Type Checking Dependent Types

Gilles Barthe

INRIA Sophia-Antipolis, France

Summary

- 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:

$$\dfrac{\dfrac{\overline{\mathrm{eqTrans}} \quad \overline{2+2=S(1+2)}}{S(1+2)=4 \Rightarrow 2+2=4} \quad \dfrac{\overline{\mathrm{eqS}} \quad \dfrac{\dfrac{\overline{\mathrm{eqTrans}} \quad \overline{1+2=S(0+2)}}{S(0+2)=3 \Rightarrow 1+2=3} \quad \dfrac{\overline{\mathrm{eqS}} \quad \overline{0+2=2}}{S(0+2)=3}}{1+2=3}}{S(1+2)=4}}{2+2=4}$$

$$\begin{array}{ll} \mathrm{eqS} & : x=y \Rightarrow Sx=Sy \\ \mathrm{eqTrans} & : x=y \Rightarrow y=z \Rightarrow x=z \end{array}$$

How to prove $2+2=4$

Computational style: use the reduction rules

$$\begin{array}{rcl} 0+m & \longrightarrow & m \\ Sn+m & \longrightarrow & S(n+m) \end{array}$$

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

Reason modulo rewriting rules:

$$\frac{4=4 \quad 2+2 \xrightarrow{*} 4}{2+2=4}$$

Reflexivity

Use computations instead of deductions!

Principle

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

If $f\ a$ reduces to $\texttt{true}$, then $C\ a\ (\texttt{refl_eq true})$ is a proof of $P\ a$

$$\dfrac{\dfrac{\vdots}{\vdash C\ a : f\ a = \texttt{true} \to P\ a} \qquad \dfrac{\vdash \texttt{refl_eq true} : \texttt{true} = \texttt{true} \qquad \color{red}{\texttt{true} = \texttt{true} \equiv f\ a = \texttt{true}}}{\vdash \texttt{refl_eq true} : f\ a = \texttt{true}}}{\vdash C\ a\ (\texttt{refl_eq true}) : P\ a}$$

Examples of reflexivity

- Equational reasoning
- Primality
- Presburger arithmetic
- Model checking
- 4-color theorem

Equational reasoning: ring

- 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 $e_1 \doteq e_2$ 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

$$2 * sin(x) * x = x * sin(x) + sin(x) * x + 0 * x$$

Synthesize automatically

e_1	$2 * v * v'$
e_2	$v' * v + v * v' + 0 * v'$

Normalize expressions and return proof

Generalized tactic

To show

$$2 * sin(x) * x = x * sin(x) + sin(x') * x + 0 * x$$

Synthesize automatically

e_1	$2 * v * v'$
e_2	$v' * v + v'' * v' + 0 * v'$

normalize expressions and return proof obligation

$$sin(x) = sin(x')$$

Example: primality proof

Pocklington 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
$$\begin{cases} a^{n-1} = 1 \pmod{n} \\ \gcd(a^{\frac{n-1}{p_i}} - 1, n) = 1 \text{ for } i = 1 \ldots t \end{cases}$$
- $p_1.p_2 \ldots p_t \geq \sqrt{n}$

then n is prime.

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

20988936657440586486151264256610222593863921

Can we use reflexivity?

18th Mersenne number: $2^{3217} - 1$

2591170860132026277762467679224415309418188875531254
...

Is conversion good enough

- 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 Γ 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

Syntax-directed algorithm

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 $\Gamma \vdash_{\mathsf{nat}} M : A$ is defined next slide
- Let $\twoheadrightarrow \rho$ be a relation on $\mathcal{T}$. $\Gamma \vdash_{\mathsf{nat}} M :\twoheadrightarrow \rho A$ if

$$\exists A' \in \mathcal{T}.\ \left\{ \begin{array}{l} \Gamma \vdash_{\mathsf{nat}} M : A' \\ A \twoheadrightarrow \rho A' \end{array} \right.$$

Typing rules

$$\langle\rangle \vdash_{\mathrm{nat}} s_1 : s_2 \qquad (s_1, s_2) \in \mathcal{A}$$

$$\frac{\Gamma \vdash_{\mathrm{nat}} A :\twoheadrightarrow_{wh} s}{\Gamma, x : A \vdash_{\mathrm{nat}} x : A} \qquad x \in V \setminus dom(\Gamma)$$

$$\frac{\Gamma \vdash_{\mathrm{nat}} A : B \quad \Gamma \vdash_{\mathrm{nat}} C :\twoheadrightarrow_{wh} s}{\Gamma, x : C \vdash_{\mathrm{nat}} A : B} \qquad \begin{array}{l} x \in V \setminus dom(\Gamma) \\ A \in V \cup \mathcal{S} \end{array}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{nat}} A :\twoheadrightarrow_{wh} s_1 \\ \Gamma, x : A \vdash_{\mathrm{nat}} B :\twoheadrightarrow_{wh} s_2\end{array}}{\Gamma \vdash_{\mathrm{nat}} (\Pi x{:}A.\ B) : s_3} \qquad (s_1, s_2, s_3) \in \mathcal{R}$$

$$\frac{\Gamma \vdash_{\mathrm{nat}} F :\twoheadrightarrow_{wh} (\Pi x{:}A'.\ B) \quad \Gamma \vdash_{\mathrm{nat}} a : A}{\Gamma \vdash_{\mathrm{nat}} F\ a : B\{x := a\}} \qquad A =_\beta A'$$

$$\frac{\Gamma, x : A \vdash_{\mathrm{nat}} b : B \quad \Gamma \vdash_{\mathrm{nat}} \Pi x{:}A.\ B : s}{\Gamma \vdash_{\mathrm{nat}} \lambda x{:}A.\ b : \Pi x{:}A.\ B}$$

Soundness and completeness?

- Soundness

$$\Gamma \vdash_{\mathsf{nat}} M : A \quad \Rightarrow \quad \Gamma \vdash M : A$$

- Completeness?

$$\Gamma \vdash M : A \quad \Rightarrow \quad \exists A' \in \mathcal{T}. \begin{cases} \Gamma \vdash_{\mathsf{nat}} M : A' \\ A =_\beta A' \end{cases}$$

What goes wrong?

Induction on the structure of derivations fails in the abstraction case!

$$\frac{\Gamma, x : A \vdash b : B \quad \Gamma \vdash (\Pi x{:}A.\ B) : s}{\Gamma \vdash \lambda x{:}A.\ b : \Pi x{:}A.\ B}$$

By induction hypothesis

$$\exists B' \in \mathcal{T}.\ \begin{cases} \Gamma, x : A \vdash_{\mathsf{nat}} b : B' \\ B =_\beta B' \end{cases}$$

and

$$\exists C \in \mathcal{T}.\ \begin{cases} \Gamma \vdash_{\mathsf{nat}} (\Pi x{:}A.\ B) : C \\ s =_\beta C \end{cases}$$

But we need

$$\exists C \in \mathcal{T}.\ \begin{cases} \Gamma \vdash_{\mathsf{nat}} (\Pi x{:}A.\ B') : C \\ s =_\beta C \end{cases}$$

Observation

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.

$$\Gamma \vdash A : s_1$$

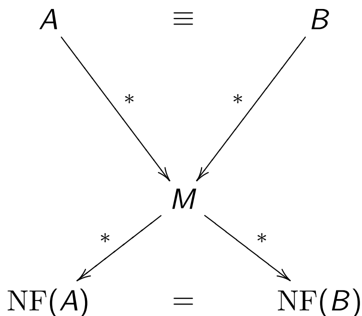
$$\Gamma, x : A \vdash B : s_2$$

We only need to find what are the possibilities for s_1 and s_2 and look at $\mathcal{R}$

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.



How to check convertibility efficiently?

A finer look at β-reduction

β-reduction $=$ β-rule $+$ contextual closure:

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

What is a context?

Weak reduction	C	$::=$	$[]\ N \mid M\ [] \mid \lambda x{:}[].\ M \mid \Pi x{:}[].\ B$
Strong reduction	C	$::=$	$[]\ N \mid M\ [] \mid \lambda x{:}[].\ M \mid \Pi x{:}[].\ B$
			$\mid \lambda x{:}T.\ [] \mid \Pi x{:}A.\ []$

- NF: Normal form using strong reduction
- WNF: Normal form using weak reduction
- WHNF: Normal form using $C ::= []\ N$

Checking conversion in the λ-calculus

$$\begin{array}{lrcl} \text{terms} & t & ::= & x \mid \lambda x.t \mid t\ t \\ \text{values(WNF)} & v & ::= & \lambda x.t \mid x\ v_1 \dots v_n \end{array}$$

Conversion algorithm:

$$\frac{t_1 = t_2}{t_1 \equiv t_2} \qquad \frac{\mathsf{WNF}(t_1) \equiv \mathsf{WNF}(t_2)}{t_1 \equiv t_2}$$

$$\frac{v_1 = v_2}{v_1 \equiv v_2} \qquad \frac{x = y \quad v_i \equiv w_i}{x\ v_1 \dots v_n \equiv y\ w_1 \dots w_n}$$

$$\frac{\mathsf{WNF}(\lambda x.M\ z) \equiv \mathsf{WNF}(\lambda y.M'\ z) \quad z \text{ fresh}}{\lambda x.M \equiv \lambda y.M'}$$

Correctness of algorithm: $t =_\beta t'$ iff $t \equiv t'$

Computing the WNF

```
type term =
Var of var | Abs of var * term | App of term * term

let rec wnf t =
match t with
| Var _ | Abs _ -> t
| App(t1, t2) ->
let v1 = wnf t1 in
let v2 = wnf t2 in
match v1 with
| Abs(x,u) -> wnf (subst u x v2)
| _ -> App(v1,v2)
```

WNF by compilation

Computing WNF is similar to execution of ML-like program

$$\lambda\text{-term} \xrightarrow{\text{Compilation}} \text{bytecode} \xrightarrow[\text{Abs. Machine}]{\text{Execution}} \text{value}$$

Problem

Compilation techniques (abstract machines) only work for closed terms. How to compute 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 v: represented as closures $[c, e]$
- Environment e : maps variables to values $[v_1; \ldots; v_n]$

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

$$[\![t]\!]k \rightsquigarrow c$$

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

$$[\![x]\!]k = \mathsf{Acc}(i);\ k$$

where i is the deBruijn index of x

Code	Env	Stack	# args
$\mathsf{Acc}(i); k$	e	s	n
k	e	$e(i).s$	n

Compilation and execution of applications

$$[\![f\ a_1\ \ldots\ a_i]\!]k \quad = \quad \mathsf{Pushra}(k);\ [\![a_i]\!]\ \ldots\ [\![a_1]\!]\ [\![f]\!]\ \mathsf{Apply}(i)$$

Code	Env	Stack	#args
$\mathsf{Pushra}(k); c$	e	s	n
c	e	$\langle k, e, n\rangle.s$	n
$\mathsf{Apply}(i)$	e	$[c, e'].v_1 \ldots v_i.\langle k, e, n\rangle.s$	n
c	e'	$v_1 \ldots v_i.\langle k, e, n\rangle.s$	i

Compilation and execution of functions

$$
\begin{aligned}
[\![\lambda x_1 \ldots \lambda x_n . t]\!] k &= \text{Closure}(c); k \\
c &= \underbrace{\text{Grab}; \ldots; \text{Grab}}_{n \text{ times}}; [\![t]\!]\text{Return}
\end{aligned}
$$

Code	Env	Stack	#args
Closure(c); k	e	s	n
k	e	$[c, e].s$	n
Grab; k	e	$v.s$	$n+1$
k	$v.e$	s	n
Return	e	$v.\langle k, e', n\rangle.s$	0
k	e'	$v.s$	n

Compilation with free variables

Code	Env	Stack	#args
$\mathrm{Acc}(i); k$	e	s	n
k	e	$e(i).s$	n

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

$$
\begin{array}{llll}
\text{Terms} & t & ::= & x \mid t\ t \mid v \\
\text{Values} & v & ::= & \lambda x.t \mid [k] \\
\text{Accumulators} & k & ::= & \tilde{x} \mid k\ v
\end{array}
$$

Reduction rules:

$$
\begin{array}{rcl}
(\lambda x.t)\ v & \longrightarrow & t\{x := v\} \\
[k]\ v & \longrightarrow & [k\ v]
\end{array}
$$

The value associated to a free variable is a function that accumulates its arguments

Encoding accumulator

- In an extended abstract machine whose moves implement exactly symbolic reduction
- Key feature of encoding: the representation of $[k]$ looks like a function
  - $\Rightarrow$ No need to test at application time whether the function is a closure or an accumulator
  - $\Rightarrow$ 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 λ-abstraction:

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

- Obvious notion of $\underline{\beta}$-reduction and $\underline{\beta}$-conversion

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

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

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

Domain-free terms are used in domain-free type systems

Convertibility vs. domain-free convertibility

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).

Soundness and Completeness

$\Gamma \vdash A : s$ and $\Gamma \vdash A' : s'$, we have

$$A =_\beta A' \qquad \text{iff} \qquad |A| =_{\underline{\beta}} |A'|$$

Extension to inductive types

- 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 ι-reduction.)
- The erasure function omits parameters in inductive definitions so checking convertibility between $cons\ A\ a\ l$ and $cons\ A'\ a'\ l'$ 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

Perimeter	Coq	Coq-vm	OCaml bytecode	OCaml natif
11	56.7s	1.68s	1.18s	0.30s
12	259s	6.50s	6.18s	1.92s
13	680s	14.8s	15.5s	4.11s

Prime numbers

		Size	time
		1234567891 (10)	
Deductive	:	3099	13.26 s
Reflexive	:	58	0.59 s
		20988936657440586486151264256610222593863921 (44)	
Deductive	:	18509	1862.52 s
Reflexive	:	95	21.30 s