ΠΣ: Dependent Types Without the Sugar

based on joint work with Nils Anders Danielsson, Andres Löh, Darin Morrison and Nicolas Oury

Thorsten Altenkirch

School of Computer Science
University of Nottingham

November 28, 2009

Agda is cool!

$$
\begin{array}{l}
\textbf{data}\ \mathit{Vec}\ (A : \mathit{Set}) : \mathbb{N} \to \mathit{Set}\ \textbf{where} \\
\quad [\,] \quad\ : \mathit{Vec}\ A\ \mathit{zero} \\
\quad \_ :: \_ : \{ n : \mathbb{N} \} \to A \to \mathit{Vec}\ A\ n \to \mathit{Vec}\ A\ (\mathit{suc}\ n)
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\ \mathit{Fin} : \mathbb{N} \to \mathit{Set}\ \textbf{where} \\
\quad \mathit{zero} : \{ n : \mathbb{N} \} \qquad\qquad\quad\ \to \mathit{Fin}\ (\mathit{suc}\ n) \\
\quad \mathit{suc}\ \ : \{ n : \mathbb{N} \} \to \mathit{Fin}\ n \to \mathit{Fin}\ (\mathit{suc}\ n)
\end{array}
$$

$$
\begin{array}{l}
\_\,!!\,\_ : \forall \{ A\ n \} \to \mathit{Vec}\ A\ n \to \mathit{Fin}\ n \to A \\
[\,] \qquad\quad !!\ () \\
(x :: xs)\ !!\ \mathit{zero} \quad\ = x \\
(x :: xs)\ !!\ (\mathit{suc}\ i) = xs\ !!\ i
\end{array}
$$

- Thanks to Ulf Norell!
- !! is statically safe, no out of range error.

The Witness Pattern

$$
\begin{aligned}
&\mathit{check} : (\Gamma : \mathit{Ctx}) \to (e : \mathit{Chk}) \to (\tau : \mathit{Type}\,) \\
&\qquad\quad \to (\Gamma \;\vdash\; e \downarrow \tau) \uplus (\Gamma \;\nvdash\; e \downarrow \tau) \\
&\mathit{synth} : (\Gamma : \mathit{Ctx}) \to (e : \mathit{Syn}) \\
&\qquad\quad \to \Sigma\ \mathit{Type}\ (\lambda\, \tau \to \Gamma \;\vdash\; e \uparrow \tau) \uplus (\Gamma \;\nvdash\; e\uparrow)
\end{aligned}
$$

- Darin Morrison implemented an *evidence carrying* type checker for simply typed λ-calculus.
- The uninformative type *Bool* is replaced by $(\Gamma \;\vdash\; e \downarrow \tau) \uplus (\Gamma \;\nvdash\; e \downarrow \tau)$.
- Program and correctness proof are one.

Why ΠΣ ?

- Agda implements many *high level features* such as:

  Datatype definitions Inductive and Coinductive families.

  Pattern matching with dependent types.

  Hidden parameters generalizing Hindley-Milner.
- Complicate metatheory
- Potential source of bugs in the implementation
- Explain high level features via a core language
- Intermediate language for compilation
- Similar role as FC for Haskell

ΠΣ in a nutshell

- Dependent function types (Π-types).
- Dependent product types (Σ-types).
- A (very) impredicative universe of types with **Type** : **Type**.
- Finite sets (enumerations) using reusable labels.
- A general mechanism for mutual recursion.
- Lifted types to control recursion.
- Structural equality for recursive definitions.
- Typechecker available on hackage (pisigma).

Partial?

- Totality is important for dependently typed programming:
  - Non-terminating proofs are not very useful.
  - Total terms of propositional types (e.g. equalities) don't need to be executed at run time.
- However, I believe it is beneficial to separate type checking from totality:
  - Mechanism of type checking is independent of totality.
  - Prototypes may fail totality checks.
  - Type soundness independent of totality.
  - Evidence for termination can be supplied independently.
  - Which notion of totality?
- Thierry Coquand is working on a similar calculus:
  The Calculus of Definitions

ΠΣ by example

1. Datatypes
2. Codata
3. Equality
4. Families
5. Universes

Datatypes

$$\begin{aligned}
&\mathit{Nat} : \mathbf{Type} = (l : \{\mathit{zero}\ \mathit{suc}\}) * \\
&\qquad\qquad \mathbf{case}\ l\ \mathbf{of}\ \{\ \mathit{zero} \to \{\mathit{unit}\} \\
&\qquad\qquad\qquad\qquad \mid \mathit{suc} \to [\mathit{Nat}]\};
\end{aligned}$$

- *Nat* is a recursively defined Σ-type.
- [. .] stops unfolding of recursive definitions.
- Derive constructors:

$$\mathit{zero} : \mathit{Nat} = ('\mathit{zero}, '\mathit{unit})$$

$$\mathit{suc} : \mathit{Nat} \to \mathit{Nat} = \lambda i \to ('\mathit{suc}, i)$$

- We use $'l$ to distinguish labels from variables.

$$
\begin{array}{l}
add : Nat \to Nat \to Nat; \\
add = \lambda m\ n \to \textbf{split}\ m\ \textbf{with}\ (lm, m') \to \\
\qquad\qquad\qquad !\textbf{case}\ lm\ \textbf{of}\ \{\ zero \to [\,n\,] \\
\qquad\qquad\qquad\qquad\qquad\qquad \mid suc \ \ \to [\,suc\ (add\ m'\ \ n)\,]\,\};
\end{array}
$$

- Recursive functions are defined using the same mechanism as recursive types.
- If $t : A$ then $[\,t\,] : \uparrow A$ (box: stops unfolding)
- If $t : \hat{}\ A$ then $!t : A$ (forcing).
- $![A] \equiv A$
- $add\ (suc\ (suc\ zero))\ (suc\ zero) \equiv$ $('suc, ('suc, ('suc, ('zero, 'unit))))$
- Use some coercions $A : \textbf{Type}$, if $A : \uparrow\textbf{Type}$ ... *(to be made explicit in future.)*

Codata

$$omega : Nat = ('suc, omega);$$

- *omega* will diverge.
- To define codata types we use lifting.

$$Stream : \mathbf{Type} \to \mathbf{Type} = \lambda A \to A * [\uparrow(Stream\ A)];$$

- We can now define corecursive programs:

$$\begin{aligned} &from : Nat \to Stream\ Nat; \\ &from = \lambda n \to (n, [from\ ('suc, n)]); \end{aligned}$$

- Evaluation of *from zero* terminates with $(zero, \mathbf{let}\ n : Nat = zero\ \mathbf{in}\ [from\ ('suc, n)])$

Mixed data / codata

- Some datatypes are mixed inductive / coinductive.
- An example is the type of stream processors:

$$
\begin{aligned}
&SP : \textbf{Type} \to \textbf{Type} \to \textbf{Type};\\
&SP = \lambda a\ b \to (l : \{\mathit{get}\ \mathit{put}\})\\
&\qquad\qquad\qquad * \textbf{case}\ l\ \textbf{of}\ \{\ \mathit{get} \to [a \to SP\ a\ b]\\
&\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{put} \to [b * \uparrow(SP\ a\ b)]\,\};
\end{aligned}
$$

- We can define the identity stream processor:

$$
\begin{aligned}
&\mathit{idsp} : (A : \textbf{Type}) \to SP\ A\ A\\
&\qquad = \lambda A \to ('\mathit{get}, \lambda a \to ('\mathit{put}, (a, [\mathit{idsp}\ A])));
\end{aligned}
$$

- We can also define an interpretation function:

$$
\mathit{eval} : (A\ B : \textbf{Type}) \to SP\ A\ B \to \mathit{Stream}\ A \to \mathit{Stream}\ B;
$$

Equality

- ΠΣ doesn't (yet) have an identity type.
- However, for 1st order types equality is definable, e.g. for *Nat*.

$$
\begin{array}{lcl}
\mathit{eqNat} : \mathit{Nat} \to \mathit{Nat} \to \mathit{Bool}; & & \\
\ldots & & \\
T & : & \mathit{Bool} \to \mathbf{Type} \\
 & = & \lambda b \to \mathbf{case}\ b\ \mathbf{of}\ \{\ \mathit{true} \to \{\mathit{unit}\} \\
 & & \qquad\qquad\qquad\quad \mid \mathit{false} \to \{\,\}\}; \\
\mathit{EqNat} & : & \mathit{Nat} \to \mathit{Nat} \to \mathbf{Type} \\
 & = & \lambda m\ n \to T\ (\mathit{eqNat}\ m\ n);
\end{array}
$$

- We can *prove* that the equality is reflexive and substitutive:

$$
\begin{array}{l}
\mathit{reflNat} : (n : \mathit{Nat}) \to \mathit{EqNat}\ n\ n; \\
\mathit{substNat} : (P : \mathit{Nat} \to \mathbf{Type}) \to (m\ n : \mathit{Nat}) \to \\
\qquad\qquad \mathit{EqNat}\ m\ n \to P\ m \to P\ n;
\end{array}
$$

- We use dependent elimination for *reflNat* and *substNat*

$$
\begin{array}{l}
\mathit{reflNat} : (n : \mathit{Nat}) \to \mathit{EqNat}\ n\ n; \\
\mathit{reflNat} = \lambda n \to \mathbf{split}\ n\ \mathbf{with}\ (\mathit{ln}, n') \to \\
\qquad\qquad\qquad\quad !\mathbf{case}\ \mathit{ln}\ \mathbf{of}\ \{\ \mathit{zero} \to [\,'\mathit{unit}\,] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{suc} \ \ \to [\,\mathit{reflNat}\ n'\,]\};
\end{array}
$$

- The type checker exploits the constraint that the scrutinee equals the constructor when checking branches.
- But only if the scrutinee is a variable.
- This is less general than in a previous version of ΠΣ.
- But simpler and suffcent for top-level pattern matching.

Families

- We can define families by recursion over the indices:

$$
\begin{array}{l}
\mathit{Vec} : \mathbf{Type} \to \mathit{Nat} \to \mathbf{Type};\\
\mathit{Vec} = \lambda A\ n \to \mathbf{split}\ n\ \mathbf{with}\ (n_l, n_r) \to\\
\qquad\qquad\qquad \mathbf{case}\ n_l\ \mathbf{of}\ \{\ \mathit{zero} \to \mathit{Unit}\\
\qquad\qquad\qquad\qquad\qquad\ \ \mid \mathit{suc}\ \ \to A * [\,\mathit{Vec}\ A\ n_r\,]\,\};
\end{array}
$$

- or by exploiting equality:

$$
\begin{array}{l}
\mathit{Vec} : \mathbf{Type} \to \mathit{Nat} \to \mathbf{Type};\\
\mathit{Vec} = \lambda A\ n \to (l : \{\,\mathit{nil}\ \mathit{cons}\,\}) *\\
\qquad\quad \mathbf{case}\ l\ \mathbf{of}\ \{\ \mathit{nil} \quad\ \to \mathit{EqNat}\ \mathit{zero}\ n\\
\qquad\qquad\qquad\quad \mid \mathit{cons} \to [(n' : \mathit{Nat}) * A * \mathit{Vec}\ A\ n'\\
\qquad\qquad\qquad\qquad\qquad\quad * \mathit{EqNat}\ (\mathit{suc}\ n')\ n]\,\};
\end{array}
$$

$$
\begin{array}{l}
\mathit{Vec} : \mathbf{Type} \to \mathit{Nat} \to \mathbf{Type}; \\
\mathit{Vec} = \lambda A\ n \to (l : \{\mathit{nil}\ \mathit{cons}\}) * \\
\qquad \mathbf{case}\ l\ \mathbf{of}\ \{\ \mathit{nil} \quad \to \mathit{EqNat}\ \mathit{zero}\ n \\
\qquad \qquad \qquad \mid \mathit{cons} \to [(n' : \mathit{Nat}) * A * \mathit{Vec}\ A\ n' \\
\qquad \qquad \qquad \qquad \qquad * \mathit{EqNat}\ (\mathit{suc}\ n')\ n]\};
\end{array}
$$

- Using equality is more general (e.g. typed λ-terms);
- corresponds to the Agda datatypes;
- and we could omit indices at runtime.

Universes

- Define a universe of datatypes with decidable equality:

$$
\begin{array}{l}
U \ : \mathbf{Type};\\
El : U \to \mathbf{Type};\\
U = (l : \{\, enum\ sigma\ box \,\}) * \\
\quad \mathbf{case}\ l\ \mathbf{of}\ \{\, enum \ \to Nat \\
\qquad\qquad\qquad \mid sigma \to [(a : U) * (El\ a \to U)] \\
\qquad\qquad\qquad \mid box \quad \to [\uparrow U]\};\\
El = \lambda a \to \mathbf{split}\ a\ \mathbf{with}\ (a_l, a_r) \to \\
\quad !\mathbf{case}\ a_l\ \mathbf{of} \\
\qquad \{\, enum \ \to [\, Fin\ a_r\,] \\
\qquad \mid sigma \to [\mathbf{split}\ a_r\ \mathbf{with}\ (b, c) \to \\
\qquad\qquad\qquad\qquad (x : El\ b) * (El\ (c\ x))] \\
\qquad \mid box \quad \to [[El\ (!a_r)]]\};
\end{array}
$$

- Arbitrary interleaving of declarations and definitions allowed.

α-equality

- Boxes ([. .]) stop unfolding of definitions:

$$\textbf{let } x : \mathit{Bool} = {}'\mathit{true} \textbf{ in } [x] \not\equiv_\beta [{}'\mathit{true}].$$

- However, we have:

$$\textbf{let } x : \mathit{Bool} = {}'\mathit{true} \textbf{ in } [x] \equiv_\alpha \textbf{let } y : \mathit{Bool} = {}'\mathit{true} \textbf{ in } [y]$$

- We have to compare the definitions:

$$\textbf{let } x : \mathit{Bool} = {}'\mathit{true} \textbf{ in } [x] \not\equiv_\beta \textbf{let } y : \mathit{Bool} = {}'\mathit{false} \textbf{ in } [y]$$

- But only if they are actually used:

$$\begin{aligned} & \textbf{let } x : \mathit{Bool} = {}'\mathit{true}, y : \mathit{Bool} = {}'\mathit{false} \textbf{ in } [x] \\ \equiv_\alpha\ & \textbf{let } z : \mathit{Bool} = {}'\mathit{true}, y : \mathit{Bool} = {}'\mathit{true} \textbf{ in } [z] \end{aligned}$$

- We specify α-equality using partial bijections on variables.
- Let expressions extend partial bijections:

$$\frac{\varphi : \Delta \sim \Delta' \vdash \psi : \Gamma \sim \Gamma' \quad \varphi ; \psi : (\Delta ; \Gamma) \sim (\Delta' ; \Gamma') \vdash t \equiv_\alpha t'}{\varphi : \Delta \sim \Delta' \vdash \textbf{let } \Gamma \textbf{ in } t \equiv_\alpha \textbf{ let } \Gamma' \textbf{ in } t'}$$

- Declarations may be identified:

$$\frac{\varphi : \Delta \sim \Delta' \vdash \psi : \Gamma \sim \Gamma' \quad \varphi ; \psi : (\Delta ; \Gamma) \sim (\Delta' ; \Gamma') \vdash A \equiv_\beta A'}{\varphi : \Delta \sim \Delta' \vdash (\psi ; (x, x')) : (\Gamma ; x : A) \sim (\Gamma' ; x' : A')}$$

- Or not:

$$\frac{\varphi : \Delta \sim \Delta' \vdash \psi ; (x, -) : \Gamma \sim \Gamma'}{\varphi : \Delta \sim \Delta' \vdash \psi : (\Gamma ; x : A) \sim \Gamma'}$$

- If identified, definitions have to agree:

$$\frac{\varphi \vdash x{\sim}x' \quad \varphi : \Delta{\sim}\Delta' \vdash \psi : \Gamma{\sim}\Gamma' \quad \varphi;\psi : (\Delta;\Gamma){\sim}(\Delta';\Gamma') \vdash t \equiv_\beta t'}{\varphi : \Delta{\sim}\Delta' \vdash \psi : (\Gamma; x = t){\sim}(\Gamma'; x' = t')}$$

- Otherwise we ignore them:

$$\frac{\varphi \vdash x{\sim}- \quad \varphi : \Delta{\sim}\Delta' \vdash \psi : \Gamma{\sim}\Gamma'}{\varphi : \Delta{\sim}\Delta' \vdash \psi : (\Gamma; x = t){\sim}\Gamma'}$$

- In the implementation we construct the partial bijection lazily:
  - If we compare two defined variables,
  - we replace both by a fresh variable,
  - and then check wether the definitions agree.
- See our paper `http://www.cs.nott.ac.uk/~txa/publ/pisigma-new.pdf` for all the rules.

What next?

- Implement an Agda-like language on top of $\Pi\Sigma$.
- Add extensional, propositional equality.
- Develop the metatheory, e.g. typesoundness.
- Implement $\Pi\Sigma$ in Agda , develop the metatheory formally.
- $\Pi\Sigma$ in $\Pi\Sigma$.
- Investigate more general constraints.
- Certificate based, extensible totality checker.
- Optimizing compiler.