Conversion and reduction in dependently-typed calculi

A Survey

Víctor López Juan

Programming Logic Group

2016-03-09

Motivation

- Agda behaves slowly on not-so-hard problems.
- This is a big obstacle, for some users more so than the lack of tactics or the intransigence of the type-checker.
- It is hard to find documentation on state-of-the art implementations (e.g. Coq, Agda).

Outline

Matita

Claudio Sacerdoti Coen. "Reduction and conversion strategies for the calculus of (co) inductive constructions: Part I". . In: *Electronic Notes in Theoretical Computer Science* 174.10 (2007), pp. 97–118

- CoC based
- Compatible with Coq proof terms.
- Focus on user interaction and type inference.

Calculus of (Co)Inductive Constructions

A subset of Matita's implementation:

t $::=$ n	de Bruijn index, $n \in [1, +\infty)$
$\mid$ c	constant
$\mid$ i	(co)inductive type
$\mid$ k	(co)inductive constructor
$\mid$ $Set \mid Prop \mid Type_i$	sorts
$\mid$ $t\ t$	application
$\mid$ $\lambda : t.t$	abstraction
$\mid$ $\lambda := t.t$	local definition
$\mid$ $\prod : t.t$	Π-type
$\mid$ $\langle t \rangle_h t\{\vec{t}\}$	case analysis
$\mid$ $\mu_l\{\overrightarrow{t : t/n_\alpha}\}$	mutual recursion
$\mid$ $\nu_l\{\overrightarrow{t : t}\}$	mutual co-recursion

Matita performance

Conversion heuristics: None, α-equivalence, and both α-equivalence and lazy δ-expansion.
Reduction strategies: Call-by-name, by-value, hybrid (not shown) and by-need.

Conversion	Red.	Total	Longest	> 30 s	> 1 s
Simple	by-name	1285.71s	29.6s	375	170
w/ α-equiv	by-name	246.76	6.9	1	15
w/ α-eq & lazy δ	by-name	199.26s	2.2s	1	2
w/ α-eq & lazy δ	by-need	201.71s	1.5s	1	3
w/ α-eq & lazy δ	by-value	220.54s	11.8s	0	19
Coq		40.87s	2.5s	0	2

Call-by-need evaluation

Based on generalized Krivine machine:

State ≡ (Environment, Term, Stack)
Environment ≡ [MVar (Bool, Configuration)]
Stack ≡ [(Environment, Term)]

- Application puts argument into Stack.
- λ-abstraction moves argument from Stack to Environment.

Other evaluation strategies use different environment and stack types.

Smart δ-expansion

When checking for conversion of two terms...

1. Reduce w/o δ-expansion
2. Reduction stops → Terms are either WHNF, or have δ-redex on head [1].
3. Compute height[2] of heads (0 if WHNF, $+\infty$ if not δ-redex).
4. Reduce term with tallest head until height matches, compare for α-equiv.

[1] Head is the head of i) the function in an application, or ii) the inductive argument in case analysis/well-founded recursion.

[2] Distance from root on implicit dependency tree.

Coq

Could not find a technical report about the current implementation.

- Kernel syntax with general let, application, and abstraction.
- Bytecode/native tactic used for intensive computation.
- Smart δ-expansion based on priorities (∞ for irrelevant terms).

λProlog

Xiaochu Qi. "An implementation of the language lambda prolog organized around higher-order pattern unification". In: *arXiv preprint arXiv:0911.5203* (2009)

copy a a.
copy (app t_1 t_2*) (app* t_3 t_4*) :- copy (*t_1 t_3*), copy (*t_2 t_4*)*
copy (abs t_1*) (abs* t_2*) :- ∀c copy (*t_1 *c) (*t_2 *c)*

- Emphasis in backtracking, existential instantiation, disjunction.
- Efficient implementation based on a Prolog abstract machine, with separate pattern-fragment solver for higher-order unification.
- Explicit substitutions to delay traversals.

The suspension calculus (I)

Andrew Gacek and Gopalan Nadathur. "A simplified suspension calculus and its relationship to other explicit substitution calculi". In: *arXiv preprint cs/0702152* (2007)

(β_s) $((\lambda t_1) t_2) \to [\![t_1, 0, (t_2, 0) :: nil]\!]$.

(r1) $[\![c, nl, e]\!] \to c$, for c a constant.

(r2) $[\![\#i, nl, nil]\!] \to \#j$, where $j = i + nl$.

(r3) $[\![\#1, nl, (t, l) :: e]\!] \to [\![t, nl - l, nil]\!]$

(r4) $[\![\#i, nl, (t, l) :: e]\!] \to [\![\#i', nl, e]\!]$, where $i' = i - 1$, for $i > 1$.

(r5) $[\![(t_1 t_2), nl, e]\!] \to ([\![t_1, nl, e]\!] [\![t_2, ol, nl, e]\!])$.

(r6) $[\![(\lambda t), nl, e]\!] \to (\lambda [\![t, 1 + nl, (\#1, 1 + nl) :: e]\!])$

The suspension calculus (II)

(m1) $[\![[\![t, nl_1, e_1]\!], nl_2, e_2]\!] \to [\![t, nl', \{e_1, nl_1, e_2\}]\!]$, where $nl' = nl_2 + (nl_1 \dot{-} len(e_2))$.

(m2) $\{e_1, nl_1, nil\} \to e_1$.

(m3) $\{nil, 0, e_2\} \to e_2$.

(m4) $\{nil, 1 + nl_1, (t, l) :: e_2\} \to \{nil, nl_1, e_2\}$

(m5) $\{(t, n) :: e_1, 1 + nl_1 (s, l) :: e_2\} \to \{(t, n) :: e_1, nl_1, e_2\}$, for $nl_1 > n$.

(m6) $\{(t, n) :: e_1, n, (s, l) :: e_2\} \to ([\![t, l, (s, l) :: e_2]\!], m) :: \{e_1, n, (s, l) :: e_2\}$, where $m = l + (n \dot{-} (len(e_2) + 1))$.

Tog

- Developed by Francesco Mazzoli
- Parametrized by several reduction strategies.
- Unification as in Agda, modulo issue 1258.
- Uses constraints for type-checking.

Time to show some stats

Plan

1. Benchmark different approaches on Tog prototype.
2. Implement promising ones on Agda.
3. Profit

Stable names

Simon Peyton Jones, Simon Marlow, and Conal Elliott. "Stretching the storage manager: weak pointers and stable names in Haskell". In: *In Koopman and Clack [23*. Springer Verlag, 1999, pp. 37–58

GC-aware pointer (equality)!
Pros:

- 100% safe, 100% leak free.
- Low overhead.
- Same framework implements value-weak hash tables.

Cons:

- Hard to exploit in current implementation.
- Evaluating a term changes its stable name.

Dimensions

Go

Fingerprinting ∅ / Stable names / Hash consing / Crypto-hash

Unification Lazy/Eager constraint generation.

δ-expansion Eager / Lazy

Memoization ∅ / Subst. / Conversion / Reduction / All

Explicit substitution No / Yes

Intentionally left out

Hashing, λσ-calculus, MVar-based sharing, Byte-code interpreter

Any thoughts?