Recursive Function Definition for Types with Binders

Michael Norrish
Michael.Norrish@nicta.com.au

Canberra Research Laboratory, National ICT Australia
Research School of Information Science and Engineering,
Australian National University,
Acton 0200, AUSTRALIA

Abstract. This work describes the proof and uses of a theorem allowing definition of recursive functions over the type of λ-calculus terms, where terms with bound variables are identified up to α-equivalence. The theorem embodies what is effectively a principle of primitive recursion, and the analogues of this theorem for other types with binders are clear. The theorem's side-conditions require that the putative definition be well-behaved with respect to fresh name generation and name permutation. A number of examples over the type of λ-calculus terms illustrate the use of the new principle.

1 Introduction

Theorem-proving tools have long supported the definition of (potentially recursive) algebraic or inductive types. Not only do the tools prove the existence of such types, and establish them within the logical environment, but they also provide methods for defining new functions over those types. Typically this is done by proving and using the new type's recursion theorem.

For example, a definition of a type of lists would assert that the new type had two constructors: nil and cons, and that the cons constructor took two arguments, an element of the parameter type α, and another list. The recursion theorem proved to accompany this type would state:

$$\begin{array}{l} \forall n\, c.\ \exists h. \\ \quad h(\mathsf{nil}) = n\ \wedge \\ \quad \forall a\, t.\ \ h(\mathsf{cons}(a,t)) = c(a,t,h(t)) \end{array}$$

This theorem states that given any n, specifying the value of the function-to-be when applied to empty lists, and given any c, specifying what should happen when the function is applied to a "cons-cell", there exists a function h that exhibits the desired behaviour. The cons behaviour, c, may refer to the component parts of the list, a and t, as well as the result of h's action on t.

For example, the existence of the map function can be demonstrated by instantiating the recursion theorem so that n is $\lambda f.\ \mathsf{nil}$, and c is $\lambda(a,\,t,\,r)\,f.\ \mathsf{cons}(f(a),r(f))$. Note how map's additional parameter has been accommodated by making the range of the function h itself a function-space.

In the friendlier world that users expect to inhabit, the user provides a definition for map that looks like

$$\begin{aligned} \mathsf{map}\ f\ \mathsf{nil} &= \mathsf{nil} \\ \mathsf{map}\ f\ (\mathsf{cons}(h,t)) &= \mathsf{cons}(f(h), \mathsf{map}\ f\ t) \end{aligned}$$

It is then the responsibility of the tool implementation to recognise that this is a primitive recursion over a list, to instantiate the recursion theorem appropriately, and to manipulate the resulting theorem so that it again looks like what the user specified. If, for example, the tool instantiates the theorem as above, it proves the existence of a map function with its parameters in the wrong order; a little more work is required to demonstrate the existence of the function that takes its function parameter first.

Finally, note that when a multi-parameter function's other arguments do not change in the recursive call (as happens with map, but not, for example, with foldl), it is also possible to instantiate the theorem differently, but to the same ultimate effect. In the case of map, n would be set to nil, and c to $\lambda(a,t,r).\ \mathsf{cons}(f(a),r)$. The resulting instantiation of the recursion theorem would have f free. This could be generalised, giving:

$$\begin{aligned} &\forall f.\ \exists h. \\ &\quad h(\mathsf{nil}) = \mathsf{nil}\ \wedge \\ &\quad \forall a\, t.\ h(\mathsf{cons}(a,t)) = \mathsf{cons}(f(a), h(t)) \end{aligned}$$

An appeal to the Axiom of Choice[1] (skolemisation) then moves the variable h out over the universally quantified f, demonstrating the existence of an h taking two parameters. This trick is not necessary with primitive recursive functions over normal inductive types, but it will be useful in some of the examples below.

This much is well-understood technology. Unfortunately, there is no comparable, simple story to be told about the type of λ-calculus terms where bound variables are identified up to α-equivalence (or indeed, any type featuring α-equivalence). Section 7 discusses other approaches to this problem. Presented here is a new approach, based on two significant sources: Gordon and Melham's characterisation of α-equivalent λ-calculus terms [5], and the Gabbay-Pitts idea of name (or atom) permutation as the basis for syntax with binding [4].

Gordon and Melham's work is a significant starting point because it defines a type in HOL (classical simple type theory) exactly corresponding to the type of (untyped) λ-calculus terms, augmented with a "constant" constructor allowing the injection of any other arbitrary type into the terms. It corresponds to a type that one might declare in SML as

```
datatype 'a term = CON of 'a
                 | VAR of string
                 | APP of 'a term * 'a term
                 | LAM of string * 'a term
```

[1] The recursion theorem can be strengthened so that $\exists h$ turns into $\exists! h$. Moving this out past universal quantifiers is then only an appeal to the Axiom of Definite Choice.

except that the bodies of abstractions (under the `LAM` constructor) are identified up to α-equivalence.

Accompanying this type are the core theorems and constants that identify it as an implementation of the λ-calculus. There is a substitution function, a function for calculating free variables, and various theorems that describe how these functions behave. There are two further important facts about the Gordon-Melham work:

- Despite their potentially misleading title, "Five axioms of alpha conversion", Gordon and Melham did not assert any new HOL axioms. Their type is constructed entirely definitionally, on top of a model of de Bruijn terms.
- Their theory of terms is first-order. By α-equivalence, the following equation holds

$$\texttt{LAM}\ v\ (\texttt{VAR}\ v) = \texttt{LAM}\ u\ (\texttt{VAR}\ u)$$

  but `LAM` is not a binder at the logical level, and there are no function spaces used. The Gordon-Melham theory is not one of higher-order abstract syntax, and there are no exotic terms.

Using the `CON` constructor, it is straightforward to construct new types with binders on top of the basic Gordon-Melham terms. In earlier work [7], I implemented the types Λ' and Λ'^{*} from Barendregt [2], and proved finiteness of developments and the standardisation theorem. That work demonstrated that the Gordon-Melham theory is a viable basis for theorem-proving with binders.

Nonetheless, in this earlier work, I had to manually define the new types, and almost all of the various functions over them. This sort of work is painful and a significant obstacle for many users. The current work describes technology for solving one of these two important problems, that of function definition. The other problem, that of defining new types, is another significant project in its own right.

My second inspiration, Gabbay's and Pitts's ideas about permutation as a basis for syntax with binders, is itself an independent approach to the problem of recursive function definition. It is discussed in this role in Section 7. My work attempts to take the permutation idea and move it into a setting where some of its fundamental assumptions no longer apply. This is valuable because permutations exhibit properties, even in HOL's classical simple type theory, that make them much easier to work with than substitutions.

The rest of this paper is arranged as follows: Section 2 provides a series of motivating examples, designed to illustrate a range of different problems in function definition. Section 3 is a discussion of how the Gordon-Melham recursion principle can be slightly adjusted, enabling the definition of a size function. Section 4 describes how a permutation or `swap` function can be defined using the same principle. Section 5 then presents the derivation of the final recursion principle. Section 6 describes how the principle forms the basis for an automatic tool for performing function definition, and how it copes with the examples of Section 2. I discuss related work in Section 7, and conclude in Section 8.

2 Motivating examples

The following functions, with their increasing complexity, provide a test for any principle of function definition. They are presented here in the form in which users would want to write them, mimicking how one might write them in a functional language with pattern-matching.

Each of the given functions respects the α-equivalence relation. A clause of the form

$$f(\texttt{LAM}\ v\ t) = E$$

has equal values E for every possible renaming of the bound variable v (possibly subject to side-conditions on the equation, see below). If $f(\texttt{LAM}\ v\ t)$ were E, but $f(\texttt{LAM}\ u\ (t[v \mapsto u]))$ were E', and these expressions had different values, then this would be a contradiction: the two input terms are equal, so their f-values must be equal too. This work's new recursion principle embodies restrictions which ensures that the new functions are well-behaved in this respect.

Case analysis: The `is_app` function distinguishes constructors without looking at their arguments.

```
is_app (CON k)   = F        is_app (VAR s)   = F
is_app (APP t u) = T        is_app (LAM v t) = F
```

Examining constructor arguments: The `rator` function pulls apart an application term and returns the first argument. On other types of term, its value is unspecified.

```
rator (APP t u) = t
```

There is a sister function, `rand` which returns the other argument of an `APP`.

Simple recursion: The `size` function returns a numeric measurement of the size of a term.

```
size (CON k)   = 1
size (VAR s)   = 1
size (APP t u) = 1 + size t + size u
size (LAM v t) = 1 + size t
```

Recursion mentioning a bound variable: The `enf` function is true of a term if it is in η-normal form. (The `FV` function returns the set of a term's free variables.)

```
enf (CON k)   = T
enf (VAR s)   = T
enf (APP t u) = enf t ∧ enf u
enf (LAM v t) = enf t ∧
                (is_app t ∧ rand t = VAR v ⇒
                 v ∈ FV (rator t))
```

Simple recursion (terms as range type): The (admittedly artificial) `stripc` function replaces all `CON` terms with $(\lambda x.x)$.

```
  stripc (CON k)    = LAM "x" (VAR "x")
  stripc (VAR s)    = VAR s
  stripc (APP t u) = APP (stripc t) (stripc u)
  stripc (LAM v t) = LAM v (stripc t)
```

Recursion with an additional parameter: Given the ternary type of possible directions to follow when passing through a term ({`Lt`,`Rt`,`In`}), corresponding to the two sub-terms of an `APP` constructor and the body of an abstraction, return the set of paths (lists of directions) to the occurrences of the given free variable in a term.

```
  v_posns v (VAR s)    = if s = v then {[]} else ∅
  v_posns v (CON k)    = ∅
  v_posns v (APP t u) = (IMAGE (CONS Lt) (v_posns v t))
                                    ∪
                          (IMAGE (CONS Rt) (v_posns v u))
v ≠ x ⇒
  v_posns v (LAM x t) = IMAGE (CONS In) (v_posns v t)
```

The `IMAGE (CONS` x`)` construction above takes a set and adds x to the front of all its elements (which are all lists). After this definition is made, it is easy to prove (by induction) that

$$v \notin \mathrm{FV}(t) \quad \Rightarrow \quad \texttt{v\_posns}\ v\ t\ = \emptyset$$

Another useful `LAM` clause immediately follows:

```
  v_posns v (LAM v t) = ∅
```

One advantage of the new recursion principle is that it automatically derives the side-condition attached to the `LAM`-clause above, necessary to make it valid.

Recursion with varying parameters (terms as range): A variant of the substitution function, which substitutes a term for a variable, but further adjusts the term being substituted by wrapping it in one application of the variable `"f"` per binder traversed.

```
  sub′ M v (VAR s)    = if v = s then M else VAR s
  sub′ M v (CON k)    = CON k
  sub′ M v (APP t u) = APP (sub′ M v t) (sub′ M v u)

v ≠ x ∧ "f" ≠ x ∧ x ∉ FV(M) ⇒
  sub′ M v (LAM x t) =
     LAM x (sub′ (APP (VAR "f") M) v t)
```

Again, the preconditions on the `LAM`-clause in this example ensure that the function respects α-equivalence. This function can be given another clause for the `LAM` constructor in the same way as for `v_posns` above, giving

```
  sub′ M v (LAM v t) = LAM v t
```

Even with this addition, the equations may not seem to provide a complete specification of the behaviour of the function. What, for example, is the behaviour if the bound variable is `"f"`? In fact, the function is well-defined, but its value may need to be calculated by first α-converting an abstraction to use a new bound variable. That this is always possible is guaranteed by the new recursion principle: it requires that there be only finitely many names to which a bound variable can not be renamed. Here, the unavailable names are v, `"f"` and the names in `FV(M)`.

3 The recursion principle: first steps

One of the Gordon-Melham theorems characterising the type of λ-calculus terms is the following principle of recursion (where $t[v \mapsto u]$ is a capture-avoiding substitution of a term u for a variable v throughout term t):

$$\begin{array}{l}
\forall con\ var\ app\ lam.\\
\quad \exists hom.\\
\qquad (\forall\, k.\ \ hom(\mathtt{CON}\, k) = con(k)) \land\\
\qquad (\forall\, s.\ \ hom(\mathtt{VAR}\, s) = var(s)) \land\\
\qquad (\forall\, t\ u.\ \ hom(\mathtt{APP}\ t\ u) = app\ (hom\, t)\ (hom\, u)\ t\ u) \land\\
\qquad (\forall\, v\, t.\ \ hom(\mathtt{LAM}\ v\ t) =\\
\qquad\qquad\qquad lam\ (\lambda y.\ hom(t[v \mapsto \mathtt{VAR}(y)]))\ (\lambda y.\ t[v \mapsto \mathtt{VAR}(y)]))
\end{array} \tag{1}$$

This differs from the usual form of a recursion theorem in the clause for `LAM`. The *lam* function is not passed the result of a recursive call, but a function instead. This function takes a string, substitutes it for the bound variable through the body, and returns the result of the recursive function applied to this. Similarly, rather than getting access to the body of the abstraction directly, *lam* only gets to see it hidden behind another function that performs a substitution.

The last of the Gordon-Melham "axioms" states the existence of a function `ABS` such that

$$\mathtt{LAM}\ v\ t \;\;=\;\; \mathtt{ABS}(\lambda y.\ t[v \mapsto \mathtt{VAR}(y)]) \tag{2}$$

Now instantiate *lam* of (1) with

$$\lambda f\ g.\ \ \mathsf{let}\, z = \mathtt{NEW}(\mathtt{FV}(\mathtt{ABS}(g)) \cup X)\ \ \mathsf{in}\ \ lam'\ (f\ z)\ z\ (g\ z)$$

The `NEW` function takes a finite set of strings, and returns a string not in that set.

Using (2), the last clause of the recursion theorem becomes

$$\begin{array}{l}
\forall v\, t.\ hom(\mathtt{LAM}\ v\ t) \;=\\
\qquad \mathsf{let}\, z = \mathtt{NEW}(\mathtt{FV}(\mathtt{LAM}\ v\ t) \cup X)\ \ \mathsf{in}\\
\qquad\quad lam'\ \ (hom(t[v \mapsto \mathtt{VAR}(z)]))\ z\ \ (t[v \mapsto \mathtt{VAR}(z)])
\end{array} \tag{3}$$

This introduces two new free variables into the theorem: lam', which now gets direct access to the result of a recursion, a bound variable and a term body; and X, an additional set of variables that is to be avoided in the choice of z.

This is a generalisation of the technique that Gordon and Melham use in [5] to define their Lgh ("length") function (similar to `size` in Section 2 above). The extra X

parameter will be vital in defining permutation in Section 4 below. It is also important to have access to the new name z, which stands in for a bound variable that has been renamed to be fresh. Though the new recursion theorem has made the types involved in the `LAM` clause slightly more palatable, the recursive call in the `LAM` clause is still over a term that has had a substitution applied to it.

In the case of `size` (as done in [5]), it is possible to separately prove by induction that `size` is invariant under variable renamings, that $\mathtt{size}(t[v \mapsto \mathtt{VAR}(y)]) = \mathtt{size}(t)$. This simplifies the `LAM`-clause so that the reference to the fresh z can disappear. This trick is not strong enough in general. It doesn't work for the `stripc` example function, as it is not the case that $\mathtt{stripc}(t[v \mapsto \mathtt{VAR}(y)]) = \mathtt{stripc}(t)$. Still, the `size` function is needed to perform induction on the size of terms, and so this first attempt at a definitional principle is used to define the `size` constant. This preliminary principle also helps with the definition of `swap` (see below), before being discarded.

4 Permutation in HOL

Next, the system must be extended with definitions of name permutation for all of the relevant types.[2] Because variables in the λ-calculus terms are of type string, names are taken to be strings. The basic action of permutation on strings is simple to define:

$$\mathtt{swapstr}\ x\ y\ s \quad \hat{=} \quad \textsf{if}\ x = s\ \textsf{then}\ y\ \textsf{else}\ (\textsf{if}\ y = s\ \textsf{then}\ x\ \textsf{else}\ s)$$

Defining a swap function over terms requires the use of the new version of the `LAM`-clause (3) and the original principle (1), with the following instantiations[3]

$$\begin{array}{lcl}
var & \mapsto & \lambda s.\ \mathtt{VAR}(\mathtt{swapstr}\ x\ y\ s) \\
con & \mapsto & \mathtt{CON} \\
app & \mapsto & \lambda rt\ ru\ t\ u.\ \mathtt{APP}\ rt\ ru \\
X & \mapsto & \{x,y\} \\
lam' & \mapsto & \lambda rt\ v\ t.\ \mathtt{LAM}\ v\ rt
\end{array}$$

After generalising over x and y, and then applying the Axiom of Choice (skolemising), this results in the following theorem:

$$\begin{array}{l}
\exists \mathtt{swap}.\ \forall x\ y. \\
\quad (\forall s.\ \mathtt{swap}\ x\ y\ (\mathtt{VAR}\ s) = \mathtt{VAR}(\mathtt{swapstr}\ x\ y\ s)) \land \\
\quad (\forall k.\ \mathtt{swap}\ x\ y\ (\mathtt{CON}\ k) = \mathtt{CON}\ k) \land \\
\quad (\forall t\ u.\ \mathtt{swap}\ x\ y\ (\mathtt{APP}\ t\ u) = \mathtt{APP}\ (\mathtt{swap}\ x\ y\ t)\ (\mathtt{swap}\ x\ y\ u)) \land \\
\quad (\forall v\ t.\ \mathtt{swap}\ x\ y\ (\mathtt{LAM}\ v\ t) = \\
\qquad\qquad \textsf{let}\ z = \mathtt{NEW}(\mathtt{FV}(\mathtt{LAM}\ v\ t) \cup \{x,y\})\ \textsf{in} \\
\qquad\qquad\quad \mathtt{LAM}\ z\ (\mathtt{swap}\ x\ y\ (t[v \mapsto \mathtt{VAR}(z)])))
\end{array} \tag{4}$$

[2] Gabbay and Pitts use the notation $(x\,y) \cdot t$ to mean the permutation of x and y in t, where x and y are names and t is generally of any type. In what follows, I use a wordier, but more explicit, notation, where each swapping function is given a different name depending on the type of the third argument.

[3] The rt and ru names are chosen because these parameters correspond to the results of recursive calls on t and u parameters respectively.

This suffices as a definition for a new constant swap, but the LAM-clause is unacceptable as it stands. It needs to be shown that

$$\texttt{swap}\ x\ y\ (\texttt{LAM}\ v\ t) \;=\; \texttt{LAM}\ (\texttt{swapstr}\ x\ y\ v)\ (\texttt{swap}\ x\ y\ t)$$

This can be done by first showing that swap distributes over substitutions of variables for variables:

$$\begin{array}{l}\texttt{swap}\ x\ y\ (t[u \mapsto \texttt{VAR}(v)]) = \\ \quad (\texttt{swap}\ x\ y\ t)[\texttt{swapstr}\ x\ y\ u \mapsto \texttt{VAR}(\texttt{swapstr}\ x\ y\ v)]\end{array} \tag{5}$$

Glossing over some of the details, this theorem suffices because it allows the swap and the substitution to move past each other in the LAM-clause of (4), and for the LAM z $(\ldots)$ there to be recognised as equal (through α-equivalence) to LAM $(\texttt{swapstr}\ x\ y\ v)$ $(\ldots)$.

The proof of (5) is by induction on the size of t.

The next important property of swap is that it can be used instead of substitution when a fresh variable is being substituted for another:

$$v \notin \texttt{FV}(t) \quad \Rightarrow \quad t[u \mapsto \texttt{VAR}(v)] = \texttt{swap}\ u\ v\ t \tag{6}$$

This means that α-equivalence can be expressed using swap:

$$v \notin \texttt{FV}(t) \quad \Rightarrow \quad \texttt{LAM}\ u\ t = \texttt{LAM}\ v\ (\texttt{swap}\ u\ v\ t)$$

This much confirms that the Gordon-Melham λ-calculus terms can be equipped with a permutation action that behaves as the Gabbay-Pitts theory requires.

5 A new recursion principle

The aim of this work is the proof of a recursion principle with a LAM clause that looks, as much as possible, like

$$\forall v\,t.\ \mathit{hom}(\texttt{LAM}\ v\ t) = \mathit{lam}'\ (\mathit{hom}(t))\ v\ t \tag{7}$$

How does one start with (3), that is:

$$\begin{array}{l}\forall v\,t.\ \mathit{hom}(\texttt{LAM}\ v\ t)\ = \\ \qquad \mathsf{let}\ z = \texttt{NEW}(\texttt{FV}(\texttt{LAM}\ v\ t) \cup X)\ \mathsf{in} \\ \qquad\quad \mathit{lam}'\ (\mathit{hom}(t[v \mapsto \texttt{VAR}(z)]))\ z\ (t[v \mapsto \texttt{VAR}(z)])\end{array}$$

and derive (7)? And what extra side-conditions need to be added to make the transformation valid?

A simple examination of the two formulas suggests that the desired strategy would be to pull out the substitutions so that there was just one, at the top-level underneath the let, and to then have that substitution "evaporate" somehow. The essence of the principle-to-come is the side-condition that allows this.

The first observation is that permutations move around terms much more readily than substitutions. Secondly, the freshness of z (it is the result of a call to `NEW`) and (6) mean that the substitutions in (3) can be replaced by permutations, giving

$$\begin{array}{l}\forall v\,t.\ \mathit{hom}(\texttt{LAM}\ v\ t)\ = \\ \quad \textsf{let}\ z = \texttt{NEW}(\texttt{FV}(\texttt{LAM}\ v\ t) \cup X)\ \textsf{in} \\ \qquad \mathit{lam}'\ (\mathit{hom}(\texttt{swap}\ z\ v\ t))\ (\texttt{swapstr}\ z\ v\ v)\ (\texttt{swap}\ z\ v\ t)\end{array}$$

To move the `swap` terms upwards, one would clearly need that

$$\mathit{hom}(\texttt{swap}\ x\ y\ t) = \texttt{swap}\ x\ y\ (\mathit{hom}(t)) \tag{8}$$

and that

$$\mathit{lam}'\ (\texttt{swap}\ x\ y\ t_1)\ (\texttt{swapstr}\ x\ y\ s)\ (\texttt{swap}\ x\ y\ t_2) = \texttt{swap}\ x\ y\ (\mathit{lam}'\ t_1\ s\ t_2)$$

The final stage is getting the `swap` $x\ y$ to “evaporate”. The obvious property to appeal to is

$$x \notin \texttt{FV}(t) \wedge y \notin \texttt{FV}(t) \Rightarrow \texttt{swap}\ x\ y\ t = t \tag{9}$$

Note the abuse of notation in this discussion of strategy: the two `swap` functions in (8) have different types. On the left, `swap` swaps strings in a λ-calculus term; on the right, `swap` is swapping strings in the result type. There are also two different `swap`s in the formula stating the desired commutativity of lam'. Finally, the `swap` in (9) is also over the result type. The final theorem has a side-condition requiring that the result type has `swap` and `FV` functions that behave appropriately. This notion of appropriateness is encoded in the `swapping` predicate, which specifies the properties that a permutation action and an accompanying free-variable function must satisfy:

$$\begin{array}{l}\texttt{swapping}\ \mathit{sw}\ \mathit{fv}\ \equiv \\ \quad (\forall x\,z.\ \ \mathit{sw}\ x\ x\ z = z)\ \wedge \\ \quad (\forall x\,y\,z.\ \ \mathit{sw}\ x\ y\ (\mathit{sw}\ x\ y\ z) = z)\ \wedge \\ \quad (\forall x\,y\,z.\ \ x \notin \mathit{fv}(z) \wedge y \notin \mathit{fv}(z) \Rightarrow \mathit{sw}\ x\ y\ z = z)\ \wedge \\ \quad (\forall x\,y\,z\,s.\ \ s \in \mathit{fv}(\mathit{sw}\ x\ y\ z) \equiv (\texttt{swapstr}\ x\ y\ s) \in \mathit{fv}(z))\end{array} \tag{10}$$

The final recursion principle is presented in Figure 1. The rest of this section explains some of its details, and comments on its proof.

5.1 Parameters

In the presence of additional parameters, satisfying (8) becomes more difficult. This is clear with the example function `sub′` (and normal substitution as well). If *hom* is taken to be `sub′ M v`, then (8) is not true. The action of the permutations must be allowed to affect the parameters. In the case of `sub′`, the appropriate theorem is actually

$$\begin{array}{l}x \neq \texttt{"f"} \wedge y \neq \texttt{"f"} \quad \Rightarrow \\ \quad \texttt{swap}\ x\ y\ (\texttt{sub}'\ M\ v\ t) = \texttt{sub}'\ (\texttt{swap}\ x\ y\ M)\ (\texttt{swapstr}\ x\ y\ v)\ (\texttt{swap}\ x\ y\ t)\end{array}$$

```
swapping rswap rFV ∧ swapping pswap pFV ∧
FINITE X ∧ (∀p. FINITE (pFV p)) ∧

(∀k p. rFV (con k p) ⊆ X ∪ pFV p) ∧
(∀s p. rFV (var s p) ⊆ {s} ∪ pFV p ∪ X) ∧
(∀t′ u′ t u p.
    (∀p. rFV (t′ p) ⊆ FV t ∪ pFV p ∪ X) ∧
    (∀p. rFV (u′ p) ⊆ FV u ∪ pFV p ∪ X) ⇒
    rFV (app t′ u′ t u p) ⊆ FV (APP t u) ∪ pFV p ∪ X) ∧
(∀t′ v t p.
    (∀p. rFV (t′ p) ⊆ FV t ∪ pFV p ∪ X) ⇒
    rFV (lam t′ v t p) ⊆ FV (LAM v t) ∪ pFV p ∪ X) ∧

(∀k x y p.
    x ∉ X ∧ y ∉ X ⇒
    (rswap x y (con k p) = con k (pswap x y p))) ∧
(∀s x y p.
    x ∉ X ∧ y ∉ X ⇒
    (rswap x y (var s p) = var (swapstr x y s) (pswap x y p))) ∧
(∀t t′ u u′ x y p.
    x ∉ X ∧ y ∉ X ⇒
    (rswap x y (app t′ u′ t u p) =
     app (swapfn pswap rswap x y t′) (swapfn pswap rswap x y u′)
         (swap x y t) (swap x y u) (pswap x y p))) ∧
(∀t′ t x y v p.
    x ∉ X ∧ y ∉ X ⇒
    (rswap x y (lam t′ v t p) =
     lam (swapfn pswap rswap x y t′) (swapstr x y v) (swap x y t)
          (pswap x y p))) ⇒

∃hom.
   (∀k p.     hom (CON k)     p = con k p) ∧
   (∀s p.     hom (VAR s)     p = var s p) ∧
   (∀t u p. hom (APP t u)   p = app (hom t) (hom u) t u p) ∧
   (∀v t p.
       v ∉ X ∪ pFV p ⇒
             (hom (LAM v t) p = lam (hom t) v t p)) ∧

   (∀t p x y.
       x ∉ X ∧ y ∉ X ⇒
       (hom (swap x y t) p = rswap x y (hom t (pswap x y p)))) ∧
   (∀t p. rFV (hom t p) ⊆ FV t ∪ pFV p ∪ X)
```

Fig. 1. The recursion principle for λ-calculus terms. The second block of antecedents requires that the function not create too many fresh names. The third block requires that the function respect permutation. The second block of properties in the conclusion state that these properties *do* hold for the resulting `hom` function. For the definition of `swapping`, see (10).

In general, not only does the result type of the desired function need `swap` and `FV` functions, but so too do any parameters. The final recursion principle explicitly acknowledges one unspecified parameter type, and both the final *hom* function, as well as the *con*, *var*, *app* and *lam* values all now take an additional parameter.[4]

It is easy to specify a permutation action for a function type, if one has permutation actions for its domain and range types. This is done below in the definition of `swapfn`. Given this, one might wonder why the final recursion principle needs its explicit treatment of parameters: is it possible instead to simply require that the range of the new function support a permutation action, and expect the use of `swapfn` to specify this when there are extra parameters? Unfortunately, this is *not* possible: the problem does not arise in the requirement that the new function respect permutations, but rather in the requirement that it not generate too many fresh names (see the second block of antecedents in Figure 1).

Consider defining the substitution function, where the free names of the additional parameters may appear in the result. Without using the parameter information, it is impossible to provide a free variable function (`rFV` in Figure 1) for the result-type (a function-space) that will satisfy the new principle's antecedents. Such an `rFV` must simultaneously return small enough sets of names to satisfy the second block of antecedents, and also have an accompanying permutation action, `rswap`. This permutation action must satisfy the requirements embodied in `swapping` and the third block of antecedents. For example, the "null" instantiation, taking `rFV` to always return the empty set, in turn requires `rswap` to be the identity function (because of the third conjunct of `swapping`'s definition (10)), and thus fails to satisfy

$$\texttt{rswap}\ x\ y\ (\mathit{var}\ s) = \mathit{var}\ (\texttt{swapstr}\ x\ y\ s)$$

where

$$\mathit{var} = \lambda s\ v\ M.\ \ \mathsf{if}\ s = v\ \mathsf{then}\ M\ \mathsf{else}\ (\texttt{VAR}\ s)$$

5.2 Proving the theorem

To begin the derivation of the final recursion principle, it is necessary to return to (1) and instantiate *lam* with

$$\lambda f\ g\ p.\ \ \mathsf{let}\ z = \texttt{NEW}(\texttt{FV}(\texttt{ABS}(g)) \cup \texttt{pFV}(p) \cup X)\ \ \mathsf{in}\ \ \mathit{lam}'\ (f\ z)\ z\ (g\ z)\ p$$

As before, $\texttt{ABS}(g)$ is equal to the original term, so that z is now fresh with respect to it as well as the parameter. The set of strings to avoid for p's sake is given by the `pFV` function. The finite set X is used to avoid those free names that are somehow implicit in the function itself. Such a name is the `"f"` present in the definition of the `sub′` example.

[4] One parameter is sufficient: additional curried parameters can be dealt with by first showing the existence of an isomorphic uncurried, or tupled, version of the function. The no parameter case is obtained by letting the parameter type be the singleton type `one`, also known as `unit`.

When the substitutions of (1) are replaced with permutations, the LAM-clause becomes

$$\begin{array}{l} \forall v\, t\, p.\ \mathit{hom}\ (\mathtt{LAM}\ v\ t)\ p\ = \\ \qquad \textsf{let}\ z = \mathtt{NEW}(\mathtt{FV}(\mathtt{LAM}\ v\ t) \cup \mathtt{pFV}(p) \cup X)\ \textsf{in} \\ \qquad\quad \mathit{lam}'\ (\mathit{hom}(\mathtt{swap}\ z\ v\ t))\ (\mathtt{swapstr}\ z\ v\ v)\ (\mathtt{swap}\ z\ v\ t)\ p \end{array}$$

The strategy sketched at the beginning of this section is still the right way to proceed, even after the complication of parameters has been introduced. Its first stage is to move permutations upwards in the above clause, appealing to commutativity results. The third block of antecedents in the final principle allow this to occur. This block also features the use of swapfn, which defines permutation on a function space, given permutation actions for the domain and range type. Its definition is

$$\mathtt{swapfn}\ \mathit{dsw}\ \mathit{rsw}\ x\ y\ f = \lambda z.\ \mathit{rsw}\ x\ y\ (f\ (\mathit{dsw}\ x\ y\ z))$$

Use of swapfn is required because the result type of *hom* is a function space, so that in expressions such as $\mathit{app}\ (\mathit{hom}(t))\ (\mathit{hom}(u))\ t\ u\ p$, the first two arguments to *app* are functions.

The final part of the strategy is to appeal to (9). The strategy is to have the swap terms in

$$\begin{array}{l} \textsf{let}\ z = \mathtt{NEW}(\mathtt{FV}(\mathtt{LAM}\ v\ t) \cup \mathtt{pFV}(p) \cup X)\ \textsf{in} \\ \quad \mathtt{rswap}\ z\ v\ (\mathit{lam}'\ (\mathit{hom}(t))\ v\ t\ (\mathtt{pswap}\ z\ v\ p)) \end{array}$$

"evaporate". The variable v is the original bound variable of the abstraction, and the condition on the equation being derived requires it to not be present in the free variables of p. Variable z shares this property by construction, so $(\mathtt{pswap}\ z\ v\ p)$ can be replaced by p. The $\mathtt{rswap}\ z\ v$ term can only be eliminated if the side-conditions in the theorem ensure that *hom*, and thus all of the helper functions, do not generate too many new names in the result. This is guaranteed by the second block of antecedents in the recursion principle.

The proof consists of showing that the *hom* known to exist from the original principle has the properties specified in the final recursion principle. It begins by showing that the new function doesn't produce too many free variables, i.e., that the theorem's conclusion's very last conjunct holds. This proof is by induction, using the original Gordon-Melham induction principle. Next, the commutativity result is shown (the second to last conjunct). This is done by an induction on the size of the term. Finally, both of these results are used according to the strategy described above, to prove the nice form of the LAM-clause.

The course of the proof of the final recursion principle also reveals exactly what properties are needed of the swap and FV functions in the theorem's two other types (parameter and range). These properties are defined by the predicate swapping (10) that appears in the final recursion principle.

6 Application and implementation

The final recursion principle allows the definition of all of the functions given in Section 2. Further, I have implemented a tool to automatically attempt those definitions where there is just one parameter. This means that definitions for all the examples except `v_posns` and `sub'` can be made entirely automatically. The tool does not cope with parameterised definitions because I have yet to implement the (rather uninteresting) logic that would translate something like $f\ x\ t\ z$ into $f\ t\ (x,z)$, and back again, as required. Currently, my code also always instantiates the X parameter with the empty set.

The implemented code includes a rudimentary database of types, which is used to provide appropriate permutation and swapping information about result types. The function definition tool can therefore instantiate all of the recursion principle without user intervention. For the simple examples, such as `size`, and even `enf`, the result type is one that doesn't support a swapping action. It is easy to see that the null-swap, $(\lambda x\ y\ z.\ z)$ and the everywhere-empty free variable function $(\lambda x.\ \emptyset)$ satisfy the requirements of `swapping` in (10). Such an instantiation also leads to immediate simplifications in the final recursion principle itself. For example, the second block of antecedents completely disappears.

After instantiation, the tool must try to discharge the side conditions. Clearly, arbitrary definition attempts might produce side-conditions that no automatic tool could be expected to discharge. At the moment, however, all of the one-parameter cases (those given above, and also all those that arose in my formalisation of the standardisation theorem) are handled by the tool with little more than a call to the standard simplifier, appropriately augmented with relevant rewrite theorems.

While the code can not yet cope with functions of more than one parameter, it is not difficult to instantiate the final recursion principle by hand. For `sub'`, the instantiation of the helper functions is as follows (where I have arbitrarily decided that the parameter type pairs the string and the term in that order):

$$\begin{array}{lll}
\mathit{var} & \mapsto & \lambda s\ (v,M).\ \textsf{if}\ s = v\ \textsf{then}\ M\ \textsf{else}\ \texttt{VAR}(s) \\
\mathit{con} & \mapsto & \lambda k\ p.\ \texttt{CON}(k) \\
\mathit{app} & \mapsto & \lambda rt\ ru\ t\ u\ p.\ \texttt{APP}\ (rt\ p)\ (ru\ p) \\
\mathit{lam} & \mapsto & \lambda rt\ u\ t\ (v,M).\ \texttt{LAM}\ u\ (rt\ (v,\ \texttt{APP}\ (\texttt{VAR}(\texttt{"f"}))\ M))
\end{array}$$

These instantiations require no creative thought to calculate, and it is clear that an automatic tool to do this work would be straightforward to implement.

Similarly, the existing database, mapping types to likely swapping and free variable functions, makes it clear what the instantiations for the following variables should be:

$$\begin{array}{lll}
\texttt{rswap} & \mapsto & \texttt{swap} \\
\texttt{rFV} & \mapsto & \texttt{FV} \\
\texttt{pswap} & \mapsto & \lambda x\ y.\ (\texttt{swapstr}\ x\ y \times \texttt{swap}\ x\ y) \\
\texttt{pFV} & \mapsto & \lambda(v,M).\ \{v\} \cup \texttt{FV}(M)
\end{array}$$

Finally, `sub'` requires X to be $\{\texttt{"f"}\}$.[5]

The instantiation for `sub'` above creates quite a complicated instance of the final recursion principle. Nonetheless, the derived side-conditions are easy to eliminate.

[5] A general rule for the calculation of X might be to include in X any names mentioned explicitly in a proposed definition. This question probably doesn't warrant much investigation, as functions like `sub'`, with their own free names, seem unlikely to arise in practice.

The definition of functions such as `rator`, where values for whole classes of argument are left unspecified, brings up one last wrinkle. The database storing information about each type should record a value in each type that has no free names (if possible: the type `string` has no such value). This value can then be provided as the result value for the omitted constructors. If this can't be done then the X parameter will need to be instantiated to cover the extra free names present in whatever value was chosen to be the value in the unspecified cases. This is something to avoid if possible, because it results in the commutativity result in the conclusion of the recursion principle retaining its annoying side-conditions.

7 Related work

There are three pieces of work closely similar to the topic of this paper. All explicitly concern themselves with the specification of a recursion (or "iteration") principle for types with binders and α-equivalence, and all three apply the developed theory in a mechanised setting. Two are the inspiration for my own work: Gordon and Melham [5], and the Gabbay-Pitts theory of Fraenkel-Mostowski sets, particularly §10.3 of Gabbay's PhD thesis [3]. The third is work by Ambler, Crole and Momigliano in [1].

Clearly, this work would have been impossible without the underlying Gordon-Melham characterisation of λ-calculus terms up to α-equivalence. My claim is that, complicated side-conditions notwithstanding, the final recursion principle in Figure 1 is an improvement on the original Gordon-Melham principle (1). This is because the new principle has a conclusion that allows functions to be defined in a way that much more closely approximates familiar and traditional principles of primitive recursion.

Inasmuch as the new principle embodies restrictions imported from the theory of FM-sets, it can not define all of those functions definable with the original principle. For example, neither principle will support the definition of a function with clause

$$f\ (\mathtt{LAM}\ v\ t) = t$$

because this is unsound. But it is not difficult to use (1) to define a function with clause

$$f\ (\mathtt{LAM}\ v\ t) = t[v \mapsto \mathtt{NEW}(\mathtt{FV}(\mathtt{LAM}\ v\ t))]$$

This returns the body of the abstraction with an arbitrary, fresh name substituted through for the bound variable. Appealing as it does to the Axiom of Choice, in a way that would allow the enumeration of all names, this function is impossible to define in the Fraenkel-Mostowski theory, and also impossible to define using Figure 1's recursion principle.

My work has been greatly inspired by the theory of permutations developed by Gabbay and Pitts. It might be characterised as an attempt to bring the nice features of this FM-theory into the world of classical higher-order logic. In this "HOL world", one need not give up the Axiom of Choice. Nor need one assert that the set of names is infinite, but that its subsets are all either finite or have finite complements. Instead those axioms of the FM-theory that are absolutely necessary for function definition in the primitive recursive style are imported as side-conditions. These side-conditions are easily discharged for definitions that are well-behaved; seemingly the vast majority. For those definitions that are not so well behaved, the HOL resident can always resort to (1); in the world of FM-sets, these definitions remain inadmissible. The only significant loss in the classical setting would appear to be the И quantifier, or at least its nice properties, such as $(\text{И}x.\neg P) \equiv \neg(\text{И}x.P)$.

Another possible advantage of the approach described here is that the user is able to choose the instantiations for `pswap` and `rswap` on a case-by-case basis. If, for example, a function used strings in a way unconnected with their role as names, one wouldn't provide `swapstr` as the permutation function, but rather the null swapping function $(\lambda x\ y\ z.\ z)$. This freedom may or may not be significant in practice. A related idea, though one that also loses this flexibility, might be to use Isabelle/HOL's axiomatic type classes to automatically associate types with appropriate permutation and free-name functions, thereby allowing the `swapping` side conditions in the recursion principle to disappear.

Finally, recent work by Ambler, Crole and Momigliano [1] presents a recursion principle for a (weak) higher-order abstract syntax view of the untyped λ-calculus (in classical Isabelle/HOL). This work gets around some of the typical problems associated with higher-order abstract syntax by working with terms-in-infinite-contexts, thereby providing a type of terms Λ where the function space $(var \to \Lambda)$ is isomorphic to the original type Λ. This is achieved by making Λ itself a function space: $(var^\infty \to \Lambda_0)$, for an underlying algebraic type Λ_0. Ambler *et al.* then define an inductive relation prop that isolates the "proper" or non-exotic values of Λ. In order to retain the use of meta-level functions in the object syntax, the proper terms are not used as the basis for the definition of a new type. So, while Λ still includes exotic terms, prop allows them to be identified.

The recursion combinator is a perfect instance of primitive recursion in its behaviour under the abstraction binder, but the extra infinite context parameters add complexity. When passing under a binder in the definition of substitution, for example, variable indices need to be incremented in both the term being substituted and the body being substituted into. This is rather reminiscent of the de Bruijn implementation of substitution.

The work by Ambler *et al.* is the first to prove a recursion principle for function definition over (weak) higher-order abstract syntax. Their paper provides pointers to a number of other HOAS approaches to the problem. Work by Schürmann, Despeyroux, and Pfenning [8], and by Washburn and Weirich [9], exemplifies one such approach. In this work, sophisticated type systems (modal λ-calculus, and first-class parametric polymorphism respectively) prevent the untrammeled use of function-spaces, thereby avoiding exotic terms while allowing iteration over these structures. Meta-theoretic reasoning about such embedded systems (e.g., proving results akin to the standardisation theorem for the untyped λ-calculus) remains a challenging area for future research.

8 Conclusion

I have presented a new recursion principle for the type of λ-terms that allows the ready definition of recursive functions over these terms. It has been proved in HOL, and motivates a definitional technique that looks as much as possible like primitive recursion. The validity of recursions that pass under binders is ensured by appeal to side-conditions that embody restrictions based on the ideas of Gabbay's and Pitts's Fraenkel-Mostowski set theory.

I have further implemented a small HOL library that allows users to write definitions in the obvious "pattern-matching" style, and which automatically discharges the FM-related side conditions. This is done with the help of a small database mapping types to information about how they support permutation and the notion of free names.

The theorem and the library have been tested on the definitions made in the course of an earlier project mechanising a substantial piece of λ-calculus theory. A sample of representative functions (all of which the theorem handles) is presented in Section 2 above.

Versions of the recursion principle for other types with binders are easy to state: in the antecedents, they simply require that all the functions standing in for the constructors of the new type (the equivalents of *var*, *con*, *app* and *lam* in Figure 1) not generate too many fresh names, and that they and permutations commute. In the conclusion of these theorems, the equations for constructors that are binders acquire side conditions stating that the recursive characterisation is invalid for finitely many choices of bound variable name. Automating the *proof* of such theorems is the key task in being able to define new types with binders automatically.

Future work In the short term, I hope to soon extend the implementation of the library to support the definition of functions with more than one parameter. This is not conceptually difficult: the work required is simply that of implementing transformations such as moving from tupled to curried arguments, and switching parameter orders.

A recursion principle that supported the definition of well-founded functions would also be very useful. This would allow definitions that recursed at arbitrary depths under binders. HOL's existing implementation of definition by well-founded recursion requires that constructors be injective, something not true of binders.

A more significant project is the development of theory and code to support the establishment of new types with binders. It is not difficult to establish new types by hand, and it is also clear what the recursion principle for new types should be. The challenge will be establishing types automatically, including the proof of their recursion principles.

Availability All of the theory and code described in this paper will be available as part of the next distribution of the HOL system.

References

1. Simon J. Ambler, Roy L. Crole, and Alberto Momigliano. A definitional approach to primitive recursion over higher order abstract syntax. In Honsell et al. [6]. Available at http://doi.acm.org/10.1145/976571.976572.
2. H. P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, Amsterdam, revised edition, 1984.
3. M. J. Gabbay. *A Theory of Inductive Definitions with Alpha-Equivalence*. PhD thesis, University of Cambridge, 2001.

4. M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *14th Annual Symposium on Logic in Computer Science*, pages 214–224. IEEE Computer Society Press, Washington, 1999.
5. A. D. Gordon and T. Melham. Five axioms of alpha conversion. In J. von Wright, J. Grundy, and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 9th International Conference, TPHOLs'96*, volume 1125 of *Lecture Notes in Computer Science*, pages 173–190. Springer-Verlag, 1996.
6. Furio Honsell, Marino Miculan, and Alberto Momigliano, editors. *Merlin 2003, Proceedings of the Second ACM SIGPLAN Workshop on Mechanized Reasoning about Languages with Variable Binding*. ACM Digital Library, 2003.
7. Michael Norrish. Mechanising Hankin and Barendregt using the Gordon-Melham axioms. In Honsell et al. [6]. Available at http://doi.acm.org/10.1145/976571.976577.
8. Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266(1–2):1–57, September 2001.
9. Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymophism. In *ICFP'03: Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, pages 249–262. ACM Press, 2003.