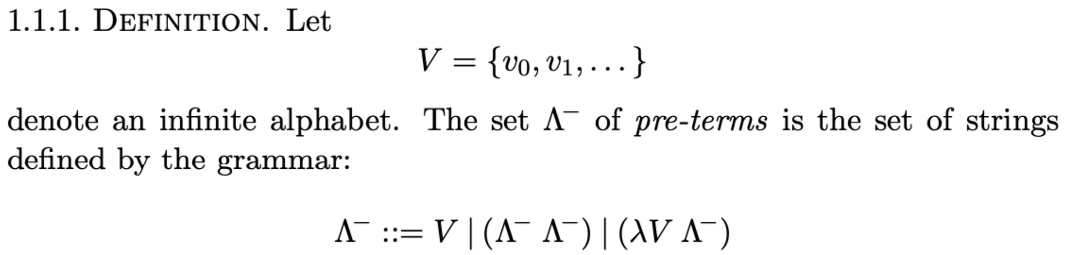
HERIOT WATT UNIVERSITY

Mechanization of Binders

Kathrin Stark

SPLV 2024

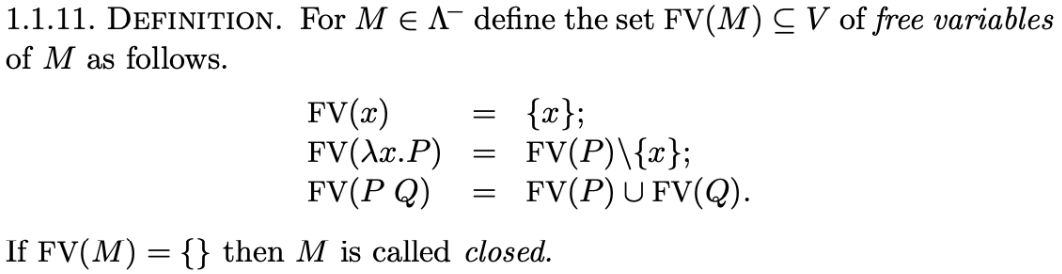


1.1.1. Definition. Let

$$V = \{v_0, v_1, \dots\}$$

denote an infinite alphabet. The set Λ^- of *pre-terms* is the set of strings defined by the grammar:

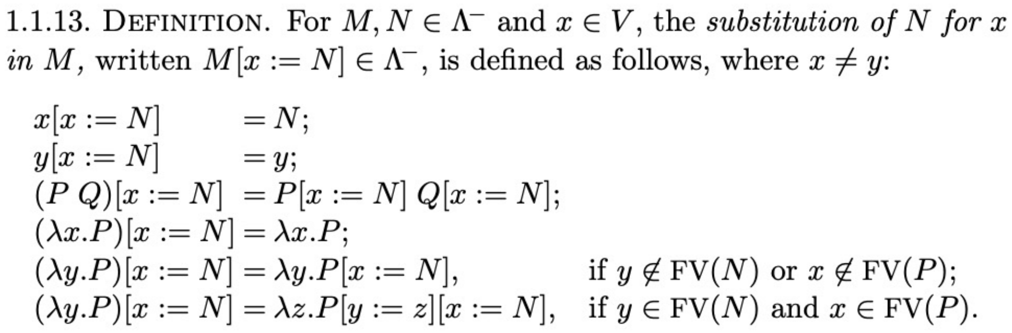
$$\Lambda^- ::= V \mid (\Lambda^- \; \Lambda^-) \mid (\lambda V \; \Lambda^-)$$



1.1.11. Definition. For $M \in \Lambda^-$ define the set $\mathrm{FV}(M) \subseteq V$ of *free variables* of M as follows.

$$\begin{array}{lcl} \mathrm{FV}(x) & = & \{x\}; \\ \mathrm{FV}(\lambda x.P) & = & \mathrm{FV}(P)\backslash\{x\}; \\ \mathrm{FV}(P\;Q) & = & \mathrm{FV}(P) \cup \mathrm{FV}(Q). \end{array}$$

If $\mathrm{FV}(M) = \{\}$ then M is called *closed.*

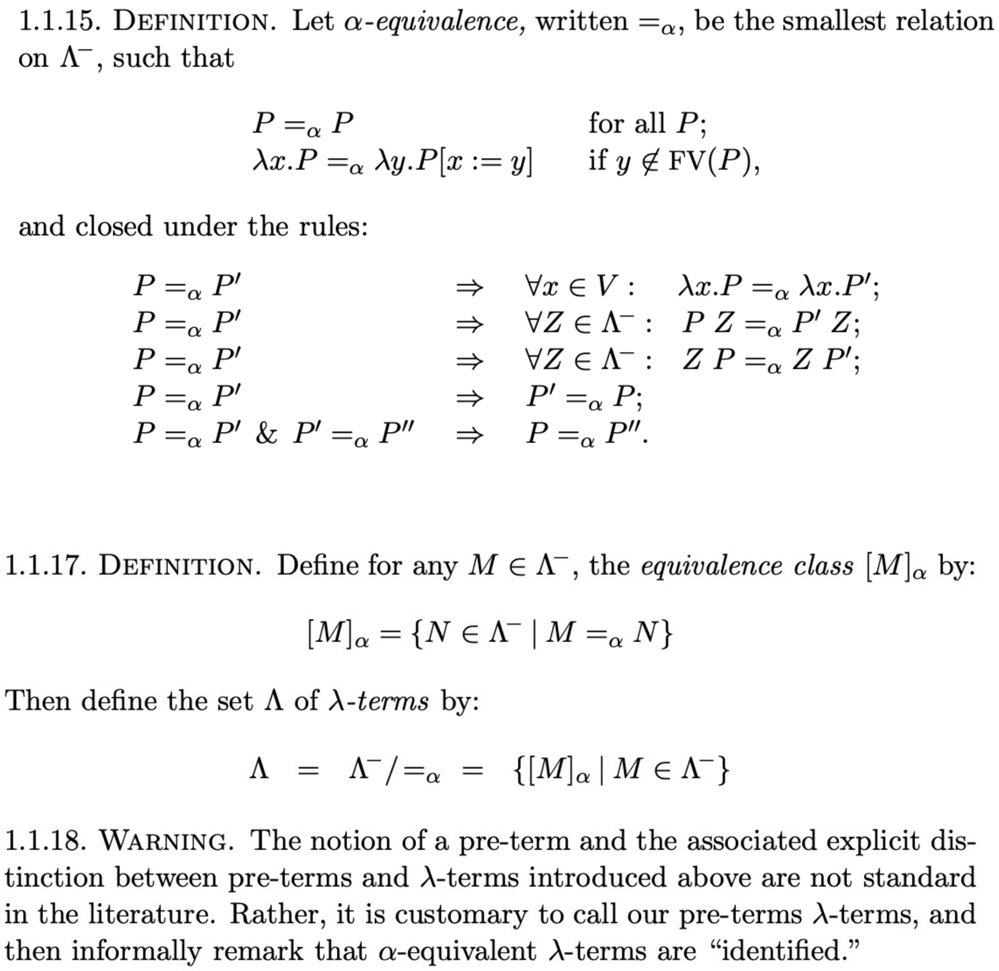


1.1.13. Definition. For $M, N \in \Lambda^-$ and $x \in V$, the *substitution of N for x in M*, written $M[x := N] \in \Lambda^-$, is defined as follows, where $x \neq y$:

$$\begin{array}{lll} x[x := N] & = N; & \\ y[x := N] & = y; & \\ (P\;Q)[x := N] & = P[x := N]\;Q[x := N]; & \\ (\lambda x.P)[x := N] & = \lambda x.P; & \\ (\lambda y.P)[x := N] & = \lambda y.P[x := N], & \text{if } y \notin \mathrm{FV}(N) \text{ or } x \notin \mathrm{FV}(P); \\ (\lambda y.P)[x := N] & = \lambda z.P[y := z][x := N], & \text{if } y \in \mathrm{FV}(N) \text{ and } x \in \mathrm{FV}(P). \end{array}$$

Define preterms…

Sørenson, Urzyczyn - Lectures on the Curry-Howard Isomorphism



1.1.15. Definition. Let *α-equivalence,* written $=_\alpha$, be the smallest relation on Λ^-, such that

$$\begin{array}{ll} P =_\alpha P & \text{for all } P; \\ \lambda x.P =_\alpha \lambda y.P[x := y] & \text{if } y \notin \mathrm{FV}(P), \end{array}$$

and closed under the rules:

$$\begin{array}{lcll} P =_\alpha P' & \Rightarrow & \forall x \in V: & \lambda x.P =_\alpha \lambda x.P'; \\ P =_\alpha P' & \Rightarrow & \forall Z \in \Lambda^-: & P\ Z =_\alpha P'\ Z; \\ P =_\alpha P' & \Rightarrow & \forall Z \in \Lambda^-: & Z\ P =_\alpha Z\ P'; \\ P =_\alpha P' & \Rightarrow & P' =_\alpha P; & \\ P =_\alpha P' \ \&\ P' =_\alpha P'' & \Rightarrow & P =_\alpha P''. & \end{array}$$

1.1.17. Definition. Define for any $M \in \Lambda^-$, the *equivalence class* $[M]_\alpha$ by:

$$[M]_\alpha = \{N \in \Lambda^- \mid M =_\alpha N\}$$

Then define the set Λ of *λ-terms* by:

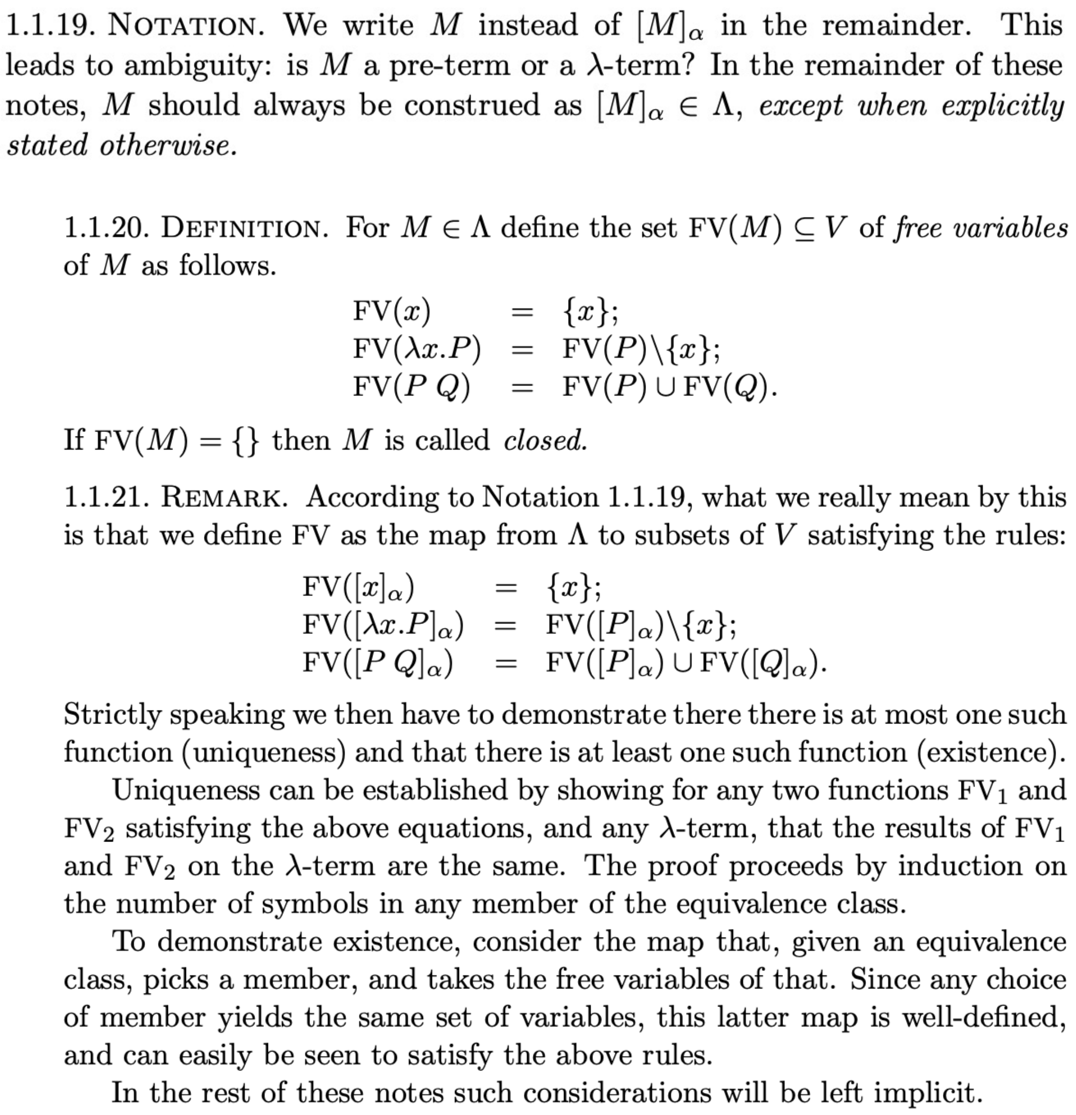
$$\Lambda \quad = \quad \Lambda^- / =_\alpha \quad = \quad \{[M]_\alpha \mid M \in \Lambda^-\}$$

1.1.18. Warning. The notion of a pre-term and the associated explicit distinction between pre-terms and λ-terms introduced above are not standard in the literature. Rather, it is customary to call our pre-terms λ-terms, and then informally remark that α-equivalent λ-terms are "identified."

… α-equivalence…

… actual terms …

Sørenson, Urzyczyn - Lectures on the Curry-Howard Isomorphism



1.1.19. Notation. We write M instead of $[M]_\alpha$ in the remainder. This leads to ambiguity: is M a pre-term or a λ-term? In the remainder of these notes, M should always be construed as $[M]_\alpha \in \Lambda$, *except when explicitly stated otherwise.*

1.1.20. Definition. For $M \in \Lambda$ define the set $\mathrm{FV}(M) \subseteq V$ of *free variables* of M as follows.

$$\begin{array}{lcl} \mathrm{FV}(x) & = & \{x\}; \\ \mathrm{FV}(\lambda x.P) & = & \mathrm{FV}(P)\backslash\{x\}; \\ \mathrm{FV}(P\ Q) & = & \mathrm{FV}(P) \cup \mathrm{FV}(Q). \end{array}$$

If $\mathrm{FV}(M) = \{\}$ then M is called *closed.*

1.1.21. Remark. According to Notation 1.1.19, what we really mean by this is that we define FV as the map from Λ to subsets of V satisfying the rules:

$$\begin{array}{lcl} \mathrm{FV}([x]_\alpha) & = & \{x\}; \\ \mathrm{FV}([\lambda x.P]_\alpha) & = & \mathrm{FV}([P]_\alpha)\backslash\{x\}; \\ \mathrm{FV}([P\ Q]_\alpha) & = & \mathrm{FV}([P]_\alpha) \cup \mathrm{FV}([Q]_\alpha). \end{array}$$

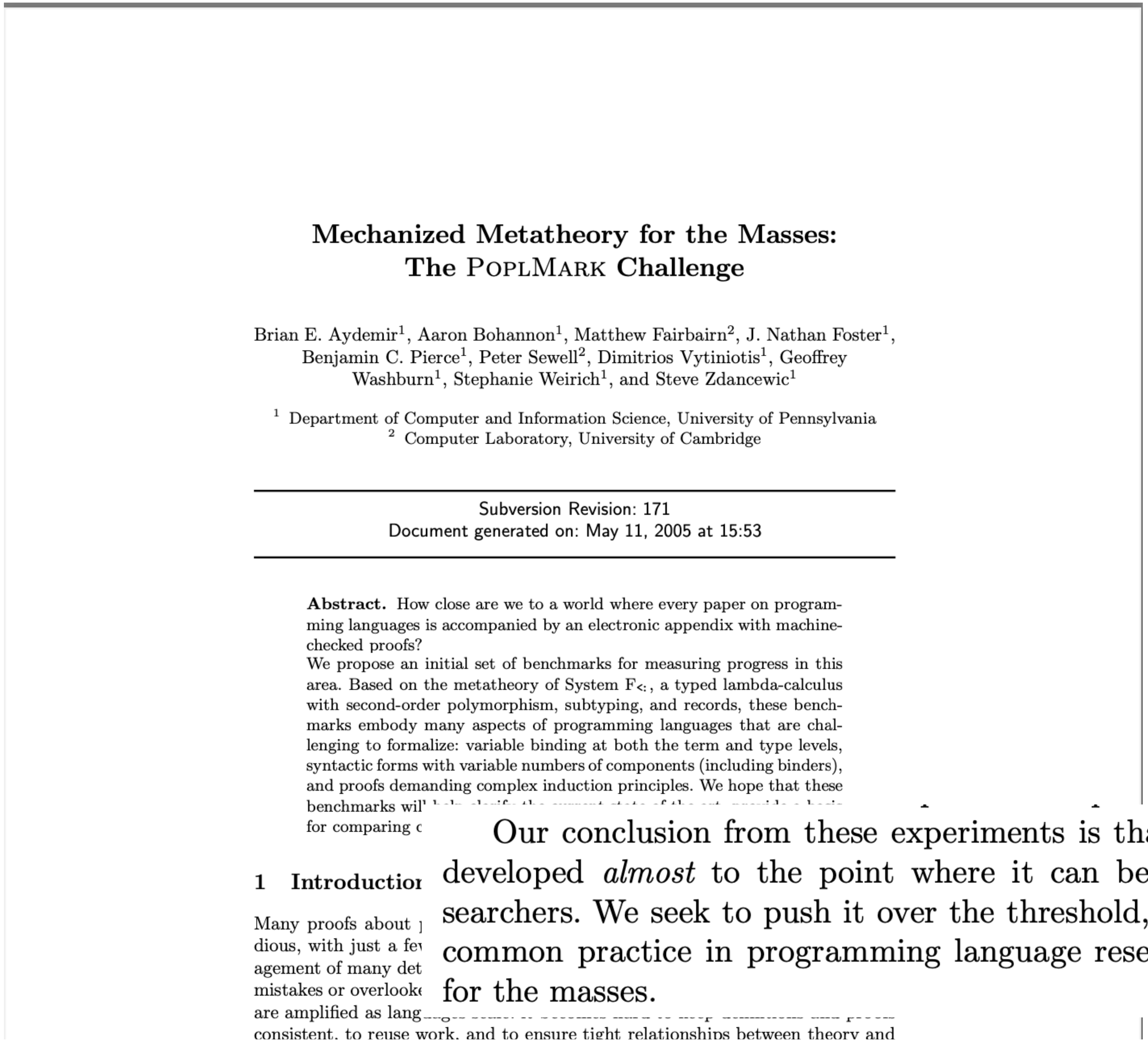
Strictly speaking we then have to demonstrate there there is at most one such function (uniqueness) and that there is at least one such function (existence).

Uniqueness can be established by showing for any two functions FV_1 and FV_2 satisfying the above equations, and any λ-term, that the results of FV_1 and FV_2 on the λ-term are the same. The proof proceeds by induction on the number of symbols in any member of the equivalence class.

To demonstrate existence, consider the map that, given an equivalence class, picks a member, and takes the free variables of that. Since any choice of member yields the same set of variables, this latter map is well-defined, and can easily be seen to satisfy the above rules.

In the rest of these notes such considerations will be left implicit.

… and definitions on terms.



Mechanized Metatheory for the Masses: The PoplMark Challenge

Brian E. Aydemir[1], Aaron Bohannon[1], Matthew Fairbairn[2], J. Nathan Foster[1],
Benjamin C. Pierce[1], Peter Sewell[2], Dimitrios Vytiniotis[1], Geoffrey
Washburn[1], Stephanie Weirich[1], and Steve Zdancewic[1]

[1] Department of Computer and Information Science, University of Pennsylvania
[2] Computer Laboratory, University of Cambridge

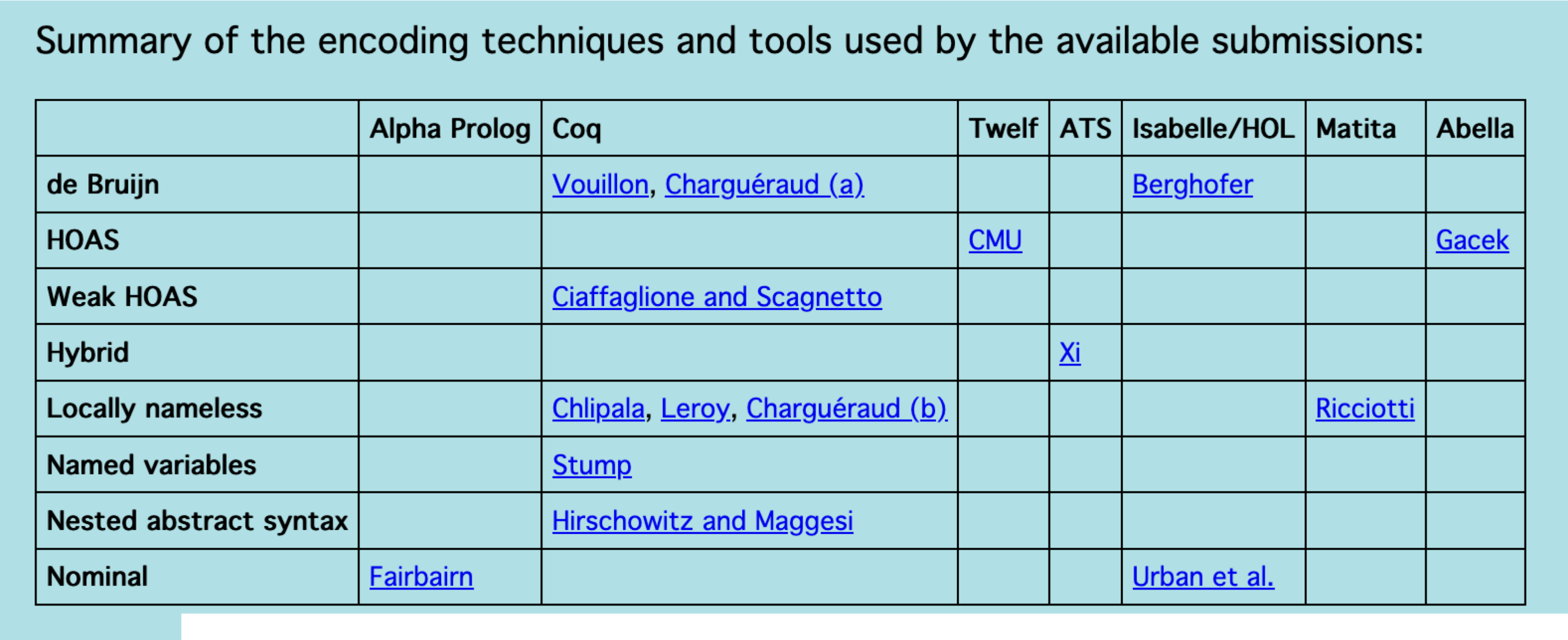
Subversion Revision: 171
Document generated on: May 11, 2005 at 15:53

Abstract. How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine-checked proofs?
We propose an initial set of benchmarks for measuring progress in this area. Based on the metatheory of System $F_{<:}$, a typed lambda-calculus with second-order polymorphism, subtyping, and records, these benchmarks embody many aspects of programming languages that are challenging to formalize: variable binding at both the term and type levels, syntactic forms with variable numbers of components (including binders), and proofs demanding complex induction principles. We hope that these benchmarks wil
for comparing c

1 Introductio

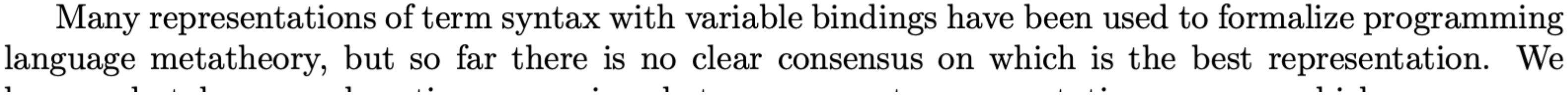
Many proofs about
dious, with just a fe
agement of many det
mistakes or overlook
are amplified as lang
consistent, to reuse work, and to ensure tight relationships between theory and

Our conclusion from these experiments is that the relevant technology has developed *almost* to the point where it can be widely used by language researchers. We seek to push it over the threshold, making the use of proof tools common practice in programming language research—mechanized metatheory for the masses.

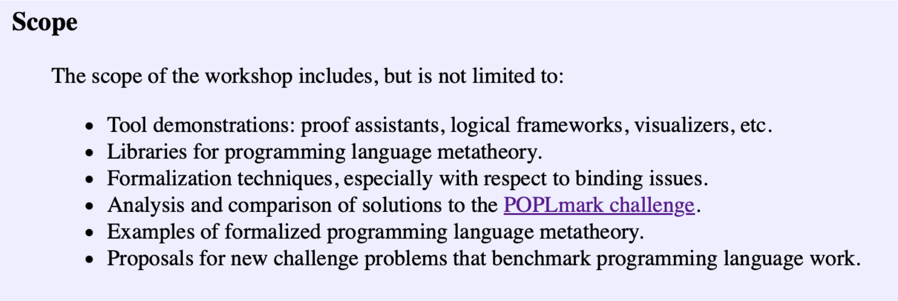


Summary of the encoding techniques and tools used by the available submissions:

	Alpha Prolog	Coq	Twelf	ATS	Isabelle/HOL	Matita	Abella
de Bruijn		Vouillon, Charguéraud (a)			Berghofer		
HOAS			CMU				Gacek
Weak HOAS		Ciaffaglione and Scagnetto					
Hybrid				Xi			
Locally nameless		Chlipala, Leroy, Charguéraud (b)				Ricciotti	
Named variables		Stump					
Nested abstract syntax		Hirschowitz and Maggesi					
Nominal	Fairbairn				Urban et al.		



Many representations of term syntax with variable bindings have been used to formalize programming language metatheory, but so far there is no clear consensus on which is the best representation. We



Scope

The scope of the workshop includes, but is not limited to:

- Tool demonstrations: proof assistants, logical frameworks, visualizers, etc.
- Libraries for programming language metatheory.
- Formalization techniques, especially with respect to binding issues.
- Analysis and comparison of solutions to the POPLmark challenge.
- Examples of formalized programming language metatheory.
- Proposals for new challenge problems that benchmark programming language work.

Some Comparisons

- Aydemir et al.: Mechanized Metatheory for the Masses: The PoplMark Challenge 2005 https://www.seas.upenn.edu/~plclub/poplmark/
- Berghofer/Urban: A Head-to-Head Comparison of de Bruijn Indices and Names 2007
- Abel et al. - POPLMark Reloaded: Mechanizing Proofs by Logical Relations 2019 https://poplmark-reloaded.github.io
- Brian Aydemir, Stephan A. Zdancewic, and Stephanie Weirich. Abstracting syntax. 2009.
- https://jesper.sikanda.be/posts/1001-syntax-representations.html 2021
- Popescu, Andrei. "Nominal Recursors as Epi-Recursors." *Proceedings of the ACM on Programming Languages* 8.POPL (2024):

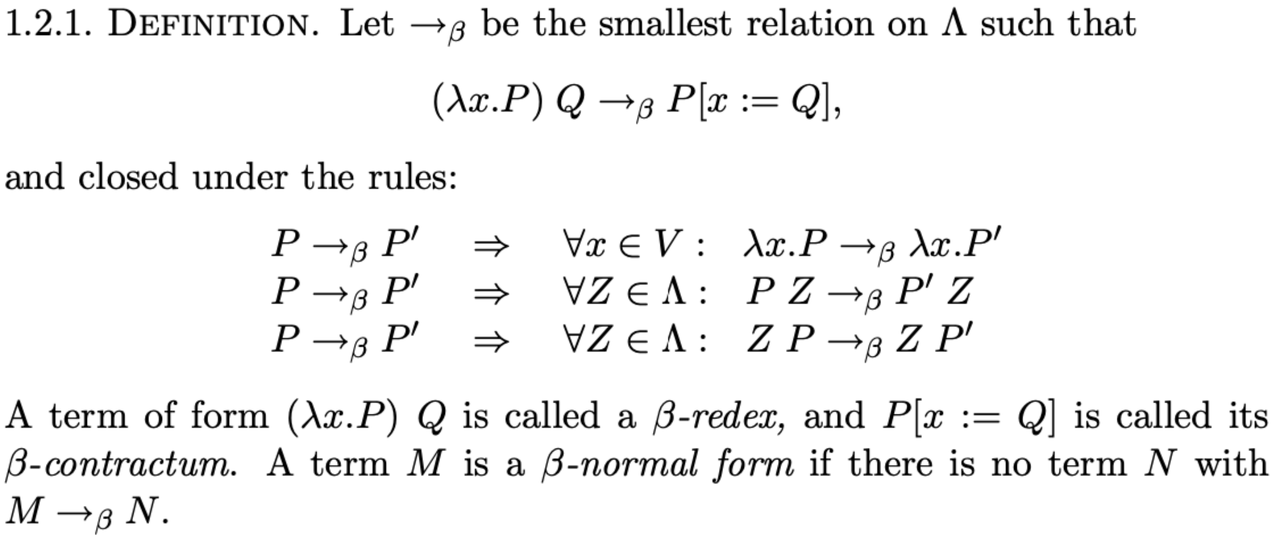
What to expect

- A short peek in different binder approaches:
  Pure de Bruijn, scoped de Bruijn, intrinsically typed, monadic, HOAS/CMTT, PHOAS, nominal, locally nameless

What *not* to expect:

- Completeness in any direction
- Less about tools/theoretical foundations

Running Example: Subject Reduction



1.2.1. Definition. Let $\to_\beta$ be the smallest relation on Λ such that

$$(\lambda x.P)\,Q \to_\beta P[x := Q],$$

and closed under the rules:

$$\begin{array}{lcll}
P \to_\beta P' & \Rightarrow & \forall x \in V: & \lambda x.P \to_\beta \lambda x.P' \\
P \to_\beta P' & \Rightarrow & \forall Z \in \Lambda: & P\,Z \to_\beta P'\,Z \\
P \to_\beta P' & \Rightarrow & \forall Z \in \Lambda: & Z\,P \to_\beta Z\,P'
\end{array}$$

A term of form $(\lambda x.P)\,Q$ is called a *β-redex,* and $P[x := Q]$ is called its *β-contractum.* A term M is a *β-normal form* if there is no term N with $M \to_\beta N$.

3.1.7. DEFINITION. The *substitution of type τ for type variable α in type σ,* written $\sigma[\alpha := \tau]$, is defined by:

$$\begin{array}{lcll} \alpha[\alpha := \tau] & = & \tau & \\ \beta[\alpha := \tau] & = & \beta & \text{if } \alpha \neq \beta \\ (\sigma_1 \to \sigma_2)[\alpha := \tau] & = & \sigma_1[\alpha := \tau] \to \sigma_2[\alpha := \tau] & \end{array}$$

The notation $\Gamma[\alpha := \tau]$ stands for the context $\{(x : \sigma[\alpha := \tau]) \mid (x : \sigma) \in \Gamma\}$.

The set C of *contexts* is the set of all sets of pairs of the form

$$\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$$

with $\tau_1, \ldots, \tau_n \in \Pi$, $x_1, \ldots, x_n \in V$ (variables of Λ) and $x_i \neq x_j$ for $i \neq j$.

3.1.8. PROPOSITION (Substitution lemma).

(i) *If* $\Gamma \vdash M : \sigma$*, then* $\Gamma[\alpha := \tau] \vdash M : \sigma[\alpha := \tau]$.

(ii) *If* $\Gamma, x : \tau \vdash M : \sigma$ *and* $\Gamma \vdash N : \tau$ *then* $\Gamma \vdash M[x := N] : \sigma$.

PROOF. By induction on the derivation of $\Gamma \vdash M : \sigma$ and generation of $\Gamma, x : \tau \vdash M : \sigma$, respectively. □

The following shows that reduction preserves typing.

3.1.9. PROPOSITION (Subject reduction). *If* $\Gamma \vdash M : \sigma$ *and* $M \to_\beta N$*, then* $\Gamma \vdash N : \sigma$.

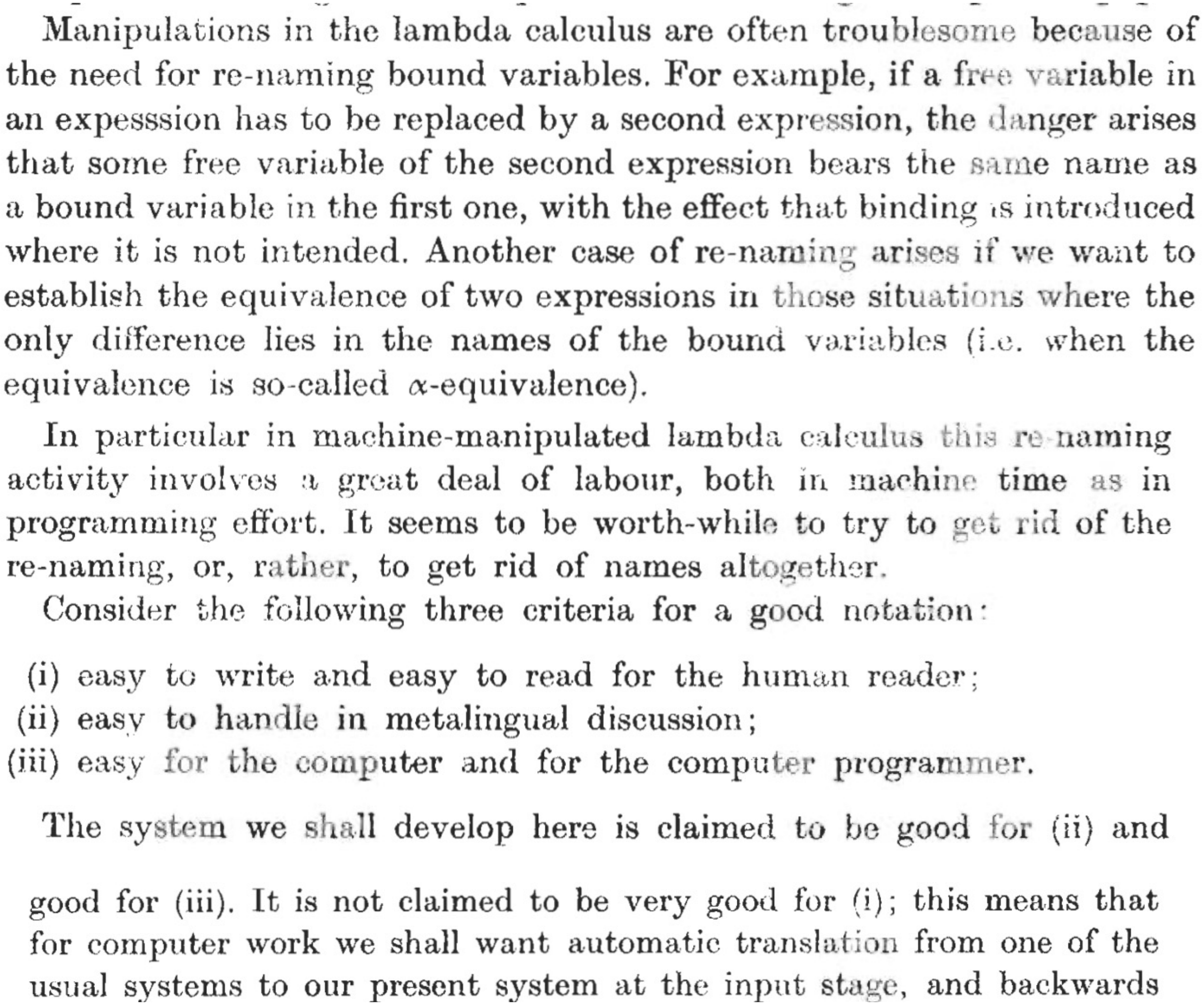
PROOF. By induction on the derivation of $M \to_\beta N$ using the substitution lemma and the generation lemma. □

Nameless Approaches

Kathrin Stark

SPLV 2024

De Bruijn Syntax



Manipulations in the lambda calculus are often troublesome because of the need for re-naming bound variables. For example, if a free variable in an expesssion has to be replaced by a second expression, the danger arises that some free variable of the second expression bears the same name as a bound variable in the first one, with the effect that binding is introduced where it is not intended. Another case of re-naming arises if we want to establish the equivalence of two expressions in those situations where the only difference lies in the names of the bound variables (i.e. when the equivalence is so-called α-equivalence).

In particular in machine-manipulated lambda calculus this re-naming activity involves a great deal of labour, both in machine time as in programming effort. It seems to be worth-while to try to get rid of the re-naming, or, rather, to get rid of names altogether.

Consider the following three criteria for a good notation:

(i) easy to write and easy to read for the human reader;
(ii) easy to handle in metalingual discussion;
(iii) easy for the computer and for the computer programmer.

The system we shall develop here is claimed to be good for (ii) and

good for (iii). It is not claimed to be very good for (i); this means that for computer work we shall want automatic translation from one of the usual systems to our present system at the input stage, and backwards

De Bruijn, Nicolaas Govert. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem." 1972

De Bruijn Syntax

- **Idea**: α-equivalence = definitional equality
- **Terms:**

```
Inductive tm: Type :=
| var_tm : nat -> tm
| app : tm -> tm -> tm
| lam : tm -> tm.
```

- **Example term:** $\lambda\, x. z\ (\lambda\, y. (x\ y)\ z) \Rightarrow \lambda. 1\ (\lambda. (1\ 0)\ 2)$

```
lam (app (var_tm 1) (lam (app (app (var_tm 1) (var_tm 0))
(var_tm 2)))
```

(Parallel) Substitutions

de Bruijn ‘72

$$(s \cdot \sigma)(x) := \begin{cases} s & \text{if } x = 0 \\ \sigma(x-1) & \text{otherwise} \end{cases}$$

$$\mathsf{id}(x) := x \qquad\qquad \uparrow(x) := x + 1$$

$$\begin{aligned} x[\sigma] &= \sigma(x) \\ (s\,t)[\sigma] &= (s[\sigma])\,(t[\sigma]) \qquad (\sigma \circ \tau)(x) = \sigma(x)[\tau] \\ (\lambda.\, s)[\sigma] &= \lambda.\,(s[\Uparrow\sigma]) \end{aligned}$$

$$\left.\Uparrow\sigma := 0 \cdot (\sigma \circ \uparrow) \quad\right\} \text{Requires again substitution}$$

Two-Level Approach: Adams, R.: Formalized metatheory with terms represented by an indexed family of types. In: Types for Proofs and Programs, Lecture Notes in Computer Science, vol. 3839, pp. 1–16. Springer Berlin Heidelberg (2006)

A convergent + complete rewriting system

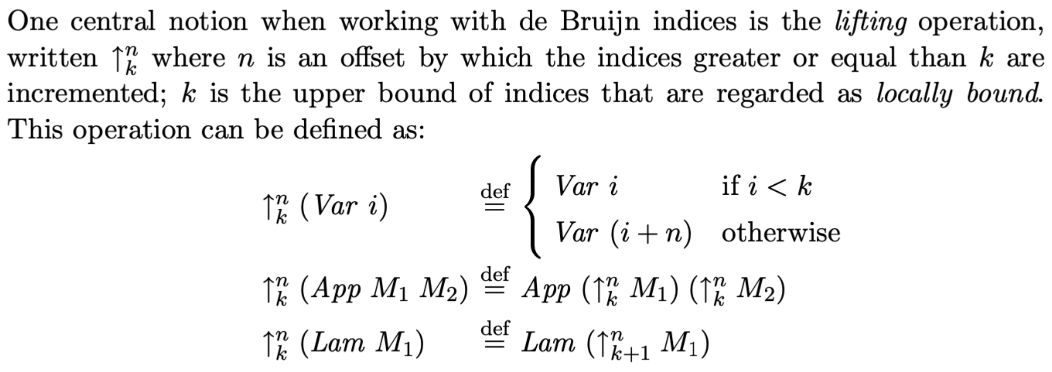
$$
\begin{aligned}
(st)[\sigma] &\equiv (s[\sigma])(t[\sigma]) & \mathsf{id} \circ \sigma &\equiv \sigma \\
(\lambda.\ s)[\sigma] &\equiv \lambda.\ (s[0 \cdot \sigma \circ \uparrow]) & \sigma \circ \mathsf{id} &\equiv \sigma \\
0[s \cdot \sigma] &\equiv s & (\sigma \circ \tau) \circ \theta &\equiv \sigma \circ (\tau \circ \theta) \\
\uparrow \circ (s \cdot \sigma) &\equiv \sigma & (s \cdot \sigma) \circ \tau &\equiv s[\tau] \cdot \sigma \circ \tau \\
s[\mathsf{id}] &\equiv s & s[\sigma][\tau] &\equiv s[\sigma \circ \tau] \\
0[\sigma] \cdot (\uparrow \circ \sigma) &\equiv \sigma & 0 \cdot \uparrow &\equiv \mathsf{id}
\end{aligned}
$$

s = t can be decided via the above rewriting system

Abadi et al.: Explicit Substitutions '96

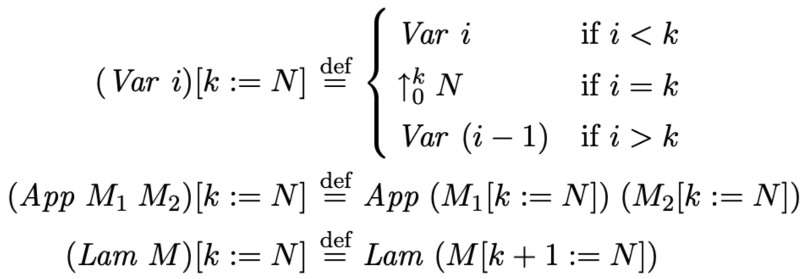
Schäfer, S., Smolka, G., Tebbi, T.: Completeness and decidability of de Bruijn substitution algebra in Coq '15

Single-Point de Bruijn Substitutions



One central notion when working with de Bruijn indices is the *lifting* operation, written $\uparrow^n_k$ where n is an offset by which the indices greater or equal than k are incremented; k is the upper bound of indices that are regarded as *locally bound.* This operation can be defined as:

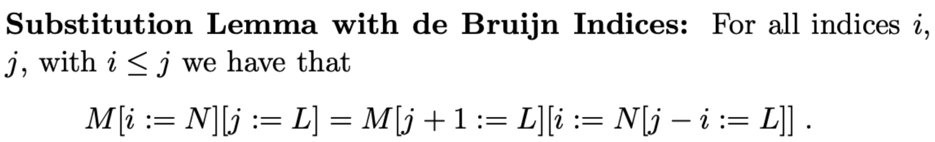
$$
\begin{aligned}
\uparrow^n_k (\mathit{Var}\ i) &\overset{\text{def}}{=} \begin{cases} \mathit{Var}\ i & \text{if } i < k \\ \mathit{Var}\ (i+n) & \text{otherwise} \end{cases} \\
\uparrow^n_k (\mathit{App}\ M_1\ M_2) &\overset{\text{def}}{=} \mathit{App}\ (\uparrow^n_k M_1)\ (\uparrow^n_k M_2) \\
\uparrow^n_k (\mathit{Lam}\ M_1) &\overset{\text{def}}{=} \mathit{Lam}\ (\uparrow^n_{k+1} M_1)
\end{aligned}
$$



$$
\begin{aligned}
(\mathit{Var}\ i)[k := N] &\overset{\text{def}}{=} \begin{cases} \mathit{Var}\ i & \text{if } i < k \\ \uparrow^k_0 N & \text{if } i = k \\ \mathit{Var}\ (i-1) & \text{if } i > k \end{cases} \\
(\mathit{App}\ M_1\ M_2)[k := N] &\overset{\text{def}}{=} \mathit{App}\ (M_1[k := N])\ (M_2[k := N]) \\
(\mathit{Lam}\ M)[k := N] &\overset{\text{def}}{=} \mathit{Lam}\ (M[k+1 := N])
\end{aligned}
$$

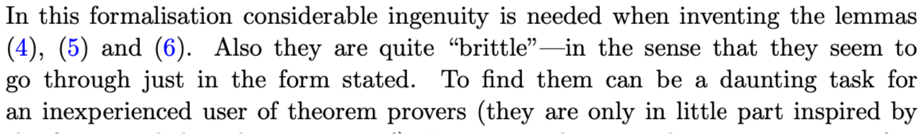
From Berghofer/Urban: A Head-to-Head Comparison of de Bruijn Indices and Names

Single-Point de Bruijn Substitutions (ctd.)

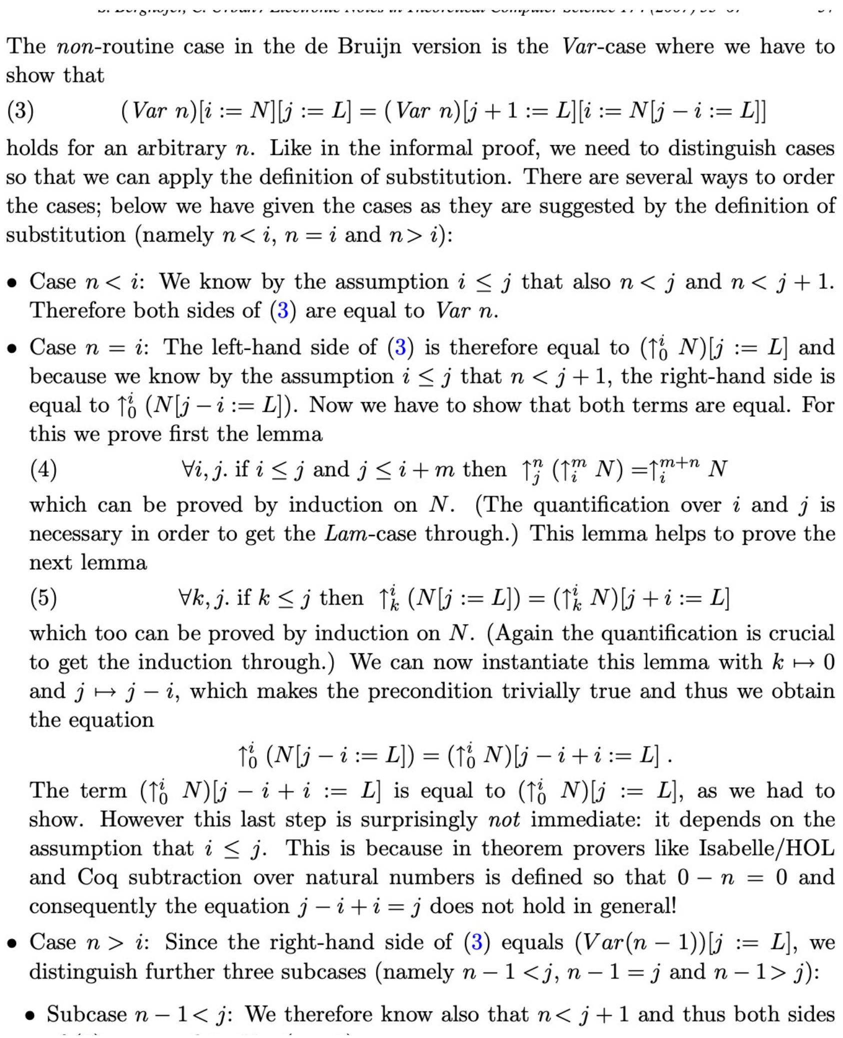


Substitution Lemma with de Bruijn Indices: For all indices i, j, with $i \leq j$ we have that

$$M[i := N][j := L] = M[j+1 := L][i := N[j-i := L]]\ .$$



In this formalisation considerable ingenuity is needed when inventing the lemmas (4), (5) and (6). Also they are quite “brittle”—in the sense that they seem to go through just in the form stated. To find them can be a daunting task for an inexperienced user of theorem provers (they are only in little part inspired by



The *non*-routine case in the de Bruijn version is the *Var*-case where we have to show that

(3) $$(\mathit{Var}\ n)[i := N][j := L] = (\mathit{Var}\ n)[j+1 := L][i := N[j-i := L]]$$

holds for an arbitrary n. Like in the informal proof, we need to distinguish cases so that we can apply the definition of substitution. There are several ways to order the cases; below we have given the cases as they are suggested by the definition of substitution (namely $n< i$, $n = i$ and $n> i$):

- Case $n < i$: We know by the assumption $i \leq j$ that also $n < j$ and $n < j+1$. Therefore both sides of (3) are equal to *Var* n.
- Case $n = i$: The left-hand side of (3) is therefore equal to $(\uparrow^i_0 N)[j := L]$ and because we know by the assumption $i \leq j$ that $n < j+1$, the right-hand side is equal to $\uparrow^i_0 (N[j-i := L])$. Now we have to show that both terms are equal. For this we prove first the lemma

  (4) $$\forall i, j.\ \text{if } i \leq j \text{ and } j \leq i+m \text{ then } \uparrow^n_j (\uparrow^m_i N) = \uparrow^{m+n}_i N$$

  which can be proved by induction on N. (The quantification over i and j is necessary in order to get the *Lam*-case through.) This lemma helps to prove the next lemma

  (5) $$\forall k, j.\ \text{if } k \leq j \text{ then } \uparrow^i_k (N[j := L]) = (\uparrow^i_k N)[j+i := L]$$

  which too can be proved by induction on N. (Again the quantification is crucial to get the induction through.) We can now instantiate this lemma with $k \mapsto 0$ and $j \mapsto j-i$, which makes the precondition trivially true and thus we obtain the equation

  $$\uparrow^i_0 (N[j-i := L]) = (\uparrow^i_0 N)[j-i+i := L]\ .$$

  The term $(\uparrow^i_0 N)[j-i+i := L]$ is equal to $(\uparrow^i_0 N)[j := L]$, as we had to show. However this last step is surprisingly *not* immediate: it depends on the assumption that $i \leq j$. This is because in theorem provers like Isabelle/HOL and Coq subtraction over natural numbers is defined so that $0 - n = 0$ and consequently the equation $j - i + i = j$ does not hold in general!
- Case $n > i$: Since the right-hand side of (3) equals $(Var(n-1))[j := L]$, we distinguish further three subcases (namely $n-1 <j$, $n-1 = j$ and $n-1> j$):
  - Subcase $n-1< j$: We therefore know also that $n< j+1$ and thus both sides

Some Variations

De Bruijn Levels

- **Terms:**

```
Inductive tm: Type :=
| var_tm : nat -> tm
| app : tm -> tm -> tm
| lam : tm -> tm.
```

- **Example term:** $\lambda\, x.\, x\ (\lambda\, y.\, x\ y) \Rightarrow \lambda.\, 0\ (\lambda.\, 0\ 1)$

Stays the same

Assumes there exists a global root node

De Bruijn, Nicolaas Govert. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem." 1972
Cregut: An abstract machine for the normalization of λ-calculus. In Proc. Conf. on Lisp and Functional Programming, pages 333–340. ACM, 1990. ("reversed De Bruijn indexing")

De Bruijn Syntax

- **Idea**: α-equivalence = definitional equality
- **Terms:**

```
Inductive tm: Type :=
| var_tm : nat -> tm
| app : tm -> tm -> tm
| lam : tm -> tm.
```

- **Example term:** $\lambda\, x.z\ (\lambda\, y.(x\ y)\ z) \Rightarrow \lambda.1\ (\lambda.(1\ 0)\ 2)$

```
lam (app (var_tm 1) (lam (app (app (var_tm 1) (var_tm 0))
(var_tm 2))))
```

Example: η-reduction rule for λ-terms

$$\frac{x \notin \mathrm{FV}\ s}{\lambda x.\ s\, x \rhd s} \qquad \frac{}{\lambda(s[\uparrow]\, 0) \rhd s}$$

- Doesn’t require dependent types/very general-purpose
- Do not constraint the size of the contexts => sometimes required for syntactic translations

Well Scoped De Bruijn Syntax

- **Idea**: α-equivalence = definitional equality
- **Terms:**

```
Inductive tm n : Type :=
| var_tm : fin n -> tm n
| app : tm n -> tm n -> tm n
| lam : tm S n -> tm n.
```

- **Example term:** $\lambda\, x. z\ (\lambda\, y. (x\ y)\ z) \Rightarrow$
  $\lambda. (suc\ zero)\ (\lambda. ((suc\ zero)\ zero)\ (suc\ (suc\ zero)))$

```
lam (app (var_tm (suc zero)) (lam (app (app (var_tm (suc
zero)) (var_tm zero)) (var_tm (suc (suc zero)))))
```

Example: η-reduction rule for λ-terms

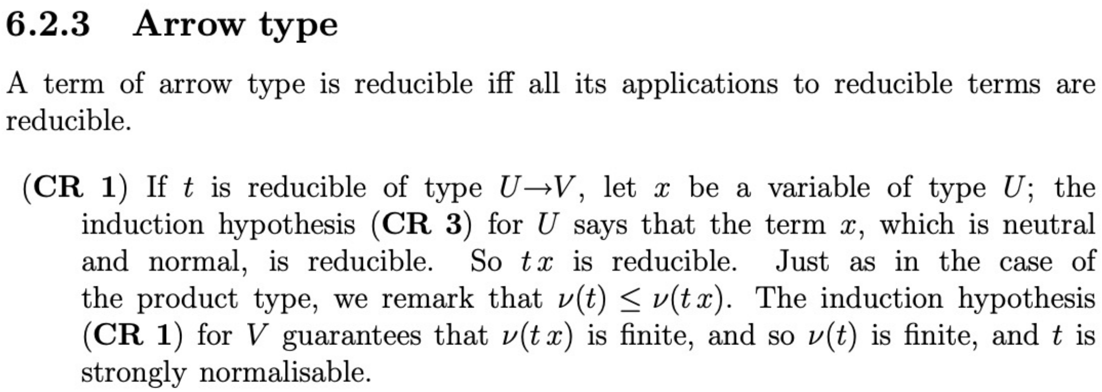
$$\frac{x \notin \mathrm{FV}\ s}{\lambda x.\ s\, x \rhd s} \qquad \frac{}{\lambda(s[\uparrow]\, 0) \rhd s}$$

Well scoped by construction – if
the shift is not included, an error is thrown

Adams, R.: Formalized metatheory with terms represented by an indexed family of types. In: Types for Proofs and Programs, Lecture Notes in Computer Science, vol. 3839, pp. 1–16. Springer Berlin Heidelberg (2006)

Bird, R., Paterson, R.: de Bruijn notation as a nested datatype. J. Funct. Program. **9**, 77–91 (1999)

Girard’s normalization proof for System F



6.2.3 Arrow type

A term of arrow type is reducible iff all its applications to reducible terms are reducible.

(**CR 1**) If t is reducible of type $U \rightarrow V$, let x be a variable of type U; the induction hypothesis (**CR 3**) for U says that the term x, which is neutral and normal, is reducible. So $t\,x$ is reducible. Just as in the case of the product type, we remark that $\nu(t) \leq \nu(t\,x)$. The induction hypothesis (**CR 1**) for V guarantees that $\nu(t\,x)$ is finite, and so $\nu(t)$ is finite, and t is strongly normalisable.

Implicitly uses ill-scoped terms
- what if in the empty context?

Monadic Terms

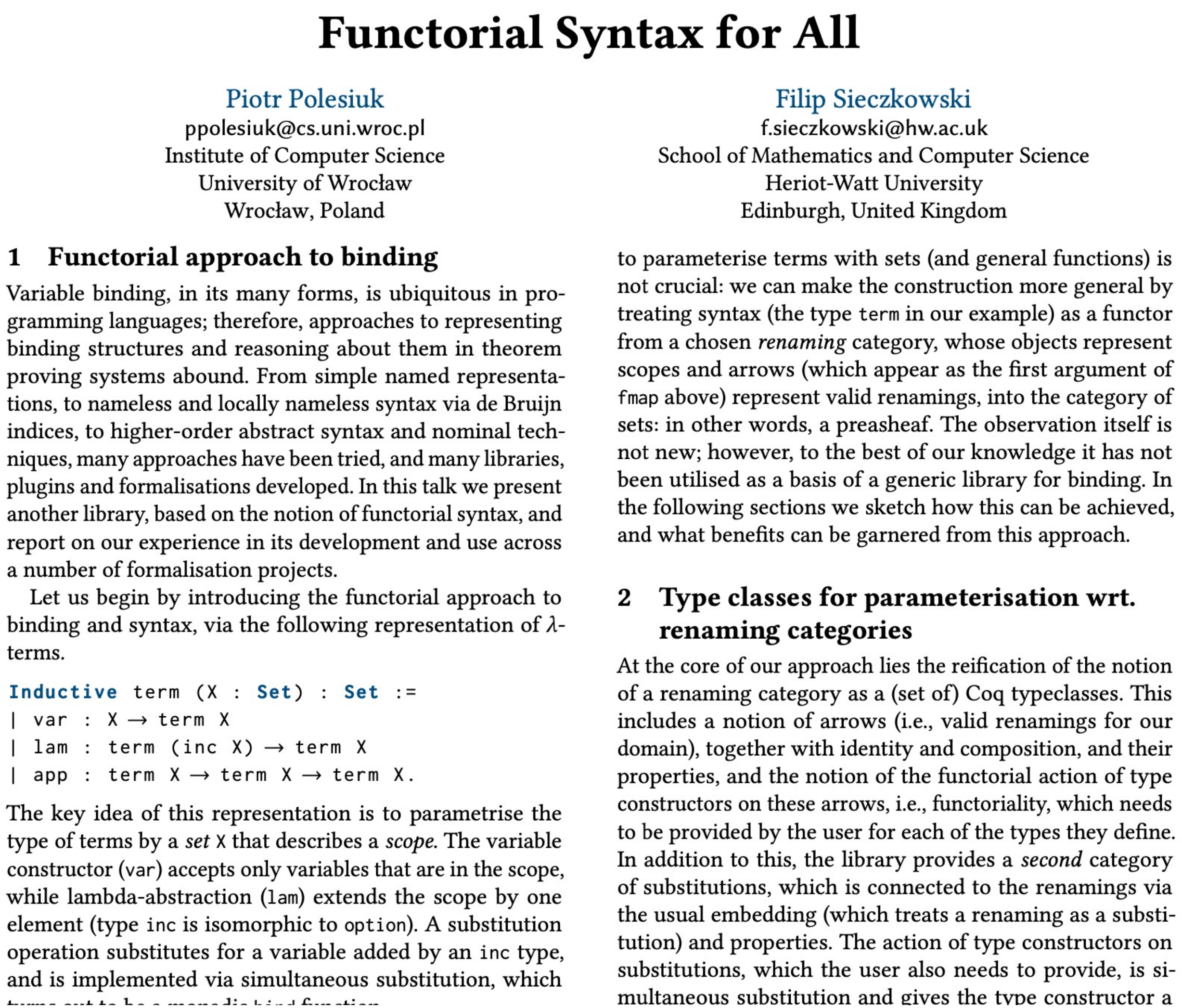
```
Inductive tm (X : Set) : Set :=
| var : X → tm X
| lam : tm (option X) → tm X
| app : tm X→ tm X→ tm X.
```

Example term: $\lambda\, x. x\ (\lambda\, y. x\ y) \Rightarrow$

```
Example ex (X : Set) : tm X :=
lam (app (var None) (lam (app (var (Some None)) (var None)))).


Fixpoint fmap {X Y : Set} (f : X → Y) (t : tm X) : tm Y.
```

Well-scoped terms can be obtained from monadic terms and vice versa in well-behaved examples: Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. „Monads Need Not Be Endofunctors“. In: **Logical Methods in Computer Science** 11.1 (Mar. 2015) (cit. on pp. 25, 166).

Discussion on a formalization for POPLMark Challenge: Hirschowitz/Maggesi: Nested Abstract Syntax in Coq. ‘09



Functorial Syntax for All

Piotr Polesiuk
ppolesiuk@cs.uni.wroc.pl
Institute of Computer Science
University of Wrocław
Wrocław, Poland

Filip Sieczkowski
f.sieczkowski@hw.ac.uk
School of Mathematics and Computer Science
Heriot-Watt University
Edinburgh, United Kingdom

1 Functorial approach to binding

Variable binding, in its many forms, is ubiquitous in programming languages; therefore, approaches to representing binding structures and reasoning about them in theorem proving systems abound. From simple named representations, to nameless and locally nameless syntax via de Bruijn indices, to higher-order abstract syntax and nominal techniques, many approaches have been tried, and many libraries, plugins and formalisations developed. In this talk we present another library, based on the notion of functorial syntax, and report on our experience in its development and use across a number of formalisation projects.

Let us begin by introducing the functorial approach to binding and syntax, via the following representation of λ-terms.

```
Inductive term (X : Set) : Set :=
| var : X → term X
| lam : term (inc X) → term X
| app : term X → term X → term X.
```

The key idea of this representation is to parametrise the type of terms by a *set* X that describes a *scope*. The variable constructor (var) accepts only variables that are in the scope, while lambda-abstraction (lam) extends the scope by one element (type inc is isomorphic to option). A substitution operation substitutes for a variable added by an inc type, and is implemented via simultaneous substitution, which turns out to be a monadic bind function.

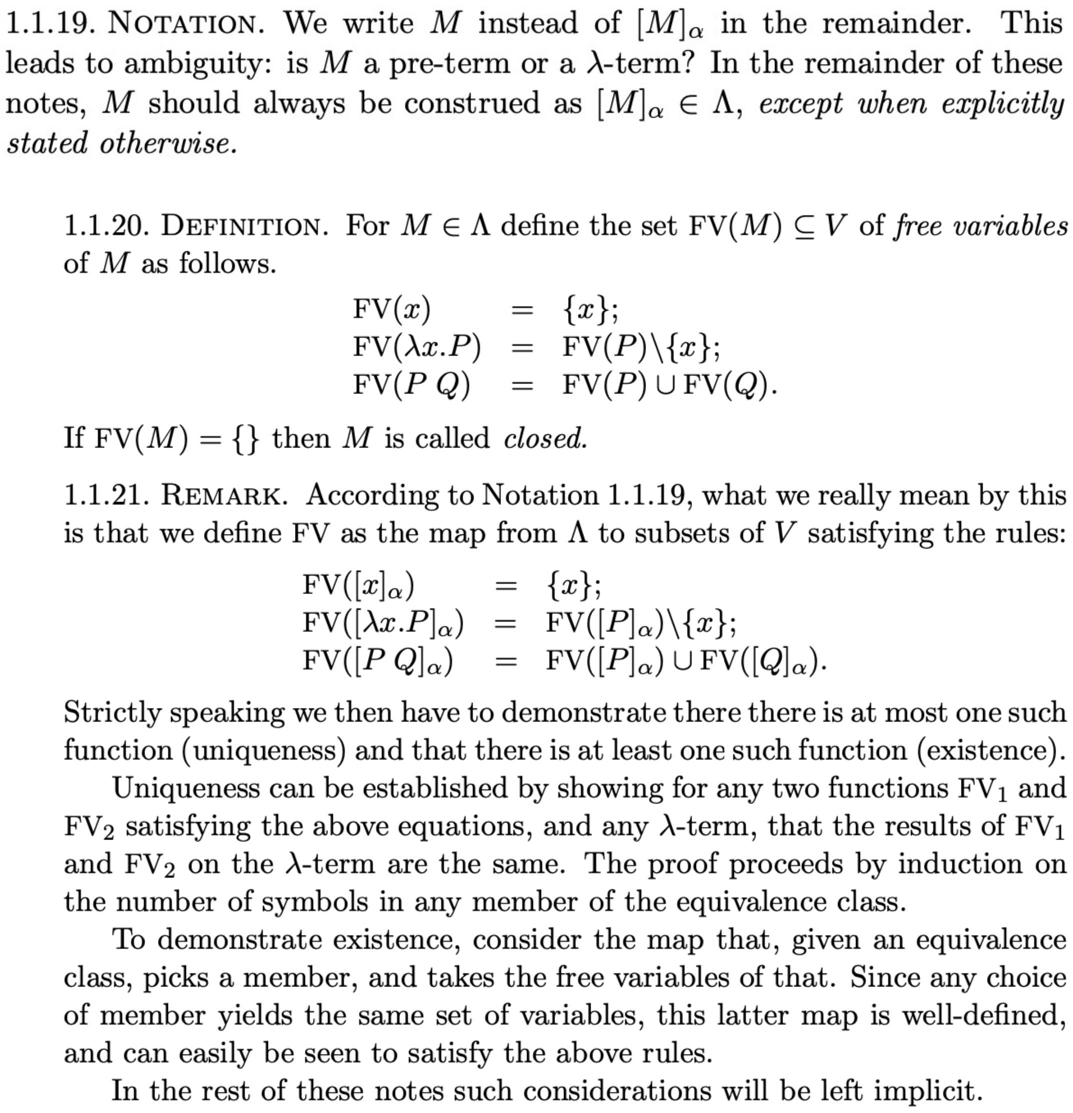
to parameterise terms with sets (and general functions) is not crucial: we can make the construction more general by treating syntax (the type term in our example) as a functor from a chosen *renaming* category, whose objects represent scopes and arrows (which appear as the first argument of fmap above) represent valid renamings, into the category of sets: in other words, a preasheaf. The observation itself is not new; however, to the best of our knowledge it has not been utilised as a basis of a generic library for binding. In the following sections we sketch how this can be achieved, and what benefits can be garnered from this approach.

2 Type classes for parameterisation wrt. renaming categories

At the core of our approach lies the reification of the notion of a renaming category as a (set of) Coq typeclasses. This includes a notion of arrows (i.e., valid renamings for our domain), together with identity and composition, and their properties, and the notion of the functorial action of type constructors on these arrows, i.e., functoriality, which needs to be provided by the user for each of the types they define. In addition to this, the library provides a *second* category of substitutions, which is connected to the renamings via the usual embedding (which treats a renaming as a substitution) and properties. The action of type constructors on substitutions, which the user also needs to provide, is simultaneous substitution and gives the type constructor a

CoqPL'24

Full version in progress



1.1.19. Notation. We write M instead of $[M]_\alpha$ in the remainder. This leads to ambiguity: is M a pre-term or a λ-term? In the remainder of these notes, M should always be construed as $[M]_\alpha \in \Lambda$, *except when explicitly stated otherwise.*

1.1.20. Definition. For $M \in \Lambda$ define the set $\mathrm{FV}(M) \subseteq V$ of *free variables* of M as follows.

$$\begin{array}{lcl} \mathrm{FV}(x) & = & \{x\}; \\ \mathrm{FV}(\lambda x.P) & = & \mathrm{FV}(P)\backslash\{x\}; \\ \mathrm{FV}(P\ Q) & = & \mathrm{FV}(P) \cup \mathrm{FV}(Q). \end{array}$$

If $\mathrm{FV}(M) = \{\}$ then M is called *closed.*

1.1.21. Remark. According to Notation 1.1.19, what we really mean by this is that we define FV as the map from Λ to subsets of V satisfying the rules:

$$\begin{array}{lcl} \mathrm{FV}([x]_\alpha) & = & \{x\}; \\ \mathrm{FV}([\lambda x.P]_\alpha) & = & \mathrm{FV}([P]_\alpha)\backslash\{x\}; \\ \mathrm{FV}([P\ Q]_\alpha) & = & \mathrm{FV}([P]_\alpha) \cup \mathrm{FV}([Q]_\alpha). \end{array}$$

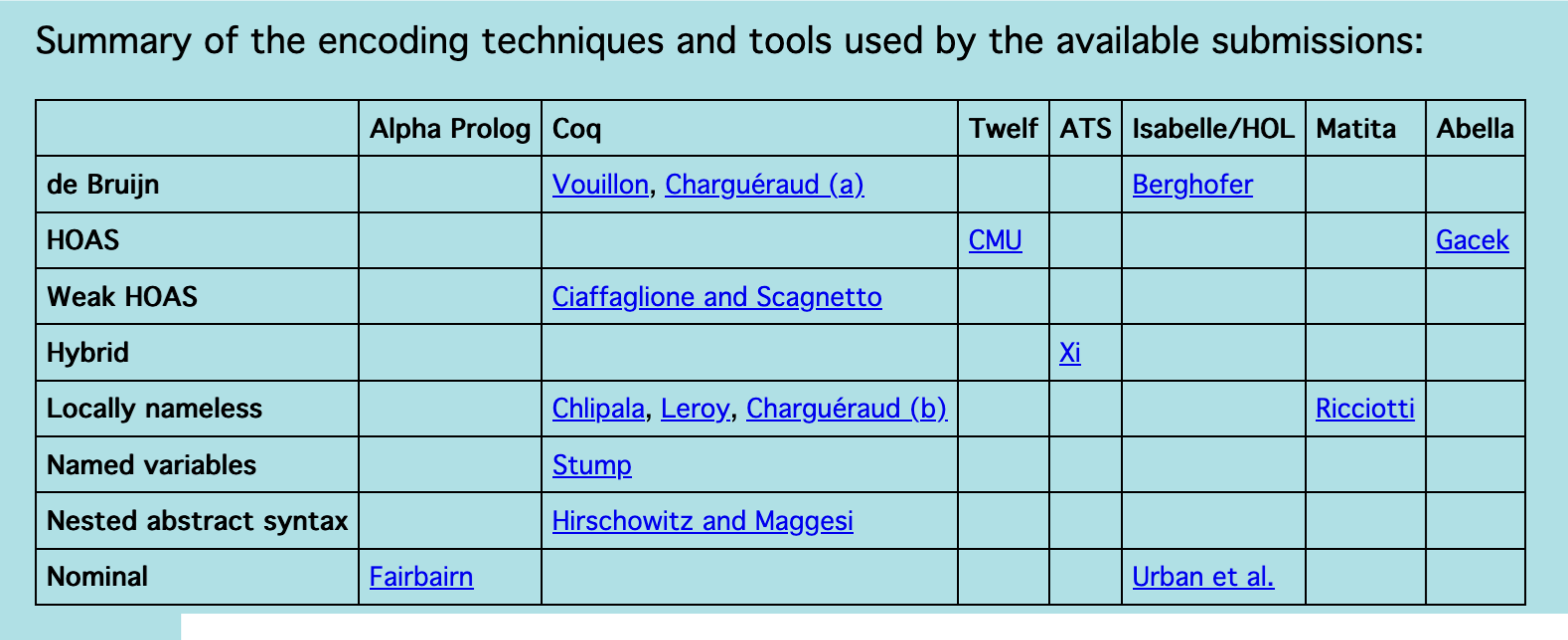
Strictly speaking we then have to demonstrate there there is at most one such function (uniqueness) and that there is at least one such function (existence).

Uniqueness can be established by showing for any two functions FV_1 and FV_2 satisfying the above equations, and any λ-term, that the results of FV_1 and FV_2 on the λ-term are the same. The proof proceeds by induction on the number of symbols in any member of the equivalence class.

To demonstrate existence, consider the map that, given an equivalence class, picks a member, and takes the free variables of that. Since any choice of member yields the same set of variables, this latter map is well-defined, and can easily be seen to satisfy the above rules.

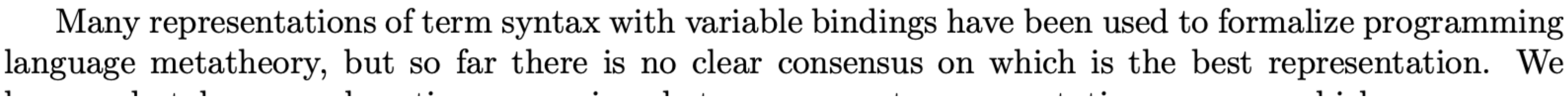
In the rest of these notes such considerations will be left implicit.

… and definitions on terms.

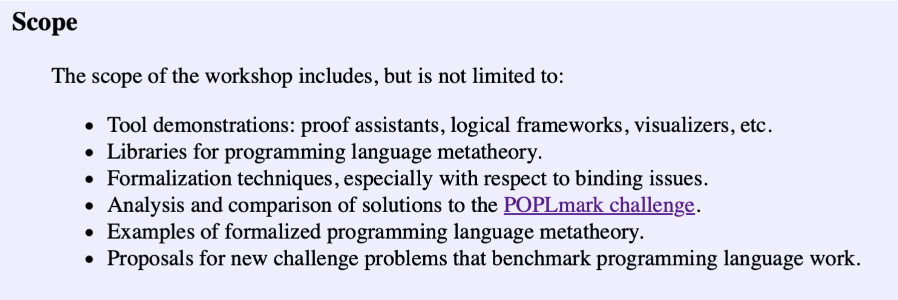


Summary of the encoding techniques and tools used by the available submissions:

	Alpha Prolog	Coq	Twelf	ATS	Isabelle/HOL	Matita	Abella
de Bruijn		Vouillon, Charguéraud (a)			Berghofer		
HOAS			CMU				Gacek
Weak HOAS		Ciaffaglione and Scagnetto					
Hybrid				Xi			
Locally nameless		Chlipala, Leroy, Charguéraud (b)				Ricciotti	
Named variables		Stump					
Nested abstract syntax		Hirschowitz and Maggesi					
Nominal	Fairbairn				Urban et al.		



Many representations of term syntax with variable bindings have been used to formalize programming language metatheory, but so far there is no clear consensus on which is the best representation. We



Scope

The scope of the workshop includes, but is not limited to:

- Tool demonstrations: proof assistants, logical frameworks, visualizers, etc.
- Libraries for programming language metatheory.
- Formalization techniques, especially with respect to binding issues.
- Analysis and comparison of solutions to the POPLmark challenge.
- Examples of formalized programming language metatheory.
- Proposals for new challenge problems that benchmark programming language work.

De Bruijn Syntax

- **Idea**: α-equivalence = definitional equality
- **Terms:**

```
Inductive tm: Type :=
| var_tm : nat -> tm
| app : tm -> tm -> tm
| lam : tm -> tm.
```

- **Example term:** $\lambda\, x. z\ (\lambda\, y. (x\ y)\ z) \Rightarrow \lambda. 1\ (\lambda. (1\ 0)\ 2)$

```
lam (app (var_tm 1) (lam (app (app (var_tm 1) (var_tm 0))
(var_tm 2))))
```

- Doesn't require dependent types/very general-purpose
- Do not constraint the size of the contexts => sometimes required for syntactic translations

Example: η-reduction rule for λ-terms

$$\frac{x \notin \mathrm{FV}\ s}{\lambda x.\ s\, x \rhd s} \qquad \frac{}{\lambda(s[\uparrow]\, 0) \rhd s}$$

(Parallel) Substitutions

de Bruijn ‘72

$$(s \cdot \sigma)(x) := \begin{cases} s & \text{if } x = 0 \\ \sigma(x-1) & \text{otherwise} \end{cases}$$

$$\mathsf{id}(x) := x \qquad\qquad \uparrow(x) := x + 1$$

$$\begin{aligned} x[\sigma] &= \sigma(x) \\ (s\,t)[\sigma] &= (s[\sigma])\,(t[\sigma]) \qquad (\sigma \circ \tau)(x) = \sigma(x)[\tau] \\ (\lambda.\, s)[\sigma] &= \lambda.\, (s[\Uparrow\sigma]) \end{aligned}$$

$$\left.\Uparrow\sigma := 0 \cdot (\sigma \circ \uparrow) \quad \right\} \text{Requires again substitution}$$

Two-Level Approach: Adams, R.: Formalized metatheory with terms represented by an indexed family of types. In: Types for Proofs and Programs, Lecture Notes in Computer Science, vol. 3839, pp. 1–16. Springer Berlin Heidelberg (2006)

Well Scoped De Bruijn Syntax

- **Idea**: α-equivalence = definitional equality
- **Terms:**

```
Inductive tm n : Type :=
| var_tm : fin n -> tm n
| app : tm n -> tm n -> tm n
| lam : tm S n -> tm n.
```

- **Example term:** $\lambda\, x. z\ (\lambda\, y. (x\ y)\ z) \Rightarrow$
$\lambda. (suc\ zero)\ (\lambda. ((suc\ zero)\ zero)\ (suc\ (suc\ zero)))$

```
lam (app (var_tm (suc zero)) (lam (app (app (var_tm (suc
zero)) (var_tm zero)) (var_tm (suc (suc zero)))))
```

Example: η-reduction rule for λ-terms

$$\frac{x \notin \mathrm{FV}\ s}{\lambda x.\ s\, x \rhd s} \qquad \frac{}{\lambda(s[\uparrow]\,0) \rhd s}$$

Well scoped by construction – if the shift is not included, an error is thrown

Adams, R.: Formalized metatheory with terms represented by an indexed family of types. In: Types for Proofs and Programs, Lecture Notes in Computer Science, vol. 3839, pp. 1–16. Springer Berlin Heidelberg (2006)

Bird, R., Paterson, R.: de Bruijn notation as a nested datatype. J. Funct. Program. **9**, 77–91 (1999)

Monadic Terms

```
Inductive tm (X : Set) : Set :=
| var : X → tm X
| lam : tm (option X) → tm X
| app : tm X→ tm X→ tm X.
```

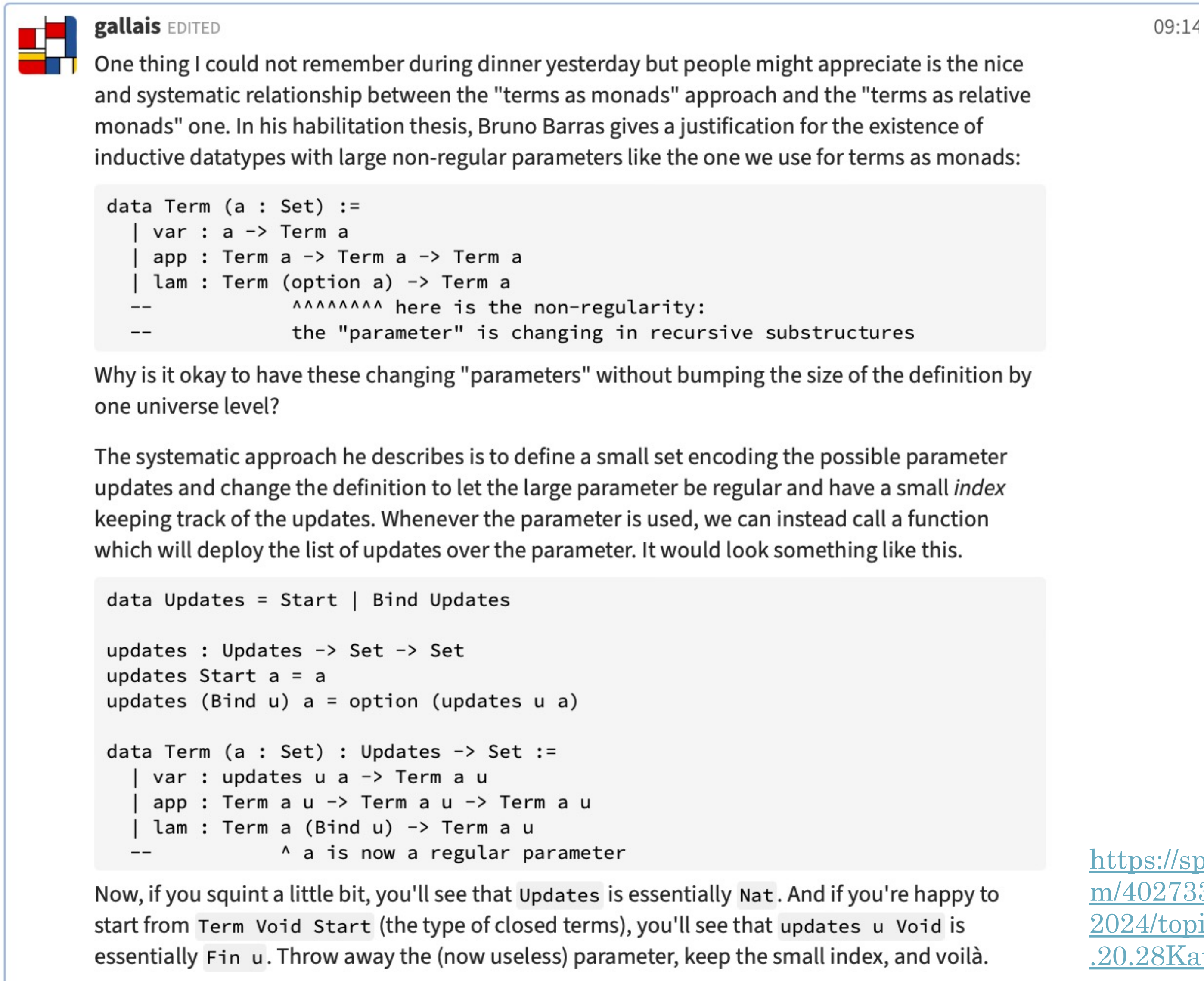
Example term: $\lambda\, x. x\ (\lambda\, y. x\ y) \Rightarrow$

```
Example ex (X : Set) : tm X :=
lam (app (var None) (lam (app (var (Some None)) (var None)))).
```

```
Fixpoint fmap {X Y : Set} (f : X → Y) (t : tm X) : tm Y.
```

Well-scoped terms can be obtained from monadic terms and vice versa in well-behaved examples: Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. „Monads Need Not Be Endofunctors“. In: **Logical Methods in Computer Science** 11.1 (Mar. 2015) (cit. on pp. 25, 166).

Discussion on a formalization for POPLMark Challenge: Hirschowitz/Maggesi: Nested Abstract Syntax in Coq. ‘09



gallais EDITED 09:14

One thing I could not remember during dinner yesterday but people might appreciate is the nice and systematic relationship between the "terms as monads" approach and the "terms as relative monads" one. In his habilitation thesis, Bruno Barras gives a justification for the existence of inductive datatypes with large non-regular parameters like the one we use for terms as monads:

```
data Term (a : Set) :=
  | var : a -> Term a
  | app : Term a -> Term a -> Term a
  | lam : Term (option a) -> Term a
  --            ^^^^^^^^ here is the non-regularity:
  --            the "parameter" is changing in recursive substructures
```

Why is it okay to have these changing "parameters" without bumping the size of the definition by one universe level?

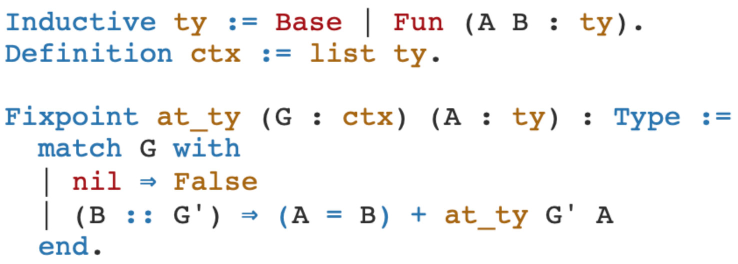
The systematic approach he describes is to define a small set encoding the possible parameter updates and change the definition to let the large parameter be regular and have a small *index* keeping track of the updates. Whenever the parameter is used, we can instead call a function which will deploy the list of updates over the parameter. It would look something like this.

```
data Updates = Start | Bind Updates

updates : Updates -> Set -> Set
updates Start a = a
updates (Bind u) a = option (updates u a)

data Term (a : Set) : Updates -> Set :=
  | var : updates u a -> Term a u
  | app : Term a u -> Term a u -> Term a u
  | lam : Term a (Bind u) -> Term a u
  --           ^ a is now a regular parameter
```

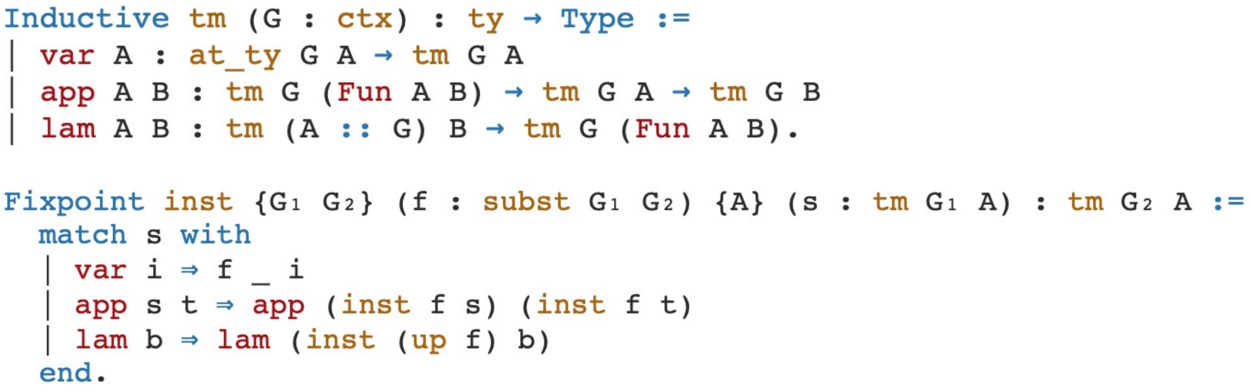
Now, if you squint a little bit, you'll see that `Updates` is essentially `Nat`. And if you're happy to start from `Term Void Start` (the type of closed terms), you'll see that `updates u Void` is essentially `Fin u`. Throw away the (now useless) parameter, keep the small index, and voilà.

https://spls.zulipchat.com/#narrow/stream/402733-splv-2024/topic/Mechanization.20of.20Binders.20.28Kathrin.20Stark.29

Intrinsically Typed Syntax



```
Inductive ty := Base | Fun (A B : ty).
Definition ctx := list ty.

Fixpoint at_ty (G : ctx) (A : ty) : Type :=
  match G with
  | nil ⇒ False
  | (B :: G') ⇒ (A = B) + at_ty G' A
  end.
```

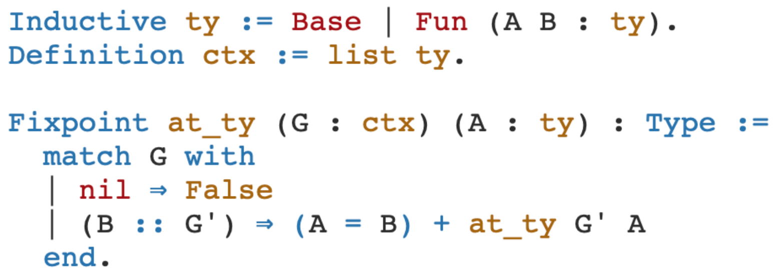


```
Inductive tm (G : ctx) : ty → Type :=
| var A : at_ty G A → tm G A
| app A B : tm G (Fun A B) → tm G A → tm G B
| lam A B : tm (A :: G) B → tm G (Fun A B).

Fixpoint inst {G₁ G₂} (f : subst G₁ G₂) {A} (s : tm G₁ A) : tm G₂ A :=
  match s with
  | var i ⇒ f _ i
  | app s t ⇒ app (inst f s) (inst f t)
  | lam b ⇒ lam (inst (up f) b)
  end.
```

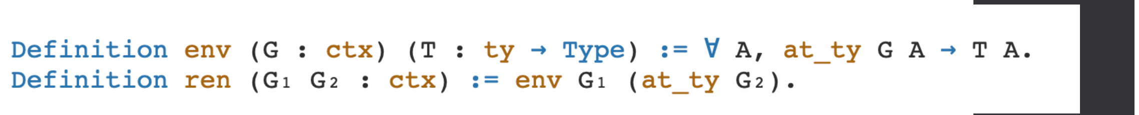
Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. ‘99
Benton, Nick, et al. "Strongly typed term representations in Coq." 2012

Types of Renamings/Substitutions

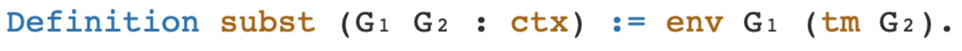
- Pure:
  - Renamings: `nat -> nat`
  - Substitutions: `nat -> tm`
- Scoped:
  - Renamings: `fin m -> fin n`
  - Substitutions: `fin m -> tm n`
- Intrinsically Typed:
  - Renamings:
  - Substitutions:



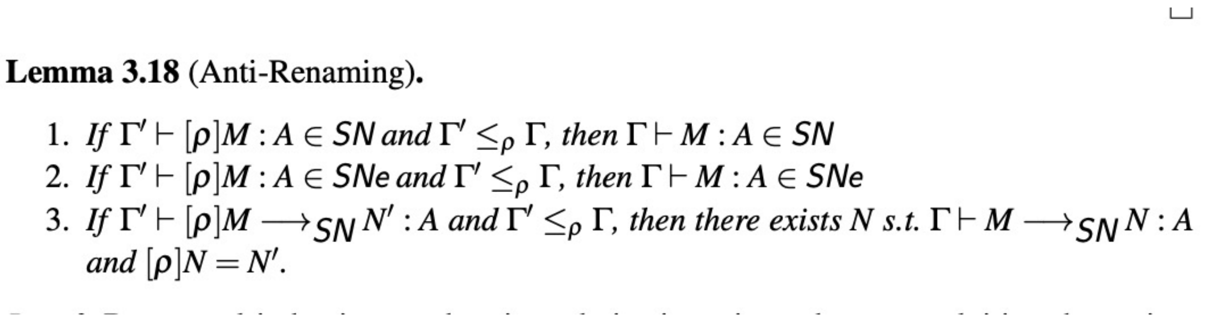
```
Inductive ty := Base | Fun (A B : ty).
Definition ctx := list ty.

Fixpoint at_ty (G : ctx) (A : ty) : Type :=
  match G with
  | nil ⇒ False
  | (B :: G') ⇒ (A = B) + at_ty G' A
  end.
```



```
Definition env (G : ctx) (T : ty → Type) := ∀ A, at_ty G A → T A.
Definition ren (G₁ G₂ : ctx) := env G₁ (at_ty G₂).
```

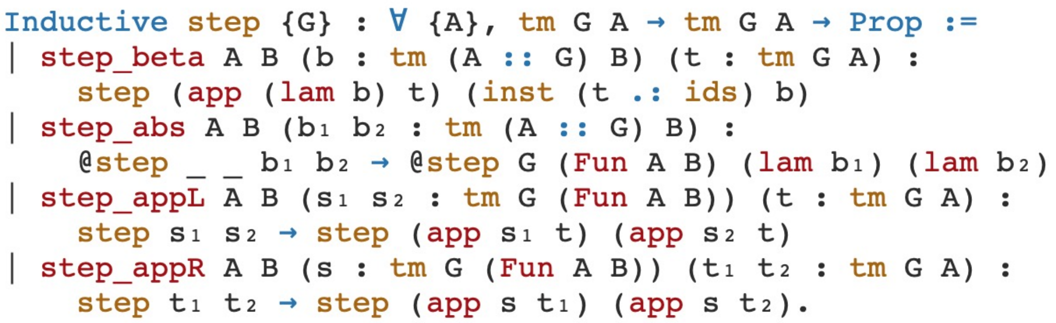


```
Definition subst (G₁ G₂ : ctx) := env G₁ (tm G₂).
```



Lemma 3.18 (Anti-Renaming).

1. *If* $\Gamma' \vdash [\rho]M : A \in \mathit{SN}$ *and* $\Gamma' \leq_\rho \Gamma$, *then* $\Gamma \vdash M : A \in \mathit{SN}$
2. *If* $\Gamma' \vdash [\rho]M : A \in \mathit{SNe}$ *and* $\Gamma' \leq_\rho \Gamma$, *then* $\Gamma \vdash M : A \in \mathit{SNe}$
3. *If* $\Gamma' \vdash [\rho]M \longrightarrow_{\mathit{SN}} N' : A$ *and* $\Gamma' \leq_\rho \Gamma$, *then there exists* N *s.t.* $\Gamma \vdash M \longrightarrow_{\mathit{SN}} N : A$ *and* $[\rho]N = N'$.



```
Inductive step {G} : ∀ {A}, tm G A → tm G A → Prop :=
| step_beta A B (b : tm (A :: G) B) (t : tm G A) :
    step (app (lam b) t) (inst (t .: ids) b)
| step_abs A B (b₁ b₂ : tm (A :: G) B) :
    @step _ _ b₁ b₂ → @step G (Fun A B) (lam b₁) (lam b₂)
| step_appL A B (s₁ s₂ : tm G (Fun A B)) (t : tm G A) :
    step s₁ s₂ → step (app s₁ t) (app s₂ t)
| step_appR A B (s : tm G (Fun A B)) (t₁ t₂ : tm G A) :
    step t₁ t₂ → step (app s t₁) (app s t₂).
```

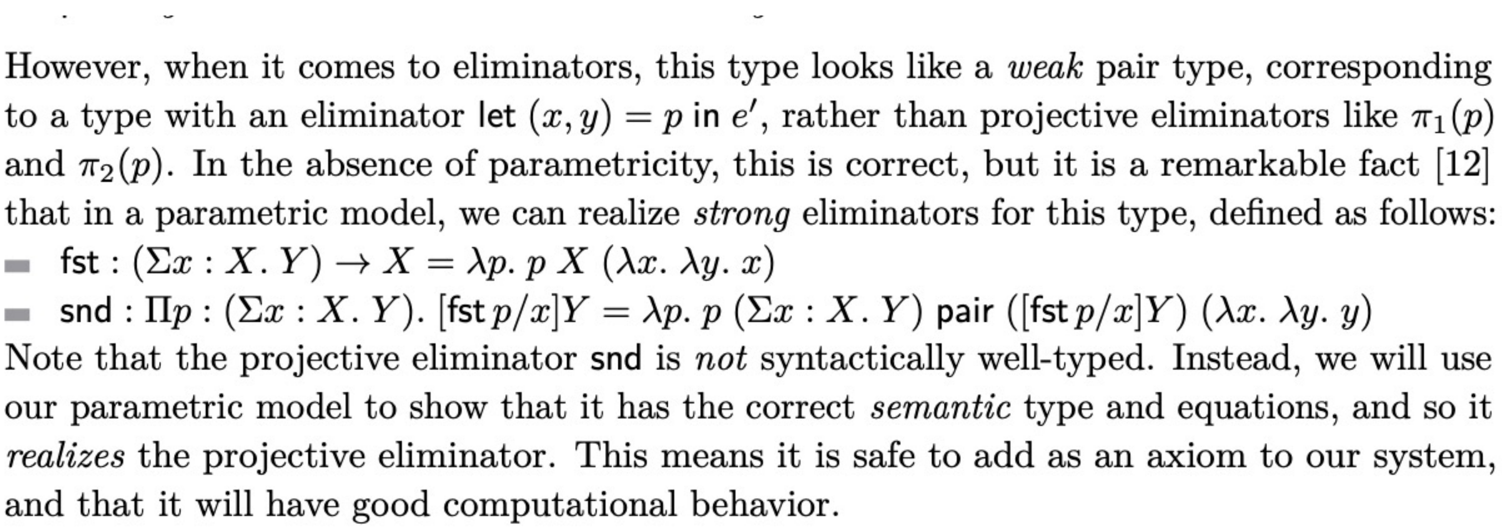
Dependently typed constructor
for type abstraction in System F:

$$\Lambda\_ \ : \ \underbrace{\mathbb{E}\,(\Gamma \circ \uparrow)}_{\text{Only applicable to arguments of contexts of exactly this form}}\, A \to \mathbb{E}\,\Gamma\,(\forall\, A)$$

$\Gamma \circ (\uparrow \circ \uparrow) = (\Gamma \circ \uparrow) \circ \uparrow$
propositionally but not definitionally

When moving to dependent types, we need inductive-inductive types to represent well-typed terms of a dependently typed language:

Thorsten Altenkirch and Ambrus Kaposi. „Type Theory in Type Theory Using Quotient Inductive Types“. In: **ACM SIGPLAN Notices** 51.1 (Jan. 2016), pp. 18–29 (cit. on pp. 42, 166).



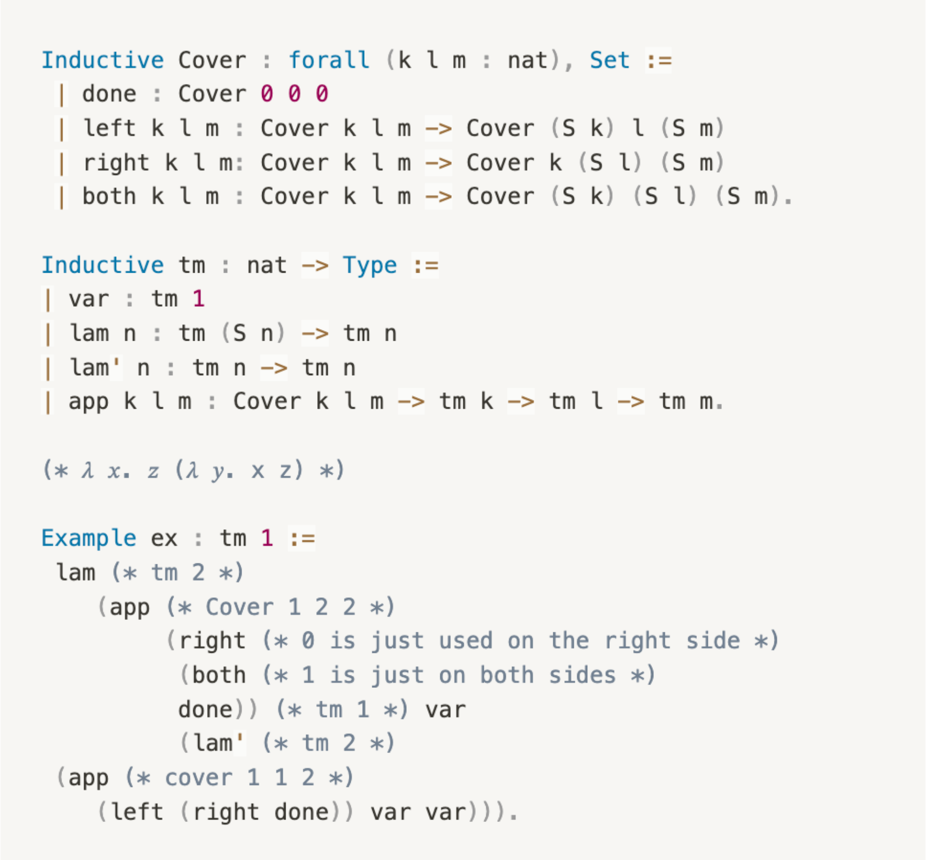
However, when it comes to eliminators, this type looks like a *weak* pair type, corresponding to a type with an eliminator $\mathsf{let}\ (x, y) = p\ \mathsf{in}\ e'$, rather than projective eliminators like $\pi_1(p)$ and $\pi_2(p)$. In the absence of parametricity, this is correct, but it is a remarkable fact [12] that in a parametric model, we can realize *strong* eliminators for this type, defined as follows:

- $\mathsf{fst} : (\Sigma x : X.\, Y) \to X = \lambda p.\ p\ X\ (\lambda x.\ \lambda y.\ x)$
- $\mathsf{snd} : \Pi p : (\Sigma x : X.\, Y).\ [\mathsf{fst}\, p/x]Y = \lambda p.\ p\ (\Sigma x : X.\, Y)\ \mathsf{pair}\ ([\mathsf{fst}\, p/x]Y)\ (\lambda x.\ \lambda y.\ y)$

Note that the projective eliminator snd is *not* syntactically well-typed. Instead, we will use our parametric model to show that it has the correct *semantic* type and equations, and so it *realizes* the projective eliminator. This means it is safe to add as an axiom to our system, and that it will have good computational behavior.

Krishnaswami, Neelakantan R., and Derek Dreyer. "Internalizing relational parametricity in the extensional calculus of constructions." *Computer Science Logic 2013 (CSL 2013)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013.

Co-De Bruijn Representation

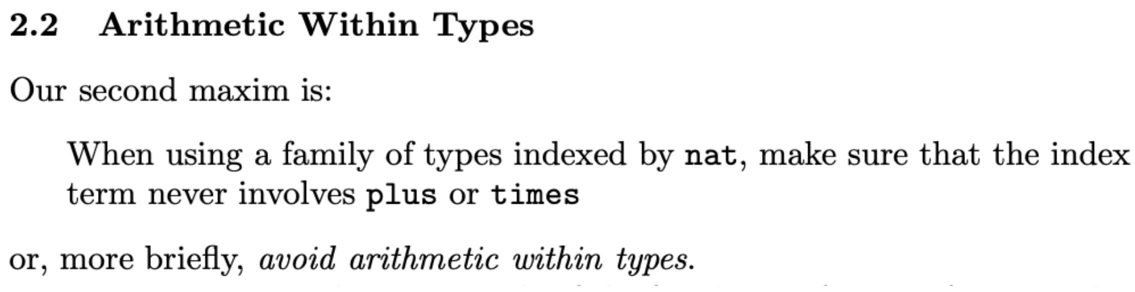


```
Inductive Cover : forall (k l m : nat), Set :=
 | done : Cover 0 0 0
 | left k l m : Cover k l m -> Cover (S k) l (S m)
 | right k l m: Cover k l m -> Cover k (S l) (S m)
 | both k l m : Cover k l m -> Cover (S k) (S l) (S m).

Inductive tm : nat -> Type :=
| var : tm 1
| lam n : tm (S n) -> tm n
| lam' n : tm n -> tm n
| app k l m : Cover k l m -> tm k -> tm l -> tm m.

(* λ x. z (λ y. x z) *)

Example ex : tm 1 :=
 lam (* tm 2 *)
    (app (* Cover 1 2 2 *)
         (right (* 0 is just used on the right side *)
          (both (* 1 is just on both sides *)
          done)) (* tm 1 *) var
          (lam' (* tm 2 *)
 (app (* cover 1 1 2 *)
    (left (right done)) var var))).
```

Everybody's Got To Be Somewhere, Conor McBride 2018
Stripped version by Jesper Cockx, https://jesper.sikanda.be/posts/1001-syntax-representations.html

Notes

Remember that we work with an encoding:



2.2 Arithmetic Within Types

Our second maxim is:

> When using a family of types indexed by **`nat`**, make sure that the index term never involves **`plus`** or **`times`**

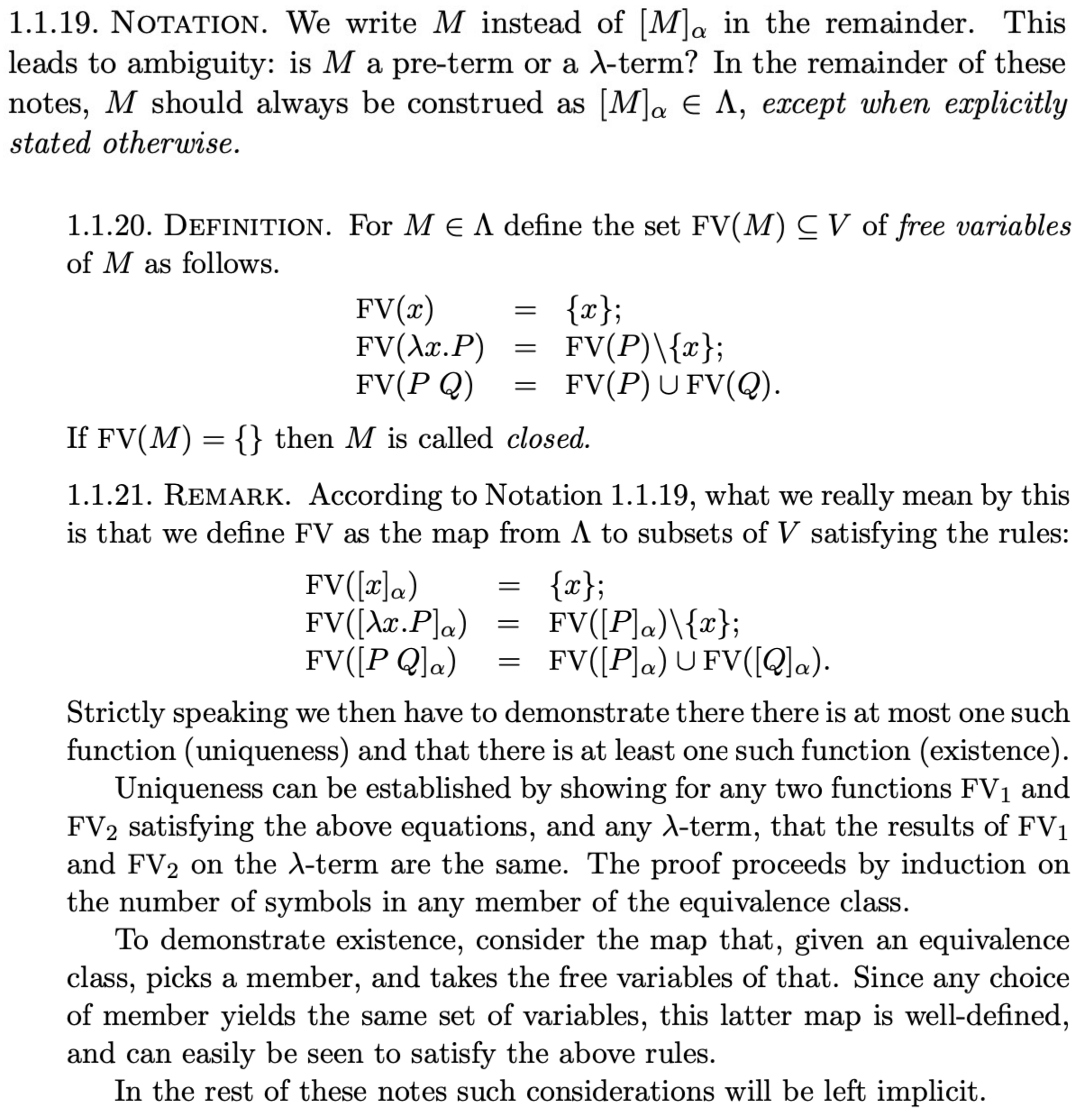
or, more briefly, *avoid arithmetic within types.*

Adams, R.: Formalized metatheory with terms represented by an indexed family of types. In: Types for Proofs and Programs (2006)

Nominal Syntax and (variants of) HOAS

Kathrin Stark

SPLV 2024



1.1.19. Notation. We write M instead of $[M]_\alpha$ in the remainder. This leads to ambiguity: is M a pre-term or a λ-term? In the remainder of these notes, M should always be construed as $[M]_\alpha \in \Lambda$, *except when explicitly stated otherwise.*

1.1.20. Definition. For $M \in \Lambda$ define the set $\text{FV}(M) \subseteq V$ of *free variables* of M as follows.

$$\begin{array}{lcl} \text{FV}(x) & = & \{x\}; \\ \text{FV}(\lambda x.P) & = & \text{FV}(P)\backslash\{x\}; \\ \text{FV}(P\,Q) & = & \text{FV}(P) \cup \text{FV}(Q). \end{array}$$

If $\text{FV}(M) = \{\}$ then M is called *closed.*

1.1.21. Remark. According to Notation 1.1.19, what we really mean by this is that we define FV as the map from Λ to subsets of V satisfying the rules:

$$\begin{array}{lcl} \text{FV}([x]_\alpha) & = & \{x\}; \\ \text{FV}([\lambda x.P]_\alpha) & = & \text{FV}([P]_\alpha)\backslash\{x\}; \\ \text{FV}([P\,Q]_\alpha) & = & \text{FV}([P]_\alpha) \cup \text{FV}([Q]_\alpha). \end{array}$$

Strictly speaking we then have to demonstrate there there is at most one such function (uniqueness) and that there is at least one such function (existence).

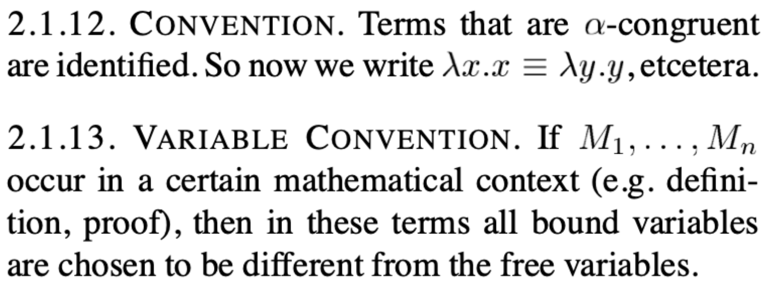
Uniqueness can be established by showing for any two functions FV_1 and FV_2 satisfying the above equations, and any λ-term, that the results of FV_1 and FV_2 on the λ-term are the same. The proof proceeds by induction on the number of symbols in any member of the equivalence class.

To demonstrate existence, consider the map that, given an equivalence class, picks a member, and takes the free variables of that. Since any choice of member yields the same set of variables, this latter map is well-defined, and can easily be seen to satisfy the above rules.

In the rest of these notes such considerations will be left implicit.

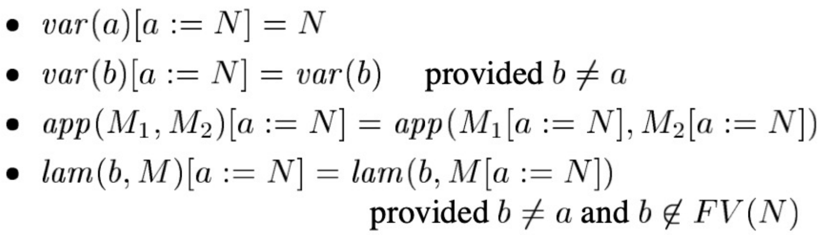
... and definitions on terms.

Barendregt Convention

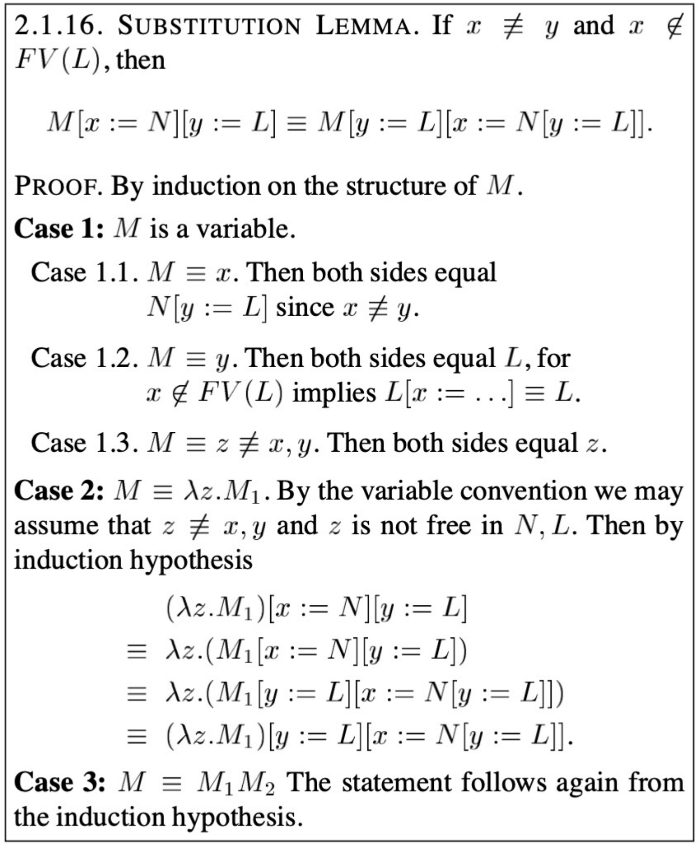


2.1.12. Convention. Terms that are α-congruent are identified. So now we write $\lambda x.x \equiv \lambda y.y$, etcetera.

2.1.13. Variable Convention. If $M_1, \ldots, M_n$ occur in a certain mathematical context (e.g. definition, proof), then in these terms all bound variables are chosen to be different from the free variables.



- $var(a)[a := N] = N$
- $var(b)[a := N] = var(b)$ provided $b \neq a$
- $app(M_1, M_2)[a := N] = app(M_1[a := N], M_2[a := N])$
- $lam(b, M)[a := N] = lam(b, M[a := N])$ provided $b \neq a$ and $b \notin FV(N)$



2.1.16. Substitution Lemma. If $x \not\equiv y$ and $x \notin FV(L)$, then

$$M[x := N][y := L] \equiv M[y := L][x := N[y := L]].$$

Proof. By induction on the structure of M.

Case 1: M is a variable.

Case 1.1. $M \equiv x$. Then both sides equal $N[y := L]$ since $x \not\equiv y$.

Case 1.2. $M \equiv y$. Then both sides equal L, for $x \notin FV(L)$ implies $L[x := \ldots] \equiv L$.

Case 1.3. $M \equiv z \not\equiv x, y$. Then both sides equal z.

Case 2: $M \equiv \lambda z.M_1$. By the variable convention we may assume that $z \not\equiv x, y$ and z is not free in N, L. Then by induction hypothesis

$$\begin{aligned} & (\lambda z.M_1)[x := N][y := L] \\ \equiv\; & \lambda z.(M_1[x := N][y := L]) \\ \equiv\; & \lambda z.(M_1[y := L][x := N[y := L]]) \\ \equiv\; & (\lambda z.M_1)[y := L][x := N[y := L]]. \end{aligned}$$

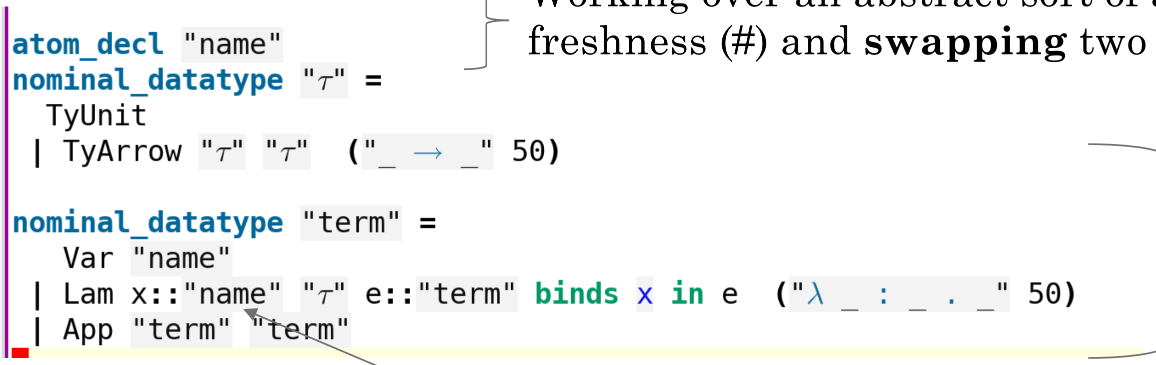
Case 3: $M \equiv M_1 M_2$ The statement follows again from the induction hypothesis.

Figure 1. Barendregt's proof of the Substitution Lemma

Examples from: Urban, Norrish: A Formal Treatment of the Barendregt Variable Convention

Equivariance:
A definition commutes with permutation.

Nominal Techniques



```
atom_decl "name"
nominal_datatype "τ" =
  TyUnit
 | TyArrow "τ" "τ"  ("_ → _" 50)

nominal_datatype "term" =
   Var "name"
 | Lam x::"name" "τ" e::"term" binds x in e  ("λ _ : _ . _" 50)
 | App "term" "term"
```

Working over an abstract sort of **atoms** - allowing freshness (#) and **swapping** two atoms

Ensures that the datatype respects equivariance

Abstracting over a new name

$$supp\ x \overset{def}{=} \{a \mid infinite\ \{b \mid (a\ b) \bullet x \neq x\}\}$$

A notion of support:
A finite set of variables that the definition may contain

There is also the derived notion for when an atom a is *fresh* for an x, defined as

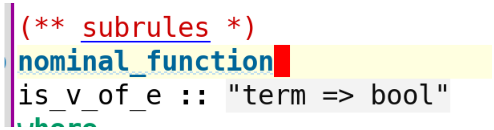
$$a \# x \overset{def}{=} a \notin supp\ x$$

Nominal Logic, a first-oder theory of names and binders – Pitts 2003.
Nominal Unification, Urban, Pitts, Gabbay – 2004.
Nominal Techniques in Isabelle/HOL. Urban/Tasson 2005.
General Bindings and Alpha-Equivalence in Nominal Isabelle, Urban/Kaliszyk ‘12

Nominal Techniques

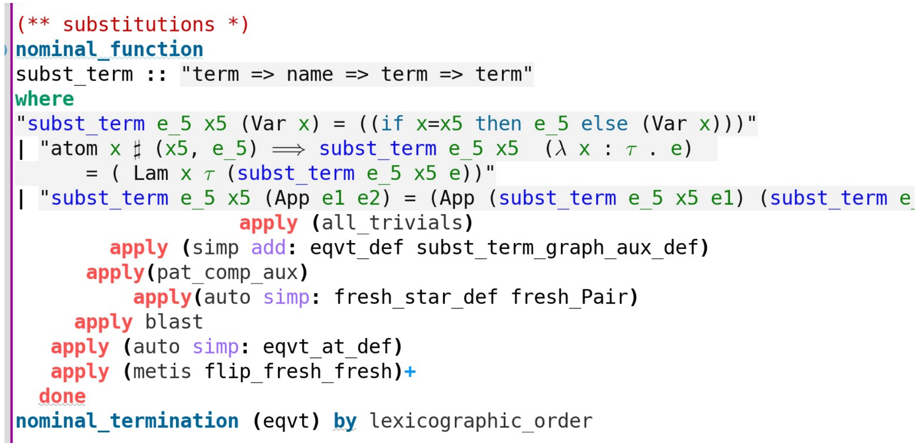
Equivariance:
A definition commutes with permutation.

Nominal definitions are shown to be equivariant:



```
(** subrules *)
nominal_function
is_v_of_e :: "term => bool"
where
```

General renaming/instantiation have to be defined:



```
(** substitutions *)
nominal_function
subst_term :: "term => name => term => term"
where
"subst_term e_5 x5 (Var x) = ((if x=x5 then e_5 else (Var x)))"
| "atom x ♯ (x5, e_5) ⟹ subst_term e_5 x5  (λ x : τ . e)
      = ( Lam x τ (subst_term e_5 x5 e))"
| "subst_term e_5 x5 (App e1 e2) = (App (subst_term e_5 x5 e1) (subst_term e
                     apply (all_trivials)
          apply (simp add: eqvt_def subst_term_graph_aux_def)
        apply(pat_comp_aux)
            apply(auto simp: fresh_star_def fresh_Pair)
       apply blast
     apply (auto simp: eqvt_at_def)
     apply (metis flip_fresh_fresh)+
   done
nominal_termination (eqvt) by lexicographic_order
```

Nominal Unification, Urban, Pitts, Gabbay – 2004.
Nominal Techniques in Isabelle/HOL. Urban/Tasson 2005.

Freshness of side conditions:
The Nominal library supports proving those/automatically derives reasoning infrastructure

- Compatible with classical reasoning

Used by

MiniSail - A kernel language for the ISA specification language SAIL

From Abstract to Concrete Gödel’s Incompleteness Theorems—Part II **Robinson Arithmetic**

Formalization of Generic Authenticated Data Structures

Modal Logics for Nominal Transition Systems **The Z Property** **Gödel’s Incompleteness Theorems**

The Correctness of Launchbury’s Natural Semantics for Lazy Evaluation

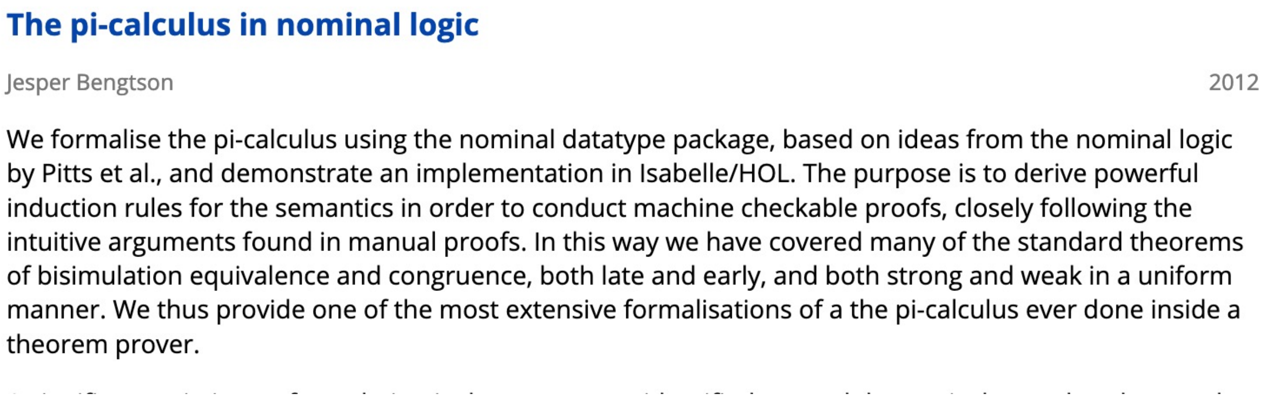
CCS in nominal logic

Jesper Bengtson 2012

We formalise a large portion of CCS as described in Milner's book 'Communication and Concurrency' using the nominal datatype package in Isabelle. Our results include many of the standard theorems of bisimulation equivalence and congruence, for both weak and strong versions. One main goal of this formalisation is to keep the machine-checked proofs as close to their pen-and-paper counterpart as possible.

This entry is described in detail in **Bengtson's thesis**.

in **Computer science/Concurrency/Process calculi**



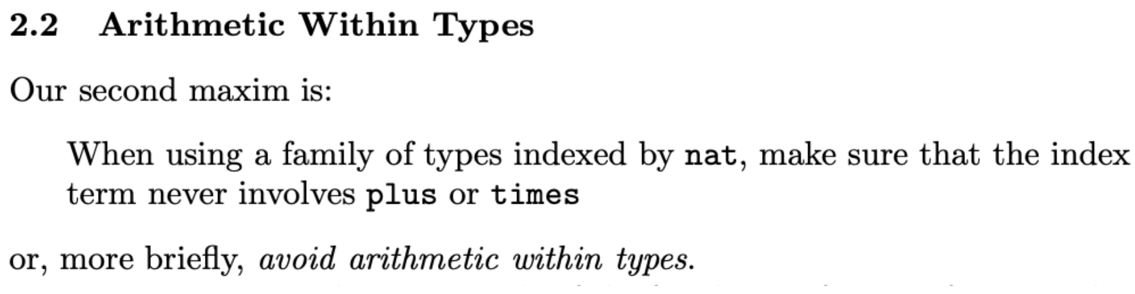
The pi-calculus in nominal logic

Jesper Bengtson 2012

We formalise the pi-calculus using the nominal datatype package, based on ideas from the nominal logic by Pitts et al., and demonstrate an implementation in Isabelle/HOL. The purpose is to derive powerful induction rules for the semantics in order to conduct machine checkable proofs, closely following the intuitive arguments found in manual proofs. In this way we have covered many of the standard theorems of bisimulation equivalence and congruence, both late and early, and both strong and weak in a uniform manner. We thus provide one of the most extensive formalisations of a the pi-calculus ever done inside a theorem prover.

Notes

Remember that we work with an encoding:



2.2 Arithmetic Within Types

Our second maxim is:

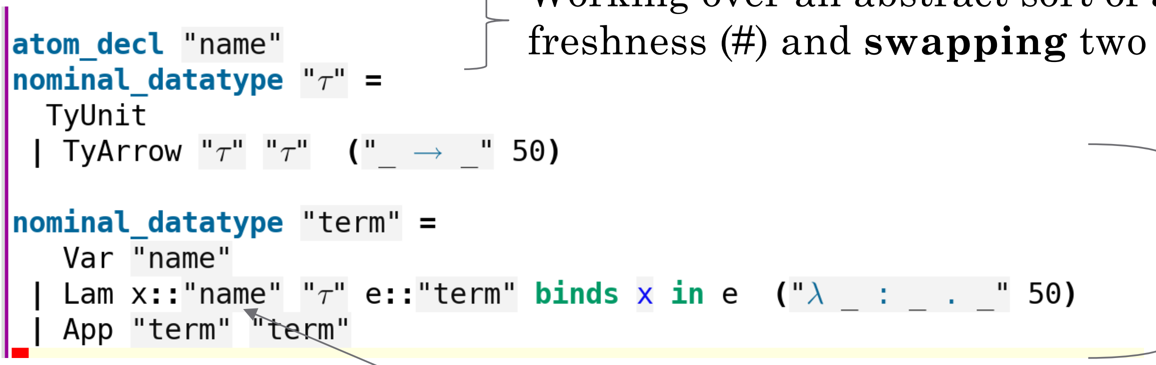
> When using a family of types indexed by **`nat`**, make sure that the index term never involves **`plus`** or **`times`**

or, more briefly, *avoid arithmetic within types.*

Adams, R.: Formalized metatheory with terms represented by an indexed family of types. In: Types for Proofs and Programs (2006)

Equivariance:
A definition commutes with permutation.

Nominal Techniques



```
atom_decl "name"
nominal_datatype "τ" =
  TyUnit
 | TyArrow "τ" "τ"  ("_ → _" 50)

nominal_datatype "term" =
   Var "name"
 | Lam x::"name" "τ" e::"term" binds x in e  ("λ _ : _ . _" 50)
 | App "term" "term"
```

Working over an abstract sort of **atoms** - allowing freshness (#) and **swapping** two atoms

Ensures that the datatype respects equivariance

Abstracting over a new name

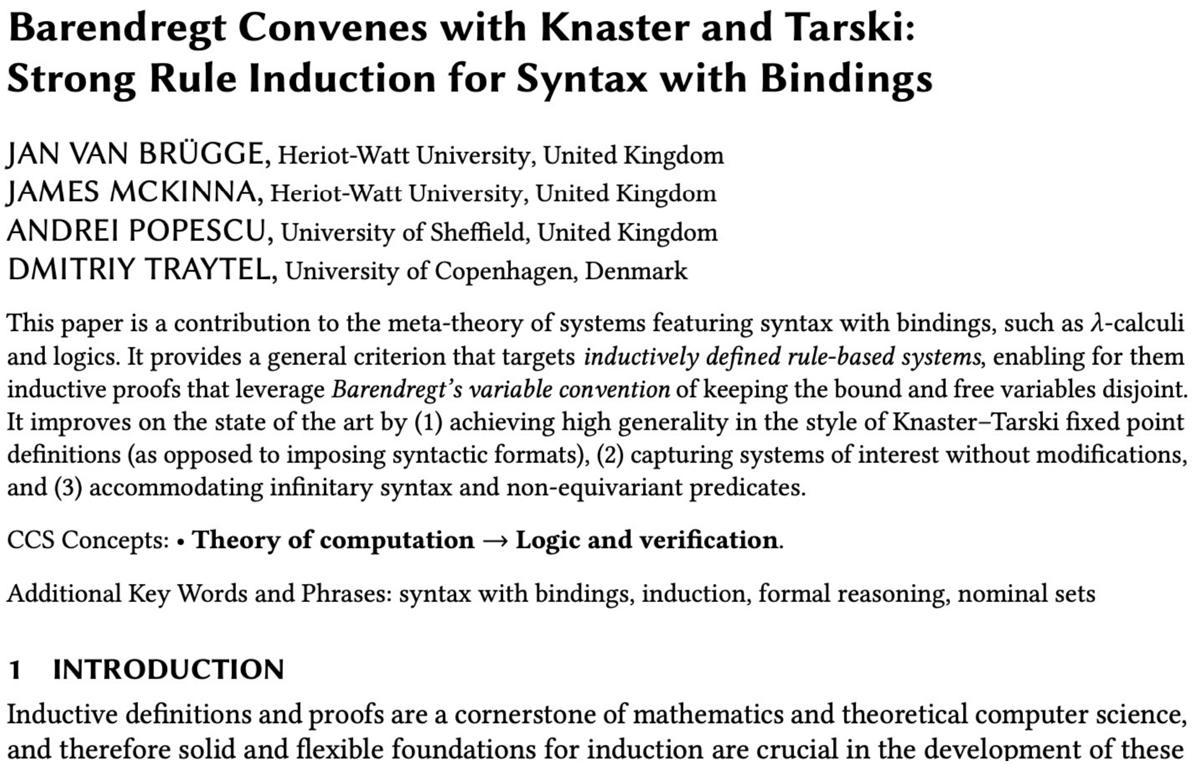
$$supp\ x \overset{def}{=} \{a \mid infinite\ \{b \mid (a\ b) \bullet x \neq x\}\}$$

A notion of support:
A finite set of variables that the definition may contain

There is also the derived notion for when an atom a is *fresh* for an x, defined as

$$a \# x \overset{def}{=} a \notin supp\ x$$

Nominal Logic, a first-oder theory of names and binders – Pitts 2003.
Nominal Unification, Urban, Pitts, Gabbay – 2004.
Nominal Techniques in Isabelle/HOL. Urban/Tasson 2005.
General Bindings and Alpha-Equivalence in Nominal Isabelle, Urban/Kaliszyk ‘12



**Barendregt Convenes with Knaster and Tarski:
Strong Rule Induction for Syntax with Bindings**

JAN VAN BRÜGGE, Heriot-Watt University, United Kingdom
JAMES MCKINNA, Heriot-Watt University, United Kingdom
ANDREI POPESCU, University of Sheffield, United Kingdom
DMITRIY TRAYTEL, University of Copenhagen, Denmark

This paper is a contribution to the meta-theory of systems featuring syntax with bindings, such as λ-calculi and logics. It provides a general criterion that targets *inductively defined rule-based systems*, enabling for them inductive proofs that leverage *Barendregt's variable convention* of keeping the bound and free variables disjoint. It improves on the state of the art by (1) achieving high generality in the style of Knaster–Tarski fixed point definitions (as opposed to imposing syntactic formats), (2) capturing systems of interest without modifications, and (3) accommodating infinitary syntax and non-equivariant predicates.

CCS Concepts: • **Theory of computation → Logic and verification**.

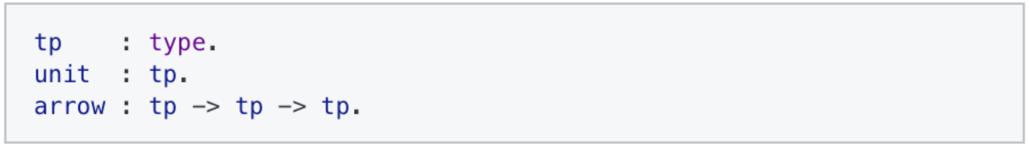
Additional Key Words and Phrases: syntax with bindings, induction, formal reasoning, nominal sets

1 INTRODUCTION

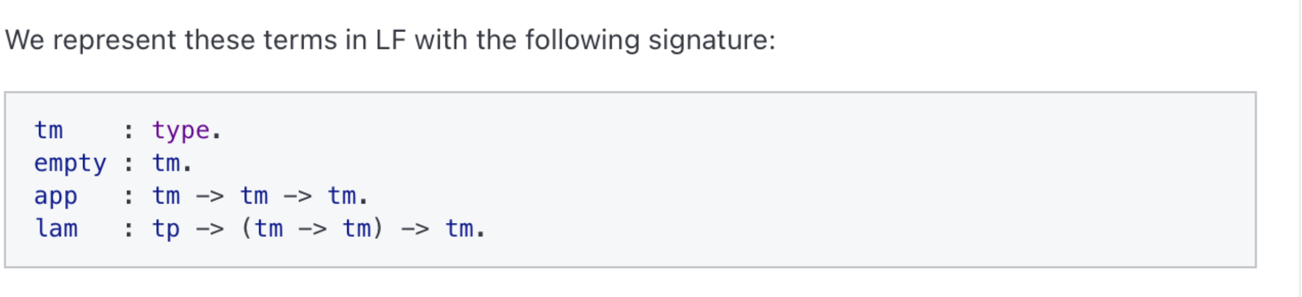
Inductive definitions and proofs are a cornerstone of mathematics and theoretical computer science, and therefore solid and flexible foundations for induction are crucial in the development of these

See Zulip

Higher-Order Abstract Syntax



```
tp    : type.
unit  : tp.
arrow : tp -> tp -> tp.
```



We represent these terms in LF with the following signature:

```
tm    : type.
empty : tm.
app   : tm -> tm -> tm.
lam   : tp -> (tm -> tm) -> tm.
```

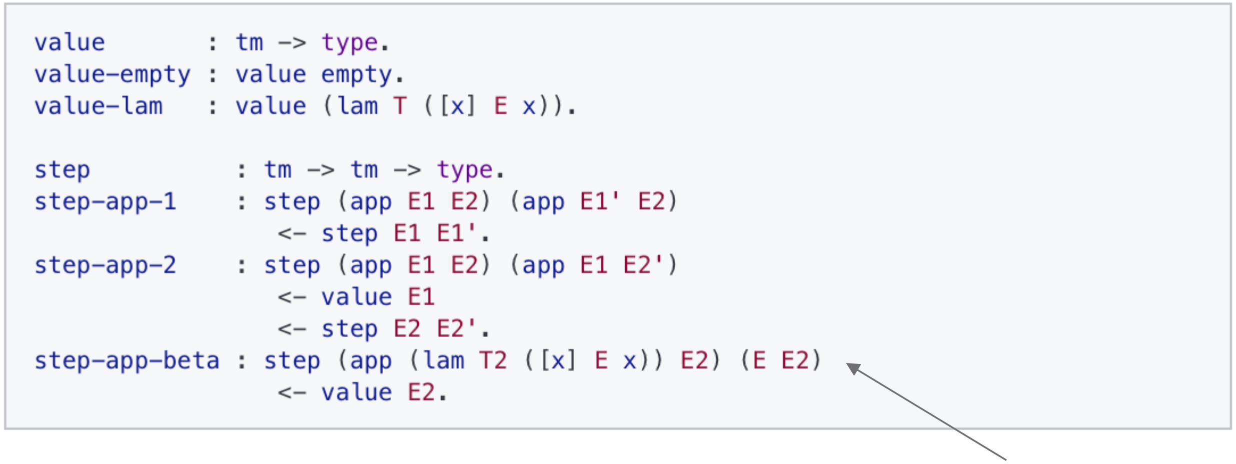
Example: $\lambda x. \lambda y. x\ y$

```
lam (arrow unit unit) ([x] (lam unit ([y] app x y))
```

α-equivalent by construction:
no way to distinguish
`[x] (lam unit ([y] app x y)`
and
`([z] (lam unit ([y] app z y)`
in the meta-theory

Well scoped by construction

Higher-Order Abstract Syntax, Pfenning/Elliot ’88
Twelf: Pfenning/Schürmann ‘99
https://twelf.org/wiki/proving-metatheorems-representing-the-syntax-of-the-stlc/



```
value       : tm -> type.
value-empty : value empty.
value-lam   : value (lam T ([x] E x)).

step          : tm -> tm -> type.
step-app-1    : step (app E1 E2) (app E1' E2)
                 <- step E1 E1'.
step-app-2    : step (app E1 E2) (app E1 E2')
                 <- value E1
                 <- step E2 E2'.
step-app-beta : step (app (lam T2 ([x] E x)) E2) (E E2)
                 <- value E2.
```

Substitutions for free

Does this correctly implement what we want?

Adequacy: The representation within the meta-language is syntactically correct (a one-to-one correspondence between the objects in the object language/the representation in the meta language) and semantically faithful.

=> The term function space tm -> tm must correspond to the type of open types with a single free variable.

The meta language matters =>
no elimination of constants possible:

Exotic term:
No corresponding object-language term with a free variable

```
lam ([x : tm]
match x with
 | empty =>
 | _ => …
end)
```

The meta language matters =>
no classical metatheory possible:

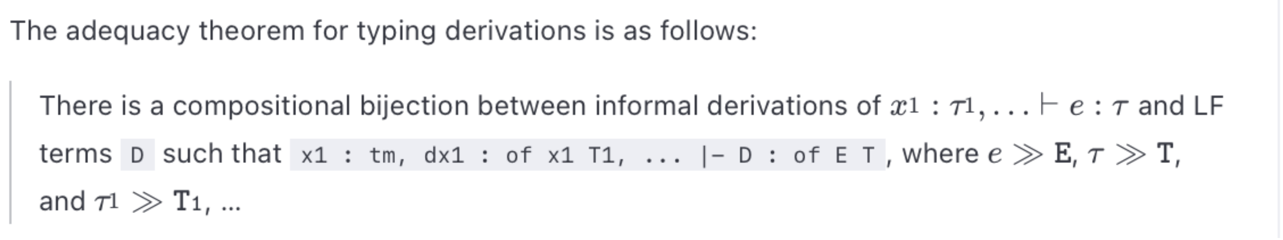
```
lam ([x : tm]
if (x = empty)
then empty
else …)
```

Mechanizing Metatheory in a Logical Framework, Harper/Licata 2007

Typing Statement

No variable rule!

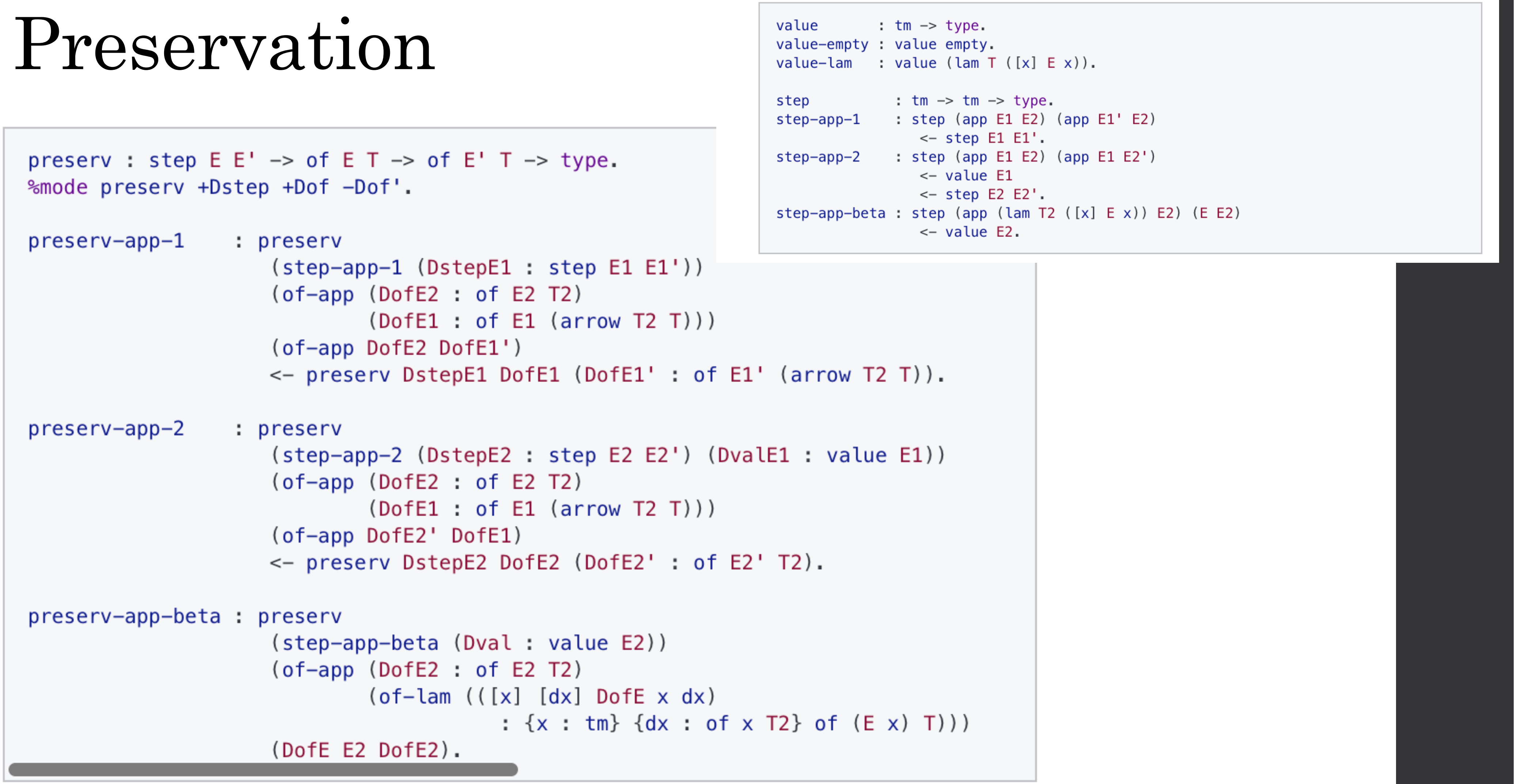
```
of : tm -> tp -> type.

of-empty : of empty unit.

of-lam : of (lam T2 ([x] E x)) (arrow T2 T) <- ({x: tm} of x T2 ->
of (E x) T).

of-app : of (app E1 E2) T <- of E1 (arrow T2 T) <- of E2 T2.
```



The adequacy theorem for typing derivations is as follows:

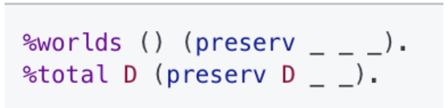
> There is a compositional bijection between informal derivations of $x_1 : \tau_1, \ldots \vdash e : \tau$ and LF terms `D` such that `x1 : tm, dx1 : of x1 T1, ... |- D : of E T`, where $e \gg \mathbf{E}$, $\tau \gg \mathbf{T}$, and $\tau_1 \gg \mathbf{T}_1$, ...

Preservation



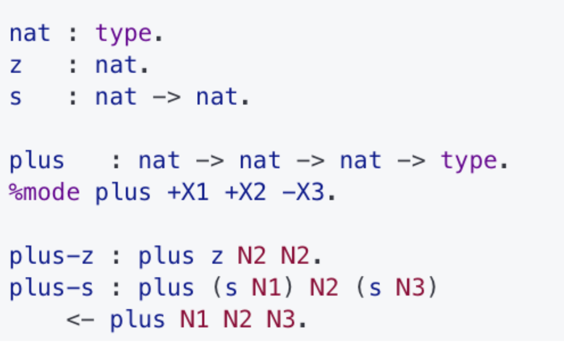
```
preserv : step E E' -> of E T -> of E' T -> type.
%mode preserv +Dstep +Dof -Dof'.

preserv-app-1    : preserv
                    (step-app-1 (DstepE1 : step E1 E1'))
                    (of-app (DofE2 : of E2 T2)
                            (DofE1 : of E1 (arrow T2 T)))
                    (of-app DofE2 DofE1')
                    <- preserv DstepE1 DofE1 (DofE1' : of E1' (arrow T2 T)).

preserv-app-2    : preserv
                    (step-app-2 (DstepE2 : step E2 E2') (DvalE1 : value E1))
                    (of-app (DofE2 : of E2 T2)
                            (DofE1 : of E1 (arrow T2 T)))
                    (of-app DofE2' DofE1)
                    <- preserv DstepE2 DofE2 (DofE2' : of E2' T2).

preserv-app-beta : preserv
                    (step-app-beta (Dval : value E2))
                    (of-app (DofE2 : of E2 T2)
                            (of-lam (([x] [dx] DofE x dx)
                                       : {x : tm} {dx : of x T2} of (E x) T)))
                    (DofE E2 DofE2).
```

```
value       : tm -> type.
value-empty : value empty.
value-lam   : value (lam T ([x] E x)).

step          : tm -> tm -> type.
step-app-1    : step (app E1 E2) (app E1' E2)
                 <- step E1 E1'.
step-app-2    : step (app E1 E2) (app E1 E2')
                 <- value E1
                 <- step E2 E2'.
step-app-beta : step (app (lam T2 ([x] E x)) E2) (E E2)
                 <- value E2.
```

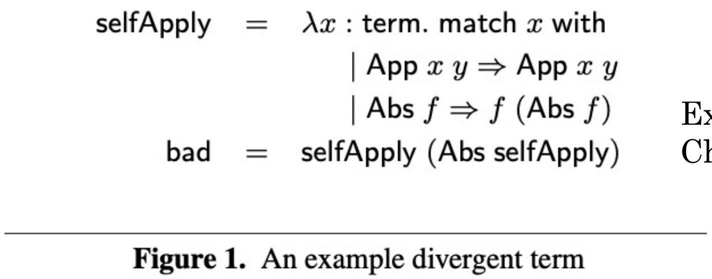


```
%worlds () (preserv _ _ _).
%total D (preserv D _ _).
```

Higher-Order Abstract Syntax



```
nat : type.
z   : nat.
s   : nat -> nat.

plus   : nat -> nat -> nat -> type.
%mode plus +X1 +X2 -X3.

plus-z : plus z N2 N2.
plus-s : plus (s N1) N2 (s N3)
    <- plus N1 N2 N3.
```

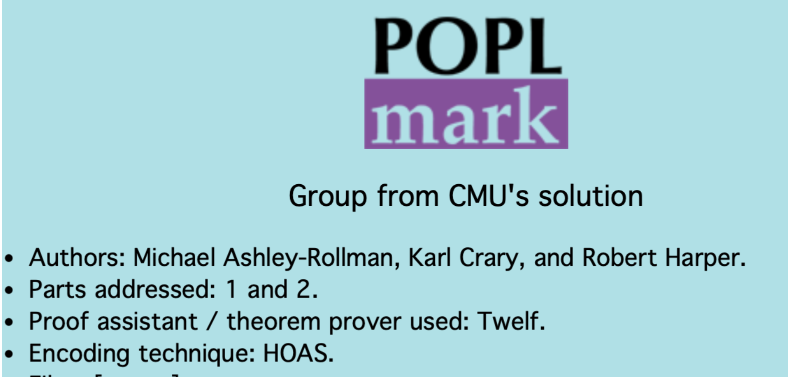


$$
\begin{aligned}
\mathsf{selfApply} &= \lambda x : \mathsf{term}.\ \mathsf{match}\ x\ \mathsf{with} \\
&\qquad \mid \mathsf{App}\ x\ y \Rightarrow \mathsf{App}\ x\ y \\
&\qquad \mid \mathsf{Abs}\ f \Rightarrow f\ (\mathsf{Abs}\ f) \\
\mathsf{bad} &= \mathsf{selfApply}\ (\mathsf{Abs}\ \mathsf{selfApply})
\end{aligned}
$$

Figure 1. An example divergent term

Example from:
Chlipala, PHOAS

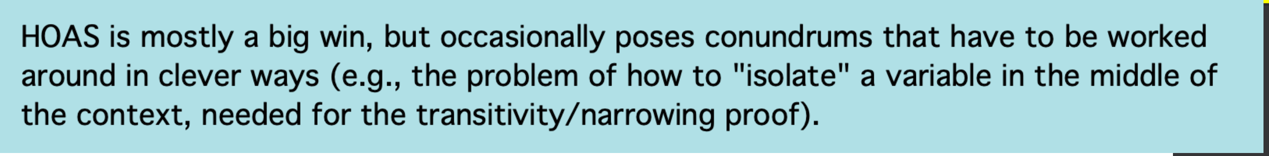
- Substitutions/substitutivity for free
- No explicit variable constructor/since variables are represented as metalevel variables they are implicit/we cannot write definitions which mention them explicitly
- Impossible to use in a general-purpose proof assistant
- Restrictions on recursive functions



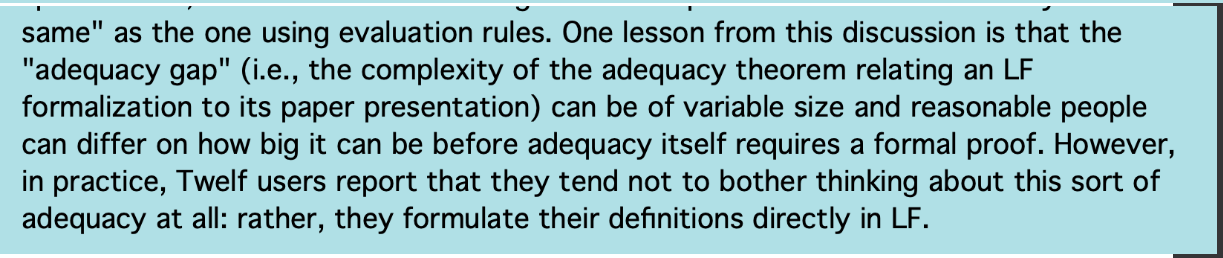
POPL
mark

Group from CMU's solution

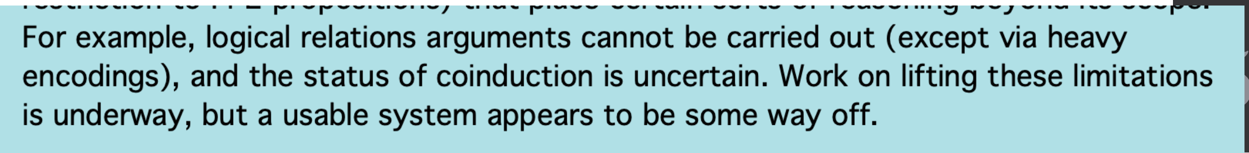
- Authors: Michael Ashley-Rollman, Karl Crary, and Robert Harper.
- Parts addressed: 1 and 2.
- Proof assistant / theorem prover used: Twelf.
- Encoding technique: HOAS.



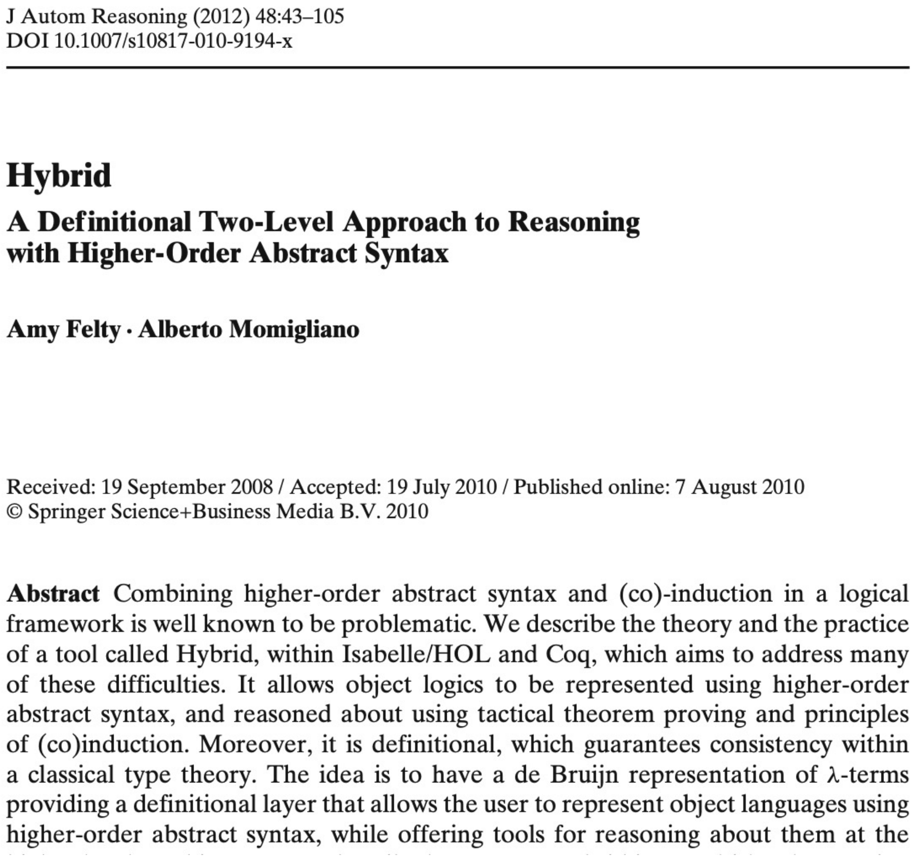
HOAS is mostly a big win, but occasionally poses conundrums that have to be worked around in clever ways (e.g., the problem of how to "isolate" a variable in the middle of the context, needed for the transitivity/narrowing proof).



same" as the one using evaluation rules. One lesson from this discussion is that the "adequacy gap" (i.e., the complexity of the adequacy theorem relating an LF formalization to its paper presentation) can be of variable size and reasonable people can differ on how big it can be before adequacy itself requires a formal proof. However, in practice, Twelf users report that they tend not to bother thinking about this sort of adequacy at all: rather, they formulate their definitions directly in LF.



restriction to Π2 propositions) that place certain sorts of reasoning beyond its scope. For example, logical relations arguments cannot be carried out (except via heavy encodings), and the status of coinduction is uncertain. Work on lifting these limitations is underway, but a usable system appears to be some way off.



J Autom Reasoning (2012) 48:43–105
DOI 10.1007/s10817-010-9194-x

Hybrid

A Definitional Two-Level Approach to Reasoning with Higher-Order Abstract Syntax

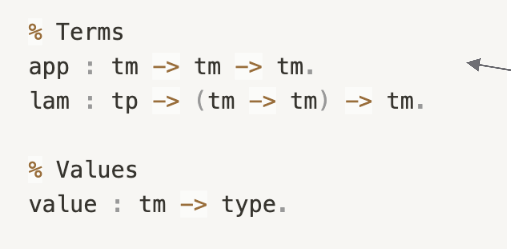
Amy Felty · Alberto Momigliano

Received: 19 September 2008 / Accepted: 19 July 2010 / Published online: 7 August 2010
© Springer Science+Business Media B.V. 2010

Abstract Combining higher-order abstract syntax and (co)-induction in a logical framework is well known to be problematic. We describe the theory and the practice of a tool called Hybrid, within Isabelle/HOL and Coq, which aims to address many of these difficulties. It allows object logics to be represented using higher-order abstract syntax, and reasoned about using tactical theorem proving and principles of (co)induction. Moreover, it is definitional, which guarantees consistency within a classical type theory. The idea is to have a de Bruijn representation of λ-terms providing a definitional layer that allows the user to represent object languages using higher-order abstract syntax, while offering tools for reasoning about them at the

Working in a model of HOAS

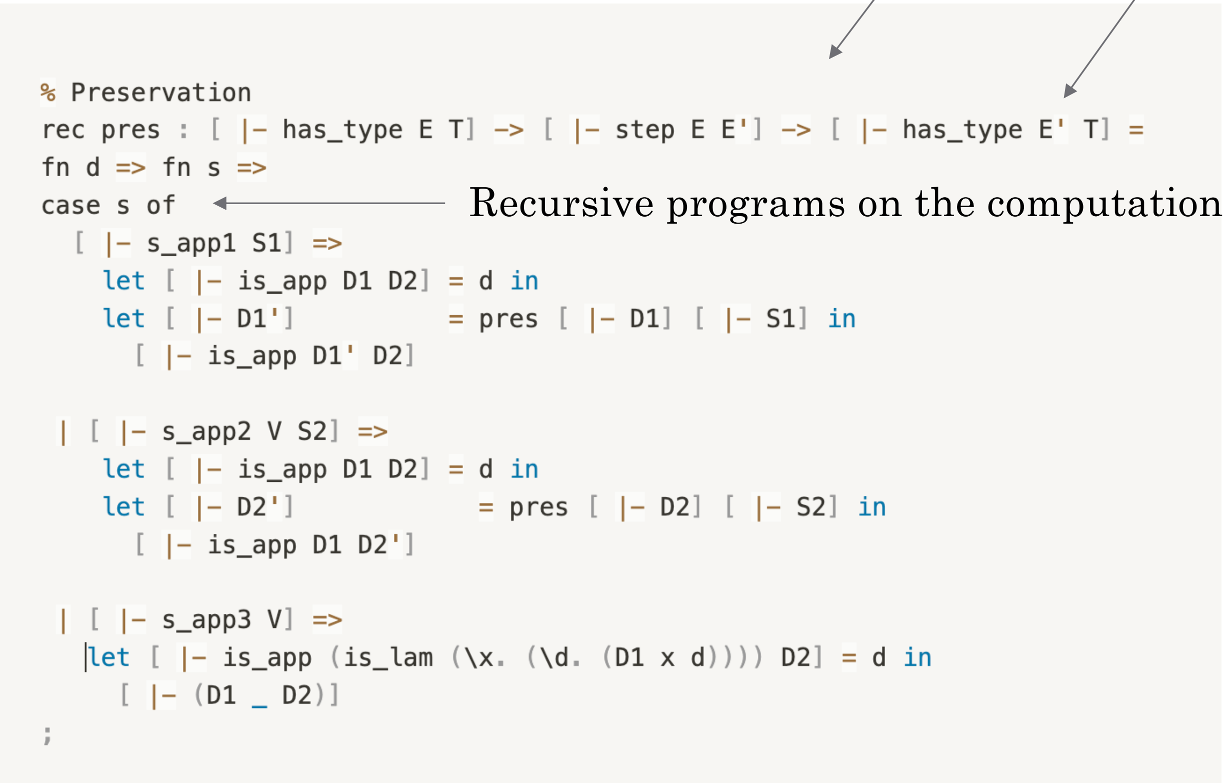
Beluga/Contextual Modal Type Theory



```
% Terms
app : tm -> tm -> tm.
lam : tp -> (tm -> tm) -> tm.

% Values
value : tm -> type.
```

Distinguishes
data
and computations:

Boxed value:
embedded into computations;
no computations inside

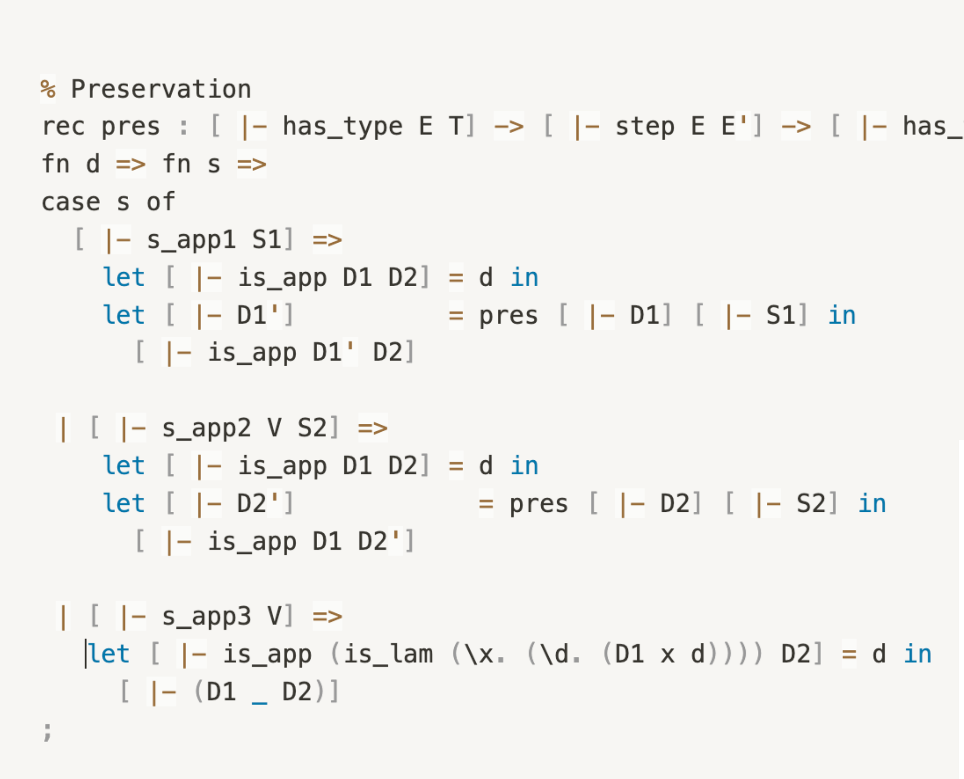


```
% Preservation
rec pres : [ |- has_type E T] -> [ |- step E E'] -> [ |- has_type E' T] =
fn d => fn s =>
case s of
  [ |- s_app1 S1] =>
    let [ |- is_app D1 D2] = d in
    let [ |- D1']          = pres [ |- D1] [ |- S1] in
      [ |- is_app D1' D2]

 | [ |- s_app2 V S2] =>
    let [ |- is_app D1 D2] = d in
    let [ |- D2']            = pres [ |- D2] [ |- S2] in
      [ |- is_app D1 D2']

 | [ |- s_app3 V] =>
   let [ |- is_app (is_lam (\x. (\d. (D1 x d)))) D2] = d in
     [ |- (D1 _ D2)]
;
```

Recursive programs on the computation level

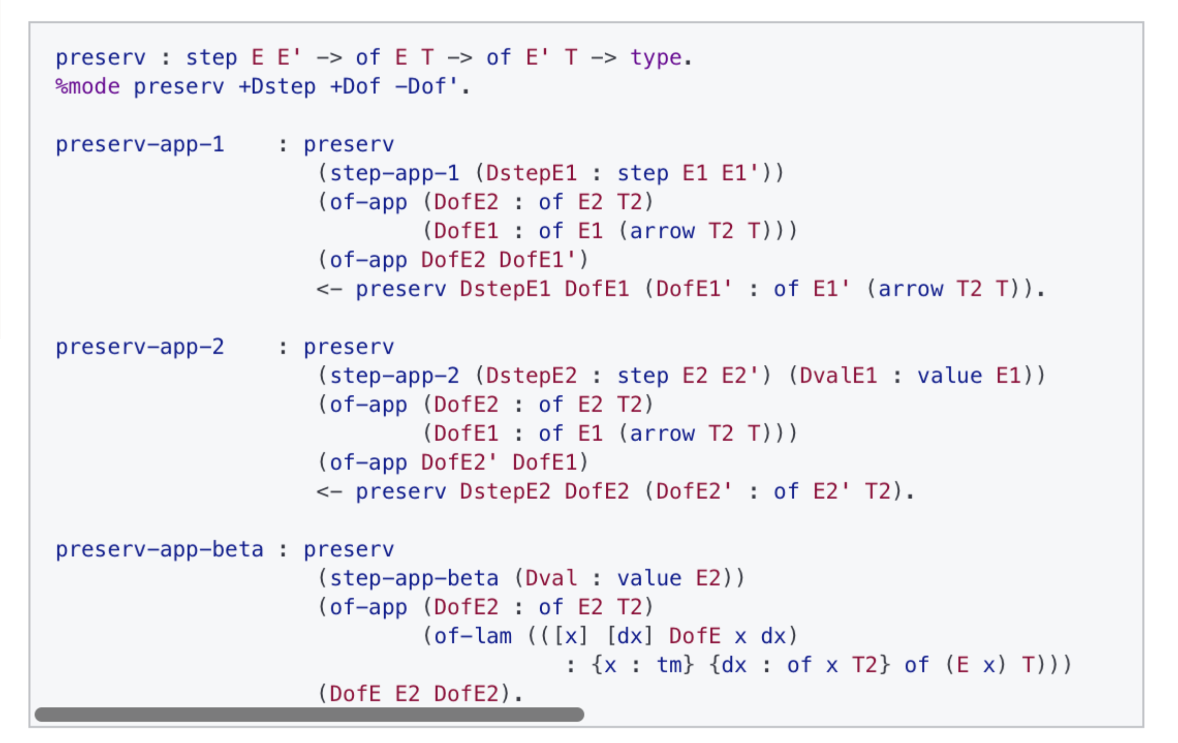
- Extension of HOAS with a modality to talk about open terms => expressivity
- Comes with a notion of context morphisms

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. „Contextual Modal Type Theory“ 2008
Pientka/Dunfield: Beluga: A framework for programming and reasoning with deductive systems 2010
Comparison with de Bruijn proofs:
Kaiser, Jonas, Brigitte Pientka, and Gert Smolka. "Relating system F and Lambda2: A case study in Coq, Abella and Beluga.” 2017
POPLMark Reloaded Challenge



```
% Preservation
rec pres : [ |- has_type E T] -> [ |- step E E'] -> [ |- has_type E' T] =
fn d => fn s =>
case s of
 [ |- s_app1 S1] =>
  let [ |- is_app D1 D2] = d in
  let [ |- D1']          = pres [ |- D1] [ |- S1] in
   [ |- is_app D1' D2]

 | [ |- s_app2 V S2] =>
  let [ |- is_app D1 D2] = d in
  let [ |- D2']           = pres [ |- D2] [ |- S2] in
   [ |- is_app D1 D2']

 | [ |- s_app3 V] =>
  let [ |- is_app (is_lam (\x. (\d. (D1 x d)))) D2] = d in
   [ |- (D1 _ D2)]
;
```

Twelf



```
preserv : step E E' -> of E T -> of E' T -> type.
%mode preserv +Dstep +Dof -Dof'.

preserv-app-1    : preserv
                    (step-app-1 (DstepE1 : step E1 E1'))
                    (of-app (DofE2 : of E2 T2)
                            (DofE1 : of E1 (arrow T2 T)))
                    (of-app DofE2 DofE1')
                    <- preserv DstepE1 DofE1 (DofE1' : of E1' (arrow T2 T)).

preserv-app-2    : preserv
                    (step-app-2 (DstepE2 : step E2 E2') (DvalE1 : value E1))
                    (of-app (DofE2 : of E2 T2)
                            (DofE1 : of E1 (arrow T2 T)))
                    (of-app DofE2' DofE1)
                    <- preserv DstepE2 DofE2 (DofE2' : of E2' T2).

preserv-app-beta : preserv
                    (step-app-beta (Dval : value E2))
                    (of-app (DofE2 : of E2 T2)
                            (of-lam (([x] [dx] DofE x dx)
                                       : {x : tm} {dx : of x T2} of (E x) T)))
                    (DofE E2 DofE2).
```

Beluga

```
schema cxt = tm A;

inductive Sn : (Γ : cxt) {M : [Γ ⊢ tm A[]]} type =
| Acc : {Γ : cxt}{A:[ ⊢ ty]}{M : [Γ ⊢ tm A[]]}
        ({M’ : [Γ ⊢ tm A[]]} {S : [Γ ⊢ step M M’]} Sn [Γ ⊢ M’])
     → Sn [Γ ⊢ M]

 rec anti_renameSN : {Γ : cxt}{Γ′ : cxt} {ρ : [Γ′ ⊢# Γ]}{M : [Γ ⊢ tm A[]]}
                     SN [Γ′ ⊢ M[ρ]] → SN [Γ ⊢ M]  =
      / total s (anti_renameSN Γ Γ′ A ρ M s) /
 mlam Γ, Γ’, ρ, M ⇒ fn s ⇒ case s of
 | SAbs s’ ⇒
      SAbs (anti_renameSN [Γ, x:tm _] [Γ′, x:tm _ ] [Γ’, x:tm _ ⊢ ρ[…], x
     ] [Γ, x:tm _ ⊢ _] s’)
 | SNeu s’ ⇒  SNeu (anti_renameSNe [Γ′ ⊢ ρ] [Γ ⊢ M] s’)
 | SRed r’ s’ ⇒
    let SNRed’ [Γ′] [Γ]  [Γ ⊢ N ] r = anti_renameSNRed [_] [_] [Γ′ ⊢ ρ] [_
      ⊢ _ ] r’ in
    let s’’ = anti_renameSN [Γ] [Γ′] [Γ′ ⊢ ρ] [Γ ⊢ N ] s’ in
      SRed r s’’
```

Weak HOAS

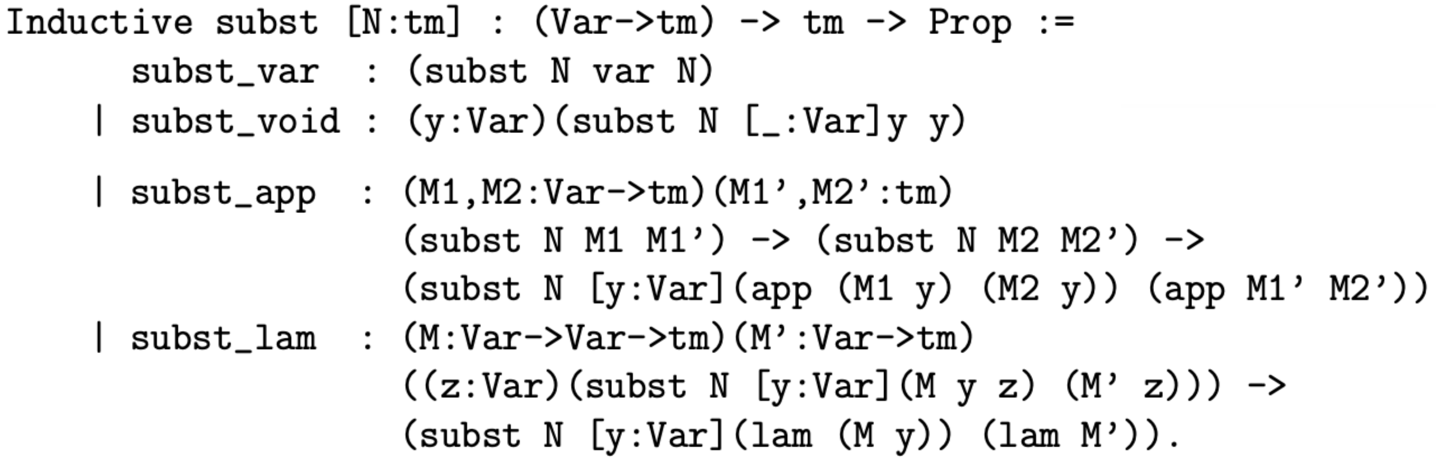
- Admits larger function spaces

```
Parameter Var : Set.

Inductive tm : Set :=
 var : Var -> tm
| app:tm->tm->tm
| lam:(Var->tm)->tm.
```

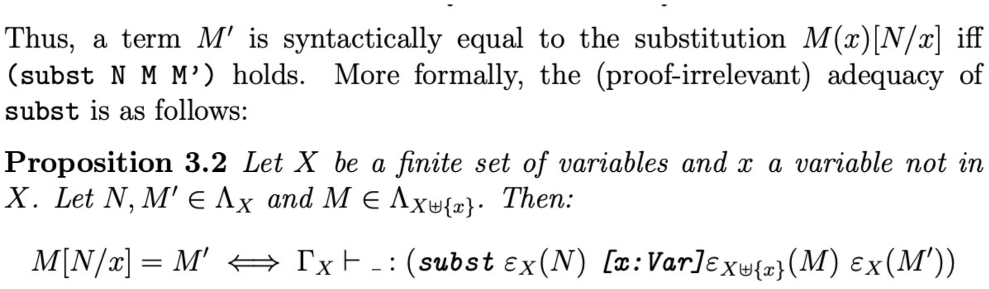
$$\lambda x. x\ (\lambda\, y. x\ y)$$

```
Example ex : tm := lam (fun x => app
(var x) (lam (fun y => app (var x)
(var y)))).
```

Martin Hofmann. „Semantical analysis of higher-order abstract syntax“. **1999.**
Honsell, Miculan, Scagnetto – The Theory of Contexts for First Order and Higher Order Abstract Syntax, 2002
Adequacy: Miculan – Developing (meta)theory of lambda-calculus in the Theory of Contexts



```
Inductive subst [N:tm] : (Var->tm) -> tm -> Prop :=
      subst_var  : (subst N var N)
    | subst_void : (y:Var)(subst N [_:Var]y y)

    | subst_app  : (M1,M2:Var->tm)(M1’,M2’:tm)
                   (subst N M1 M1’) -> (subst N M2 M2’) ->
                   (subst N [y:Var](app (M1 y) (M2 y)) (app M1’ M2’))
    | subst_lam  : (M:Var->Var->tm)(M’:Var->tm)
                   ((z:Var)(subst N [y:Var](M y z) (M’ z))) ->
                   (subst N [y:Var](lam (M y)) (lam M’)).
```



Thus, a term M' is syntactically equal to the substitution $M(x)[N/x]$ iff `(subst N M M’)` holds. More formally, the (proof-irrelevant) adequacy of `subst` is as follows:

Proposition 3.2 *Let X be a finite set of variables and x a variable not in X. Let $N, M' \in \Lambda_X$ and $M \in \Lambda_{X \uplus \{x\}}$. Then:*

$$M[N/x] = M' \iff \Gamma_X \vdash \_ : (\textit{\textbf{subst}}\ \varepsilon_X(N)\ \textit{\textbf{[x:Var]}}\varepsilon_{X\uplus\{x\}}(M)\ \varepsilon_X(M'))$$

Requires again adequacy

Honsell, Miculan, Scagnetto – The Theory of Contexts for First Order and Higher Order Abstract Syntax, 2002

```
Parameter Var : Set.

Inductive tm : Set :=
 var : Var -> tm
| app:tm->tm->tm
| lam:(Var->tm)->tm.

Inductive countvars : tm -> nat -> Prop :=
| cv_var x : countvars (var x) 1
| cv_app s t m n: countvars s m
             -> countvars t n -> countvars (app s t) (m + n)
| cv_lam f n: forall x, countvars (f x) n -> countvars (lam f) n.

Fixpoint countvars (t : tm) : nat :=
match t with
| var x => 1
| app s t => countvars s + countvars t
| lam f => countvars (f ?) end.
```

Parametric Higher-Order Abstract Syntax (PHOAS)

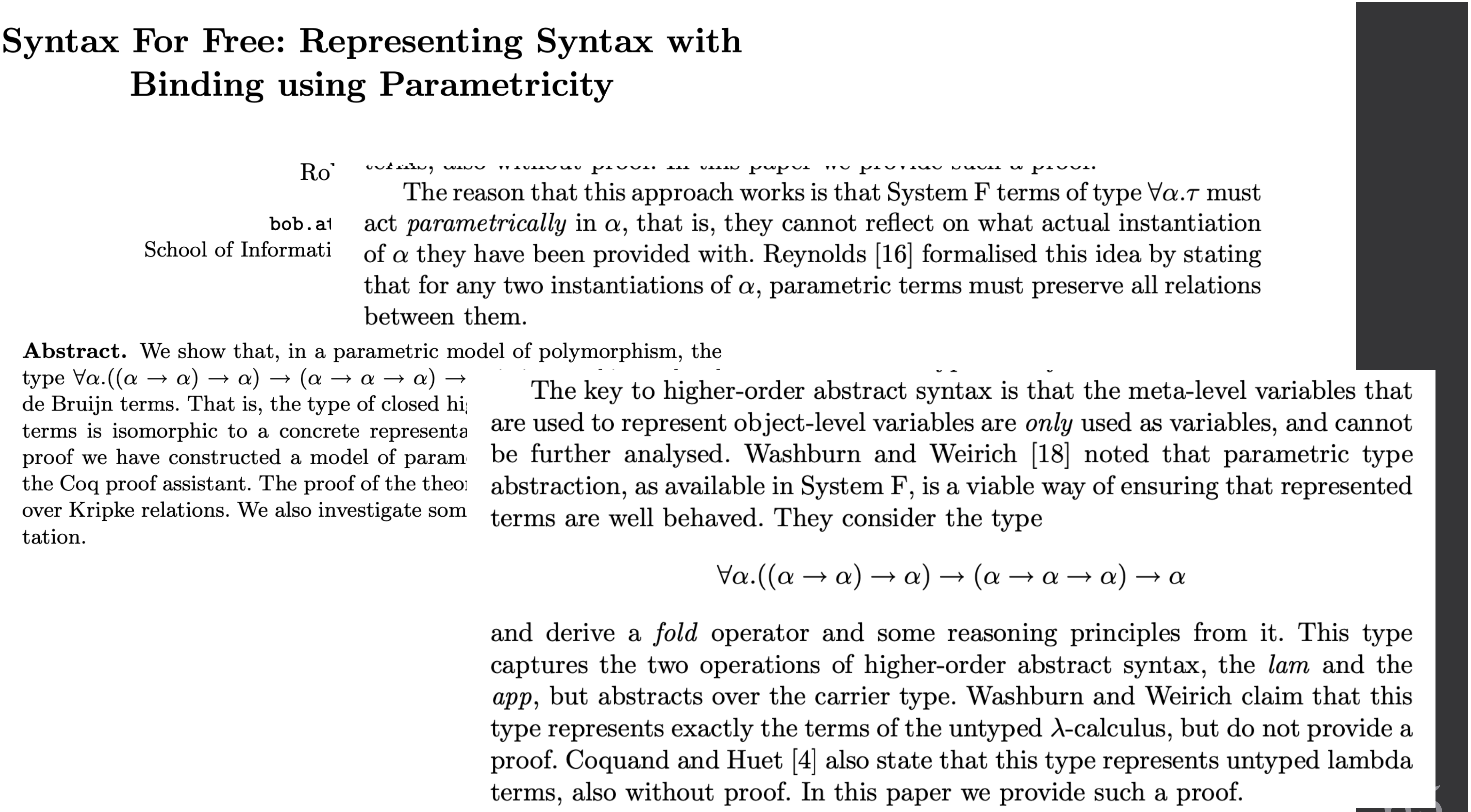
```
Section tm.

Variable var : Type.

Inductive tm : Type :=
| Var : var -> tm
| App' : tm -> tm -> tm
| Abs' : (var -> tm) -> tm.

End tm.

Definition Tm := forall X, tm X.
```

```
Fixpoint count (e: tm unit) : nat :=
match e with
| Var _ x => 1
| App' _ s t => count s + count t
| Abs' _ s => count (s tt) end.

Definition Count (e : Tm) : nat :=
count (e unit).
```

Washburn, Geoffrey, and Stephanie Weirich. "Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism." *ACM SIGPLAN Notices* 38.9 (2003): 249-262.
Chlipala, Adam. "Parametric higher-order abstract syntax for mechanized semantics." *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. 2008.
Proof of adequacy: Atkey, Syntax For Free: Representing Syntax with Binding using Parametricity

Substitution

```
Fixpoint subst {X: Type} (s : tm (tm X)) :=
 match s with
| Var _ x => x
| App' _ s t => App' _ (subst s) (subst t)
| Abs' _ s => Abs' _ (fun x => subst (s (Var _ x)))
end.

Definition Subst (s : forall X, X -> tm X) (t : Tm) : Tm :=
fun X => subst (s _ (t X)).
```



Syntax For Free: Representing Syntax with Binding using Parametricity

Ro

bob.at
School of Informati

Abstract. We show that, in a parametric model of polymorphism, the type $\forall\alpha.((\alpha \to \alpha) \to \alpha) \to (\alpha \to \alpha \to \alpha) \to$
de Bruijn terms. That is, the type of closed hi
terms is isomorphic to a concrete representa
proof we have constructed a model of param
the Coq proof assistant. The proof of the theo
over Kripke relations. We also investigate som
tation.

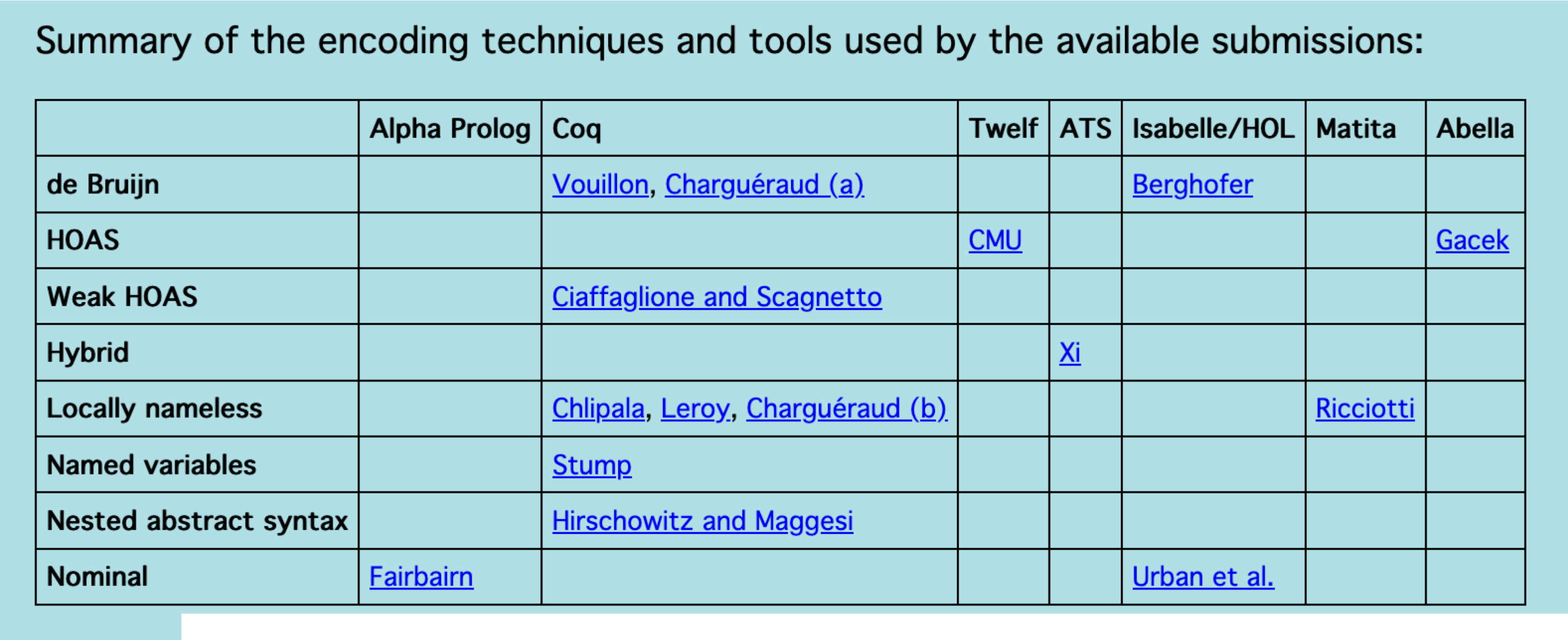
terms, also without proof. In this paper we provide such a proof.

The reason that this approach works is that System F terms of type $\forall\alpha.\tau$ must act *parametrically* in α, that is, they cannot reflect on what actual instantiation of α they have been provided with. Reynolds [16] formalised this idea by stating that for any two instantiations of α, parametric terms must preserve all relations between them.

The key to higher-order abstract syntax is that the meta-level variables that are used to represent object-level variables are *only* used as variables, and cannot be further analysed. Washburn and Weirich [18] noted that parametric type abstraction, as available in System F, is a viable way of ensuring that represented terms are well behaved. They consider the type

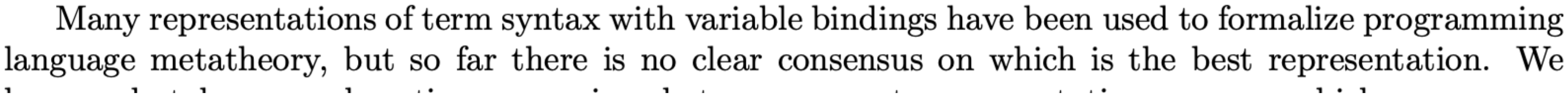
$$\forall\alpha.((\alpha \to \alpha) \to \alpha) \to (\alpha \to \alpha \to \alpha) \to \alpha$$

and derive a *fold* operator and some reasoning principles from it. This type captures the two operations of higher-order abstract syntax, the *lam* and the *app*, but abstracts over the carrier type. Washburn and Weirich claim that this type represents exactly the terms of the untyped λ-calculus, but do not provide a proof. Coquand and Huet [4] also state that this type represents untyped lambda terms, also without proof. In this paper we provide such a proof.

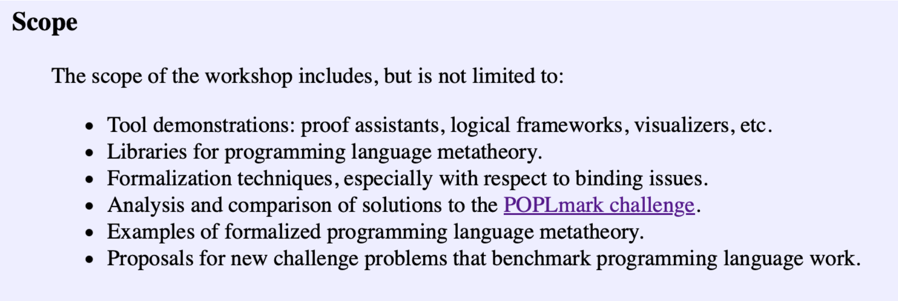


Summary of the encoding techniques and tools used by the available submissions:

	Alpha Prolog	Coq	Twelf	ATS	Isabelle/HOL	Matita	Abella
de Bruijn		Vouillon, Charguéraud (a)			Berghofer		
HOAS			CMU				Gacek
Weak HOAS		Ciaffaglione and Scagnetto					
Hybrid				Xi			
Locally nameless		Chlipala, Leroy, Charguéraud (b)				Ricciotti	
Named variables		Stump					
Nested abstract syntax		Hirschowitz and Maggesi					
Nominal	Fairbairn				Urban et al.		



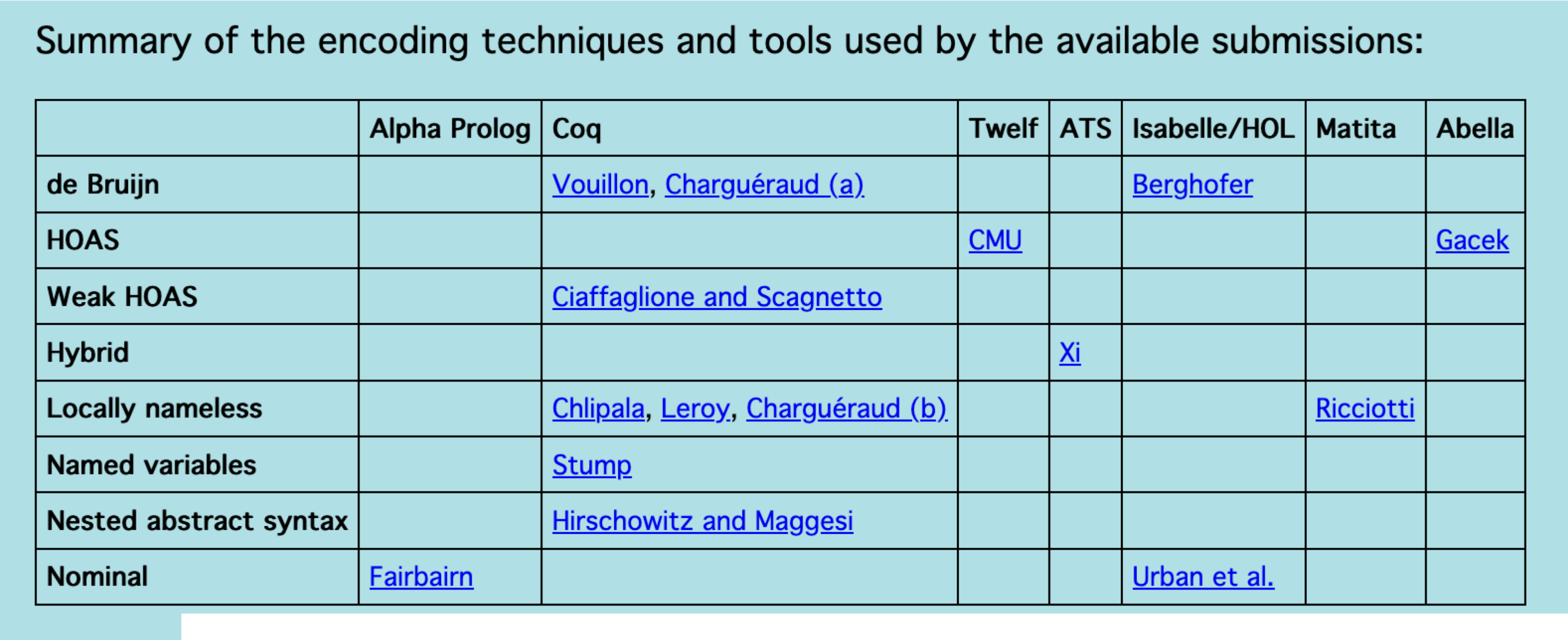
Many representations of term syntax with variable bindings have been used to formalize programming language metatheory, but so far there is no clear consensus on which is the best representation. We



Scope

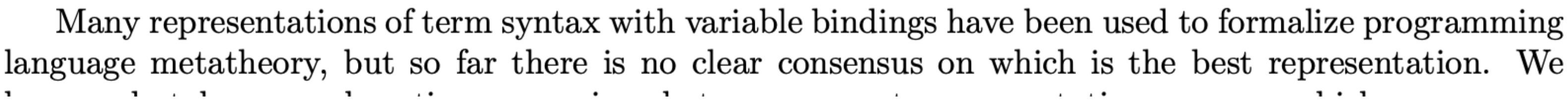
The scope of the workshop includes, but is not limited to:

- Tool demonstrations: proof assistants, logical frameworks, visualizers, etc.
- Libraries for programming language metatheory.
- Formalization techniques, especially with respect to binding issues.
- Analysis and comparison of solutions to the POPLmark challenge.
- Examples of formalized programming language metatheory.
- Proposals for new challenge problems that benchmark programming language work.

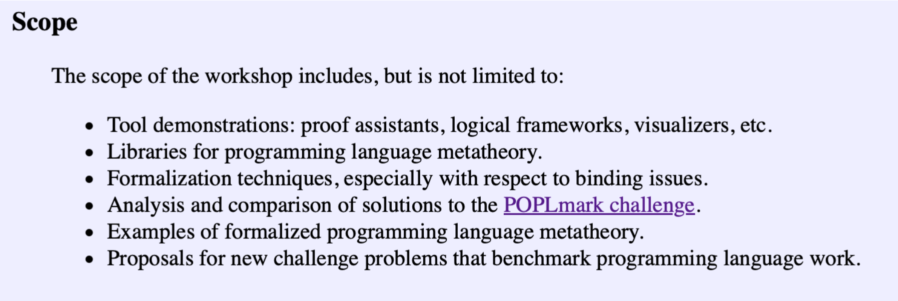


Summary of the encoding techniques and tools used by the available submissions:

	Alpha Prolog	Coq	Twelf	ATS	Isabelle/HOL	Matita	Abella
de Bruijn		Vouillon, Charguéraud (a)			Berghofer		
HOAS			CMU				Gacek
Weak HOAS		Ciaffaglione and Scagnetto					
Hybrid				Xi			
Locally nameless		Chlipala, Leroy, Charguéraud (b)				Ricciotti	
Named variables		Stump					
Nested abstract syntax		Hirschowitz and Maggesi					
Nominal	Fairbairn				Urban et al.		



Many representations of term syntax with variable bindings have been used to formalize programming language metatheory, but so far there is no clear consensus on which is the best representation. We



Scope

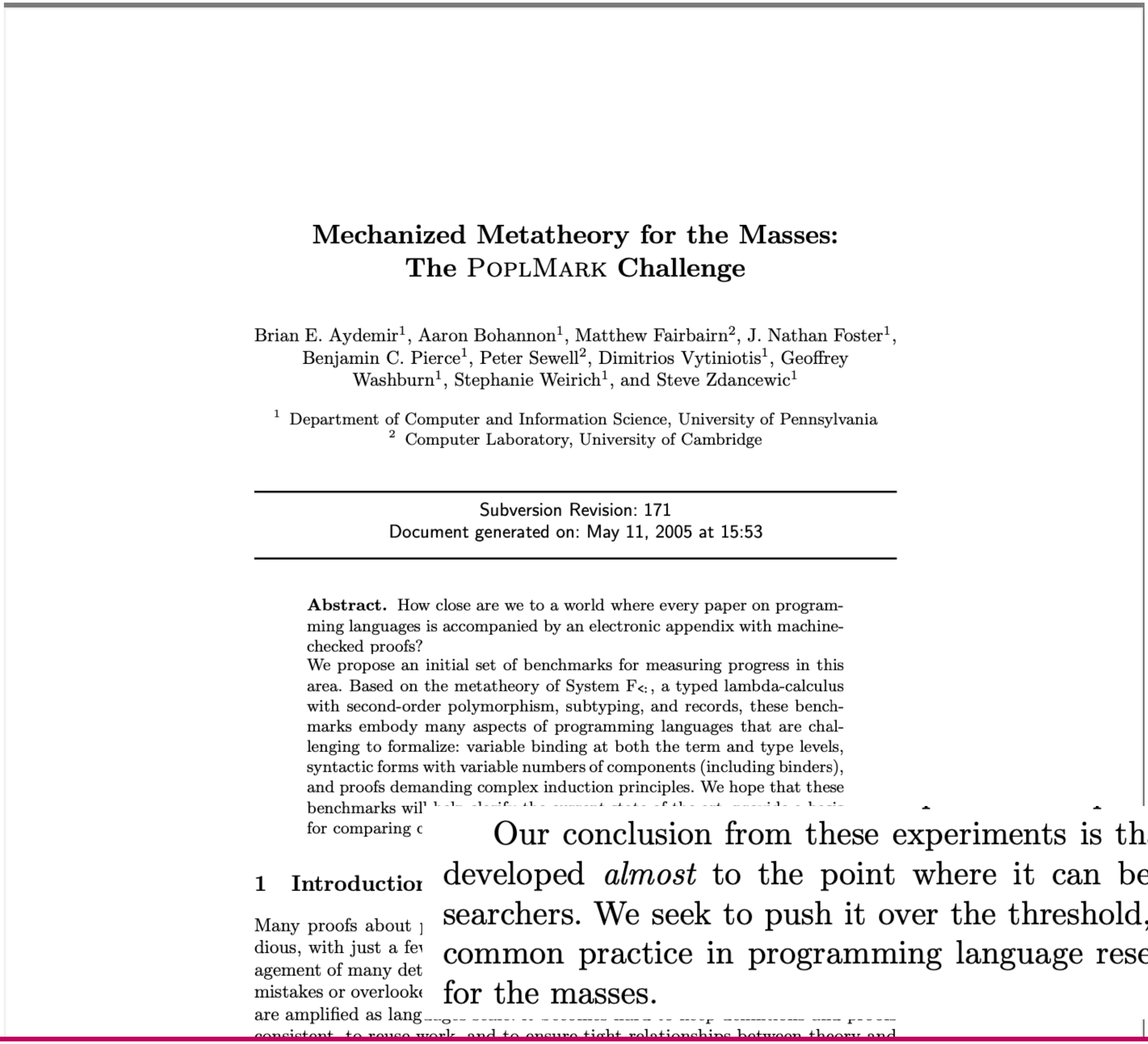
The scope of the workshop includes, but is not limited to:

- Tool demonstrations: proof assistants, logical frameworks, visualizers, etc.
- Libraries for programming language metatheory.
- Formalization techniques, especially with respect to binding issues.
- Analysis and comparison of solutions to the POPLmark challenge.
- Examples of formalized programming language metatheory.
- Proposals for new challenge problems that benchmark programming language work.

Separating Bound and Free Variables

Kathrin Stark

SPLV 2024



Mechanized Metatheory for the Masses: The PoplMark Challenge

Brian E. Aydemir[1], Aaron Bohannon[1], Matthew Fairbairn[2], J. Nathan Foster[1], Benjamin C. Pierce[1], Peter Sewell[2], Dimitrios Vytiniotis[1], Geoffrey Washburn[1], Stephanie Weirich[1], and Steve Zdancewic[1]

[1] Department of Computer and Information Science, University of Pennsylvania
[2] Computer Laboratory, University of Cambridge

Subversion Revision: 171
Document generated on: May 11, 2005 at 15:53

Abstract. How close are we to a world where every paper on programming languages is accompanied by an electronic appendix with machine-checked proofs?

We propose an initial set of benchmarks for measuring progress in this area. Based on the metatheory of System $F_{<:}$, a typed lambda-calculus with second-order polymorphism, subtyping, and records, these benchmarks embody many aspects of programming languages that are challenging to formalize: variable binding at both the term and type levels, syntactic forms with variable numbers of components (including binders), and proofs demanding complex induction principles. We hope that these benchmarks wil

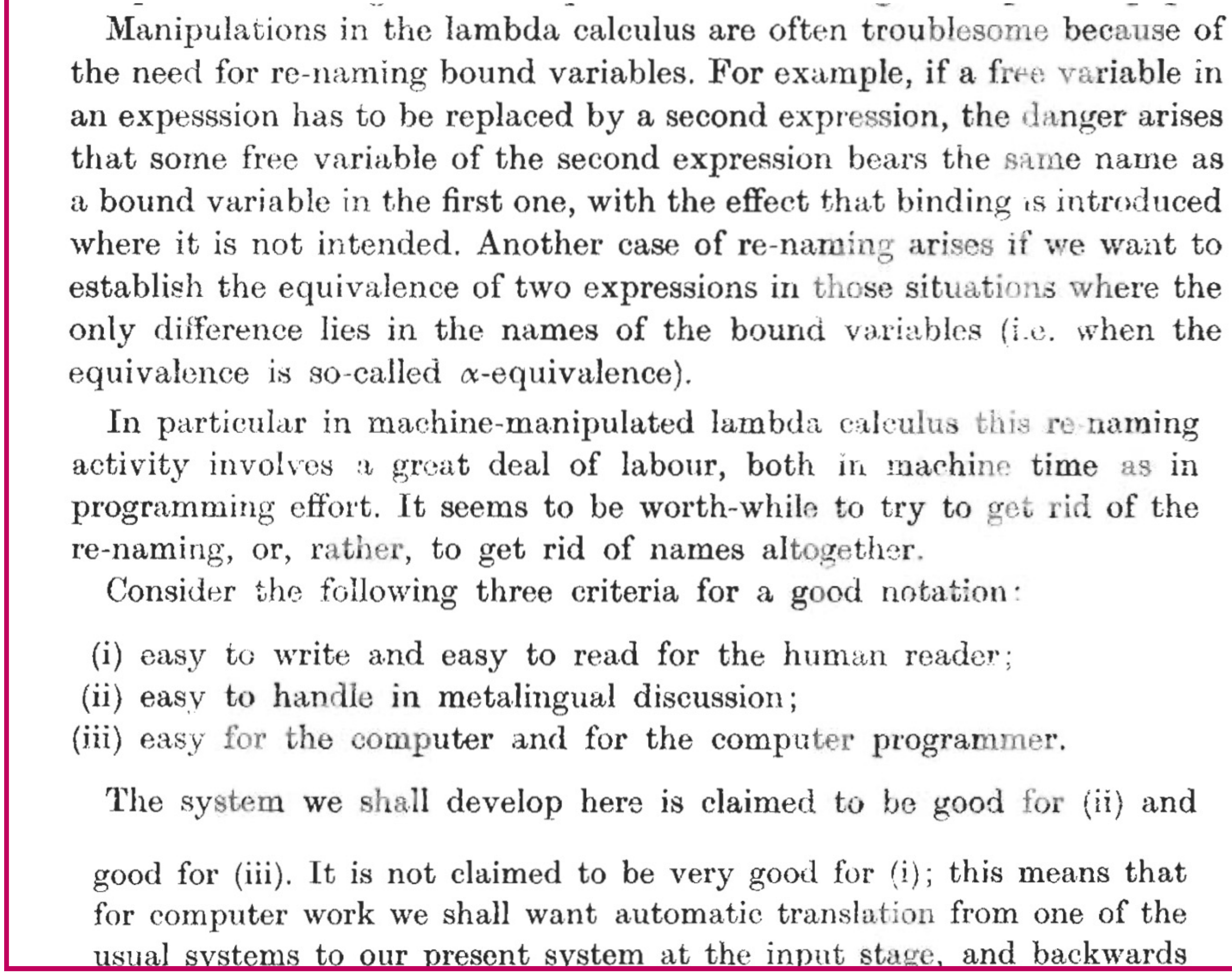
for comparing c

1 Introductio

Many proofs about
dious, with just a fe
agement of many det
mistakes or overlook
are amplified as lang
consistent, to reuse work, and to ensure tight relationships between theory and

Our conclusion from these experiments is that the relevant technology has developed *almost* to the point where it can be widely used by language researchers. We seek to push it over the threshold, making the use of proof tools common practice in programming language research—mechanized metatheory for the masses.

De Bruijn Syntax



Manipulations in the lambda calculus are often troublesome because of the need for re-naming bound variables. For example, if a free variable in an expesssion has to be replaced by a second expression, the danger arises that some free variable of the second expression bears the same name as a bound variable in the first one, with the effect that binding is introduced where it is not intended. Another case of re-naming arises if we want to establish the equivalence of two expressions in those situations where the only difference lies in the names of the bound variables (i.e. when the equivalence is so-called α-equivalence).

In particular in machine-manipulated lambda calculus this re-naming activity involves a great deal of labour, both in machine time as in programming effort. It seems to be worth-while to try to get rid of the re-naming, or, rather, to get rid of names altogether.

Consider the following three criteria for a good notation:

(i) easy to write and easy to read for the human reader;
(ii) easy to handle in metalingual discussion;
(iii) easy for the computer and for the computer programmer.

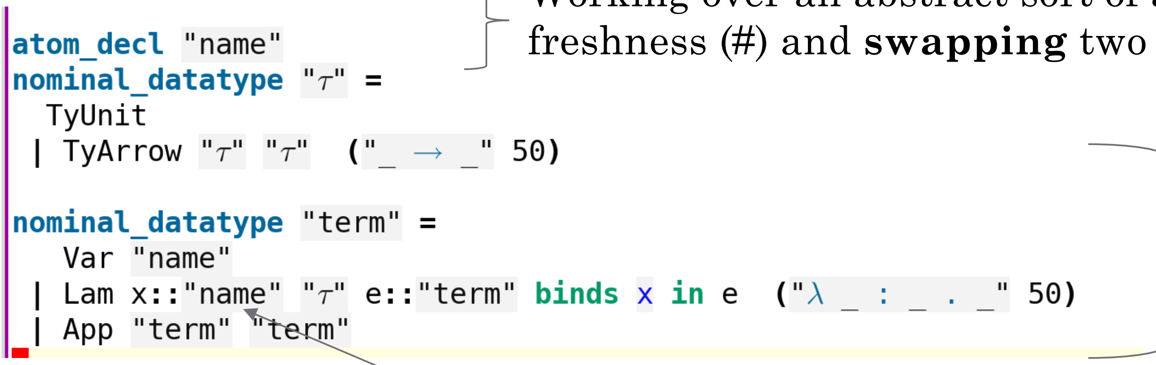
The system we shall develop here is claimed to be good for (ii) and

good for (iii). It is not claimed to be very good for (i); this means that for computer work we shall want automatic translation from one of the usual systems to our present system at the input stage, and backwards

De Bruijn, Nicolaas Govert. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem." 1972

Nominal Techniques

Equivariance:
A definition commutes with permutation.



```
atom_decl "name"
nominal_datatype "τ" =
  TyUnit
 | TyArrow "τ" "τ"  ("_ → _" 50)

nominal_datatype "term" =
   Var "name"
 | Lam x::"name" "τ" e::"term" binds x in e  ("λ _ : _ . _" 50)
 | App "term" "term"
```

Working over an abstract sort of **atoms** - allowing freshness (#) and **swapping** two atoms

Ensures that the datatype respects equivariance

Abstracting over a new name

$$supp\ x \overset{def}{=} \{a \mid infinite\ \{b \mid (a\ b) \bullet x \neq x\}\}$$

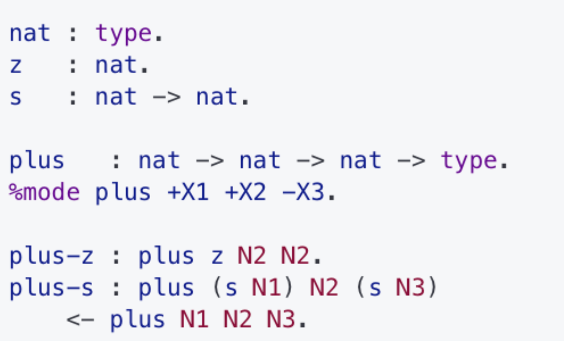
A notion of support:
A finite set of variables that the definition may contain

There is also the derived notion for when an atom a is *fresh* for an x, defined as

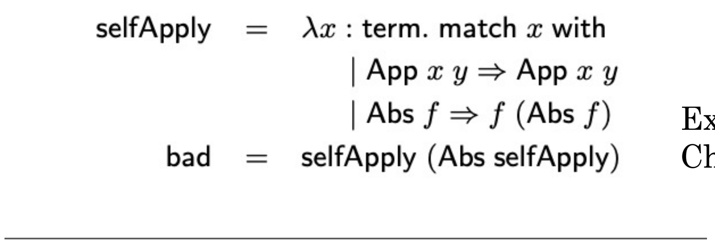
$$a \# x \overset{def}{=} a \notin supp\ x$$

Nominal Logic, a first-oder theory of names and binders – Pitts 2003.
Nominal Unification, Urban, Pitts, Gabbay – 2004.
Nominal Techniques in Isabelle/HOL. Urban/Tasson 2005.
General Bindings and Alpha-Equivalence in Nominal Isabelle, Urban/Kaliszyk '12

Higher-Order Abstract Syntax



```
nat : type.
z   : nat.
s   : nat -> nat.

plus   : nat -> nat -> nat -> type.
%mode plus +X1 +X2 -X3.

plus-z : plus z N2 N2.
plus-s : plus (s N1) N2 (s N3)
    <- plus N1 N2 N3.
```

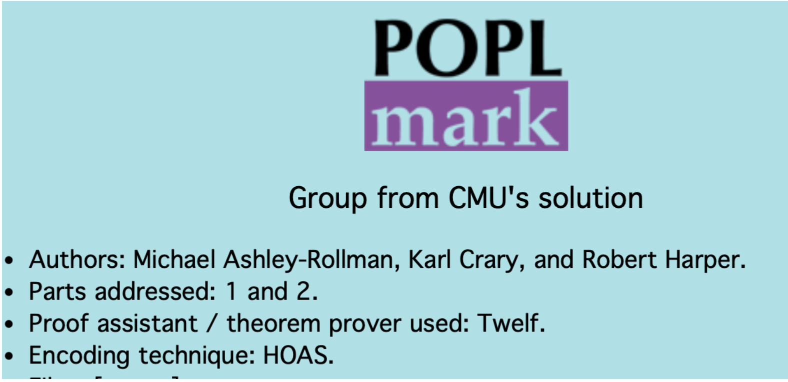


$$
\begin{array}{rcl}
\mathsf{selfApply} & = & \lambda x : \mathsf{term}.\ \mathsf{match}\ x\ \mathsf{with} \\
 & & \quad \mid \mathsf{App}\ x\ y \Rightarrow \mathsf{App}\ x\ y \\
 & & \quad \mid \mathsf{Abs}\ f \Rightarrow f\ (\mathsf{Abs}\ f) \\
\mathsf{bad} & = & \mathsf{selfApply}\ (\mathsf{Abs}\ \mathsf{selfApply})
\end{array}
$$

Figure 1. An example divergent term

Example from: Chlipala, PHOAS

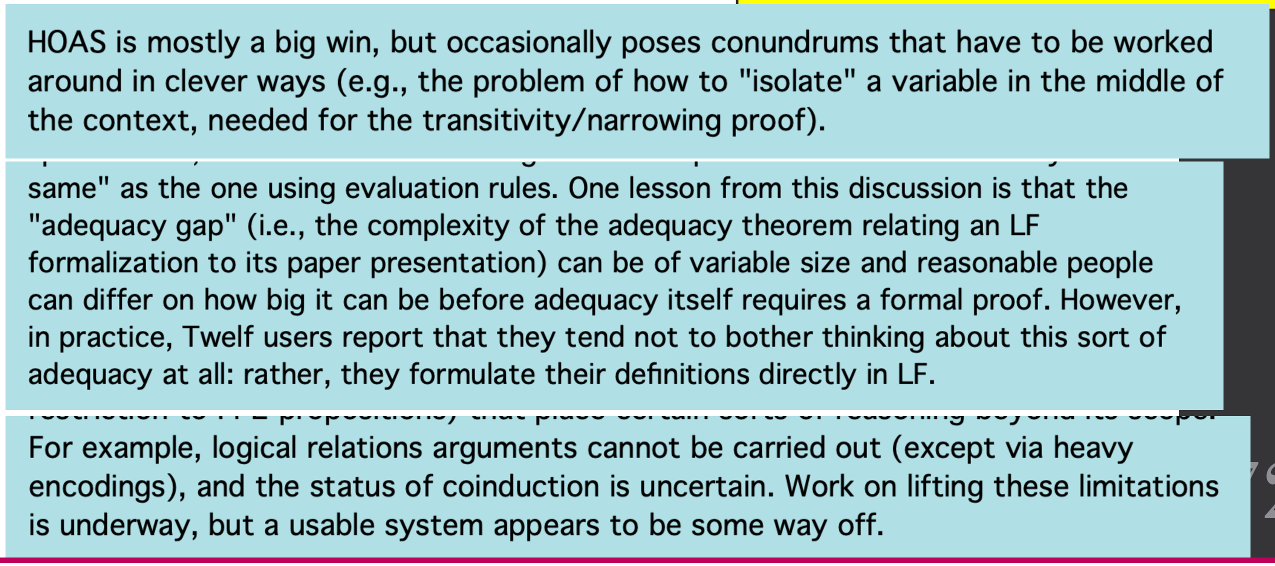
- Substitutions/substitutivity for free
- No explicit variable constructor/since variables are represented as metalevel variables they are implicit/we cannot write definitions which mention them explicitly
- Impossible to use in a general-purpose proof assistant
- Restrictions on recursive functions



POPL mark

Group from CMU's solution

- Authors: Michael Ashley-Rollman, Karl Crary, and Robert Harper.
- Parts addressed: 1 and 2.
- Proof assistant / theorem prover used: Twelf.
- Encoding technique: HOAS.

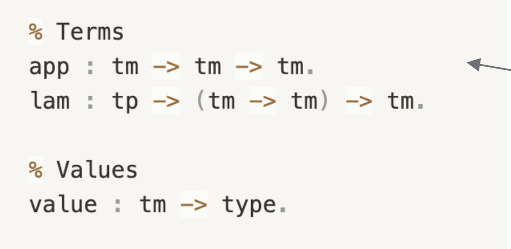


HOAS is mostly a big win, but occasionally poses conundrums that have to be worked around in clever ways (e.g., the problem of how to "isolate" a variable in the middle of the context, needed for the transitivity/narrowing proof).

same" as the one using evaluation rules. One lesson from this discussion is that the "adequacy gap" (i.e., the complexity of the adequacy theorem relating an LF formalization to its paper presentation) can be of variable size and reasonable people can differ on how big it can be before adequacy itself requires a formal proof. However, in practice, Twelf users report that they tend not to bother thinking about this sort of adequacy at all: rather, they formulate their definitions directly in LF.

restriction to Π2 propositions) that place certain sorts of reasoning beyond its scope. For example, logical relations arguments cannot be carried out (except via heavy encodings), and the status of coinduction is uncertain. Work on lifting these limitations is underway, but a usable system appears to be some way off.

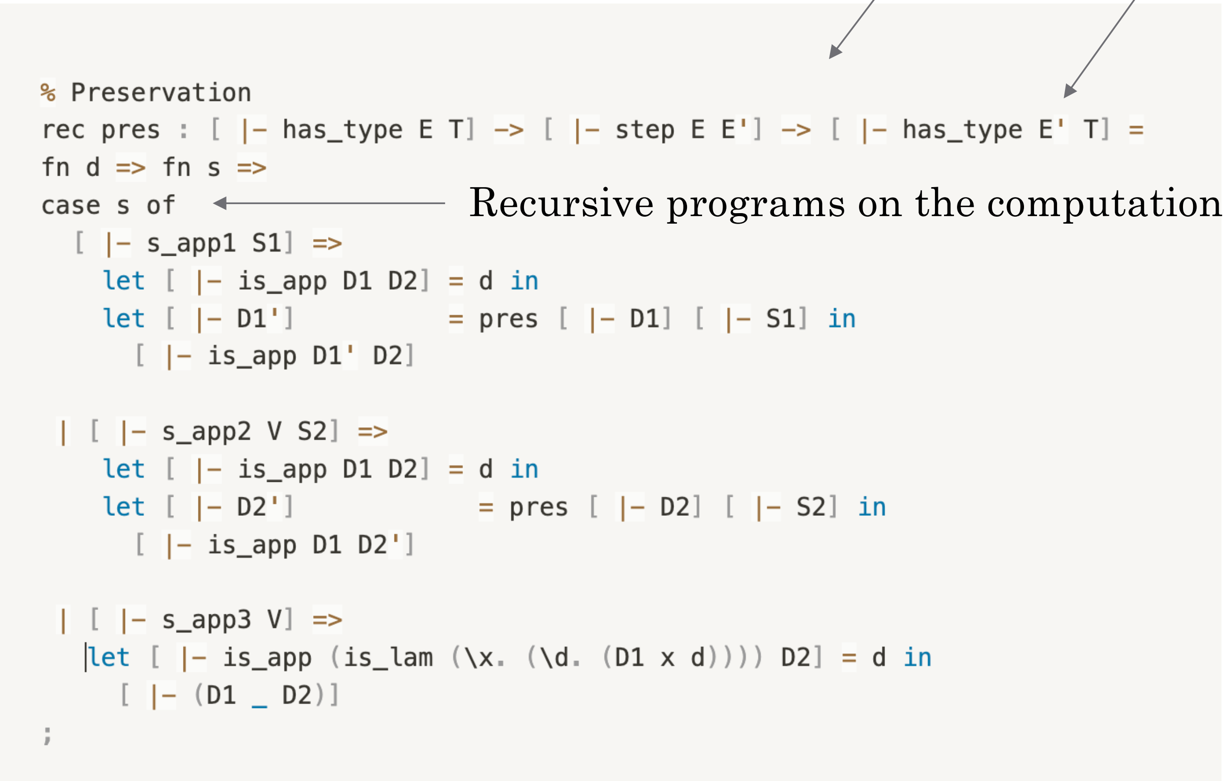
Beluga/Contextual Modal Type Theory



```
% Terms
app : tm -> tm -> tm.
lam : tp -> (tm -> tm) -> tm.

% Values
value : tm -> type.
```

Distinguishes data and computations:

Boxed value: embedded into computations; no computations inside



```
% Preservation
rec pres : [ |- has_type E T] -> [ |- step E E'] -> [ |- has_type E' T] =
fn d => fn s =>
case s of
  [ |- s_app1 S1] =>
    let [ |- is_app D1 D2] = d in
    let [ |- D1']           = pres [ |- D1] [ |- S1] in
      [ |- is_app D1' D2]

 | [ |- s_app2 V S2] =>
    let [ |- is_app D1 D2] = d in
    let [ |- D2']             = pres [ |- D2] [ |- S2] in
      [ |- is_app D1 D2']

 | [ |- s_app3 V] =>
    let [ |- is_app (is_lam (\x. (\d. (D1 x d)))) D2] = d in
     [ |- (D1 _ D2)]
;
```

Recursive programs on the computation level

- Extension of HOAS with a modality to talk about open terms => expressivity
- Comes with a notion of context morphisms

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. „Contextual Modal Type Theory“ 2008

Pientka/Dunfield: **Beluga**: A framework for programming and reasoning with deductive systems 2010

Comparison with de Bruijn proofs:

Kaiser, Jonas, Brigitte Pientka, and Gert Smolka. "Relating system F and Lambda2: A case study in Coq, Abella and Beluga." 2017

POPLMark Reloaded Challenge

Parametric Higher-Order Abstract Syntax (PHOAS)

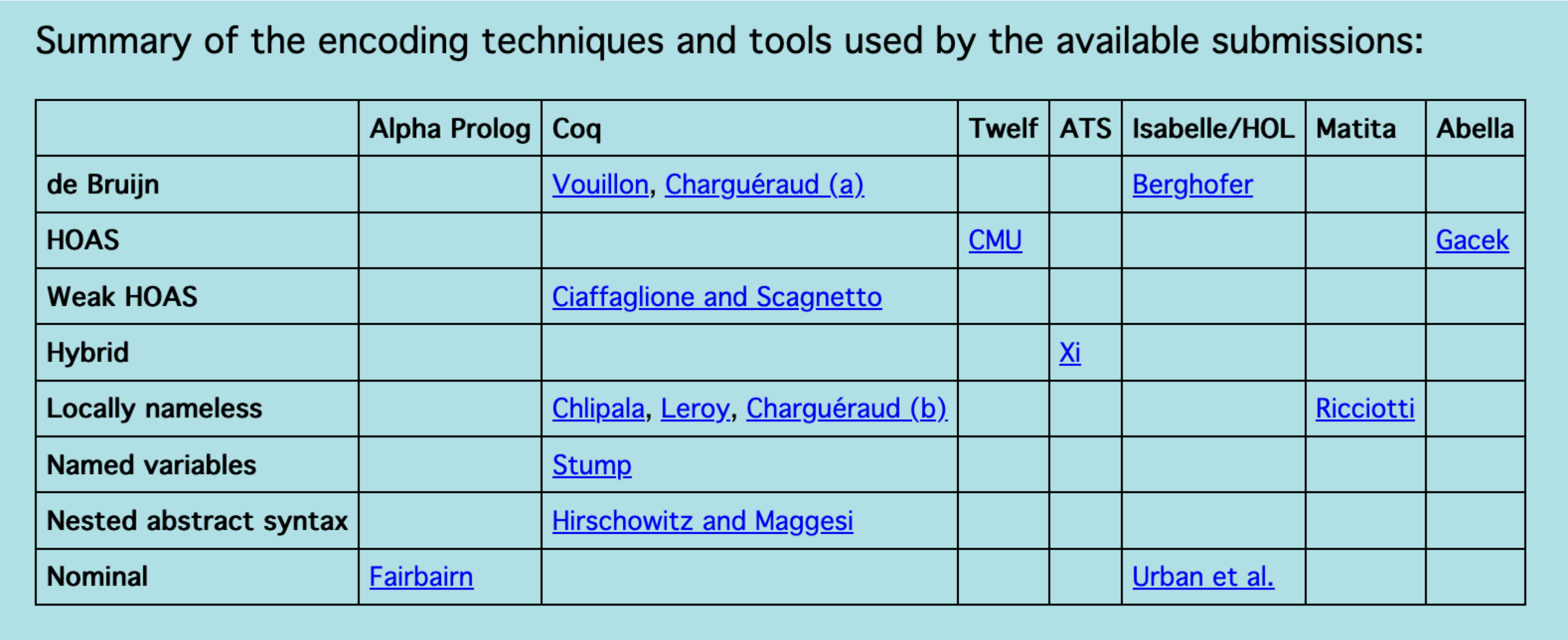
```
Section tm.

Variable var : Type.

Inductive tm : Type :=
| Var : var -> tm
| App' : tm -> tm -> tm
| Abs' : (var -> tm) -> tm.

End tm.

Definition Tm := forall X, tm X.
```

```
Fixpoint count (e: tm unit) : nat :=
match e with
| Var _ x => 1
| App' _ s t => count s + count t
| Abs' _ s => count (s tt) end.

Definition Count (e : Tm) : nat :=
count (e unit).
```

Washburn, Geoffrey, and Stephanie Weirich. "Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism." *ACM SIGPLAN Notices* 38.9 (2003): 249-262.

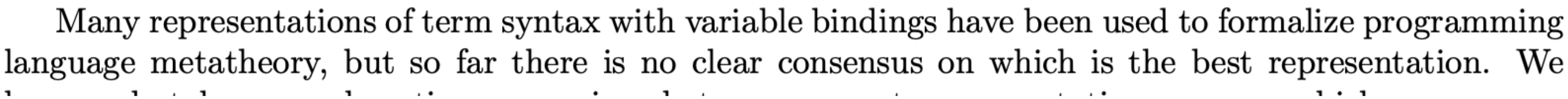
Chlipala, Adam. "Parametric higher-order abstract syntax for mechanized semantics." *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*. 2008.

Proof of adequacy: Atkey, Syntax For Free: Representing Syntax with Binding using Parametricity

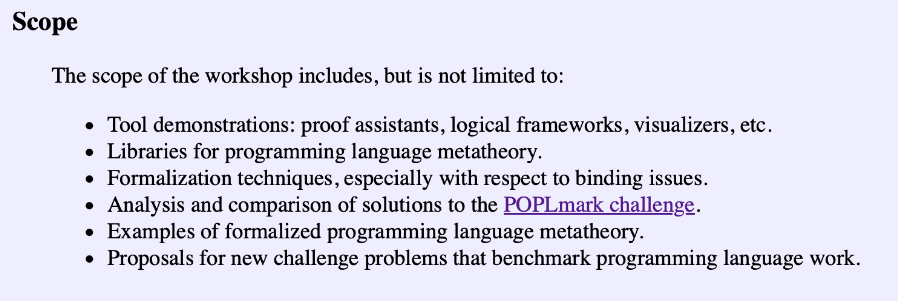


Summary of the encoding techniques and tools used by the available submissions:

	Alpha Prolog	Coq	Twelf	ATS	Isabelle/HOL	Matita	Abella
de Bruijn		Vouillon, Charguéraud (a)			Berghofer		
HOAS			CMU				Gacek
Weak HOAS		Ciaffaglione and Scagnetto					
Hybrid				Xi			
Locally nameless		Chlipala, Leroy, Charguéraud (b)				Ricciotti	
Named variables		Stump					
Nested abstract syntax		Hirschowitz and Maggesi					
Nominal	Fairbairn				Urban et al.		



Many representations of term syntax with variable bindings have been used to formalize programming language metatheory, but so far there is no clear consensus on which is the best representation. We

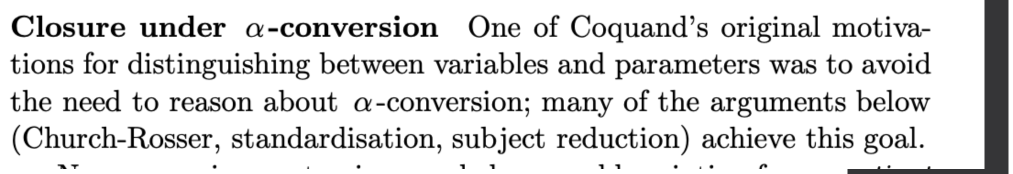


Scope

The scope of the workshop includes, but is not limited to:

- Tool demonstrations: proof assistants, logical frameworks, visualizers, etc.
- Libraries for programming language metatheory.
- Formalization techniques, especially with respect to binding issues.
- Analysis and comparison of solutions to the POPLmark challenge.
- Examples of formalized programming language metatheory.
- Proposals for new challenge problems that benchmark programming language work.

Locally Named Syntax



Closure under α-conversion One of Coquand's original motivations for distinguishing between variables and parameters was to avoid the need to reason about α-conversion; many of the arguments below (Church-Rosser, standardisation, subject reduction) achieve this goal.

```
Inductive typ : Set :=
| typ_base : typ
| typ_arrow : typ -> typ -> typ.
```

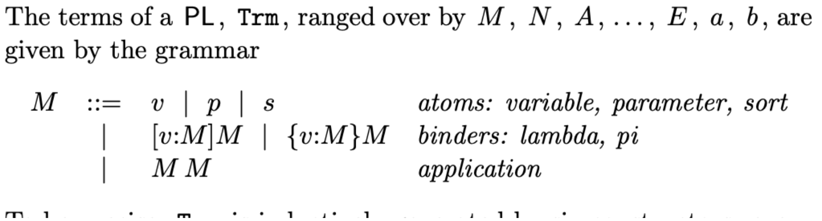
```
Inductive exp : Set :=
| bvar : name -> exp (* bound variables *)
| fvar : name -> exp (* free variables *)
| abs : binder -> exp -> exp
| app : exp -> exp -> exp.
```

Example term: $\lambda\, x. z\ (\lambda\, y. (x\ y)\ z)$

```
abs “x” (app (fvar “z”) (lam “y” (app (app (bvar
x”) (bvar “y”)) (fvar “y”))))
```

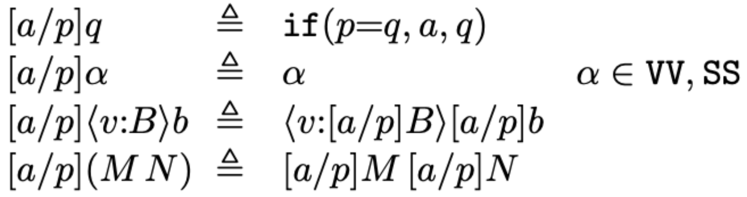
Some Lambda Calculus and Type Theory Formalized, McKinna/Pollack ‘99

Some Lambda Calculus and Type Theory Formalized, McKinna/Pollack ‘99



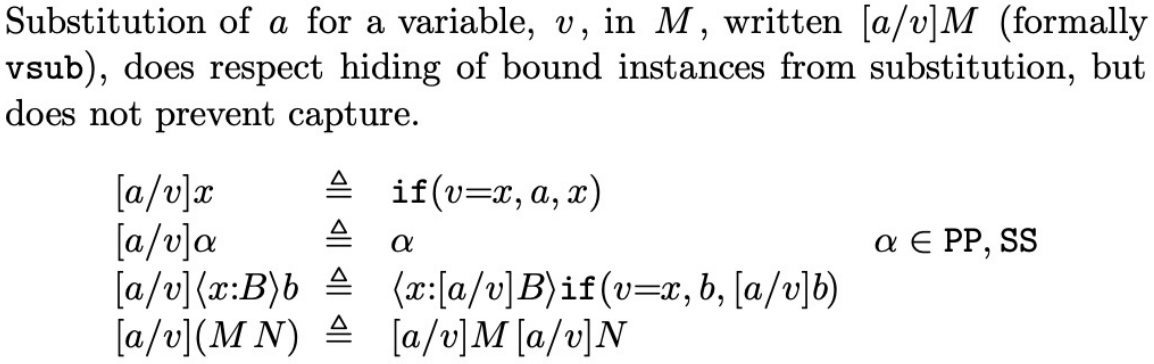
The terms of a PL, Trm, ranged over by M, N, A, …, E, a, b, are given by the grammar

$$\begin{array}{lcll} M & ::= & v \mid p \mid s & \textit{atoms: variable, parameter, sort} \\ & \mid & [v{:}M]M \mid \{v{:}M\}M & \textit{binders: lambda, pi} \\ & \mid & M\,M & \textit{application} \end{array}$$



$$\begin{array}{lcll} [a/p]q & \triangleq & \mathtt{if}(p{=}q, a, q) & \\ [a/p]\alpha & \triangleq & \alpha & \alpha \in \mathsf{VV}, \mathsf{SS} \\ [a/p]\langle v{:}B\rangle b & \triangleq & \langle v{:}[a/p]B\rangle [a/p]b & \\ [a/p](M\,N) & \triangleq & [a/p]M\,[a/p]N & \end{array}$$

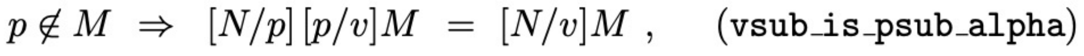
Substitution of parameters
=> No binding instances in terms!



Substitution of a for a variable, v, in M, written $[a/v]M$ (formally **vsub**), does respect hiding of bound instances from substitution, but does not prevent capture.

$$\begin{array}{lcll} [a/v]x & \triangleq & \mathtt{if}(v{=}x, a, x) & \\ [a/v]\alpha & \triangleq & \alpha & \alpha \in \mathsf{PP}, \mathsf{SS} \\ [a/v]\langle x{:}B\rangle b & \triangleq & \langle x{:}[a/v]B\rangle \mathtt{if}(v{=}x, b, [a/v]b) & \\ [a/v](M\,N) & \triangleq & [a/v]M\,[a/v]N & \end{array}$$

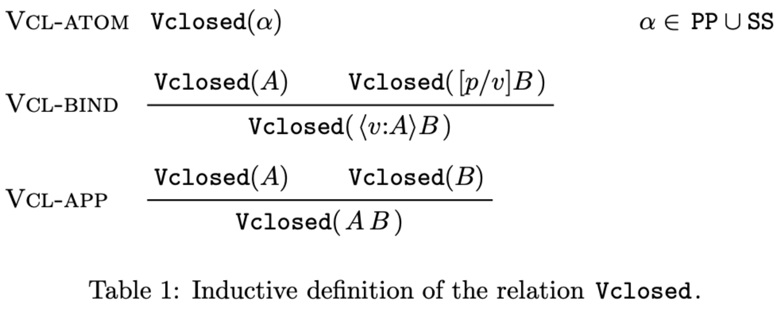
Note:
Not capture-avoiding



$$p \notin M \;\Rightarrow\; [N/p]\,[p/v]M \;=\; [N/v]M\ , \qquad (\mathtt{vsub\_is\_psub\_alpha})$$

Usual invariant:
A term is *closed*,
i.e. all bound variables are in scope.

What is a good induction principle for closed terms?

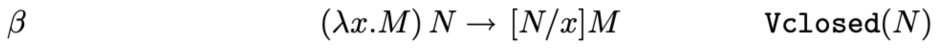


VCL-ATOM $\texttt{Vclosed}(\alpha)$ $\qquad \alpha \in \mathsf{PP} \cup \mathsf{SS}$

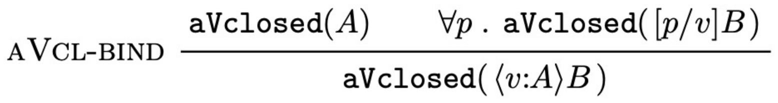
VCL-BIND $\dfrac{\texttt{Vclosed}(A) \qquad \texttt{Vclosed}(\,[p/v]B\,)}{\texttt{Vclosed}(\,\langle v{:}A\rangle B\,)}$

VCL-APP $\dfrac{\texttt{Vclosed}(A) \qquad \texttt{Vclosed}(B)}{\texttt{Vclosed}(\,A\,B\,)}$

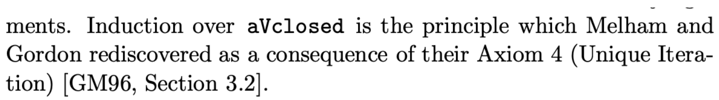
Table 1: Inductive definition of the relation `Vclosed`.



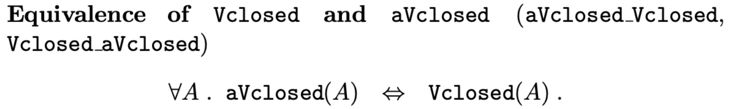
$\beta \qquad (\lambda x.M)\,N \to [N/x]M \qquad \texttt{Vclosed}(N)$



AVCL-BIND $\dfrac{\texttt{aVclosed}(A) \qquad \forall p\;.\;\texttt{aVclosed}(\,[p/v]B\,)}{\texttt{aVclosed}(\,\langle v{:}A\rangle B\,)}$



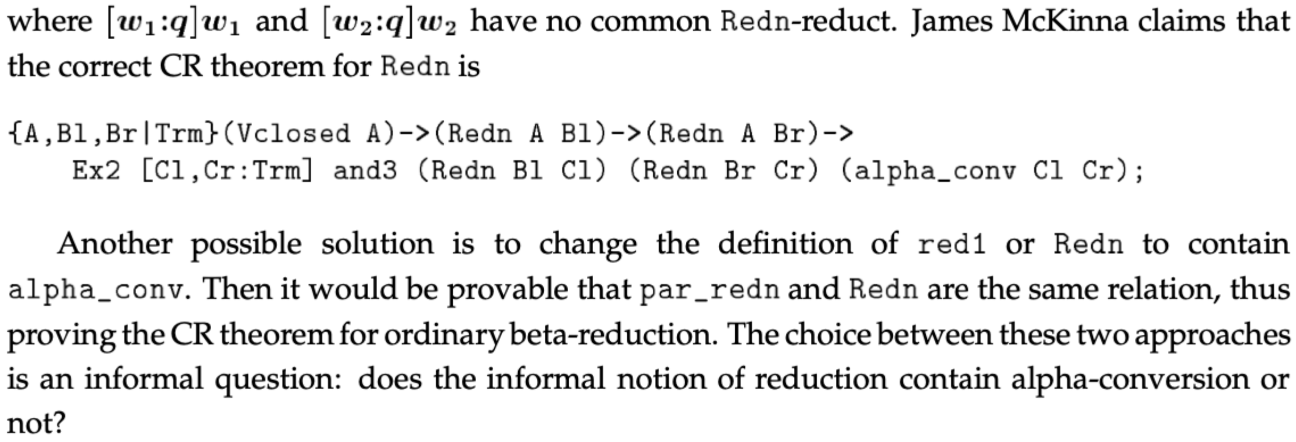
ments. Induction over `aVclosed` is the principle which Melham and Gordon rediscovered as a consequence of their Axiom 4 (Unique Iteration) [GM96, Section 3.2].



Equivalence of `Vclosed` **and** `aVclosed` (`aVclosed_Vclosed`, `Vclosed_aVclosed`)

$$\forall A\;.\;\texttt{aVclosed}(A) \;\Leftrightarrow\; \texttt{Vclosed}(A)\;.$$

Some Lambda Calculus and Type Theory Formalized, McKinna/Pollack ‘99



where $[\boldsymbol{w}_1{:}\boldsymbol{q}]\boldsymbol{w}_1$ and $[\boldsymbol{w}_2{:}\boldsymbol{q}]\boldsymbol{w}_2$ have no common `Redn`-reduct. James McKinna claims that the correct CR theorem for `Redn` is

```
{A,Bl,Br|Trm}(Vclosed A)->(Redn A Bl)->(Redn A Br)->
     Ex2 [Cl,Cr:Trm] and3 (Redn Bl Cl) (Redn Br Cr) (alpha_conv Cl Cr);
```

Another possible solution is to change the definition of `red1` or `Redn` to contain `alpha_conv`. Then it would be provable that `par_redn` and `Redn` are the same relation, thus proving the CR theorem for ordinary beta-reduction. The choice between these two approaches is an informal question: does the informal notion of reduction contain alpha-conversion or not?

Pollack, Robert. *The Theory of LEGO*. Diss. University of Edinburgh, 1995.

β-reduction has the diamond property only up to α conversion

Locally Nameless Syntax

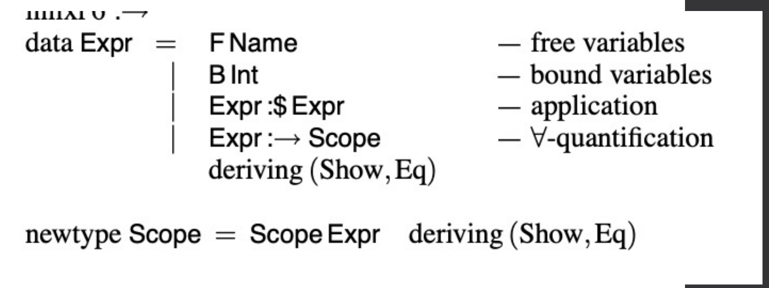
```
Inductive typ : Set :=
| typ_base : typ
| typ_arrow : typ -> typ -> typ.
```

```
Inductive exp : Set :=
| bvar : nat -> exp (* bound variables *)
| fvar : name -> exp (* free variables *)
| abs : exp -> exp
| app : exp -> exp -> exp.
```

Example term: $\lambda\, x. z\; (\lambda\, y. (x\; y)\; z)$

```
abs (app (fvar z) (lam (app (app (bvar 1) (bvar
0)) (fvar z))))
```

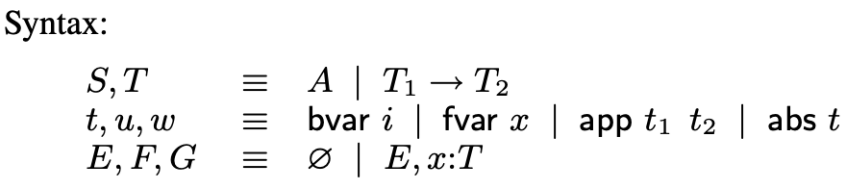
Usual invariant:
A term is *locally closed*,
i.e. all bound variables are in scope.



```
infixr 6 :→
data Expr  =   F Name              — free variables
           |   B Int               — bound variables
           |   Expr :$ Expr        — application
           |   Expr :→ Scope       — ∀-quantification
               deriving (Show, Eq)

newtype Scope  =  Scope Expr   deriving (Show, Eq)
```

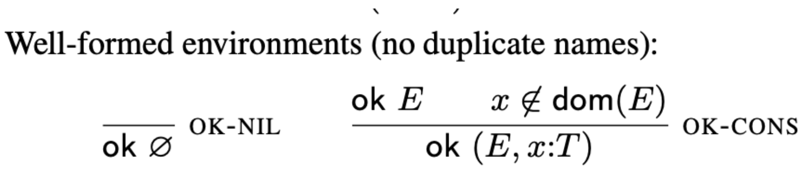
Explicit distinction in McBride/McKinna ‘04

G. Huet. The Constructive Engine. ‘89
R. Pollack. Closure under alpha-conversion. ‘93
I am not a number- I am a free variable – Conor McBride, James McKinna ‘04
Aydemir et al., Engineering Formal Metatheory ‘08



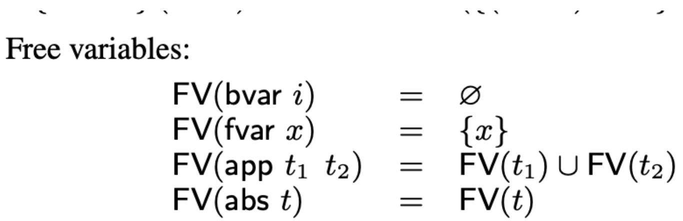
Syntax:

$$\begin{array}{lcl} S, T & \equiv & A \mid T_1 \to T_2 \\ t, u, w & \equiv & \mathsf{bvar}\ i \mid \mathsf{fvar}\ x \mid \mathsf{app}\ t_1\ t_2 \mid \mathsf{abs}\ t \\ E, F, G & \equiv & \varnothing \mid E, x{:}T \end{array}$$



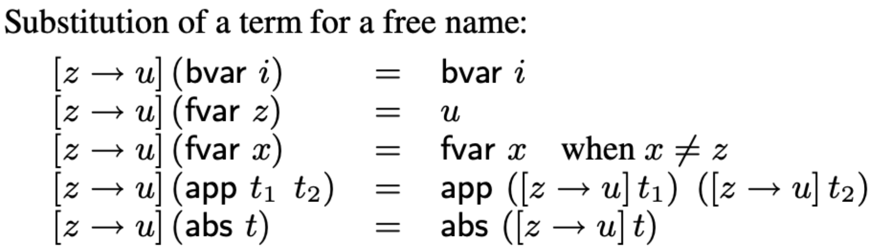
Well-formed environments (no duplicate names):

$$\frac{}{\mathsf{ok}\ \varnothing}\ \text{OK-NIL} \qquad \frac{\mathsf{ok}\ E \qquad x \notin \mathsf{dom}(E)}{\mathsf{ok}\ (E, x{:}T)}\ \text{OK-CONS}$$



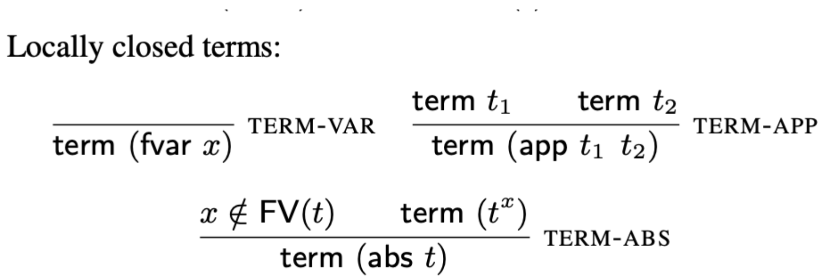
Free variables:

$$\begin{array}{lcl} \mathsf{FV}(\mathsf{bvar}\ i) & = & \varnothing \\ \mathsf{FV}(\mathsf{fvar}\ x) & = & \{x\} \\ \mathsf{FV}(\mathsf{app}\ t_1\ t_2) & = & \mathsf{FV}(t_1) \cup \mathsf{FV}(t_2) \\ \mathsf{FV}(\mathsf{abs}\ t) & = & \mathsf{FV}(t) \end{array}$$



Substitution of a term for a free name:

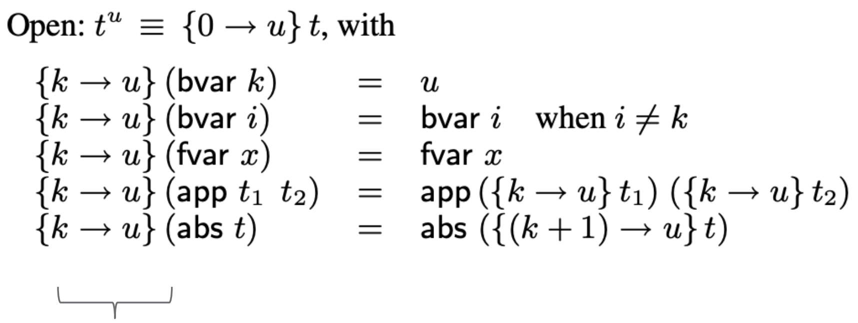
$$\begin{array}{lcl} [z \to u]\,(\mathsf{bvar}\ i) & = & \mathsf{bvar}\ i \\ [z \to u]\,(\mathsf{fvar}\ z) & = & u \\ [z \to u]\,(\mathsf{fvar}\ x) & = & \mathsf{fvar}\ x \quad \text{when } x \neq z \\ [z \to u]\,(\mathsf{app}\ t_1\ t_2) & = & \mathsf{app}\ ([z \to u]\,t_1)\ ([z \to u]\,t_2) \\ [z \to u]\,(\mathsf{abs}\ t) & = & \mathsf{abs}\ ([z \to u]\,t) \end{array}$$



Locally closed terms:

$$\frac{}{\mathsf{term}\ (\mathsf{fvar}\ x)}\ \text{TERM-VAR} \qquad \frac{\mathsf{term}\ t_1 \qquad \mathsf{term}\ t_2}{\mathsf{term}\ (\mathsf{app}\ t_1\ t_2)}\ \text{TERM-APP}$$

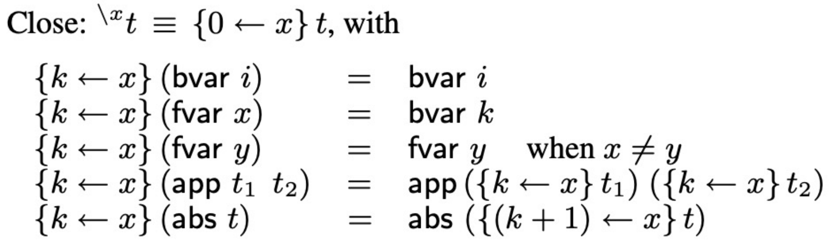
$$\frac{x \notin \mathsf{FV}(t) \qquad \mathsf{term}\ (t^x)}{\mathsf{term}\ (\mathsf{abs}\ t)}\ \text{TERM-ABS}$$



Open: $t^u \equiv \{0 \to u\}\,t$, with

$$\begin{array}{lcl} \underbrace{\{k \to u\}}\,(\mathsf{bvar}\ k) & = & u \\ \{k \to u\}\,(\mathsf{bvar}\ i) & = & \mathsf{bvar}\ i \quad \text{when } i \neq k \\ \{k \to u\}\,(\mathsf{fvar}\ x) & = & \mathsf{fvar}\ x \\ \{k \to u\}\,(\mathsf{app}\ t_1\ t_2) & = & \mathsf{app}\,(\{k \to u\}\,t_1)\ (\{k \to u\}\,t_2) \\ \{k \to u\}\,(\mathsf{abs}\ t) & = & \mathsf{abs}\ (\{(k+1) \to u\}\,t) \end{array}$$

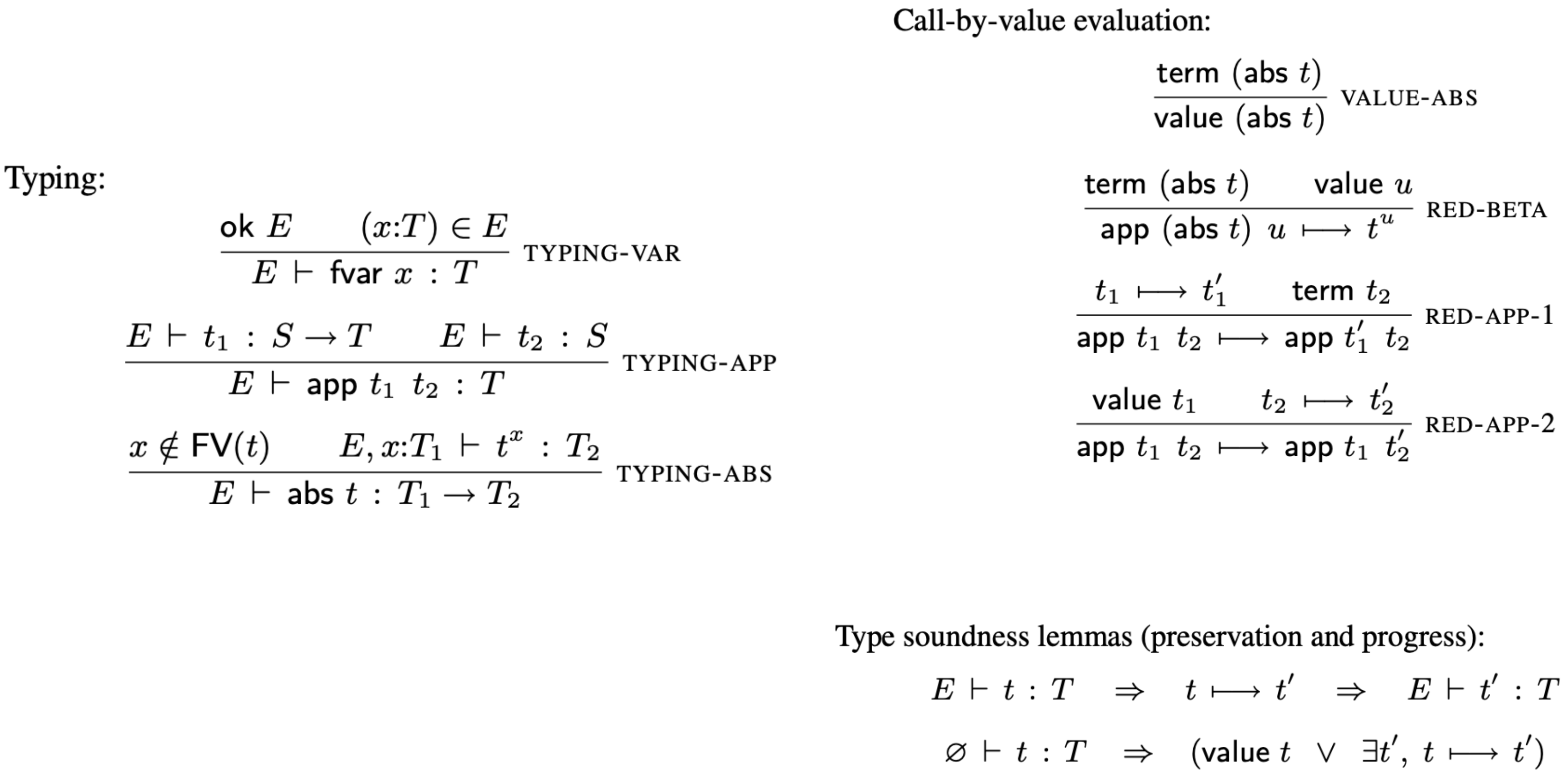
Note: Only works if 0 is the only unbound index



Close: $^{\backslash x}t \equiv \{0 \leftarrow x\}\,t$, with

$$\begin{array}{lcl} \{k \leftarrow x\}\,(\mathsf{bvar}\ i) & = & \mathsf{bvar}\ i \\ \{k \leftarrow x\}\,(\mathsf{fvar}\ x) & = & \mathsf{bvar}\ k \\ \{k \leftarrow x\}\,(\mathsf{fvar}\ y) & = & \mathsf{fvar}\ y \quad \text{when } x \neq y \\ \{k \leftarrow x\}\,(\mathsf{app}\ t_1\ t_2) & = & \mathsf{app}\,(\{k \leftarrow x\}\,t_1)\ (\{k \leftarrow x\}\,t_2) \\ \{k \leftarrow x\}\,(\mathsf{abs}\ t) & = & \mathsf{abs}\ (\{(k+1) \leftarrow x\}\,t) \end{array}$$

From: Aydemir et al., Engineering Formal Metatheory



Typing:

$$\frac{\mathsf{ok}\ E \qquad (x{:}T) \in E}{E \vdash \mathsf{fvar}\ x\ :\ T}\ \textsc{typing-var}$$

$$\frac{E \vdash t_1\ :\ S \rightarrow T \qquad E \vdash t_2\ :\ S}{E \vdash \mathsf{app}\ t_1\ \ t_2\ :\ T}\ \textsc{typing-app}$$

$$\frac{x \notin \mathsf{FV}(t) \qquad E, x{:}T_1 \vdash t^x\ :\ T_2}{E \vdash \mathsf{abs}\ t\ :\ T_1 \rightarrow T_2}\ \textsc{typing-abs}$$

Call-by-value evaluation:

$$\frac{\mathsf{term}\ (\mathsf{abs}\ t)}{\mathsf{value}\ (\mathsf{abs}\ t)}\ \textsc{value-abs}$$

$$\frac{\mathsf{term}\ (\mathsf{abs}\ t) \qquad \mathsf{value}\ u}{\mathsf{app}\ (\mathsf{abs}\ t)\ u \longmapsto t^u}\ \textsc{red-beta}$$

$$\frac{t_1 \longmapsto t_1' \qquad \mathsf{term}\ t_2}{\mathsf{app}\ t_1\ \ t_2 \longmapsto \mathsf{app}\ t_1'\ \ t_2}\ \textsc{red-app-1}$$

$$\frac{\mathsf{value}\ t_1 \qquad t_2 \longmapsto t_2'}{\mathsf{app}\ t_1\ \ t_2 \longmapsto \mathsf{app}\ t_1\ \ t_2'}\ \textsc{red-app-2}$$

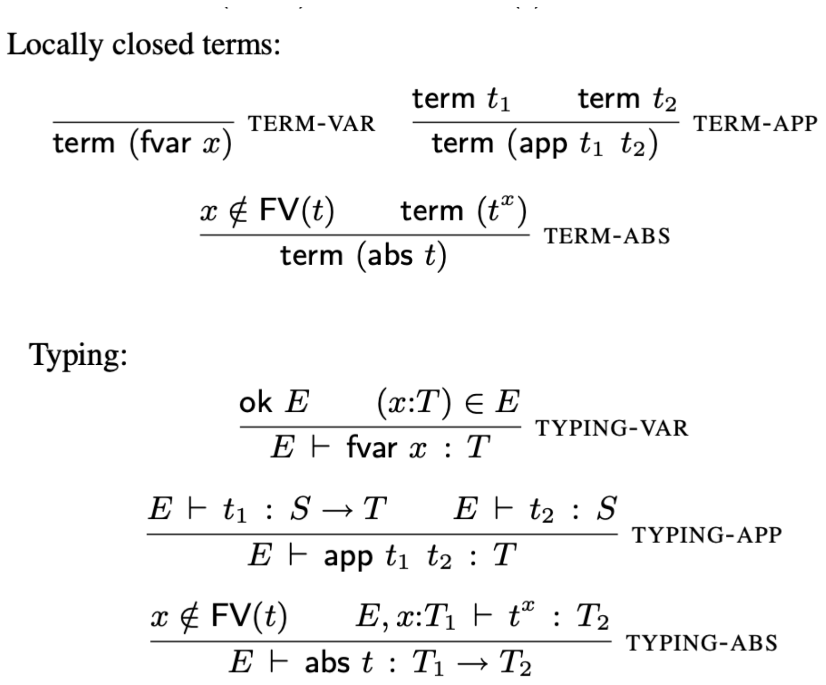
Type soundness lemmas (preservation and progress):

$$E \vdash t : T \quad \Rightarrow \quad t \longmapsto t' \quad \Rightarrow \quad E \vdash t'\ :\ T$$

$$\varnothing \vdash t\ :\ T \quad \Rightarrow \quad (\mathsf{value}\ t\ \ \vee\ \ \exists t',\ t \longmapsto t')$$

From: Aydemir et al., Engineering Formal Metatheory

Cofinite Quantification



Locally closed terms:

$$\frac{}{\mathsf{term}\ (\mathsf{fvar}\ x)}\ \textsc{term-var} \qquad \frac{\mathsf{term}\ t_1 \qquad \mathsf{term}\ t_2}{\mathsf{term}\ (\mathsf{app}\ t_1\ t_2)}\ \textsc{term-app}$$

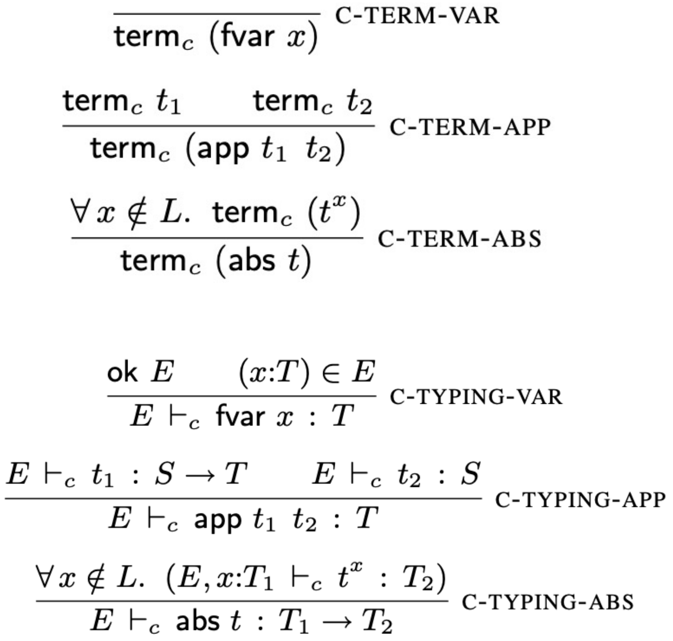
$$\frac{x \notin \mathsf{FV}(t) \qquad \mathsf{term}\ (t^x)}{\mathsf{term}\ (\mathsf{abs}\ t)}\ \textsc{term-abs}$$

Typing:

$$\frac{\mathsf{ok}\ E \qquad (x{:}T) \in E}{E \vdash \mathsf{fvar}\ x\ :\ T}\ \textsc{typing-var}$$

$$\frac{E \vdash t_1\ :\ S \to T \qquad E \vdash t_2\ :\ S}{E \vdash \mathsf{app}\ t_1\ t_2\ :\ T}\ \textsc{typing-app}$$

$$\frac{x \notin \mathsf{FV}(t) \qquad E, x{:}T_1 \vdash t^x\ :\ T_2}{E \vdash \mathsf{abs}\ t\ :\ T_1 \to T_2}\ \textsc{typing-abs}$$



$$\frac{}{\mathsf{term}_c\ (\mathsf{fvar}\ x)}\ \textsc{c-term-var}$$

$$\frac{\mathsf{term}_c\ t_1 \qquad \mathsf{term}_c\ t_2}{\mathsf{term}_c\ (\mathsf{app}\ t_1\ t_2)}\ \textsc{c-term-app}$$

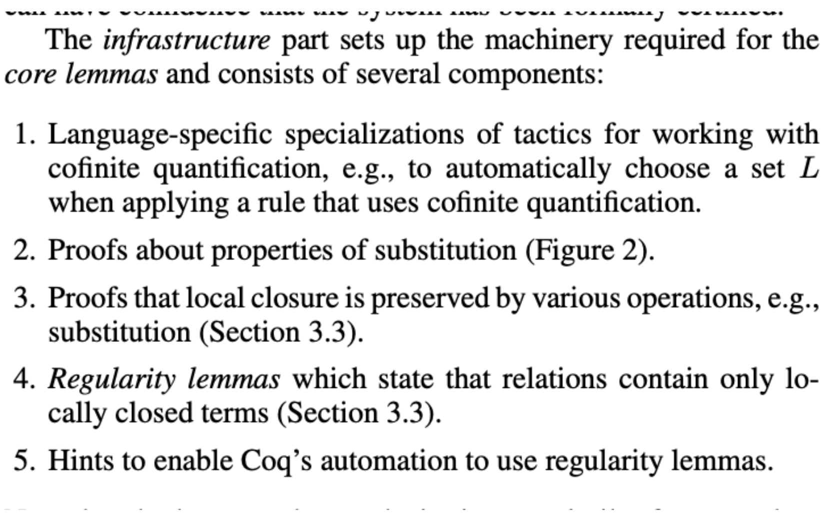
$$\frac{\forall\, x \notin L.\ \ \mathsf{term}_c\ (t^x)}{\mathsf{term}_c\ (\mathsf{abs}\ t)}\ \textsc{c-term-abs}$$

$$\frac{\mathsf{ok}\ E \qquad (x{:}T) \in E}{E \vdash_c \mathsf{fvar}\ x\ :\ T}\ \textsc{c-typing-var}$$

$$\frac{E \vdash_c t_1\ :\ S \to T \qquad E \vdash_c t_2\ :\ S}{E \vdash_c \mathsf{app}\ t_1\ t_2\ :\ T}\ \textsc{c-typing-app}$$

$$\frac{\forall\, x \notin L.\ \ (E, x{:}T_1 \vdash_c t^x\ :\ T_2)}{E \vdash_c \mathsf{abs}\ t\ :\ T_1 \to T_2}\ \textsc{c-typing-abs}$$

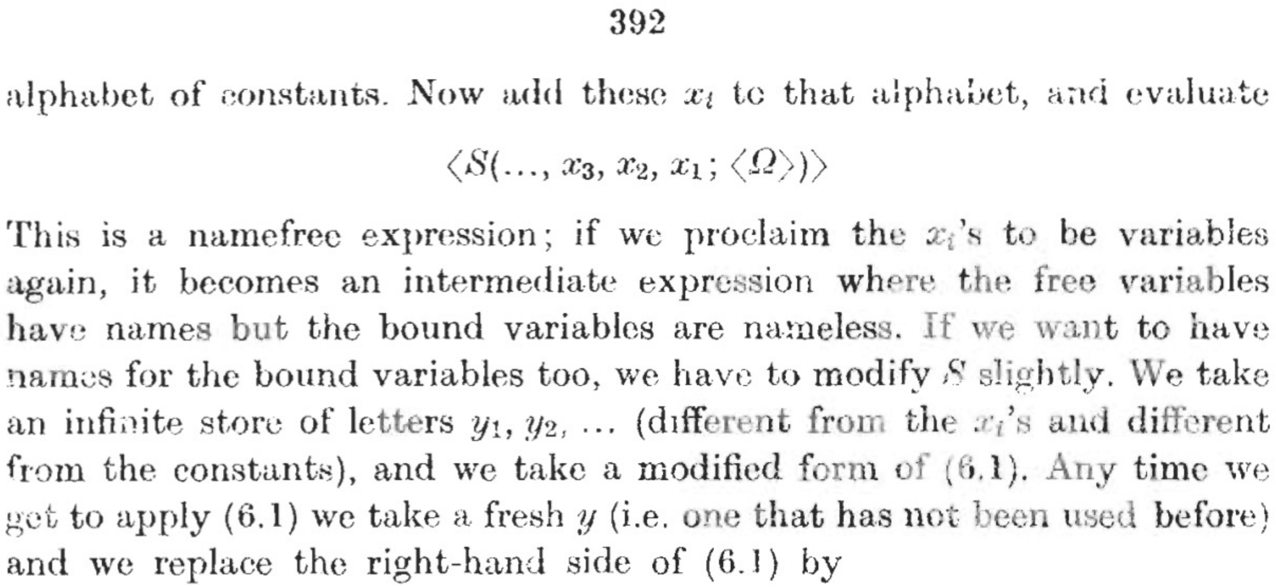
- Literature Review/Comparison of Different Versions of Locally Named/Locally Nameless:
  - Aydemir, Brian, et al. "Engineering formal metatheory." *ACM SigPlan notices* 43.1 (2008): 3-15.
- Comparison of different variants of locally nameless (different sorts, collapsed, tagged)
  - Brian Aydemir, Stephan A. Zdancewic, and Stephanie Weirich. Abstracting syntax. 2009.



can have confidence that the system has been formally certified.

The *infrastructure* part sets up the machinery required for the *core lemmas* and consists of several components:

1. Language-specific specializations of tactics for working with cofinite quantification, e.g., to automatically choose a set L when applying a rule that uses cofinite quantification.
2. Proofs about properties of substitution (Figure 2).
3. Proofs that local closure is preserved by various operations, e.g., substitution (Section 3.3).
4. *Regularity lemmas* which state that relations contain only locally closed terms (Section 3.3).
5. Hints to enable Coq's automation to use regularity lemmas.

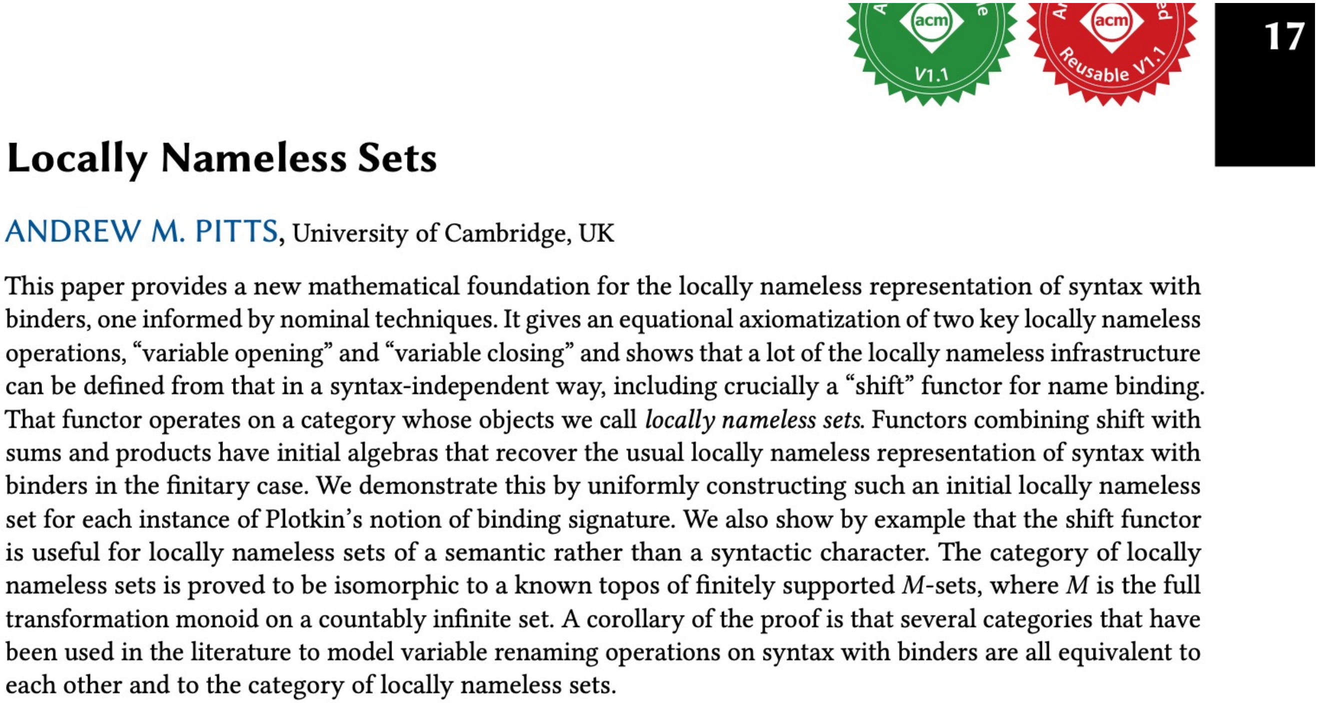


alphabet of constants. Now add these x_i to that alphabet, and evaluate

$$\langle S(\ldots, x_3, x_2, x_1; \langle \Omega \rangle)\rangle$$

This is a namefree expression; if we proclaim the x_i's to be variables again, it becomes an intermediate expression where the free variables have names but the bound variables are nameless. If we want to have names for the bound variables too, we have to modify S slightly. We take an infinite store of letters $y_1, y_2, \ldots$ (different from the x_i's and different from the constants), and we take a modified form of (6.1). Any time we get to apply (6.1) we take a fresh y (i.e. one that has not been used before) and we replace the right-hand side of (6.1) by

De Bruijn, Nicolaas Govert. "Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem." 1972

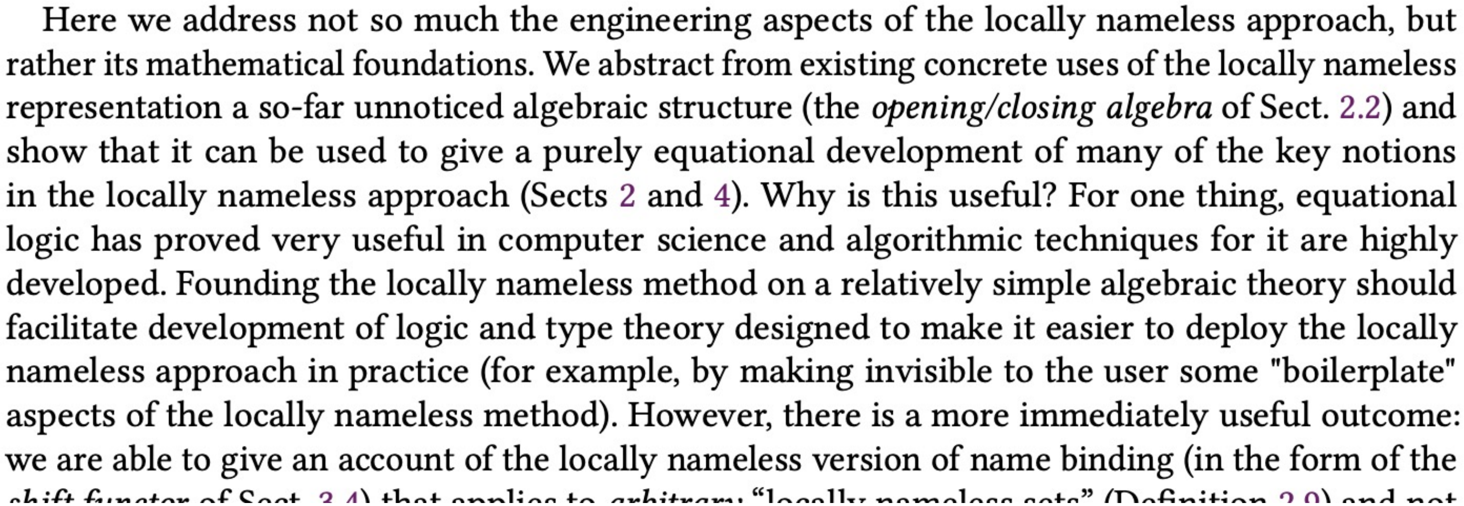


17

Locally Nameless Sets

ANDREW M. PITTS, University of Cambridge, UK

This paper provides a new mathematical foundation for the locally nameless representation of syntax with binders, one informed by nominal techniques. It gives an equational axiomatization of two key locally nameless operations, “variable opening” and “variable closing” and shows that a lot of the locally nameless infrastructure can be defined from that in a syntax-independent way, including crucially a “shift” functor for name binding. That functor operates on a category whose objects we call *locally nameless sets*. Functors combining shift with sums and products have initial algebras that recover the usual locally nameless representation of syntax with binders in the finitary case. We demonstrate this by uniformly constructing such an initial locally nameless set for each instance of Plotkin’s notion of binding signature. We also show by example that the shift functor is useful for locally nameless sets of a semantic rather than a syntactic character. The category of locally nameless sets is proved to be isomorphic to a known topos of finitely supported M-sets, where M is the full transformation monoid on a countably infinite set. A corollary of the proof is that several categories that have been used in the literature to model variable renaming operations on syntax with binders are all equivalent to each other and to the category of locally nameless sets.

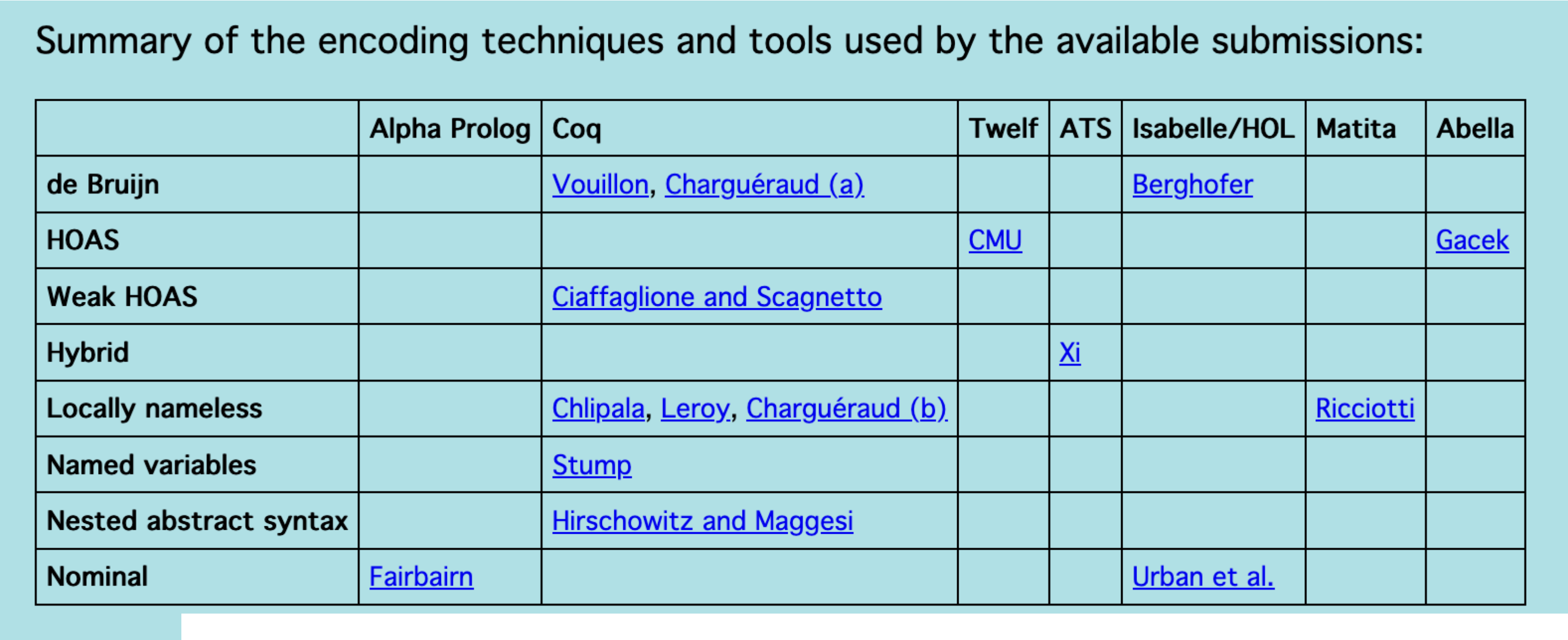


Here we address not so much the engineering aspects of the locally nameless approach, but rather its mathematical foundations. We abstract from existing concrete uses of the locally nameless representation a so-far unnoticed algebraic structure (the *opening/closing algebra* of Sect. 2.2) and show that it can be used to give a purely equational development of many of the key notions in the locally nameless approach (Sects 2 and 4). Why is this useful? For one thing, equational logic has proved very useful in computer science and algorithmic techniques for it are highly developed. Founding the locally nameless method on a relatively simple algebraic theory should facilitate development of logic and type theory designed to make it easier to deploy the locally nameless approach in practice (for example, by making invisible to the user some "boilerplate" aspects of the locally nameless method). However, there is a more immediately useful outcome: we are able to give an account of the locally nameless version of name binding (in the form of the *shift functor* of Sect. 3.4) that applies to *arbitrary* “locally nameless sets” (Definition 2.9) and not

Conclusion?

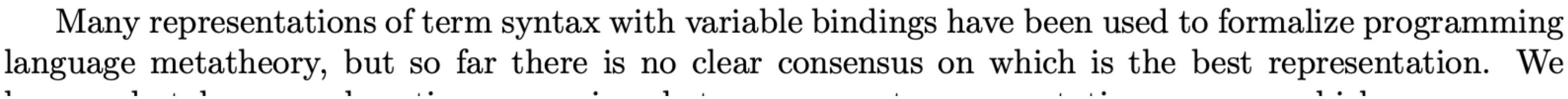
Kathrin Stark

SPLV 2024

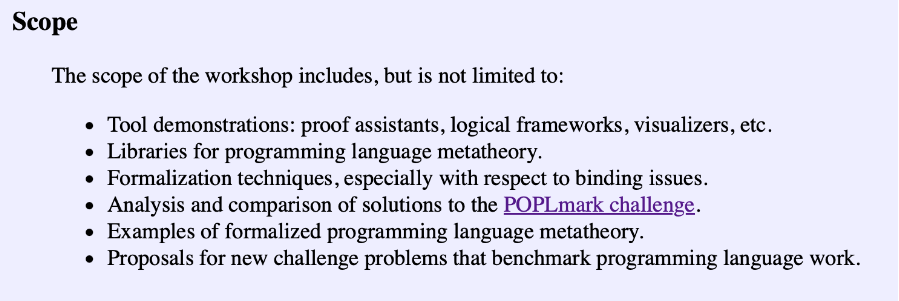


Summary of the encoding techniques and tools used by the available submissions:

	Alpha Prolog	Coq	Twelf	ATS	Isabelle/HOL	Matita	Abella
de Bruijn		Vouillon, Charguéraud (a)			Berghofer		
HOAS			CMU				Gacek
Weak HOAS		Ciaffaglione and Scagnetto					
Hybrid				Xi			
Locally nameless		Chlipala, Leroy, Charguéraud (b)				Ricciotti	
Named variables		Stump					
Nested abstract syntax		Hirschowitz and Maggesi					
Nominal	Fairbairn				Urban et al.		



Many representations of term syntax with variable bindings have been used to formalize programming language metatheory, but so far there is no clear consensus on which is the best representation. We



Scope

The scope of the workshop includes, but is not limited to:

- Tool demonstrations: proof assistants, logical frameworks, visualizers, etc.
- Libraries for programming language metatheory.
- Formalization techniques, especially with respect to binding issues.
- Analysis and comparison of solutions to the POPLmark challenge.
- Examples of formalized programming language metatheory.
- Proposals for new challenge problems that benchmark programming language work.

What to expect

- A short peek in different binder approaches:
  Pure de Bruijn, scoped de Bruijn, intrinsically typed, monadic, HOAS/CMTT, PHOAS, nominal, locally nameless

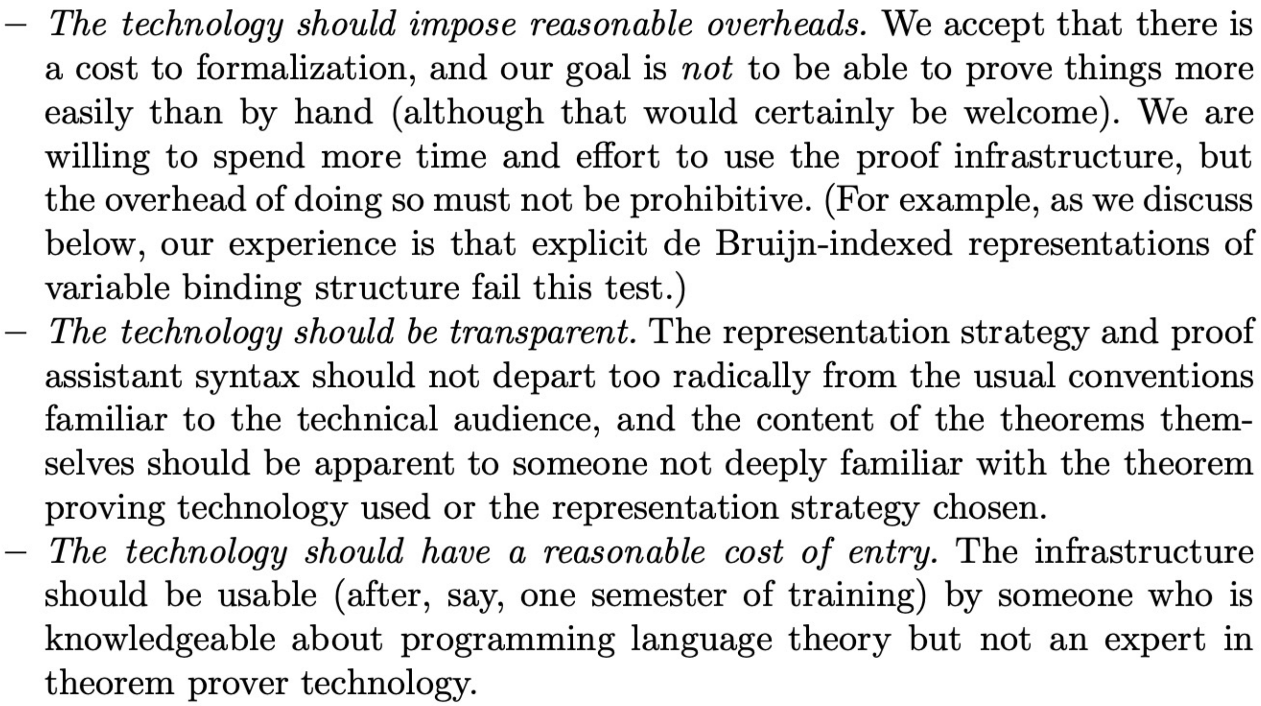
What *not* to expect:

- Completeness in any direction
- Less about tools/theoretical foundations

Some Comparisons

- Aydemir et al.: Mechanized Metatheory for the Masses: The PoplMark Challenge 2005 https://www.seas.upenn.edu/~plclub/poplmark/
- Berghofer/Urban: A Head-to-Head Comparison of de Bruijn Indices and Names 2007
- Abel et al. - POPLMark Reloaded: Mechanizing Proofs by Logical Relations 2019 https://poplmark-reloaded.github.io
- Aydemir et al. Engineering Formal Metatheory. 2008.
- Brian Aydemir, Stephan A. Zdancewic, and Stephanie Weirich. Abstracting syntax. 2009.
- https://jesper.sikanda.be/posts/1001-syntax-representations.html 2021
- Popescu, Andrei. "Nominal Recursors as Epi-Recursors." *Proceedings of the ACM on Programming Languages* 8.POPL (2024):

Criteria
POPLMark Challenge



- *The technology should impose reasonable overheads.* We accept that there is a cost to formalization, and our goal is *not* to be able to prove things more easily than by hand (although that would certainly be welcome). We are willing to spend more time and effort to use the proof infrastructure, but the overhead of doing so must not be prohibitive. (For example, as we discuss below, our experience is that explicit de Bruijn-indexed representations of variable binding structure fail this test.)
- *The technology should be transparent.* The representation strategy and proof assistant syntax should not depart too radically from the usual conventions familiar to the technical audience, and the content of the theorems themselves should be apparent to someone not deeply familiar with the theorem proving technology used or the representation strategy chosen.
- *The technology should have a reasonable cost of entry.* The infrastructure should be usable (after, say, one semester of training) by someone who is knowledgeable about programming language theory but not an expert in theorem prover technology.

	Bare	DeBr	LoNa	Nom	PHOAS	WTDB	WTDB+N	FreshL	NaPa	ASG	NomPa	CoDB	CoDB+N
First-order representation	X	X	X	X		X	X	X	X	X	X	X	X
Named variables	X		X	X	X		X	X		X	X		X
Enforces α-equivalence		X	X		X	X			X		X	X	
Enforces well-scopedness					X	X	X	X	X	X	X	X	X
No mixing of scopes							X	X		X	X		X
Enforces freshness				X				X		X	X		
Abstract interface				X					X	X	X		
Strengthening is no-op	X											X	X

https://jesper.sikanda.be/posts/1001-syntax-representations.html

- **First-order representation**: does the representation avoid the use of meta-level functions as part of the data structure? If not, it can be difficult or impossible to do things like checking equality of terms or pretty-printing.

- **Named variables**: When I write down a piece of syntax, are variables represented by their names or anonymously? This provides some measure of readability by humans.

- **Enforces α-equivalence**: Does the representation enforce that two α-equivalent terms are treated in the same way?

- **Enforces well-scopedness**: Does the representation enforce that names can only be used when they are in fact in scope?

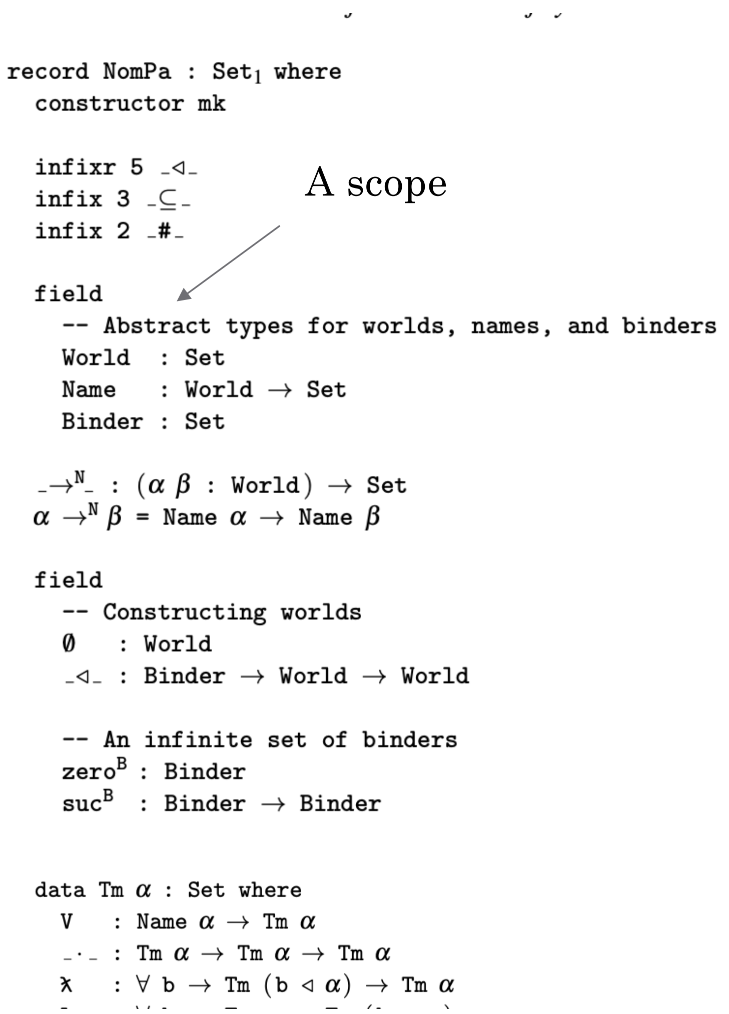
- **No mixing of scopes**: Does the representation enforce that a name that comes from one scope is not used in a different scope?

- **Abstract interface**: Does the representation provide an abstract interface that can be instantiated in different ways?

- **Enforces freshness**: Does the representation allow us to require that names must be fresh at certain positions in the syntax?

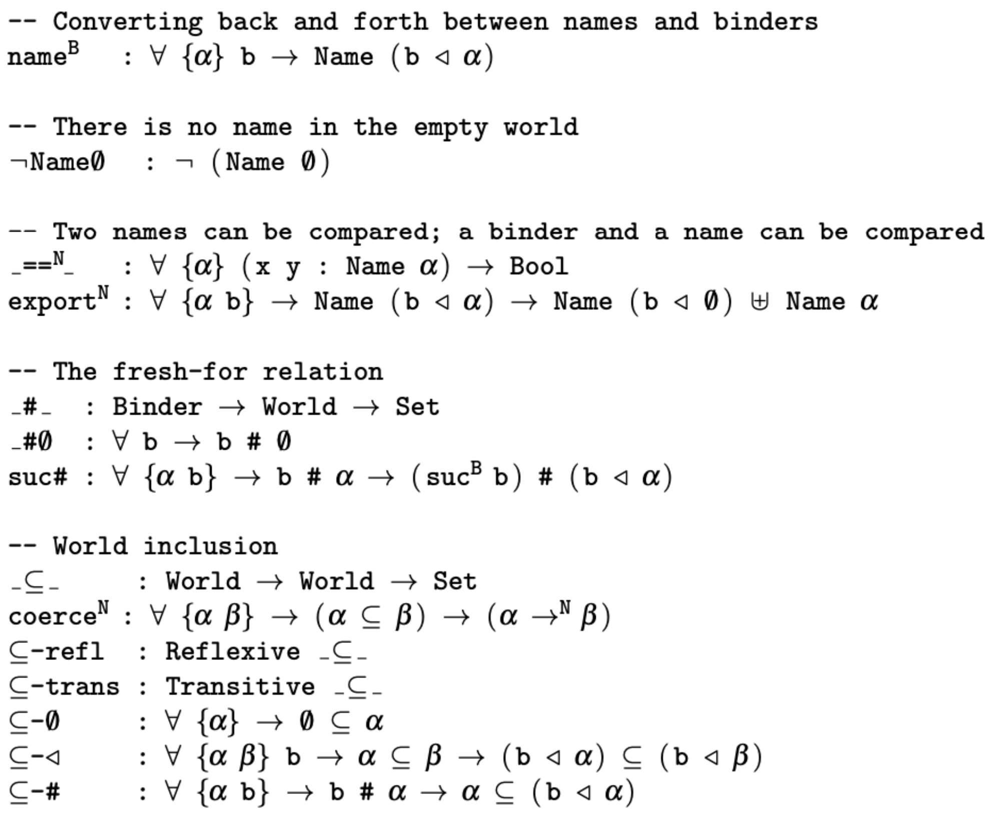
- **Strengthening is no-op**: Can we remove unused names from the scope without having to change the syntax?

Abstract Interfaces



```
record NomPa : Set_1 where
  constructor mk

  infixr 5 _◁_
  infix 3 _⊆_
  infix 2 _#_

  field
    -- Abstract types for worlds, names, and binders
    World  : Set
    Name   : World → Set
    Binder : Set

  _→^N_ : (α β : World) → Set
  α →^N β = Name α → Name β

  field
    -- Constructing worlds
    ∅   : World
    _◁_ : Binder → World → World

    -- An infinite set of binders
    zero^B : Binder
    suc^B  : Binder → Binder


  data Tm α : Set where
    V   : Name α → Tm α
    _·_ : Tm α → Tm α → Tm α
    ƛ   : ∀ b → Tm (b ◁ α) → Tm α
```

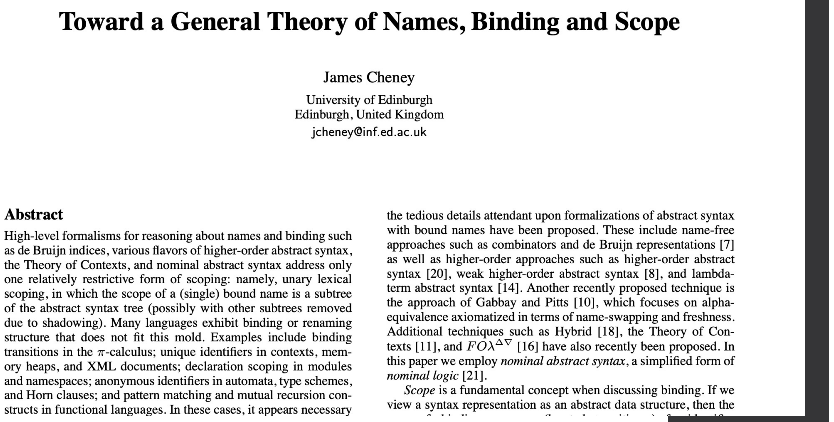
A scope



```
-- Converting back and forth between names and binders
name^B   : ∀ {α} b → Name (b ◁ α)

-- There is no name in the empty world
¬Name∅   : ¬ (Name ∅)

-- Two names can be compared; a binder and a name can be compared
_==^N_   : ∀ {α} (x y : Name α) → Bool
export^N : ∀ {α b} → Name (b ◁ α) → Name (b ◁ ∅) ⊎ Name α

-- The fresh-for relation
_#_  : Binder → World → Set
_#∅  : ∀ b → b # ∅
suc# : ∀ {α b} → b # α → (suc^B b) # (b ◁ α)

-- World inclusion
_⊆_      : World → World → Set
coerce^N : ∀ {α β} → (α ⊆ β) → (α →^N β)
⊆-refl   : Reflexive _⊆_
⊆-trans  : Transitive _⊆_
⊆-∅      : ∀ {α} → ∅ ⊆ α
⊆-◁      : ∀ {α β} b → α ⊆ β → (b ◁ α) ⊆ (b ◁ β)
⊆-#      : ∀ {α b} → b # α → α ⊆ (b ◁ α)
```

A Unified Treatment of Syntax with Binders, Pouillard/Pottier ‘12

- Which proof assistant do you use?
  - Quotients?
  - Do you require (admitting) classical reasoning?
  - Automation?
  - Are you ok with using a special-purpose proof assistant?
  - What are available tools/libraries?
  - Has somebody proven similar results; e.g. on a subset of the language?
- Do you care about proving theorems about the meta-theory or about writing terms in the language?
- How much do you care about readability?
- Do you care about performance?
- How much do you need to know about the reasoning principles?
- What kind of binders do you want to formalise?
  - Are full binders used – or just quantifiers?
  - Linear Logic?
  - Something else?
- Do you need to manipulate the context?
- Do you care about (formalizing) adequacy?
- Do you want renamings to be first-class? Injective renamings?
- Respecting renaming/substitutivity?
- Binder approach = Representation of Syntax + Substitutions + Reasoning Principles
  - For example: variants of locally nameless; de Bruijn, well scoped de Bruijn…



Toward a General Theory of Names, Binding and Scope

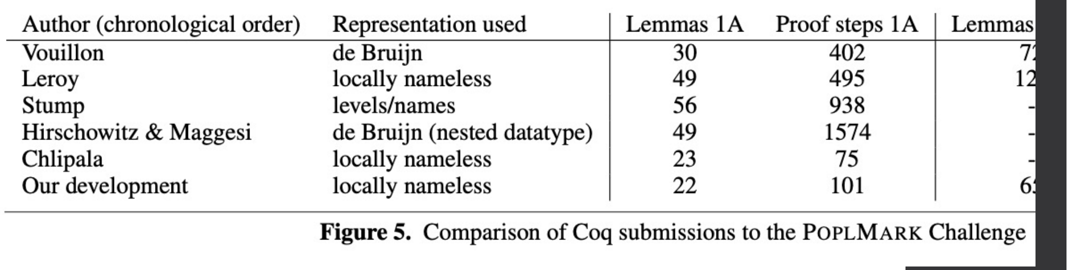
James Cheney

University of Edinburgh
Edinburgh, United Kingdom
jcheney@inf.ed.ac.uk

Abstract

High-level formalisms for reasoning about names and binding such as de Bruijn indices, various flavors of higher-order abstract syntax, the Theory of Contexts, and nominal abstract syntax address only one relatively restrictive form of scoping: namely, unary lexical scoping, in which the scope of a (single) bound name is a subtree of the abstract syntax tree (possibly with other subtrees removed due to shadowing). Many languages exhibit binding or renaming structure that does not fit this mold. Examples include binding transitions in the π-calculus; unique identifiers in contexts, memory heaps, and XML documents; declaration scoping in modules and namespaces; anonymous identifiers in automata, type schemes, and Horn clauses; and pattern matching and mutual recursion constructs in functional languages. In these cases, it appears necessary the tedious details attendant upon formalizations of abstract syntax with bound names have been proposed. These include name-free approaches such as combinators and de Bruijn representations [7] as well as higher-order approaches such as higher-order abstract syntax [20], weak higher-order abstract syntax [8], and lambda-term abstract syntax [14]. Another recently proposed technique is the approach of Gabbay and Pitts [10], which focuses on alpha-equivalence axiomatized in terms of name-swapping and freshness. Additional techniques such as Hybrid [18], the Theory of Contexts [11], and $FO\lambda^{\Delta\nabla}$ [16] have also recently been proposed. In this paper we employ *nominal abstract syntax*, a simplified form of *nominal logic* [21].

Scope is a fundamental concept when discussing binding. If we view a syntax representation as an abstract data structure, then the



Author (chronological order)	Representation used	Lemmas 1A	Proof steps 1A	Lemmas
Vouillon	de Bruijn	30	402	72
Leroy	locally nameless	49	495	12
Stump	levels/names	56	938	-
Hirschowitz & Maggesi	de Bruijn (nested datatype)	49	1574	-
Chlipala	locally nameless	23	75	-
Our development	locally nameless	22	101	65

Figure 5. Comparison of Coq submissions to the POPLMARK Challenge