Constraints on recursion in the Hume expression language

Greg Michaelson

Department of Computing and Electrical Engineering
Heriot-Watt University
greg@cee.hw.ac.uk, +44 131 451 3422

Abstract. Hume is a proposed new environment for constructing safety critical systems. Here, possible constraints on recursion in the Hume expression language, to aid termination determinacy, are discussed and structural operational semantics for static syntactic checks for simple and nested recursion are presented.

1 Introduction

Hume is the Higher-order Unified Meta-Environment, a language and tool set intended for the construction of safety critical systems. Hume is based on:

- a multi-process almost-finite-state coordination language
- an almost-functional expression language for defining one-shot process behaviours
- a rich set of finite base and constructed types

Appendix 1 shows the Hume for a till in a chip shop[1]. A full account of the current state of Hume may be found in [HM00].

A major requirement for the Hume expression language(HEL) is that termination should be decidable. For Turing complete(TC) languages, termination is undecidable and heuristics for testing termination do not scale well. The following sections discuss possible constraints on recursion in HEL to aid analysis of termination.

2 No Recursion

The simplest way to ensure termination is to forbid recursion. This would give an impossibly restrictive language, so it might be extended with higher-order functions (HOFs) with known termination properties on finite types. A basic set for lists and integers might include:

[1] Buying fried food after a hard evening investigating the effects of alcohol on the higher thought processes is undoubtedly safety critical.

$$
\begin{array}{l}
map\ f\ [] = [] \\
map\ f\ (h:t) = f\ h : map\ f\ t \\
\\
foldr\ f\ b\ [] = b \\
foldr\ f\ b\ (h:t) = f\ h\ (foldr\ f\ b\ t) \\
\\
boundmin\ p\ b\ [] = b\ | \\
boundmin\ p\ b\ (h:t) = if\ p\ b\ h \\
\qquad\qquad\qquad\qquad\quad then\ boundmin\ p\ h\ t \\
\qquad\qquad\qquad\qquad\quad else\ boundmin\ p\ b\ t \\
\\
scan\ f\ b\ [] = [] \\
scan\ f\ b\ (h:t) = f\ h : scan\ f\ (f\ h)\ t \\
\\
iter\ n\ f\ b = b \\
iter\ n\ f\ b = iter\ (n-1)\ f\ (f\ b)
\end{array}
$$

Thus, HEL would effectively act as a coordination language for HOFs, analogous to Backus's FP[Bac78]. For example, the function `disps` in the till example in Appendix 1:

```
disps::int -> int -> displays
disps n v =
 if n=0
 then <>
 else (disps (n-1) (v/10))+<dispcode (v%10)>
```

could be written using *iter* as:

```
disp'::(int,displays) -> (int,displays)
disp' (v,q) = (v/10,q+<dispcode (v%10)>)

disps:: int -> int -> displays
disps n v = let (v',q') = iter n disp' (v,<>) in q'
```

The following structural operation semantics checks for recursion.

3 Identifying Recursion

3.1 Overview

The rules below are for a simple functional language:

$$
\begin{array}{lll}
exp \rightarrow name \mid & & \text{identifier} \\
\quad \lambda name.exp \mid & & \text{function} \\
\quad (exp\ exp) \mid & & \text{application} \\
\quad \texttt{if}\ exp\ \texttt{then}\ exp\ \texttt{else}\ exp & & \text{conditional} \\
\\
def \rightarrow \texttt{def}\ name = exp & & \text{definition} \\
\\
defs \rightarrow def \mid def; defs & & \text{definitions}
\end{array}
$$

These rules scale naturally to full HEL.

The central idea is that a function may call itself *directly*, with a reference to its associated name in its body, or *indirectly*, with a call to further functions in its body which ultimately call the original function. To explore the latter possibility, the function expression in a function application may be *simplified* to see if it is equivalent to calling the original function. It may also be expanded through *one-step-reduction*, which applies one stage of β-reduction, which may move occurrences of the function name in argument expression positions into function expression positions.

An initial *definition environment* is formed by associating every defined name with its value. Subsequently, a *visited environment* is maintained, to record which names have been replaced with their associated right hand sides, to prevent infinite replacement.

The following notation is used:

- D - definition environment - name/value associations
- $D \leftarrow_d defs$ - D is the definition bindings from $defs$
- V - visited environment - names
- $name \in_c exp$ - $name$ is called by exp
- $name \Leftarrow_s exp$ - exp simplifies to $name$
- $exp' \leftarrow_r exp$ - exp reduces in one step to exp'
- $exp[name/exp']$ - replace $name$ with exp' in exp

Note that these are purely static syntactic checks. Checking for non-recursive function calls which are equivalent to recursion, for example through the sue of Y, is undecidable.

3.2 Definitions

$$\boxed{\vdash D \leftarrow_d def; defs}$$

$$\frac{\begin{array}{c}\vdash D \leftarrow_d def \\ D \vdash D' \leftarrow_d defs\end{array}}{\vdash D + D' \leftarrow_d def; defs} \quad (1)$$

The definition environment from a sequence of definitions is formed by joining the definition environments from all definitions together.

$$\boxed{\vdash D \leftarrow_d \texttt{def}\ name = exp}$$

$$\frac{}{\vdash (name, exp) \leftarrow_d \texttt{def}\ name = exp} \tag{2}$$

The definition environment from a definition is formed by associating the defined name with the corresponding expression.

3.3 Simplifies from

$$\boxed{D, V \vdash name \Leftarrow_s name'}$$

$$\frac{name = name'}{name \Leftarrow_s name'} \tag{3}$$

A name simplifies from a second name if they are the same.

$$\frac{\begin{array}{c} D, V \vdash name \notin V \\ D, V \vdash (name', exp) \in D \\ D, name' + V \vdash name \Leftarrow_s exp \end{array}}{D, V \vdash name \Leftarrow_s name'} \tag{4}$$

A name simplifies from a name if the second name has not been visited and there is a definitions binding for that name, and the first name simplifies from the defined value.

$$\boxed{D, V \vdash name \Leftarrow_s \texttt{if}\ exp\ \texttt{then}\ exp'\ \texttt{else}\ exp''}$$

$$\frac{D, V \vdash name \Leftarrow_s exp'}{D, V \vdash name \Leftarrow_s \texttt{if}\ exp\ \texttt{then}\ exp'\ \texttt{else}\ exp''} \tag{5}$$

$$\frac{D, V \vdash name \Leftarrow_s exp''}{D, V \vdash name \Leftarrow_s \texttt{if}\ exp\ \texttt{then}\ exp'\ \texttt{else}\ exp''} \tag{6}$$

A name simplifies from a conditional expression if that name simplifies from either the then-option or the else-option.

$$\boxed{D, V \vdash name \Leftarrow_s (exp\ exp')}$$

$$\frac{\begin{array}{c} D, V \vdash (V', exp'') \leftarrow_r (exp\ exp') \\ exp'' \neq (exp\ exp') \\ D, V' \vdash name \Leftarrow_s exp'' \end{array}}{D, V' \vdash name \Leftarrow_s (exp\ exp')} \tag{7}$$

A name simplifies from a function application if the application one-step-reduces to an expression which is not the same as the application, and the name simplifies from that expression.

3.4 One-step-reduces

$$\boxed{D, V \vdash (V', exp') \leftarrow_r (name\ exp)}$$

$$\frac{D, V \vdash name \in V}{D, V \vdash (V,(name\ exp)) \leftarrow_r (name\ exp)} \tag{8}$$

An application of a name to an expression is not reduced if the name has been visited.

$$\frac{\begin{array}{c} D, V \vdash name \notin V \\ D, V \vdash (name, exp) \in D \\ D, name + V \vdash (V', exp') \leftarrow_r (exp\ exp) \end{array}}{D, V \vdash (V', exp') \leftarrow_r (name\ exp)} \tag{9}$$

An application of a name to an expression is reduced to a new expression, with a new visited environment, if the name has not been visited, there is a definition binding for the name, and the bound value for the name applied to the original expression reduces to the new expression and visited environment.

$$\boxed{D, V \vdash (V', exp'') \leftarrow_r (\lambda name.exp\ exp')}$$

$$\frac{D, V \vdash exp'' = exp[name/exp']}{D, V \vdash (V, exp'') \leftarrow_r (\lambda name.exp\ exp')} \tag{10}$$

An application of a function to an expression is reduced to a new expression, with the old visited environment, by replacing all occurrences of the bound variable in the body with the argument expression.

$$\boxed{D, V \vdash (V', exp''') \leftarrow_r ((exp\ exp')\ exp'')}$$

$$\frac{\begin{array}{c} D, V \vdash (V', exp''') \leftarrow_r (exp\ exp') \\ exp''' = (exp\ exp') \end{array}}{D, V \vdash (V,((exp\ exp')\ exp'') \leftarrow_r ((exp\ exp')\ exp'')} \tag{11}$$

An application of a function application to an expression is not reduced, and the visited environment is unchanged, if reducing the function application does not change it.

$$\frac{\begin{array}{c} D, V \vdash (V', exp''') \leftarrow_r (exp\ exp') \\ exp''' \neq (exp\ exp') \end{array}}{D, V \vdash (V,(exp'''\ exp'') \leftarrow_r ((exp\ exp')\ exp'')} \tag{12}$$

An application of a function application to an expression is reduced to a new expression applied to the second expression, with a new visited environment, if reducing the function application gives that new expression and visited environment.

$$\boxed{D, V \vdash (V', exp'') \leftarrow_r (exp\ exp')}$$

$$\frac{}{D, V \vdash (V,(exp\ exp') \leftarrow_r (exp\ exp')} \tag{13}$$

Any other form of function application is unchanged.

3.5 Called by

$$\boxed{D, V \vdash name \in_c \lambda name'.exp}$$

$$\frac{D, V \vdash name \in_c exp}{D, V \vdash name \in_c \lambda name'.exp} \tag{14}$$

A name is called by a function if it is called by the function body.

$$\boxed{D, V \vdash name \in_c \texttt{if}\ exp\ \texttt{then}\ exp'\ \texttt{else}\ exp''}$$

$$\frac{D, V \vdash name \in_c exp}{D, V \vdash name \in_c \texttt{if}\ exp\ \texttt{then}\ exp'\ \texttt{else}\ exp''} \tag{15}$$

$$\frac{D, V \vdash name \in_c exp'}{D, V \vdash name \in_c \texttt{if}\ exp\ \texttt{then}\ exp'\ \texttt{else}\ exp''} \tag{16}$$

$$\frac{D, V \vdash name \in_c exp''}{D, V \vdash name \in_c \texttt{if}\ exp\ \texttt{then}\ exp'\ \texttt{else}\ exp''} \tag{17}$$

A name is called by a conditional expression if it is called by the condition, the then-option or the else-option.

$$\boxed{D, V \vdash name \in_c (exp\ exp')}$$

$$\frac{D, V \vdash name \Leftarrow_s exp}{D, V \vdash name \in_c (exp\ exp')} \tag{18}$$

A name is called by an application if the function expression simplifies to the name.

$$\frac{D, V \vdash name \in_c exp}{D, V \vdash name \in_c (exp\ exp')} \tag{19}$$

A name is called by an application if it is called by the function expression.

$$\frac{D, V \vdash name \in_c exp'}{D, V \vdash name \in_c (exp\ exp')} \tag{20}$$

A name is called by an application if it is called by the argument expression.

$$\frac{D, V \vdash (V', exp'') \leftarrow_r (exp\ exp') \quad exp'' \neq (exp\ exp') \quad D, V' \vdash name \in_c exp''}{D, V \vdash name \in_c (exp\ exp')} \tag{21}$$

A name is called by an application if the application one-step-reduces to a different expression with a new visited environment and the name is called by that new expression and visited environment.

4 Primitive Recursion

Another alternative is to ground HEL in primitive recursion(PR) as all PR programs terminate and have easily characterisable time and space behaviour. However, as with TC programs, equivalence and complexity analysis of PR programs are undecidable, with complexity analysis heuristics often leading to further recursive forms which are not straightforward to characterise succinctly.

Crudely, PR differs from TC general recursion in forbidding unbounded minimisation. However, determining whether or not an arbitrary TC program is PR is also undecidable. Applying automatic program transformation and equivalence proof techniques to develop PRness heuristics would be a very interesting research project.

The classic formulation of PR provides for schemata for:

- incrementation
- constants
- selection from arguments
- function composition
- primitive recursion with base and increment-recursion cases

Thus Kleene [Kle52] characterises number theoretic PR as:

(I) $\varphi(x) = x'$
(II) $\varphi(x_1, ..., x_n) = q$
(III) $\varphi(x_1, ..., x_n) = x_i$
(IV) $\varphi(x_1, ..., x_n) = \chi(\chi_1(x_1, ..., x_n), ..., \chi_m(x_1, ..., x_n))$
(Va) $\varphi(0) = q$
(Va) $\varphi(y') = \psi(y, \varphi(y))$
(Vb) $\varphi(0, x_2, ..., x_n) = \psi(x_2, ..., x_n)$
(Vb) $\varphi(y', x_2, ..., x_n) = \chi(y, \varphi(y, x_2, ..., x_n), x_2, ..., x_n)$

where:

- q is a natural number
- i is an integer: $1 \leq i \leq n$
- n and m are positive integers
- ψ, χ, χ_i are number theoretic functions

Such schema may be extended to any data type which may be formed constructively from base values and finite constructor functions.

It has been shown[Péter67] that further extension to this schema retain PRness, in particular:

- course of values recursion, with several cases for (Vb) with base values other than 0
- simultaneous recursion, where several functions are defined in terms of each other
- recursion with substitution for parameters
- recursion with respect to several variables

Thus a more general primitive recursive form might be permitted, say:

$$
\begin{aligned}
&pr\ b_1\ x_1...x_n = f_1\ b_1\ x_1...x_n \\
&pr\ b_2\ x_1...x_n = f_2\ b_2\ x_1...x_n \\
&... \\
&pr\ y_1\ x_1...x_n = g_1\ y_1\ x_1...x_n\ (pr\ (h_1\ y_1)\ (i_{11}\ x_{11})...(i_{1n}\ x_n)) \\
&pr\ y_2\ x_1...x_n = g_2\ y_2\ x_1...x_n\ (pr\ (h_2\ y_2)\ (i_{21}\ x_{21})...(i_{2n}\ x_n)) \\
&...
\end{aligned}
$$

where:

- b_i are base patterns
- f_i, g_i and i_{ij} are primitive recursive
- $h_i...$ are "size reducing" for the type of b_i and y_i

Note that this schema is unnecessarily rigid. In practice arbitrary pattern sequences would be allowed on the left hand sides of definitions and a wider range of PRish constructs would be allowed on the right hand side.

For example, `disps` above could be simply rewritten:

```
disps::int -> int -> displays
disps 0 v = <>
disps n v = (disps (n-1) (v/10))+<dispcode (v%10)>
```

Burstall[Bur87] proposed the use of an extended *indcase* notation in a functional context, to define inductive cases from inductively defined data types. Here, notation is introduced to constrain recursion to always act on a component of

the "argument" to the *indcase* i.e. a component of the data type pattern on which a match is made. Burstall gives a number of examples, which suggest that use of *indcase* may be as forced as that of pure HOFs discussed above.

Turner's Elementary Strong Functional Programming distinguishes between PR and non-PR forms [Tur95]. The PR form allows any form where the right hand side may refer to syntactic sub-constructs of left hand side parameters. Turner's recently concluded project investigated abstract interpretation for determining termination. Turner suggests [2] Abel's termination checker [Abe99] as another avenue of investigation.

5 Identifying Nested Double Recursion

There are further subtleties in that nested double recursion may give otherwise PR programs greater complexity than true PR. For example, Ackermann showed that the function now named for him:

$$\alpha(0, a) = 0$$
$$\alpha(1, a) = 1$$
$$\alpha(n, a) = a$$

$$\xi'(0, b, a) = a + b$$
$$\xi'(n', 0, a) = \alpha(n, a)$$
$$\xi'(n', b', a) = \xi'(n, \xi'(n', b, a), a)$$
$$\xi(a) = \xi'(a, a, a)$$

has a faster growth rate than PR[3]. Here, the nested call to ξ' is unbounded in n'. On the other hand, many unbounded nested double recursive programs terminate.

The above rules for identifying recursion may be extended to check for nested double recursion. While this does not guarantee PR, the absence of nested double recursion would greatly simplify termination checks, and establishing time and space bounds.

Detecting self-nesting extends the detection of self-calling, by checking a function associated with a name for a direct or indirect occurrence of that name in an application's function expression, and for a direct or indirect call to that name in that application's argument expression. We write $name \in_n exp$ for *name* is nested by *exp*.

$$\boxed{D, V \vdash name \in_n \lambda name'.exp}$$

[2] Private communication.
[3] This formulation is after Kleene[Kle52].

$$\frac{name' \neq name \quad D, V \vdash name \in_n exp}{D, V \vdash name \in_n \lambda name'.exp} \tag{22}$$

A name is nested by a function if the bound variable is not the name and the name is nested by the body.

$$\boxed{D, V \vdash name \in_n name'}$$

$$\frac{D, V \vdash name' \notin V \quad D, V \vdash (name', exp) \in D \quad D, name' + V \vdash name \in_n exp}{D, V \vdash name \in_n name'} \tag{23}$$

A name is nested by a name if the second name has not already been visited, there is a binding for the second name in the definitions and the first name is nested by the defined value.

$$\boxed{D, V \vdash name \in_n \texttt{if}\ exp\ \texttt{then}\ exp'\ \texttt{else}\ exp''}$$

$$\frac{D, V \vdash name \in_n exp}{D, V \vdash name \in_n \texttt{if}\ exp\ \texttt{then}\ exp'\ \texttt{else}\ exp''} \tag{24}$$

$$\frac{D, V \vdash name \in_n exp'}{D, V \vdash name \in_n \texttt{if}\ exp\ \texttt{then}\ exp'\ \texttt{else}\ exp''} \tag{25}$$

$$\frac{D, V \vdash name \in_n exp''}{D, V \vdash name \in_n \texttt{if}\ exp\ \texttt{then}\ exp'\ \texttt{else}\ exp''} \tag{26}$$

A name is nested by a conditional if it is nested by the condition, then-option or else-option.

$$\boxed{D, V \vdash name \in_n (exp\ exp')}$$

$$\frac{D, V \vdash name \in_n exp}{D, V \vdash name \in_n (exp\ exp')} \tag{27}$$

$$\frac{D, V \vdash name \in_n exp'}{D, V \vdash name \in_n (exp\ exp')} \tag{28}$$

A name is nested by a function application if it is nested by the function expression or the argument expression.

$$\frac{\begin{array}{c} D, V \vdash name \Leftarrow_s exp \\ D, V \vdash name \in_c exp' \end{array}}{D, V \vdash name \in_n (exp\ exp')} \tag{29}$$

A name is nested by a function application if the function expression simplifies to the name and the name is called by the argument expression.

$$\frac{\begin{array}{c} D, V \vdash (V', exp'') \leftarrow_r (exp\ exp') \\ exp'' \neq (exp\ exp') \\ D, V' \vdash name \in_n exp'' \end{array}}{D, V \vdash name \in_n (exp\ exp')} \tag{30}$$

A name is nested by a function application if the application one-step-reduces to a new application which nests the name.

6 Rule Testing

These above rules have been implemented in Standard ML and tested on a variety of examples displaying simple and double nested recursion. For example, recursion is found in:

```
def r1 = fn f => fn n => if = n 0 then 1 else f (- n 1);
def r2 = fn n => (r1 r2) n

calls r2 fn n => ((r1 r2) n)
calls r2 ((r1 r2) n)
...
reduce (r1 r2)
reduce (fn f => fn n => if ((= n) 0)
```

```
                        then 1
                        else (f ((- n) 1))) r2
⇒ fn n => if ((= n) 0) then 1 else (r2 ((- n) 1))
calls r2 fn n => if ((= n) 0) then 1 else (r2 ((- n) 1))
...
calls r2 (r2 ((- n) 1))
simplifiesto r2 r2
true
```

For example, nested double recursion is found in:

```
def id = fn x => if true then x else y;
def double8 = fn n =>
               if = n 0
               then 1
               else id double8 (double8 (- n 1))

nests double8 fn n =>
               if ((= n) 0)
               then 1
               else ((id double8) (double8 ((- n) 1)))
nests double8 if ((= n) 0)
              then 1
              else ((id double8) (double8 ((- n) 1)))
...
nests double8 ((id double8) (double8 ((- n) 1)))
...
simplifiesto double8 (id double8)
reduce (id double8)
reduce (fn x => if true then x else y) double8
⇒ if true then double8 else y
simplifiesto double8 if true then double8 else y
...
true
calls double8 (double8 ((- n) 1))
simplifiesto double8 double8
true
```

7 Conclusions

A more general alternative for HEL is to allow a effectively unrestricted functional programming scheme, and rely on type systems, for example based on

Martin-Löf [NPS90], and/or termination checks to constrain programs. We propose to provide a relatively free HEL syntax over bounded data types and to provide a variety of tools that check for varying degrees of constraints on programs.

Hume is very much in its infancy and this paper should be viewed as a contribution to Hume's ongoing evolution. Once Hume's design has stabilised and a reference system has been constructed, it will be easier to gauge the practical implications of strong and weak restrictions on recursion for substantive programming.

Appendix 1 - Hume Example: Chip Shop Till

The till has a keyboard, with keys for items of food, numbers and to clear the till, and an 8-character 7-segment display. The till decodes the keys and drives the display. The display always shows the current rolling cost. The clear key resets the rolling cost and display to 0. The till holds a menu which relates food items to costs. When a multi-digit quantity is followed by a food item, the till multiplies the cost by the quantity and adds the result to the rolling cost. The till is shown in Figure 1.

Fig. 1. Till boxes and wiring

Keyboard signals are modeled as discriminated unions, with functions to decode numeric keys to numbers and to find the prices of food items:

```
union digit = one | two | three | four | five |
              six | seven | eight | nine | zero

union item = pie | cola | chips | fish

union key = D digit | I item | clear

decode::digit -> int
decode d =
 case d of
 zero -> 0
 ...
 nine -> 9

menu::item -> int
menu i =
 case i of
 pie -> 85
 cola -> 95
 chips -> 70
 fish -> 145
```

A 7-character display is modeled as a vector of bits, with functions to generate the segment settings on one display from a digit, and the settings on all eight displays from an integer:

```
type bit = word 1
type display = vector 7 of bit
type displays = vector 8 of display

dispcode::int -> display
dispcode c =
 case c of
  0 -> <1,1,1,1,1,1,0>
  ...
  9 -> <1,1,1,0,0,1,1>

disps::int -> int -> displays
disps n v = if n=0
            then <>
            else (disps (n-1) (v/10))+<dispcode (v%10)>

disp8::int -> displays
```

```
disp8 v = disps 8 v
```

A till has inputs for the next key, rolling cost for all items so far, and partial quantity for the next item. It also has outputs for the displays, as a vector of 8 display vectors, and for the rolling cost and quantity:

```
box till
in k::key,cost::int,quant::int
out codes::displays,cost'::int,quant'::int
 match
  (clear,c,q) -> (disp8 0,0,0)
  (D d,c,q) -> (disp8 c,c,10*q+decode d)
  (I i,c,q) -> let c' = c+q*menu i in (disp8 c',c',0)
```

The 8 display vectors must be fanned out to 8 separate displays:

```
box fanout
in codes::displays
out disp1::display,disp2:;display,disp3::display,disp4::display,
    disp5::display,disp6:;display,disp7::display,dusp8::display
match
 <d1,d2,d3,d4,d5,d6,d7,d8> -> (disp1,disp2,disp3,disp4,
                               disp5,disp6,disp7,disp8)
```

Similarly, each 7 bit display vector must be fanned out to 7 separate segments:

```
box display_driver
in code::display
out seg1::bit,seg2::bit,seg3::bit,seg4::bit,
    seg5::bit,seg6::bit,seg7::bit
 match
  <b1,b2,b3,b4,b5,b6,b7> -> (seg1,seg2,seg3,seg4,seg5,seg6,seg7)

replicate dd as 8 * display_driver
```

Finally, the till is wired to the keyboard and the display fanout, with the rolling cost and quantity accumulating from outputs to inputs. The fanout is then wired to each display:

```
wire till (keyboard,till.cost',till.quant')
          (fanout.codes,
           till.cost initially 0,till.quant initially 0)

wire fanout (till.codes)
            (dd{1}.code,dd{2}.code,dd{3}.code,dd{4}.code,
             dd{5}.code,dd{6}.code,dd{7}.code,dd{8}.code)
```

```
for i from 1 to 8
 wire dd{i} (fanout.disp{i})
             (s{i}1,s{i}2,s{i}3,s{i}4,s{i}5,s{i}6,s{i}7)
```

Note that `i` is macro expanded. Note that `sij` is the `j`th segment wire on the `i`th physical display character.

Acknowledgments

I am pleased to acknowledge Hume's co-designer Kevin Hammond, for considerable discussion of the ideas presented here, and David Turner, for clarifications of properties of primitive recursion and their implications for language design. I would like to thank Hans-Wolfgang Loidl, Robert Pointon and Phil Trinder for helpful comments on this paper.

References

[Abe99] A. Abel. Eine semantische Analyse strukturelle Rekursion. Diploma Dissertation, February 1999.

[Bac78] J. Backus. Can Programming be Liberated from the Von Neumann Style? *Communications of the ACM*, 21(8):287–307, 1978.

[Bur87] R. Burstall. Inductively Defined Functions in Functional Programming Languages. Technical Report ECS-LFCS-87-25, LFCS, University of Edinburgh, 1987.

[HM00] K. Hammond and G. Michaelson. *The Hume Report 0.0.* University of St Andrews and Heriot-Watt University, July 2000.

[Kle52] S. C. Kleene. *Introduction to Meta-mathematics.* North-Holland, 1952.

[NPS90] B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory.* Oxford, 1990.

[Péter67] R. Péter. *Recursive Functions.* Academic Press, 1967.

[Tur95] D. Turner. Elementary Strong Functional Programming. In R. Plasmeijer and P. Hartel, editors, *First International Symposium on Functional Programming Languages in Education*, volume 1022 of *LNCS*, pages 1–13, Nijmegen, The Netherlands, December 1995. Springer-Verlag.