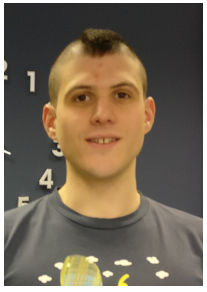
Terms with Bindings as an Abstract Data Type

Jasmin Blanchette, Lorenzo Gheri, Andrei Popescu, Dmitriy Traytel



Vrije Universiteit Amsterdam Middlesex University London ETH Zürich

Terms of the λ-calculus

Var infinite set of variables, ranged over by x, y, z etc.

The set Trm of λ-terms, ranged over by t, s etc., defined by grammar:

$$t \ ::= \ \mathsf{Vr}\ x \ \mid \ \mathsf{Ap}\ t_1\ t_2 \ \mid \ \mathsf{Lm}\ x\ t$$

... with the proviso that terms are equated (identified) modulo α-equivalence (a.k.a. naming equivalence)

(Will often omit the injection Vr of variables into terms.)

E.g., Lm x x is considered equal to Lm y y

Terms as an abstract data type (ADT)

Trm endowed with algebraic structure, given by operators such as:

- the constructors Vr, Ap, Lm
- (capture-avoiding) substitution
  $\_[\_ / \_] : \mathsf{Trm} \to \mathsf{Trm} \to \mathsf{Var} \to \mathsf{Trm}$
  e.g., $(\mathsf{Lm}\ x\ (\mathsf{Ap}\ x\ y))\ [\mathsf{Ap}\ x\ x\ /\ y] = \mathsf{Lm}\ x'\ (\mathsf{Ap}\ x'\ (\mathsf{Ap}\ x\ x))$
- swapping $\_[\_ \wedge \_] : \mathsf{Trm} \to \mathsf{Var} \to \mathsf{Var} \to \mathsf{Trm}$
  e.g., $(\mathsf{Lm}\ x\ (\mathsf{Ap}\ x\ y))\ [x \wedge y] = \mathsf{Lm}\ y\ (\mathsf{Ap}\ y\ x)$
- (finite) permutation
  $\mathsf{Perm} = \{\sigma : \mathsf{Vr} \to \mathsf{Vr} \mid \{x \mid \sigma\ x \neq x\}\ \text{finite}\ \}$
  $\_[\_] : \mathsf{Trm} \to \mathsf{Perm} \to \mathsf{Trm}$
  e.g., $(\mathsf{Lm}\ x\ (\mathsf{Ap}\ z\ y))\ [x \mapsto y, y \mapsto z, z \mapsto x] = \mathsf{Lm}\ y\ (\mathsf{Ap}\ x\ z)$
- freshness $\_\mathsf{fresh}\_ : \mathsf{Var} \to \mathsf{Trm} \to \mathsf{Bool}$
  e.g., $x\ \mathsf{fresh}\ \mathsf{Lm}\ x\ x$

Terms as an abstract data type (ADT)

Properties of the term algebra

- Various basic properties of the operators, e.g.,
  1. x fresh t implies x fresh $s[t/x]$
  2. $\{x \in \mathrm{Var} \mid \neg\, x \text{ fresh } t\}$ is finite
- Reasoning principle – induction
- Definition principle – recursion

A subset of the above will characterize the Trm algebra uniquely up to isomorphism.

ADT characterization vs concrete representation

The particular representation – quotient, de Bruijn, weak/strong HOAS, locally named/nameless – does not matter in the end: it's the same Platonic concept!

What matters is the end product:

- How expressive/useful are the (inductive) reasoning and (recursive) definition principles?
- How expressive and modular is the construction of binding structures?

Focus: recursion principles

We want such principles to be:

- Expressive: cover functions of interest, cover complex binding structures
- Easy to use: do not require complex verifications in order for the definitions to go through

First, some not very useful principles:

1. Free datatype of raw terms
2. de Bruijn
3. Gordon-Melham / weak HOAS

1. Work with the free datatype of raw terms (no α-equivalence)

$$t \ ::= \ \mathsf{Vr}\ x \ \mid \ \mathsf{Ap}\ t_1\ t_2 \ \mid \ \mathsf{Lm}\ x\ t$$

Advantage: Can immediately define in proof assistants as standard datatypes:

$$\mathsf{datatype\ Trm} \ = \ \mathsf{Vr\ Var} \ \mid \ \mathsf{Ap\ Trm\ Trm} \ \mid \ \mathsf{Lm\ Var\ Trm}$$

This yields the standard free recursor.

Major disadvantages:

- Substitution is not well-behaved
- Most of the times we would need to prove that the function is invariant under α-equivalence—which is usually very complex

2. Work with a de Bruijn encoding

$$t \ ::= \ n \ | \ \text{Ap}\ t_1\ t_2 \ | \ \text{DBLm}\ t$$

λ-abstraction takes no variable input, bound variables replaced by numbers indicating which λ binds them.

Advantage: again, a free datatype

Major disadvantages:

- Dangling references DBLm 3 – number 3 refers to non-existing DBLm in the term
- Recursor talks about a fixed variable to be bound (via DBLm)
- In the end still must define a proper Lm, or keep encoding everything painfully using DBLm

But see some intelligent workarounds: Saving de Bruijn (Norrish/Vestergaard 2007), Locally nameless (Charguéraud 2012), Functor categories (Fiore et al. 1999)

3. Regard abstraction as taking a function as input

Despeyroux et. al 1995 (weak HOAS), Gordon/Melham 1996
Regard terms as a subset of the datatype:

datatype Termoid $=$ Vr Var $\mid$ Ap Termoid Termoid $\mid$ LLm (Var $\rightarrow$ Termoid)

Then Lm x t is defined as LLm ($\lambda y.\ t[y/x]$).
Proper subset: LLm($\lambda x.$ if $x = y$ then ... else ...) incorrect, "exotic" term.

Advantage: again, free-datatype recursor

Major disadvantages:

- Use LLm applied to restricted function space instead of Lm
- Cannot easily define useful functions

Some not very useful recursion principles

Summary of the disadvantages:

- The recursor inherited from raw-term encodings suffers from lack of abstraction (notably substituion not well behaved)
- The recursor inherited from de Bruijn or functional (weak HOAS) encodings replaces the standard λ-abstraction with a different primitive

Some more useful recursion principles

The Nominal Logic recursion principle

Michael Norrish's improvement

Our own contribution

Preliminaries: basic properties of terms I

Freshness versus constructors
(Fr1) $z \neq x \;\Rightarrow\; z$ fresh Vr x
(Fr2) z fresh $s \;\Rightarrow\; z$ fresh $t \;\Rightarrow\; z$ fresh Ap $s\ t$
(Fr3) $z = x \;\vee\; z$ fresh $t \;\Rightarrow\; z$ fresh Lm $x\ t$

Swapping versus constructors
(SwVr) (Vr x) $[z_1 \wedge z_2] =$ Vr $(x[z_1 \wedge z_2])$
(SwAp) (Ap $s\ t$) $[z_1 \wedge z_2] =$ Ap $(s\,[z_1 \wedge z_2])\,(t\,[z_1 \wedge z_2])$
(SwLm) (Lm $x\ t$) $[z_1 \wedge z_2] =$ Lm $(x\,[z_1 \wedge z_2])\,(t\,[z_1 \wedge z_2])$

Algebraic properties of swapping
(SwId) $t\,[z \wedge z] = t$
(SwInv) $t\,[x \wedge y]\,[x \wedge y] = t$
(SwComp) $t\,[x \wedge y]\,[z_1 \wedge z_2] = (t\,[z_1 \wedge z_2])\,[(x\,[z_1 \wedge z_2]) \wedge (y\,[z_1 \wedge z_2])]$

Algebraic properties of swapping versus freshness
(SwFr) x fresh $t \Rightarrow y$ fresh $t \Rightarrow t\,[x \wedge y] = t$
(FrSw) z fresh $t\,[x \wedge y] \Leftrightarrow z[x \wedge y]$ fresh t

Preliminaries: basic properties of terms II

Permutation versus constructors
(PmVr) $(\mathsf{Vr}\ x)\,[\sigma] = \mathsf{Vr}\ (\sigma\ x)$
(PmAp) $(\mathsf{Ap}\ s\ t)\,[\sigma] = \mathsf{Ap}\,(s\,[\sigma])\,(t\,[\sigma])$
(PmLm) $(\mathsf{Lm}\ x\ t)\,[\sigma] = \mathsf{Lm}\ (\sigma\ x)\ (t\,[\sigma])$

Algebraic properties of permutation
(PmId) $t\,[\mathsf{id}] = t$
(PmComp) $t\,[\sigma]\,[\tau] = t\,[\tau \circ \sigma]$

Algebraic properties of permutation versus freshness
(PmFr) x fresh $\sigma \Rightarrow t\,[\sigma] = t$
(FrPm) z fresh $t\,[\sigma] \Leftrightarrow z[\sigma^{-1}]$ fresh t

Preliminaries: basic properties of terms III

Substitution versus constructors

(Sb1) $(\mathsf{Vr}\ x)\,[s/z] = (\mathsf{if}\ x = z\ \mathsf{then}\ s\ \mathsf{else}\ \mathsf{Vr}\ x)$

(Sb2) $(\mathsf{Ap}\ t_1\ t_2)\,[s/z] = \mathsf{Ap}\,(t_1\,[s/z])\,(t_2\,[s/z])$

(Sb3) $x \neq z \Rightarrow x\ \mathsf{fresh}\ s \Rightarrow (\mathsf{Lm}\ x\ t)\,[s/z] = \mathsf{Lm}\ x\ (t\,[s/z])$

Abstraction rules

(SwCong) $z \notin \{x_1, x_2\} \Rightarrow z\ \mathsf{fresh}\ t_1, t_2 \Rightarrow t_1[z{\wedge}x_1] = t_2[z{\wedge}x_1]$
$\Rightarrow \mathsf{Lm}\ x_1\ t_1 = \mathsf{Lm}\ x_2\ t_2$

(SwRen) $z \neq x \Rightarrow z\ \mathsf{fresh}\ t \Rightarrow \mathsf{Lm}\ x\ t = \mathsf{Lm}\ z\ (t[z{\wedge}x])$

(SbCong) $z \notin \{x_1, x_2\} \Rightarrow z\ \mathsf{fresh}\ t_1, t_2 \Rightarrow t_1[(\mathsf{Vr}\ z)/x_1] = t_2[(\mathsf{Vr}\ z)/x_1]$
$\Rightarrow \mathsf{Lm}\ x_1\ t_1 = \mathsf{Lm}\ x_2\ t_2$

(SbRen) $z \neq x \Rightarrow z\ \mathsf{fresh}\ t \Rightarrow \mathsf{Lm}\ x\ t = \mathsf{Lm}\ z\ (t[(\mathsf{Vr}\ z)/x])$

Preliminaries: basic properties of terms IV

Finite support
(FinSupp) $\exists X.\ \ X$ finite and $\forall x, y \notin X.\ t[x \mapsto y, y \mapsto x] = t$

Definability of freshness from permutations
(FrFromPm) x fresh $t \iff \{y \mid t[x \mapsto y, y \mapsto x] \neq t\}$ finite

Definability of freshness from swapping
(FrFromSw) x fresh $t \iff \{y \mid t[x \wedge y] \neq t\}$ finite

Freshness condition for binders (barebone version)
(FCB) $\exists x.\ \forall t.\ x$ fresh Lm $x\ t$

Preliminaries: algebras (models)

All the above properties make sense
not only for the set Trm of terms
but also for any set A endowed with operators having the given arities, e.g.,

- the constructors
  $\mathsf{Vr} : \mathsf{Var} \to A$
  $\mathsf{Ap} : A \to A \to A$
  $\mathsf{Lm} : \mathsf{Var} \to A \to A$
- substitution $\_[\_ / \_] : A \to A \to \mathsf{Var} \to A$
- swapping $\_[\_ \wedge \_] : A \to \mathsf{Var} \to \mathsf{Var} \to A$
- $\_[\_] : A \to \mathsf{Perm} \to A$
- $\_\mathsf{fresh}\_ : \mathsf{Var} \to A \to \mathsf{Bool}$

Recursion principles: barebone versions

THEOREM. Trm is **initial object** in the following categories of algebras:

1) Pitts	2) Norrish	3) Our results
PmVr PmAp PmLm	SwVr SwAp SwLm	SwVr SwAp SwLm
PmId PmComp	SwId SwInv	SwCong
	SwFr FrSw	
FrFromPm FCB	FrVr FrAp FrLm	FrVr FrAp FrLm
~~FinSupp~~		

1') Pitts swap-based	2') Norrish perm-based	3') Us with renaming
SwVr SwAp SwLm	PmVr PmAp PmLm	SwVr SwAp SwLm
SwId SwInv SwComp	PmId PmComp	SwRen
	PmFr PmSw	
FrFromSw FCB	FrVr FrAp FrLm	FrVr FrAp FrLm

Expressiveness (generality): $1 = 1' \leq 2 = 2' \leq 3' \leq 3$

(Norrish 2004, Pitts 2006 based on previous work with Gabbay, Urban/Berghofer 2006, Gheri/Popescu 2017, BGPT 2018)

Substitution-based variations

THEOREM. Trm is **initial object** in the following categories of algebras:

1) Our results	2) Our results
SbVr SbAp SbLm	SbVr SbAp SbLm
SbRen	SbCong
FrVr FrAp FrLm	FrVr FrAp FrLm

Expressiveness (generality): $1 \leq 2$

(Popescu/Gunter 2011, Gheri/Popescu 2017, BGPT 2018)

Parenthesis: recursion from initial algebra

Initiality: Given any algebra $\mathcal{A}$, there exists a unique morphism from Trm algebra to $\mathcal{A}$.

Given a set A, in order to define a function $H : \mathsf{Trm} \to A$, **organize it as an algebra**, i.e.,

1. define an algebraic structure: $\mathsf{Vr}^A : \mathsf{Var} \to A$, $\mathsf{Ap}^A : A \to A \to A$, $\mathsf{Lm}^A : \mathsf{Var} \to A \to A$, $\_ \wedge^A \_ : A \to \mathsf{Var} \to \mathsf{Var} \to A$, etc.
2. Verify that this satisfies the necessary properties (e.g., SwVr, SwAp)

In exchange, you get back an algebra morphism, i.e., a function $H : \mathsf{Trm} \to A$ that commutes with the operators, e.g.,

$$H\,(\mathsf{Vr}\ x) = \mathsf{Vr}^A\ x$$

$$H\,(\mathsf{Ap}\ t_1\ t_2) = \mathsf{Ap}^A\ (H\ t_1)\ (H\ t_2)$$

$$H\,(\mathsf{Lm}\ x\ t) = \mathsf{Lm}^A\ x\ (H\ t)$$

$$H\,(t\ [x \wedge y]) = (H\ t)\ [x \wedge^A y]$$

The commutation clauses **are** the recursive definition.

Intuition

We want a recursion principle that allows us to recurse over the standard constructors:

$$H\ (\mathsf{Vr}\ x)\ =\ \ldots x \ldots$$
$$H\ (\mathsf{Ap}\ t_1\ t_2)\ =\ \ldots H\ t_1 \ldots H\ t_2 \ldots$$
$$H\ (\mathsf{Lm}\ x\ t)\ =\ \ldots x \ldots H\ t \ldots$$

Intuition

We want a recursion principle that allows us to recurse over the standard constructors:

$$H\ (\mathsf{Vr}\ x)\ =\ \ldots x \ldots$$
$$H\ (\mathsf{Ap}\ t_1\ t_2)\ =\ \ldots H\ t_1 \ldots H\ t_2 \ldots$$
$$H\ (\mathsf{Lm}\ x\ t)\ =\ \ldots x \ldots H\ t \ldots$$

To "help" recursion, we must describe the behavior of the intended function H w.r.t. other operators besides constructors. E.g.,

$$H\ (t[x \wedge y])\ =\ \ldots H\ t \ldots x \ldots y \ldots$$
$$x\ \mathsf{fresh}\ H\ t\ \Rightarrow\ \ldots H\ t \ldots x \ldots y \ldots$$

Example: the depth function

depth : Trm $\to \mathbb{N}$

$$\text{depth}\ (\text{Vr}\ x) = 1$$
$$\text{depth}\ (\text{Ap}\ t_1\ t_2) = \max\left(\text{depth}\ t_1, \text{depth}\ t_2\right) + 1$$
$$\text{depth}\ (\text{Lm}\ x\ t) = \text{depth}\ t + 1$$

How to make this into a well-defined recursive function?

$$\text{depth}\ (t\ [x \wedge y]) = \text{depth}\ t$$

x fresh t implies True (vacuous)

In algebraic translation, the above means: Have organized $\mathbb{N}$ as an algebra as follows:

$$\text{Vr}^{\mathbb{N}}\ x = 1 \qquad m\ [x \wedge^{\mathbb{N}} y] = m$$
$$\text{Ap}^{\mathbb{N}}\ m\ n = m + n + 1 \qquad x\ \text{fresh}^{\mathbb{N}}\ m = \text{True}$$
$$\text{Lm}^{\mathbb{N}}\ x\ m = m + 1$$

Can use recursion theorems 1, 2, 3 or 3' – must verify some trivial identities.

Full-fledged recursors

The previous results give us only **iterators**

Obtain full-fledged recursors by:

- extending iteration to primitive recursion (general-purpose)
- factoring in variables to be "avoided" (binding-specific) – the Barendregt convention

From barebone to full-fledged recursors I

Iteration

- the constructors
  $\mathsf{Vr} : \mathsf{Var} \to A$
  $\mathsf{Ap} : A \to A \to A$
  $\mathsf{Lm} : \mathsf{Var} \to A \to A$
- swapping $\_[\_\wedge\_] : A \to \mathsf{Var} \to \mathsf{Var} \to A$

  etc.

$$H\ (\mathsf{Var}\ x) = \mathsf{Var}^A\ x$$
$$H\ (\mathsf{Ap}\ t_1\ t_2) = \mathsf{Ap}^A\ \ (H\ t_1)\ \ (H\ t_2)$$
$$H\ (\mathsf{Lm}\ x\ t) = \mathsf{Lm}^A\ x\ \ (H\ t)$$
$$H\ (t\ [x \wedge y]) = (H\ t)\ [x \wedge^A y]$$

From barebone to full-fledged recursors I

Iteration $\mapsto$ **Primitive recursion**

- the constructors
  $\mathsf{Vr} : \mathsf{Var} \to A$
  $\mathsf{Ap} : \mathsf{Trm} \to A \to \mathsf{Trm} \to A \to A$
  $\mathsf{Lm} : \mathsf{Var} \to \mathsf{Trm} \to A \to A$
- swapping $\_[\_\wedge\_] : \mathsf{Trm} \to A \to \mathsf{Var} \to \mathsf{Var} \to A$

  etc.

$$H\ (\mathsf{Var}\ x) = \mathsf{Var}^A\ x$$
$$H\ (\mathsf{Ap}\ t_1\ t_2) = \mathsf{Ap}^A\ t_1\ (H\ t_1)\ t_2\ (H\ t_2)$$
$$H\ (\mathsf{Lm}\ x\ t) = \mathsf{Lm}^A\ x\ t\ (H\ t)$$
$$H\ (t\ [x \wedge y]) = (t, H\ t)\ [x \wedge^A y]$$

From barebone to full-fledged recursors I

Iteration $\mapsto$ **Primitive recursion**

- the constructors
  $\mathsf{Vr} : \mathsf{Var} \to A$
  $\mathsf{Ap} : \mathsf{Trm} \to A \to \mathsf{Trm} \to A \to A$
  $\mathsf{Lm} : \mathsf{Var} \to \mathsf{Trm} \to A \to A$
- swapping $\_[\_\wedge\_] : \mathsf{Trm} \to A \to \mathsf{Var} \to \mathsf{Var} \to A$

  etc.

$$H\,(\mathsf{Var}\ x) = \mathsf{Var}^A\ x$$
$$H\,(\mathsf{Ap}\ t_1\ t_2) = \mathsf{Ap}^A\ t_1\ (H\ t_1)\ t_2\ (H\ t_2)$$
$$H\,(\mathsf{Lm}\ x\ t) = \mathsf{Lm}^A\ x\ t\ (H\ t)$$
$$H\,(t\,[x\wedge y]) = (t, H\ t)\,[x\wedge^A y]$$

Can use not only returned value of H, but also original term

From barebone to full-fledged recursors II

1. Fix a finite set of variables X.

2. Amend all algebraic properties to assume freshness of the binding variables w.r.t. X. E.g.,

   (SwRen) $z \neq x \Rightarrow z$ fresh $t \Rightarrow z \notin X \Rightarrow x \notin X \Rightarrow$
   $\mathsf{Lm}\ x\ t = \mathsf{Lm}\ z\ (t[z \wedge x])$
   (FCB) $\exists x.\ x \notin X\ \wedge\ \forall t.\ x$ fresh $\mathsf{Lm}\ x\ t$

3. Obtain correspondingly amended recursive clauses. E.g.,

   $x \notin X \Rightarrow H\ (\mathsf{Lm}\ x\ t) = \mathsf{Lm}^A\ x\ t\ (H\ t)$
   $x, y \notin X \Rightarrow H\ (t\ [x \wedge y]) = (t, H\ t)\ [x \wedge^A y]$

From barebone to full-fledged recursors II

1. Fix a finite set of variables X.

2. Amend all algebraic properties to assume freshness of the binding variables w.r.t. X. E.g.,

   (SwRen) $z \neq x \Rightarrow z$ fresh $t \Rightarrow z \notin X \Rightarrow x \notin X \Rightarrow$
   $\mathsf{Lm}\ x\ t = \mathsf{Lm}\ z\ (t[z \wedge x])$
   (FCB) $\exists x.\ x \notin X\ \wedge\ \forall t.\ x$ fresh $\mathsf{Lm}\ x\ t$

3. Obtain correspondingly amended recursive clauses. E.g.,

   $x \notin X \Rightarrow H\ (\mathsf{Lm}\ x\ t) = \mathsf{Lm}^A\ x\ t\ (H\ t)$
   $x, y \notin X \Rightarrow H\ (t\ [x \wedge y]) = (t, H\ t)\ [x \wedge^A y]$

E.g., when defining substitution:

Fix x, s
Take $X = \mathsf{FVars}\ s \cup \{x\}$
Clause for Lm:
$y \neq x \Rightarrow y$ fresh $s \Rightarrow (\mathsf{Lm}\ y\ t)[s/x] = \mathsf{Lm}\ y\ (t[s/x])$

From barebone to full-fledged recursors II

1. Fix a finite set of variables X.

2. Amend all algebraic properties to assume freshness of the binding variables w.r.t. X. E.g.,

   (SwRen) $z \neq x \Rightarrow z$ fresh $t \Rightarrow z \notin X \Rightarrow x \notin X \Rightarrow$
   $\mathsf{Lm}\ x\ t = \mathsf{Lm}\ z\ (t[z \wedge x])$
   (FCB) $\exists x.\ x \notin X\ \wedge\ \forall t.\ x$ fresh $\mathsf{Lm}\ x\ t$

3. Obtain correspondingly amended recursive clauses. E.g.,

   $x \notin X \Rightarrow H\ (\mathsf{Lm}\ x\ t) = \mathsf{Lm}^A\ x\ t\ (H\ t)$
   $x, y \notin X \Rightarrow H\ (t\ [x \wedge y]) = (t, H\ t)\ [x \wedge^A y]$

E.g., when defining substitution:

   Fix x, s
   Take $X = \mathsf{FVars}\ s \cup \{x\}$
   Clause for Lm:
   $y \neq x \Rightarrow y$ fresh $s \Rightarrow (\mathsf{Lm}\ y\ t)[s/x] = \mathsf{Lm}\ y\ (t[s/x])$

Note: Can go even further, and assume that X varies across the recursive calls.

Summary I

1. When working with λ-terms, we prefer to consider

   the standard constructors Vr, Ap, Lm : Var $\to$ Trm $\to$ Trm

   rather than constructors "made up" from various encodings, e.g., DBLm : Trm $\to$ Trm or LLm : (Var $\to$ Trm) $\to$ Trm

Why?

Summary I

1. When working with λ-terms, we prefer to consider

   the standard constructors Vr, Ap, Lm : Var $\to$ Trm $\to$ Trm

   rather than constructors "made up" from various encodings, e.g., DBLm : Trm $\to$ Trm or LLm : (Var $\to$ Trm) $\to$ Trm

Why? Because our constructions in logic and programming languages refer to them.

Summary I

1. When working with λ-terms, we prefer to consider

   the standard constructors Vr, Ap, Lm : Var $\to$ Trm $\to$ Trm

   rather than constructors "made up" from various encodings, e.g., DBLm : Trm $\to$ Trm or LLm : (Var $\to$ Trm) $\to$ Trm

Why? Because our constructions in logic and programming languages refer to them.

2. We also prefer to work with λ-terms quotiented to α-equivalence
Why?

Summary I

1. When working with λ-terms, we prefer to consider

   the standard constructors Vr, Ap, Lm : Var $\to$ Trm $\to$ Trm

   rather than constructors "made up" from various encodings, e.g., DBLm : Trm $\to$ Trm or LLm : (Var $\to$ Trm) $\to$ Trm

Why? Because our constructions in logic and programming languages refer to them.

2. We also prefer to work with λ-terms quotiented to α-equivalence

Why? Because important operators such as substitution are well-behaved on quotiented terms only.

Summary II

3. Because (α-quotiented) λ-terms are not a free datatype, to recurse over them while referring to their standard constructor, i.e., write recursive clauses

$$H\ (\mathsf{Lm}\ x\ t)\ =\ \ldots x \ldots t \ldots H\ t \ldots$$

must pay a price:

- consider more operators (e.g., freshness, swapping, permutation, substitution)
- verify algebraic laws for the target domain

Summary II

3. Because (α-quotiented) λ-terms are not a free datatype, to recurse over them while referring to their standard constructor, i.e., write recursive clauses

$$H\ (\mathsf{Lm}\ x\ t)\ =\ \ldots x \ldots t \ldots H\ t \ldots$$

must pay a price:

- consider more operators (e.g., freshness, swapping, permutation, substitution)
- verify algebraic laws for the target domain

In return, we get:

- not only commutation with the constructors (the traditional recursive clauses)
- but also useful commutation with the additional operators (preservation of freshness, "substitution lemmas")

Summary III

4. Want our recursors to be as expressive as possible (be able to define large classes of functions).

We classified and compared recursors from the literature and improved on their expressiveness.

Our motivation:

- general theory of syntax with bindings, formalized in Isabelle/HOL
- user-friendly definitional package under development

Summary III

4. Want our recursors to be as expressive as possible (be able to define large classes of functions).

We classified and compared recursors from the literature and improved on their expressiveness.

Our motivation:

- general theory of syntax with bindings, formalized in Isabelle/HOL
- user-friendly definitional package under development

Questions not discussed in this talk:

- How do we formally compare the expressiveness of different recursors?
- How do the results scale to arbitrary syntaxes with bindings?

Summary III

4. Want our recursors to be as expressive as possible (be able to define large classes of functions).

We classified and compared recursors from the literature and improved on their expressiveness.

Our motivation:

- general theory of syntax with bindings, formalized in Isabelle/HOL
- user-friendly definitional package under development

Questions not discussed in this talk:

- How do we formally compare the expressiveness of different recursors?
- How do the results scale to arbitrary syntaxes with bindings?

Thank you

Reserve Slides

Comparing expressiveness of recursion principles I

We fix

- a "base" category $\mathcal{B}$
- and an object $T \in |\mathcal{B}|$

A recursion principle for $(\mathcal{B}, T)$ is an "extension" category $\mathcal{C}$ together with a "reduct" functor $R : \mathcal{C} \to \mathcal{B}$ such that

- $\mathcal{C}$ has an initial object I
- $R\ I = T$ (can also assume $R\ I \simeq T$, but assume "$=$" for simplicity)

Intuition: The objects of $\mathcal{C}$ extend those of $\mathcal{B}$ with additional structure. In particular, I extends T.

How it works: Let $B \in |\mathcal{B}|$, an "intended target domain". To define a function $f : T \to B$, we do the following:

(Step 1) We "extend" B to an object of $\mathcal{C}$, i.e., take $C \in |\mathcal{C}|$ such that $R\ C = B$

(Step 2) We obtain $g : I \to C$ by initiality

(Step 3) We take $f = R\ g : T = R\ I \to R\ C = B$

Comparing expressiveness of recursion principles II

Let $(\mathcal{C}_1, R_1, I_1)$ and $(\mathcal{C}_2, R_2, I_2)$ be two recursion principles for $(\mathcal{B}, T)$. We say that $(\mathcal{C}_1, R_1, I_1)$ encompasses (is at least as expressive as) $(\mathcal{C}_2, R_2, I_2)$, written $(\mathcal{C}_1, R_1, I_1) \geq (\mathcal{C}_2, R_2, I_2)$, if there exists a functor $F : \mathcal{C}_1 \to \mathcal{C}_2$ such that:

(1) $R_2 \circ F = R_1$

(2) for all $C_2 \in |\mathcal{C}_2|$ there exists $C_1 \in |\mathcal{C}_1|$ and a morphism $h : F\ C_1 \to C_2$ s.t. $R_1\ C_1 = R_2\ C_2$ and $R_2\ h = 1_B$

(By initiality, condition 2 implies $I_2 \simeq F\ I_1$. For simplicity, we will assume $I_2 = F\ I_1$.)

Note: The image of F through $\mathcal{C}_1$ is something like an initial segment of $\mathcal{C}_2$.

Comparing expressiveness of recursion principles III

Intuition: The first principle can simulate the second principle.
Let $B \in |\mathcal{B}|$.

(Step 1) We take $C_2 \in |\mathcal{C}_2|$ such that $R_2\ C_2 = B$.
We take C_1 and $h : F\ C_1 \to C_2$ etc.

(Step 2) We obtain $g_2 : I_2 \to C_2$ by initiality.
We obtain $g_1 : I_1 \to C_1$ by initiality.
We note that $g_2 = h \circ F\ g_1$ by initiality.

(Step 3) We take $f_2 = R_2\ g_2$.
We take $f_1 = R_1\ g_1$.
We note that $f_2 = f_1$.

Thus, via the "simulation" F, we can use the first principle to the same effect as the second.

Back to terms

Starting point: We want a recursion principle that allows us to recurse over the standard constructors:

$$H\ (\mathsf{Vr}\ x) = \ldots x \ldots$$
$$H\ (\mathsf{Ap}\ t_1\ t_2) = \ldots (H\ t_1) \ldots (H\ t_2)$$
$$H\ (\mathsf{Lm}\ x\ t) = \ldots x \ldots (H\ t) \ldots$$

Hence take $\mathcal{B}$ to be Alg_{Σ_0}, the category of algebras over the signature $\Sigma_0 = \{(\mathsf{Vr}_x)_{x \in \mathsf{Var}}, \mathsf{Ap}, (\mathsf{Lm}_x)_{x \in \mathsf{Var}}\}$.

To "help" recursion, we need to extend Σ_0 to larger signatures Σ, factoring in the freshness predicate, the swapping, permutation and substitution operator, etc. So the extensions $\mathcal{C}$ will be classes of Σ-algebras satisfying various properties.

All our expressiveness comparison results are then instances of the abstract framework.