Matita NG: reduction and type-checking

Andrea Asperti <asperti@cs.unibo.it>
Claudio Sacerdoti Coen <sacerdot@cs.unibo.it>
Enrico Tassi <tassi@cs.unibo.it>
Wilmer Ricciotti <ricciott@cs.unibo.it>

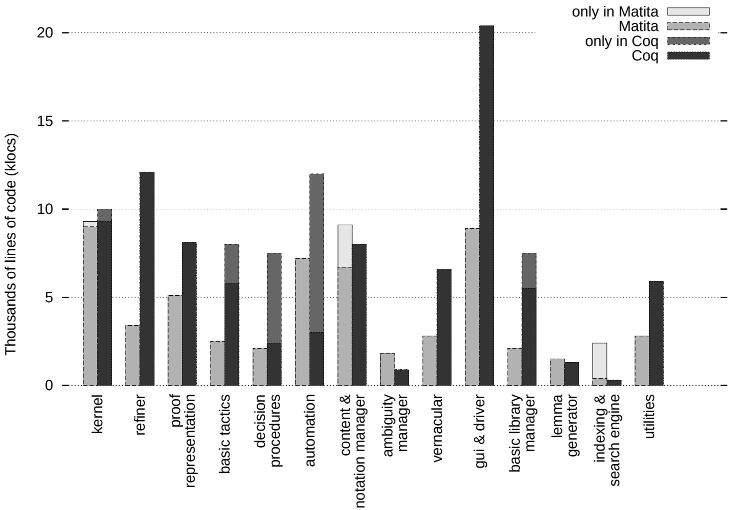
University of Bologna

Aussois, 12/05/2009

Outline

Outline

1. Matita NG
2. Reduction and Type Checking
3. Conclusions



only in Matita
Matita
only in Coq
Coq
Thousands of lines of code (klocs)
0
5
10
15
20
kernel
refiner
proof representation
basic tactics
decision procedures
automation
content & notation manager
ambiguity manager
vernacular
gui & driver
basic library manager
lemma generator
indexing & search engine
utilities

Towards Matita 1.xx (Matita NG)

Motivations:

- smaller code size
- simpler code size, easier to maintain and debug
- fix wrong design decisions; improve design decisions; try new design decisions
- experiment with new features (e.g. proof irrelevance)
- completely change the look&feel

Plan:

- entirely re-implement the system from inside out
- provide back&forth translation towards the old components (for immediate testing)
- side effect: generalize all logic independent components

Towards Matita 1.xx (Matita NG)

Already done:

- Reduction and Type Checking
  code: 2,783/7,783 = 36%, functions: 62/100 = 62%
  Special Issue on Interactive Proving and Proof Checking, Sadhana

Almost done:

- Unification and Refinement (Type Inference)
  code: 2,572/4,885 = 53%, functions: 23/41 = 56%

Work in progress:

- Tactic engine and tactics (huge improvement, but. . .)

Future work:

- Library management, consistency management, session management
- User interface

Outline

The Calculus of (Co)Inductive Constructions

$$
\begin{array}{rcl}
t & ::= & \lambda v : t.t \mid (t\ t) \mid \Pi v : t.t \mid \texttt{Let}\ v : t := t\ \texttt{in}\ t \mid x \mid s \mid c \\
 & \mid & \overline{\{i : t := \overline{k : t}\}} \mid co\overline{\{i : t := \overline{k : t}\}} \\
 & \mid & t.i \mid t.k \mid t.\texttt{Match}\ t\ \texttt{return}\ t\ \texttt{with}\ \vec{t} \\
 & \mid & \overline{\{f : t := t\}}.f_i \mid co\overline{\{g : t := t\}}.g_i \\
 & \mid & ?_i[\bar{t}] \\
s & ::= & \textbf{Prop} \mid \textbf{Set} \mid \textbf{Type}_i \\
d & ::= & c : t := t \mid c : t
\end{array}
$$

Universes: checked

Reduction: $\beta + \zeta + \delta + \iota +$ unfold $+$ co-unfold $+$ meta-subst

Conversion:

structural for (co)inductive types and (co)recursive functions (up to permutation); nominal for declarations; none for definitions

The Calculus of (Co)Inductive Constructions in Coq

$$
\begin{array}{lcl}
t & ::= & \lambda v : t.t \mid (t\ t) \mid \Pi v : t.t \mid \texttt{Let}\ v : t := t\ \texttt{in}\ t \mid x \mid s \mid c \\
 & \mid & i \mid k \mid \texttt{Match}\ t\ \texttt{in}\ i\ \texttt{return}\ t\ \texttt{with}\ \vec{t} \\
 & \mid & \{\overline{f : t := t}\}.f_i \mid co\{\overline{g : t := t}\}.g_i \\
 & \mid & ?_i[\bar{t}] \\
s & ::= & \textbf{Prop} \mid \textbf{Set} \mid \textbf{Max}\{\overline{\textbf{Type}_q \mid \textbf{Succ}(\textbf{Type}_q)}\} \\
d & ::= & c : t := t \mid c : t \mid \overline{\lambda x : t}.\{\overline{i : t := \overline{k : t}}\} \mid \overline{\lambda x : t}.co\{\overline{i : t := \overline{k : t}}\}
\end{array}
$$

Universes: inferred (constraint programming), algebraic

Reduction: $\beta + \zeta + \delta + \iota +$ unfold $+$ co-unfold $+$ meta-substitution

Conversion (ignoring modules):

structural for (co)recursive functions (up to permutation)

nominal for (co)inductive types and declarations;

(none for definitions);

The Calculus of (Co)Inductive Constructions in Matita

$$
\begin{array}{rcl}
t & ::= & \lambda v : t.t \mid (t\ t) \mid \Pi v : t.t \mid \texttt{Let}\ v : t := t\ \texttt{in}\ t \mid x \mid s \mid c \\
 & \mid & i \mid k \mid \texttt{Match}\ t\ \texttt{in}\ i\ \texttt{return}\ t\ \texttt{with}\ \vec{t} \\
 & \mid & f_i \mid g_i \\
 & \mid & ?_i[s, \bar{t}] \mid ?_i(s, n) \\
s & ::= & \textbf{Prop} \mid \textbf{Set} \mid \textbf{Max}\{\overline{\textbf{Type}_u \mid \textbf{Succ}(\textbf{Type}_u)}\} \\
d & ::= & c : t := t \mid c : t \mid \overline{\lambda x : t}.\{\overline{i : t := \overline{k : t}}\} \mid \overline{\lambda x : t}.co\{\overline{i : t := \overline{k : t}}\} \\
 & \mid & \overline{\{f : t := t\}} \mid co\overline{\{f : t := t\}}
\end{array}
$$

Universes: checked, user declared, algebraic

Reduction: $\beta + \zeta + \delta + \iota +$ unfold $+$ co-unfold $+$ meta-substitution

Conversion:

nominal for declarations, (co)inductive types, (co)-recursive functions; (none for definitions)

Non first-order (co)recursive definitions

First-order recursive definitions:

$$(\lambda x.\overline{\{f : T := t\}}.f_i)\ M \triangleright \overline{\{f : T[M/x] := t[M/x]\}}.f_i$$

Non first-order recursive definitions:

$$(\lambda x.f_i\ t[x])\ M \triangleright f_i\ t[M]$$

I.e. together with nominal conversion, this makes a closure!

Non first-order (co)recursive definitions

First-order recursive definitions:

$$\begin{array}{lll} \overline{\{f : T := t\}}.f_i\ k & \rhd & t_i[\overline{\{f : T := t\}}/\overline{f}]\ k \\ & \rhd & M[\overline{\{f : T[N[k]] := t[N[k]]\}}] \end{array}$$

Non first-order recursive definitions:

$$f_i\ M\ k \rhd t_i\ M\ k \rhd L[t_j\ P[k]]$$

Non first-order (co)recursive definitions

Pros (so far):

- Reduction machine with recursive environments (major speed up, help the GC)
- Greatly simplified conversion checks
- No `simplify` tactic (URRAH!)
- No artificial duplication of top-level mutual recursive definitions (i.e. for all i, $f_i : T_i := \overline{\{f : T := t\}}.f_i$)

Cons (so far):

- Less conversion (seem useless)
- No nested definitions (but difficult to reason on)

Non first-order (co)recursive definitions

λ-lifting at work:

```
let rec f n :=
 match n with
    O => O
  | S m =>
     let rec g x :=
      match x with
         O => n
       | S k => g k + f m
     in
      g m
```

```
let rec g n f x :=
 match x with
    O => n
  | S k => g k + f n
```

```
let rec f n :=
 match n with
    O => O
  | S m => g n f m
```

But the r.h.s. is NOT accepted by Coq's guardedness conditions $\Rightarrow$ in Matita recursive definitions can be passed around to other recursive definitions

Non first-order (co)recursive definitions

can λ-lifting do this?

```
(f:(let rec g n :=
     match n with O => O | S m => g m in g y) -> T)
 (x: let rec g n :=
      match n with O => O+O | S m => g m in g y)
```

$$\Downarrow$$

```
  let rec g k n :=
   match n with O => k | S m => g n m
  ...
  (f : g O y -> T) (x: g (O+O) y)
```

Non first-order (co)recursive definitions

can λ-lifting do this (up to conversion)?

i.e.

$$
\begin{array}{lll}
M_1 & \lhd & (\lambda x.\mathtt{let\ rec}\ f\ n := \ldots\ \mathtt{in}\ f)\ N_1 \\
M_2 & \lhd & (\lambda x.\mathtt{let\ rec}\ f\ n := \ldots\ \mathtt{in}\ f)\ N_2
\end{array}
$$

Non first-order (co)recursive definitions

Achievements:

1. Incomplete algorithm to map Coq λ-terms into new ones
   - Claim: we are functionally complete (???)
   - Is type-preserving λ-lifting a decidable problem?
2. Extended positivity checks to allow passing (co)recursive functions around
   - Something I am ashamed of (at least in public. . .)
   - Accepts a (slightly) more understandable class of definitions
   - Still some work (makes the code more complex) to accept a reasonable class of (co)-recursive definitions over non (co)-recursive types

Checked, algebraic universes

$$\frac{}{\mathbf{Max}\{\mathbf{Type}_u\} : \mathbf{Max}\{\mathbf{Succ}(\mathbf{Type}_u)\}}$$

$$\frac{S : \mathbf{Max}\{\overline{u_1}\} \qquad T : \mathbf{Max}\{\overline{u_2}\}}{\Pi x : S.T : \mathbf{Max}\{\overline{u_1 @ u_2}\}\}}$$

$$\frac{f : s_2 \rightarrow T \qquad x : s_1 \qquad s_1 \leq s_2}{f\ x : T}$$

Checked, algebraic universes

$$\frac{\forall i,j.\ u_i \preceq v_j}{\textbf{Max}\{\overline{u}\} \leq \textbf{Max}\{\overline{v}\}}$$

$$\frac{u_i \leq v_j}{\textbf{Succ}(u_i) \preceq \textbf{Succ}(v_j)} \qquad \frac{u \leq v}{u \preceq \textbf{Succ}(v)} \qquad \frac{u < v}{\textbf{Succ}(u) \preceq v}$$

$$\frac{u < w \in E \qquad w \leq v}{u \leq v} \qquad \frac{}{u \leq u} \qquad \frac{u \leq w \in E \qquad w \leq v}{u \leq v}$$

$$\frac{u < w \in E \qquad w \leq v}{u < v} \qquad \frac{u \leq w \in E \qquad w < v}{u < v}$$

Aciclicity: $\nexists u, v.\ u < v \wedge v < u$

Checked, algebraic universes

Pros:

- The universe graph is very small, aciclicity check very quick and done once
- Customazibale PTS (also w.r.t impredicativity, computational content, etc.)
- Universe errors are localized and immediately given
- Major reduction in code size, complexity and efficiency
- Makes predicative mathematicians (Sambin) happy
- Easy to lift universes (at the library level only)
- The user must care about universes

Cons:

- The user must care about universes
- Cannot take successor of non user provided universe

Compacts local contexts (explicit substitutions) for metavariables

Metasenv: $\Gamma_i \vdash ?_i : T_i$

Subst: $\Gamma_i \vdash ?_i : T_i := t_i$

Occurrences: $\Delta \vdash ?_i[\bar{t}] : T_i[\bar{t}/\bar{x}]$ where $x_i : t_i \in \Gamma_i$

Example:

$\vdash ?_1 : A \rightarrow B \rightarrow C$

```
intros (x y);
```

$x : A, y : B \vdash ?_2 : C$

$\vdash ?_1 := \lambda x : A.\lambda y : B.?_2[x, y]$

Example:

$y : A \vdash ?_1 : B$

$(\lambda x : A.?_1[x])\ M \triangleright ?_1[M]$

Example:

$y : A \vdash ?_1 : A \times A := (a, a)$

$?_1[M] \triangleright (M, M)$

Compacts local contexts (explicit substitutions) for metavariables

Canonical contexts (in the metasenv/subst) are usually large

Most of the time local contexts are [`Rel k+1`,...,`Rel k+n`]

Major space/time optimization, improved sharing:
represent them as $[k, n]$

Major drawback:
must efficient code, greater complexity and code size

Outline

Conclusions (1/2)

Changes to the calculus:

- nominal, top-level only (co)recursive definitions
- (co)recursive definitions can be passed to other (co)recursive definitions
- algebraic universes (already in Coq)

Changes to the implementation:

- checked algebraic universes
- compact representation of explicit substitutions for metavariables

Conclusions (2/2)

Top-level (co)recursive definitions:

- major reduction in code size
- reduction/conversion speed-up
- recursive environments for reduction machines
- simplification under control

Checked algebraic universes:

- major reduction in code size and simplification of data structures
- major speed up
- customizable PTS
- understandable and localized universe errors

References

A.Asperti, W.Ricciotti, C.Sacerdoti Coen, E.Tassi.
A compact kernel for the calculus of inductive constructions.
In Special Issue on Iteractive Proving and Proof Checking of the Academy Journal of Engineering Sciences (Sadhana) of the Indian Academy of Sciences. SADHANA (BANGALORE). vol. 34(1), pp. 71 - 144 ISSN: 0256-2499, 2009