Recent News about Normalization by Evaluation for Martin-Löf Type Theory

Peter Dybjer
(joint work with Andreas Abel and Thierry Coquand)

Proglog-CVS seminar
Göteborg, 4 April, 2007

After AIM4

- Talk on nbe for Martin-Löf type theory, AIM 4, Senri, May 2006.
- Paper by Andreas Abel, Klaus Aehlig, Peter Dybjer to be presented at MFPS XXIII, New Orleans, 11-14 April 2007. This paper shows correctness of an eta-expanding nbe-algorithm for a version of Martin-Löf type theory with *untyped conversion*
- Implementation by Nils-Anders Danielsson of nbe for an LF-style Martin-Löf type theory in AgdaLight using dependent types to express correctness ("internal type theory"). Paper in the proceedings of TYPES 2006.

After AIM 5

- Talk by Thierry autumn 2006 on applications of nbe to metatheory of Martin-Löf type theory (Agda 2 core language). Especially injectivity of Π and correctness of type checking algorithm. Issues of eta-conversion, LF-style vs universe-style Martin-Löf type theory.
- Paper by Andreas Abel, Thierry Coquand, Peter Dybjer to be presented at LiCS, Wroclaw, July 2007. This paper shows correctness of an eta-expanding nbe-algorithm for a version of Martin-Löf type theory with *typed equality judgements*. Algorithm closer to what's used in Agda and Coq (and in Gregoire and Leroy's byte-code compiler)
- Cwf-based metatheory of type theory.
- Internal type theory revisited. Cwf-based metatheory implemented in Agda 2. (Ulf Norell, Alexandre Buisse)

Today

- Beta-normalizing nbe-algorithm of Martin-Löf type theory with $\Pi, \mathrm{N}, \mathrm{U}$. Semantics as normal forms in higher-order abstract syntax. **Aim:** to have a more Agda-related and also perhaps more elegant implementation of nbe.
- This algorithm can be modified to give long eta-normal forms. Most of our work has dealt with such algorithms: we want to work with type theory with eta-conversion. Not presented today though.
- Categories with families - cwfs. **Aim:** to get away from the "a version of Martin-Löf type theory"-syndrome.

Martin-Löf Type Theory

Types and terms (types are terms - universe à la Russell)

$$\begin{array}{rcl} a & ::= & \Pi\, a\, a \mid \mathrm{N} \mid \mathrm{U} \\ & \mid & v_i \mid \lambda a \mid a\, a \\ & \mid & 0 \mid \mathrm{S}\, a \mid \mathrm{R}\, a\, a\, a \end{array}$$

Normal types and terms

$$t ::= \Pi\, t\, t \mid \mathrm{N} \mid \mathrm{U} \mid \lambda t \mid 0 \mid \mathrm{S}\, t \mid s$$

Neutral types and terms

$$s ::= v_i \mid s\, t \mid \mathrm{R}\, t\, t\, s$$

β-normal forms of type theory

First-order syntax of normal and neutral types and terms:

$$\begin{aligned} t &::= \Pi\, t\, t \mid \mathrm{N} \mid \mathrm{U} \mid \lambda t \mid 0 \mid \mathrm{S}\, t \mid s \\ s &::= v_i \mid s\, t \mid \mathrm{R}\, t\, t\, s \end{aligned}$$

"There is no model of normal forms; normality is not closed under application (and recursion)".
Higher-order abstract syntax (as domain equations)

$$\begin{aligned} D &::= (D \times (D \to D)) + 1 + 1 + (D \to D) + 1 + D + E \\ E &::= N + (E \times D) + (D \times D \times E) \end{aligned}$$

Normal and neutral terms in hoas in Haskell

Higher-order abstract syntax

as domain equations

$$\begin{aligned} D &::= (D \times (D \to D)) + 1 + 1 + (D \to D) + 1 + D + E \\ E &::= N + E \times D + D \times D \times E \end{aligned}$$

as Haskell data types

```
data D = PiD D (D -> D) | NatD | UD
       | LamD (D -> D) | ZD | SuccD D | Up E

data E = VarE Int | AppE E D | RecE D D E
```

Application and recursion of normal forms in hoas

```
data D = Up E | LamD (D -> D) | ZD | SuccD D

data E = VarE Int | AppE E D | RecE D D E

appD :: D -> D -> D

appD (LamD f) d = f d
appD (Up e)   d = Up (AppE e d)

recD :: D -> D -> D -> D

recD z s  ZeroD     = z
recD z s (SuccD d) = s `appD` d `appD` (recD z s d)
recD z s (Up r)    = Up (RecE z s r)
```

Evaluation of terms in model of normal forms in hoas

```
data Te = Pi Te Te | Nat | U
        | V Int | Lam Te | App Te Te
        | Z | Succ Te | Rec Te Te Te

eval :: Te -> [D] -> D

eval (Var i)      ds = lookup ds i
eval (App r s)    ds = appD (eval r ds) (eval s ds)
eval (Lam r)      ds = LamD (\d -> eval r (d:ds))
eval Zero         ds = ZeroD
eval (Succ r)     ds = SuccD (eval r ds)
eval (Rec r s t) ds = recD (eval r ds) (eval s ds) (eval t d
eval (Pi a b)     ds = PiD (eval a ds) (\d -> eval b (d:ds))
eval Nat          ds = NatD eval U ds = UD
```

From hoas to foas (printing, reflection, quoting)

Normal and neutral terms in first-order syntax (using de Bruijn levels):

```
data Nf = PiN Nf Nf | LamN Nf
        | NatN | ZeroN | SuccN Nf | UN | UpN Ne

data Ne = VarN int | AppN Ne Nf | RecN Nf Nf Ne

down :: D -> Int -> Nf
downE :: E -> Int -> Ne

down (Up e)     k = downE e k
down (PiD d g)  k = PiN (down d k) (down (g (VarN k)) (k+1))
down (LamD f)   k = LamN (down (f (VarN k)) (k+1))
down  NatD      k = NatN
down  ZeroD     k = ZeroN
down (SuccD d)  k = SuccN (down d k)
down  UD        k = UN
```

From hoas to foas: neutral terms

```
downE :: E -> Int -> Ne

downE (VarE l)      k = VarN l
downE (AppE n d)    k = AppN (downE n k) (down d k)
downE (RecE z s n) k = RecN (down z k) (down s k) (downE n k
```

The decision problem for equality of terms and types

We now define (for closed terms a)

```
nbe a = down (eval a []) 0
```

We can then show that if a, a' : A, then we can decide equality by computing nbe and comparing normal forms:

```
a conv a' iff nbe a = nbe a'
```

Note however, that since types are special terms we can also use the algorithm for correct types a, a' and check whether they are equal. Note also that nbe does not use types at all. It is similar to Aehlig and Joachimski's untyped nbe which lazily computes the Böhm tree of a as an element of Nf!

Is this nbe?

Two differences from Berger-Schwichtenberg-style nbe (tdpe):

1. Untyped nbe which β-normalizes but does not η-expand, cf Aehlig and Joachimski's untyped nbe. (A modified algorithm η-expands using types.)
2. Reflection from syntax to semantics is avoided; instead we use a set of "semantic neutral terms" aka neutral terms in hoas. Cf Agda type-checker, Gregoire and Leroy's byte-code compiler for Coq.

The "a version of Martin-Löf type theory" problem

- There are many versions of Martin-Löf type theory. Are they equivalent? Which is the right one?
- Abstract characterization of Martin-Löf type theory as the initial category with families (cwf) with extra structure $(\Pi, \mathrm{N}, \mathrm{U})$. This is a notion defined up to isomorphism.
- The initial category with families can be constructed using categorical combinators for cwfs.
- It is easy to modify the nbe-algorithm to work on the initial cwf.
- This yields clear approach to metatheory of Martin-Löf type theory.

Rest of talk:

- Nbe for categorical combinators
- Introduction to cwfs
  - Categorical models of dependent types
  - Definition of cwf
  - The generalized algebraic theory (gat) of cwfs

Categorical combinators for Martin-Löf type theory

Language of terms a and s

$$\begin{array}{rcl} a & ::= & a\gamma \mid \mathrm{q} \\ & \mid & \lambda(a) \mid \mathrm{ap}(a,a) \\ & \mid & 0 \mid \mathrm{S}(a) \mid \mathrm{R}(a,a,a) \\ & \mid & \Pi(a,a) \mid \mathrm{N} \mid \mathrm{U} \end{array}$$

Substitutions (sequences of terms)

$$\gamma \quad ::= \quad \gamma\circ\gamma \mid \mathrm{id} \mid \langle\rangle \mid \langle\gamma, a\rangle \mid \mathrm{p}$$

Raw cwf-terms and substitutions in Haskell

We can also define implicit syntax. Let's do it in Haskell:

```
data Te = Sub Te Ts | Q
        | Pi Te Te | Lam Te | App Te Te
        | Nat | Zero | Succ Te | Rec Te Te Te
        | U

data Ts = Empty | Ext Ts Te | P | Comp Ts Ts | Id
```

We will define the normalization function for this.

Evaluation in cwf of normal forms in hoas

```
eval :: Te -> [D] -> D

eval (Sub a as)  ds = eval a (evals as ds)
eval Q           ds = head ds
eval (Pi a b)    ds = PiD (eval a ds) (\d -> eval b (d:ds))
eval (App r s)   ds = appD (eval r ds) (eval s ds)
eval (Lam r)     ds = LamD (\d -> eval r (d:ds))
eval (Nat)       ds = NatD
eval (Zero)      ds = ZeroD
eval (Succ r)    ds = SuccD (eval r ds)
eval (Rec r s t) ds = recD (eval r ds) (eval s ds) (eval t d
eval (U)         ds = UD
```

Evaluation of substitutions

```
evals :: Ts -> [E] -> [D]

evals Empty          ds = []
evals (Ext as a)     ds = (eval a ds):(evals as ds)
evals P              ds = tail ds
evals Id             ds = ds
evals (Comp bs as) ds = evals bs (evals bbas ds)
```

The cwf of normal forms in hoas can be defined in terms of partial equivalence relations (pers) given by an *inductive-recursive* definition of *equal normal types and equal normal terms*.

Normal forms for cwfs

- The normal forms for cwfs have the same structure as before.
- Same sets D and E for hoas and Nf and Ne for foas.
- Nf and Ne can be injected into the set of categorical combinators.

What is a model of dependent type theory?

Categorical models:

- Cartmell 1978. Categories with attributes and contextual categories, model of intensional type theory. Generalized algebraic theories.
- Seely 1984. Locally cartesian closed categories as models of extensional type theory.
- Many variations and extensions in late 80s and early 90s (Taylor, Hyland, Pitts, Ehrhard, Streicher, Ritter)

Martin-Löf:

- About models of intutionistic type theory and the notion of definitional equality (ca 1974)
- Work with PhD student Bo Hagerf - notion of model of type theory (ca 1983)
- Substitution calculus (ca 1990)

The category of contexts

A mismatch between typed lambda calculus (and more generally functions as terms with variables) and category theory:

Typed lambda calculus n-place functions represented by terms with at most n free variables

Category theory 1-place functions represented by arrows

Solution: build a category of contexts

Objects: contexts $x_1 : A_1, \ldots, x_m : A_m$

Morphisms: substitutions $x_1 = a_1[y_1, \ldots, y_n], \ldots, x_m = a_m[y_1, \ldots, y_n]$ is an arrow with source $y_1 : B_1, \ldots, y_n : B_n$ and target $x_1 : A_1, \ldots, x_m : A_m$.

This idea is basic to all (?) categorical models of type theory. In explicit substitution calculi it becomes a part of the formal system.

Dependent types as fibred sets

A family of sets

$$B(x)\ (x \in A)$$

can be represented by a *projection* function

$$\mathrm{p}_{A,B} : \sum_{x \in A} B(x) \to A$$

$$\mathrm{p}_{A,B}(x, y) = x$$

A function

$$f(x) \in B(x)\ (x \in A)$$

is then represented by a *section*

$$g : A \to \sum_{x \in A} B(x)$$

$$g(x) = (x, f(x))$$

of this projection:

$$\mathrm{p}_{A,B}(g(x)) = x$$

Categories with families are not based on fibrations

When types are represented by projections, substitution in types can be represented by *pullbacks*. This idea is basic to all (?) notions of model for dependent types except categories with families.

- Because of the coding of families of sets (dependent types) as fibrations and of substitution as pullback, the correspondence between syntax of type theory and categorical notions of model is non-trivial.
- The aim is to get a direct correspondence between notion of model and syntax, and yet to be completely algebraic: variable free and everything is expressed using operations and equations.
- Categories with families is an uncategorical categorical notion of model of type theory. It is otherwise very close to Cartmell's *categories with attributes*.

Categories with families

A *category with families* consists of

- A category C - the category of contexts.
- A family-valued functor $T : C^{op} \longrightarrow \mathbf{Fam}$ - to model types, terms and substitution.
- Extra structure to model the construction of contexts as lists of types, of substitutions as lists of terms, and the rules of assumption and weakening.

The category **Fam** of families of sets

Let **Fam** be the category of families of sets with

Objects: families of sets $(B(x))_{x \in A}$

Arrows: with source $(B(x))_{x \in A}$ and target $(B'(x'))_{x' \in A'}$ consist of pairs of

- functions $f : A \longrightarrow A'$
- families of functions $(g(x) : B(x) \longrightarrow B'(f(x)))_{x \in A}$.

Category with families

- C is the *category of contexts*. Its objects are called *contexts* and its morphisms are called *substitutions*.
- $T : C^{op} \longrightarrow \mathbf{Fam}$ is the functor which

  object part maps a context Γ to the family of sets of terms $\{a \mid \Gamma \vdash a : A\}$ indexed by the set of types $\{A \mid \Gamma \vdash A\ type\}$ in Γ.

  arrow part maps a substitution γ to a pair of functions which perform substitution of γ in types and terms respectively. We write $A\gamma$ for substitution of γ in a type A and $a\gamma$ for substitution of γ in the term a.

The extra structure for context formation, assumption, etc

terminal object [ ] of C called the *empty context*. The unique arrow into [ ] is the empty substitution.

context comprehension - an operation which to an object Γ of C and a type A in Γ associates four components modelling

context extension: an object $\Gamma; A$ of C;

weakening: a morphism $\mathrm{p}_{\Gamma,A} : \Gamma; A \longrightarrow \Gamma$ of C - the *first projection)*

assumption: a term $\mathrm{q}_{\Gamma,A} \in \Gamma; A \vdash A\mathrm{p}_{\Gamma,A}$ - the *second projection*

substitution extension: for each object Δ in C, morphism $\gamma : \Delta \longrightarrow \Gamma$, and term $a \in \Delta \vdash A\gamma$, there is a unique morphism $\theta = \langle \gamma, a \rangle : \Delta \longrightarrow \Gamma; A$, such that $p \circ \theta = \gamma$ and $q\theta = a$. This is the *universal property* of context comprehension.

The cwf of sets and families of sets and families of families of sets

A basic example of a cwf is obtained by letting

$$
\begin{aligned}
C &= \textit{the category of small sets} \\
T(\Gamma) &= \textit{the family of } \Gamma - \textit{indexed families of small sets} \\
A\delta(x) &= A(\delta(x)) \\
a\delta(x) &= a(\delta(x)) \\
[\,] &= 1 \\
\Gamma; A &= \sum_{x \in \Gamma} A(x)
\end{aligned}
$$

The generalized algebraic theory of cwfs

The notion of a category with families can be formalized as a *generalized algebraic theory* in the sense of Cartmell. Generalized algebraic theories generalize many-sorted algebraic theories and are based on a framework of dependent types. They have four parts:

- A list of *sort symbols* with dependent typings.
- A list of *operator symbols* with dependent typings.
- A list of *equations* between well-formed *sort expressions* (not needed here).
- A list of *equations* between well-formed *terms*.

The generalized algebraic theory of categories

Sort symbols:

$$\mathrm{Ctxt}\ \mathrm{sort}$$

$$\frac{\Delta, \Gamma : \mathrm{Ctxt}}{\Delta \to \Gamma\ \mathrm{sort}}$$

Operator symbols:

$$\frac{\Theta, \Delta, \Gamma : \mathrm{Ctxt} \qquad \gamma : \Delta \to \Gamma \qquad \delta : \Theta \to \Delta}{\gamma \circ \delta : \Theta \to \Gamma}$$

$$\frac{\Gamma : \mathrm{Ctxt}}{\mathrm{id} : \Gamma \to \Gamma}$$

Equations:

$$\begin{aligned} (\gamma \circ \delta) \circ \theta &= \gamma \circ (\delta \circ \theta) \\ \mathrm{id} \circ \gamma &= \gamma \\ \gamma \circ \mathrm{id} &= \gamma \end{aligned}$$

Note that we use implicit arguments, official syntax of gats use explicit

Rules for family-valued functors

Sort symbols:

$$\frac{\Gamma : \mathrm{Ctxt}}{\mathrm{Ty}(\Gamma)\ \mathrm{sort}}$$

$$\frac{\Gamma : \mathrm{Ctxt} \qquad A : \mathrm{Ty}(\Gamma)}{\Gamma \vdash A\ \mathrm{sort}}$$

Operator symbols:

$$\frac{\Delta, \Gamma : \mathrm{Ctxt} \qquad A : \mathrm{Ty}(\Gamma) \qquad \gamma : \Delta \to \Gamma}{A\gamma : \mathrm{Ty}(\Delta)}$$

$$\frac{\Delta, \Gamma : \mathrm{Ctxt} \quad A : \mathrm{Ty}(\Gamma) \quad a : \Gamma \vdash A \quad \gamma : \Delta \to \Gamma}{a\gamma : \Delta \vdash A\gamma}$$

Equations:

$$\begin{aligned} A(\gamma \circ \delta) &= A\gamma\delta \\ A\mathrm{id} &= A \\ a(\gamma \circ \delta) &= a\gamma\delta \\ a\mathrm{id} &= a \end{aligned}$$

Rules for the terminal object

Operator symbols:

$$[\,] : \mathrm{Ctxt}$$

$$\frac{\Gamma : \mathrm{Ctxt}}{\langle\rangle : \Gamma \to [\,]}$$

Equations

$$\begin{aligned} \langle\rangle \circ \gamma &= \langle\rangle \\ \mathrm{id} &= \langle\rangle \end{aligned}$$

Rules for context comprehension

Operator symbols:

$$\frac{\Gamma : \mathrm{Ctxt} \qquad A : \mathrm{Ty}(\Gamma)}{\Gamma ; A : \mathrm{Ctxt}}$$

$$\frac{\Delta, \Gamma : \mathrm{Ctxt} \quad A : \mathrm{Ty}(\Gamma) \quad \gamma : \Delta \to \Gamma \quad a : \Delta \vdash A\gamma}{\langle \gamma, a \rangle : \Delta \to \Gamma ; A}$$

$$\frac{\Gamma : \mathrm{Ctxt} \qquad A : \mathrm{Ty}(\Gamma)}{\mathrm{p} : \Gamma ; A \to \Gamma}$$

$$\frac{\Gamma : \mathrm{Ctxt} \qquad A : \mathrm{Ty}(\Gamma)}{\mathrm{q} : \Gamma ; A \vdash A\mathrm{p}}$$

Equations:

$$\begin{aligned} \mathrm{p} \circ \langle \gamma, a \rangle &= \gamma \\ \mathrm{q} \langle \gamma, a \rangle &= a \\ \langle \delta, a \rangle \circ \gamma &= \langle \delta \circ \gamma, a\gamma \rangle \\ \mathrm{id} &= \langle \mathrm{p}, \mathrm{q} \rangle \end{aligned}$$

Cwfs and Martin-Löf's substitution calculus

Note the correspondence between

- *sort symbols* in the gat of cwfs and *judgement forms* of Martin-Löf's substitution calculus
- *operator symbols* in the gat of cwfs and *inference rules* of Martin-Löf's substitution calculus

But note also that there are

- *no* sort symbols which correspond to *equality judgements*
- *no* operator symbols which correspond to *general equality rules* (equivalence and congruence reasoning).

Instead general equality reasoning is inherited from the metalanguage of gats.

- With the cwf-rules in place it is straightforward to add rules for the various type formers!

Rules for Π

Operator symbols:

$$\frac{\Gamma : \mathrm{Ctxt} \qquad A : \mathrm{Ty}(\Gamma) \qquad B : \mathrm{Ty}(\Gamma; A)}{\Pi(A, B) : \mathrm{Ty}(\Gamma)}$$

$$\frac{\Gamma : \mathrm{Ctxt} \quad A : \mathrm{Ty}(\Gamma) \quad B : \mathrm{Ty}(\Gamma; A) \quad b : \Gamma; A \vdash B}{\lambda(b) : \Gamma \vdash \Pi(A, B)}$$

$$\frac{\Gamma : \mathrm{Ctxt} \quad A : \mathrm{Ty}(\Gamma) \quad B : \mathrm{Ty}(\Gamma; A) \quad c : \Gamma \vdash \Pi(A, B) \quad a : \Gamma \vdash A}{\mathrm{ap}(c, a) : \Gamma \vdash B\langle \mathrm{id}, a\rangle}$$

Equations:

$$\begin{aligned}
\Pi(A, B)\gamma &= \Pi(A\gamma, B\langle \gamma \circ \mathrm{p}, \mathrm{q}\rangle) \\
\lambda(b)\gamma &= \lambda(b\langle \gamma \circ \mathrm{p}, \mathrm{q}\rangle) \\
\mathrm{ap}(c, a)\gamma &= \mathrm{ap}(c\gamma, a\gamma) \\
\mathrm{ap}(\lambda(b), a) &= b\langle \mathrm{id}, a\rangle \\
\lambda(\mathrm{ap}(c\mathrm{p}, \mathrm{q})) &= c
\end{aligned}$$

The initial cwf with extra structure

We can define a notion of cwf-morphism, and construct the initial cwf. However, without extra structure it has no types and is not very interesting. Consider initial cwfs with rules for $\Pi, \mathrm{N}, \mathrm{U}$. This *term model* can be constructed as a special case of Cartmell's construction of the initial model of a gat.

Explicit syntax of initial gat of cwfs

$$
\begin{array}{rcl}
\Gamma & ::= & [\,] \mid \Gamma; A \\
\gamma & ::= & \gamma \circ_{\Gamma,\Gamma,\Gamma} \gamma \mid \mathrm{id}_\Gamma \mid \langle\rangle_\Gamma \mid \langle \gamma, a \rangle_{\Gamma,\Gamma,A} \mid \mathrm{p}_{\Gamma,A} \\
A & ::= & A\gamma_{\Gamma,\Gamma} \\
 & \mid & \Pi_\Gamma(A,A) \mid \mathrm{N}_\Gamma \mid \mathrm{U}_\Gamma \mid \mathrm{T}_\Gamma(a) \\
a & ::= & a\gamma_{\Gamma,\Gamma,A} \mid \mathrm{q}_{\Gamma,A} \\
 & \mid & \lambda_{\Gamma,A,A}(a) \mid \mathrm{ap}_{\Gamma,A,A}(a,a) \\
 & \mid & 0_\Gamma \mid \mathrm{S}_\Gamma(a) \mid \mathrm{R}_\Gamma(A,a,a,a) \\
 & \mid & \hat{\Pi}_\Gamma(a,a) \mid \hat{\mathrm{N}}_\Gamma
\end{array}
$$

The intial cwf - judgements

By using an inference system for generalized algebraic theories, we can then deduce the following valid typings and equalities of the generalized algebraic theory of cwf with $\Pi, \mathrm{N}, \mathrm{U}$.

$$\Gamma : \mathrm{Ctxt}$$
$$\Gamma = \Gamma' : \mathrm{Ctxt}$$
$$\gamma : \Gamma \to \Gamma'$$
$$\gamma = \gamma' : \Gamma \to \Gamma'$$
$$A : \mathrm{Ty}(\Gamma)$$
$$A = A' : \mathrm{Ty}(\Gamma)$$
$$a : \Gamma \vdash A$$
$$a = a' : \Gamma \vdash A$$

Properties of the initial cwf with $\Pi, \mathrm{N}, \mathrm{U}$

- All four equality judgements are decidable, e g given $a, a' : \Gamma \vdash A$ it is decidable whether $a = a' : \Gamma \vdash A$. Similarly for the three other equality judgements. (Proof using nbe)
- (Type inference) Given a raw explicit term a, we can decide whether there are Γ, A such that $a : \Gamma \vdash A$.
- We can prove that Π is injective, that is, if $\Pi_\Gamma(A, B) = \Pi'_\Gamma(A', B')$ then $\Gamma = \Gamma', A = A', B = B'$.

To do list

- Nbe for explicit syntax with eta-conversion. Proof of decidability of equality.
- Justification of implicit syntax. Prove that it gives rise to initial cwf too.
- Proof of correctness of the type-checking algorithm using nbe for implicit syntax.
- Extension to the Calculus of Constructions.