Journal of Universal Computer Science, vol. 23, no. 1 (2017), 89-111
submitted: 2/8/16, accepted: 4/12/16, appeared: 28/1/17 © J.UCS

A General Method for Defining Objects by Recursion over Syntax

Simon Thompson
(University of Kent, U.K.
S.J.Thompson@kent.ac.uk)

Abstract: In this paper we look back to work done in the late 1980s, and that looked at links between grammars, data types and recursion principles, and illustrating it with examples that include enumerations of types and developing a structure editor. The work is introduced by a historical foreword, and closes with an afterword that discusses some of the subsequent developments of these ideas.

Key Words: algebraic data type, coinduction, generic programming, grammar, higher-order function, Miranda, recursor, syntax, verification
Category: D.1.1, D.2.4, D.3.3

1 Historical foreword

This paper was written in the spring of 1987, and presented at the Third British Theoretical Computer Science Colloquium, which took place in April 1987 at Leicester University. It is presented essentially unchanged from then, apart from adding this historical perspective to the introduction, updating the bibliography, adding a small number of explanatory footnotes, correcting a number of typos and concluding with an afterword that gives some pointers to subsequent developments of the work reported here.

The paper is dedicated to David Turner, who made a number of insightful comments about it at the time; working with David in the 1980s and 1990s was a privilege and a fantastic education. I would also like to acknowledge Ron Knott's work, as well as the contributions of Allan Grimley, who was also a colleague at the time. The paper uses Miranda [Turner 1985], of course; anyone familiar with Haskell should have little difficulty in understanding it.

The motivation for this investigation was a talk presented in late 1986 to the Theory Seminar at the University of Kent by Ron Knott of the University of Surrey [Knott 1987]; related work can be found in [Burge 1978, Fairbairn 1987]. Knott in his talk discussed definitions of functions related to context-free grammars, and in particular showed how both generators and parsers could be seen as being defined in essentially the same way, using *continuations.* Because I found continuations rather difficult to grasp properly – I find that I could understand definitions written using them, but had difficulty in writing my own definitions – I wanted to see if I could think about the problem myself in the context of

Miranda, and at the same time address two issues which had occurred to me in the course of Ron's talk.

The work explores some known ideas, but also hints at ideas which subsequently would be crystallised to become generic programming; it draws early links between this and dependently typed programming too, and this was a topic on which I was to write a book not long after.

2 Introduction

This paper examines the link between the definition of context free grammars, algebraic data types in the Miranda functional programming language, and definitions of functions or other values, such as enumerations and structure editors. The work explores some known ideas, but also hints at ideas which subsequently would be crystallised to become generic programming; it also draws early links between this and dependently typed programming.

- Miranda has a natural way of representing grammars as types, specifically *algebraic* data types, and so it is possible that we could build an operator which embodied the definition principle, in the same way that the `foldr` operator represents a form of primitive recursion over lists. (Remember that the analogues of `foldr` can be defined in a uniform way for all regular, non-nested algebraic data types.)
- One class of syntax-directed definition which I have found interesting is the *structure editor*. It seemed from the discussion during the talk that the continuation method did not allow us to build such editors. Any new scheme that we introduce should be able to treat such a case. We discuss this further in Section 11

In Section 3 we introduce the algebraic type mechanism and show how two different links are made with grammars. We pursue the one whereby each grammar is represented by an algebraic type, or a collection of mutually recursive algebraic types. In Section 4 we look at primitive recursion over these types, before we meet the generalisation, motivated by an example, in Section 5.

We give a full treatment of that example – of enumerating a type – in Section 6, and in Section 7 show how this 'general' recursion is indeed a generalisation of primitive recursion. Section 8 looks at the example of the structure editor. In the latter sections we concentrate on one example type, that of lists, but it should be clear that this material is of general applicability. (We hope that the material in Section 3 suggests that this is indeed the case.)

Section 9 examines the generality of the definition scheme, and then in Section 10 we discuss the effects on program verification. Our alternative approach, and its links with Knott's work are explored in Section 11, and we conclude in Section 13 with a discussion on the uniformity of the definitions we have made, and the links between this and the richness of the type system which we use.

3 Grammars and algebraic types

In this section we look at the relation between grammars and the algebraic types of Miranda [Turner 1985]. Algebraic types are specified by means of *constructors*, as in the following example, where we introduce binary trees, with integer labels at their leaves.

```
tree ::= Leaf integer |
         Node tree tree
```

`Leaf` and `Node` are the constructors in this example. We can read this as saying that a `tree`[1] is either a `Leaf` which contains an integer, or ('`|`') is a `Node` with two sub-trees. We see in this example that a type can be defined so as to be recursive.

```
Leaf 27
Node (Leaf 44) (Node (Leaf 2) (Leaf 3))
```

are two examples of objects of this type.

We can think of this definition in a number of different ways

- Each of the alternatives, delimited by `'|'`, introduces one variant of a variant record declaration, with the types of the fields following the name itself.
- Each of the alternatives introduces a function, which returns an object of type `tree`. The arguments to the function are listed after its name. As all functions in Miranda are curried (that is taking a single argument, and so allowing partial parametrisation), their types in the example will be:

  ```
  Leaf :: integer -> tree
  Node :: tree -> tree -> tree
  ```

  These functions are special in that they have no accompanying evaluation rules — they are pure *constructors* which create values of the type concerned.
- Each alternative tells us one sort of node in the parse tree for (expressions denoting) these objects, which can be thought of syntactically.

This last alternative can be illustrated clearly by the following example, in which we first present a grammar, in BNF form, and then a type.

[1] Note that in Miranda type *names* use identifiers that begin with small letters; type *variables*, which we will use later in the paper, are denoted `*`, `**`, `***` and so forth.

```
<expr> ::= <num> | <expr> + <expr> |
           <expr> - <expr> | ...
```

is a portion of a grammar for simple numerical expressions, which has the corresponding type definition

```
expr ::= Num num | Add expr expr |
         Sub expr expr | ...
```

which constitutes the collection of parse trees for the given grammar.

Most grammars of any significance contain a number of syntactic categories, each of which may be defined in terms of the others. For instance, in the syntax for an imperative language we might see

```
<comm> ::= while <expr> do <comm> | ...
<expr> ::= do <comm> return <expr> | ...
```

Such a grammar will be represented by a collection of mutually recursive algebraic types:

```
comm ::= While_Do expr comm | ...
expr ::= Do_Return comm expr | ...
```

Simple grammars which are defined by a single equation (such as the first expression grammar we saw above) can be represented in Miranda in a slightly different way. Using the language of *regular expressions* we can represent such grammars as objects of a regular expression type such as

```
grammar ::= Empty |
            Token char |
            Or  grammar grammar |
            And grammar grammar
```

The type of expressions

```
<expr> ::= <num> | <expr> + <expr> |
           <expr> - <expr> | ...
```

can be represented by the object

```
expr =
  numeric $Or
  (expr $And Token '+' $And expr) $Or
  (expr $And Token '-' $And expr)
```

where `$And, $Or` are the infix forms of the (prefix) functions `And, Or`. Note that we can express more general grammars than regular ones in this notation, since the *objects* of the type can themselves be defined by recursion. Indeed, we see an example of this in the `expr` type here.

We shall discuss this approach further in Section 11.

(The following paragraph can easily be omitted on first reading.)

Before we close this section we remark that the Miranda algebraic type mechanism is slightly more general than might be guessed from the examples we saw above. In those examples, each 'node' had a finite collection of *predecessors* — for example, for our original tree type, a `Leaf` node has one numeric predecessor and an inner node has two `tree` predecessors. The types of the predecessor objects can be rather more general, and indeed can be of any type denoted by a type expression, and not just by a type name. One example which we can give is

```
ordinal ::= Zero |
            Successor ordinal |
            Limit (nat -> ordinal)
```

A `Limit` object has only one predecessor, which is a function with range type `ordinal` and domain type the natural numbers (`nat`). This node represents a node with an *infinite* number of `ordinal` predecessors. Such types, in which the type under definition only appears to the right of the function space constructor `->` we call *covariant*. (We have not been completely rigorous in our definition of the term 'covariant', but the reader should not find it too hard to fill in the details for him- or her-self.) One other remark might be relevant. If we see a type which contains a tuple type as an argument type to one of its constructors, we can replace that tuple type by its components separately — in other words we assume that all our constructors are fully *curried*. For instance, we replace

```
  Constr (t1,t2) (t2,t3)
```

by

```
  Constr t1 t2 t2 t3
```

In the next section we look at how functions are defined over algebraic types.

4 Primitive Recursion

Primitive recursion is the fundamental process whereby we define functions over algebraic types. We saw in the last section how objects of these types were constructed, using constructors such as `Leaf` and `Node`. In this section we show how functions are defined by recursion over the structure of an object, a process which we call *primitive recursion*. We start with an example. Given our `tree` type, we can write a function to add the values contained in the tree thus:

```
add (Leaf n)     = n
add (Node t1 t2) = add t1 + add t2
```

Observe that in Miranda we use *pattern matching* to specify alternative cases in definitions and to access the components of structured objects.

The values at the leaves can be calculated outright, but at the inner nodes we have to call the function `add` recursively on the component parts. What is special about such a recursion is that, assuming the tree is finite, eventually we reach the base case, and so assuming that our combining operation always terminates, as `+` indeed does, then the defined function, `add` here, will be total.

We can look at this as an application of a general process. To define a function

```
tree -> result
```

we need to supply two things.

- We should supply its values outright on the leaf nodes, so we should supply a function

  ```
  st :: num -> result
  ```

  In the case above, `result` is `num` and we supply the *identity* function.

- We need to supply the means by which we calculate the value of the function at `Node t1 t2` in terms of its values at `t1` and `t2`. We therefore need to supply a function of type

  ```
  comb :: result -> result -> result
  ```

  We used `(+)` in our example above.

Given these two objects, we define our primitive recursive function, `fun` say, straightforwardly:

```
fun (Leaf n)     = st n
fun (Node t1 t2) = comb (fun t1) (fun t2)
```

We can define a higher-order function, `prim_tree`, which produces this result, given the parameters:

```
prim_tree st comb (Leaf n) = st n
prim_tree st comb (Node t1 t2)
   = comb (prim_tree st comb t1) (prim_tree st comb t2)
```

or

```
prim_tree st comb
  = fun
    where
    fun (Leaf n)     = st n
    fun (Node t1 t2) = comb (fun t1) (fun t2)
```

Another example of a primitive recursor is the function `foldr` from the Miranda standard environment which defines functions over lists; this is, in fact, the primitive recursion operator for the list type, which can be seen to be (equivalent to) an algebraic type. Specifically, it has constructors

```
[]  :: [*]
(:) :: * -> [*] -> [*]
```

(where `(:)` is the prefix form of the infix operator '`:`').

It should be clear that we can derive the primitive recursion operation for *any* algebraic type straight from its definition. Just to make this plain, we examine the operation from a slightly different viewpoint. In performing the recursion we used two parameters,

```
st   :: num -> result
comb :: result -> result -> result
```

Compare their types with those of the constructors

```
Leaf :: num -> tree
Node :: tree -> tree -> tree
```

we see that the types are the same, except that `result` has replaced `tree` throughout. We can think of them as *shadow constructors* and of the operation of primitive recursion as *taking an object of the type and replacing each constructor by its shadow.* For example, in `adding` we replace `Leaf` by `id` and `Node` by `(+)` so we turn

```
Node (Leaf 2) (Node (Leaf 3) (Leaf 5))
```

into

```
(+)    id 2   ( (+)   id 3     id 5 )
```

giving

```
(+)     2     ( (+)    3        5   ) = 2+(3+5) = 10
```

This explanation is clearly one which allows us to define the primitive recursor for each algebraic type we encounter. We shall return to the question of the *uniformity* of this definition in Section 12.

The idea of shadow constructors motivates our generalisation, which is to be found in the next section. Before we close, we look at the definition of primitive recursion over algebraic types with mutually recursive definitions.

We saw in the last section that grammars lead to mutually recursive types — how do we perform structural recursion over such types? Very much in the way in which we proceed over a single type.

Since we define our types simultaneously, it should be no surprise that for a collection of mutually recursive types we define a collection of functions *simultaneously* over all the types. Suppose that we have types `t1,...,tk` then we define functions

```
fun_i :: ti -> result_i
```

simultaneously from a collection of shadow constructors whose types are derived by replacing each occurrence of `ti` by `result_i`. We can enshrine such a definition in a higher-order function, which will return the tuple of functions as its result, or we could define `k` operators which are mutually recursive, each of which returns the function over the appropriate type. We see examples of these recursions when we look at the *semantics* of a programming language, for instance. Returning to the example we looked at in the last section, for the grammar represented by

```
comm ::= While_Do expr comm | ...
expr ::= Do_Return comm expr | ...
```

we define the functions

```
comm_value :: comm -> state_transformation
expr_value :: expr -> dependent_value
```

by primitive recursion over the syntax. Each of the functions will, on the appropriate nodes, call the other function on the appropriate sub-components, as for instance, we need to evaluate the expression `e` in `While_Do e c` in order to determine the effect of the loop.

In the following section we look at the generalisation we sought to make.

5 General recursion

The idea we introduce here is related to primitive recursion, at least in the way that the definition is parametrised. Recall that primitive recursion can be thought of as acting thus: given a type (say `[eric]`) with constructors

```
[]  :: [eric]
(:) :: eric -> [eric] -> [eric]
```

we introduce a primitive recursive function of type

```
f :: [eric] -> range
```

by supplying *shadow constructors* of the form

```
sha_nil  :: range
sha_cons :: eric -> range -> range
```

The function `f` works on a particular element of the type `[eric]` by replacing each constructor in the element by its shadow. If we choose the definitions

```
sha_nil  = 0
sha_cons = (+) . weight
```

where `weight :: eric -> num`, the primitive recursion produces a weighted sum of the list.

The idea we introduce here is also based on the idea of shadow constructors, but with a more general emphasis (we shall see that primitive recursion is a special case of our definition in Section 7). We take as a motivating example that of enumerating, in a potentially infinite list, all the objects of a particular type. As in the explanation above, we choose the type of lists of the arbitrarily chosen type `eric`.

Lists are constructed by applications of the constructors, and so our enumeration of the objects will consist of

– An enumeration of the objects constructed using `[]`, and

– An enumeration of the objects constructed using `(:)`, that is the enumeration of the non-empty lists.

We can produce the list of all empty lists outright

```
[[]] :: [[eric]]
```

but how are we to produce the list of non-empty lists? If we were given the enumeration of the whole type (here we are thinking *recursively*, note) we could construct the non-empty enumeration by prepending each element of type `eric` to every list in the enumeration. If we consider the function which performs this operation of prepending an item to every list in an enumeration, we see that it is of type

```
prepend :: eric -> [[eric]] -> [[eric]]
```

These two functions *shadow* the constructors in the way we saw above, since their types are the types of the constructors with the result type, `[[eric]]`, replacing the domain type, `[eric]`, throughout.

Our definition will be complete once we have described how we use, or *combine* these two functions to give our result. We combine them in a particular way, in that we only apply `prepend` to the object we seek to define. In general, we only combine the *saturated* forms of the shadow constructors, where a shadow constructor is saturated by applying it to the result. More formally, we say that a constructor is saturated by having the `result` passed to each of its arguments of result type. In formal terms

```
enumerate
  = combine [[]] prepend_sat
    where
    prepend_sat a = prepend a enumerate
```

and where `combine` combines the two saturated shadows in the way we outlined above.

This definition is perfectly general. Suppose that we are supplied with shadow constructors of the form

```
shadow_nil  :: target
shadow_cons :: eric -> target -> target
```

Assuming that we have the result of our definition, `result :: target`, in the usual recursive tradition, we substitute it for each argument of type `target` in the shadow constructors, giving us the *saturated functions*

```
shadow_nil' :: target
shadow_nil'  = shadow_nil

shadow_cons'  :: eric -> target
shadow_cons' e = shadow_cons e result
```

Now we introduce the final parameter of our recursion — a function which *combines* these two saturated constructors to give us something of type `target`. The type of this function is therefore

```
comb :: target -> ( eric -> target ) -> target
```

If we replace the arbitrary type identifiers `target` and `eric` with proper Miranda type variables `* , **` then we can state the type and definition of our polymorphic definition scheme, which we call `gen_rec`

```
gen_rec ::    (* -> ( ** -> * ) -> *) ->
```

```
                 * ->
                 (** -> * -> *) ->
                 *
```

and the definition, restated in Miranda, is

```
gen_rec comb shadow_nil shadow_cons
      = result
        where
        result = comb shadow_nil' shadow_cons'
        shadow_nil'    = shadow_nil
        shadow_cons' e = shadow_cons e result
```

There should be no difficulty in seeing how the analogue of `gen_rec` is to be defined for other algebraic types. We simply perform the shadowing process and combination as above.

In the following section we specify the details of the enumeration we introduced above.

6 Enumerating the lists of values of a given type

Given an enumeration of a type, `eric`, say, we can construct an enumeration of the type of lists of `eric`. We use our `gen_rec` function to build this object of type `[[eric]]`. The definition was hinted at in the previous section. We present it now, and follow it with an explanation.

```
enum :: [eric] -> [[eric]]

enum listing
      = gen_rec intermingle shadow_nil shadow_cons
        where
        shadow_nil  = [[]]
        shadow_cons = map . (:)
        intermingle l1 f = l1 ++ interleave (map f listing)
        interleave (a:x) = merge a (interleave x)
                           where
                           merge (a:x) y = a : merge y x
```

The shadow of `nil` is the enumeration of the `nil` lists `[[]]`, and the shadow of `cons` is the function which conses its first argument onto each list in its second. We combine the two functions first by constructing an enumeration of the non-empty lists

```
interleave (map shadow_cons' listing)
```

where

```
shadow_cons' a = shadow_cons a result
```

and then by appending the two enumerations together. The object `listing` is the enumeration of the type `eric`, i.e. an object of type `[eric]`, and `interleave` is a function which takes an infinite list of lists and interleaves them.[2]

7 Application – primitive recursion

Recall the function `foldr`, which we introduced in Section 5. This is the object which performs primitive recursion over the type of lists. We shall re-define the operation here, using the `gen_rec` function.

```
new_foldr :: (** -> * -> *) -> * -> [**] -> *
```

The object which we are trying to define is of type

```
[joe] -> target
```

and suppose that we are trying to define

```
new_foldr trans st
```

To define a function of type `[joe] -> target` using `gen_rec` our shadow constructors need to be of type:

```
shadow_nil  :: [joe] -> target
shadow_cons :: joe -> ( [joe] -> target ) ->
               ( [joe] -> target )
```

What are these to be, and how are their partial applications to be combined, i.e. how are we to define the `comb` argument? Consider the latter question first. Suppose that the `shadow_nil'` function provides the *base* part of the function, and that the `shadow_cons'` provides the non-base part: how are these to be combined?

```
comb :: ( [joe] -> target ) ->
        ( joe -> ( [joe] -> target ) ) ->
        ( [joe] -> target )

comb f g []    = f []
comb f g (a:x) = g a x
```

[2] Note that the definition here assumes that the type `eric` is infinite, and so that no base cases are needed for the functions `interleave` and `merge`: it is an exercise to add these for the finite case.

How is the base part of the function defined from the starting value `st`?

```
shadow_nil x = st
```

i.e.

```
shadow_nil = const st
```

Now, the non-base part of the function is given by `shadow_cons'` where

```
shadow_cons' a = shadow_cons a result
```

so we require that

```
shadow_cons a result y
      = trans a (result y)
      = ( (trans a) . result ) y
```

therefore

```
shadow_cons a result
      = (trans a) . result
      = ( (.) (trans a) ) result
```

and

```
shadow_cons a
      = (.) (trans a)
      = (.) (.) trans a
```

so finally,

```
shadow_cons = (.) (.) trans
```

These definitions of the shadowing constructors in terms of the parameters to `foldr` are completely general, so we say, replacing `joe`, `target` by `**`, `*`

```
mk_shadow_nil :: * -> ( [**] -> * )

mk_shadow_nil = const

mk_shadow_cons :: ( ** -> * -> * ) ->
                  ( ** -> ( [**] -> * ) -> ( [**] -> * ) )

mk_shadow_cons = (.) (.)
```

and re-write the definition thus:

```
new_foldr f st
      = gen_rec comb shadow_nil shadow_cons
        where
        shadow_nil  = mk_shadow_nil st
        shadow_cons = mk_shadow_cons f
```

This completes our re-definition of `foldr` in terms of our new definition scheme `gen_rec`. Now we go on to look at a final example.

8 A structure editor

In this section we build a structure editor for lists of `joe`, using an input device for the type `joe`. This latter might be another structure editor, but could be another form of device. The framework we use is that of interactions, and we depend in particular on functions whose definitions are to be found in [Thompson 1986].

An object of type `interact start stop` is to be thought of as an interactive program which starts with a state value of type `start` and terminates with a state value of type `stop`. `uni` is the one element type, defined by `uni ::= Uni`, and so a natural type for the structure editor will be `interact uni [joe]`. In turn, our defining function will be of type

```
struct_ed :: interact uni joe -> interact uni [joe]
```

where the argument is the input routine for the `joe` type. Now, let

```
struct_ed in_elem
      = gen_rec (stick in_elem) nil_ed cons_ed
```

How do we define the two component editors, `nil_ed, cons_ed` and the combining functional? To deal with the simplest first, the `nil_ed` simply returns the only possible nil list:

```
nil_ed :: interact uni [joe]
nil_ed  = start []
```

The `cons_ed` is allowed (in the final `result`) to access `result` itself, using the parameter `edit`. This is used to input the tail of the list, onto which the first argument is consed:

```
cons_ed :: joe -> (interact uni [joe]) -> (interact uni [joe])

cons_ed a edit
      = seq3    ( writeln "Please enter the tail of the list" )
                edit
                ( apply ((:) a) )
```

Finally we have to `stick` the components together:

```
stick :: (interact uni joe) ->
         (interact uni [joe]) ->
         (joe -> interact uni [joe]) ->
         (interact uni [joe])
```

where the first argument is the input routine for `joe`, and the second and third are the component editors. We first ask the user to make a choice between the two alternatives, done in rather a crude way by asking for a zero or non-zero integer. Based on that choice we either ask for an empty list using `nil_ed'`, or ask for the head and tail of a non-empty list. In the latter case we use `in_elem` to input the head of the list, and then `cons_ed'` for the tail.

```
stick in_elem nil_ed' cons_ed'
  = seq ( in_int ("Enter an integer, zero for the empty" ++
                       newline ++
                       "list or non-zero for a non-empty list: ")
                      "" (const True)  )
                ( alt ( (=) 0 )
                      nil_arm
                      cons_arm )
        where
        nil_arm = seq forget nil_ed'
        cons_arm = seq3
                   forget
                   (writeln
                      "Please enter the head of the list")
                   (pass_param in_elem cons_ed')
```

Note that the definition we have outlined is generic in the type `joe`, and so the definition will be polymorphic, as we have seen others in previous sections to be.

9 How general is gen_rec?

An arbitrary recursion can be written in the form

```
x = f x                           (1)
```

that is, arbitrary recursive definitions are the fixed points of the appropriate functions. Consider the case of `gen_rec` for lists once again.

```
result = comb shadow_nil'         (2)
              shadow_cons'
```

where

```
shadow_cons' e = shadow_cons e result
```

We can rewrite equation (1) in the form of equation (2) if we say

```
shadow_cons e result = result
```

and

```
comb a b = f (b ⊥)
```

since

```
result = comb shadow_nil' shadow_cons'
       = f (shadow_cons' ⊥)
       = f result
```

which is a rewriting of equation (1) in terms of the variable `result`.

We will be able to recast an arbitrary recursion in terms of the `gen_rec` operator for *any recursively defined type*, so we see that in the interesting cases, general recursion is as general as it might be. (Note that the operators `gen_rec` for *non*-recursive types are simply a form of composition operator, and embody as much recursion as there is in the type definition — that is *none*.)

Given this generality, what can we say in favour of this operator? It certainly allows us to make definitions which *seem* to share a common structure actually *have* that common structure, and thus make our definitions more comprehensible. In the case of more complex functions, like the structure editor, we have given a principle according to which our definitions are almost mechanical — given a type definition and the general recursor for that type it is plain to see how to write the structure editor. There are also advantages in the way that we structure proofs, a topic we look at next.

10 Program verification

We prove properties of recursively defined objects by fixed point induction (see, for example, [Paulson 1985]). If `x` is defined by `x = f x` then

$$\frac{\mathtt{P}(\perp) \qquad \mathtt{P(y)} \Rightarrow \mathtt{P(f\ y)}}{\mathtt{P(x)}}$$

This is a powerful inference principle, and will give us properties of functions defined by `gen_rec`. However there is a principle of induction which more directly reflects the definition of `gen_rec`:

$$\frac{\begin{array}{c}\mathtt{P}(\perp)\\ \mathtt{Q(shadow\_nil)}\\ \mathtt{P(f)} \Rightarrow \forall\mathtt{a.\ R(shadow\_cons\ f\ a)}\\ (\mathtt{Q(x)} \wedge \forall\mathtt{a.\ R(shadow\_cons\ f\ a)}) \Rightarrow \mathtt{P(comb\ x\ f)}\end{array}}{\mathtt{P(result)}}$$

We should sound a note of caution at this point. In practice, most of the properties of recursively defined objects, such as the enumeration of Section 5, will not be proved directly using fixed point induction. Instead, in [Thompson 1989] we suggest alternative rules which reflect our intuitive reasoning more accurately. We expect that we shall find the appropriate variants of these rules which correspond to `gen_rec`, but the topic is still under investigation at the time of writing.

11 Primitive recursion over the type of grammars

Recall that in Section 3 we introduced the algebraic type `grammar` of grammars:

```
grammar ::= Empty |
            Token char |
            Or  grammar grammar |
            And grammar grammar
```

As for other algebraic types, the principal form of definition over this type is primitive recursion.

```
prim_grammar :: * ->
                (char -> *) ->
                (* -> * -> *) ->
                (* -> * -> *) ->
                grammar -> *
```

with a definition given by

```
prim_grammar emp base proc_or proc_and
  = f
    where
    f Empty       = emp
    f (Token tok) = base tok
    f (Or g1 g2)  = proc_or  (f g1) (f g2)
    f (And g1 g2) = proc_and (f g1) (f g2)
```

Our first example of its use is to write a function which, when given a grammar, generates all the strings of that grammar, as a list of character strings.

```
generate :: grammar -> [[char]]
```

with the definition

```
generate = prim_grammar emp lit altern conc
           where
```

```
emp     = [""]
lit tok = [[tok]]
altern (a:x) y = a : altern y x
altern []    y = y
conc x y = map konk [ (p,q) // p <- x ; q <- y ]
             where
             konk (a,b) = a++b
```

`emp` is the list of words generated by the grammar `Empty`, that is the list consisting of the empty string, and similarly, `lit` is the function which when given a token (i.e. a character) returns the list of strings generated by that token. The interesting cases are those corresponding to the `Or` and `And` constructors, which are replaced by the *shadow* constructors

```
altern :: [[char]] -> [[char]] -> [[char]]
conc   :: [[char]] -> [[char]] -> [[char]]
```

What are the effects of these functions? `altern` takes two lists of strings and interleaves them. This is necessary, rather than a simple `join` (or append, `++`), because one or both of the lists may be infinite.

The `conc` function takes two lists of strings and returns the list made by concatenating each string from the first with each string from the second.

Before we go any further, perhaps we should give some *examples* of grammars:

```
zero , one , ones , bin :: grammar
zero = Token '0'
one =  Token '1'
ones = one $Or (one $And ones)
bin = zero $Or one $Or ( (zero $Or one) $And bin )
```

`ones` is the grammar generating the strings of ones, and `bin` generates all strings made up from 0 and 1.

As these are grammars, we should be able to represent parse trees for the grammars in some way. We introduce the type `tree` to do that. Observe that this is isomorphic to the subtype of grammar generated by the constructors `Empty`, `Token` and `And`.

```
tree ::= Null |
         Leaf char |
         Node tree tree
```

Now we can introduce the parser generating function, defined over the `grammar` type. Parsers will be built from *partial parsers*, i.e. objects of type

```
parser == [char] -> [ ( [char] , tree ) ]
```

The functions return a list, each item of which consists of the results of a partial parse, i.e. the remainder of their input, together with the parse tree built so far.

```
parse :: grammar -> parser
parse = prim_grammar parse_emp
                     parse_tok
                     parse_or
                     parse_and
```

Here again we see primitive recursion in action. It is parametrised by the four shadow_constructors which are the arguments above.

```
parse_emp :: parser
parse_emp l = [ ( l , Null ) ]
```

The result of recognising an empty string as prefix is to return the same string of characters, together with the parse tree `Null` for the `Empty` string.

```
parse_tok :: char -> parser
parse_tok ch (a:x) = [ ( x , Leaf ch ) ] , ch = a
                   = []                  , otherwise
parse_tok ch []    = []
```

Again, parsing a token presents no problem: we either return an empty list, signifying failure, in the case that input is exhausted or that the head of the input fails to be the token sought. We return the input with the recognised token removed, and the parse tree for that token in the case of success.

```
parse_or :: parser -> parser -> parser
parse_or p1 p2 l = p1 l ++ p2 l
```

In case of trying to recognise a string from either of two grammars, we simply apply the parsers for each grammar to the input, and return the combined list of *all* their results.

```
parse_and :: parser -> parser -> parser
parse_and p1 p2 l
  = [ ( rest , Node t1 t2 ) | (x,t1) <- p1 l ;
                              (rest,t2) <- p2 x ]
```

Finally, in the case of `And`, we apply the parsers in turn, and build the successful parse trees by gluing together the two component trees into a larger tree.

As is often the case, the function being defined is not itself built by primitive recursion, rather it is a special case of such a function. Such is the case here with the ‘top-level’ parsing function `top_parse`.

```
top_parse :: grammar -> ([char] -> tree)
top_parse gr inl
        = hd outtrees , # outtrees = 1
        = error "error" , otherwise
          where
          outtrees = [ t | (x,t) <- parse gr inl  ; x = "" ]
```

This bears a similarity to Knott's work in [Knott 1987], but there is one major difference. Because he is working in a language in which he cannot introduce a type of grammars in the way we did here, he chose to use a *functional* representation of grammars, defining his generator parser functions and the like *via* continuations. His approach is otherwise equivalent, but we feel that the definition principle is much clearer when seen as *primitive recursion over the appropriate type*.

Two important questions remain.

- Can we use this approach to define structure editors, as we did in 8 above?
- How uniform are our definitions of primitive and general recursion, given in Sections 7 and 5 respectively?

We look at the latter question in the following section, but conclude this section with a discussion of the former.

We cannot easily define a structure editor using the very general definition principles enshrined in `prim_grammar`, for consider the following: Suppose that we are trying to define the structure editor for a grammar of the form

```
gram1 $Or gram2
```

from structure editors for the two halves — given that these cannot "identify themselves", we have no way of indicating at this level what the choices for the user are. All that we can say is something of the form "enter something or something else"! It might be thought that we can get around this by getting the editor to output a description of itself, so that a higher level editor can use that description in its prompting. The unavoidable difficulty with this is that the only appropriate description we can give is the grammar *itself*, which in all the interesting (i.e. *recursive*) cases will be *infinite*!

In the 'tighter' realm which we explored earlier, we see that we can describe structure editors — the price we pay is, of course, some lack of uniformity. We postpone any further discussion until the next section.

12 The grammar type and dependent types

Recall that in Section 3 we showed an equivalence between algebraic types and grammars. Under this analogy, we would expect a parser, or structure editor to produce an object of the particular type in question as its output. This will *not* be the case with the parsers we saw in the last section, which will produce parse trees which *represent* values of some type, rather than produce *values of that type itself*. We call the latter kind of parser *strong* and the former *weak*. The parsers we produce type by type using the `gen_rec` functions will be strong, and so we see that the two constructions we have outlined so far are different. What do we need to do to be able to make our general recursions uniform, or alternatively to make our weak parsers strong? We essentially need to be able to map

```
value :: grammar -> universe_of_types
```

matching each *representation* of a grammar (or algebraic type) with the type that it represents. It is a limitation of the Miranda type system that we cannot do this — such a universal type *is* available in Martin-Löf's type theory [Martin-Löf 1984], and represents one of the fundamental uses of such a construction.

13 Conclusions

In this paper we have shown that the apparent uniformity in a class of structure- or grammar-oriented definitions can be made manifest in two different ways, each of which has its advantages; in the final section we have also noted that we can achieve a fusion of the two if we work within a sufficiently powerful type system.

14 Afterword

The work reported here is unfinished, "in progress". Subsequent authors have developed and described the ideas in a more systematic and compelling way.

Generic programming. One theme is generic functional programming. Disentangling recursion and co-recursion, and building links with category theory [Hinze and Wu 2016] explore foundational questions, whereas focussing on the practical, [Rodriguez et al. 2008] survey the various generic programming extensions for Haskell available in 2008 and [Hinze 2006] shows how generic programs can, in fact, be written in standard Haskell without language extensions. [Garcia et al. 2007] widen the discussion to examine support for generic programming in eight languages including Standard ML, C++, Eiffel and Java as well as Haskell.

Recursion operators. Another approach that focusses on development driven by data is the Bird-Meertens formalism, or 'Squiggol', which builds programs from specifications through equational reasoning; an early introduction is given by [Gibbons 1994]. Later developments built on the different forms of recursion operators termed 'bananas', 'lenses' and so forth [Meijer et al. 1991].

Data and co-data. While Miranda coalesced finite, infinite and partial data into each algebraic data type this has a downside in identifying recursion and co-recursion. This can be seen in our enumeration of all lists of an infinite type: the definitions don't have base cases, and are productive in a technical sense. Turner, in this journal [Turner 2004], made a proposal for "elementary strong" functional programming, that made all functions total ("strong") while eschewing dependent types ("elementary"). While this paper was influential on discussions of the future of functional programming development, it is only relatively recently that strong functional programming languages, such as Agda [Norell 2008] and Idris [Brady 2013], have emerged, albeit in the context of dependent types.

References

[Brady 2013] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5), 2013.

[Burge 1978] W. Burge. *Recursive Programming Techniques*. Addison-Wesley, 1978.

[Fairbairn 1987] Jon Fairbairn. Making form follow function. *Software – Practice and Experience*, 17(6):379–386, 1987. Note: seen as a preprint / technical report at the time of writing.

[Garcia et al. 2007] Ronald Garcia, Jaakko Jarvi, Andrew Lumsdaine, Jeremy Siek and Jeremiah Willcock. An extended comparative study of language support for generic programming. *Journal of Functional Programming*, 17(2), 2007.

[Gibbons 1994] Jeremy Gibbons. An introduction to the Bird-Meertens formalism. *New Zealand Formal Program Development Colloquium Seminar, Hamilton*, 1994.

[Hinze 2006] Ralf Hinze. Generics for the masses. *Journal of Functional Programming*, 16(4-5), 2006.

[Hinze and Wu 2016] Ralf Hinze and Nicolas Wu. Unifying structured recursion schemes: An Extended Study. *Journal of Functional Programming*, 26, 2016.

[Knott 1987] Ron Knott. Declarative Programming. Unpublished book manuscript, 1987.

[Martin-Löf 1984] Per Martin-Löf. Constructive Mathematics and Computer Programming. In C. A. R. Hoare and J. C. Shepherdson, editors, *Mathematical Logic and Programming Languages*. Prentice Hall, 1985.

[Meijer et al. 1991] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire, In *Conference on Functional Programming Languages and Computer Architecture*, Springer, 1991.

[Norell 2008] Ulf Norell. Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming, 2008*, Springer-Verlag, 2009.

[Paulson 1985] Lawrence C. Paulson. Interactive Theorem Proving with Cambridge LCF. Technical Report 80, University of Cambridge Computer Laboratory, November 1985.

[Rodriguez et al. 2008] Alexey Rodriguez, Johan Jeuring, Patrik Jansson, Alex Gerdes, Oleg Kiselyo and Bruno C d S Oliveira. Comparing libraries for generic programming in Haskell. ACM Sigplan Notices, 44(2), ACM, 2008.

[Thompson 1986] Simon J. Thompson. Writing interactive programs in Miranda. Technical Report 40, Computing Laboratory, University of Kent at Canterbury,

Canterbury, Kent, U.K., 1986, Note: this work was published in a more developed form as a chapter of [Turner 1990], the post-proceedings of one of the *Year of Programming* events at the University of Texas, Austin that took place in 1987.

[Thompson 1989] Simon J. Thompson. A logic for Miranda. *Formal Aspects of Computing*, 1(1):339–365, March 1989. Note: this work subsequently published; at the time it was in draft.

[Turner 1985] David A. Turner. Miranda: a non-strict functional language with polymorphic types. In J. P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*. Springer-Verlag, 1985.

[Turner 1990] David A. Turner (ed.). Research Topics in Functional Programming. Addison-Wesley, 1990.

[Turner 2004] David A. Turner. Total Functional Programming. *Journal of Universal Computer Science*, 10(7), 2004.