Laws in Miranda

Simon Thompson

Computing Laboratory,
University of Kent at Canterbury,
Canterbury, Kent, CT2 7NF, U.K.

ABSTRACT

We discuss the Miranda programming language, giving a brief tour of its major features before we narrow down to considering laws in the language. Laws provide a means of implementing non-free data types — two particular classes of examples are constrained data types and data types with memoised information. We discuss these examples and then go on to look at how we can derive laws from function definitions, giving an example which incorporates both of the above ideas. Finally we show how we can prove properties of these 'lawful' types and functions over them.

1. Introduction

Miranda is a functional programming language which introduces the novel feature of laws. These laws are intended to model equationally defined algebraic data types, and are implemented using term rewriting. Our first aim in this paper is to explain the types, using a number of examples, and to examine their implementation.

It should be clear how powerful such a system is; we have the freedom to write all kinds of complex rewriting systems as laws. Our main aim in the paper is to discuss "responsible" ways of using the power of the laws construct. For two classes of example we can **derive** laws from functions defined in Miranda; properties of the functions will transfer to properties of the data types. More generally we show how to discover properties of objects of any 'lawful' type, using **pattern expressions** and deduction rules for such expressions.

We address two other topics briefly. Certain properties are **invariant** under certain laws, and these invariants can be combined with termination properties in a useful way. Finally we look at functions defined over lawful types. We begin, however, with a short introduction to the Miranda language itself.

2. Miranda

Miranda (*) is a functional programming language developed by David Turner [1,2], and a descendant of SASL and KRC, using ideas from HOPE and ML as well.

The basic paradigm of computation is that of calculation using recursion equations. Here we exhibit a function computing `2` to the power `m` .

(*) Miranda is a trademark of Research Software Ltd.

```
exp 0 = 1
exp m = 2 * square (exp (m div 2))
              , m rem 2 ~= 0
        = square (exp (m div 2))
          where
          square p = p * p
```

The first equation matching an expression is used in evaluating the expression. Equations can be executed conditionally, depending on the truth of guards. Local definitions are given in where clauses. As an example we compute `exp 2` . The left hand side of the first equation fails to match with the expression, so we move to the second. A match does take place so we then look at the guard of the first right hand side, `m rem 2 ~= 0` which fails to be true when `m` is `2` . We therefore move to the second right hand side, which is unguarded and rewrite our expression thus:

```
exp 2 = square ( exp 1 )
```

The local definition of `square` and the function definition itself are then used in further rewriting the expression, finally yielding the result.

Distinctive features of the language include

- Functions are 'first-class citizens'.
  Functions can take functions as arguments and return functions as results, so that functions are **higher order** in general. If we write

  ```
  applytwice f x = f(f x)
  ```

  then if `succ x = x+1` ,

  ```
  applytwice succ
  ```

  is the function which adds 2 to its argument.

- Lazy evaluation (a feature not shared by HOPE, ML nor by most LISPs)
  We can construct **infinite** data items, like the infinite list of even numbers:

  ```
  numbers      = numbers' 0
  numbers' n   = n : (numbers' (n+2))
  ```

  A note on syntax: `n : x` is the list, whose first item is `n`, stuck on the front of the list `x` . (In other words ':' is infix `cons`). We use `[2,3,4]` as shorthand for

  ```
  2:(3:(4:[]))
  ```

  As we can see from the following definition, we can use **pattern matching** in function definitions:

  ```
  head (a:x) = a
  ```

  What happens if we evaluate `head numbers` ? This is a function application, so in strict languages we would expect `numbers` to be evaluated fully before being passed to `head` . As this leads to an infinite computation, we must expect non-termination from our evaluation. In a lazy context `numbers` is passed unevaluated, and is only evaluated so far as is needed by the `head` function. In this case we have

  ```
  numbers = numbers' 0
          = 0 : (numbers' 2)
  ```

  and in the final term we have enough information to conclude that

  ```
  head numbers = 0
  ```

- Strong typing:
  Type checking of a Miranda script (list of equations and type definitions) is performed at compile time. The specific type system used is the Milner system [10] which allows generic function definitions, or 'polymorphic functions'. Consider the example

```
length [ ]   = 0
length (a:x) = 1 + (length x)
```

What is most remarkable is the generality of the definition. We don't need to know the type of objects in the list (lists are homogeneous in Miranda) in order to define length. In fact we can state (or the compiler can deduce) that

```
length ::  [*]  ->  num
```

That is, `length` is a function (`->`) from lists of some arbitrary type (`[*]`) to numbers (`num`).

- ZF expressions: These set expressions embody the generate and test paradigm — details can be found in [2].
- User-defined types

  ```
  tree * ::= Nil      |
             Node * (tree *) (tree*)
  ```

  a `(tree *)` is either `Nil` or is a `Node` consisting of an entry of type * and the left and right children (subtrees). This can be compared with a variant record type in Pascal.

  These types are called algebraic. There is also a type **abstraction** mechanism available — we don't discuss this here, but it is worth bearing in mind the impact the construction we are about to describe has on the usefulness (or breadth of application) of the abstract type mechanism.

3. Laws

We are allowed to introduce **laws** which constrain an algebraic data type, or cause some kind of **reductions** on the data items. We introduce the construct via some examples.

Ordered lists

```
olist * ::= Onil |
            Ocons * (olist *).

Ocons a (Ocons b x)
   => Ocons b (Ocons a x)     , a>b
```

Ordered sets are implemented by adding the extra rule

```
Oadd a (Oadd b x)
   => Oadd a x                , a=b
```

Integers

```
integer ::= Zero |
            Succ integer |
            Pred integer
Succ (Pred n) => n
Pred (Succ n) => n
```

What do these three type definitions mean? The first example defines the type of ordered lists — these are constructed from the same **constructors** as lists, but will have their elements ordered by the law that is supplied. Any object of the type containing a pair of elements with the larger immediately to the left of the smaller will automatically be rewritten to have the pair reversed. The overall effect of this is to keep the elements of such objects in ascending order. (We shall see below how we can prove that this is indeed the case.) Ordered sets are similar, except that as well as ordering the elements, we automatically remove duplicate elements from the set as well.

We can represent the integers, by allowing two constructor functions `Succ` and `Pred` together with the nullary constructor `Zero` . The only difficulty with this is that we lose uniqueness of representation — `Zero` can also be represented by `Succ ( Pred Zero )` , for instance. The laws given in the type definition ensure that each integer is represented uniquely; integers will have one of the following forms:

```
Zero
Succ ( Succ .. ( Succ Zero ) .. )
Pred ( Pred .. ( Pred Zero ) .. )
```

What mechanism is used to implement the laws? We should first be clear about what we intend such a lawful type to be. For the example of ordered lists we were clear that we wanted an ordered list to be a list of a particular form. This is true in general — types with laws should be thought of as **subtypes** of their associated **free types**, where by associated free type (or AFT) we mean the type with the same constructors and no laws. The members of the subtype will be elements of the AFT to which none of the laws applies; elements in **normal form** in other words.

How should we think of the constructors of our lawful type, and in particular how should we interpret expressions of the form

```
Ocons 3 ( Ocons 2 Onil )
```

Usually expressions just involving constructors are values (that is they are in normal form), but this expression is not a value of type `olist num`, but rather an expression which evaluates to a value of that type. We should therefore think of occurrences of the constructors as occurrences of **functions** `Ocons'` , `Onil'` which are evaluated to produce expressions in the AFT to which none of the rules apply. What are the evaluation rules for the functions `Ocons'` , `Onil'` ? As there is no law rewriting `Onil` we have

```
Onil' = Onil
```

and similarly

```
Ocons' a Onil = Ocons a Onil
```

We do perform some rewriting on the last kind of expression, so we write the definition

```
Ocons' a (Ocons b x) = Ocons' b (Ocons' a x) , a>b
                     = Ocons  a (Ocons  b x)
```

We can see that for our particular example `Ocons 3 (Ocons 2 Onil)` we first replace each unprimed constructor with its function equivalent, and evaluating the expression in the usual way we find that

```
Ocons' 3 (Ocons' 2 Onil')
    = Ocons' 3 (Ocons' 2 Onil) = ...
    = Ocons  2 (Ocons  3 Onil)
```

so that the final value printed is indeed the ordered version of the original list.

Implicit in our description is a rewriting **strategy**. A strategy is a rule which stipulates which particular sub-expression of a given expression is rewritten when there is a choice of such sub-expressions. Two commonly used strategies are leftmost outermost and innermost leftmost but the Miranda system uses a third one which we might call **lazy innermost**. The strategy is distinctive in that it allows rewriting to be performed on some **infinite** data items, a facility which is not allowed by the two aforementioned strategies, or not at least in their obvious implementations. In many of the examples we explore in this paper, lazy innermost behaves in the same way as innermost first. We now give an example. Suppose that we define

```
x = Ocons 2 x
```

and we try to find its value. This means evaluating the expression `Ocons' 2 x` which, because of the case analysis in the definition of `Ocons'` causes the evaluation of `x` to be initiated recursively, so giving us a non-terminating computation.

We shall return to the effect of laws on infinite data items below. Implicit in the next three sections is the assumption that we are investigating the effect of laws over **strict** data types, that is data types whose constructors are strict. These are the traditional free algebras often known as abstract data types. These can be thought of as sub-types of the corresponding lazy types, if the reader is so inclined.

Finally we should say something about functions defined over lawful types. For example, we define

```
ohead (Ocons a x) = a
```

What will be the value of

```
ohead (Ocons 3 (Ocons 2 Onil))   ?
```

Note that in an evaluation we replace constructors with their primed equivalents, and that the function definition of `ohead` involves unprimed constructors. There is therefore no confusion about how evaluation is to proceed, as `ohead` cannot perform any successful pattern matching until the argument has been reduced to normal form, or has suffered some significant reduction, at least.

```
ohead (Ocons' 3 (Ocons' 2 Onil'))
= ohead (Ocons' 3 (Ocons' 2 Onil))
= ...
= ohead (Ocons  2 (Ocons  3 Onil))
= 2
```

so that we see that `ohead` has indeed produced the head of an ordered list, its smallest element.

4. Memoising function values in data.

Functional programs are traditionally thought of as being inefficient. One well-known source of such inefficiencies lies in the recomputation of previously computed values during the evaluation of an expression. Memoising is the process whereby these values are stored and re-used, rather than being re-computed. Michie [3] was the first to discuss the topic, and interest has recently been revived by Hughes' paper [5].

The particular flavour of memoisation we discuss here is one in which we store the function value for a particular data item as a part of that item itself — this is the reason that it has been dubbed memoisation 'in the data'. We introduce the technique using an example. The height of a tree is the length of its longest branch. Given the type of trees as above, their height can be computed thus:

```
height Nil = 0
height (Node n t1 t2)
           = (max (height t1)
                  (height t2))+1
```

We memoise the function by adding an extra field to each record. i.e. an extra argument to each constructor, in the following way:

```
mtree * ::= Mnil num |
            Mnode * (mtree *) (mtree*) num

Mnil n  =>  Mnil 0            , n ~= 0
Mnode x t1 t2 n
        =>  Mnode x t1 t2 m   , n ~= m
              where
              m = max (height t1)
                      (height t2) +1

height (Mnil n) = n
height (Mnode x t1 t2 n) = n
```

The laws rewrite the data items so that the value in the final field of each record is the correct height value. Two observations should be made. The `height` function is now calculated by looking up the value in the appropriate field, rather than explicitly. If we once evaluate the height of a tree, this value will be stored in the appropriate field, and re-evaluation can be performed simply by extracting that value. Secondly, because of the way that laws are implemented, we have to include the guards so that no expression rewrites to itself, a situation which results in an infinite loop.

In [12] we present the example of memoising the length of a list in the list itself. The reader might like to consult that paper or try for him- or herself a sample evaluation showing that the value is indeed memoised and can be retrieved at little cost subsequently.

The transformations we have described in this section are ones which can easily be mechanised. If we have any function defined by **primitive recursion** over an algebraic type, then using the technique of this section we can memoise its calculation.

We can think of the work in this section as showing how to incorporate functions `f :: t -> t'` computing attributes of data items into the data items themselves. As we observed above, we turn the type `t` into a subtype of the product of `t` and `t'`. These laws do not affect the nature of the data themselves, they only augment the data with additional information. In the next section we look at functions and laws which modify data objects themselves.

5. Laws modifying data items

Under what circumstances can we incorporate functions `f :: t -> t` into the definition of the type `t` itself? It is only sensible to add laws of the form `x => fx` if we know that the sequence of reductions under the law will terminate. In other words we need the sequence `x, fx, f(fx), ...` to be eventually constant. A particular case of this is when $f^2=f$, that is when the function `f` is idempotent; for instance, sorting of lists and balancing of trees are both idempotent operations. A more general example is provided by the `shuffle` function, which is not idempotent, but which does reach a fixed point on every (finite) input list.

```
shuffle [ ] = [ ]
shuffle [a] = [a]
shuffle (a:b:x) = a : (shuffle (b:x))    , a<=b
                = b : (shuffle (a:x))
```

The existence of such fixed points on every list is the basis of the bubble sort algorithm.

Now we show how we can derive the corresponding laws from the function definitions. What advantages has this?

- We can deduce that objects of the non-free type are in the range of the function. Any property of objects in the range will be inherited by objects of the non-free type.
- In more complex cases, it is by no means obvious how we code the laws for the particular functions. Our first attempts to write laws performing tree balancing failed — we produced correct rules first time when we used a derivation technique. This example is discussed in the next section.

We consider an example, that of insertion sort over `olist *`

```
olist * ::= Onil |
        Ocons * (olist *)

sort Onil          = Onil
sort (Ocons a x)   = insert a (sort x)

insert a Onil  = Ocons a Onil
insert a (Ocons b y) = Ocons a (Ocons b y)
                  , a<=b
                  = Ocons b (insert a x)
```

We want to find (terminating) rules which ensure that

```
sort l = l     (C)
```

for every `olist l`. In order to do this we use symbolic rewriting of the definitions along the lines of Darlington's folding and unfolding transformations [12].

Now `sort Onil = Onil`, so `(C)` is satisfied trivially in the case that `l=Onil`, and so we we don't have any rewrite rule with left hand side `Onil`.

The other variant of `Olist` is `Ocons a x`.

```
sort (Ocons a x) =
      insert a (sort x)
```

in order for any more rewriting to occur we need to do a case analysis on `x`. If `x` is `Onil` we find

```
sort (Ocons a Onil) = Ocons a Onil
```

so again if `l = Ocons a Onil` we are in a case where no rewriting is needed. Now we try `x = Ocons b y`.

```
sort (Ocons a x)
 =   insert a (sort (Ocons b y))
```

Now we assume that the rules have been doing some work already: by structural induction we assume that

```
sort (Ocons b y) = Ocons b y
```

so

```
sort (Ocons a x) =
     insert a (Ocons b y).
```

Using the last clause for insert, we have 2 cases. If `a<=b`,

```
sort (Ocons a x)
= Ocons a (Ocons b y)
= Ocons a x
```

so no rewriting is performed in this case. If `a>b`

```
sort (Ocons a x)
= Ocons b (insert a y)
```

we assume `y` is sorted, by induction again

```
= Ocons b (insert a (sort y))
= Ocons b (sort (Ocons a y))
```

Finally we assume `Ocons a y` is sorted, so

```
= Ocons b (Ocons a y)
```

Since `a>b` this is a non-trivial rewriting, that is its left-hand side is different from its right, and we are left with the rule

```
Ocons a (Ocons b x)
   => Ocons b (Ocons a x)      , a>b
```

Remarks:
1. We have assumed a lazy innermost (or innermost) reduction strategy in this derivation.
2. We can apply the same derivation method to the `shuffle` function defined above — the result is the same rewriting rule as we derived above. 3. This technique will not always work, in the sense that we might not achieve termination. This is most likely to happen when the guards in the function involve elements of the data type itself. This will (or could) lead to laws which are guarded by expressions of that form. It is very easy to generate non-terminating rewriting systems in that way. We would suggest the heuristic that:

No guard on a law should involve evaluation of an element of the type itself.

We shall apply this stipulation again when we discuss verification.
One example which the reader might like to try is the example of the function which removes all occurrences but the first of every element in a list. This function can be thought of as a "`makeset`" operation. We can show that there is no natural set of laws which implements this. Indeed we can show that there is no set of laws which removes all duplicates from a list when the only operation we are allowed to perform on the elements is equality checking. The techniques we use in the proof are introduced in section 8 below. Note that we showed (in the second of our introductory examples above) that we could remove all duplicates if we used an ordering relation on the elements.
3. The assumption we made about strictness has to be used here to justify our induction proof.

6. Tree Balancing — A Longer Example

In this section we exhibit a larger example, that of the data type of balanced trees. We say that a tree is **balanced** if each sub tree is balanced, and that the difference in height of the left and right sub-trees is one at most. (These trees are sometimes called AVL trees.) We write reduction rules which ensure that a data item is kept in balanced form. In order to judge when balancing rules need to be applied we have to calculate the height of the trees in question — we use our **memoising** technique to avoid needless re-computation of these values. Thus our example is of both the types discussed in the previous sections.
In our first attempts to formulate the laws for tree balancing we aimed to write the laws directly. These attempts failed to produce terminating rules. We used the transformation technique outlined in the last section to produce the following set. We would suggest that such a procedure should be followed in an example of any size, as it seems to provide the only reliable means of producing suitable rules in such cases.

```
tree * ::= Nil num |
      Node * (tree *) (tree *) num

Nil n => Nil 0 , n~=0
Node n tl t2 h => Node n tl t2 k , h~=k
         where
         k = max [ height tl , height t2 ] + 1

Node n tl t2 h
    => (Node m (Node n tl s3 0) s4 0)
              , ht2 > (htl + 1) & hs4 > hs3
    => (Node l sl (Node n s2 t2 0) 0)
              , htl > (ht2 + 1) & hsl > hs2

    => (Node o (Node n tl ql 0) (Node m q2 s4 0) 0)
              , ht2 > (htl + 1)
    => (Node r (Node l sl rl 0) (Node n r2 t2 0) 0)
              , htl > (ht2 + 1)

    where
    Node l sl s2 htl' = tl
    Node m s3 s4 ht2' = t2
    Node o ql q2 hs3' = s3
    Node r rl r2 hs2' = s2

    htl = height tl
    ht2 = height t2
    hsl = height sl
    hs2 = height s2
    hs3 = height s3
    hs4 = height s4

height (Nil n) = n
height (Node x tl t2 n) = n
```

We should explain how we derived the rules. The algorithm itself comes from Knuth's [6].
We wrote a function which balanced objects of the type

```
tree * ::= Nil |
      Node * (tree *) (tree *)
```

according to Knuth's algorithm, using the height function as defined in section 4. We then wrote the laws which allowed values of `height` to be memoised in the final field of the modified type defined above and finally wrote the laws which calculated the function itself.

7. Laws and term rewriting systems

In this section we discuss two general issues of relevance to term rewriting in general, and of interest in our discussion here. A general reference is provided by [7]. We begin the discussion with some definitions. A term of the type is in **normal form** if no rewriting law can be applied to it. We shall write $t_1 t_2$ to mean that $t_1\text{->}^*t_2$ if for some k 1, $t^1, t^2, \ldots, t^k$

$$t_1 = t^1 \text{->} t^2 \text{->} \cdots \text{->} t^k = t_2$$

and $t_1\text{->}^+t_2$ for $t_1\text{->}^*t_2$ in the case that $k>1$. A term t has a normal form if for some t' in normal form, $t\text{->}^*t'$. A system of laws is **Noetherian** if and only if each sequence of reductions terminates (in a normal form) after a finite number of steps.

A system is **Church-Rosser** or **confluent** if and only if for all t_1, t_2 and t_3, if $t_1\text{->}^*t_2$, $t_1\text{->}^*t_3$ there exists t_4 so that

$$t_2\text{->}^*t_4 \quad \text{and} \quad t_3\text{->}^*t_4$$

We say that the system is **locally confluent** if and only if for all t_1, t_2 and t_3, if $t_1\text{->}t_2$, $t_1\text{->}t_3$ there exists t_4 so that

$$t_2\text{->}^*t_4 \quad \text{and} \quad t_3\text{->}^*t_4$$

If a system is locally confluent and Noetherian then it is Church-Rosser [7]. If a system is Church-Rosser then the normal form of a term is unique, if it exists, so that a locally confluent Noetherian (or **sensible**) system is one in which each term has a unique normal form. The consequence of this for Miranda is that in deducing properties of sensible lawful types we do not need to know the reduction strategy used in the implementation.

Each of the rewriting systems we have introduced is Church-Rosser and, over the strict types, Noetherian. It would be unrealistic to expect a system dealing with infinite data items to be Noetherian, as a (completed) normal form may only lie at the end of an infinite derivation.

How can we show that the systems we introduced are sensible? In section 4 we treated primitive recursion over strict types - it is well known that primitive recursion preserves totality, and a proof of this will show that our system is Noetherian. To show the Church-Rosser property, we argue informally thus: at any stage there if two laws can be applied to a term they can be applied in such a way that the application of one does not interfere with the application of the other. This can be made precise and forms the basis of the Knuth-Bendix theorem (cf [7] where a number of general techniques for proving confluence are enumerated).

We can argue that the type `olist *` is sensible as follows. First we show that the system is Noetherian: Note that the length of a term is preserved by the laws, and observe that the elements of the list are also preserved. Showing that a set of laws is Noetherian is rather like showing that a loop in an imperative program terminates; with each term we associate a non-negative expression whose value is decreased each time we perform a rewriting on the term. In this case we choose the expression

$$\sum_{\substack{i<j \\ a_i > a_j}} (a_i - a_j)$$

Assuming that the pair $..a_i\, a_{i+1}..$ are swopped in a rewriting the only change made to the sum is the loss of the contribution from $a_i - a_{i+1}$. As the sum is non-negative we are assured that any sequence of rewritings will terminate. The Church-Rosser property follows if we note that the bag of elements of the list is preserved by the laws. It is easy to see that every list can be rewritten to its ordered form using a succession of swops of adjacent pairs, giving us a common reduct of each pair of reducts of a list. General methods for proving the Noetherian property can be found in [8].

We would like any set of laws to have the Church-Rosser property — any set which relies for its meaning on a particular reduction order is intrinsically unreliable. We did mention reduction order in our discussion of memoising — note that our interest there was in efficiency rather than in semantics. As a general rule we do have to consider implementations when we discuss efficiency. The implementation regime can have another effect in a situation where termination is not guaranteed, even if the Church-Rosser property holds. Termination may be achieved under one strategy and not under another. Again we would suggest that we should try to write systems of laws with the termination property. Such a suggestion seems to preclude rewriting on infinite objects, whereas one of the most pleasant features of Miranda is the ability to deal with such things. One way of defending this proscription is to make a distinction between computation and data: there is no problem in seeing computations as proceeding indefinitely, but that we should see data as having a definite form, which is not infinitely mutable. This is only one person's opinion and might well be changed if useful examples of rewriting on such objects were discovered. One example which springs to mind is that of "real" real numbers, which can be represented by infinite lists of approximations. A system of laws which reduced such a real to some kind of normal form — this would not be a normal form in the usual sense, but rather a head normal form of some kind. Clearly there is work to be done here.

8. Properties of objects of lawful type

If we look at objects of the type of ordered lists

```
olist * ::= Onil |
            Ocons * (olist *)

Ocons a (Ocons b x)
  => Ocons b (Ocons a x)       , a>b
```

the properties of the objects can be deduced from the knowledge that no law can be applied. We know in this case that any match with

```
Ocons a (Ocons b x)
```

in an ordered list implies that `a<=b`. We can write this more formally thus:

```
(Ocons a (Ocons b x) ; a<=b)    (=F say)
```

Our general **pattern formulas** look like

```
(p ; g)
```

Where `p` is a **pattern**, `g` a formula and the free variables of `g` are free in `p`.

From a law of the form

```
p => p' , g
```

we can deduce the pattern formula

(p ; *not* g)

where *not* is the logical negation operator.

We say that a particular object l of type `olist` satisfies `(p;g)` if every possible binding of values to free variables given by a match with a subobject of l satisfies `g`.

For example

```
Oc  7  (Oc  9  Onil)  satisfies  F
Oc  9  (Oc  7  Onil)  does not satisfy  F
```

F expresses sortedness in a succinct and possibly canonical way.

We can deduce pattern formulas from other pattern formulas, in the following ways:

$$\frac{(p;g)\; g => q}{(p;q)} \qquad \frac{}{(p;\mathrm{true})} \qquad \frac{(p;g)\,(p;q)}{(p;g\&q)}$$

$$\frac{(p;g)}{(p[q/x];[q/x])} \qquad \frac{(q;r)}{(p[q/x];r)} \quad (\text{ if } x \textit{ occurs free in } p \text{ })$$

All these rules but one provide means of changing the formula part, rather than the pattern part. The last rule provides an example of a rule which puts a more complicated pattern on the left hand side. If we introduce the special pattern no, which matches with nothing, then every pair of patterns can be unified, that is we can find a pattern which is obtained by substitution from both patterns. We write $u(p_1,p_2)$ for the most general unifier. If S is the substitution then

$$\frac{(p_1;g_1)\,(p_2;g_2)}{(u(p_1,p_2);\, g_1[S]\&g_2[S])}$$

Where $g[S]$ is the result of performing the substitution S on g.

Using the rules we have given, from

```
(Ocons a (Ocons b x)    ; a<=b).
```

we can deduce, for instance, the formula:

```
(Ocons a (Ocons b (Ocons c x))    ; a<=c).
```

The pattern formula approach allows us to express properties of objects in a succinct way — we can rewrite the pattern formulas in more familiar terms as predicates coded as Boolean-valued functions. We do this now, turning the following pattern formula into the boolean function `prop`.

```
(Ocons a (Ocons b x)    ; a<=b)
```

expresses no constraint on `Onil` or on `Ocons a Onil` so we can say

```
    prop (Onil) = true
prop (OCons a Onil) = true.
```

What constraints are applied to

```
Ocons a (Ocons b x)?
```

There is the outermost level at which a match **does** occur, so we require `a<=b` Also we require that the property is true at inner levels: `prop (Ocons b x)` so

```
prop (Ocons a (Ocons b x)) =
        (a<=b) & prop (Ocons b x).
```

This is the traditional definition for sortedness of lists, written in a functional style. Note that our pattern formula allows us to express order without any recursion.

9. Invariance Properties

We can think of the pattern properties we deduced in the last section as **termination** properties — they express the state of affairs in which no laws can be applied or in other words they express the properties of normal forms. What properties does the normal form of a particular expression have? We can establish this by establishing which properties are **invariant** under the laws.

In the example of ordered lists, the bag of members of the list is preserved; in the example of ordered sets, the set of elements is.

Consider one more example:

```
alist ::= Anil |
          Acons num alist

Acons a (Acons b x)
  => Acons (a+b)/2 (Acons (a+b)/2 x)    , a~=b
```

On termination

```
(Acons a (Acons b x)    ; a=b)
```

all elements of the list are equal. However, the **sum** of the elements of the list has been preserved, so putting the 2 facts together, each element of the list has been transformed to the average of the elements of the original list.

10. Properties of functions over lawful objects

The preceding investigations have indicated how we might discover properties of objects of lawful type. How can we find properties of functions defined over lawful types?

As each such function has a analogue (the 'associated function' let us say) defined over the associated free type, then we can expect that for some types, functions and properties, properties of the associated function will transfer to properties of the function itself. In [13] we elucidate these circumstances of so-called **congruence**. We also discuss how to deduce properties in the general case. The basic method is one of structural induction, but we are in a situation in which we must tread carefully in order to avoid constructing logically inconsistent deduction systems. Again, more details are to be found in [13].

11. Conclusion and Acknowledgements

We have shown the general law mechanism in Miranda, and have shown two general paradigms for use: memoising and putting data into standard "reduced" form. In these cases we have shown that we can deduce properties of the objects of the new types from the form of the laws or the definition of the function from which the laws were synthesised. Clearly there are other examples to be found, and questions to be answered about these examples. Of especial interest are useful examples of laws rewriting infinite expressions. One other possible line of exploration which we have not followed here is the general area of conditional rewriting systems, (see [9], for instance), systems which might involve items of the lawful type in guards of laws used in its definition.

Thanks are due to the Theoretical Computer Science Seminar at the University of Kent for listening patiently to an early version of these ideas, to David Turner for explaining the novel features of the lazy innermost strategy and to Judith Farmer for typing the first draft of the paper.

12. References

[1] Turner, D.A. Functional Programs as Executable Specifications, in C.A.R. Hoare (ed) Mathematical Logic and Programming Languages, Prentice Hall, 1985.

[2] Turner, D.A. Miranda: A non-strict functional language with polymorphic types, in J.-P. Jouannaud (ed.) Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201, Springer Verlag, 1985.

[3] Michie, D. 'Memo' functions and machine learning, Nature, No. 218, April 1968.

[4] Turner D.A. A new implementation technique for applicative languages, Software — Practice and Experience, 9, 31-49, 1979.

[5] Hughes, J. Lazy Memo-Functions, in J.-P. Jouannaud (ed.) Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201, Springer Verlag, 1985.

[6] Knuth D.E. The Art of Computer Programming, Vol 3, pp451-454, Addison-Wesley, 1973.

[7] Huet G. and D. C. Oppen, Equations and Rewrite Rules: A Survey, in R. Book (ed.), Formal Languages: Perspectives and Open Problems, Academic Press, 1980.

[8] Dershowitz, N., Orderings for Term-Rewriting Systems, Theoret. Comp. Sci., 17, (1982), 279-301.

[9] Kaplan, S., Conditional Rewrite Rules, Theoret. Comp. Sci., 33, (1984), 175-193.

[10] Milner, R., A Theory of Type Polymorphism in Programming, Jour. Comp. Sys. Sci, 17 (1978).

[11] Darlington, J., Program Transformation, in Darlington, J., Henderson, P. and Turner, D. (eds.) Functional Programming and its Applications, Cambridge U.P. (1982).

[12] Thompson, S.J., Laws in Miranda, Computing Laboratory Report 35, University of Kent at Canterbury, 1985.

[13] Thompson, S.J., Proving properties of functions over lawful types Computing Laboratory Report, University of Kent at Canterbury, to appear 1986.