ΠΣ

a core language for dependently typed programming

work in suspended progress

thanks to Nicolas Oury
and Nils Anders Danielsson

Thorsten Altenkirch

School of Computer Science
University of Nottingham

May 14, 2009
Motivation

- Complexity of total dependently typed languages like Agda, CIC, Epigram, ... 
- Implicit arguments, schemes for inductive families, pattern matching, coinductive types, termination analysis, universes, ... 
- Goal: small core language 
- Develop metatheory (formally) 
- Stable core language 
- Address issues, e.g. corecursion and induction-recursion
Partial language with full dependent types
Core language: doesn’t address high level features
(e.g. implicit arguments, convenient pattern matching, ...)
Structural instead of nominal type system
(recursive defns are first class)
Mutual (local) recursive definitions
Constructs to control unfolding of recursion
Few type constructors:
- \textbf{Type : Type}
- \Pi-types
- \Sigma-types
- Finite types (sets of labels)
- Equality types
Eliminated constraints (too complicated) in favour of
equality types.

$\alpha$-equivalence of recursive definitions

Different approach to control unfolding
(omitted boxes and lifting types).

New implementation (unfinished).
Related work

- Lennart Augustsson’s *Cayenne*
  - 1998
  - Partial and fully dependent
  - but not a core language

- Thierry Coquand’s *Calculus of Definitions*
  - 2008-
  - similar to ΠΣ
  - Nominal approach to recursion
Type-checking is undecidable
Logically inconsistent
Type unsound after eliminating proofs
Basic mechanisms independent of totality
Separate definition of total sublanguages
Totality checker
Optimisations and guarantees depend on totality analysis
ΠΣ by example: Natural numbers

Nat : Type
Nat = l : { z, s }

* case l of
  z → { unit }
  s → .Nat

zero : Nat
zero = ( z, unit )

succ : Nat → Nat
succ = λn → ( s, n )

Labelled sums are encoded using Σ-types and enumerations.
Recursive types use general recursion.
Recursive variables can be marked with . to stop infinite unfolding.
Lazy addition

\[\text{add} : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}\]
\[\text{add} = \lambda m \ n \rightarrow \text{split} \ m \ \text{with} \ (l, m') \]
\[\rightarrow \text{case} \ l \ \text{of} \]
\[z \rightarrow n\]
\[| \ s \rightarrow \text{succ} \ (.\text{add} \ m' \ n)\]

- Eliminators \text{split}, \text{case} can only analyze variables to support dependent elimination.

- We use the same recursion mechanism as for the definition of recursive types.

- This definition of \text{add} is lazy, we have that
  \[\text{add} \ (\text{succ} \ m) \ n = \text{succ} \ (.\text{add} \ m' \ n)\]
  but not
  \[\text{add} \ 2 \ 1 = 3\]
Eager addition

\[ add : \text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat} \]
\[ add = \lambda m \ n \rightarrow \text{split} \ m \ \text{with} \ (l, m') \]
\[ \rightarrow !\text{case} \ l \ \text{of} \]
\[ z \rightarrow n \]
\[ | \ s \rightarrow \text{succ} (.add m' \ n) \]

! forces delayed recursive definitions.
\[ add (\text{succ} \ m) \ n = !\text{succ} (.add m' \ n) = \text{succ} (add m' \ n) \]
and hence \( add \ 2 \ 1 = 3 \)

Eager vs lazy correspond to inductive vs coinductive types.
Vectors, recursively

\[ Vec : Type \to Nat \to Type \]
\[ Vec = \lambda A \ n \to \text{split } n \text{ with } (l, n') \]
\[ \to ! \text{case } l \text{ of} \]
\[ \begin{align*}
  z & \to \{ \text{unit} \\
  s & \to A \star . Vec A \ n'
\end{align*} \]

- Recursive programs (like `vappend`) have to analyze the indices.
- Doesn’t extend easily for more complicated types (like typed \( \lambda \) terms).
Vectors using equality

\[ Vec : \textbf{Type} \rightarrow \textbf{Nat} \rightarrow \textbf{Type} \]
\[ Vec = \lambda A \ n \rightarrow l : \{ \text{nil}, \text{cons} \} \]
\* \texttt{case} \ l \ \texttt{of} \]
\* \texttt{nil} \rightarrow \{ \texttt{unit} \}
\* \texttt{cons} \rightarrow A \ast n' : \textbf{Nat} \ast \textbf{Vec} A n' \ast n = \texttt{succ} n'

- Seems suitable to encode inductive families ala Agda and dependently typed pattern matching
- Need equality types. What is the right way to do them?
Introduction

ΠΣ by example

Design details

Conclusions?

Induction - recursion

\[ U : \text{Type} \]
\[ El : U \rightarrow \text{Type} \]
\[ U = l : \{ \text{nat}, \pi \} \]
\[ \star \text{case } l \text{ of} \]
\[ \quad \text{nat} \rightarrow \{ \text{unit} \} \]
\[ \quad | \text{pi} \rightarrow a : .U \star El a \rightarrow \text{Type} \]
\[ El = \lambda a \rightarrow \text{split } a \text{ with } (l, a') \]
\[ \rightarrow !\text{case } l \text{ of} \]
\[ \quad \text{nat} \rightarrow \text{Nat} \]
\[ \quad | \text{pi} \rightarrow \text{split } a' \text{ with} \]
\[ \quad \quad (b, f) \rightarrow (x : .El b) \rightarrow .El (f x) \]
Mutual inductive definitions

- Let bindings are a sequence of
  - Declarations
    \[ x : \sigma \]
  - Definitions
    \[ x = t \]
  given that \( \sigma : \text{Type} \).

- In \( \Pi\Sigma \) any sequence of declarations is allowed, as long as:
  - Every definition, declaration type checks with respect to the definitions and declarations made before.
  - Every variable which is declared, will be defined.
Delayed variables

**Before:** boxed definitions to delay evaluation

\[
\begin{align*}
  a &: A \\
  [a] &: A \bot \\
  a' &: A \bot \\
  ![a] &: A
\end{align*}
\]

- \([a]\) stops unfolding of recursive definitions from outside.
- Too inflexible, freezes too much.
  but Nisse seems to like it . . .
- **Now:** only variables are delayed
  (.\(x\) instead of \([x]\)).
- Invisible for the type system (no \(A \bot\)).
- Forcing propagates during evaluation, e.g.

\[
!(t, u) \equiv !t, !u
\]
Equality of definitions

- Evaluating
  \[
  \text{let } \Gamma \text{ in } x
  \]
  returns a closure.
- How to we compare these closures?
Equality quiz

- **let** \( x : Nat, x = 3 \) **in**. \( x \equiv 3 \)?
  - No, we would have to unfold a frozen variable.

- **let** \( x : Nat, x = 3 \) **in** \( 4 \equiv 4 \)?
  - Yes, sure \( 4 \equiv 4 \).

- **let** \( x : Nat, x = 3 \) **in** \( x \equiv \) **let** \( x : Nat, x = 3 \) **in** \( x \)?
  - Yes, sure \( x \equiv x \) in the same context.

- **let** \( x : Nat, x = 3 \) **in** \( x \equiv \) **let** \( x : Nat, x = 4 \) **in** \( x \)?
  - No, this would be unsound!

- **let** \( x : Nat, x = 3 \) **in** \( x \equiv \) **let** \( y : Nat, y = 4 \) **in** \( y \)?
  - Yes, there are the same up to \( \alpha \)-conversion.
Equivalence of recursive definitions (Nicolas Oury)

- How do we decide that

\[ \text{let } \Gamma \text{ in } x \equiv \text{let } \Delta \text{ in } y? \]

- We look up

\[ t = \Gamma (x) \]
\[ u = \Delta (y) \]

- We associate both \( x \) and \( y \) with the same fresh variable \( z \) (by modifying the environment).

- We continue comparing

\[ t \equiv u \]
Equivalence of recursive definitions

- How to specify this equivalence on the level of rules?

\[
\text{let } \Gamma \\text{ in } t \equiv \text{let } \Delta \\text{ in } u
\]

- Given a partial bijection

\[
\phi : \text{FV } \Gamma \simeq \text{FV } \Gamma
\]

- We require that all definitions and the body are equivalent upto \( \phi \):

\[
\forall x \phi y. \Gamma(x) \simeq_\phi \Delta(x)
\]

\[
t \simeq_\phi u
\]

- In general we have to define \( \alpha \)-equivalence upto a partial bijection of free variables.
Metatheoretic properties (desired)

- Type soundness
- Sound wrt typing rules
- Partial completeness
What is next?

- Finish implementation!
- How to delay unfolding?
- Which equality to use (intensional, Observational Type Theory)?
- Establish metatheoretic properties.
What is later?

- Translate high-level features
  ((co)datatype declarations, pattern matching, implicit variables, . . . )
- Implement in Agda.
- Totality checker