Verifying a Semantic $\beta\eta$-Conversion Test for Martin-Löf Type Theory

Andreas Abel[1]
Thierry Coquand[2] Peter Dybjer[2]

[1]Ludwig-Maximilians-University Munich
[2]Chalmers University of Technology

Mathematics of Program Construction
Marseille, France
18 July 2008

Background

- Dependently typed languages allow specification, implementation, and verification in the same language.
  - Strong data invariants.
  - Pre- and post-conditions.
  - Soundness.
- Programs (e.g., `add`) can occur in types of other programs (e.g., `append`).

  ```
  append : (n m : Nat) -> Vec n -> Vec m -> Vec (add n m)
  ```
- Type equality can be established
  - automatically, e.g., `Vec (add 0 m) = Vec m` (by computation), or
  - by proof, e.g., `Vec (add n m) = Vec (add m n)`.
- Goal: establish more equalities automatically.

Building η into Definitional Equality

- Coq's definitional equality is β ($+\ \delta + \iota$).
- The stronger definitional equality, the fewer the user has to revert to equality proofs.
- Why not η? ($f = \lambda x.\, f\, x$ if x new)
- Validates, for instance, $f = \mathsf{comp}\, f\, \mathsf{id}$.
- But η complicates the meta theory.
- Twelf, Epigram, and Agda check for $\beta\eta$-convertibility.
- Twelf's type-directed conversion check has been verified by Harper & Pfenning (2005).
- This work: towards verification of Epigram and Agda's equality check.

Language

- Core type theory:
  - Dependent function types $\mathsf{Fun}\, A\, \lambda x B$ (= `(x : A) -> B`) with η.
  - Predicative universes $\mathsf{Set}_0, \mathsf{Set}_1, \ldots$.
  - Natural numbers.
- We handle *large eliminations* (types defined by cases and recursion), in contrast to Harper & Pfenning (2005).
- Scales to Σ types with surjective pairing.
- Goal: handle all types with at most one constructor (Π, Σ, 1, 0, singleton types).
- Not a goal?: handle enumeration types (2, disjoint sums, …).

Syntax of Terms and Types

- Lambda-calculus with constants

$$
\begin{array}{lcll}
r, s, t & ::= & c \mid x \mid \lambda x.t \mid r\, s & \\
c & ::= & \mathsf{N} & \text{type of natural numbers} \\
 & & \mathsf{z} & \text{zero} \\
 & & \mathsf{s} & \text{successor} \\
 & & \mathsf{rec} & \text{primitive recursion} \\
 & & \mathsf{Fun} & \text{function space constructor} \\
 & & \mathsf{Set}_i & \text{universe of sets of level } i
\end{array}
$$

- $\Pi x\!:\!A.B$ (Agda: `(x : A) -> B`) is written $\mathsf{Fun}\, A\, (\lambda x.B)$.

Judgements

- Essential judgements

$$\begin{array}{ll} \Gamma \vdash t : A & t \text{ has type } A \text{ in } \Gamma \\ \Gamma \vdash t = t' : A & t \text{ and } t' \text{ are equal expressions of type } A \text{ in } \Gamma \end{array}$$

- Typing of functions:

$$\frac{\Gamma, x\!:\!A \vdash t : B}{\Gamma \vdash \lambda x.t : \mathsf{Fun}\, A\, (\lambda x.B)} \qquad \frac{\Gamma \vdash r : \mathsf{Fun}\, A\, (\lambda x.B) \qquad \Gamma \vdash s : A}{\Gamma \vdash r\, s : B[s/x]}$$

Set formation rules

- Small types (sets):

$$\frac{}{\Gamma \vdash \mathsf{N} : \mathsf{Set}_0} \qquad \frac{\Gamma \vdash A : \mathsf{Set}_i \qquad \Gamma, x : A \vdash B : \mathsf{Set}_i}{\Gamma \vdash \mathsf{Fun}\, A\, (\lambda x. B) : \mathsf{Set}_i}$$

- Set_0 includes types defined by recursion like $\mathsf{Vec}\, A\, n$.
- (Large) types:

$$\frac{\Gamma \vdash A : \mathsf{Set}_i}{\Gamma \vdash A : \mathsf{Set}_{i+1}} \qquad \frac{}{\Gamma \vdash \mathsf{Set}_i : \mathsf{Set}_{i+1}}$$

- E.g., $\mathsf{Fun}\ \mathsf{Set}_0\ (\lambda A.\, A \to (\mathsf{N} \to A)) : \mathsf{Set}_1$.
  In Agda: `(A : Set) -> A -> N -> A : Set1`.

Equality

- Conversion rule:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A = A' : \mathsf{Set}_i}{\Gamma \vdash t : A'}$$

- Type checking requires checking type equality!
- Equality axioms:

$$(\beta)\ \frac{\Gamma, x\!:\!A \vdash t : B \qquad \Gamma \vdash s : A}{\Gamma \vdash (\lambda x.t)\, s = t[s/x] : B[s/x]}$$

$$(\eta)\ \frac{\Gamma \vdash t : \mathsf{Fun}\, A\, (\lambda x.B)}{\Gamma \vdash (\lambda x.\, t\, x) = t : \mathsf{Fun}\, A\, (\lambda x.B)}\ x \notin \mathsf{FV}(t)$$

- Add computation axioms for primitive recursion.

The Type Checking Task

- Input a sequence of typed definitions in β-normal form

$$\begin{array}{llll} x_0 & : A_0 & = & t_0 \\ & \vdots & & \\ x_{n-1} & : A_{n-1} & = & t_{n-1} \end{array}$$

- Check the sequence in order
  1. check that A_i is well-formed
  2. evaluate A_i to X_i in current environment
  3. check that t_i is of type X_i
  4. evaluate t_i to d_i in current environment
  5. add binding $x_i : X_i = d_i$ to environment
- Type conversion: need to check type values X, X' for equality

Values

- In implementation of type theory, values could be:
  1. Normal forms (Agda 2)
  2. Weak head normal forms (Constructive Engine, Pollack)
  3. Explicit substitutions (Twelf)
  4. Closures (Epigram 2)
  5. Virtual machine code (Coq, Grégoire & Leroy (2002))
  6. Compiled code (Cayenne, Dirk Kleeblatt)
- Need symbolic execution at compile time.
- Abstract over implementation via applicative structures.

Applicative Structure

- Domain D of values with 2 operations:
  1. Application $\_ \cdot \_ : \mathsf{D} \times \mathsf{D} \to \mathsf{D}$
  2. Evaluation $\_\,\_ : \mathsf{Exp} \times (\mathsf{Var} \to \mathsf{D}) \to \mathsf{D}$.
- Laws:

$$\begin{array}{rcll} c\rho & = & c & \qquad \text{e.g. } \mathsf{Fun}, \mathsf{Set}_i \\ x\rho & = & \rho(x) & \\ (r\,s)\rho & = & r\rho \cdot s\rho & \\ (\lambda x t)\rho \cdot d & = & t(\rho, x = d) & \end{array}$$

- Variables $\mathsf{x}_1, \mathsf{x}_2 \in \mathsf{D}$ aka de Bruijn levels, generic values Coquand (1996).
- Neutral objects $x_i \cdot d_1 \cdot \ldots \cdot d_k$ are eliminations of variables aka atomic objects / accumulators.

Checking Type Equality

- Comparing type values

$\Delta \vdash X = X' \Uparrow \mathsf{Set} \leadsto i$	X and X' are equal types at level i
$\Delta \vdash e = e' \Downarrow X$	neutral e and e' are equal, inferring type X
$\Delta \vdash d = d' \Uparrow X$	d and d' are equal, checked at type X

- Roots:
  1. Setting of Coquand (1996)
  2. Type-directed η-equality of Harper & Pfenning (2005), extended to dependent types
  3. Implementations: Agdalight, Epigram 2

Algorithmic Equality

- *Type mode* $\Delta \vdash X = X' \Uparrow \mathsf{Set} \leadsto i$ (inputs: Δ, X, X', output: i or fail).

$$\frac{}{\Delta \vdash \mathsf{Set}_i = \mathsf{Set}_i \Uparrow \mathsf{Set} \leadsto i+1}$$

$$\frac{\Delta \vdash X = X' \Uparrow \mathsf{Set} \leadsto i \qquad \Delta, \mathsf{x}_\Delta : X \vdash F \cdot \mathsf{x}_\Delta = F' \cdot \mathsf{x}_\Delta \Uparrow \mathsf{Set} \leadsto j}{\Delta \vdash \mathsf{Fun}\, X\, F = \mathsf{Fun}\, X'\, F' \Uparrow \mathsf{Set} \leadsto \mathsf{max}(i, j)}$$

$$\frac{\Delta \vdash E = E' \Downarrow \mathsf{Set}_i}{\Delta \vdash E = E' \Uparrow \mathsf{Set} \leadsto i}$$

- Arbitrary choice: asymmetric.

Algorithmic Equality

Inference mode $\Delta \vdash e = e' \Downarrow X$ (inputs: Δ,e,e', output: X or fail).

$$\frac{}{\Delta \vdash x = x \Downarrow \Delta(x)} \qquad \frac{\Delta \vdash e = e' \Downarrow \mathsf{Fun}\, X\, F \qquad \Delta \vdash d = d' \Uparrow X}{\Delta \vdash e\, d = e'\, d' \Downarrow F \cdot d}$$

Checking mode $\Delta \vdash d = d' \Uparrow X$ (inputs: Δ, d, d', X, output: succeed or fail).

$$\frac{\Delta \vdash e = e' \Downarrow E_1 \qquad \Delta \vdash E_1 = E_2 \Downarrow \mathsf{Set}_i}{\Delta \vdash e = e' \Uparrow E_2}$$

$$\frac{\Delta, \mathsf{x}_\Delta : X \vdash f \cdot \mathsf{x}_\Delta = f' \cdot \mathsf{x}_\Delta \Uparrow F \cdot \mathsf{x}_\Delta}{\Delta \vdash f = f' \Uparrow \mathsf{Fun}\, X\, F} \qquad \frac{\Delta \vdash X = X' \Uparrow \mathsf{Set} \leadsto i}{\Delta \vdash X = X' \Uparrow \mathsf{Set}_j}\; i \leq j$$

Verification of Algorithmic Equality

- Completeness: Any two judgmentally equal expressions are recognized equal by the algorithm.
  $\vdash t = t' : A$ implies $\vdash t\rho_{\mathsf{id}} = t'\rho_{\mathsf{id}} \Uparrow A\rho_{\mathsf{id}}$.

- Soundness: Any two well-typed expressions recognized as equal are also judgmentally equal.
  $\vdash t, t' : A$ and $\vdash t\rho_{\mathsf{id}} = t'\rho_{\mathsf{id}} \Uparrow A\rho_{\mathsf{id}}$ imply $\vdash t = t' : A$.

- Termination: the equality algorithm terminates on all well-typed expressions.

Towards a Kripke model

- Completeness of algorithmic equality usually established via Kripke logical relation *(semantic equality)*

$$\Delta \vdash d = d' : X$$

- At base type X this could be defined as $\Delta \vdash d = d' \Uparrow X$.
- Should model declarative judgements.
- Problem: transitivity of algorithmic equality non-trivial because of asymmetries.
- Solution: two objects at base type shall be equal if they reify to the same term.

Contextual reification

- Reification converts values to η-long β-normal forms.
- Reification of neutral objects $x\,\vec{d}$ involves reification of arguments d_i at their types.
- Thus, must be parameterized by context Δ and type X.
- Structure similar to algorithmic equality.

$$\begin{array}{l} \Delta \vdash X \searrow A \Uparrow \mathsf{Set} \rightsquigarrow i \\ \Delta \vdash e \searrow u \Downarrow X \\ \Delta \vdash d \searrow t \Uparrow X \end{array}$$

- Reification of functions (η-expansion):

$$\frac{\Delta, x\!:\!X \vdash f \cdot x \searrow t \Uparrow F \cdot x}{\Delta \vdash f \searrow \lambda x t \Uparrow \mathsf{Fun}\, X\, F}$$

Completeness

- Objects that reify to the same term are algorithmically equal.

Lemma

If $\Delta \vdash d \searrow t \Uparrow X$ *and* $\Delta' \vdash d' \searrow t \Uparrow X'$ *then* $\Delta \vdash d = d' \Uparrow X$.

- Kripke logical relation between objects in a semantic typing environment.
  - for base types: $\Delta \vdash d : X \circledS \Delta' \vdash d' : X'$ iff $\Delta \vdash d \searrow t \Uparrow X$ and $\Delta' \vdash d' \searrow t \Uparrow X'$ for some t,
  - for function types: $\Delta \vdash f : \mathsf{Fun}\, X\, F \circledS \Delta' \vdash f' : \mathsf{Fun}\, X'\, F'$ iff $\hat{\Delta} \vdash d : X \circledS \hat{\Delta}' \vdash d' : X'$ implies $\hat{\Delta} \vdash f \cdot d : F \cdot d \circledS \hat{\Delta}' \vdash f' \cdot d' : F' \cdot d'$.
- Symmetric and transitive by construction.
- Semantic equality $\Delta \vdash d = d' : X$ iff $\Delta \vdash d : X \circledS \Delta \vdash d' : X$.

Validity

- Define $\Delta \vdash \rho = \rho' : \Gamma$ iff $\Delta \vdash \rho(x) = \rho'(x) : \Gamma(x)$ for all x.

Theorem (Fundamental theorem)

If $\Gamma \vdash t = t' : A$ and $\Delta \vdash \rho = \rho' : \Gamma$ then $\Delta \vdash t\rho = t'\rho' : A\rho$.

- Implies completeness of algorithmic equality.

Soundness

- Easy for algorithmic equality defined on *terms*.
- Uses substitution principle for declarative judgements.
- Substitution principle fails for algorithmic equality.

$$\frac{\Delta, \mathsf{x}_\Delta : X \vdash f \cdot \mathsf{x}_\Delta = f' \cdot \mathsf{x}_\Delta \Uparrow F \cdot \mathsf{x}_\Delta}{\Delta \vdash f = f' \Uparrow \mathsf{Fun}\, X\, F}$$

- But it should hold for all values that come from syntax.
- Need to strengthen our notion of semantic equality by incorporating substitutions (Coquand et al., 2005).

Strong Semantic Equality

- Equip D with reevaluation $d\rho \in \mathsf{D}$.
- Define *strong semantic equality* by

$$\Theta \models d = d' : X \iff \forall \Delta \vdash \rho = \rho' : \Theta.\ \Delta \vdash d\rho = d'\rho' : X\rho$$

- Algorithmic equality is sound for strong semantic equality.
- Strong semantic equality models declarative judgements.

Logical Relation between Syntax and Semantics

Theorem (Soundness)

If $\Gamma \vdash t, t' : A$ *and* $\Gamma\rho_{\mathsf{id}} \vdash t\rho_{\mathsf{id}} = t'\rho_{\mathsf{id}} \Uparrow A\rho_{\mathsf{id}}$ *then* $\Gamma \vdash t = t' : A$.

Proof.

Define a Kripke logical relation $\Gamma \vdash t : A \circledR \Delta \vdash d : X$ between syntax and semantics.

For base types X, it holds if $\Delta \vdash d \searrow t' \Uparrow X$ and $\Gamma \vdash t = t' : A$. □

Conclusions

- Verified $\beta\eta$-conversion test which scales to universes and large eliminations.
- Necessary tools came from Normalization-by-Evaluation.
- From the distance: algorithm is β-evaluation followed by η-expansion.
- Future work: scale to singleton types.

Related Work

- Martin-Löf 1975: NbE for Type Theory (weak conversion)
- Martin-Löf 2004: Talk on NbE (philosophical justification)
- Altenkirch Hofmann Streicher 1996: NbE for λ-free System F
- Gregoire Leroy 2002: β-normalization by compilation for CIC
- Coquand Pollack Takeyama 2003: LF with singleton types
- Danielsson 2006: strongly typed NbE for LF
- Altenkirch Chapman 2007: big step normalization

Strong Validity

- Define $\Delta \models \rho = \rho' : \Gamma$ iff $\Delta \models \rho(x) = \rho'(x) : \Gamma(x)$ for all x.

Theorem (Fundamental theorem)

If $\Gamma \vdash t = t' : A$ and $\Delta \models \rho = \rho' : \Gamma$ then $\Delta \models t\rho = t'\rho' : A\rho$.

- Implies completeness of algorithmic equality.

Example: A Regular Expression Matcher in Agda (N.A.Danielsson)

```
data RegExp : Set where
  0   : RegExp    -- Matches nothing.
  eps : RegExp    -- Matches the empty string.
  +   : RegExp -> RegExp -> RegExp  -- Choice.

data in : [ carrier ] -> RegExp -> Set where
  matches-eps : [] in eps
  matches-+l  : forall {xs re re'}
              -> xs in re  -> xs in (re + re')
  matches-+r  : forall {xs re re'}
              -> xs in re' -> xs in (re + re')
```

Example: A Regular Expression Matcher in Agda (N.A.Danielsson)

```
matches : (xs : [ carrier ]) -> (re : RegExp) ->
          Maybe (xs in re)
matches [] eps         = just matches-eps
matches xs (re + re') with matches xs re
... | just p  = just (matches-+l p)
... | nothing with matches xs re'
... | just p  = just (matches-+r) p)
... | nothing = nothing
```

T. Coquand (1996). 'An Algorithm for Type-Checking Dependent Types'. In *Mathematics of Program Construction. Selected Papers from the Third International Conference on the Mathematics of Program Construction (July 17–21, 1995, Kloster Irsee, Germany)*, vol. 26 of *Science of Computer Programming*, pp. 167–177. Elsevier Science.

T. Coquand, et al. (2005). 'A Logical Framework with Dependently Typed Records'. *Fundamenta Informaticae* **65**(1-2):113–134.

B. Grégoire & X. Leroy (2002). 'A compiled implementation of strong reduction'. In *Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming (ICFP '02), Pittsburgh, Pennsylvania, USA, October 4-6, 2002*, vol. 37 of *SIGPLAN Notices*, pp. 235–246. ACM Press.

R. Harper & F. Pfenning (2005). 'On Equivalence and Canonical Forms in the LF Type Theory'. *ACM Transactions on Computational Logic* **6**(1):61–101.