A. D. Gordon and T. Melham, 'Five Axioms of Alpha Conversion', in *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, edited by J. von Wright, J. Grundy and J. Harrison, Lecture Notes in Computer Science, vol. 1125 (Springer-Verlag, 1996), pp. 173–190.

Five Axioms of Alpha-Conversion

Andrew D. Gordon[1] and Tom Melham[2]

[1]University of Cambridge Computer Laboratory,
New Museums Site, Cambridge CB2 3QG, UK. `adg@cl.cam.ac.uk`
[2]Department of Computing Science, University of Glasgow,
Glasgow G12 8QQ, Scotland. `tfm@dcs.gla.ac.uk`

Abstract. *We present five axioms of name-carrying lambda-terms identified up to alpha-conversion—that is, up to renaming of bound variables. We assume constructors for constants, variables, application and lambda-abstraction. Other constants represent a function* Fv *that returns the set of free variables in a term and a function that substitutes a term for a variable free in another term. Our axioms are (1) equations relating* Fv *and each constructor, (2) equations relating substitution and each constructor, (3) alpha-conversion itself, (4) unique existence of functions on lambda-terms defined by structural iteration, and (5) construction of lambda-abstractions given certain functions from variables to terms. By building a model from de Bruijn's nameless lambda-terms, we show that our five axioms are a conservative extension of HOL. Theorems provable from the axioms include distinctness, injectivity and an exhaustion principle for the constructors, principles of structural induction and primitive recursion on lambda-terms, Hindley and Seldin's substitution lemmas and the existence of their length function. These theorems and the model have been mechanically checked in the Cambridge HOL system.*

The axioms presented in this paper are intended to give a simple, abstract characterisation of untyped lambda-terms, with constants, identified up to alpha-conversion, that is, renaming of bound variables. We were led to develop these axioms because we are interested in representing the syntax of programming languages with binding operators within a theorem prover. The difficulty of correctly defining substitution on lambda-terms is notorious. Previous experience with the pi-calculus (Milner, Parrow, and Walker 1992) in HOL (Melham 1994) suggests that developing substitution and binding operators directly is a tedious and error-prone business. Instead, to avoid error and repetition, we advocate first developing a metatheory of untyped lambda-terms, and secondly deriving syntax for a particular programming language as abbreviations for untyped lambda-terms. We will show in section 4 how to do this for a finitary pi-calculus.

Given higher-order logic, as implemented in the Cambridge HOL system (Gordon and Melham 1993), what we are after is a logical type $(\alpha)\mathit{term}$ that stands for the set of lambda-terms, where α is the type of constants. Terms are generated by the four constructors:

$\mathsf{Con} : \alpha \to (\alpha)\mathit{term}$	(constants)
$\mathsf{Var} : \mathit{string} \to (\alpha)\mathit{term}$	(variables)
$\mathsf{App} : (\alpha)\mathit{term} \to (\alpha)\mathit{term} \to (\alpha)\mathit{term}$	(applications)
$\mathsf{Lam} : \mathit{string} \to (\alpha)\mathit{term} \to (\alpha)\mathit{term}$	(lambda-abstractions)

Consider the concrete recursive type—the free algebra—generated by these constructors. Concrete recursive types are implemented in HOL using Melham's type definition package (Gordon and Melham 1993, Chapter 20). Given these constructors, the package proves the existence of a type characterized by the single axiom:

$$\begin{array}{l}
\vdash \forall \mathit{con} : \alpha \to \beta. \\
\quad \forall \mathit{var} : \mathit{string} \to \beta. \\
\quad \forall \mathit{app} : \beta \to \beta \to (\alpha)\mathit{term} \to (\alpha)\mathit{term} \to \beta. \\
\quad \forall \mathit{lam} : \beta \to \mathit{string} \to (\alpha)\mathit{term} \to \beta. \\
\quad \exists! \mathit{hom} : (\alpha)\mathit{term} \to \beta. \\
\qquad \forall k.\ \mathit{hom}(\mathsf{Con}\ k) = \mathit{con}\ k \ \wedge \\
\qquad \forall x.\ \mathit{hom}(\mathsf{Var}\ x) = \mathit{var}\ x \ \wedge \\
\qquad \forall t\, u.\ \mathit{hom}(\mathsf{App}\ t\ u) = \mathit{app}\ (\mathit{hom}\ t)\ (\mathit{hom}\ u)\ t\ u \ \wedge \\
\qquad \forall x\, u.\ \mathit{hom}(\mathsf{Lam}\ x\ u) = \mathit{lam}\ (\mathit{hom}\ u)\ x\ u
\end{array}$$

The axiom allows for the definition of functions by *primitive recursion*, where functions *con*, *var*, *app* and *lam* determine the outcome of the function when applied to each constructor, given access to the outcome of recursive calls and to the arguments of the constructor. In fact Melham's tool derives this axiom from a simpler *iteration* axiom. Iteration also allows for the definition of functions by recursion, but the functions *con*, *var*, *app* and *lam* have no direct access to the constructor arguments, only to the outcomes of recursive calls.

Here the type $(\alpha)\mathit{term}$ is a free algebra; all the constructors are injective. Two lambda-abstractions are equal just if their bound variables and their bodies are equal. Instead we are after a type in which terms are identified up to alpha-conversion, that is, in which two lambda-abstractions are equal just if their bodies are equal when the bound variables are renamed to a fresh variable.

1 The Axioms

The main contribution of this paper is to present five axioms for untyped lambda-terms identified up to alpha-conversion, to show how various reasoning principles derive from these axioms, and to show that the axioms are sound. The first three axioms are well-known (Curry and Feys 1958; Barendregt 1984); the fourth and fifth are new. The model we present in section 2 is based on earlier work (Gordon 1994) which showed how untyped lambda-terms could be modelled by de Bruijn terms.

Gordon was not concerned to specify axioms characterising the type of untyped lambda-terms, and did not consider how to define functions by recursion on these terms. Instead our new work shows the importance of an axiom for iteration. In section 3 it allows us to derive primitive recursion (analogous to the axiom displayed above), structural induction, finiteness of free variables, and the function returning the length of a lambda-term, which were all taken straight from the model in Gordon's earlier work. It also allows functions to be defined by recursion on lambda-terms, which was not previously considered.

In addition to the constructors introduced above, our axioms employ three further functions:

$$\begin{array}{ll} \mathsf{Fv} & : (\alpha)\mathit{term} \to (\mathit{string})\mathit{set} \\ \_[\_/\_] & : (\alpha)\mathit{term} \to ((\alpha)\mathit{term} \times \mathit{string}) \to (\alpha)\mathit{term} \\ \mathsf{Abs} & : (\mathit{string} \to (\alpha)\mathit{term}) \to (\alpha)\mathit{term} \end{array}$$

$\mathsf{Fv}(u)$ returns the set of free variables in the term u and $u[t/x]$ produces the result of substituting the term t for the free occurrences of the variable x in the term u. (The partially curried type is needed to make substitution an infix operator in Cambridge HOL.) The function Abs , discussed in detail later, maps certain logical (meta) functions, namely those which carry out a variable-for-variable substitution on terms, to corresponding lambda-abstractions in $(\alpha)\mathit{term}$.

1.1 Free Variables

Axiom one defines the free variables of each constructor:

$$\begin{array}{l} \vdash \forall k.\ \mathsf{Fv}(\mathsf{Con}\ k) = \{\} \wedge \\ \quad \forall x.\ \mathsf{Fv}(\mathsf{Var}\ x) = \{x\} \wedge \\ \quad \forall t\, u.\ \mathsf{Fv}(\mathsf{App}\ t\ u) = \mathsf{Fv}(t) \cup \mathsf{Fv}(u) \wedge \\ \quad \forall x\, u.\ \mathsf{Fv}(\mathsf{Lam}\ x\ u) = \mathsf{Fv}(u) - \{x\} \end{array} \qquad \text{(Axiom 1)}$$

We have expressed this axiom using set-theoretic notation and the type $(\alpha)\mathit{set}$ of sets whose elements are all of type α. This type is an abbreviation for the type of characteristic functions, $\alpha \to \mathit{bool}$. All the set-theoretic operators we use can easily be defined as operations on this type.

Our axioms need not assert that $\mathsf{Fv}(u)$ is always a finite set, for this follows as a theorem from the five axioms.

1.2 Substitution

Axiom two defines the interaction of substitution with each constructor:

$$\begin{array}{l} \vdash \forall k\, u\, x.\ (\mathsf{Con}\ k)[u/x] = \mathsf{Con}\ k \wedge \\ \quad \forall u\, x.\ (\mathsf{Var}\ x)[u/x] = u \wedge \\ \quad \forall u\, x\, y.\ (x{\neq}y) \supset (\mathsf{Var}\ y)[u/x] = \mathsf{Var}\ y \wedge \\ \quad \forall t\, u\, v\, x.\ (\mathsf{App}\ t\ u)[v/x] = \mathsf{App}\ (t[v/x])\ (u[v/x]) \wedge \\ \quad \forall x\, t\, u.\ (\mathsf{Lam}\ x\ t)[u/x] = \mathsf{Lam}\ x\ t \wedge \\ \quad \forall x\, y\, u.\ (x{\neq}y) \wedge y \notin (\mathsf{Fv}\ u) \supset \forall t.\ (\mathsf{Lam}\ y\ t)[u/x] = \mathsf{Lam}\ y\ (t[u/x]) \end{array} \qquad \text{(Axiom 2)}$$

Consider the situation of trying to push a substitution $\_[u/x]$ into $\mathsf{Lam}\ y\ t$ when the bound variable y occurs free in u. It is necessary to avoid the capture of any free occurrences of y in u. Our axiom two does not immediately apply in this situation. But since we identify lambda-terms up to alpha-conversion, we can use axiom three—alpha-conversion—to rename the bound variable y so that the last part of axiom two does apply.

In contrast, Curry and Feys (1958) incorporate this renaming into their definition of substitution—at the cost of an arbitrary choice of renamed variable—because they define substitution directly on the free algebra of lambda-terms and derive alpha-conversion later. On the other hand, Barendregt (1984) avoids this situation via his variable convention, which here permits us to assume that the bound variable y is different from any variable occurring free in u. Stoughton (1988) presents a definition of substitution by structural recursion—the definition by Curry and Feys is by recursion on the length of the term—which always renames bound variables.

1.3 Alpha-Conversion

Axiom three asserts the arbitrariness of bound variables:

$$\vdash \forall y\, u\, x.\; y \notin \mathsf{Fv}(\mathsf{Lam}\ x\ u) \supset (\mathsf{Lam}\ x\ u = \mathsf{Lam}\ y\ (u[\mathsf{Var}\ y/x])) \qquad \text{(Axiom 3)}$$

This is alpha-conversion; two lambda-abstractions may be equal but have distinct bound variables. A consequence is that no logical function is definable that distinguishes such terms.

There is a weaker version of the alpha-conversion axiom,

$$\vdash \forall y\, u.\; y \notin \mathsf{Fv}(u) \supset \forall x.\; \mathsf{Lam}\ x\ u = \mathsf{Lam}\ y\ (u[\mathsf{Var}\ y/x])$$

which in fact follows from the stronger axiom above. We adopt the stronger form because it immediately tells us that

$$\vdash \forall x\, u.\; \mathsf{Lam}\ x\ u = \mathsf{Lam}\ x\ (u[\mathsf{Var}\ x/x])$$

which turns out to be important in later proofs.

1.4 Unique Iteration

Axiom four asserts the unique existence of functions defined by iteration over the structure of terms:

$$\begin{aligned}
\vdash\ & \forall con : \alpha \to \beta. && \text{(Axiom 4)}\\
& \quad \forall var : string \to \beta.\\
& \quad \forall app : \beta \to \beta \to \beta.\\
& \quad \forall abs : (string \to \beta) \to \beta.\\
& \quad \exists! hom : (\alpha)term \to \beta.\\
& \qquad \forall k.\; hom(\mathsf{Con}\ k) = con\ k\ \wedge\\
& \qquad \forall x.\; hom(\mathsf{Var}\ x) = var\ x\ \wedge\\
& \qquad \forall t\, u.\; hom(\mathsf{App}\ t\ u) = app\ (hom\ t)\ (hom\ u)\ \wedge\\
& \qquad \forall x\, u.\; hom(\mathsf{Lam}\ x\ u) = abs\ (\lambda y.\; hom(u[\mathsf{Var}\ y/x]))
\end{aligned}$$

Suppose we want to define a function *hom* of type $(\alpha)term \to \beta$ on lambda-terms by recursion. Given functions *con*, *var*, *app* and *abs* that specify how *hom* treats each of the four constructors, the axiom asserts that such a *hom* exists and moreover is unique. It is analogous to the iteration axiom characterizing concrete recursive types, mentioned in the introduction. As we discuss in section 3 many properties, such as the fact that constructors yield distinct terms, follow from this axiom, in much the same way as analogous properties follow from the single axiom of concrete recursive types.

The difference between this axiom and the ones for concrete recursive types is in the recursion equation for lambda-abstractions. The value of $hom(\mathsf{Lam}\ x\ u)$ is determined (by the parameter *abs*) to be $abs(\lambda y.\ hom(u[\mathsf{Var}\ y/x]))$ but not $abs(hom\ u)$. It cannot be the latter because that would fix the arbitrary bound variable to be x and allow us to distinguish alpha-equivalent terms. Instead the function *abs* is supplied with a function that will yield $hom(u[\mathsf{Var}\ y/x])$ for any $\mathsf{Var}\ y$ to be substituted for x in u. This function, rather than just u, should be regarded as the 'body' of the original lambda-term $\mathsf{Lam}\ x\ u$. So *abs* can work on $hom(u[\mathsf{Var}\ y/x])$ provided it chooses a name y for the arbitrary bound variable x. To employ this principle of iteration in practice, we appear to need one final axiom.

1.5 Abstraction

Axiom five asserts that from any function of type $string \to (\alpha)term$ that represents the body of a lambda-abstraction one can reconstruct 'the' lambda-abstraction itself:

$$\vdash \forall x\, u.\ \mathsf{Abs}(\lambda y.\ u[\mathsf{Var}\ y/x]) = \mathsf{Lam}\ x\ u \qquad \text{(Axiom 5)}$$

where the constant Abs has the type $(string \to (\alpha)term) \to (\alpha)term$. (Remember that λ signifies lambda-abstraction in the HOL logic itself, and that Lam signifies the lambda-abstraction of the type $(\alpha)term$ of untyped lambda-terms embedded in HOL.)

This axiom could, of course, be taken as a *definition* of the Lam constructor. Our axiom set is therefore redundant (Lam is eliminable). But we retain Lam because it clarifies the presentation and serves to highlight the correspondence between lambda-abstractions in $(\alpha)term$ and certain meta-level functions in $string \to (\alpha)term$.

The existence of Abs is of importance primarily because it lets us build lambda-abstractions from lambda-bodies. Consider, for example, the problem of defining a function that uses structural iteration to build a copy of any given lambda-term. We take

$$\begin{aligned}
\beta\ &:= (\alpha)term\\
con &:= \mathsf{Con}\\
var &:= \mathsf{Var}\\
app &:= \mathsf{App}\\
abs\ &:= \mathsf{Abs}
\end{aligned}$$

in the unique iteration axiom. This gives

$$\begin{array}{l}\vdash \exists! hom : (\alpha)term{\rightarrow}(\alpha)term. \\ \quad \forall k.\ hom(\mathsf{Con}\ k) = \mathsf{Con}\ k \wedge \\ \quad \forall x.\ hom(\mathsf{Var}\ x) = \mathsf{Var}\ x \wedge \\ \quad \forall t\, u.\ hom(\mathsf{App}\ t\ u) = \mathsf{App}\ (hom\ t)\ (hom\ u) \wedge \\ \quad \forall x\, u.\ hom(\mathsf{Lam}\ x\ u) = \mathsf{Abs}(\lambda y.\ hom(u[\mathsf{Var}\ y/x]))\end{array}$$

In the Lam equation we use Abs to reconstruct the abstraction. In the others, we can simply employ the appropriate constructor.

It is easy to see that this gives us a function that copies terms. The theorem states the unique existence of any function *hom* satisfying these equations. But the identity on terms, $\lambda u.\ u$, is just such a function (the actual proof makes use of the Abs axiom). Hence the function whose existence is asserted is itself the identity.

As will be seen in later sections, the Abs function may also be used more generally for getting fresh variables ('genvars') to supply to bodies of lambda-abstractions.

2 A Model of the Axioms

In this section we briefly recall the construction used by Gordon (1994), and discuss in some detail how to model axioms four and five. We begin with the free algebra of de Bruijn's nameless lambda-terms (de Bruijn 1972).

dCon	$: \alpha \rightarrow (\alpha)db$	(constants)
dVar	$: string \rightarrow (\alpha)db$	(free variables)
dBound	$: num \rightarrow (\alpha)db$	(bound variables)
dApp	$: (\alpha)db \rightarrow (\alpha)db \rightarrow (\alpha)db$	(applications)
dAbs	$: (\alpha)db \rightarrow (\alpha)db$	(lambda-abstractions)

Consider an occurrence of dBound i enclosed by j dAbs's in a term. If $i < j$ then it refers to the $(i+1)$'th enclosing dAbs. If $i \geq j$ then we say it is *dangling*, and that it is a *reference to parameter* $i - j$ of the term. We model $(\alpha)term$ by the *proper* de Bruijn terms, that is, those with no dangling indexes. Sometimes dangling indexes are used to represent free variables, but here we use the dVar constructor instead.

We can define $\mathsf{dFv}(d)$, the free variables of term d, by primitive recursion. Name-carrying lambda-abstraction and substitution can be defined as follows,

$$\vdash\ \mathsf{dLam}\ x\ d = \mathsf{dAbs}(\mathsf{Abst}\ 0\ x\ d)$$

$$\vdash\ d[d'/x] = \mathsf{Inst}\ 0\ (\mathsf{Abst}\ 0\ x\ d)\ d'$$

where the term Abst $i\ x\ d$ is obtained by turning each occurrence of dVar x in d into a reference to parameter i, and the term Inst $i\ d\ d'$ is obtained by instantiating each reference to parameter i in d to the term d'. An important property is that the set inductively defined from the constructors dCon, dVar, dApp and

dLam is exactly the set of proper de Bruijn terms. Given these definitions it is straightforward to model axioms one, two and three. See Gordon (1994) for a fuller discussion.

2.1 Soundness of the Iteration Axiom

First fix functions of the following types.

$$\begin{aligned}
&\mathit{con} : \alpha \to \beta\\
&\mathit{var} : \mathit{string} \to \beta\\
&\mathit{app} : \beta \to \beta \to \beta\\
&\mathit{abs} : (\mathit{string} \to \beta) \to \beta
\end{aligned}$$

To model axiom four, iteration, it suffices to construct a function *hom* such that

$$\begin{aligned}
&\forall k.\ \mathit{hom}(\mathsf{dCon}\ k) = \mathit{con}\ k\ \wedge\\
&\forall x.\ \mathit{hom}(\mathsf{dVar}\ x) = \mathit{var}\ x\ \wedge\\
&\forall d\ d'.\ \mathit{hom}(\mathsf{dApp}\ d\ d') = \mathit{app}\ (\mathit{hom}\ d)\ (\mathit{hom}\ d')\ \wedge\\
&\forall x\ d.\ \mathit{hom}(\mathsf{dLam}\ x\ d) = \mathit{abs}\ (\lambda y.\ \mathit{hom}\ (d[\mathsf{dVar}\ y/x]))
\end{aligned}$$

and moreover to show that *hom* is the unique function on proper de Bruijn terms to satisfy these equations. We shall refer to these equations as (Hom Spec).

The substitution in the last part of (Hom Spec) prevents us from defining *hom* by primitive recursion. Instead we define *hom* indirectly in terms of another function, *chom*, which uses Landin's idea (1964) of a closure to represent the substitution in the last part of (Hom Spec). Let the *degree* of a term be 0 if it contains no dangling index, and otherwise one more than the greatest parameter referred to by a dangling index. Let a *closure* be a pair (ys, d) with d a possibly improper de Bruijn term, and $ys{:}(\mathit{string})\mathit{list}$ a list of variable names of length no less than the degree of d. We can think of a closure $([y_0, \ldots, y_{n-1}], d)$ as standing for d with each reference to parameter i instantiated to $\mathsf{dVar}\ y_i$. We now define *chom ys d*, where (ys, d) is intended to be a closure, by primitive recursion on de Bruijn terms.

$$\begin{aligned}
&\forall k.\ \mathit{chom}\ \mathit{ys}\ (\mathsf{dCon}\ k) = \mathit{con}\ k\ \wedge\\
&\forall x.\ \mathit{chom}\ \mathit{ys}\ (\mathsf{dVar}\ x) = \mathit{var}\ x\ \wedge\\
&\forall i.\ \mathit{chom}\ \mathit{ys}\ (\mathsf{dBound}\ i) = \mathit{var}\ (i < \mathsf{Length}\ \mathit{ys} \Rightarrow \mathsf{El}\ i\ \mathit{ys} \mid \mathsf{Arb})\ \wedge\\
&\forall d\, d'.\ \mathit{chom}\ \mathit{ys}\ (\mathsf{dApp}\ d\ d') = \mathit{app}\ (\mathit{chom}\ \mathit{ys}\ d)\ (\mathit{chom}\ \mathit{ys}\ d')\ \wedge\\
&\forall x\, d.\ \mathit{chom}\ \mathit{ys}\ (\mathsf{dAbs}\ d) = \mathit{abs}\ (\lambda y.\ \mathit{chom}\ (\mathsf{Cons}\ y\ \mathit{ys})\ d)
\end{aligned}$$

The constant Arb has an arbitrary value, but provided that (ys, d) is a closure, *chom ys d* will not depend on Arb. We take *hom* to be *chom*[ ]. It is easy to see that this definition satisfies the first three equations in (Hom Spec). For the fourth, concerning dLam, we need a lemma that if any two closures (ys, d) and (ys', d') stand for the same lambda-term, in the sense given above, then

$\mathit{chom}\ ys\ d = \mathit{chom}\ ys'\ d'$. This is proved by structural induction on d, and allows us to calculate the following, for any d.

$$\begin{aligned}
\mathit{hom}(\mathsf{dAbs}\ d) &= \mathit{chom}\ [\,]\ (\mathsf{dAbs}\ d)\\
&= \mathit{abs}\ (\lambda y.\ \mathit{chom}\ [y]\ d)\\
&= \mathit{abs}\ (\lambda y.\ \mathit{chom}\ [\,]\ (\mathsf{Inst}\ 0\ d\ (\mathsf{dVar}\ y)))\\
&= \mathit{abs}\ (\lambda y.\ \mathit{hom}\ (\mathsf{Inst}\ 0\ d\ (\mathsf{dVar}\ y)))
\end{aligned}$$

By this, and the definitions of substitution and dLam the final part of (Hom Spec) follows.

$$\begin{aligned}
\mathit{hom}(\mathsf{dLam}\ x\ d) &= \mathit{hom}(\mathsf{dAbs}(\mathsf{Abst}\ 0\ x\ d))\\
&= \mathit{abs}\ (\lambda y.\ \mathit{hom}\ (\mathsf{Inst}\ 0\ (\mathsf{Abst}\ 0\ x\ d)\ (\mathsf{dVar}\ y)))\\
&= \mathit{abs}\ (\lambda y.\ \mathit{hom}\ (d[\mathsf{dVar}\ y/x]))
\end{aligned}$$

There does exist, then, a function *hom* satisfying (Hom Spec). Uniqueness follows by an induction on the length of the term, where the length of a de Bruijn term is the number of constructors it contains. (Length is definable by primitive recursion on de Bruijn terms.)

2.2 Soundness of the Abs Axiom

Here is a sketch of how to define a model for axiom five, concerning the Abs function. The essence of the proof is that Abs can be modelled by the function $\mathit{abs} : (\mathit{string} \to (\alpha)\mathit{db}) \to (\alpha)\mathit{db}$ defined by

$$\begin{aligned}
\mathit{abs}(f) = {}&\mathsf{let}\ Y = \cap y.\ \mathsf{dFv}(f\ y)\ \mathsf{in}\\
&\quad \mathsf{let}\ z = \mathsf{New}\ Y\ \mathsf{in}\\
&\qquad \mathsf{dLam}\ z\ (f\ z)
\end{aligned}$$

where New chooses a fresh string not in a given finite set of strings. The idea is that *abs* will be supplied with a function of the form $\lambda y.\ u[\mathsf{dVar}\ y/x]$ (that is, the body of a lambda-term). It then finds a fresh variable z and reconstructs the original lambda-term by building an alpha-equivalent one by substituting z into the body and abstracting over it using dLam.

The details are as follows. One can show that the free variables of $\mathsf{dLam}\ x\ u$ are a subset of the free variables of $u[\mathsf{dVar}\ y/x]$ for any choice of y:

$$\vdash \forall x\, y\, u.\ \mathsf{Proper}(u) \supset (\mathsf{dFv}(\mathsf{dLam}\ x\ u) \subseteq \mathsf{dFv}(u[\mathsf{dVar}\ y/x]))$$

The proof is by induction on length of the term u. Hence the free variables of $\mathsf{dLam}\ x\ u$ are contained in the intersection of the whole y-indexed family of free-variable sets:

$$\vdash \forall u.\ \mathsf{Proper}(u) \supset (\mathsf{dFv}(\mathsf{dLam}\ x\ u) \subseteq \cap y.\ \mathsf{dFv}(u[\mathsf{dVar}\ y/x]))$$

Moreover, one can show that the containing set is finite, since it is the intersection of a family of finite sets. (The function dFv always produces finite sets.) Hence one can find a fresh variable, avoiding any variable free in $\mathsf{dLam}\ x\ u$, by choosing a variable outside this finite set:

$$\vdash \forall u.\ \mathsf{Proper}(u) \supset \mathsf{New}(\cap y.\ \mathsf{dFv}(u[\mathsf{dVar}\ y/x])) \notin \mathsf{dFv}(\mathsf{dLam}\ x\ u)$$

Call this fresh variable z. But then by alpha conversion, we know that

$$\vdash \mathsf{dLam}\ z\ (u[\mathsf{dVar}\ z/x]) = \mathsf{dLam}\ x\ u$$

giving us the required lambda term.

3 Theorems Provable from the Axioms

This section discusses some of the theorems derivable from our five axioms. We begin by deriving a recursion scheme for defining functions over lambda-terms and then use it to prove a new principle of structural induction for terms. We then illustrate the utility of these results by defining a length function on terms.

Also provable from our axioms are the theorems of Gordon (1994) stating distinctness, injectivity and an exhaustion principle for the constructors. The substitution lemmas 1.14 and 1.15 of Hindley and Seldin (1986) are also provable. Discussion of these theorems is omitted here.

3.1 Recursion Scheme

The unique iteration axiom allows us to define functions only by structural iteration over terms. A more general definition pattern is supplied by the *recursion scheme* theorem:

$$\begin{array}{l}
\vdash \forall con : \alpha \to \beta. \\
\quad \forall var : string \to \beta. \\
\quad \forall app : \beta \to \beta \to (\alpha)term \to (\alpha)term \to \beta. \\
\quad \forall abs : (string \to \beta) \to (string \to (\alpha)term) \to \beta. \\
\quad \exists! hom : (\alpha)term \to \beta. \\
\quad\quad \forall k.\ hom(\mathsf{Con}\ k) = con\ k\ \wedge \\
\quad\quad \forall x.\ hom(\mathsf{Var}\ x) = var\ x\ \wedge \\
\quad\quad \forall t\, u.\ hom(\mathsf{App}\ t\ u) = app\ (hom\ t)\ (hom\ u)\ t\ u\ \wedge \\
\quad\quad \forall x\, u.\ hom(\mathsf{Lam}\ x\ u) = abs\ (\lambda y.\ hom(u[\mathsf{Var}\ y/x]))\ (\lambda y.\ u[\mathsf{Var}\ y/x])
\end{array}$$

Here, *hom* is defined not only in terms of its values on the subterms of each kind of constructor, but also in terms of the subterms themselves. In the defining equation for $hom(\mathsf{App}\ t\ u)$, the parameter *app* has access not just to $hom\ t$ and $hom\ u$ but also to t and u. Likewise, in the defining equation for $hom(\mathsf{Lam}\ x\ u)$ the parameter *abs* may use the body.

The recursion scheme follows from axioms four and five. Suppose we have arbitrary parameter functions similar to those in the recursion except that they are (in part) paired:

$$
\begin{array}{l}
\underline{con} : \alpha \to \beta \\
\underline{var} : \mathit{string} \to \beta \\
\underline{app} : (\beta \times (\alpha)\mathit{term}) \to (\beta \times (\alpha)\mathit{term}) \to \beta \\
\underline{abs} : (\mathit{string} \to (\beta \times (\alpha)\mathit{term})) \to \beta
\end{array}
$$

Then instantiate the unique iteration axiom with

$$
\begin{array}{ll}
\beta & := (\alpha)\mathit{term} \times \beta \\
\mathit{con} & := \lambda k.\ (\mathsf{Con}\ k, \underline{con}\ k) \\
\mathit{var} & := \lambda x.\ (\mathsf{Var}\ x, \underline{var}\ k) \\
\mathit{app} & := \lambda p\, q.\ (\mathsf{App}\ (\mathsf{Fst}\ p)\ (\mathsf{Fst}\ q), \underline{app}\ p\ q) \\
\mathit{abs} & := \lambda f{:}\mathit{string}{\to}((\alpha)\mathit{term} \times \beta).\ (\mathsf{Abs}(\mathsf{Fst} \circ f), \underline{abs}\ f)
\end{array}
$$

to get a unique function *hom* of type

$$(\alpha)\mathit{term} \to ((\alpha)\mathit{term} \times \beta)$$

that produces a pair consisting of a rebuilt copy of its input, together with a recursively calculated result of type β. At each stage in the recursion, a copy of the 'lower' structures is available, having been delivered by the recursive call.

Now *hom* produces a pair, and so can be split into unique pair of functions:

$$
\begin{array}{l}
\vdash \exists!(f,g) : ((\alpha)\mathit{term}{\to}(\alpha)\mathit{term}) \times ((\alpha)\mathit{term}{\to}\beta). \\
\quad \forall k.\ f(\mathsf{Con}\ k) = \mathsf{Con}\ k\ \wedge \\
\quad \forall x.\ f(\mathsf{Var}\ x) = \mathsf{Var}\ x\ \wedge \\
\quad \forall t\, u.\ f(\mathsf{App}\ t\ u) = \mathsf{App}\ (f\ t)\ (f\ u)\ \wedge \\
\quad \forall x\, u.\ f(\mathsf{Lam}\ x\ u) = \mathsf{Abs}(\lambda y.\ f(u[\mathsf{Var}\ y/x]))\ \wedge \\
\quad \forall k.\ g(\mathsf{Con}\ k) = \underline{con}\ k\ \wedge \\
\quad \forall x.\ g(\mathsf{Var}\ x) = \underline{var}\ x\ \wedge \\
\quad \forall t\, u.\ g(\mathsf{App}\ t\ u) = \underline{app}\ (f\ t, g\ t)\ (f\ u, g\ u)\ \wedge \\
\quad \forall x\, u.\ g(\mathsf{Lam}\ x\ u) = \underline{abs}\ (\lambda y.\ f(u[\mathsf{Var}\ y/x]), g(u[\mathsf{Var}\ y/x]))
\end{array}
$$

From these equations, one can easily see that f is copying the term by rebuilding it and g is computing the result using the copies produced by f along the way.

The next step is to observe that f must be the identity. As section 1.5 showed, any function satisfying the first four equations in this theorem equals $\lambda u.\, u$. We can therefore replace f by the identity in the defining equations for g, giving:

$$
\begin{array}{l}
\vdash \exists! g : (\alpha)\mathit{term}{\to}\beta. \\
\quad \forall k.\ g(\mathsf{Con}\ k) = \underline{con}\ k\ \wedge \\
\quad \forall x.\ g(\mathsf{Var}\ x) = \underline{var}\ x\ \wedge \\
\quad \forall t\, u.\ g(\mathsf{App}\ t\ u) = \underline{app}\ (t, g\ t)\ (u, g\ u)\ \wedge \\
\quad \forall x\, u.\ g(\mathsf{Lam}\ x\ u) = \underline{abs}\ (\lambda y.\ (u[\mathsf{Var}\ y/x]), g(u[\mathsf{Var}\ y/x]))
\end{array}
$$

But this is just the recursion scheme theorem, up to a little currying of the functions $\underline{app}$ and $\underline{abs}$.

This construction resembles Church's definition of the predecessor on natural numbers in the lambda-calculus (Church 1941). The actual construction used here was inspired, in part, by the derivation in Lambek and Scott (1986) of a recursion scheme for a natural numbers object in a cartesian closed category.

3.2 Deriving Induction

Gordon's previous work produced two principles of induction for lambda-terms. The first involves the standard notion of the *length* of a term, and the second involves specification of finite sets of variables from which one may assume certain bound variables are distinct. Both principles are derivable in the present setting from a more primitive notion of induction, which itself follows from the recursion scheme as usual.

The derivation of this induction principle proceeds as follows. We suppose here that P is a fixed but arbitrary predicate on lambda-terms. Then take

$$\begin{array}{ll}
\beta & := \mathit{bool} \\
\mathit{con} & := \lambda k.\, \mathsf{T} \\
\mathit{var} & := \lambda x.\, \mathsf{T} \\
\mathit{app} & := \lambda p\, q.\, \lambda t\, u.\, (p \wedge q) \vee P(\mathsf{App}\ t\ u) \\
\mathit{abs} & := \lambda f{:}\mathit{string}{\rightarrow}\mathit{bool}.\, \lambda g{:}\mathit{string}{\rightarrow}(\alpha)\mathit{term}.\, (\forall y.\ f\ y) \vee P(\mathsf{Abs}\ g)
\end{array}$$

in the recursion theorem to get

$$\begin{array}{l}
\vdash \exists! \mathit{hom} : (\alpha)\mathit{term} \rightarrow \mathit{bool}. \\
\quad \forall k.\ \mathit{hom}(\mathsf{Con}\ k) \wedge \\
\quad \forall x.\ \mathit{hom}(\mathsf{Var}\ x) \wedge \\
\quad \forall t\, u.\ \mathit{hom}(\mathsf{App}\ t\ u) = ((\mathit{hom}\ t) \wedge (\mathit{hom}\ u)) \vee P(\mathsf{App}\ t\ u) \wedge \\
\quad \forall x\, u.\ \mathit{hom}(\mathsf{Lam}\ x\ u) = (\forall y.\ \mathit{hom}(u[\mathsf{Var}\ y/x])) \vee P(\mathsf{Abs}\ \lambda y.\ u[\mathsf{Var}\ y/x])
\end{array}$$

Now, observe that $\lambda u.\, \mathsf{T}$ is just such a *hom* as is asserted to exist uniquely here. Hence any other function satisfying the above equations is constant true on the set of all lambda terms. In particular, the predicate P has this property, and so we have:

$$\begin{array}{l}
\vdash \forall k.\ P(\mathsf{Con}\ k) \wedge \\
\quad \forall x.\ P(\mathsf{Var}\ x) \wedge \\
\quad \forall t\, u.\ P(\mathsf{App}\ t\ u) = ((P\ t) \wedge (P\ u)) \vee P(\mathsf{App}\ t\ u) \wedge \\
\quad \forall x\, u.\ P(\mathsf{Lam}\ x\ u) = (\forall y.\ P(u[\mathsf{Var}\ y/x])) \vee P(\mathsf{Abs}(\lambda y.\ u[\mathsf{Var}\ y/x])) \\
\qquad \supset \\
\quad \forall u.\ P(u)
\end{array}$$

The Abs axiom lets us simplify this to

$$\begin{array}{l}\vdash \forall k.\ P(\mathsf{Con}\ k) \land \\ \quad \forall x.\ P(\mathsf{Var}\ x) \land \\ \quad \forall t\,u.\ P(\mathsf{App}\ t\ u) = ((P\ t) \land (P\ u)) \lor P(\mathsf{App}\ t\ u) \land \\ \quad \forall x\,u.\ P(\mathsf{Lam}\ x\ u) = (\forall y.\ P(u[\mathsf{Var}\ y/x])) \lor P(\mathsf{Lam}\ x\ u) \\ \qquad \supset \\ \quad \forall u.\ P(u)\end{array}$$

Finally, using the fact that $(A = B \lor A) = (B \supset A)$, we get our induction principle:

$$\begin{array}{l}\vdash \forall P\colon (\alpha)\mathit{term} \to \mathit{bool}. \\ \qquad \forall k.\ P(\mathsf{Con}\ k) \land \\ \qquad \forall x.\ P(\mathsf{Var}\ x) \land \\ \qquad \forall tu.\ P(t) \land P(u) \supset P(\mathsf{App}\ t\ u) \land \\ \qquad \forall x\,u.(\forall y.\ P(u[\mathsf{Var}\ y/x])) \supset P(\mathsf{Lam}\ x\ u) \\ \qquad\quad \supset \\ \qquad \forall u.\ P(u)\end{array}$$

For the Con, Var and App constructors, the proof obligations are just the same as ordinary structural induction. But in the Lam case, we may assume the induction hypothesis that P holds under all substitutions of a variable for the specific bound variable involved.

Examples

We can illustrate the induction principle just derived by using it to prove that the identity substitution has no effect:

$$\vdash \forall u\,z.\ u[\mathsf{Var}\ z/z] = u$$

This was an early lemma in Gordon's development and is part of one of Hindley and Seldin's substitution theorems.

The proof proceeds by induction on u. Only the Lam case is of any interest. The induction hypothesis is

$$\vdash \forall y\,z.\ (u[\mathsf{Var}\ y/x])[\mathsf{Var}\ z/z] = u[\mathsf{Var}\ y/x]$$

and we need to show

$$(\mathsf{Lam}\ x\ u)[\mathsf{Var}\ z/z] = \mathsf{Lam}\ x\ u$$

The case where $z = x$ is trivial, so let us suppose $z \neq x$. Now, specialise the induction hypothesis to get

$$\vdash (u[\mathsf{Var}\ x/x])[\mathsf{Var}\ z/z] = u[\mathsf{Var}\ x/x]$$

and apply $\mathsf{Lam}\ x$ to both sides:

$$\vdash \mathsf{Lam}\ x\ (u[\mathsf{Var}\ x/x][\mathsf{Var}\ z/z]) = \mathsf{Lam}\ x\ (u[\mathsf{Var}\ x/x])$$

Since $z \neq x$ we can draw the substitution for z outwards:

$$\vdash (\mathsf{Lam}\ x\ (u[\mathsf{Var}\ x/x]))[\mathsf{Var}\ z/z] = \mathsf{Lam}\ x\ (u[\mathsf{Var}\ x/x])$$

But, as observed in section 1.3, our alpha-conversion axiom tells us immediately that $\mathsf{Lam}\ x\ (u[\mathsf{Var}\ x/x])$ is just $\mathsf{Lam}\ x\ u$, and so we are finished.

The identity substitution theorem allows us to proceed to an inductive proof that free variables of a term are finite:

$$\vdash \forall u.\ \mathsf{Finite}(\mathsf{Fv}\ u)$$

This key theorem, whose actual proof we omit, lets us choose fresh variables not free in a given term u, since we know that there is always a string outside any finite set of strings (for example take a primed variant).

3.3 Definition of a Length Function

We now turn our attention to the problem of defining the standard notion of the *length* of a term in our theory. The length of a term is a count of the number of syntactic constructors in it. The concept is usually formalised by a function

$$\mathsf{Lgh} : (\alpha)\mathit{term} \to \mathit{num}$$

with the property

$$\begin{aligned}\vdash\ & \forall k.\ \mathsf{Lgh}(\mathsf{Con}\ k) = 1\ \wedge \\ & \forall x.\ \mathsf{Lgh}(\mathsf{Var}\ x) = 1\ \wedge \\ & \forall t\,u.\ \mathsf{Lgh}(\mathsf{App}\ t\ u) = (\mathsf{Lgh}\ t) + (\mathsf{Lgh}\ u)\ \wedge \\ & \forall x\,u.\ \mathsf{Lgh}(\mathsf{Lam}\ x\ u) = (\mathsf{Lgh}\ u) + 1\end{aligned}$$

But the equation for Lam in this recursive ‘definition’ does not conform to the pattern of our recursion scheme. We must therefore make a somewhat indirect definition, from which the above theorem is derivable. First, we need some machinery for handling sequences of variable renamings. This turns out to have rather general utility, and so is presented in some detail.

General Renamings

We are interested in arbitrary finite sequences of substitutions, which we may represent formally by an infix function

$$\mathsf{ISub} : (\alpha)\mathit{term} \to ((\alpha)\mathit{term} \times \mathit{string})\mathit{list} \to (\alpha)\mathit{term}$$

defined by primitive recursion on lists as follows:

$$\vdash \forall u.\ u\ \mathsf{ISub}\ [\,] = u\ \wedge\ \forall u\,t\,x\,\sigma.\ u\ \mathsf{ISub}\ (\mathsf{Cons}\ (t,x)\ \sigma) = (u[t/x])\ \mathsf{ISub}\ \sigma$$

The function takes a term and a list of term-variable substitution pairs and applies all the substitutions in sequence. We call this an *iterated* substitution, a name which incidentally serves to distinguish it from the simultaneous parallel substitution commonly seen in other contexts.

An *iterated renaming* is an iterated substitution of variables for variables. We define the predicate Renaming inductively as follows:

$\vdash \mathsf{Renaming}\ [\,]$	always
$\vdash \mathsf{Renaming}(\mathsf{Cons}\ (\mathsf{Var}\ x, y)\ \sigma)$	if $\mathsf{Renaming}\ \sigma$

As usual, the definition gives us rules (and HOL tactics) for the Renaming predicate, together with the corresponding rule induction principle.

Derivation of Length

We may now proceed to derive the desired length function. Begin by taking

$$\begin{aligned}
\beta\ &:= \mathit{num}\\
\mathit{con}\ &:= \lambda k.\, 1\\
\mathit{var}\ &:= \lambda x.\, 1\\
\mathit{app}\ &:= \lambda n\, m.\, \lambda t\, u.\, n + m\\
\mathit{abs}\ &:= \lambda f.\, \lambda g.\, \mathsf{let}\ v = \mathsf{New}(\mathsf{Fv}(\mathsf{Abs}\ g))\ \mathsf{in}\ f(v) + 1
\end{aligned}$$

in the recursion scheme theorem. Applying the Abs axiom gives us:

$$\begin{aligned}
\vdash \exists ! \mathit{hom} : (\alpha)\mathit{term} \to \mathit{num}.\\
&\forall k.\ \mathit{hom}(\mathsf{Con}\ k) = 1\ \wedge\\
&\forall x.\ \mathit{hom}(\mathsf{Var}\ x) = 1\ \wedge\\
&\forall t\, u.\ \mathit{hom}(\mathsf{App}\ t\ u) = (\mathit{hom}\ t) + (\mathit{hom}\ u)\ \wedge\\
&\forall x\, u.\ \mathit{hom}(\mathsf{Lam}\ x\ u) = \mathsf{let}\ v = \mathsf{New}(\mathsf{Fv}(\mathsf{Lam}\ x\ u))\ \mathsf{in}\ \mathit{hom}(u[\mathsf{Var}\ v/x]) + 1
\end{aligned}$$

The key idea is that in the last equation, we have used Abs to reconstruct the term Lam x u. We can then generate a fresh variable Var v not free in this term, substitute this variable into the body, and then take the length of the result.

The next step is to show that the choice of variable to substitute into the body can, in fact, be made arbitrarily. We prove that if *hom* is as defined above, then

$$\vdash \forall u\, \sigma.\ \mathsf{Renaming}\ \sigma \supset (\mathit{hom}\ (u\ \mathsf{ISub}\ \sigma) = \mathit{hom}\ u)$$

So *hom* is invariant under iterated variable renaming. The proof is a straightforward induction on u, using alpha conversion to avoid variable name clashes in the Lam case.

Since a single substitution is also a renaming, we can replace $\mathit{hom}(u[\mathsf{Var}\ v/x])$ in our indirect definition with *hom* u. The fresh variable v then no longer plays a role and can be eliminated, giving equations for the length function in exactly the desired form.

4 An Application of the Axioms

This section shows how we can derive syntax for a particular programming language as a set of abbreviations of untyped lambda-terms. Our particular example is pi-calculus. The syntax of pi-calculus is built up from a denumerable set of *names*, x, y or z, and the set of *processes*, p or q, given by the following syntax.

$p ::=$	processes
$\overline{x}y.p$	(send)
$x(y).p$	(receive, y bound)
$p \mid q$	(parallel composition)
$(\nu x)p$	(restriction, x bound)
$\mathbf{0}$	(zero process)

By convention binding occurrences are parenthesised. This is a particularly simple, finitary pi-calculus, but its syntax suffices to make our point, which is to demonstrate how this syntax can be encoded using lambda-terms. We shall not discuss the operational semantics of pi-calculus, though it too can be represented within HOL.

We shall encode pi-calculus by introducing a new syntactic constructor for each kind of process. We introduce a syntactic constructor k of arity n by the following definition scheme, where $\underline{k}$ is a *string* constant representing k:

$$\vdash k\ t_1\ t_2 \cdots\ t_n = \mathsf{App}\ (\cdots \mathsf{App}\ (\mathsf{App}\ \underline{k}\ t_1)\ t_2 \cdots)\ t_n$$

String constants and binary applications suffice to encode syntactic constructors of arbitrary arity. We wrote a simple tool to automate this scheme. Axioms one and two generalise to such constructors as follows.

$$\vdash \mathsf{Fv}(k\ t_1 \cdots\ t_n) = \mathsf{Fv}\ t_1 \cup \cdots\ \cup \mathsf{Fv}\ t_n$$

$$\vdash (k\ t_1 \cdots\ t_n)[u/x] = k\ (t_1[u/x]) \cdots\ (t_n[u/x])$$

To encode pi-calculus we introduce syntactic constructors Send, Recv, Par, Res and Zero, with arities 3, 2, 2, 1 and 0 respectively. We represent a free occurrence of a name x by the lambda-term $\mathsf{Var}\ x$, and binding occurrences by $\mathsf{Lam}\ x$. Given this preparation we can represent the syntax above by the following inductive definition of a predicate, Proc, on lambda-terms of type $(\mathit{string})\mathit{term}$.

$\vdash \mathsf{Proc}(\mathsf{Send}\ (\mathsf{Var}\ x)\ (\mathsf{Var}\ y)\ p)$	if $\mathsf{Proc}\ p$	$(\overline{x}y.p)$
$\vdash \mathsf{Proc}(\mathsf{Recv}\ (\mathsf{Var}\ x)\ (\mathsf{Lam}\ y\ p))$	if $\mathsf{Proc}\ p$	$(x(y).p)$
$\vdash \mathsf{Proc}(\mathsf{Par}\ p\ q)$	if $\mathsf{Proc}\ p$ and $\mathsf{Proc}\ q$	$(p \mid q)$
$\vdash \mathsf{Proc}(\mathsf{Res}\ (\mathsf{Lam}\ x\ p))$	if $\mathsf{Proc}\ p$	$((\nu x)p)$
$\vdash \mathsf{Proc}(\mathsf{Zero})$	always	$(\mathbf{0})$

Our axiomatised type of lambda-terms allows us to formalise the syntax given at the beginning of the section by this inductive definition within HOL. Rule induction on the Proc predicate formalises structural induction on pi-calculus processes.

For instance, to prove that the set of processes is closed under substitution of a name for a name,

$$\vdash \forall p.\ \mathsf{Proc}\ p \supset \forall x\,y.\ \mathsf{Proc}\ (p[\mathsf{Var}\ x/y])$$

we prove the more general hypothesis that the set of processes is closed under iterated variable renaming,

$$\vdash\ \forall p.\ \mathsf{Proc}\ p \supset \forall \sigma.\ \mathsf{Renaming}\ \sigma \supset \mathsf{Proc}\ (p\ \mathsf{ISub}\ \sigma)$$

by rule induction on Proc, that is, structural induction.

5 Related Work

The idea of a metatheory of syntax has a long history, going back at least to Church's encoding of higher-order logic within simply-typed lambda-calculus. Martin-Löf's theory of arities is essentially the same idea (Nordström, Petersson, and Smith 1990). The idea is now widely used to represent syntax in theorem-provers such as Paulson's Isabelle (1994). Church, Martin-Löf and Paulson all encode syntax using simply-typed lambda-terms, identified up to alpha-beta-conversion. Types are needed to avoid meaningless divergent terms. We encode syntax using untyped lambda-terms, identified only up to alpha-conversion. We need to represent substitution as a separate function; in a system with beta-conversion it is represented simply as application of an abstraction to a term. On the other hand, our metatheory based on alpha-conversion supports structural induction more directly than one based on alpha-beta-conversion, where one would need to perform induction on the size of the normal-form of a term.

We have advocated representing syntax as a type within a mechanised logic. This is sometimes known as 'deep embedding' (Boulton, Gordon, Gordon, Harrison, Herbert, and Van Tassel 1992). Embeddings based either on de Bruijn terms or a free algebra of name-carrying terms are now quite common (see Gordon (1994) for a survey). We are aware of several recent strands of work on deep embedding that focus on the interaction between substitution and bound variables. Talcott (1993) proposed a generic theory of binding structures, now implemented in Isabelle by Matthews (1995). McKinna and Pollack (1993) proposed a scheme of binding based on two kinds of variables, that allows a straightforward definition of substitution and yet avoids the possibility of variable capture. They implemented it in Lego (Pollack 1994), and it has recently been re-implemented in Isabelle (Owens 1995). Our axiom five, which relates logical and embedded abstractions, is reminiscent of higher-order abstract syntax (Pfenning and Elliott 1988; Despeyroux and Hirschowitz 1994), in which variable binding in the embedded syntax is implemented via the lambda-abstraction in the logic itself.

6 Conclusions

We advocated untyped lambda-terms, identified up to alpha-conversion, as a metatheory suitable for representing the syntax of a formalism within a logic. An application would be proofs about the operational semantics of pi-calculus. Towards this end we proposed five axioms of such lambda-terms, showed them sound for de Bruijn terms, and derived a collection of useful theorems. All the proofs have been checked in HOL.

The main improvements in this paper with respect to Gordon (1994) are the presentation of five basic axioms—which could confidently be postulated in some other theorem prover—and the possibility of defining functions by iteration and primitive recursion on lambda-terms.

Acknowledgement Gordon is supported by a Royal Society University Research Fellowship. We are grateful to our colleagues at Cambridge and Glasgow and the anonymous referees for their comments.

References

Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics* (Revised ed.), Volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland.

Boulton, R., A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel (1992). Experience with embedding hardware description languages in HOL. See Stavridou, Melham, and Boute (1992), pp. 129–156.

Church, A. (1941). *The Calculi of Lambda-Conversion*. Princeton University Press.

Curry, H. B. and R. Feys (1958). *Combinatory Logic*, Volume 1. North-Holland.

de Bruijn, N. G. (1972). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae 34*, 381–392.

Despeyroux, J. and A. Hirschowitz (1994, July). Higher-order abstract syntax with induction in Coq. In F. Pfenning (Ed.), *Fifth International Conference on Logic Programming and Automated Reasoning (LPAR'94), Kiev*, Volume 882 of *LNAI*, pp. 159–173. Springer-Verlag.

Gordon, A. D. (1994). A mechanisation of name-carrying syntax up to alpha-conversion. In J. J. Joyce and C.-J. H. Seger (Eds.), *Higher Order Logic Theorem Proving and its Applications. Proceedings, 1993*, Number 780 in Lecture Notes in Computer Science, pp. 414–426. Springer-Verlag.

Gordon, M. J. C. and T. F. Melham (Eds.) (1993). *Introduction to HOL: A theorem-proving environment for higher-order logic*. Cambridge University Press.

Hindley, J. R. and J. P. Seldin (1986). *Introduction to Combinators and the λ-calculus*. Cambridge University Press.

Lambek, J. and P. J. Scott (1986). *Introduction to higher order categorical logic*. Cambridge University Press.

Landin, P. J. (1964, January). The mechanical evaluation of expressions. *Computer Journal 6*, 308–320.

Matthews, S. (1995, September). Implementing FS_0 in Isabelle: adding structure at the metalevel. In L. C. Paulson (Ed.), *Proceedings of the First Isabelle Users Workshop*. Available as Technical Report 379, University of Cambridge Computer Laboratory.

McKinna, J. and R. Pollack (1993). 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*, pp. 289–305. Springer-Verlag.

Melham, T. F. (1994). A mechanized theory of the π-calculus in HOL. *Nordic Journal of Computing 1*, 50–76.

Milner, R., J. Parrow, and D. Walker (1992). A calculus of mobile processes, parts I and II. *Information and Computation 100*, 1–40 and 41–77.

Nordström, B., K. Petersson, and J. M. Smith (1990). *Programming in Martin-Löf's Type Theory*. Clarendon Press, Oxford.

Owens, C. (1995, September). Coding binding and substitution explicitly in Isabelle. In L. C. Paulson (Ed.), *Proceedings of the First Isabelle Users Workshop*. Available as Technical Report 379, University of Cambridge Computer Laboratory.

Paulson, L. C. (1994). *Isabelle: A Generic Theorem Prover*, Volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag.

Pfenning, F. and C. Elliott (1988, June). Higher-order abstract syntax. In *Proceedings of the ACM SIGPLAN '88 Symposium on Language Design and Implementation*, pp. 199–208.

Pollack, R. (1994). *The Theory of LEGO*. Ph. D. thesis, University of Edinburgh.

Stavridou, V., T. F. Melham, and R. T. Boute (Eds.) (1992). IFIP Transactions A-10. North-Holland.

Stoughton, A. (1988). Substitution revisited. *Theoretical Computer Science 59*, 317–325.

Talcott, C. L. (1993). A theory of binding structures and applications to rewriting. *Theoretical Computer Science 112*, 99–143.