To appear in Proceedings of HUG'93: International Higher-Order Logic Theorem Proving Workshop, August 11–13, 1993, University of British Columbia. Springer Lecture Notes in Computer Science, 1994.

A Mechanisation of Name-carrying Syntax up to Alpha-conversion

Andrew D. Gordon

Programming Methodology Group
Department of Computing Science
Chalmers University of Technology /
University of Gothenburg
412 96 Gothenburg, Sweden
gordon@cs.chalmers.se

Abstract. We present a new strategy for representing syntax in a mechanised logic. We define an underlying type of de Bruijn terms, define an operation of named lambda-abstraction, and hence inductively define a set of conventional name-carrying terms. The result is a mechanisation of the practice of most authors studying formal calculi: to work with conventional name-carrying notation and substitution, but to identify terms up to alpha-conversion. This strategy falls between most previous works, which either treat bound variable names literally or dispense with them altogether. The theory has been implemented in the Cambridge HOL system and used in an experimental application.

There is great interest in using theorem provers to prove properties of lambda-calculi and programming languages [3, 7, 9, 14, 15, 18, 19, 21, 25, 26, 27, 28] One basic issue is the representation in logic of syntax, and in particular, how to represent bound variables. One strategy is to represent bound variables using variable names, as in conventional syntax. Another is to use de Bruijn indices [8]. An advantage of the first is that theorems are expressed in a conventional, human-readable form. An advantage of the second is that substitution is easy to define and the problems of variable name clashes never arise. But neither exactly corresponds to the practice in most written studies[1] of formal calculi where it is convenient to identify syntax up to alpha-conversion for doing proofs, but at the same time retain bound variable names for readability.

This paper presents a novel method for representing syntax, combining these two strategies. The key idea is that conventional lambda-abstraction can be defined on de Bruijn terms, and then one can prove that the conventional inductive definition of name-carrying notation generates exactly the proper de Bruijn terms (that is, terms with no unmatched indices). From this key theorem we can obtain a conventional theory of syntax and substitution [13], in which logical equality corresponds to alpha-conversion. The advantage of this mixed strategy is that theorems can be expressed in a conventional form, without de

[1] Church and Rosser [6] and Barendregt [4] are prominent examples.

Bruijn encoding, and although in application proofs renaming of bound variables is sometimes necessary, it is easy to support because logical equality is up to alpha-conversion.

The particular motivation for this paper is to work towards an environment for mechanised proofs about functional programs based on their operational semantics. As a step towards such an environment, we have programmed all the theory given here in the Cambridge HOL system [10], and have used it to support elementary reasoning about Abramsky's lazy lambda-calculus [2]. The point of the paper is not to discuss applications, but simply to explain the syntactic strategy, which may be of use in other theorem provers. Section 1 reviews conventional and de Bruijn notation, and Section 2 gives a type of de Bruijn terms in HOL. Section 3 is the crux of the paper: the inductive definition of name-carrying terms. Section 4 outlines the main components of the derived theory, and Section 5 outlines how it might be applied to represent an example syntax. Section 6 concludes with a discussion of related and future work.

Cambridge HOL supports Gordon's polymorphic formulation of Church's classical higher order logic. Theorems proved in the HOL system are indicated here with a turnstyle |- and written in HOL's fairly conventional notation [10]. We use Melham's encoding of sets as logical predicates [16].

1 Review of Name-carrying Versus de Bruijn Notation

As usual, we consider each term of the conventional notation for lambda-calculus—referred to as *name-carrying* by de Bruijn—to be either a constant, a, a variable, x, an application $t_1\ t_2$ or a lambda-abstraction $\lambda x.\, t_0$, where each t_i is a term.

De Bruijn's key idea was to erase the name x from a lambda-abstraction $\lambda x.\, t$, and replace each free occurrence of x in t by a numeric index, equal to the level of the occurrence. The *level* of an occurrence of a subterm u within a term t, is the number of λ's in t that enclose u. So name-carrying $\lambda x.\, (\lambda y.\, x)x$ becomes de Bruijn $\lambda.\, (\lambda.\, 1)0$. An occurrence of an index i in a term is *matched* iff i is less than the level of the occurrence. In $\lambda.\, (01)$ each of the indices occurs at level 1, so index 0 is matched but 1 is not.

A matched occurrence of an index represents a bound variable. How is a free variable represented? For an entirely nameless notation, de Bruijn suggested that the names of free variables could be replaced by unmatched indices interpreted via a fixed enumeration of variable names. Instead, we adopt another convention suggested by de Bruijn, in which "free variables have names but the bound variables are nameless" [8, p392]. We will refer to this simply as de Bruijn notation. For instance, name-carrying $x(\lambda y.\, axy)$ becomes de Bruijn $x(\lambda.\, ax0)$.

For each occurrence of an index i at level ℓ define its *degree* to be 0 if it is matched, otherwise the positive number $i - \ell + 1$. The *degree* of a de Bruijn term is the maximum degree of all index occurrences in the term. So $\lambda.\, (01)$ has degree 1 and $x(\lambda.\, ax0)$ degree 0. A de Bruijn term is *proper* iff its degree is 0, that is, when no unmatched indices occur in it. The strategy proposed here

rests essentially on the fact that the name-carrying terms can be modelled up to alpha-conversion by the proper de Bruijn terms.

2 A HOL Type of de Bruijn Terms

This section shows how de Bruijn terms and their basic operations can be expressed in the HOL logic. The following grammar specifies a HOL type of de Bruijn terms.

```
db = dCON *          (constant)
   | dVAR string     (free variable)
   | dBOUND num      (bound variable)
   | dABS db         (abstraction)
   | dAPP db db      (application)
```

In HOL, Melham's recursive type package [19] proves automatically the existence of a type which is the free algebra given by the grammar, and proves theorems supporting primitive recursive definitions and proofs by structural induction.

The type obtained, `* db`, is polymorphic. Constants are represented by values in the polymorphic type `*`. This polymorphic type would be instantiated to a type capable of representing all the symbols used in a particular syntax. Variable names are represented by strings. All we need to know about the type of names is that it is infinite—so that fresh names can always be generated—and this is easy to prove about strings. If desired, one could parameterise `db` on a type of variable names, as Melham does [18], but then theorems would need to bear the assumption that the type of names was infinite.

We introduce new HOL constants, `dDEG` and `dFV` of types `* db -> num` and `* db -> string set`, to express respectively the degree and set of free variables of a de Bruijn term.

```
|- (dDEG (dCON c)   = 0) /\
   (dDEG (dVAR x)   = 0) /\
   (dDEG (dBOUND n) = SUC n) /\
   (dDEG (dABS t)   = PRE (dDEG t)) /\
   (dDEG (dAPP t u) = MAX (dDEG t) (dDEG u))

|- (dFV (dCON c)   = {}) /\
   (dFV (dVAR x)   = {x}) /\
   (dFV (dBOUND n) = {}) /\
   (dFV (dABS t)   = dFV t) /\
   (dFV (dAPP t u) = (dFV t) UNION (dFV u))
```

Its not hard to see that that the structural definition of `dDEG` is equivalent to the one given in Section 2.

To support variable abstraction and substitution we adopt two of de Bruijn's operations in the form given by Paulson [23].[2]

```
|- (Abst i x (dCON c)   = dCON c) /\
   (Abst i x (dVAR y)   = ((x = y) => dBOUND i | dVAR y)) /\
   (Abst i x (dBOUND j) = dBOUND j) /\
   (Abst i x (dABS t)   = dABS (Abst (SUC i) x t)) /\
   (Abst i x (dAPP t u) = dAPP (Abst i x t) (Abst i x u))

|- (Inst i (dCON c)     u = dCON c) /\
   (Inst i (dVAR x)     u = dVAR x) /\
   (Inst i (dBOUND j)   u = ((i = j) => u | dBOUND j)) /\
   (Inst i (dABS t)     u = dABS (Inst (SUC i) t u)) /\
   (Inst i (dAPP t1 t2) u = dAPP (Inst i t1 u) (Inst i t2 u))
```

The intention is that `Abst i x t` is the term obtained by replacing each free occurrence of variable `x` in `t` with index `i` (or a greater index within abstractions). `Inst i t u` is meant to be the term obtained by replacing each bound variable `i` in `t` with `u` (which is assumed to be proper).

3 An Inductive Definition of Name-carrying Terms

This section explains our strategy for mechanising a name-carrying notation. Although the underlying type is one of de Bruijn terms, we wish to recover the conventional operation of free variable abstraction. We can define a new operation `dLAMBDA` of type `string -> * db -> * db` from the operations `dABS` and `Abst`.

```
|- dLAMBDA x t = dABS (Abst 0 x t)
```

Intuitively `dLAMBDA` binds free occurrences of variable `x` in `t`. With `dLAMBDA` we have a name-carrying notation that denotes de Bruijn terms. For instance, we can prove the following by rewriting in HOL, that name-carrying $\lambda x.\,(\lambda y.\,x)x$ equals de Bruijn $\lambda.\,(\lambda.\,1)0$.

```
|- dLAMBDA ‘x‘ (dAPP (dLAMBDA ‘y‘ (dVAR ‘x‘)) (dVAR ‘x‘)) =
   dABS(dAPP(dABS(dBOUND 1))(dBOUND 0))
```

Since the underlying type erases bound variable names, equality on the derived name-carrying notation is up to alpha-conversion.

```
|- dLAMBDA ‘x‘ (dAPP(dLAMBDA ‘y‘ (dVAR ‘x‘))(dVAR ‘x‘)) =
   dLAMBDA ‘y‘ (dAPP(dLAMBDA ‘x‘ (dVAR ‘y‘))(dVAR ‘y‘))
```

[2] `Abst` and `Inst` correspond to Paulson's `abstract` and `subst`. `Inst` is a slight simplification of `subst` because we are only interested in instantiating proper terms for bound variables.

We have operations now to express all conventional notation in de Bruijn form, and so we are free to adopt the conventional inductive definition of name-carrying terms.

$$\frac{}{\texttt{META (dCON c)}} \qquad \frac{}{\texttt{META (dVAR x)}}$$

$$\frac{\texttt{META t}}{\texttt{META (dLAMBDA x t)}} \qquad \frac{\texttt{META t} \quad \texttt{META u}}{\texttt{META (dAPP t u)}}$$

In HOL, Melham's tool for inductive definitions [17] proves automatically that `META` is the least predicate closed under the given rules. The name of the predicate comes from its intended use as a single logical type able to encode the syntax of a range of object languages. This method is sometimes known as a *meta-theory of syntax* and was promoted by Martin-Löf and others [22], and implemented in systems such as Elf [25] and Isabelle [24]. The point is to define substitution and alpha-conversion once and for all in the meta-theory.

The key result of the implemented theory is that the terms satisfying this conventional inductive definition are precisely the proper de Bruijn terms.

```
|- !t. META t = (dDEG t = 0)
```

The forwards direction is a rule induction on the derivation of the predicate `META`. The backwards direction is a course-of-values induction on the length of the term `t` (a measure discussed in the next section).

4 A HOL Type of Name-carrying Terms up to Alpha-conversion

The next step is to define a subtype of `* db`, called `* meta`, whose elements are precisely the proper de Bruijn terms, or equivalently, conventional name-carrying lambda-terms up to alpha-conversion.

Types in HOL can be modelled as non-empty sets. A new type can be introduced in bijection with a given non-empty subset of an existing type. This is sound in the sense that it preserves the property of the logic having a standard model [10]. In the present case, we define the new type `* meta`, whose elements are precisely the set of proper de Bruijn terms in the existing type `* db`. It is easy to prove that the set of proper de Bruijn terms is not empty. Tools in the HOL system prove automatically some basic properties of the bijection between `* meta` and the elements of type `* db` satisfying predicate `META`. We omit the details. Operations on type `* db` that are closed under the predicate `META` can be used to induce equivalent operations on type `* meta`.

The basic proposition of the paper is that applications needing a representation of name-carrying syntax can be built using this new type. In the remainder of this section we outline the basic theory implemented for this type.

Basic Properties of meta-terms

We refer to values of the new type as meta-terms. There are four basic constructors of meta-terms, corresponding to the four rules of the inductive definition of META, that is, constants, variables, applications and lambda-abstractions.

```
mCON    : * -> * meta
mVAR    : string -> * meta
'       : * meta -> * meta -> * meta
mLAMBDA : string -> * meta -> * meta
```

The binary application constructor ' is infix. These four constructors exhaust the meta-terms.

```
|- !m.
    (?c. m = mCON c) \/
    (?x. m = mVAR x) \/
    (?x n. m = mLAMBDA x n) \/
    (?m1 m2. m = m1 ' m2)
```

There are also theorems stating that each of the constructors yields distinct terms, and that the three constructors mCON, mVAR and ' are one-one. The exhaustion theorem follows from the inductive characterisation of meta-terms, META, whereas the others follow from the definition of de Bruijn terms as a free algebra.

To obtain substitution on meta-terms, we first define substitution on de Bruijn terms.

```
|- !t u x. t dSUB (u,x) = Inst 0 (Abst 0 x t) u
```

De Bruijn term t dSUB (u,x) is intended to be term t with each free occurrence of x replaced by u. This operation on the representation type is used to induce a substitution operation on meta-terms that we denote with infix /. The meta-term (m / x) n is meant to be meta-term n with each free occurrence of x replaced by meta-term m.

Given that FV of type * meta -> string set is the operation induced by the free variable function dFV, we can prove alpha-conversion as a logical equation.

```
|- !m x y. ~y IN (FV m) ==>
     (mLAMBDA x m = mLAMBDA y (((mVAR y) / x) m))
```

We can prove the expected distributive laws for substitution,

```
|- !a m x. (m / x)(mCON a) = mCON a
|- !m x. (m / x)(mVAR x) = m
|- !m x y. ~(x = y) ==> ((m / x)(mVAR y) = mVAR y)
|- !m n x. (n / x)(mLAMBDA x m) = mLAMBDA x m
|- !m n x y.
    ~(x = y) /\ ~y IN (FV n) ==>
    ((n / x)(mLAMBDA y m) = mLAMBDA y ((n / x) m))
|- !m n p x. (p / x)(m ' n) = ((p / x) m) ' ((p / x) n)
```

by first proving the equivalent properties about de Bruijn terms.

Structural Induction

The inductive definition of `meta`-terms gives rise to the following induction principle:

```
|- !P: (*)meta -> bool.
    (!x. P(mCON x)) /\
    (!x. P(mVAR x)) /\
    (!m. P m ==> (!x. P(mLAMBDA x m))) /\
    (!m n. P m /\ P n ==> P(m ' n))
   ==>
    (!m. P m)
```

One difficulty found in applications with this induction principle is that the hypothesis for proving `P(mLAMBDA x m)` is too weak; it allows `P m` to be assumed, but if the bound variable has to be renamed, one needs to assume `P ((mVAR z / x) m)` for some fresh variable `z`.

One solution to this problem rests on the fact that variable renaming does not alter the *length* of a term, the number of syntactic constructors it contains. This is a standard notion [13], that can be induced on `meta`-terms via a primitive recursion on the type of de Bruijn terms.

```
|- (LGH (mCON c)      = 1) /\
   (LGH (mVAR x)      = 1) /\
   (LGH (mLAMBDA x m) = 1 + (LGH m)) /\
   (LGH (m ' n)       = (LGH m) + (LGH n))
```

This measure is invariant under variable renaming [13, Lemma 1.14(d)].

```
|- !m x y. LGH (((mVAR x) / y) m) = LGH m
```

Now we can prove a new induction principle, which strengthens the induction hypothesis for `mLAMBDA`-terms to hold for all `meta`-terms `n` of the same length as `m`.

```
|- !P: (*)meta -> bool.
    (!x. P(mCON x)) /\
    (!x. P(mVAR x)) /\
    (!m.
      (!n. (LGH n = LGH m) ==> P n)
     ==>
      (!x. P(mLAMBDA x m))) /\
    (!m n. P m /\ P n ==> P(m ' n))
   ==>
    (!m. P m)
```

The proof of this principle is by course-of-values induction on the length of the `meta`-term.

This second induction theorem solves the problem that bound variables may need to be renamed, but does so using the LGH measure. One can avoid explicit mention of the measure by using a third induction theorem, which follows easily from the second.

```
|- !P.
    (!x. P(mVAR x)) /\
    (!x. P(mCON x)) /\
    (?X. FINITE X /\
       (!x m. ~x IN X /\ P m ==> P(mLAMBDA x m))) /\
    (!m n. P m /\ P n ==> P(m ' n))
   ==>
    (!m. P m)
```

This is a convenient formulation of structural induction. To prove the case for mLAMBDA-terms, one first must select a set X of variables, and then prove that predicate P holds for (mLAMBDA x m) from the assumptions that P holds of m, and, crucially, that bound variable x is distinct from everything in X. By specifying a suitable X one gets to assume that x is a fresh variable. The clause for mLAMBDA-terms captures the intuition that one can always assume that bound variables are distinct from everything else of interest. Tactics have been programmed in ML to automate application of this theorem and, for the case of mLAMBDA-terms, to select sets X and prove them finite.

Derived Theorems

The theorems mentioned in the last two subsections about meta-terms are sufficient to derive the following results, without recourse to theorems about the underlying type of de Bruijn terms.

Using the third form of structural induction, we can derive all the properties of substitution used by Hindley and Seldin in their text on lambda-calculus [13]. Here is their Lemma 1.14,

```
|- !x m. ((mVAR x) / x) m = m
|- !x n m. ~x IN (FV m) ==> ((n / x) m = m)
|- !x n m. x IN (FV m) ==>
     (FV((n / x) m) = (FV n) UNION ((FV m) DELETE x))
|- !m x y. LGH(((mVAR x) / y )m) = LGH m
```

and their Lemma 1.15.

```
|- !x y n m. ~y IN (FV m) ==>
     ((n / y)(((mVAR y) / x) m) = (n / x) m)
|- !x y m. ~y IN (FV m) ==>
     (((mVAR x) / y)(((mVAR y) / x) m) = m)
|- !x y p q m.
    ~y IN (FV p) /\ ~(x = y) ==>
    ((p / x)((q / y) m) = (((p / x) q) / y)((p / x) m))
```

```
|- !x y p q m.
     ~y IN (FV p) /\ ~x IN (FV q) /\ ~(x = y) ==>
     ((p / x)((q / y) m) = (q / y)((p / x) m))
|- !x p q m. (p / x)((q / x) m) = (((p / x) q) / x) m
```

The third part of their Lemma 1.14 shows that formalised substitution `/` never accidentally captures free variables. If some `y` is free in `n` it remains free in `(n / x) m`.

Given alpha-conversion, Hindley and Seldin's lemmas and proof of the infinity of string names, it is possible to derive theorems concerning `mLAMBDA` analogous to the one-one theorems proved for the other constructors.

```
|- !x y m n.
    (mLAMBDA x m = mLAMBDA y n) ==> (m = ((mVAR x) / y) n)
|- !m n x y.
    (mLAMBDA x m = mLAMBDA y n) =
    (?z.
      ~z IN (FV m) /\
      ~z IN (FV n) /\
      (((mVAR z) / x) m = ((mVAR z) / y) n))
```

5 How the `meta` Type Can Be Applied

Here is an example grammar to illustrate how `meta`-terms could play the role of a meta-theory of syntax in HOL.

```
e ::= x
    | NUM n | (e1 PLUS e2)
    | BOOL b | (IF e1 e2 e3)
    | (LAMBDA x e) | (APP e1 e2)
    | LET x = e1 IN e2
```

To encode the syntactic constants and literals we can use the following HOL type

```
con = NumLit num
    | BoolLit bool
    | SynCon string
```

and then represent terms of the grammar as elements of type `(con)meta`, using the logical constants defined as follows.

```
|- NUM n       = mCON (NumLit n)
|- e1 PLUS e2  = (mCON(SynCon `PLUS`) ' e1) ' e2
|- BOOL b      = mCON (BoolLit b)
|- IF e1 e2 e3 = ((mCON(SynCon `IF`) ' e1) ' e2) ' e3
|- LAMBDA x e  = mCON(SynCon `LAMBDA`) ' (mLAMBDA x e)
|- APP e1 e2   = (mCON(SynCon `APP`) ' e1) ' e2
|- LET x e1 e2 = (mCON(SynCon `LET`) ' e1) ' (mLAMBDA x e2)
```

One can then give the set of terms in the grammar as an inductively-defined predicate on `meta`-terms. This is not the only way to embed this grammar; it remains future work to investigate alternatives and how syntactic properties can be derived for the new logical constants.

6 Related Work, Status and Future Plans

This paper has presented a strategy for representing syntax within a mechanised logic: construct the type of name-carrying terms—identified up to alpha-conversion—from an inductively defined subset of de Bruijn terms. This method mixes two strategies used in previous work: to represent name-carrying syntax directly [15, 19, 21, 26, 27, 28] and to adopt some variant of de Bruijn notation [3, 14, 27]. Coquand [7] pursues a fourth strategy based on explicit substitutions [1]. More experience with the different strategies is needed before they can be meaningfully compared, but the mixed strategy advocated here is at least a plausible method for directly mechanising the practice of many authors studying formal calculi: to work with name-carrying notation and to identify terms up to alpha-conversion.

Melham's π-calculus work includes another logical formulation of conventional name-carrying notation and substitution, with terms identified up to alpha-conversion. In his approach, a natural alternative to the present one, the underlying type is a free algebra of name-carrying syntax for the π-calculus. On this underlying type he defines substitution and alpha-conversion [18], and then gives a type of syntax up to alpha-conversion as the quotient of the underlying type under alpha-conversion.[3] Of course, this construction would work for lambda-calculus as well.

Hence we have two independently constructed representations for the same abstract type of syntax up to alpha-conversion. Each representation has its merits. Name-carrying syntax up to literal equality would be needed to represent language definitions, such as that of Standard ML [20], for instance, where syntax is not identified up to alpha-conversion. On the other hand, de Bruijn notation is a common implementation technique, and so a logical theory of de Bruijn notation is needed if such implementations are to be verified. An important question left open by the present work is to find an axiomatisation of name-carrying syntax up to alpha-conversion. Melham's quotient construction may be of help here, and in any case a useful extension of the present work would be to add a third type, of name-carrying syntax up to literal equality, and investigate its relation to the other two.

Section 5 showed how a grammar might be embedded in HOL using the `meta` type. This is reminiscent of how logical frameworks like Isabelle [24] or Elf [25] are used to embed object languages and logics in a fixed meta-logic. One significant difference is that to the best of my knowledge in neither Isabelle nor Elf can theorems about substitution such as the ones from Section 4 be proved in the meta-logic.

[3] Private communication with T. F. Melham, June 1993.

De Bruijn notation has been used to implement several theorem provers, such as his own AUTOMATH and Paulson's Isabelle. In Isabelle, for instance, syntax is represented as an ML type using de Bruijn indices. The human interacting with Isabelle sees and types a name-carrying notation, which is mapped to and from the internal de Bruijn notation by ML functions. An important difference from Isabelle is that in our approach the type of de Bruijn terms is represented in the logic rather than the programming metalanguage.

The difficult part of embedding syntax is dealing with bound variables. The approach of this paper is *first-order* in the sense that the variable-binding operation of the embedded syntax (`mLAMBDA`) is distinct from variable-binding in the logic. One might think of adopting a *higher-order* approach in which logical lambda-abstraction binds variables. A first attempt would be to construct a type with the same four constructors as `* meta` but with typing `(* meta -> * meta) -> * meta` for `mLAMBDA`. Such a type cannot be constructed using Melham's recursive types package, and in fact Gunter [12] has shown by a cardinality argument that this approach is impossible in HOL, if the constructors are injective (which we would want). Roughly speaking, the HOL function space is much bigger than the space of syntactic lambda-abstractions. A better approach is to give `mLAMBDA` the typing `(string -> * meta) -> * meta`. Despeyroux, Felty and Hirschowitz are investigating this embedding using Coq [9].

There is another, in some respects simpler, approach to representing languages in logic, pioneered by Gordon [11], in which each phrase of syntax is mapped metalinguistically to a logical constant that represents not the syntax itself but its meaning. When embedding a language of imperative commands, for instance, the meaning might be a relation between two states. The present approach is quite different in that syntax itself is represented in the logic, which is needed for instance to support operational language definitions. The two approaches—meaning versus syntax—are sometimes distinguished respectively with terms 'semantic' versus 'syntactic', or 'shallow' versus 'deep' [5].

The status of this work is that all the theory mentioned has been implemented in the Cambridge HOL system. Using the theory, Abramsky's lazy lambda-calculus has been mechanised in HOL, and elementary facts proved. The `meta` type can be used to encode other syntaxes using appropriate constants, just as the single ML type of terms in Isabelle can encode a range of syntaxes. The goal of future work is to build a single HOL theory able to encode any name-carrying syntax, together with tools to support definition of new syntaxes, recursive functions and proofs by induction, and hence to offer a general solution to the notorious problems of bound variables and substitution.

Acknowledgements

Comments from Catarina Coquand, Graham Hutton, Karsten Kehler Holst, Konrad Slind and the anonymous referees, and several discussions with Tom Melham were very helpful.

References

1. Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.
2. Samson Abramsky. The lazy lambda calculus. In David Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley, 1990.
3. Thorsten Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In *TLCA '93 International Conference on Typed Lambda Calculi and Applications, Utrecht, 16–18 March 1993*, volume 664 of *Lecture Notes in Computer Science*, pages 13–28. Springer-Verlag, 1993.
4. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, revised edition, 1984.
5. Richard Boulton, Andrew Gordon, Mike Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *Theorem Provers in Circuit Design: Theory, Practice and Experience: Proceedings of the IFIP TC10/WG 10.2 International Conference, Nijmegen, June 1992*, IFIP Transactions A-10, pages 129–156. North-Holland, 1992.
6. Alonzo Church and J. B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 36(3):472–482, May 1936.
7. Catarina Coquand. A machine assisted semantical analysis of simply typed lambda calculus. In P. Dybjer, J. Hughes, A. Moran, and B. Nordström, editors, *Proceedings of El Wintermöte*, pages 92–100. Programming Methodology Group, Chalmers University of Technology and University of Gothenburg, June 1993. Available as Report 73.
8. N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.
9. Joëlle Despeyroux and André Hirschowitz. Higher-order abstract syntax and induction. Transparencies for a talk at the Types BRA Workshop on Proving Properties of Programming Languages, Sophia-Antipolis, September 1993.
10. M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem-proving environment for higher-order logic*. Cambridge University Press, 1993.
11. Michael J. C. Gordon. Mechanizing programming logics in higher order logic. Technical Report 145, University of Cambridge Computer Laboratory, September 1988.
12. Elsa L. Gunter. Why we can't have `SML` style `datatype` declarations in `HOL`. In L. Claesen and M. Gordon, editors, *Higher Order Logic Theorem Proving and its Applications*, pages 365–372, Leuven, 1992. IMEC.
13. J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.
14. Gérard Huet. Residual theory in λ-calculus: A complete Gallina development. Preprint, 1992.
15. James McKinna and Robert Pollack. Pure Type Systems formalized. In *TLCA '93 International Conference on Typed Lambda Calculi and Applications, Utrecht, 16–18 March 1993*, volume 664 of *Lecture Notes in Computer Science*, pages 289–305. Springer-Verlag, 1993.
16. T. F. Melham. *The HOL* `pred_sets` *Library*. University of Cambridge Computer Laboratory, February 1992.

17. Thomas F. Melham. A package for inductive relation definitions in HOL. In *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, Davis, California*, pages 350–357. IEEE Computer Society Press, 1991.
18. Thomas F. Melham. A mechanized theory of the π-calculus in HOL. Technical Report 244, University of Cambridge Computer Laboratory, January 1992.
19. Thomas Frederick Melham. *Formalizing Abstraction Mechanisms for Hardware Verification in Higher Order Logic.* PhD thesis, University of Cambridge Computer Laboratory, August 1990. Available as Technical Report 201.
20. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, Mass., 1990.
21. Monica Nesi. A formalization of the process algebra CCS in higher order logic. Technical Report 278, University of Cambridge Computer Laboratory, December 1992.
22. Bengt Nordström. Martin-Löf's type theory as a programming logic. Report 27, Programming Methodology Group, Chalmers University of Technology and University of Gothenburg, September 1986.
23. Lawrence C. Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.
24. Lawrence C. Paulson. The Isabelle reference manual. Internal report, University of Cambridge Computer Laboratory, 1992.
25. Frank Pfenning. A proof of the Church-Rosser theorem and its representation in a logical framework. Technical Report CMU-CS-92-186, Computer Science Dept., Carnegie Mellon University, September 1992.
26. Randy Pollack. Closure under alpha-conversion. Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, September 1993.
27. N. Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the ACM*, 35(3):475–522, July 1988.
28. J. von Wright. Representing higher-order logic proofs in HOL. University of Cambridge Computer Laboratory, July 1993.

This article was processed using the LaTeX macro package with LLNCS style