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`).
  
  \[
  \text{append} : (n \ m : \text{Nat}) \rightarrow \text{Vec} \ n \rightarrow \text{Vec} \ m \rightarrow \text{Vec} \ (\text{add} \ n \ m)
  \]
- Type equality can be established
  - automatically, e.g., \(\text{Vec} \ (\text{add} \ 0 \ m) = \text{Vec} \ m\) (by computation), or
  - by proof, e.g., \(\text{Vec} \ (\text{add} \ n \ m) = \text{Vec} \ (\text{add} \ m \ n)\).
- Goal: establish more equalities automatically.
Building $\eta$ into Definitional Equality

- Coq’s definitional equality is $\beta$ (+ $\delta$ + $\iota$).
- The stronger definitional equality, the fewer the user has to revert to equality proofs.
- Why not $\eta$? ($f = \lambda x. f \times \text{if } x \text{ new}$)
- Validates, for instance, $f = \text{comp } f \; \text{id}$.
- But $\eta$ 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 $\text{Fun } A \lambda x B \ (\equiv (x : A) \rightarrow B)$ with $\eta$.
  - Predicative universes $\text{Set}_0, \text{Set}_1, \ldots$.
  - Natural numbers.

- We handle large eliminations (types defined by cases and recursion), in contrast to Harper & Pfenning (2005).

- Scales to $\Sigma$ types with surjective pairing.

- Goal: handle all types with at most one constructor ($\Pi$, $\Sigma$, $1$, $0$, singleton types).

- Not a goal?: handle enumeration types ($2$, disjoint sums, $\ldots$).
Syntax of Terms and Types

- Lambda-calculus with constants

\[ r, s, t ::= c \mid x \mid \lambda x. t \mid r s \]

- \( c ::= N \) type of natural numbers
- \( z \) zero
- \( s \) successor
- \( \text{rec} \) primitive recursion
- \( \text{Fun} \) function space constructor
- \( \text{Set}_i \) universe of sets of level \( i \)

- \( \Pi x : A.B \) (Agda: \((x : A) \to B\)) is written \( \text{Fun } A (\lambda x.B) \).
Judgements

• Essential judgements

\[ \Gamma \vdash t : A \quad t \text{ has type } A \text{ in } \Gamma \]
\[ \Gamma \vdash t = t' : A \quad t \text{ and } t' \text{ are equal expressions of type } A \text{ in } \Gamma \]

• Typing of functions:

\[ \Gamma, x : A \vdash t : B \quad \Gamma \vdash \lambda x.t : \text{Fun } A(\lambda x.B) \]
\[ \Gamma \vdash r : \text{Fun } A(\lambda x.B) \quad \Gamma \vdash s : A \]
\[ \Gamma \vdash r s : B[s/x] \]
Set formation rules

- **Small types (sets):**
  
  \[
  \begin{align*}
  \Gamma & \vdash A : \text{Set}_i & \Gamma, x : A & \vdash B : \text{Set}_i \\
  \Gamma & \vdash N : \text{Set}_0 & \Gamma & \vdash \text{Fun} A (\lambda x. B) : \text{Set}_i
  \end{align*}
  \]

- **Set_0** includes types defined by recursion like \( \text{Vec} A n \).

- **(Large) types:**
  
  \[
  \begin{align*}
  \Gamma & \vdash A : \text{Set}_i \\
  \Gamma & \vdash A : \text{Set}_{i+1} & \Gamma & \vdash \text{Set}_i : \text{Set}_{i+1}
  \end{align*}
  \]

- E.g., \( \text{Fun Set}_0 (\lambda A. A \to (N \to A)) : \text{Set}_1 \).
  
  In Agda: \((A : \text{Set}) \to A \to N \to A : \text{Set}1\).
Equality

- Conversion rule:

\[
\frac{\Gamma \vdash t : A \quad \Gamma \vdash A = A' : \text{Set}}{\Gamma \vdash t : A'}
\]

- Type checking requires checking type equality!

- Equality axioms:

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

\[
(\eta) \quad \frac{\Gamma \vdash t : \text{Fun } A (\lambda x. B)}{\Gamma \vdash (\lambda x. t x) = t : \text{Fun } A (\lambda x. B)} \quad x \not\in \text{FV}(t)
\]

- Add computation axioms for primitive recursion.
The Type Checking Task

- Input a sequence of typed definitions in $\beta$-normal form

  \[
  \begin{align*}
  x_0 & : A_0 = t_0 \\
  & \vdots \\
  x_{n-1} & : A_{n-1} = t_{n-1}
  \end{align*}
  \]

- 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 $\cdot : D \times D \to D$
  2. Evaluation $\cdot : \text{Exp} \times (\text{Var} \to D) \to D$.

- **Laws:**
  
  \[
  \begin{align*}
  c\rho &= c & \text{e.g. Fun, Set}_i \\
  x\rho &= \rho(x) \\
  (r \cdot s)\rho &= r\rho \cdot s\rho \\
  (\lambda x t)\rho \cdot d &= t(\rho, x = d)
  \end{align*}
  \]

- **Variables** $x_1, x_2 \in D$ 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

\( \Delta \vdash X = X' \uparrow \text{Set} \rightsquigarrow i \)  \( X \) 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 \( \eta \)-equality of Harper & Pfenning (2005), extended to dependent types
3. Implementations: Agdalight, Epigram 2
Algorithmic Equality

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

\[
\Delta \vdash \text{Set}_i = \text{Set}_i \uparrow \text{Set} \leadsto i + 1
\]

\[
\Delta \vdash X = X' \uparrow \text{Set} \leadsto i \quad \Delta, x_\Delta : X \vdash F \cdot x_\Delta = F' \cdot x_\Delta \uparrow \text{Set} \leadsto j
\]

\[
\Delta \vdash \text{Fun } X \ F = \text{Fun } X' \ F' \uparrow \text{Set} \leadsto \max(i, j)
\]

\[
\Delta \vdash E = E' \downarrow \text{Set}_i
\]

\[
\Delta \vdash E = E' \uparrow \text{Set} \leadsto i
\]

- Arbitrary choice: asymmetric.
Algorithmic Equality

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

\[
\Delta \vdash x = x \Downarrow \Delta(x) \quad \Delta \vdash e = e' \Downarrow \text{Fun} X F \quad \Delta \vdash d = d' \Uparrow X \quad \Delta \vdash e d = e' d' \Downarrow F \cdot d
\]

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

\[
\Delta \vdash e = e' \Downarrow E_1 \quad \Delta \vdash E_1 = E_2 \Downarrow \text{Set}_i \\
\Delta \vdash e = e' \Uparrow E_2 \\
\Delta, x_\Delta : X \vdash f \cdot x_\Delta = f' \cdot x_\Delta \Uparrow F \cdot x_\Delta \\
\Delta \vdash f = f' \Uparrow \text{Fun} X F \\
\Delta \vdash X = X' \Uparrow \text{Set} \rightsquigarrow i \\
i \leq j
\]
Verification of Algorithmic Equality

- Completeness: Any two judgmentally equal expressions are recognized equal by the algorithm.
  \[ \vdash t = t' : A \text{ implies } \vdash t\rho_{id} = t'\rho_{id} \upharpoonright A\rho_{id}. \]

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

- Termination: the equality algorithm terminates on all well-typed expressions.
Towards a Kripke model

- Completeness of algorithmic equality usually established via Kripke logical relation \((\text{semantic equality})\)

\[
\Delta \vdash d = d' : X
\]

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

\[
\begin{align*}
\Delta \vdash X \downarrow A \uparrow \text{Set} \leadsto i \\
\Delta \vdash e \downarrow u \downarrow X \\
\Delta \vdash d \downarrow t \uparrow X
\end{align*}
\]

- Reification of functions ($\eta$-expansion):

\[
\frac{
\Delta, x : X \vdash f \cdot x \downarrow t \uparrow F \cdot x
}{\Delta \vdash f \downarrow \lambda x t \uparrow \text{Fun } X F}
\]
Completeness

Objects that reify to the same term are algorithmically equal.

**Lemma**

If $\Delta \vdash d \downarrow t \uparrow X$ and $\Delta' \vdash d' \downarrow t \uparrow X'$ then $\Delta \vdash d = d' \uparrow X$.

- Kripke logical relation between objects in a semantic typing environment.
  - for base types: $\Delta \vdash d : X \circ \Delta' \vdash d' : X'$ iff $\Delta \vdash d \downarrow t \uparrow X$ and $\Delta' \vdash d' \downarrow t \uparrow X'$ for some $t$,
  - for function types: $\Delta \vdash f : \text{Fun} X F \circ \Delta' \vdash f' : \text{Fun} X' F'$ iff $\hat{\Delta} \vdash d : X \circ \hat{\Delta}' \vdash d' : X'$ implies $\hat{\Delta} \vdash f \cdot d : F \cdot d \circ \hat{\Delta}' \vdash f' \cdot d' : F' \cdot d'$.

- Symmetric and transitive by construction.

- Semantic equality $\Delta \vdash d = d' : X$ iff $\Delta \vdash d : X \circ \Delta \vdash d' : X$. 
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.

\[ \Delta, x_\Delta : X \vdash f \cdot x_\Delta = f' \cdot x_\Delta \uparrow F \cdot x_\Delta \]
\[ \Delta \vdash f = f' \uparrow \text{Fun } X F \]

- 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 \( \mathcal{D} \) with reevaluation \( d\rho \in \mathcal{D} \).
- Define strong semantic equality by

\[
\Theta \vdash d = d' : X \iff \forall \Delta \vdash \rho = \rho' : \Theta. \Delta \vdash d\rho = d'\rho' : X\rho
\]

- Algorithmic equality is sound for strong semantic equality.
- Strong semantic equality models declarative judgements.
Theorem (Soundness)

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

Proof.

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

For base types $X$, it holds if $\Delta \vdash d \searrow t' \uparrow X$ and $\Gamma \vdash t = t' : A$. $\square$
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 $\beta$-evaluation followed by $\eta$-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 $\lambda$-free System F
- Gregoire Leroy 2002: $\beta$-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.

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

