Programming and Proving with Dependently Typed Lambda Encodings

Aaron Stump
Computer Science
The University of Iowa
Iowa City, Iowa
Type theory based on lambda encodings

- **Streamline** the type theory
  - All data are lambda-encoded
  - No primitive constructors, pattern-matching
  - Simpler meta-theory for the type theory

- **Expand** its range with higher-order encodings
  - Inductive datatypes require (strict) positivity in Coq/Agda
  - With lambda-encodings, can go negative!

Where $\text{trm} t \equiv x \mid t \mid \lambda x.t$

**Encoded as**

\[
\text{trm} = \forall X : \ast. (X \rightarrow X \rightarrow X) \rightarrow ((X \rightarrow X) \rightarrow X) \rightarrow X
\]

- May enable simpler meta-theory for object languages
- New range of tools to apply
Calculus of Dependent Lambda Eliminations (CDLE)

- Like Calculus of Constructions plus
  - functions with erased arguments (specificational, type is $\forall x : T. T'$)
  - constructor-constrained recursive types $\nu X : \kappa | \Theta. T$
  - lifting operator $\uparrow_L t$

- Semantics for types, consistency proof

- Realizability-inspired type assignment rules

\[
\Gamma \vdash t : T \quad \Gamma \vdash t : [t/x]T' \\
\Gamma \vdash t : \nu X : T. T'
\]

\[
\Gamma \vdash t' : T \quad t =_{\beta} t' \\
\Gamma \vdash t : T
\]

- Problem: how to implement?
First attempt: evidential typing \( \Gamma \vdash e :: t : T \)

Unworkably painful!

Insight: problematic rules used mainly for kinding \( \nu \)-types

Special-case treatment of those

Dependent intersection types \( \nu x : T. T' \) completely hidden

Cedille implementation
  
  ▶ New (wrote over winter break)
  ▶ Around 2800 lines of Agda
  ▶ 180-line grammar
  ▶ around 300 lines of elisp
Datatypes in Cedille

- Can always use definitions as in System $F_\omega$

  $$\text{CNat} \iff \star = \forall \; X : \star . \; (X \to X) \to X \to X .$$

  - For these you get lifting
  - But no dependent eliminations/induction
  - Can be negative
  - Church-encoding only

- Can declare top-level recursive types

  $$\text{rec Nat} \mid S : \text{Nat} \to \text{Nat} , \; Z : \text{Nat} = \forall \; P : \text{Nat} \to \star . \;
  \left( \prod n : \text{Nat} . \; P n \to P (S n) \right) \to P Z \to P \text{ self}$$

  with

  $$S = \lambda n . \; \Lambda \; P . \; \lambda s . \; \lambda z . \; s n (n \cdot P s z) , \;
  Z = \Lambda \; P . \; \lambda s . \; \lambda z . \; z.$$

  - Lifting, dependent eliminations
  - Must be positive-recursive
  - All encodings supported (Church, Parigot, others)
Recursive types in Cedille

\[
\text{rec Nat | } S : \text{Nat} \to \text{Nat} , \ Z : \text{Nat} = \ \\
\forall P : \text{Nat} \to * . \ \\
(\Pi n : \text{Nat} . \ P n \to P (S n)) \to P Z \to P \text{self} \ \\
\text{with} \ \\
S = \lambda n . \ \Lambda P . \ \lambda s . \ \lambda z . \ s n (n \cdot P s z) , \ \\
Z = \Lambda P . \ \lambda s . \ \lambda z . \ z .
\]

- \text{self is implicitly } \iota\text{-bound}
- First declare constructors
- Then define them
- For definition \( x = t \), type-check \( t \) under assumption \( x = t \)
- Afterwards, \( \iota \)-type instantiated immediately on use
Rule induction

- Not all inductions are dependent eliminations(!)
  - Dependent elimination with $x$ proves $P \ x$
  - If $P$ does not depend on $x$, this is overkill
- PL meta-theory mostly uses rule induction

**Theorem (Type Preservation)**

If $\Gamma \vdash t : T$ and $t \leadsto t'$ then $\Gamma \vdash t' : T$.

- Can Church-encode the derivations
- But if you need inversion, should use Parigot
Emacs mode

- Cedille has an emacs mode for editing Cedille files
- Based on a generic structured-editing mode by Carl Olson

A span is \([\text{label}, \text{start-pos}, \text{end-pos}, \text{attributes}]\)
- Spans communicated in JSON
- Cedille sends all type information, in span attributes
- Monadic style for writing the backend (type checker)
Type checking

- Terms are annotated
  - $\Lambda x : \kappa . t$ for type abstraction
  - $t \cdot T$ for type instantiation
  - $\Lambda x : T . t$ for erased-term abstraction
  - $t - t'$ for erased-term application

- Use local type inference
  - Checking $t \leftarrow T$
  - Synthesizing $t \rightarrow T$
  - Can drop some annotations ($\lambda x . t$ instead of $\lambda x : T . t$)

- Conversion relation
  - Check if types are $\beta$-equivalent
  - All annotations erased from terms
Equality types

- $t \simeq t'$ means $t$ is $\beta$-equivalent to $t'$
- Term constructs for equational reasoning
  - For $\epsilon \ t \leftarrow t_1 \simeq t_2$, head-normalize $t_1$ and $t_2$ and check against $t$
  - For $\epsilon \ t \Rightarrow$, synthesize $t_1 \simeq t_2$ from $t$ and head-normalize the sides
  - For $\rho \ t \leftarrow t' \leftarrow T$, synthesize $t_1 \simeq t_2$ from $t$ and rewrite $t_1$ to $t_2$ in $T$ before checking $t'$
  - For $\rho \ t \leftarrow t' \Rightarrow$, synthesize $t_1 \simeq t_2$ from $t$ and $T$ from $t'$, then rewrite $t_1$ to $t_2$ in $T$
  - ...
  - $\delta \ t \leftarrow T$ succeeds if $t$ synthesizes an obviously impossible equation
- Head normalization seems helpful for controlling reduction
A couple higher-order examples

1. Closedness preservation
   1. Define untyped $\lambda$-terms with free variables
   2. Define $\beta$-reduction
   3. Prove that reduction cannot introduce a free variable

2. Type preservation for STLC
   1. Define annotated terms, reduction
   2. Define typing algorithm
   3. Prove preservation
Cedille in action
Conclusion

- Cedille implementation of Calc. of Dep. Lambda Eliminations
  - Top-level datatype definitions with ctor-constrained rec. types
  - Emacs mode with structured editing
  - Equality types and rewriting

- Next direction: exploring higher-order encodings
  - Cedille is first dependent type theory supporting these
  - Could greatly simplify meta-programming/-proving
  - Uncharted territory...
    - Non-dependent encodings ($F_\omega$)
    - Dependent ones ($\nu X : \kappa | \Theta. T$)
    - Computing types from terms
    - Algorithmic definitions
Conclusion

- Cedille implementation of Calc. of Dep. Lambda Eliminations
  - Top-level datatype definitions with ctor-constrained rec. types
  - Emacs mode with structured editing
  - Equality types and rewriting

- Next direction: exploring higher-order encodings
  - Cedille is first dependent type theory supporting these
  - Could greatly simplify meta-programming/-proving
  - Uncharted territory...
    - Non-dependent encodings ($F_\omega$)
    - Dependent ones ($\nu X : \kappa | \Theta. T$)
    - Computing types from terms
    - Algorithmic definitions

Thank you!