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

ΠΣ in a nutshell

- Partial language with full dependent types
- Core language : doesn't address high level featues (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:
  - **Type** : **Type**
  - Π-types
  - Σ-types
  - Finite types (sets of labels)
  - Equality types

Changes since last time

- Eliminated constraints (too complicated) in favour of equality types.
- α-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

Partial?

- Type-checking is undecidable
- Logically inconsistent
- Type unsound after eliminating proofs

Partial!

- Basic mechanisms independent of totality
- Separate definition of total sublanguages
- Totality checker
- Optimisations and guarantees depend on totality analysis

ΠΣ by example: Natural numbers

$$
\begin{array}{l}
\mathit{Nat} : \mathbf{Type} \\
\mathit{Nat} = \mathit{l} : \{z, s\} \\
\qquad * \ \mathbf{case}\ \mathit{l}\ \mathbf{of} \\
\qquad\qquad z \to \{\mathit{unit}\} \\
\qquad\qquad \mid s \to .\mathit{Nat}
\end{array}
$$

$$
\begin{array}{l}
\mathit{zero} : \mathit{Nat} \\
\mathit{zero} = (z, \mathit{unit})
\end{array}
$$

$$
\begin{array}{l}
\mathit{succ} : \mathit{Nat} \to \mathit{Nat} \\
\mathit{succ} = \lambda n \to (s, n)
\end{array}
$$

- 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

$$
\begin{array}{l}
\mathit{add} : \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
\mathit{add} = \lambda m\ n \to \textbf{split}\ m\ \mathit{with}\ (l, m') \\
\qquad\qquad \to \textbf{case}\ l\ \textbf{of} \\
\qquad\qquad\quad\ \ z \to n \\
\qquad\qquad\quad \mid\ s \to \mathit{succ}\ (.\mathit{add}\ m'\ n)
\end{array}
$$

- Eliminators **split**, **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 *add* is lazy, we have that
  $\mathit{add}\ (\mathit{succ}\ m)\ n = \mathit{succ}\ (.\mathit{add}\ m'\ n)$
  but not
  $\mathit{add}\ 2\ 1 = 3$

Eager addition

$$
\begin{aligned}
&\mathit{add} : \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
&\mathit{add} = \lambda m\ n \to \textbf{split}\ m\ \textit{with}\ (l, m') \\
&\qquad\qquad \to\ !\textbf{case}\ l\ \textbf{of} \\
&\qquad\qquad\qquad\quad z \to n \\
&\qquad\qquad\qquad \mid s \to \mathit{succ}\ (.\mathit{add}\ m'\ n)
\end{aligned}
$$

- ! forces delayed recursive definitions.
  $\mathit{add}\ (\mathit{succ}\ m)\ n = !\mathit{succ}\ (.\mathit{add}\ m'\ n) = \mathit{succ}\ (\mathit{add}\ m'\ n)$
  and hence $\mathit{add}\ 2\ 1 = 3$
- Eager vs lazy correspond to inductive vs coinductive types.

Vectors, recursively

$$
\begin{aligned}
&\mathit{Vec} : \mathbf{Type} \to \mathit{Nat} \to \mathbf{Type} \\
&\mathit{Vec} = \lambda A\ n \to \mathbf{split}\ n\ \mathit{with}\ (l, n') \\
&\qquad\qquad \to\ !\mathbf{case}\ l\ \mathbf{of} \\
&\qquad\qquad\quad z \to \{\mathit{unit}\} \\
&\qquad\qquad\quad s \to A * .\mathit{Vec}\ A\ n'
\end{aligned}
$$

- Recursive programs (like *vappend*) have to analyze the indices.
- Doesn't extend easily for more complicated types (like typed λ terms).

Vectors using equality

$$
\begin{array}{l}
\mathit{Vec} : \mathbf{Type} \to \mathit{Nat} \to \mathbf{Type} \\
\mathit{Vec} = \lambda A\ n \to I : \{\mathit{nil}, \mathit{cons}\} \\
\quad * \ \mathbf{case}\ I\ \mathbf{of} \\
\qquad \mathit{nil} \to \{\mathit{unit}\} \\
\qquad \mathit{cons} \to A * n' : \mathit{Nat} * \mathit{Vec}\ A\ n' * n = \mathit{succ}\ n'
\end{array}
$$

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

Induction - recursion

$$
\begin{array}{l}
U : \mathbf{Type} \\
El : U \to \mathbf{Type} \\
U = l : \{nat, pi\} \\
\quad * \ \mathbf{case}\ l\ \mathbf{of} \\
\qquad nat \to \{unit\} \\
\qquad \mid\ pi \to a : .U * El\ a \to \mathbf{Type} \\
El = \lambda a \to \mathbf{split}\ a\ with\ (l, a') \\
\quad \to\ !\mathbf{case}\ l\ \mathbf{of} \\
\qquad nat \to Nat \\
\qquad \mid\ pi \ \to \mathbf{split}\ a'\ with \\
\qquad\qquad (b, f) \to (x : .El\ b) \to .El\ (f\ x)
\end{array}
$$

Mutual inductive definitions

- Let bindings are a sequence of

  Declarations

  $$x : \sigma$$

  given that σ : **Type**.

  Definitions

  $$x = t$$

  given that $x : \sigma$ and $t : \sigma$.
- In ΠΣ 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

$$\frac{a : A}{[a] : A_\perp} \qquad \frac{a' : A_\perp}{!a' : A} \qquad ![a] \equiv a$$

- $[a]$ 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 $A_\perp$).
- Forcing propagates during evaluation, e.g.

$$!(t, u) \equiv !t, !u$$

Equality of definitions

- Evaluating

  $$\textbf{let}\ \Gamma\ \textbf{in}.x$$

  returns a closure.
- How to we compare these closures?

Equality quiz

- **let** $x : \mathit{Nat}, x = 3$ **in**.$x \equiv 3$?
  No, we would have to unfold a frozen variable.
- **let** $x : \mathit{Nat}, x = 3$ **in** $4 \equiv 4$?
  Yes, sure $4 \equiv 4$.
- **let** $x : \mathit{Nat}, x = 3$ **in**.$x \equiv$ **let** $x : \mathit{Nat}, x = 3$ **in**.x ?
  Yes, sure $x \equiv x$ in the same context.
- **let** $x : \mathit{Nat}, x = 3$ **in**.$x \equiv$ **let** $x : \mathit{Nat}, x = 4$ **in**.x ?
  No, this would be unsound!
- **let** $x : \mathit{Nat}, x = 3$ **in**.$x \equiv$ **let** $y : \mathit{Nat}, y = 4$ **in**.y ?
  Yes, there are the same upto α-conversion.

Equivalence of recursive definitions (Nicolas Oury)

- How do we decide that

$$\textbf{let}\ \Gamma\ \textbf{in}.x \equiv \textbf{let}\ \Delta\ \textbf{in}.y?$$

- We look up

$$\begin{aligned} t &= \Gamma\,(x) \\ u &= \Delta\,(y) \end{aligned}$$

- 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?

$$\textbf{let } \Gamma \textbf{ in } t \equiv \textbf{let } \Delta \textbf{ in } u$$

- Given a partial bijection

$$\phi : \mathrm{FV}\,\Gamma \simeq \mathrm{FV}\,\Gamma$$

- We require that all definitions and the body are equivalent upto ϕ:

$$\forall x \phi y . \Gamma(x) \simeq_\phi \Delta(x)$$

$$t \simeq_\phi u$$

- In general we have to define α-equivelence 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