Type Checking Dependent Types

Gilles Barthe

INRIA Sophia-Antipolis, France
Conversion is useful for small proofs and automation
How to do type-checking for a type theory with conversion?
Conversion rule

1. User point of view:
   Why we need it and how we can use it?

2. Implementor point of view:
   - Type inference Algorithm with conversion rule
   - How to get an efficient conversion test
Why we need it?

Because we do not want to do large and boring proofs $2 + 2 = 4$ in a deduction style:

\[
\begin{align*}
\text{eqTrans} & \quad 2 + 2 = S(1 + 2) \\
S(1 + 2) = 4 & \quad \Rightarrow \quad 2 + 2 = 4
\end{align*}
\]

\[
\begin{align*}
\text{eqTrans} & \quad 1 + 2 = S(0 + 2) \\
S(0 + 2) = 3 & \quad \Rightarrow \quad 1 + 2 = 3
\end{align*}
\]

\[
\begin{align*}
\text{eqS} & \quad 0 + 2 = 2 \\
S(0 + 2) & = 3
\end{align*}
\]

\[
2 + 2 = 4
\]

\[
\text{eqS} \quad : x = y \Rightarrow Sx = Sy
\]

\[
\text{eqTrans} : x = y \Rightarrow y = z \Rightarrow x = z
\]
How to prove $2 + 2 = 4$

Computational style: use the reduction rules

\[
\begin{align*}
0 + m & \rightarrow m \\
Sn + m & \rightarrow S(n + m)
\end{align*}
\]

$2 + 2 \rightarrow S(1 + 2) \rightarrow SS(0 + 2) \rightarrow SS(2)$

Reason modulo rewriting rules:

\[
\begin{align*}
4 = 4 & \quad 2 + 2 \xrightarrow{\ast} 4 \\
\frac{2 + 2 = 4}{2 + 2} = 4
\end{align*}
\]
Use computations instead of deductions!

**Principle**

- A predicate $P : T \rightarrow \text{Prop}$
- A decision procedure $f : T \rightarrow \text{bool}$
- A correctness lemma $C : \forall x : T. \, f \, x = \text{true} \rightarrow P \, x$

If $f \, a$ reduces to $\text{true}$, then $C \, a \, (\text{refl\_eq} \, \text{true})$ is a proof of $P \, a$
Examples of reflexivity

- Equational reasoning
- Primality
- Presburger arithmetic
- Model checking
- 4-color theorem
Problem: given two terms \( t_1 \) and \( t_2 \) of the same type, decide whether they are equal.

Approach:
- view \( t_1 \) and \( t_2 \) as interpretations of expressions \( e_1 \) and \( e_2 \) of an equational theory
- check whether \( e_1 \equiv e_2 \) is provable in the equational theory
- if so, conclude that \( t_1 \) is equal to \( t_2 \)

Provability in the equational theory is established by comparing normal forms of expressions.
Example

To show

$$2 \ast \sin(x) \ast x = x \ast \sin(x) + \sin(x) \ast x + 0 \ast x$$

Synthesize automatically

$$\begin{array}{c|c}
  e_1 & 2 \ast v \ast v' \\
  e_2 & v' \ast v + v \ast v' + 0 \ast v'
\end{array}$$

Normalize expressions and return proof
To show

\[ 2 \times \sin(x) \times x = x \times \sin(x) + \sin(x') \times x + 0 \times x \]

Synthesize automatically

\[
\begin{array}{c|c}
  \text{e}_1 & 2 \times v \times v' \\
  \text{e}_2 & v' \times v + v'' \times v' + 0 \times v'
\end{array}
\]

normalize expressions and return proof obligation

\[ \sin(x) = \sin(x') \]
Pocklington’s criterion (formal proof by Oostdijk and Caprotti):

Let $n$ be a positive integer, if

- $n - 1 = q \ p_1 \ldots p_t$ where $p_i$ are prime numbers
- there exists $a$ such that
  \[
  \begin{aligned}
  a^{n-1} &\equiv 1 \pmod{n} \\
  \gcd(a^{n-1/p_i} - 1, n) &= 1 \text{ for } i = 1 \ldots t
  \end{aligned}
  \]
- $p_1 \cdot p_2 \ldots p_t \geq \sqrt{n}$

then $n$ is prime.

Using deduction style, it takes 18509 lines to prove the primality of

\[20988936657440586486151264256610222593863921\]

Can we use reflexivity?
A 969 digits number. The proof is 8461 chars!
Conversion is useful, but sometimes it is not strong enough. Vec (n+m) A is not convertible with Vec (m+n) A. Some of these problems can be solved by a non-standard equality (John Major equality).

For some extensions e.g. records conversion must be typed.
Type checking vs type inference

- Type-checking: is a given judgement $\Gamma \vdash M : B$ derivable?
- Type-inference: given $\Gamma$ and $M$, is there a $B$ s.t. $\Gamma \vdash M : B$ derivable?

Rule of thumb:

- TC is decidable for complex type systems à la Church: e.g. for the Calculus of Constructions (Coquand 1985)
- TC is undecidable for complex type systems à la Curry: e.g. for second-order type assignment system (Wells 1994) and domain-free type system
Due to R. Pollack. Ignores issues of convertibility.

- Weak-head reduction $\rightarrow_{wh}$ is defined by

$$ (\lambda x:A. \ P) \ Q \rightarrow_{wh} \ P\{x := Q\} $$

- The relation $\Gamma \vdash_{nat} M : A$ is defined next slide

- Let $\rightarrow \rho$ be a relation on $\mathcal{T}$. $\Gamma \vdash_{nat} M : \rightarrow \rho A$ if

$$ \exists A' \in \mathcal{T}. \ \begin{cases} \Gamma \vdash_{nat} M : A' \\ A \rightarrow \rho A' \end{cases} $$
Typing rules

\[
\begin{align*}
\langle \rangle & \vdash_{\text{nat}} s_1 : s_2 \\
\Gamma & \vdash_{\text{nat}} A : \rightsquigarrow_{\text{wh}} s \\
\Gamma, x : A & \vdash_{\text{nat}} x : A \\
\Gamma & \vdash_{\text{nat}} A : B \quad \Gamma & \vdash_{\text{nat}} C : \rightsquigarrow_{\text{wh}} s \\
\Gamma, x : C & \vdash_{\text{nat}} A : B \\
\Gamma & \vdash_{\text{nat}} A : \rightsquigarrow_{\text{wh}} s_1 \\
\Gamma, x : A & \vdash_{\text{nat}} B : \rightsquigarrow_{\text{wh}} s_2 \\
\Gamma & \vdash_{\text{nat}} (\Pi x : A. B) : s_3 \\
\Gamma & \vdash_{\text{nat}} F : \rightsquigarrow_{\text{wh}} (\Pi x : A'. B) \quad \Gamma & \vdash_{\text{nat}} a : A \\
\Gamma & \vdash_{\text{nat}} F a : B\{x := a\} \\
\Gamma, x : A & \vdash_{\text{nat}} b : B \quad \Gamma & \vdash_{\text{nat}} \Pi x : A. B : s \\
\Gamma & \vdash_{\text{nat}} \lambda x : A. b : \Pi x : A. B
\end{align*}
\]
Soundness and completeness?

- **Soundness**

\[ \Gamma \vdash_{\text{nat}} M : A \implies \Gamma \vdash M : A \]

- **Completeness?**

\[ \Gamma \vdash M : A \implies \exists A' \in T. \left\{ \begin{array}{l} \Gamma \vdash_{\text{nat}} M : A' \\ A \equiv_{\beta} A' \end{array} \right. \]
Induction on the structure of derivations fails in the abstraction case!

\[ \Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x : A. B) : s \]
\[ \Gamma \vdash \lambda x : A. b : \Pi x : A. B \]

By induction hypothesis

\[ \exists B' \in T. \left\{ \begin{array}{l}
\Gamma, x : A \vdash_{\text{nat}} b : B' \\
B =_{\beta} B'
\end{array} \right. \]

and

\[ \exists C \in T. \left\{ \begin{array}{l}
\Gamma \vdash_{\text{nat}} (\Pi x : A. B) : C \\
s =_{\beta} C
\end{array} \right. \]

But we need

\[ \exists C \in T. \left\{ \begin{array}{l}
\Gamma \vdash_{\text{nat}} (\Pi x : A. B') : C \\
s =_{\beta} C
\end{array} \right. \]
The premise of the abstraction rule is needed but contains much redundancy!
From $\Gamma, x : A \vdash M : B$ we already know that $A$ and $B$ are legal types, i.e.

$$\Gamma \vdash A : s_1$$

$$\Gamma, x : A \vdash B : s_2$$

We only need to find what are the possibilities for $s_1$ and $s_2$ and look at $R$
Different solutions according to the class of type systems considered
Checking convertibility

Testing convertibility of two terms is decidable for type systems whose expressions are (strongly) normalizing, using confluence of β-reduction.

\[
\begin{array}{c}
A \quad \equiv \quad B \\
\downarrow \quad \quad \quad \downarrow \\
\ast \quad \quad \quad \ast \\
M \\
\downarrow \quad \quad \downarrow \\
\ast \quad \quad \ast \\
NF(A) \quad = \quad NF(B)
\end{array}
\]

How to check convertibility efficiently?
A finer look at $\beta$-reduction

$\beta$-reduction = $\beta$-rule + contextual closure:

$$
\frac{t \xrightarrow{\beta} t'}{C(t) \xrightarrow{\beta} C(t')}
$$

What is a context?

<table>
<thead>
<tr>
<th>Weak reduction</th>
<th>$C ::= []N \mid M[] \mid \lambda x:]. M \mid \Pi x:]. B$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong reduction</td>
<td>$C ::= []N \mid M[] \mid \lambda x:]. M \mid \Pi x:]. B$</td>
</tr>
<tr>
<td></td>
<td>$\mid \lambda x:T.[] \mid \Pi x:A.[]$</td>
</tr>
</tbody>
</table>

- NF: Normal form using strong reduction
- WNF: Normal form using weak reduction
- WHNF: Normal form using $C ::= []N$
Checking conversion in the $\lambda$-calculus

terms $\quad t ::= x \mid \lambda x. t \mid t \ t$

values (WNF) $\quad v ::= \lambda x. t \mid x \ v_1 \ldots v_n$

Conversion algorithm:

\[
\begin{align*}
\frac{t_1 = t_2}{t_1 \equiv t_2} & & \frac{\text{WNF}(t_1) \equiv \text{WNF}(t_2)}{t_1 \equiv t_2} \\
\frac{v_1 = v_2}{v_1 \equiv v_2} & & \frac{x = y \quad v_i \equiv w_i}{x \ v_1 \ldots v_n \equiv y \ w_1 \ldots w_n} \\
\frac{\text{WNF}(\lambda x. M \ z) \equiv \text{WNF}(\lambda y. M' \ z) \quad z \text{ fresh}}{\lambda x. M \equiv \lambda y. M'}
\end{align*}
\]

Correctness of algorithm: $t =_\beta t' \text{ iff } t \equiv t'$
Computing the WNF

\[
\text{type term} = \\
\text{Var of var} \mid \text{Abs of var} \ast \text{term} \mid \text{App of term} \ast \text{term}
\]

let rec wnf t =
match t with
| Var _ | Abs _ \to t
| App(t1, t2) \to
let v1 = wnf t1 in
let v2 = wnf t2 in
match v1 with
| Abs(x,u) \to wnf (subst u x v2)
| _ \to App(v1,v2)
Computing WNF is similar to execution of ML-like program

\[
\begin{align*}
\lambda\text{-term} & \xrightarrow{\text{Compilation}} \text{bytecode} & \text{Execution} \xrightarrow{\text{Abs. Machine}} \text{value}
\end{align*}
\]

**Problem**

Compilation techniques (abstract machines) only work for closed terms. How to compute \( \text{WNF}(\lambda x. Mz) \)?
ZINC abstract machine

ZINC : a stack based abstract machine in call by value
- Instructions : Acc, Closure, Grab, Pushra, Apply, Return
- Values $\nu$: represented as closures $[c, e]$
- Environment $e$: maps variables to values $[\nu_1; \ldots; \nu_n]$

Components of the machine:
- $c$ code pointer
- $e$ environment
- $s$ stack (arguments + intermediate results + return address)
- $n$ number of arguments available on the top of $s$
Compilation scheme and execution of variables

Compilation scheme

\[ [t]k \leadsto c \]

The resulting code \( c \) computes the value corresponding to \( t \), pushes it on top of the stack, then restarts the execution of \( k \)

\[ [x]k = \text{Acc}(i); \ k \]

where \( i \) is the deBruijn index of \( x \)

<table>
<thead>
<tr>
<th>Code</th>
<th>Env</th>
<th>Stack</th>
<th># args</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acc( (i) ); ( k )</td>
<td>( e )</td>
<td>( s )</td>
<td>( n )</td>
</tr>
<tr>
<td>( k )</td>
<td>( e )</td>
<td>( e(i).s )</td>
<td>( n )</td>
</tr>
</tbody>
</table>
Compilation and execution of applications

\[
[f \ a_1 \ldots \ a_i]_k = \text{Pushra}(k); \\
[a_i] \ldots [a_1] [f] \text{Apply}(i)
\]

<table>
<thead>
<tr>
<th>Code</th>
<th>Env</th>
<th>Stack</th>
<th>#args</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pushra(k); (c)</td>
<td>(e)</td>
<td>(s\ n)</td>
<td></td>
</tr>
<tr>
<td>(c)</td>
<td>(e)</td>
<td>(\langle k, e, n \rangle.s\ n)</td>
<td></td>
</tr>
<tr>
<td>Apply(i)</td>
<td>(e)</td>
<td>([c, e'].v_1 \ldots v_j.\langle k, e, n \rangle.s \ n)</td>
<td></td>
</tr>
<tr>
<td>(c)</td>
<td>(e')</td>
<td>(v_1 \ldots v_j.\langle k, e, n \rangle.s \ i)</td>
<td></td>
</tr>
</tbody>
</table>
Compilation and execution of functions

\[ \lambda x_1 \ldots \lambda x_n.t \] \] \( k = \text{Closure}(c); k \\
\]
\[ c = \{ \text{Grab}; \ldots; \text{Grab}; [t]\} \text{Return} \]

\[ n \text{ times} \]

<table>
<thead>
<tr>
<th>Code</th>
<th>Env</th>
<th>Stack</th>
<th>#args</th>
</tr>
</thead>
<tbody>
<tr>
<td>Closure(c); k</td>
<td>e</td>
<td>s</td>
<td>n</td>
</tr>
<tr>
<td>k</td>
<td>e</td>
<td>[c, e].s</td>
<td>n</td>
</tr>
<tr>
<td>Grab; k</td>
<td>e</td>
<td>v.s</td>
<td>n + 1</td>
</tr>
<tr>
<td>k</td>
<td>v.e</td>
<td>s</td>
<td>n</td>
</tr>
<tr>
<td>Return</td>
<td>e</td>
<td>v.\langle k, e', n\rangle.s</td>
<td>0</td>
</tr>
<tr>
<td>k</td>
<td>e'</td>
<td>v.s</td>
<td>n</td>
</tr>
</tbody>
</table>
Compilation with free variables

<table>
<thead>
<tr>
<th>Code</th>
<th>Env</th>
<th>Stack</th>
<th>#args</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Acc}(i); k$</td>
<td>$e$</td>
<td>$s$</td>
<td>$n$</td>
</tr>
<tr>
<td>$k$</td>
<td>$e$</td>
<td>$e(i).s$</td>
<td>$n$</td>
</tr>
</tbody>
</table>

Free variables have no associated value in the environment

**The trick**

Add values for free variables

**Issues:**

- What should be the value associated to a free variable?
- What happens when this value is applied?
Symbolic calculus

Extension of calculus that allows to provide the computational behavior of a free variable

Terms \( t ::= x \mid t \ t \mid v \)

Values \( v ::= \lambda x. t \mid [k] \)

Accumulators \( k ::= \tilde{x} \mid k \ v \)

Reduction rules:

\[(\lambda x.t) \ v \longrightarrow t\{x := v\}\]
\[[k] \ v \longrightarrow [k \ v]\]

The value associated to a free variable is a function that accumulates its arguments
In an extended abstract machine whose moves implement exactly symbolic reduction

Key feature of encoding: the representation of \([k]\) looks like a function

\[ \Rightarrow \text{No need to test at application time whether the function is a closure or an accumulator} \]

\[ \Rightarrow \text{No overhead on evaluation of closed terms} \]
Correctness of method

How do we know that the method is correct, i.e. sound and complete

- Some information is erased
- Some information is compiled

We must prove the correctness of each
Domain-free terms

- Do not carry type of variable in $\lambda$-abstraction:

$$\mathcal{E} = V \mid S \mid \mathcal{E}\mathcal{E} \mid \lambda V.\mathcal{E} \mid \Pi V : \mathcal{E}\mathcal{E}$$

- Obvious notion of $\beta$-reduction and $\beta$-conversion

$$(\lambda x. M) \ N \rightarrow_\beta M\{x := N\}$$

- Obvious erasure function $\| : \mathcal{E} \rightarrow \mathcal{E}$ with

$$\|\lambda x : A. M\| = \lambda x. \|M\|$$

Domain-free terms are used in domain-free type systems
One can show that domain-free type systems coincide with type systems à la Church for normalizing type theories (counter-example with non-normalizing type theory).

\[ \Gamma \vdash A : s \text{ and } \Gamma \vdash A' : s', \text{ we have} \]
\[ A \equiv_{\beta} A' \quad \text{iff} \quad |A| \equiv_{\beta} |A'| \]
The compilation method extends to inductive types using an extended abstract machine that deals with case analysis and letrec. (The method does not introduce overhead for $\iota$-reduction.)

The erasure function omits parameters in inductive definitions so checking convertibility between $\text{cons } A\ a\ l$ and $\text{cons } A'\ a'\ l'$ does not involve checking convertibility between $A$ and $A'$. (The correctness result for domain-free checking extends to inductive types)
### Experimental results: 4-color theorem

<table>
<thead>
<tr>
<th>Perimeter</th>
<th>Coq</th>
<th>Coq-vm</th>
<th>OCaml bytecode</th>
<th>OCaml natif</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>56.7s</td>
<td>1.68s</td>
<td>1.18s</td>
<td>0.30s</td>
</tr>
<tr>
<td>12</td>
<td>259s</td>
<td>6.50s</td>
<td>6.18s</td>
<td>1.92s</td>
</tr>
<tr>
<td>13</td>
<td>680s</td>
<td>14.8s</td>
<td>15.5s</td>
<td>4.11s</td>
</tr>
<tr>
<td>Size</td>
<td>time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1234567891 (10)</td>
<td>13.26 s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deductive : 3099</td>
<td>Reflexive : 58</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deductive : 18509</td>
<td>Reflexive : 95</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20988936657440586486151264256610222593863921 (44)</td>
<td>1862.52 s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deductive : 18509</td>
<td>Reflexive : 95</td>
<td>21.30 s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>