A Combinator and Presheaf Topos Model for Primitive Recursion over Higher Order Abstract Syntax [⋆]

S. J. Ambler(S.Ambler@mcs.le.ac.uk)
R. L. Crole(R.Crole@mcs.le.ac.uk) &
A. Momigliano (A.Momigliano@mcs.le.ac.uk)

Department of Computer Science, University of Leicester,
Leicester, LE1 7RH, U.K.

Abstract. We describe a theory Bsyntax, which we have implemented in Isabelle HOL, and prove the existence of a combinator for primitive recursion over HOAS. The definition of the combinator is facilitated by the use of terms with *infinite* contexts. An immediate payoff is that we obtain higher order simultaneous substitution as a function for free. We have defined a presheaf model of Bsyntax, providing additional semantic validation of Bsyntax's principles of recursion. We mention an application of our work to mechanized reasoning about the compiler intermediate language MIL-lite.
Keywords: initial algebras; higher order abstract syntax; Isabelle HOL; λ-calculus; primitive recursion; topos theory.

1 Introduction

Higher order abstract syntax (HOAS) has been the subject of considerable research effort over the last few years. The fundamental idea dates back to Church [3]. HOAS is well established for *encoding* the syntax of object logics in a theorem prover. However, we also want to *reason about* the syntax of an object logic, and this is likely to involve the use of principles of induction and coinduction.

It is known that problems arise when combining HOAS with principles of induction and coinduction (see for example [5, 10, 1]). One answer is the system of *hybrid syntax* introduced in [1]. This gives a type of Isabelle/HOL expressions which support several schemes of induction over HOAS. Another problem concerns how to define recursive functions over the terms of HOAS. The issue is how to define recursive calls over λ-binders (see [10, 9]). In fact a method for defining recursive functions over the terms of HOAS forms the central topic of this paper.

Why would we want to define recursive functions over the terms of HOAS? If we wish to reason about object logics, then we will probably want to employ definitions by primitive recursion. In particular, to encode the operational semantics of an object level programming language we may require capture avoiding substitution, which can be defined by primitive recursion (with parameters). Similarly we might want to determine the occurrence of free variables within an expression, or the size of an expression. Each of these functions can be presented by defining its graph as

[⋆] This work was supported by EPSRC grant number GR/M98555.

a relation and proving it to be total and functional. A better approach, outlined in this paper, is to *define them uniformly via a combinator for primitive recursion.*

The main contributions of our work are to code in Isabelle HOL

- a presentation of weak HOAS using a λ-calculus of *terms with infinite contexts* which supports a strong principle of structural induction;
- a proof of the existence of a *combinator for primitive recursion over HOAS*;
- a representation of a *substantial object logic*, namely the compiler intermediate language MIL-lite [2], together with machine proofs of properties of the system;

and to present a mathematical account of

- a *presheaf topos model*, from which we obtain further semantic validation of recursion principles by exhibiting the types over which recursion takes place as initial algebras.

We refer to our Isabelle HOL Theories for HOAS by the name `Bsyntax` (binding syntax). In Section 2 we introduce a datatype for HOAS and show how to identify a subtype of λ-*calculus terms-in-context.* In Section 3 we motivate and introduce a combinator for primitive recursion. In Section 4 we show how to capture the semantics of a very simple (object) programming language within `Bsyntax`, and remark that the ideas scale up to the MIL-lite language of Benton and Kennedy. In Section 5 we present a presheaf topos model of our version of HOAS, and indicate that our combinator captures the universal property of categorical primitive recursive definitions.

2 An Encoding of λ-calculus Terms-in-Infinite-Contexts

The datatype $exp \quad ::= \quad \mathsf{V}\ var \mid \mathsf{L}\ (var \Rightarrow exp) \mid exp\ \mathsf{A}\ exp$ is used to encode (weak) HOAS. Using this datatype, we shall produce, by recursion, terms which correspond to those of the λ-calculus, thus providing an adequate representation. To achieve this we work with *terms-in-context* [4, 6]. The reason for this is that a recursive call over a functional expression of type $var \Rightarrow exp$ will require a fresh call over the body of the term, and the body will (possibly) contain a new free variable. A context is required to track the creation of these free variables. Implementing finite contexts, with all of the associated book-keeping operations such as weakening and strengthening, can be awkward, so in fact we work with *terms-in-infinite-contexts.*

In particular, a context will be a *stream* of variables, realized as a term of type $var\ stream \overset{\text{def}}{=} nat \Rightarrow var$. One reason for working with infinite contexts is that some of the book-keeping tasks can be implemented neatly by making use of the properties of functions defined on $var\ stream$. In particular, we make use of the functions which compute the nth element of, the tail of, and head of a list l. They are denoted by $(l\ !\ n)$, $\mathsf{tl}\ l$ and $\mathsf{hd}\ l$, and u consed onto l is denoted by $u \# l$. Note that these functions are total on streams: over finite lists, tail is not total, which can complicate matters in a theorem prover such as Isabelle HOL where one must constantly check to ensure that operations are not performed on empty lists.

Terms-in-infinite-contexts are elements of a type $eic \overset{\text{def}}{=} var\ stream \Rightarrow exp$. A typical term-in-context (in η-long form) looks like $\lambda\ l.\ e\ l$. The bound variable l

$$\mathsf{prop}\ (\lambda\ l.\ \mathsf{V}\ (l\ !\ n)) \qquad \frac{\mathsf{prop}\ e_1 \qquad \mathsf{prop}\ e_2}{\mathsf{prop}\ (\lambda\ l.\ (e_1\ l)\ \mathsf{A}\ (e_2\ l))} \qquad \frac{\mathsf{prop}\ (\lambda\ l.\ e\ (\mathsf{hd}\ l)\ (\mathsf{tl}\ l))}{\mathsf{prop}\ (\lambda\ l.\ \mathsf{L}\ u.\ e\ u\ l)}$$

Table 1. Definition of prop Terms-in-Infinite-Contexts (in Isabelle HOL)

provides an enumerable supply of terms $(l\ !\ i)$ of type *var* and these can be used within the scope of binding to stand for the variables of the context. Note that $eic \cong var \Rightarrow eic$ via the standard stream isomorphism. This means that the fresh free variable alluded to above can be "absorbed" into the infinite context. We obtain a neat definition of the predicate $\mathsf{prop} :: eic \Rightarrow bool$ identifying proper terms of type *eic*, given in Table 1. These proper terms provide an *adequate* representation of the λ-calculus and hence the basis for the HOAS encoding of an object logic.

We give some examples, and general comments. Consider

$$\lambda\ l.\ \mathsf{L}\ u.\ (\mathsf{V}\ u)\ \mathsf{A}\ (\mathsf{V}\ (l\ !\ 4))\ \mathsf{A}\ (\mathsf{V}\ (l\ !\ 8))$$

Roughly speaking, this is an encoding of a λ-calculus term $\Lambda U.U\ V_4\ V_8$. One has to take care in understanding the meaning of (for example) $\lambda\ l.\ \mathsf{V}\ (l\ !\ 4)$. Recall that $l :: var\ stream$. So $\lambda\ l.\ \mathsf{V}\ (l\ !\ 4)$ is the fourth "actual" variable in a fixed enumeration. In fact we will think of it as the *fourth projection* of an arbitrary infinite sequence of variables. Note also that we claim that the binder λl gives rise to a notion of context, and that traditionally contexts consist of *distinct* variables. This is indeed the case here, as we can prove that $\lambda\ l.\ \mathsf{V}\ (l\ !\ n) = \lambda\ l.\ \mathsf{V}\ (l\ !\ m)$ just in case $m = n$. We refer to $\lambda\ l.\ \mathsf{V}\ (l\ !\ n)$ as the *nth (variable) projection*. Moving to the definition of proper abstractions, consider for example

$$\frac{\mathsf{prop}\ (\quad \lambda\ l.\ (\mathsf{V}\ (\mathsf{hd}\ l))\ \mathsf{A}\ (\mathsf{V}\ (l\ !\ 5))\ \mathsf{A}\ (\mathsf{V}\ (l\ !\ 9)) \quad)}{\mathsf{prop}\ (\quad \lambda\ l.\ \mathsf{L}\ u.\ (\mathsf{V}\ u)\ \mathsf{A}\ (\mathsf{V}\ (l\ !\ 4))\ \mathsf{A}\ (\mathsf{V}\ (l\ !\ 8)) \quad)}$$

Notice that in our system, when variables are bound by the L binder, binding is forced to occur over the 0th projection $(l\ !\ 0) = \mathsf{hd}\ l$. Note also that the effect of replacing $\mathsf{tl}\ l$ by l is to decrease all other projection indices by 1 when an abstraction is formed. This is a key point, and will be fundamental to achieving a definition of a recursion combinator. To help understand the formulation of the λl binder, note that the types $var \Rightarrow eic$ and eic are *isomorphic*, with $\lambda\ u.\ \lambda\ l.\ e\ u\ l :: var \Rightarrow eic$ corresponding to $\lambda\ l.\ e\ (\mathsf{hd}\ l)\ (\mathsf{tl}\ l) :: eic$. This isomorphism holds because of the definitional property of a stream of variables. Thus, roughly speaking, properness of $\lambda\ l.\ \mathsf{L}\ u.\ e\ u\ l$ occurs just in case properness of $\lambda\ l.\ e\ (\mathsf{hd}\ l)\ (\mathsf{tl}\ l)$ occurs—hence the definition in Table 1. Many results about the `Bsyntax` system have been proved in Isabelle HOL. In such cases we indicate this as follows

Theorem 1 (Isabelle HOL). *The definition in Table 1 specifies a monotone operator yielding a well-defined inductive definition.*

3 A Combinator for Primitive Recursion

Our next contribution is the construction of a combinator synr for primitive recursion over terms of type *eic*, and hence over HOAS. The type of our Isabelle HOL combinator synr is

$$(nat \Rightarrow B) \Rightarrow (B \Rightarrow B) \Rightarrow (B \Rightarrow B \Rightarrow B) \Rightarrow eic \Rightarrow B$$

We give the graph of our combinator as an inductive definition in higher order logic, and a typical instance of its use looks like synr vf lf af e r. The idea is that one examines the (inductive) structure of e, to see if it is a (proper) variable, application, or abstraction term of type *eic*. Depending on which case is in play, r is the result of recursively calling, respectively, one of the functions vf :: $nat \Rightarrow B$, af :: $B \Rightarrow B$ or lf :: $B \Rightarrow B \Rightarrow B$. The graph is formally defined by

$$\mathsf{synr\ vf\ lf\ af}\ (\lambda\, l.\ \mathsf{V}\ (l\ !\ n))\ (\mathsf{vf}\ n)$$

$$\frac{\mathsf{synr\ vf\ lf\ af}\ (\lambda\, l.\ e\ (\mathsf{hd}\ l)\ (\mathsf{tl}\ l))\ x}{\mathsf{synr\ vf\ lf\ af}\ (\lambda\, l.\ \mathsf{L}\ u.\ e\ u\ l)\ (\mathsf{lf}\ x)} \qquad \frac{\mathsf{synr\ vf\ lf\ af}\ e_1\ y \qquad \mathsf{synr\ vf\ lf\ af}\ e_2\ z}{\mathsf{synr\ vf\ lf\ af}\ (\lambda\, l.\ (e_1\ l)\ \mathsf{A}\ (e_2\ l))\ (\mathsf{af}\ y\ z)}$$

Theorem 2 (Isabelle HOL). *The relation* synr *specified above can be shown to be a Isabelle HOL function, with the required properties of a combinator for primitive recursion.*

We finish this section by giving a very simple example showing how synr works in practice; and we discuss higher order substitution. Take the type B to be *nat*. Define vf $\stackrel{\text{def}}{=} \lambda\, n.\ 1$ and lf $\stackrel{\text{def}}{=} \lambda\, n.\ n$ and af $\stackrel{\text{def}}{=} \lambda\, n.\ \lambda\, m.\ n+m$. Then synr vf lf af will compute the number of *non-binding* occurrences of all variables in a term; informally, this is the number of occurrences of V . The function vf counts each such variable once; lf does not alter the count, as traversing a binder should simply ignore the binding occurrence; and af adds the number of such variables from the two sub-terms of an application. For example, the number N of such variables in $\lambda\, l.\ \mathsf{L}\ u.\ (\mathsf{V}\ u)\ \mathsf{A}\ (\mathsf{V}\ (l\ !\ 3))$ is 2, and is given by

$$\begin{aligned}
N &= \mathsf{synr\ vf\ lf\ af}\ (\lambda\, l.\ \mathsf{L}\ u.\ (\mathsf{V}\ u)\ \mathsf{A}\ (\mathsf{V}\ (l\ !\ 3)))\\
&= \mathsf{lf}\ (\mathsf{synr\ vf\ lf\ af}\ ((\mathsf{V}\ (l\ !\ 0))\ \mathsf{A}\ (\mathsf{V}\ (l\ !\ 4))))\\
&= \mathsf{lf}\ (\mathsf{af}\ (\mathsf{synr\ vf\ lf\ af}\ (\lambda\, l.\ \mathsf{V}\ (l\ !\ 0)))\ (\mathsf{synr\ vf\ lf\ af}\ (\lambda\, l.\ \mathsf{V}\ (l\ !\ 4))))\\
&= \mathsf{lf}\ (\mathsf{af}\ (\mathsf{vf}\ 0)\ (\mathsf{vf}\ 4)) = (\lambda\, n.\ n)\ ((\lambda\, n.\ \lambda\, m.\ n+m)\ 1\ 1) = 2
\end{aligned}$$

The recursion combinator can also provide a uniform method for defining functions such as capture-avoiding substitution by primitive recursion on meta-level syntax. Indeed, in order to use `Bsyntax` to represent operational semantics, we must be able to represent substitution. We can define "standard" substitution via the recursion combinator, as a recursive function over *eic*, with the expected type

$$\mathsf{synr\ vf\ af\ lf} :: eic \Rightarrow var \Rightarrow eic \Rightarrow eic$$

for suitable values of vf, lf, and af. However, in order to make proper use of HOAS, we want to be able to define higher order substitution—the recursion operator achieves this in a neat and systematic way. In Table 3 (see Section 4) the higher order substitution function hosub has the expected type $(var \Rightarrow eic) \Rightarrow eic \Rightarrow eic$, and is recursive over $var \Rightarrow eic$.

$$\Gamma \vdash \lambda\, l.\, \mathsf{V}\ (l\ !\ n) :: (\Gamma\ !\ n) \qquad \Gamma \vdash \lambda\, l.\, \mathsf{Int}\ z :: \mathit{int} \qquad \frac{\Gamma \vdash \lambda\, l.\, e\ l :: \tau}{\Gamma \vdash \lambda\, l.\, \mathsf{Val}\ (e\ l) :: CT\ \tau}$$

$$\frac{\Gamma \vdash \lambda\, l.\, e_1\ l :: \mathit{int} \quad \Gamma \vdash \lambda\, l.\, e_2\ l :: \mathit{int}}{\Gamma \vdash \lambda\, l.\, (e_1\ l) + (e_2\ l) :: \mathit{int}}$$

$$\frac{\Gamma \vdash \lambda\, l.\, e_1\ l :: CT\ \tau_1 \quad \tau_1\ \#\ \Gamma \vdash \lambda\, l.\, e_2\ (\mathsf{hd}\ l)\ (\mathsf{tl}\ l) :: CT\ \tau_2}{\Gamma \vdash \lambda\, l.\, \mathsf{Let}\ x \Leftarrow e_1\ l\ \mathsf{in}\ e_2\ x\ l :: CT\ \tau_2}$$

Table 2. A Type Assignment Relation

$$\lambda\, l.\, (\mathsf{Int}\ z) + (\mathsf{Int}\ z') \Downarrow \lambda\, l.\, \mathsf{Int}\ z + z'$$

$$\lambda\, l.\, \mathsf{Val}\ (e\ l) \Downarrow \lambda\, l.\, e\ l \qquad \frac{\lambda\, l.\, e_1\ l \Downarrow \lambda\, l.\, v_1\ l \quad \mathsf{hosub}\ e_2\ (\lambda\, l.\, v_1\ l) \Downarrow \lambda\, l.\, v\ l}{\lambda\, l.\, \mathsf{Let}\ x \Leftarrow e_1\ l\ \mathsf{in}\ e_2\ x\ l \Downarrow \lambda\, l.\, v\ l}$$

Table 3. An Evaluation Relation

4 Applications to Object Level Languages

In order to illustrate how our ideas are applied in practice, we define a small (object level) language, showing how we can express its static and dynamic semantics. While the language is elementary, we later show that our methodology can indeed be successfully applied to a much more complex language. The types are given by integers and computation types $CT\ \tau$ [7]. The terms of the language are given by

$$\mathsf{Int}\ z \stackrel{\text{def}}{=} \mathsf{C}\ (\mathsf{stringof}\ \mathsf{z}) \qquad \mathsf{Val}\ e \stackrel{\text{def}}{=} (\mathsf{C}\ \mathtt{Val})\ \mathsf{A}\ e$$
$$e_1 + e_2 \stackrel{\text{def}}{=} (\mathsf{C}\ \mathtt{Add})\ \mathsf{A}\ e_1\ \mathsf{A}\ e_2 \qquad \mathsf{Let}\ x \Leftarrow e_1\ \mathsf{in}\ e_2\ x \stackrel{\text{def}}{=} (\mathsf{C}\ \mathtt{Let})\ \mathsf{A}\ e_1\ \mathsf{A}\ (\mathsf{L}\ x.\ e_2\ x)$$

Note that in this section we make use of `Bsyntax` constants; these were not discussed in Section 2, but can be added without additional technical complications. Each constructor C has type *string* ⇒ *exp*, and we use strings to give "names" (such as `Add`) to the constants of our object level language; this is the standard approach in logical frameworks [8]. Note that the let terms of a computational monad contain a binder (x above is bound) and this is captured by meta-level (`Bsyntax`) binding.

We define a ternary type assignment relation, with typical relationship $\Gamma \vdash e :: \tau$, and carrier *types stream* ∗ *eic* ∗ *types*. The idea is that object level terms are represented by meta-level terms of type *eic*, and that object level contexts which supply types to (free) object variables are represented by a stream of types. The relation is inductively defined using the rules in Table 2. We also define an evaluation semantics, relating terms of type *eic*, in Table 3—the substitution function hosub was introduced in Section 3.

A specific goal of our work is to investigate the viability of encoding and reasoning about *effect based compiler transformations*. We have focussed on Benton and Kennedy's MIL-lite [2], a typed intermediate language with computation types, which can be used to validate compiler transformations. The type system contains integers, integer references, functions, products, sums, and effect based computations. Moreover, a subtyping relation is induced by effect inclusion. We have encoded MIL-lite in `Bsyntax`, illustrating that our approach is applicable to non-trivial languages.

5 A Presheaf Topos Model

We give a presheaf model of `Bsyntax`. In this section we only outline the key ideas. First, why do we want to consider such a model? We have claimed that we have defined a combinator for defining functions over *eic*, that is functions of type $eic \Rightarrow B$, by primitive recursion. We can make this statement precise by interpreting it categorically. Suppose that *eic* and B can be modelled by objects in a category. Then synr should correspond to a "fold" operator, which maps an algebra morphism $g{:}\, TB \rightarrow B$ (for a certain functor T) to the unique mediating morphism $\mathsf{synr}\ g{:}\ eic \rightarrow B$, where *eic* is an initial algebra for T. This is exactly what is meant by a definition of recursive functions over *eic*.

In slightly more detail, we can in fact define a category with objects *var* and *eic*, possessing an endofunctor $T\ \xi \overset{\text{def}}{=} var + \xi + \xi^2$, and such that *eic* gives rise to an initial algebra $T\ eic \rightarrow eic$. We can also show that there is an initial algebra $T\ (var \Rightarrow eic) \rightarrow (var \Rightarrow eic)$. This semantically validates the higher order functions (including substitution) which are defined by primitive recursion over the type $var \Rightarrow eic$; see the end of Section 3.

$\mathbb{F}_\omega$ is the full subcategory of $\mathcal{S}et$ whose objects are the Peano sets $0, 1, 2, \ldots$ and ω. Our categorical model is in fact $\mathcal{S}et^{\mathbb{F}_\omega}$, a topos of presheaves, and *eic* is, roughly speaking, a functor that maps n to the set of proper terms over n variables. The calculations required to establish the existence of the initial algebras are omitted from this short paper.

6 Summary

This paper provides a very broad outline of a new approach for combining primitive recursion and HOAS in a consistent way, and a brief discussion of categorical models. A considerable volume of examples, theory and applications will appear in a forthcoming journal article. The authors wish to thank the referees of *Computer Science Logic 2003, Vienna, Austria* for comments on the original paper.

References

1. Simon Ambler, Roy Crole, and Alberto Momigliano. Combining higher order abstract syntax with tactical theorem proving and (co)induction. In V. A. Carreño, editor, *Proceedings of the 15th International Conference on Theorem Proving in Higher Order Logics, Hampton, VA, 1-3 August 2002*, volume 2342 of *LNCS*. Springer Verlag, 2002.

2. N. Benton and A. Kennedy. Monads, effects and transformations. *Electronic Notes in Theoretical Computer Science*, 26, 1999.
3. A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
4. Joëlle Despeyroux and André Hirschowitz. Higher-order abstract syntax with induction in Coq. In Frank Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 159–173, Kiev, Ukraine, July 1994. Springer-Verlag LNAI 822.
5. Martin Hofmann. Semantical analysis of higher-order abstract syntax. In *Proc. of 14th Ann. IEEE Symp. on Logic in Computer Science, LICS'99, Trento, Italy, 2–5 July 1999*, pages 204–213. IEEE Computer Society Press, Los Alamitos, CA, 1999.
6. Raymond McDowell and Dale Miller. Reasoning with higher-order abstract syntax in a logical framework. *ACM Transactions on Computational Logic*, 3(1):80–136, January 2002.
7. E. Moggi. Notions of computation and monads. *Theoretical Computer Science*, 93:55–92, 1989.
8. F. Pfenning. Computation and deduction. Lecture notes, 277 pp. Revised 1994, 1996.
9. Carsten Schürmann. Recursion for higher-order encodings. In *Proceedings of Computer Science Logic (CSL 2001)*, volume 2142 of *Lecture Notes in Computer Science*, pages 585–599, 2001.
10. Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266(1–2):1–57, September 2001.