pi C : \mathsf{Nat} \to *. C\, 0 \to C\, 1) \to \Pi X : *. X
\end{array}
$$

- $0 =_{\mathsf{Nat}} 1 \to \bot$ is underivable.
  - $\vdash_{cc} t : 0 \neq_{\mathsf{Nat}} 1$ implies $\vdash_F |t| : |0 \neq_{\mathsf{Nat}} 1|$
  - $|0 =_{\mathsf{Nat}} 1 \to \bot| := \Pi C.(C \to C) \to \Pi X.X$ in $\mathbf{F}$.

Church Encoding: Underivability of $0 \neq 1$

- Calculus of Construction($\mathbf{CC}$)

$$
\begin{array}{lcl}
x =_A y & := & \Pi C : A \to *.C\ x \to C\ y \\
\bot & := & \Pi X : *.X \\
0 =_{\mathsf{Nat}} 1 \to \bot & := & (\Pi C : \mathsf{Nat} \to *.C\ 0 \to C\ 1) \to \Pi X : *.X
\end{array}
$$

- $0 =_{\mathsf{Nat}} 1 \to \bot$ is underivable.
  - $\vdash_{cc} t : 0 \neq_{\mathsf{Nat}} 1$ implies $\vdash_F |t| : |0 \neq_{\mathsf{Nat}} 1|$
  - $|0 =_{\mathsf{Nat}} 1 \to \bot| := \Pi C.(C \to C) \to \Pi X.X$ in $\mathbf{F}$.
- Reason: Principle of explosion.

Church Encoding: Underivability of $0 \neq 1$

- Calculus of Construction:

$$
\begin{array}{lcl}
x =_A y & := & \Pi C : A \to *.C\ x \to C\ y \\
\boxed{\perp} & := & \Pi A : *.\Pi x : A.\Pi y : A.x =_A y \\
0 =_{\mathsf{Nat}} 1 \to \boxed{\perp} & := & (\Pi C : \mathsf{Nat} \to *.C\ 0 \to C\ 1) \\
 & & \to (\Pi A : *.\Pi x : A.\Pi y : A.x =_A y)
\end{array}
$$

- $\boxed{\perp}$ is uninhabited in $\mathbf{CC}$.
- $0 =_{\mathsf{Nat}} 1 \to \boxed{\perp}$ is derivable in $\mathbf{CC}$.
- Weak principle of explosion.

Church Encoding: Underivability of Induction Principle

- Induction is expressable in **CC**.
  $\Pi P : \mathsf{Nat} \to *.(\Pi y : \mathsf{Nat}.(P y \to P(\mathsf{S} y))) \to P\ \bar{0} \to \Pi x : \mathsf{Nat}.P\ x$.

[1] Metamathematical investigations of a calculus of constructions, T. Coquand.

Church Encoding: Underivability of Induction Principle

- Induction is expressable in $\mathbf{CC}$.
  $\Pi P : \mathsf{Nat} \to *.(\Pi y : \mathsf{Nat}.(Py \to P(\mathsf{S}y))) \to P\ \bar{0} \to \Pi x : \mathsf{Nat}.P\ x$.
- Induction is not provable in $\mathbf{CC}$ [1].

[1] Metamathematical investigations of a calculus of constructions, T. Coquand.

Church Encoding: Underivability of Induction Principle

- Induction is expressable in $\mathbf{CC}$.
  $\Pi P : \mathsf{Nat} \to *.(\Pi y : \mathsf{Nat}.(P y \to P(\mathsf{S} y))) \to P\ \bar{0} \to \Pi x : \mathsf{Nat}.P\ x$.
- Induction is not provable in $\mathbf{CC}$ [1].
- Self Type: $\iota x.F$.
  $\dfrac{\Gamma \vdash t : \iota x.F}{\Gamma \vdash t : [t/x]F} \quad \dfrac{\Gamma \vdash t : [t/x]F}{\Gamma \vdash t : \iota x.F}$

[1] Metamathematical investigations of a calculus of constructions, T. Coquand.

Church Encoding: Underivability of Induction Principle

- Induction is expressable in $\mathbf{CC}$.
  $\Pi P : \mathsf{Nat} \to *.(\Pi y : \mathsf{Nat}.(Py \to P(\mathsf{S}y))) \to P\ \bar{0} \to \Pi x : \mathsf{Nat}.P\ x$.
- Induction is not provable in $\mathbf{CC}$ [1].
- Self Type: $\iota x.F$.
  $$\frac{\Gamma \vdash t : \iota x.F}{\Gamma \vdash t : [t/x]F} \qquad \frac{\Gamma \vdash t : [t/x]F}{\Gamma \vdash t : \iota x.F}$$
- We also need recursive definition.
  $\mathsf{Nat} :=$
  $\iota x.\Pi P : \mathsf{Nat} \to *.(\Pi y : \mathsf{Nat}.(Py \to P(\mathsf{S}y))) \to P\ \bar{0} \to P\ x$

[1] Metamathematical investigations of a calculus of constructions, T. Coquand.

Church Encoding: Underivability of Induction Principle

- Induction is expressable in $\mathbf{CC}$.
  $\Pi P : \mathsf{Nat} \to *.(\Pi y : \mathsf{Nat}.(Py \to P(\mathsf{S}y))) \to P\ \bar{0} \to \Pi x : \mathsf{Nat}.P\ x$.
- Induction is not provable in $\mathbf{CC}$ [1].
- Self Type: $\iota x.F$.
  $$\frac{\Gamma \vdash t : \iota x.F}{\Gamma \vdash t : [t/x]F} \qquad \frac{\Gamma \vdash t : [t/x]F}{\Gamma \vdash t : \iota x.F}$$
- We also need recursive definition.
  $\mathsf{Nat} :=$
  $\iota x.\Pi P : \mathsf{Nat} \to *.(\Pi y : \mathsf{Nat}.(Py \to P(\mathsf{S}y))) \to P\ \bar{0} \to P\ x$
- $\bar{0} : \mathsf{Nat}, \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}$

[1] Metamathematical investigations of a calculus of constructions, T. Coquand.

Church Encoding: Underivability of Induction Principle

- Induction is expressable in $\mathbf{CC}$.
  $\Pi P : \mathsf{Nat} \to *.(\Pi y : \mathsf{Nat}.(Py \to P(\mathsf{S}y))) \to P\ \bar{0} \to \Pi x : \mathsf{Nat}.P\ x$.
- Induction is not provable in $\mathbf{CC}$ [1].
- Self Type: $\iota x.F$.
  $\dfrac{\Gamma \vdash t : \iota x.F}{\Gamma \vdash t : [t/x]F} \quad \dfrac{\Gamma \vdash t : [t/x]F}{\Gamma \vdash t : \iota x.F}$
- We also need recursive definition.
  $\mathsf{Nat} :=$
  $\iota x.\Pi P : \mathsf{Nat} \to *.(\Pi y : \mathsf{Nat}.(Py \to P(\mathsf{S}y))) \to P\ \bar{0} \to P\ x$
- $\bar{0} : \mathsf{Nat}, \mathsf{S} : \mathsf{Nat} \to \mathsf{Nat}$
- Induction now is derivable.
  $ind := \lambda s.\lambda z.\lambda n.n\ s\ z$.

---

[1] Metamathematical investigations of a calculus of constructions, T. Coquand.

Lambda Encodings with Dependent Type

Summary

- $0 \neq 1$ is provable with a change of notion of contradiction.
- Introduce Self type to derive induction principle.
- Devised a type system called $\mathbf{S}$.
- We prove $\mathbf{S}$ is consistent and type preserving.

Outline

- Introduction
- An Attempt to Expressiveness through Internalization.
  A Framework for Internalizing Relations into Type Theory. Peng Fu, Aaron Stump, Jeff Vaughan. PSATTT'11
- Lambda Encodings with Dependent Type.
  Self Types for Dependently Typed Lambda Encodings. Peng Fu, Aaron Stump. RTA-TLCA 2014
- Lambda Encoding with Comprehension.
- Implementation and Future Improvements.

Lambda Encoding with Comprehension

Motivation.

- Understand self type.
- Reason about Scott numerals.
  - Direct translation from functional program to Scott-encoded lambda term.
  - Resistance to intuitionistic typing.

Lambda Encoding with Comprehension

- Self type mechanism:
  $\Gamma \vdash t : \iota x.F \Leftrightarrow \Gamma \vdash t : [t/x]F$

Lambda Encoding with Comprehension

- Self type mechanism:
  $\Gamma \vdash t : \iota x.F \Leftrightarrow \Gamma \vdash t : [t/x]F$
- Comprehension:
  $t \epsilon \{x \mid F[x]\} \Leftrightarrow F[t]$

Lambda Encoding with Comprehension

- Self type mechanism:
  $\Gamma \vdash t : \iota x.F \Leftrightarrow \Gamma \vdash t : [t/x]F$
- Comprehension:
  $t \epsilon \{x \mid F[x]\} \Leftrightarrow F[t]$
- What if?
  $t \epsilon (\iota x.F[x]) \Leftrightarrow F[t]$

Lambda Encoding with Comprehension

- Self type mechanism:
  $\Gamma \vdash t : \iota x.F \Leftrightarrow \Gamma \vdash t : [t/x]F$
- Comprehension:
  $t \epsilon \{x \mid F[x]\} \Leftrightarrow F[t]$
- What if?
  $t \epsilon (\iota x.F[x]) \Leftrightarrow F[t]$
- In the work of internalization.
  $t \epsilon T$ as type

Lambda Encoding with Comprehension

- Self type mechanism:
  $\Gamma \vdash t : \iota x.F \Leftrightarrow \Gamma \vdash t : [t/x]F$
- Comprehension:
  $t \epsilon \{x \mid F[x]\} \Leftrightarrow F[t]$
- What if?
  $t \epsilon (\iota x.F[x]) \Leftrightarrow F[t]$
- In the work of internalization.
  $t \epsilon T$ as type
- $t \epsilon (\iota x.F[x])$ as a formula.

Lambda Encoding with Comprehension

System $\mathfrak{G}$ in six rules.

$$\frac{(a : F) \in \Gamma}{\Gamma \vdash a : F}$$

$$\boxed{\frac{\Gamma \vdash p : F_1 \quad F_1 \cong F_2}{\Gamma \vdash \mathrm{cmp}\ p : F_2}}$$

$$\frac{\Gamma \vdash p : F \quad \alpha \notin \mathrm{FV}(\Gamma)}{\Gamma \vdash \mathrm{ug}\ \alpha\ .\ p : \forall \alpha . F}$$

$$\frac{\Gamma \vdash p : \forall \alpha . F}{\Gamma \vdash \mathrm{inst}\ p\ \mathrm{by}\ Q : [Q/\alpha]F}$$

$$\frac{\Gamma, a : F_1 \vdash p : F_2}{\Gamma \vdash \mathrm{discharge}\ a : F_1\ .\ p : F_1 \to F_2}$$

$$\frac{\Gamma \vdash p : F_1 \to F_2 \quad \Gamma \vdash p' : F_1}{\Gamma \vdash \mathrm{mp}\ p\ \mathrm{by}\ p' : F_2}$$

System $\mathfrak{G}$: Basic Assumptions

$$\frac{\Gamma \vdash p : F_1 \quad F_1 \cong F_2}{\Gamma \vdash \text{cmp } p : F_2}$$

- Lambda conversion:
  $(\lambda x.t)t' =_\beta [t'/x]t$
- Axiom of extension:
  $F[t_1] \cong F[t_2]$ if $t_1 =_\beta t_2$
- Comprehension Axiom:
  $t\epsilon(\iota x.F) \cong [t/x]F$

System $\mathfrak{G}$: Results

- Relatively easy to prove consistency.
- Subject reduction (type preservation).
- Proved Peano's 9 axioms inside $\mathfrak{G}$.
- Derived Strong induction within $\mathfrak{G}$.
- Ability to reason about Scott encodings.
- Ability to reason about possibly diverging functions.

Outline

- Introduction
- An Attempt to Expressiveness through Internalization.
  A Framework for Internalizing Relations into Type Theory. Peng Fu, Aaron Stump, Jeff Vaughan. PSATTT'11
- Lambda Encodings with Dependent Type.
  Self Types for Dependently Typed Lambda Encodings. Peng Fu, Aaron Stump. RTA-TLCA 2014
- Lambda Encoding with Comprehension.
- Implementation and Future Improvements.

Gottlob: Implemented Features

- 2800 lines of Haskell code(without comments).
- Inferred formula from proof.
- Hindley-Milner polymorphic type inference for program.
- Program is translated to Scott encoded lambda term.
- Automatically derive(and proof-check) induction principle.
- User-defined proof tactic: automated generate proofs.

Gottlob: Examples

```
tactic cmpinst p s = cmp inst p by s

tactic id F =  discharge a : F . a

div n m = case n < m of
                true -> zero
                false -> succ (div (n - m) m)

theorem divi. forall n m . n :: Nat -> m :: Nat ->
Le zero m -> Le (div n m) n <+> Eq (div n m) n

theorem assoc . forall a1 a2 a3 U . a1 :: List U ->
         Eq ((a1 ++ a2) ++ a3) (a1 ++ (a2 ++ a3))
```

Gottlob: A Taste of Proof

```
Eq a b = forall C . a :: C -> b :: C
theorem trans. forall a b c . Eq a b -> Eq b c -> Eq a c
proof
        [m1] : Eq a b
        [m2] : Eq b c
        [m3] : a :: C
        d1 = inst cmp m1 by C -- : a :: C -> b :: C
        d2 = mp d1 by m3 -- : b :: C
        d3 = inst cmp m2 by C  -- : b :: C -> c :: C
        d4 = mp d3 by d2 -- : c :: C
        d5 = invcmp ug C. discharge m3 . d4 : Eq a c
        d6 = ug a . ug b . ug c . discharge m1 .
            discharge m2 . d5
qed
```

Gottlob: Surface

```
data List A where
   nil :: List A
   cons :: A -> List A -> List A
  deriving Ind

(++) nil l = l
(++) (cons u l') l = cons u (l'++ l)
```

Gottlob: Behind the Scene

```
nil = \ nil . \ cons . nil
cons = \ a2 . \ a1 . \ nil . \ cons . cons a2 a1
(++) = \ u1 . \ u2 . u1 u2 (\ u3 . \ u4 . cons u3 ((++)
    u4 u2))
(++) :: forall A . List A -> List A -> List A
List : (i -> o) -> i -> o =
iota U . iota x . forall List . nil :: List U ->
(forall x . x :: U -> forall x0 . x0 :: List U -> cons x
     x0 :: List U) -> x :: List U
IndList : o =
forall U . forall List0 . nil :: List0 U -> (forall x .
    x :: U -> forall x0 . x0 :: List0 U -> cons x x0 ::
    List0 U) -> forall x . x :: List U -> x :: List0 U
```

Gottlob: Future Improvements

- More case studies.
- Usability (find opportunity to automate proof).
- Compilation or a REPL like environment.

Thank You!

- My advisor Prof. Aaron Stump.
- My dissertation committee: Prof. Cesare Tinelli, Prof. Kasturi Varadarajan, Prof. Ted Herman, Prof. Douglas Jones.
- All the audiences.