Equality for λ-terms with list primitives

Pierre Boutillier
under the supervision of Conor McBride

May - July 2009

Plan

1. A goal
2. An implementation
3. A formalisation

Ideas

- Intentional equality decision
- Internalize specific laws for common operators
- Normalizing in two phases vs extending rewrite rules

Semantic rules

Tips

- $\dfrac{\textit{several} \quad \textit{hypothesis}}{\textit{conclusion}}$ or $\dfrac{}{\textit{axiom}}$
- Trust names to get what object are
- Functionnal rules presentation follows the calculus

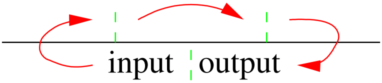

Figure: Rule schema

Variable value storage makes life easier

$$\textsc{Env} \quad := \quad \varepsilon \mid (\textsc{Env}, v := \textsc{ChkTm})$$

Figure: Definition of environment

Closure and Environment

- Closure is a pair of term and environment
- Value of variable in a term is its definition if it exists
- In a way, substitution is delayed

Variable value storage makes life easier

$$\textsc{Env} \quad := \quad \varepsilon \mid (\textsc{Env}, v := \textsc{ChkTm})$$

Figure: Definition of environment

Closure and Environment

- Closure is a pair of term and environment
- Value of variable in a term is its definition if it exists
- In a way, substitution is delayed

$$(\varepsilon, x := \lambda y.y ! \lambda z.x) = (\varepsilon, x := \lambda y.y ! \lambda z.\lambda y.y)$$
$$(\varepsilon, x := \lambda y.y ! \lambda z.x) \neq (\varepsilon ! \lambda z.x)$$
$$(\varepsilon, x := \lambda y.y ! \lambda z.x) = (\varepsilon ! \lambda z.\lambda y.y) = (\varepsilon, x := \lambda y.t ! \lambda z.\lambda y.y)$$

A Gödel system T like λ-calculus : types and terms

$$\textsc{Ty} \quad := \quad \mathbf{1} \mid \textsc{Ty} \rightarrow \textsc{Ty} \mid$$

A Gödel system T like λ-calculus : types and terms

$$
\begin{aligned}
\textsc{Ty} \quad := \quad & \mathbf{1} \mid \textsc{Ty} \to \textsc{Ty} \mid \\
& \textsc{Ty} \times \textsc{Ty} \mid \\
& \textsc{Ty}\ \mathit{List}
\end{aligned}
$$

Figure: Definition of types

A Gödel system T like λ-calculus : types and terms

$$
\begin{aligned}
\textsc{Ty} \quad &:= \quad \mathbf{1} \mid \textsc{Ty} \rightarrow \textsc{Ty} \mid \\
&\quad\;\; \textsc{Ty} \times \textsc{Ty} \mid \\
&\quad\;\; \textsc{Ty}\ \mathit{List}
\end{aligned}
$$

Figure: Definition of types

$$
\begin{aligned}
\textsc{Elim} \quad &:= \quad \mathbf{rec}_{\mathit{Ty}}\ \textsc{ChkTm}\ \textsc{ChkTm} \mid \\
&\quad\;\; \mathbf{first} \mid \mathbf{map}_{\mathit{Ty}}\ \textsc{ChkTm} \mid \\
&\quad\;\; \mathbf{second} \mid \mathbf{app}\ \textsc{ChkTm}
\end{aligned}
$$

Figure: Definition of Primitive elimination operators

A Gödel system T like λ-calculus : types and terms

$$\begin{aligned}\textsc{Ty} \quad &:= \quad \mathbf{1} \mid \textsc{Ty} \to \textsc{Ty} \mid \\ &\quad\;\; \textsc{Ty} \times \textsc{Ty} \mid \\ &\quad\;\; \textsc{Ty}\ \mathit{List}\end{aligned}$$

Figure: Definition of types

$$\begin{aligned}\textsc{Elim} \quad &:= \quad \mathbf{rec}_{Ty}\ \textsc{ChkTm}\ \textsc{ChkTm} \mid \\ &\quad\;\; \mathbf{first} \mid \mathbf{map}_{Ty}\ \textsc{ChkTm} \mid \\ &\quad\;\; \mathbf{second} \mid \mathbf{app}\ \textsc{ChkTm}\end{aligned}$$

Figure: Definition of Primitive elimination operators

$$\begin{aligned}\textsc{InfTm} \quad &:= \quad v \mid \textsc{Elim}\ \textsc{InfTm} \mid \textsc{ChkTm} : \mathit{Ty} \\ \textsc{ChkTm} \quad &:= \quad \lambda v.\textsc{ChkTm} \mid () \mid (\textsc{ChkTm}, \textsc{ChkTm}) \mid \underline{\textsc{InfTm}}\end{aligned}$$

Figure: Definition of **inferable term** and **checkable term**

What does it stand for ?

calculus behaviour and meaning

1. Everything are functions
2. The arrow type stand for this
3. Calculus goes when a term is destruct

What does it stand for ?

calculus behaviour and meaning

1. Everything are functions
2. The arrow type stand for this
3. Calculus goes when a term is destruct

Practically

- $(\textbf{app } s)\ \lambda x.t$ β-reduce to $t[s/x]$
- $t[s/x]$ stand for t where every occurrence of x is replaced by s

What does it stand for ?

calculus behaviour and meaning

1. Everything are functions
2. The arrow type stand for this
3. Calculus goes when a term is destruct

Practically

- $(\mathbf{app}\ s)\ \lambda x.t$ β-reduce to $t[s/x]$
- $t[s/x]$ stand for t where every occurrence of x is replaced by s

Bidirectionnal type checking

- Equivalent to Chuch presentation of term.
- Ready for dependant type system
- Type unicity of term allow type follow to normalize without "most general type" problem.

What are equals terms ?

α-conversion

- Variables of λ are mute so $\lambda x.x \equiv \lambda y.y$
- If you change a part of a term, ambiguitys can occur.

β equivalence

- Represents a step of calculus. Is the consequence of an elimination.
- Expressed by substitution of variables for terms alone but only a environment change for closures.

η-equality

Suppose that f is a function, There is no diference between f and $\lambda x.(\textbf{app}\ x)\ f$ behavour but syntaxes can be different.

Open term complement for list

- $\textbf{map}_\sigma\ f\ (\textbf{map}_\tau\ g\ l) \equiv \textbf{map}_\sigma\ (f \circ g)\ l$
- $\textbf{map}_\tau\ id\ l \equiv l$
- $\textbf{map}_\tau\ f\ (\textbf{append}\ ys\ xs) \equiv \textbf{append}\ (\textbf{map}_\tau\ f\ ys)\ (\textbf{map}_\tau\ f\ xs)$

What are equals terms ?

α-conversion

- Variables of λ are mute so $\lambda x.x \equiv \lambda y.y$
- If you change a part of a term, ambiguitys can occur.

β equivalence

- Represents a step of calculus. Is the consequence of an elimination.
- Expressed by substitution of variables for terms alone but only a environment change for closures.

η-equality

Suppose that f is a function, There is no diference between f and $\lambda x.(\mathbf{app}\ x)\ f$ behavour but syntaxes can be different.

Open term complement for list

- $\mathbf{map}_\sigma\ f\ (\mathbf{map}_\tau\ g\ l) \equiv \mathbf{map}_\sigma\ (f \circ g)\ l$
- $\mathbf{map}_\tau\ id\ l \equiv l$
- $\mathbf{map}_\tau\ f\ (\mathbf{append}\ ys\ xs) \equiv \mathbf{append}\ (\mathbf{map}_\tau\ f\ ys)\ (\mathbf{map}_\tau\ f\ xs)$

Normalisation by evaluation

Normalisation by evaluation

Syntactic transformation

$$\frac{\gamma \vdash s \Downarrow s' \qquad \gamma \vdash t \Downarrow t'}{\gamma \vdash (s,t) \Downarrow (s',t')} \qquad \frac{\gamma \vdash f \Downarrow f' \qquad \gamma \vdash s \Downarrow s' \qquad f' \ @\ s' \ \rightarrow \ v}{\gamma \vdash f\ \ s \Downarrow v}$$

$$\frac{}{\gamma, x := v, \gamma' \vdash x \Downarrow v} \qquad \frac{}{\gamma \vdash \lambda x.t \Downarrow \lambda\ [\gamma]\ x.t}$$

- No simplification is made under λ

Normalisation by evaluation

Syntactic transformation

$$\frac{\gamma \vdash s \Downarrow s' \qquad \gamma \vdash t \Downarrow t'}{\gamma \vdash (s, t) \Downarrow (s', t')} \qquad \frac{\gamma \vdash f \Downarrow f' \qquad \gamma \vdash s \Downarrow s' \qquad f' \mathbin{@} s' \;\rightarrow\; v}{\gamma \vdash f\ s \Downarrow v}$$

$$\frac{}{\gamma, x := v, \gamma' \vdash x \Downarrow v} \qquad \frac{}{\gamma \vdash \lambda x.t \Downarrow \lambda\ [\gamma]\ x.t}$$

- No simplification is made under λ

Computation

$$\frac{}{e \mathbin{@} \underline{t} \;\rightarrow\; \underline{e\ t}} \qquad \frac{f \mathbin{@} h \;\rightarrow\; v \qquad \mathbf{map}_\tau\ f \mathbin{@} l \;\rightarrow\; w}{\mathbf{map}_\tau\ f \mathbin{@} h :: l \;\rightarrow\; v :: w}$$

$$\frac{\gamma, x := s \vdash t \Downarrow v}{\mathbf{app}\ s \mathbin{@} \lambda\ [\gamma]\ x.t \;\rightarrow\; v} \qquad \frac{}{\mathbf{map}_\tau\ f \mathbin{@} [] \;\rightarrow\; []}$$

Type based simplification

Principe

- Follow unique Church form term types
- All non elementary typed terms are expanded
- Simplification rules go bottom up
- New declared variable are build to evaluate under λ
- Terms are syntaxtally rewrite on a writable way

$$\frac{\Gamma \vdash tyCtxt}{\Gamma \vdash \mathbf{1} \ni x \Rightarrow ()} \qquad \frac{\Gamma \vdash t \Uparrow t' \in \tau}{\Gamma \vdash \tau List \ni \underline{t} \Uparrow \underline{t'}} \qquad \frac{\overline{\Gamma, x : \sigma} \vdash \mathbf{app}\ x\ f \Downarrow v}{\Gamma \vdash \sigma \to \tau \ni f \Rightarrow v}$$

Type based simplification

Principe

- Follow unique Church form term types
- All non elementary typed terms are expanded
- Simplification rules go bottom up
- New declared variable are build to evaluate under λ
- Terms are syntaxtally rewrite on a writable way

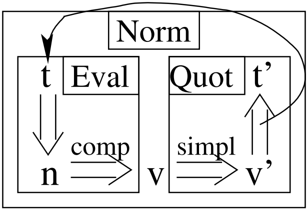

Figure: General picture

$$\frac{\Gamma \vdash tyCtxt}{\Gamma \vdash \mathbf{1} \ni x \Rightarrow ()} \qquad \frac{\Gamma \vdash t \Uparrow t' \in \tau}{\Gamma \vdash \tau List \ni \underline{t} \Uparrow \underline{t'}} \qquad \frac{\overline{\Gamma, x : \sigma} \vdash \mathbf{app}\ x\ f \Downarrow v}{\Gamma \vdash \sigma \to \tau \ni f \Rightarrow v}$$

Variable representation

Naïve $\lambda x.\lambda y.\mathbf{map}_\tau\ f\ x : \sigma$ Hard to compare and to compute but easily readable by human.

Fonctional language **fun** $x \rightarrow$ **fun** $y \rightarrow$ **List.map** $f\ x$ difficulties to compute are hide. Impossible to compare or to show.

deBruijn index $\lambda.\lambda.\mathbf{map}_\tau\ ?\ 1 : \sigma$ unreadable by human but canonical form to compare. Tricky but unambigous to compute.

Locally nameless $\lambda.\lambda.\mathbf{map}_\tau\ f\ 1 : \sigma$ Avoid mess of free variable index but still canonical unambigous form.

Toolbox

```
val var_care_inf f_def f_lam f_var env term
val var_care_check f_def f_lam f_var env term
```

Structure

```
val evalInf : Ttype.value list ->
  (Ttype.name * Ttype.value) list ->
  Ttype.infTerm -> Ttype.value
val evalCheck : Ttype.value list ->
  (Ttype.name * Ttype.value) list ->
  Ttype.checkTerm -> Ttype.value

val equiv_fun : int -> Ttype.tType ->
  Ttype.value -> Ttype.value -> bool
val simplify : int -> Ttype.neutral ->
  Ttype.neutral
val quoteValue : int -> Ttype.tType ->
  Ttype.value -> Ttype.checkTerm
val quoteNeu : int -> Ttype.neutral ->
  Ttype.infTerm * Ttype.tType
```

Results

Experimental discovery

- Going bottom up catches the most simplification
  $$map\ (map\ id)\ l = l$$
- Develop vs factorize when ordering simplification
  $$map\ \ +1\ (append\ (map\ \ +1\ x)\ y) = append\ (map\ \ +2\ x)\ (map\ \ +1\ y)$$
- η-expantion is exactly non elementary types destruction
  makes neutrals bigger and leave value unchanged
- Over non elementary types, Identity function has different form. Equality with it must be done with it's quotation over the given type.
  $$\lambda x.x : (ONE, ONE) = \lambda x.(first\ x, second\ x) : (ONE, ONE)$$

Demonstration

\x. x : (a->a)->(a->a)	\ 1.\ 2.(1)2
(a List) List->(a List) List \ 1.map {a List} (\x. map {a} (\y. y) x) (1)	\ 1.1
(c -> b) -> (b -> a) -> c List -> a List \g f xs. (map {a} f (((\x. append (map {b} g x) ()) : c List -> b List) xs))	\ 1 2 3.map {a} (\ 4.(2)(1)4) (3)
let swap : (a * b) -> b * a = \ x. (second x, first x) let swap2 : (b * a) -> a * b = \ x. (second x, first x) (a * b) List -> (a * b) List \ x. map {(a * b)} swap2 (map {(b * a)} swap x)	\ 1.1

Big step evaluation rules

Exact functionnal language behaviour with no scheduling question

- Strong normalization makes strategies equivalent
- Determinism is obvious, there is only one rule for each constructor
- Functionnality implys terminaison to ensure consistance

Dependant types makes life rigourous

Only deBruijn indexes are used here because definition are ommited

Environments are characterized by the number of declared variable and of all kind variable it store.

Terms/Values are charaterized by their kind, their direction of typing and how many variable they are dealing with.

This homogenous presentation gives a lot of factorization in proof.

Less index care as possible

1. Thanks to closure, only adding a delared variable at the end of the environment is require while you compute under λ.
2. Other defined term of the environment must still speak about same things.
3. *Weakening* ensure this by converting every number to it's succesor.
4. The less elementary operation aren't necessary.

Sanity for everyone

Environment and context

From types for declared variables, you can get types for defined one by typing it.

You want that in a well typed closure, allowed operations keep everything well formed.

Normalisation preserves types

Evaluation, computation, simplification must preserve types. But typing rules are made to follow the structure and the calculus ...

Elimination form case impose a stronger induction hypothesis than the obvious one that ensure a well type as output of all possible type as input.

Soundness

Three kinds of equality rules

Structural rules to say that in different context terms are equals if their sub-terms are.

Computational rules that expose one step of calculus in a given environment.

Simplification rule which describe the valid transformation of a term in an environment.

Proof principle

decorating derivation with equationnal rule that makes a normalistion tree a rules list linked by transitivity.

Proof requirement

(postcomposition compatibility and eval-quote unicity)

To conclude

Bilan

- A new desert to explore
- New ideas to model the λ-calculus behaviour
- Bricks to make formal proof
- Even more use for types

What's next ?

1. Completeness
2. Generec completeness condition over sets of simplification rules
3. What it become with dependant type
4. There is more data structure than list

Questions ?