Science of Computer Programming 13 (1989/90) 181-218
North-Holland

LAWFUL FUNCTIONS AND PROGRAM VERIFICATION IN MIRANDA

Simon THOMPSON
Computing Laboratory, University of Kent at Canterbury, Canterbury, Kent, CT2 7NF, UK

Communicated by J. Darlington
Received September 1987
Revised June 1989

Abstract. Laws in the Miranda programming language provide a means of implementing non-free algebraic types, by means of term rewriting. In this paper we investigate program verification in such a context. Specifically, we look at how to deduce properties of functions over these "lawful" types. After examining the general problem, we look at a particular class of functions, the *faithful* functions. For such functions we are able, in a direct manner, to transfer properties of functions from free types to non-free types. We introduce sufficient model theory to explain these transfer results, and then find characterisations of various classes of faithful functions. Then we investigate an application of this technique to general, unfaithful, situations. In conclusion we survey Wadler's work on views and assess the utility of laws and views.

1. Introduction

The Miranda[1] programming language is purely functional and combines features from a number of earlier such languages. It features lazy (or "demand-driven") evaluation, allows the user to define new types, contains a mechanism for program modularisation and is strongly typed. We survey the language in Section 3—further details can be found in [9].

Program design considerations often dictate that we model certain objects by data items which are kept in some sort of standard, or *normal*, form. Common sense suggests that we try to separate the concerns of

- data manipulation, and
- preserving standard form

as much as possible. This motivates the introduction of a novel language feature: when we introduce an algebraic data type, by giving its constructors, we can specify laws, which are *rewrite rules*, which rewrite expressions involving the constructors. We present a number of examples in Section 4 and in [8].

Functional programming languages are notable for their amenability to formal treatment, in particular to transformation and verification, [1]. How does the

[1] Miranda is a trademark of Research Software Ltd.

introduction of laws affect this important property? In [8] we investigated ways that we could establish properties of data types with laws. We showed that for a number of classes of example we could *derive* laws from function definitions. Specifically, we showed that we could derive the law for the type of ordered lists from a definition of a function performing insertion sort and that the memoisation [6] of function values could be achieved using laws which were derived automatically from the function definition itself. We also saw that there was a general means by which we could derive properties of objects of lawful type, by means of deduction using *pattern expressions*. This is only half the story, unfortunately. Once we have defined our lawful type we shall *use* it and this will involve defining functions over that type—we need to be able to infer properties of such functions. That is the aim of this paper.

In Section 5 we discuss the meaning and implementation of laws and lawful functions: we decide that associated with every lawful type is a free type (the *associated free type* or *AFT*) over which the constructors of the lawful type are *functions*. Functions over the lawful type are interpreted as functions over the AFT.

In the following section we look at the denotational semantics of the types, and on the basis of this show how properties of the lawful type and lawful functions can be expressed. After discussing various methods of proof, we turn, in Section 7 to examining an example. We give a type of ordered sets, implemented by ordered lists without repetitions, and prove the characteristic properties of the cardinality function over this type. We also assess the proof techniques introduced in Section 6 in the light of this example.

Many of the functions which act over lawful types have a special property. Their *related* functions, i.e., the functions which are given by their definitions when interpreted over the AFT with the laws ignored, act *independently* of the laws. For example, the sum of a list of numbers is independent of the ordering of the list. We call such related functions *faithful*, relative to the set of laws under consideration. In Section 8 we prove that certain properties of faithful functions *transfer* from the lawless to the lawful situation. (The mathematical prerequisites for this material appear in Section 2.) Section 9 introduces a necessary and sufficient condition for a function to be faithful, and this is examined further in the section that follows. We give a number of classes of faithful functions in Section 11.

One of the advantages of functional programming can be seen here. The classes we mentioned are characterised as classes of applications of certain *higher-order* functions. In languages without such functions we would have to look rather harder to characterise faithfulness, and perhaps to prove it time and time again for classes of intuitively similar operations.

In Section 12 we show that faithfulness is a useful notion even in a general context. We introduce the idea of a *faithful representative* of an unfaithful function. For example, the minimum function is a faithful representative of the head function over ordered lists. We show how a faithful representative is used in a crucial lemma from the cardinality characterisation theorem of Section 7.

Wadler has proposed a views construct [5, 10] which is related to that of laws. We survey this in Section 13 and assess it in the conclusion, where we also assess laws.

The appendices contain proofs which we felt would interrupt the flow of exposition if they had been left to succeed their statements.

We might conclude that laws in Miranda are a "mixed blessing". They add a certain complexity and also a potential unpredictability if we cannot give them a suitable formal treatment. We hope to have shown that we can still *prove* properties of functions in a lawful situation, and indeed that such proofs may be unnecessary, if a function is faithful.

2. Mathematical preliminaries

In this section we give a brief sketch of the basics of *model theory*, in order to give a formal foundation to a result we prove below. Further background on logic and model theory can be found in [3, 4].

Properties of objects are expressed by propositions in a language, when the objects in question are deemed to be the interpretations of particular symbols. Our concern here will be languages which can be used to assert formal properties of functions. A function symbol in this language can be interpreted in myriad ways. Our concern will be to explore the *link* between two different interpretations. In one example we shall interpret a particular symbol by intersect and intersect* for example.

If f and g are symbols for functions, then the formula

$$\forall x,y,z \,.\, f\ x\ (g\ y\ z) = g\ (f\ x\ y)(f\ x\ z)$$

will be either true or false depending upon how we interpret the symbols f and g. An *interpretation* of a statement is provided by a *structure*, which will consist of

- a domain (or domains) which form the interpretations of the type(s) of objects,
- functions over the domains which will interpret the function symbols of the language.

We can give some examples, for the formula above.

- A structure could consist of the domain of natural numbers, with f interpreted by $*$ and g by $+$. This interpretation makes the formula true, as it asserts that multiplication distributes over addition. If we swop the interpretations, assigning $*$ to g and $+$ to f the formula is *false*—addition does *not* distribute over multiplication.
- Referring forward to the terminology of Section 7, call $\mathscr{A}$ the structure with domain oset, and with f, g interpreted by intersect and concat. Similarly, call $\mathscr{A}^*$ the structure with domain oset′, and with f, g interpreted by intersect* and concat*.

We write "$\models$" for the relation "models" or makes true. For instance,

$$\mathscr{A} \models \forall x,y,z \,.\, f\ x\ (g\ y\ z) = g\ (f\ x\ y)\ (f\ x\ z)$$

but it is not the case that

$$\mathscr{A} \models \forall x,y \,.\, f\ x\ y = x$$

The two structures are not unconnected, of course. There is a function which links the two:

```
norm :: oset' → oset
```

We call a function h :: $\mathscr{A}^* \to \mathscr{A}$ a *homomorphism* if it *respects the operations of the functions*, i.e.,

```
h (intersect* x y) = intersect (h x) (h y)
h (concat* x y) = concat (h x) (h y)
```
(1)

Our logical result concerns the preservation of validity of certain formulas. We call a formula *positive* if it is built from equations using only the connectives $\wedge$, $\vee$, $\forall$ and $\exists$. Such formulas are called positive because it is impossible to use them to express differences between objects, such as the simplest difference, an *in*equality.

How can we read the properties (1)? They have the consequence of preserving equations: Suppose that

```
intersect* x x = concat* y z
```

Now,

```
h (intersect* x x) = h (concat* y z)
```

because h is a function. The left-hand side equals

```
intersect (h x) (h x)
```

and the right-hand side

```
concat (h y) (h z)
```

so that

```
intersect (h x) (h x) = concat (h y) (h z)
```

Suppose that, in fact,

```
∀x . ∃y,z . intersect* x x = concat* y z
```

we therefore have

```
∀x . ∃y,z . intersect (h x) (h x) = concat (h y) (h z)
```

Recall that, in full, this says

```
∀x::oset' . ∃y,z::oset'.
  intersect (h x) (h x) = concat (h y) (h z)
```

and if h is an *onto* or *surjective* function, then,

```
∀x::oset . ∃y,z :: oset . intersect x x = concat y z
```

This shows how a formula may be carried over from the lawless situation ($\mathscr{A}^*$) to the lawful ($\mathscr{A}$)—we can prove a general result in a similar way, using induction over the complexity of positive formulas:

Preservation Theorem. *Suppose that* h :: $\mathscr{A}^* \to \mathscr{A}$ *is an onto homomorphism, and* ϕ *is a positive formula. Then, if* $\mathscr{A}^* \models \phi$ *then* $\mathscr{A} \models \phi$*—onto homomorphisms preserve the truth of positive formulas.*

3. The Miranda language: An overview

Miranda is a functional programming language, in which functions and other objects are defined by (conditional) equations, and in which programs, or *scripts*, are collections of definitions. The evaluation of expressions (which refer to the objects defined in a script) corresponds to program execution in imperative languages.

Consider an example. The value of the function perfnum, from numbers to booleans, is True if and only if its argument is perfect, that is equal to the sum of its proper divisors.

```
perfnum x = False,                ~posint x
          = (sumdivs x = x),      otherwise
            where
            posint x = (x > 0) & integer x
            sumdivs n
              = sum (filter (divs n) [1..n div 2])
            divs n m = (n rem m = 0)
```

The expression in the first clause following the comma, ~posint x, is a *guard* on the clause, the second clause being the default. Definitions local to the equation follow the where. The predefined function filter removes the elements of the list [1..n div 2] which fail to satisfy the predicate divs n. We can see that divs n is itself a function and is passed as a parameter to filter—functions are treated just as other data objects in Miranda, a characteristic property of functional languages. Now consider sum :: [num] → num adding the elements of a list:

```
sum []    = 0
sum (a:x) = a + sum x
```

The equations here contain *patterns* on their left-hand sides. These serve a twofold purpose: they act as guards, and if their condition is satisfied they cause a composite data item to have its components selected by pattern matching with the component variables. If we use [2, 3] as shorthand for 2:(3:[]) (lists are built from the empty list [] using the infix constructor ":") then

```
sum [2,3] = sum (2:[3]) = 2 + sum [3] = etc.
```

This definition is one of a class of similar ones, which involve *folding* an operator into a list, "from the right":

```
foldr op st []     = st
foldr op st (a:x) = op a (foldr op st x)
```

(st is the starting value). sum is foldr (+) 0, if (+) is prefix plus. Concatenation of lists, concat, can be defined similarly:

```
concat x y = foldr (:) y x
```

where (:) is the prefix form of ":". We shall meet these functions again below.

Remark 3.1. We have encountered *two* uses of the symbol "=" in Miranda. In a definition we use "=" to separate the name from the expression with which it is associated, thus:

```
name = expression
```

We also use it as an *operator*, which returns a boolean value. In this guise it appears in boolean expressions and in particular in guards which appear on the right-hand sides of expressions. In the following definition the first use is a *defining* use, and the second a operator:

```
fac n = 1,                  n = 0
      = n * fac (n - 1),    otherwise
```

The value that a boolean expression of the form

```
l = r
```

can take is one of

```
True,  False,  ⊥.
```

We shall encounter a third use of the symbol, as forming a logical predicate, in the informal mathematical *meta*language in which we conduct our discussion. This is not computable equality, as it is a logical axiom that for every x,

```
x = x
```

is true, even if x is given the value ⊥. This is not the case with the boolean operator, which will be undefined on the "undefined" value ⊥.

The equations of a Miranda script are interpreted as logical assertions, so the defining equality can be seen as of this sort. However, this will not always be the case. The one exception is for definitions of functions over lawful types. We say more about this in Section 5 after looking at the details of our interpretation of types with laws.

We hope that no confusion is caused by this multiple use of a single symbol. The careful reader might like to replace all logical assertions of equality with a triple bar, for example, but should have no difficulty in so doing.

4. Algebraic types in Miranda

A *constructor* is a particular kind of function, the effect of which is to form a composite data item from its component parts. A well-known constructor is the (Lisp) cons, which builds the list

```
cons a x
```

from the item a and the list x. cons a x is the list whose first item (or head) is a and whose remainder (or tail) is x. In the Miranda language there is a facility for defining types built using constructors—such types are called *algebraic*. For example, we might define a type of numerical lists thus:

```
numlist ::= Nil |
            Cons num numlist
```

This declares two constructor functions. Nil is a nullary function (or constant), the null list, and Cons has the functionality of the cons discussed above. Note the Miranda syntactic convention that function names being with small letters and that constructors begin with capitals.

In the Miranda notation for types

```
Nil  :: numlist
Cons :: num → numlist → numlist
```

constructors such as Nil and Cons have no *computational* content—they simply stand for themselves. We might like to think of some data items as being maintained in some normal form as the items are constructed, and it is to this end that the *laws* mechanism is a feature of Miranda. While declaring an algebraic type such as

```
olist ::= Onil |
          Ocons num olist
```

the user can specify one or more *laws* which are applied to keep the data in a particular form. In our example we aim to keep the lists *ordered* by writing

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

Instead of the constructor standing for itself, we have added some computational information to it, so that an expression like

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

does not denote a list whose first element is 3; rather it denotes an ordered list, consisting of 2 followed by 3.

Another example is the type poly of *polynomials*:

```
poly ::= Null |
         Term coeff power poly
```

where coeff and power are synonyms for num and poly is a special type of linked list. Each Term node contains the information about a polynomial term. Our laws are

```
Term 0 n p => p
```

Terms with zero coefficient are removed, whilst

```
Term a n (Term b m p)
  => Term (a + b) n p,          n=m
  => Term b m (Term a n p),     n<m
```

ensure (respectively) that terms of the same power are amalgamated, and that the terms are held in descending power order.

A third example is the type of rationals, given by

```
rational ::= Rat num num
```

with the law

```
Rat a b=> error "zero denominator",   b=0
       => Rat (-a) (-b),              b<0
       => Rat a', b',                 g>1
          where
          a' = a div g
          b' = b div g
          g = gcd a b
```

Rationals are reduced to their lowest terms with a positive denominator, and an error message is produced, halting evaluation, by a rational with zero denominator.

A number of other examples, including a type of AVL trees and a mechanism for memoising values of functions "in the data", are given in [8].

The examples show the utility of the construction: assuming that the system "manages" the data type according to the laws, we can, while programming, confine ourselves to the essentials of the algorithms. Looking at the case of the polynomials, addition becomes simple concatenation, whilst multiplication is performed by taking all possible termwise products. The programmer does not have to concern him- or herself with the (re)normalisation of data items after computation.

In concluding this section we should note that a naïve interpretation of laws as equations between data items is inconsistent. Re-examining the type of olists, we can define

```
head (Ocons a x) = a
```

Now, if we interpret the law as an equality, we have

```
3 = head (Ocons 3 (Ocons 2 Onil))
  = head (Ocons 2 (Ocons 3 Onil))
  = 2
```

which is a contradiction. In the next section we state clearly how the lawful types are defined, and why contradictions such as these do not occur.

5. Explaining the laws mechanism

How do we give a formal account of this behaviour, in the context of a functional programming language? We focus on a particular type, that of olists in our discussion, but it should be clear that the techniques we use are applicable to *all* lawful types—we use the example to make the explanation clearer.

We observe that the information contained in the declaration of a lawful type is of two distinct kinds:

(1) Information about how to construct a type is provided if we ignore the laws. We call the type thus defined the *associated free type* or *AFT* and write its declaration in primed form

```
olist' ::= Onil' |
           Ocons' num olist'
```

to reinforce the distinction.

(2) The objects of the type olist will be objects of type olist' which are of a special form. This is because they are formed only by means of the *functions* Onil and Ocons. The laws define these functions:

```
Ocons :: num → olist' → olist'
Ocons a (Ocons' b x)
  = Ocons b (Ocons a x),    a>b
  = Ocons' a (Ocons' b x),  otherwise
Ocons a Onil' = Ocons' a Onil'
```

Ocons is a function which returns an olist' when applied to a number and an olist'. When that list is null, Onil', the result returned is the singleton olist', with member a. When the list is non-null, there is a case analysis. If a>b, the head element of the list argument, the front of the list is rebuilt, using Ocons recursively, with the order of a and b reversed. Otherwise, the result is the olist' with the front elements in the same order. In other words, Ocons is changed to Ocons' *except* when the law should be invoked, in which case the swop of elements takes place, and the conversion is invoked recursively. Since no law is associated with Onil, we have,

```
Onil :: olist'
Onil = Onil'
```

Consider an example, where we use the symbol "⇢" to indicate "is rewritten to by the evaluator"

```
Ocons 3 (Ocons 2 Onil)
⇢ Ocons 3 (Ocons 2 Onil')
⇢ Ocons 3 (Ocons' 2 Onil')
⇢ Ocons 2 (Ocons 3 Onil')
⇢ Ocons 2 (Ocons' 3 Onil')
⇢ Ocons' 2 (Ocons' 3 Onil')
```

Note that in the example that evaluation proceeds in a leftmost-outermost fashion. It may not appear so to do, but observe that the *pattern matching* in the definition of Ocons forces the (at least partial) evaluation of its argument.

Remark 5.1. The explanation above suggests that this mechanism has something in common with an *abstract* type definition. The *implementation* of the type is given by olist', and access to this is provided by the functions Onil and Ocons. Indeed, all the properties of lawful types can be provided by the abstype mechanism, apart from *pattern matching* as used, for example, in the definition of functions over lawful types. According to Turner, this provided one of the original motivations for the work.

To recap, we have explained how to interpret a type definition with laws as a declaration of the associated free type together with functions over that type, which are defined by the laws. This explains how expressions containing occurrences of the lawful constructors (Onil and Ocons) are given values. We should explain how the other use of the constructors is interpreted.

Constructors can appear within patterns on the left-hand sides of definitions such as that of head in Section 4. We have already decided to interpret olists as olist's, and as Ocons is a *function* rather than a constructor, we replace it with the constructor Ocons':

```
head :: olist → num                                    (2)
head (Ocons' a x) = a
```

The reader will see that this has the intended effect if s/he examines

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

The list argument is ordered, and so its head should be 2. In order to evaluate the head of that list, the evaluator has to perform a pattern match on the argument, which gives the reduction

```
→ head (Ocons 3 (Ocons 2 Onil'))
→ ...
→ head (Ocons' 2 (Ocons' 3 Onil'))
→ 2
```

as desired.

We noted in the overview (Section 3) that not every definition could be read as a logical truth, and we gave an example illustrating that at the end of Section 4. We can see by the example above how the implementation avoids the inconsistency: Ocons is treated as a function and not as a constructor, and therefore we do not pattern match against it, thereby avoiding the deduction that

```
head (Ocons 3 (Ocons 2 Onil)) = 3
```

Remark on notation. In Turner's explanation [9] the functions are primed and the constructors of the AFT are unprimed, the *opposite* of our choice here. There are advantages to each, but we felt that our choice was more convenient for our purposes as it leaves unchanged expressions to be evaluated. We apologise for any unintended confusion that the choice may have caused.

Clearly the definition (2) yields a definition of an *extended* function:

```
head' :: olist' → num
head' (Ocons' a x) = a
```

Indeed, head is simply the restriction of head' to the ordered members of olist'. Because of this identity, we will not make a distinction between these two functions, or in general between any function and its primed version. (Note, however, that we will continue to make a distinction between the primed and unprimed versions of *constructors*—the primed version of a constructor is a pure constructor, whereas the unprimed version is a function.)

Finally, we look at the example of the concatenation function on ordered lists. It has the Miranda definition

```
concat :: olist → olist → olist
concat (Ocons a x) y = Ocons a (concat x y)        (3)
concat Onil y = y
```

This will be interpreted

```
concat (Ocons' a x) y = Ocons a (concat x y)
concat Onil' y = y
```

Occurrences of Ocons on the left-hand sides of the equations have been replaced by Ocons', whereas on the right-hand sides calls to Ocons remain. Again, this is as we intend, as the reader might like to convince her/himself that

```
concat (Ocons' 2 (Ocons' 3 Onil')) (Ocons' 1 Onil')
--→ Ocons' 1 (Ocons' 2 (Ocons' 3 Onil'))
```

the result of which is ordered.

Definition 5.2. There is a *related* function, whose behaviour is different in general. We write concat* for this related function, which results from replacing *all* occurrences of Ocons by Ocons', and is the function given by the definition (3) when the law is ignored completely:

```
concat* :: olist' → olist' → olist'
concat* (Ocons' a x) y = Ocons' a (concat* x y)
concat* Onil' y = y
```

Its behaviour is different:

```
concat* (Ocons' 2 (Ocons' 3 Onil')) (Ocons' 1 Onil')
--→ Ocons' 2 (Ocons' 3 (Ocons' 1 Onil'))
```

In the sequel we shall see that in some circumstances a function and its related function, (like concat and concat*), will exhibit similar behaviour, and that this can be exploited for the purposes of program verification. (In some cases the function definitions of f and f* are identical—we shall sometimes drop the star in such circumstances.) We turn to this general topic in Section 8.

6. Semantics and proof

The explanation of types with laws in Section 5 is operational—we have shown how expressions involving the lawful constructors can be given meaning by means of a syntactic transformation. We can give a denotational explanation, if we recast the interpretation of the lawful constructors. Any expression which involves these operators has a primed analogue (in the AFT). The effect of replacing the primed constructors by the unprimed variants and evaluating the resulting expression is given by the function:

```
norm :: olist' → olist'
norm Onil' = Onil'
norm (Ocons' a Onil') = Ocons' a Onil'
norm (Ocons' a (Ocons' b Onil'))
  = Ocons' a (Ocons' c y),          ~(a>c)
  = norm (Ocons' c (Ocons' a y)),   otherwise
    where
    Ocons' c y = norm (Ocons' b x)
```

The effect of the final equation can be rendered thus in English: "First normalise the tail of the expression, gving the expression Ocons' c y. If c is no smaller than a simply cons a onto the front of the result; otherwise, swop c and a and re-normalise the result."

There are standard methods for giving a denotational semantics to lists, [2], and these work equally well for any algebraic type. Given such an interpretation of the AFT we can interpret the lawful type as *the range of the function* norm. We interpret functions over this type as we explained in Section 5. The function norm forms a *retraction mapping* on the domain of lists; this approach is mentioned in a remark in [9].

Given such an interpretation we can see one route by which we can prove properties of objects of the type olist and of functions over this type. To prove a result of the form

```
∀x::olist . P(x)
```

we can instead show

```
∀y::olist' . P(norm y)
```

In exactly the same way,

```
∃x::olist . P(x)
```

is equivalent to

```
∃y::olist' . P(norm y)
```

Algebraic types in Miranda will, in general, contain infinite and partial elements. If we write "⊥" (pronounced "bottom") for the undefined element, then

```
⊥
2:3:⊥
2:⊥:3:⊥
1:2:3:· · ·:n:· · ·
```

will all be members of the type [num]. (The constructor ":" associates to the right.) In a similar way a lawful type can contain infinite and partial objects. If we write

```
elist ::= Enil |
          Econs num elist
Econs a x => x,   odd a
```

then

```
2:4:⊥
2:4:6:· · ·:2*n:· · ·
```

will be elists. (It is to explain the presence of such items that a denotational approach is necessary—it is only such an approach that can explain

```
2:4:6:· · ·:2*n:· · ·
```

as the "infinite normal form" of the infinite list

```
1:2:3:· · ·:n:· · ·.)
```

Can we define a function

```
normal :: olist' → bool
```

which identifies the olists? For each x we require that normal x is either True or False, i.e. non-bottom. By the monotonicity of computable functions, as ⊥ is an olist we must have

```
normal x = True
```

for every x!

We could relax our restriction, and ask only that normal x be non-bottom on finite definite or infinite lists.

A list is *finite definite* if and only if it is terminated by Onil' and not ⊥. The finite definite lists are exactly those for which the following function returns True:

```
findef Onil' = True
findef (Ocons' a x) = findef x
```

Again, since the (interpretation of) a computable function must be *continuous*, the fact that for an infinite list ilist in normal form

$$\text{normal ilist} = \text{True}$$

must be based on a *finite amount of information about* ilist. There will obviously be other, nonnormal, lists which share these properties yet which are not normal, since normality is an *infinitary* property. Roughly it is expressed by

$$\forall n\ \text{ilist!n} <= \text{ilist!}(n+1)$$

where x!m is the mth member of x. If we restrict ourselves to the finite definite members of olist', we *can* derive a normality predicate from the laws:

$$\text{normal Onil'} = \text{True}$$
$$\text{normal (Ocons' a Onil')} = \text{True}$$
$$\text{normal (Ocons' a (Ocons' b x))} = \sim(a>b)\ \&\ \text{normal (Ocons' b x)}$$

Since we are able to identify the finite definite olists in this way we can prove results for such lists in a different manner:

$$\forall x :: \text{findef(olist)} \,.\, P(x)$$

will be true if and only of

$$\forall x :: \text{findef(olist')} \,.\, [(\text{normal } x = \text{True}) \Rightarrow P(x)]$$

We can prove results of the latter form by a straightforward structural induction. We discuss the *pros* and *cons* of the two approaches in the following section. The retraction method (using norm) is more powerful, but in a wide class of cases the predicate approach (using normal) results in simpler proofs.

Remark 6.1. So far our exposition here has been completely general—this remark is not. In the particular case of olists there will be *no* infinite or non-trivial partial objects. We can prove that

$$\forall x::\text{olist'}\ (\text{findef (norm } x) = \text{True} \ \vee\ (\text{norm } x) = \bot)$$

The result is proved by a structural induction over olist', with a subsidiary induction over the number of "out-of-order pairs" in the lists of a particular length. The proof may be found in Appendix A.

Since we can see olist as a subtype of olist', then clearly any result of the form

$$\forall x::\text{olist'} \,.\, P(x)$$

has as an immediate corollary the fact that

$$\forall x::\text{olist} \,.\, P(x)$$

This is a trivial example of a logical *preservation* result. We shall see a more subtle and useful example in Section 8.

7. An example: The cardinality of ordered sets

A lawful type of ordered sets of numbers is given by

```
oset ::= Empty |
         Add num oset
Add a (Add b x) => Add a x,          a=b
                => Add b (Add a x),  a>b
```

We define the cardinality function on ordered sets thus:

```
card :: oset → num
card Empty = 0
card (Add a x) = 1 + card x
```

and the membership function is given by

```
member a Empty = False
member a (Add b x) = True,          a=b
                   = member a x     otherwise
```

We aim to show that, for all x and a,

(1) member a x = False ⇒ card (Add a x) = 1 + card x
(2) member a x = True ⇒ card (Add a x) = card x

The laws give us definitions of the Add function over oset',

```
Add :: num → oset' → oset'
Add a Empty' = Add' a Empty'
Add a (Add' b y) = Add' a (Add' b y),  a<b
                 = Add' b y,           a=b
                 = Add b (Add a y),    a>b
```

the norm function

```
norm :: oset' → oset'
norm Empty' = Empty'
norm (Add' a Empty') = Add' a Empty'
norm (Add' a (Add' b y))
  = Add' a (Add' c z),            a<c
  = Add' c z,                     a=c
  = norm (Add' c (Add' a z)),     a>c
    where
    Add' c z = norm (Add' b y)
```

and the normality predicate:

```
normal :: oset' → bool
normal Empty' = True
normal (Add' a Empty') = True
normal (Add' a (Add' b y)) = (a<b) & normal (Add' b y)
```

Consider the proof of (1). We only expect to prove this for finite definite sets, x, as only such sets have a (finite) cardinality—a proof of this latter fact is left as an exercise for the reader; it follows much the same lines as the finite definiteness proof for ordered lists which is found in Appendix A. Formally, therefore, we want to show that

```
∀x::findef(oset)
    (member a x = False ⇒ card (Add a x) = 1 + card x)
```

The finite definite osets are precisely the images of the finite definite oset's under norm (exercise), so one way to prove the result is to show

```
(A1)  ∀y::findef(oset')
          (member a (norm y) = False ⇒
            card (Add a (norm y)) = 1 + card (norm y))
```

Alternatively, we can use our normality test and show that

```
(B1)  ∀x::findef(oset')
          (normal x = True & member a x = False ⇒
            card (Add a x) = 1 + card x)
```

Which of the two methods should we use? The expressions whose identity we have to prove are simpler in (B1) than (A1), as norm does not appear embedded in the former. This suggests that (B1) might be the easier to demonstrate. More evidence is provided for this hypothesis when we think about how induction proofs of each might proceed. In proving (A1) we would be faced with showing something like

```
card (Add a (norm (Add c y))) = 1 + card (norm (Add c y)))
```

on the basis of

```
card (Add a (norm y)) = 1 + card (norm y))
```

and sundry other propositions. The difficulty in effecting such a proof lies in the fact that norm (Add c y) and norm y will *not* necessarily be related in any simple way (indeed this is precisely the point of the introduction of the *nontrivial* law), and so the induction will not succeed directly. We *can* produce a proof, which contains a subsidiary induction over the lengths of oset' objects, noting that norm preserves length.

On the basis of the discussion above, it seems that to prove results for all *finite definite* objects we are best advised to use the normality function. In cases like those of olist and osets this technique will be sufficient to establish full universal quantifications, as we saw from the result in the appendix.

In a case where there are nontrivial partial and infinite objects, we will need to use the norm function together with a more subtle, "admissible", induction to prove general universal results. In the remainder of this section we give the proofs of (1)

and (2), in the form of

```
(B1)  ∀x::findef(oset')
          (normal x = True & member a x = False ⇒
            card (Add a x) = 1 + card x)
```

and

```
(B2)  ∀x::findef(oset')
          (normal x = True & member a x = True ⇒
            card (Add a x) = card x)
```

Before we look at the details of the proof we define an auxiliary function

```
sta a Empty = True
sta a (Add b x) = (a<b) & sta a x
```

sta is meant to stand for smaller than all. We use this in the proof of (B1). Our proof proceeds by means of a number of lemmas, some of whose proofs depend upon the *faithfulness* properties we discuss subsequently. Although they could all be proved without recourse to such definitions or transfer results, the proofs are simplified by such means.

7.1. The proof of (B1)

We are to prove

```
(normal x = True & member a x = False ⇒
  card (Add a x) = 1 + card x)
```

for all finite definite x of type oset'. The proof proceeds by induction. Take x = Empty' first. The hypotheses of the implication are both true, and the conclusion states

```
card (Add a Empty') = 1 + card Empty'
```

By definition of Add,

```
Add a Empty' = Add' a Empty'
```

and so the left-hand side of the conclusion is

```
card (Add' a Empty') = 1 + card Empty'
```

as required. Now consider the case that

```
x = Add' b y
```

We assume the result for y and also assume the hypotheses of the implication.

```
(1)  normal (Add' b y) = True
(2)  member a (Add' b y) = False
```

By Lemma B.3 from Appendix B, (1) has the consequence that

```
(3)  sta b y = True
```

and by definition of the normality function,

(4) normal y = True

The definition of the member function means that

(5) a ~= b

Now, by (5), we can exclude the equality case from the definition of Add:

```
Add a (Add' b y) = Add' a (Add' b y),   a<b
                 = Add b (Add a y),     a>b
```

In the first case,

```
card (Add a x) = card (Add' a x)
               = 1 + card x
```

as required. In the second case,

```
card (Add a x) = card (Add b (Add a y))
```

which gives, by Lemma B.5,

```
card (Add a x) = card (Add' b (Add a y))
               = 1 + card (Add a y)
```

By (4) and the consequence of (2) that

```
member a y = False
```

we can apply the induction hypothesis to conclude that

```
card (Add a y) = 1 + card y
```

so that

```
card (Add a x) = 2 + card y
```

Now, x = Add' b y and

```
card (Add' b y) = 1 + card y
```

This means that

```
card (Add a x) = 2 + card y
               = 1 + (1 + card y)
               = 1 + card x
```

as we wanted.

7.2. The proof of (B2)

In this subsection we look at the proof of the other of the pair of characteristic (conditional) equations for the cardinality function over oset. In the course of the

proof we will appeal to a lemma whose proof we defer until after our discussion of faithfulness. Recall that (B2) states

```
normal x = True & member a x = True ⇒
  card (Add a x) = card x
```

The proof is by induction over x::oset'. The base case is trivial, since

```
member a Empty' = False
```

Now, we assume the result is true for y and try to plrove it for

```
x = Add' b y
```

To effect the proof we assume the hypotheses of the theorem,

```
(1)  normal (Add' b y) = True
(2)  member a (Add' b y) = True
```

and attempt to prove the conclusion.

From (2) and the definition of the member function, we have either

```
a=b                                                    (4)
```

or

```
member a y = True                                      (5)
```

In the case of (4),

```
Add a x = Add a (Add' b y)
        = Add' b y = x
```

so

```
card (Add a x) = card x
```

Now consider the case (5), and let us examine Add a x.

```
Add a x = Add' a (Add' b y),   a<b
        = Add b (Add a y),     a>b
```

In the first case, we know that, by Lemma B.3,

```
sta b y = True
```

and by Lemma B.4 we have

```
sta a y = True
```

Using Lemma B.6 we get

```
member a y = False
```

in contradiction to (5). Now, finally, we look at the second clause. Our induction allows us to conclude that

```
card (Add a y) = card y
```

If we can conclude that

```
member b (Add a y) = False                                    (6)
```

then we can use (B1) to conclude that

```
card (Add a x) = card (Add b (Add a y))
               = 1 + card (Add a y)
               = 1 + card y
               = card x
```

as required. In order to conclude (6), we use the following lemma:

Lemma 7.1. member b (Add a y) = member b (Add' a y)

We postpone a proof of this until we have discussed the notion of faithfulness which formalises the idea of the operation of a function being *independent* of the laws on a particular type. Intuitively, member is of this sort. The proof appears in Section 11.

8. A preservation result

In the foregoing we saw how results about functions over lawful types could be proved. Functions over such types have analogues over their AFTs—we called these analogues the *related* functions in Section 5. In this section we reintroduce the definition of the related function, before we give a general method by which results about the related function can be transferred to the function itself.

Recall that given a function f over a lawful type, the *related* function, f*, over the AFT is the function whose definition results from replacing every occurrence of a lawful constructor by its primed version (see Definition 5.2). If the definition of a function involves another function using the lawful type, this should be replaced by its starred version too.

The intersection function over ordered sets is given by

```
intersect :: oset → oset → oset
intersect x Empty = Empty
intersect x (Add a y)
   = Add a (intersect x y),    member a x
   = intersect x y,            otherwise
```

The related function is defined by

```
intersect* :: oset' → oset' → oset'
intersect* x Empty' = Empty'
intersect* x (Add' a y)
   = Add' a (intersect* x y),    member* a x
   = intersect* x y,             otherwise
```

Note that the type oset′ is (isomorphic to) the type of numerical lists, and that intersect* is a standard list manipulating function. Amongst its properties is

```
intersect* x (concat* y z)
  = concat* (intersect* x y) (intersect* x z)
```

where the concatenation, or union, function over ordered sets is given by

```
concat :: oset → oset → oset
concat Empty x = x
concat (Add a y) x = Add a (concat y x)
```

Under what circumstances do results about a function like intersect* transfer to the lawful situation? We explore that question presently, but we should first note one of the implications of these "transfer" results.

> Many theorems which we prove for functions over a lawless type will carry over to a lawful types (for which the former is the AFT)—this re-usability is a desirable feature, and is only to be expected, since we will in general expect to re-use functions, and such re-use is underpinned by the transfer of properties.

Before we go any further we look at an important example. The head function over ordered sets has the definition

```
head (Add a x) = a
```

Recall that we explained the action of head by saying that

```
head (Add′ a x) = a
```

which is a definition *identical* to that of head*:

```
head* (Add′ a x) = a
```

However they have rather *different* behaviour, *relative to their respective constructors.* head* acts on the AFT, and simply takes the first element of an olist′, whereas head works over the olist type. This means that it is *not* in general the case that

```
head (Add a x) = a
```

In other words, a function over a lawful type will not necessarily satisfy its defining equations, when interpreted as an equation between expressions. We should not be surprised at this when we remember the evaluation of

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

in Section 5.

Our interest here is in the transfer of properties of the related function to the lawful function, when the properties of the former are expressed relative to the lawless constructor—it is with respect to this constructor that such properties will naturally be expressed, whereas properties of the head function itself tend to be expressed in terms of the lawful constructor Add.

We can now apply the model theory we outlined in Section 2 to give a transfer result.

What is the situation, in general? We have a *possible* homomorphism norm in every situation. We call a function

$$f^* :: t' \to t'$$

faithful if when we form the structures $\mathscr{A}$ and $\mathscr{A}^*$ from it, then norm is a homomorphism between them. So,

Transfer Theorem. *If* f* *is faithful, the positive properties of* f* *carry over to* f.

Every *equation* is positive as it has the form

$$\forall x,y,.. \; f\ x \ldots = g\ y \ldots$$

We can also express the existence of particular values (with positive properties), and using both quantifiers, express the fact that a function is *onto*:

$$\forall x \exists y \,.\, x = f\ y$$

which has the consequence that f has a left inverse.

What does it mean for norm to be a homomorphism between the two structures for f? It means that

the value of the function f is independent of the laws of the lawful type.

Once we observe this it should be clear why properties are preserved: the laws have no effect on the function. We have achieved a *separation of concerns*—the law handles the (re-)normalisation of the data, and we operate on these data in a law-independent way, not concerning ourselves with the details of the normal form. Clearly, not all operations on lawful objects can be of this kind, but many will be.

The result is not simply of formal value. As we remarked above, any positive properties of functions will be carried over if we *re-use* those functions in a lawful context. These library function, such as the list concatenation operator, ++ (which is concat* in fact) will appear in a number of lawful environments.

Before we go on to find a characterisation of faithful behaviour, we should explain the details of the correspondence in the case where types *other* than the lawful type are involved. All we need to do is to explain how the possible homomorphism, norm, is extended to these types. Since we are not concerned with their lawful behaviour (if any), we simply map them to themselves—norm is extended by the identity function.

In the next section we examine the definition of faithfulness further, and aim to find a characterisation of it.

9. Characterising faithfulness

Recall that we call a (one argument) function

$$f^* :: t' \to s$$

faithful if for each x::t',

$$\text{norm } (f^*\ x) = f\ (\text{norm } x) \tag{7}$$

If we write $a \sim b$ for

$$\text{norm } a = \text{norm } b$$

then (7) implies that

$$x \sim y \Rightarrow (f^*\ x) \sim (f^* y) \tag{8}$$

In the case that s is distinct from t, since norm is the identity on s, we have the particular case that

$$x \sim y \Rightarrow (f^*\ x) = (f^*\ y) \tag{9}$$

In fact, (8) is equivalent to (7). We show that (8) implies (7) now.

Proof. Recall that we defined f by

$$f\ (\text{norm } x) = f'\ (\text{norm } x)$$

Now, since f' contains only instances of constructors of the lawful type, and none of the AFT, its results will be normalised, so

$$f'\ (\text{norm } x) = \text{norm } (f'\ (\text{norm } x)) \tag{10}$$

How do f' and f* differ? Only in the constructors that they contain. f* contains the (primed) constructors of the AFT whereas f' contains the constructors of the lawful type. It is not hard to see that for any x and y, if $x \sim y$ then

$$c \ldots x \ldots \sim c' \ldots y \ldots$$

for any constructor c and its primed version. Using an induction proof (specifically a fixed point induction) we have

$$\text{norm } (f'\ x) = \text{norm } (f^*\ x)$$

for every f and x, so

$$\text{norm } (f'\ (\text{norm } x)) = \text{norm } (f^*\ (\text{norm } x))$$

Now, we can use (8), since

$$\text{norm } x \sim x$$

to give

$$\text{norm } (f^*\ (\text{norm } x)) = \text{norm } (f^*\ x)$$

Putting together this chain of equalities, we have

$$f\ (norm\ x) = norm\ (f^*\ x)$$

as required. □

To summarise, we have shown in this section that

$$x \sim y \Rightarrow (f^*\ x) \sim (f^*\ y)$$

(and the special case of

$$x \sim y \Rightarrow (f^*\ x) = (f^*\ y)$$

when the types s and t are distinct) are sufficient conditions for the function f^* to be faithful.

10. Proving faithfulness

We saw in the last section that there is a premium in showing that functions are faithful. In this section we examine a way of characterising the relation $\sim$ in order to find a means by which we can prove particular functions are faithful. In this section we return to looking at our first example, which was of a type olist of ordered lists. We shall make some remarks about ordered sets and the type oset in the following section.

We characterise the equivalence relation $\sim$ in the particular case of ordered lists, but the same method used here will apply in any lawful situation.

Recall the definition of olists in Section 4, and remember that $x \sim y$ is defined by $norm\ x = norm\ y$.

How do we characterise $\sim$? We certainly require that

(R1) $x \sim x$
(R2) $x \sim y \Rightarrow y \sim x$
(R3) $(x \sim y)\ \&\ (y \sim z) \Rightarrow x \sim z$

(where $\Rightarrow$ is the logical implication symbol) which are the three axioms for equivalence relations. We also want closure under expression substitution:

(R4′) $x \sim y \Rightarrow C[x] \sim C[y]$

where C[_] is any context, and finally closure under the law:

(R5′) $a{:}b{:}x \sim b{:}a{:}x$ if $a > b$

Note that we have used an infix version of Ocons′, (:) for brevity, and also to underline the fact that the AFT of olist is simply the type [num].

We can simplify (R4′) and (R5′) somewhat

(R4) $x \sim y \Rightarrow a{:}x \sim a{:}y$
(R5) $a{:}b{:}x \sim b{:}a{:}x$

(R4') simplifies since any context is built by iterating cons, and (R5') simplifies in the presence of the other rules. How do (R1)–(R5) characterise the equivalence relation? The relation ~ is the *smallest* relation satisfying these axioms, an inductive definition [7] in other words. The minimality property can be stated as an induction principle:

Principle of induction for ~. If the relation P has the property that

(I1) P(x, x)
(I2) P(x, y) ⇒ P(y, x)
(I3) P(x, y) & P(y, z) ⇒ P(x, z)
(I4) P(x, y) ⇒ P(a:x, a:y)
(I5) P(a:b:x, b:a:x)

then, for all x and y, x ~ y ⇒ P(x, y)

The principle is simply a restatement of the minimality of ~. In general such a minimal relation need not exist. Consider the smallest equivalence relation such that either 1 and 2 are related or 1 and 3 are related—there are two minimal solutions, but no minimum one. On the other hand, it is easy to see for our example that:

- there is some relation satisfying the axioms: the relation relating everything to everything else;
- given a collection of relations satisfying (R1)–(R5), then their intersection will satisfy (R1)–(R5);

So the smallest such relation is the intersection of the set of all such relations.

The principle of induction is simply a restatement of the definition of ~, as it states that if P satisfies (R1)–(R5) (or (I1)–(I5)) it is a superset of the equivalence relation.

Our aim in proving the faithfulness of the function f is to show that

P(x,y): f x ~ f y

has the properties (I1)–(I5). For *any* function f, P has the properties (I1)–(I3), a simple exercise for the reader, so we should show:

(F1) f x ~ f y ⇒ f (a:x) ~ f (a:y)
(F2) f (a:b:x) ~ f (b:a:x)

The approach outlined in this and the previous sections will generalise to a situation in which we might have infinite objects in our domain of interpretation (the oddlist type, defined above, exemplified this). We say x ~ y if x and y are given the same interpretation (in the semantics) and we can find a similar, but *infinitary*, inductive characterisation of this relation. We find conditions similar to (F1) and (F2) augmented by a clause which requires that the application of f commutes with limits. This clause will not in fact be necessary, as f will be continuous and therefore commute

with limits. Before we close this section we should clarify our definition of faithfulness for a function f :: t' → s, a function which takes an object of AFT as argument and returns an object of unrelated type as result. We require that

$$x \sim y \Rightarrow f\ x = f\ y$$

and the conditions on f to which this leads are

(F1′) $f\ x = f\ y \Rightarrow f\ (a{:}x) = f\ (a{:}y)$
(F2′) $f\ (a{:}b{:}x) = f\ (b{:}a{:}x)$

11. Exhibiting some faithful functions

In this section we give some examples of faithful functions and properties which we can transfer. Consider first sum, defined in Section 2. sum satisfies (F1′) and (F2′), i.e., if sum x = sum y then

$$\begin{aligned} \text{sum}\ (a{:}x) &= a + \text{sum}\ x \\ &= a + \text{sum}\ y \\ &= \text{sum}\ (a{:}y) \end{aligned}$$

and

$$\begin{aligned} \text{sum}\ (a{:}b{:}x) &= a + \text{sum}\ (b{:}x) \\ &= a + (b + \text{sum}\ x) \\ &= (a + b) + \text{sum}\ x \\ &= (b + a) + \text{sum}\ x \\ &= b + (a + \text{sum}\ x) \\ &= b + \text{sum}\ (a{:}x) \\ &= \text{sum}\ (b{:}a{:}x) \end{aligned}$$

Note that the only properties of the operator + which we use are its associativity and commutativity. Now, since that we observed that

$$\text{sum} = \text{foldr}\ (+)\ 0$$

it is not hard to see that the argument above is a special case of the proof of the following Lemma:

Lemma 11.1. *If* op :: * → * → * *is commutative and associative, then*

foldr op st

is faithful for any value st.

Corollary 11.2. *If we define* product *and* min *by*

product = foldr (*) 1
min = foldr min_pair infinity

where infinity *is an "imaginary" greatest integer, then they are both faithful.*

We can push the result slightly further. An operator

```
op :: * → ** → **
```

is *left commutative* if

```
op a (op b c) = op b (op a c)
```

for all a, b, c. By a proof similar to the above, we have:

Lemma 11.3. *If* op *is left commutative, then* foldr op st *is faithful for every* st.

Corollary 11.4. *The constant concatenation functions* concat_x *defined*

```
concat_x = foldr (:) x
```

are faithful.

The corollary is proved by the analogue of the lemma for the conditions (F1) and (F2)—we weaken the requirements ot left commutativity *up to equivalence of normal form.*

We can prove that concat itself is faithful, since (:) satisfies the hypothesis of the following lemma.

Lemma 11.5. *If* f *is faithful in its second argument, then* foldr f *is faithful in its first argument.*

We can now show an example. The equation

```
sum x + sum y = sum (concat x y)
```

which involves faithful functions, will be a theorem for olists as well as for lists (the AFT of olist).

There are a number of other general results, some of which concern the full primitive recursion operator on lists, of which foldr is a special case. We give one last result for foldr here. Observe that

```
(foldr g st) . (map h) = foldr f st
```

where f x y = g (h x) y and map is the higher-order function which applies a function to all the members of a list. Now, if g is left commutative, then so is f, irrespective of h (an exercise for the reader). This implies:

Lemma 11.6. *If* g *is left commutative, then*

```
(foldr g st) . (map h)
```

is faithful for any h *and* st.

One example of such a function is

```
alleven = (foldr (&) True) . (map even)
```

Clearly there are similar general results for other types, like the type of ordered sets, defined in Section 7, the polynomials defined above, etc. We note that the explicit use of higher-order operations in function definitions contributes not only to their comprehensibility, but also allows us to infer properties of functions more easily. If we had kept our explicit recursive definitions of sum, product, etc., we would need to prove a new theorem on faithfulness for each function—the general result for foldr does the work once and for all.

The operation foldr embodies primitive recursion over lists, and so we have shown how to verify faithfulness for a wide class of functions. For each algebraic data type there is a corresponding "recursor", and we can prove analogous results for these operators. Take, for instance, the type of ordered sets, which we saw in Section 7.

The function foldr op st will be faithful for *ordered sets* if op is left commutative and *left idempotent*, i.e.,

```
op a (op a c) = op a c
```

for all a and c.

The maximum and minimum functions on sets are defined by folding left commutative and left idempotent operators into the list. We can also show that a suitable definition of the member function is faithful by such a method:

We can see that member is defined by the following equation (or equivalently satisfies the equation):

```
member a = (foldr (∨) False) . (map ((=) a))
```

Now, ∨ is associative, commutative and idempotent, so that it is left commutative (by the first two of these) and left idempotent (by the first and the last). This means that by the analogue of Lemma 11.6, the function itself is faithful.

Note that in our use of this property in Lemma 7.1, we require that

```
member a (Add b y) = member a (Add' b y)
```

This is a consequence of faithfulness, since

```
member b z = member b (norm z)
```

for any b and z, and so the two applications above have the same value.

12. Faithful representatives of functions

In Section 7 we saw that not every function over a lawful type would have a faithful related function. The example we have discussed at some length is the head

function on ordered sets of lists. Even if head itself is not faithful, we can sometimes find a faithful function which represents it. We have already remarked that the min functions on lists is faithful—moreover it *agrees with the* head *function on normal forms*:

```
normal x = True ⇒ min x = head x
```

a fact which it is easy to prove by induction. We can then use the fact that this function is faithful in proofs of properties of head. We do this in proving the implication of Lemma B.1 (Appendix B):

Lemma 12.1

```
normal y = True ⇒ normal (Add a y) = True
```

Proof. The proof is by induction on y. The result obviously holds for y = Empty', so consider the case of y = Add' b z where we assume the result for z. We also assume the hypothesis of the implication, that is the normality of y.

```
Add a (Add' b z) = Add' a (Add' b z),   a<b
                 = Add' b z,            a=b
                 = Add b (Add a z),     a>b
```

The results of the first two clauses are obviously normal. Since we assumed that y was normal, we have

```
normal z = True
```

so by our induction hypothesis,

```
normal (Add a z) = True                    (11)
```

In order to show that

```
normal (Add b (Add a z)) = True
```

we need (11) and to be able to conclude that

```
b < head (Add a z)                         (12)
```

Since we know that Add a z is normal, we can replace head by its faithful representative, min, and so try to prove

```
b < min (Add a z)                          (13)
```

Now, since min is faithful,

```
min (Add a z) = bi_min a (min z)
```

We know that b<a, by assumption, and that b < min z, since we have assumed the normality of Add' b z. This allows us to conclude (13) and so (12), as required to complete the proof. □

Every function will have a faithful representative, given by the composition

```
f . norm
```

and so, in principle at least, we can apply these methods in any lawful situation.

13. Views

In this section we look at related work on views, particularly from the point of view of program verification. Views were described by Wadler, first in [10] and then later in the Haskell draft standard, [5, Section 5.1.4]. Views are intended to allow pattern matching and data abstraction to "cohabit". This is achieved by introducing a free algebraic type, the *viewing* or *view* type, terms of which describe objects of another type, the *viewed* type. The two types are related by the functions toView and fromView which go to the view type from the viewed type and vice versa, and which are intended to relate the objects of the viewed type to the images by which they are accessed. Observe that in the earlier, but more expansive, [10], these functions were called in and out.

There are also other differences between the two expositions, in particular about the circumstances in which a view is legitimate. Specifically, we need to state the condition under which an expression in the view type can be said to denote a unique member of the viewed type. Wadler [10] calls for toView and fromView to provide an isomorphism between (subsets of) the types, whereas this is relaxed but made more rigorous in the later definition [5]. (In our account of the definition we omit type variables—they only add to the notational overhead.)

Suppose the view declaration takes the form

```
view T = c1 T11. .T1k1 |
         . .           |
         cn. .
where
  fromView = . . .
  toView = . . .
```

then expressions constructed using operations ci are to denote objects of type T. Associated with the definition is a free type

```
View = c1' T11. .T1k1 |
       . .            |
       cn'. .
```

and the functions fromView and toView can then be viewed as ordinary functions

```
fromView :: View → T
toView :: T → View
```

An example is given by

```
view (num, num) = Ratio num num
where
  fromView (Ratio n d) = (n, d)
  toView (n, d) = error "zero denominator",   d=0
                = toView (-n, -d),            d<0
                = Ratio (n div g) (d div g),  otherwise
                  where
                    g = gcd n d
```

and indeed this is a possible implementation of rationals as described in the Haskell report [5, Section 6.4.3]). In this case, we have a type

```
View = Ratio' num num
where
  fromView (Ratio' n d) = (n, d)
  toView (n, d) = error "zero denominator",   d=0
                = toView (-n, -d),            d<0
                = Ratio' (n div g) (d div g), otherwise
                  where
                    g = gcd n d
```

We can now state the condition for legitimacy of a view expression. Expressions of the form

```
ci x1..xin      Ratio 34 8
```

are intended to denote expressions of the type T (respectively (num,num)). The elements are defined to be

```
fromView (ci' x1 .. xin)      fromView (Ratio' 34 8)
```

if and only if the condition

```
ci' x1..xin = toView (fromView (ci' x1..xin))                    (14)
```

is met. If (14) fails, then the expression is *undefined.* For many examples this condition is met—in particular for the majority of Wadler's examples in [10] the condition is obviously true. It is not always the case, however. For instance take the example of Ratio 34 8.

```
toView (fromView (Ratio' 34 8))
  = toView (34, 8)
  = Ratio' 17 4
```

So condition (14) fails here. A similar failure occurs if we define the obvious view analogue of ordered lists, thus:

```
view [num] = Onil |
             Ocons num [num]
where
  toView [] = Onil
  toView (a:x) = insert a (toView x)
                 where
                 insert a Onil = Ocons a Onil
                 insert a (Ocons b y)
                   = Ocons a (Ocons b y),   a≤b
                   = Ocons b (insert a y),  otherwise
  fromView Onil = []
  fromView (Ocons a x) = a:x
```

Checking the condition for

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

we find that

```
toView (fromView (Ocons' 3 (Ocons' 2 Onil')))
  = toView [3, 2]
  = insert 3 (Ocons' 2 Onil')
  = Ocons' 2 (Ocons' 3 Onil')
```

and so all such *unordered* expressions using the constructor Ocons are formally undefined. This contrasts with the lawful treatment under which such an expression is taken to denote the ordered list with elements 2 and 3. This aspect of views has consequences both for implementation and for program verification.

We usually interpret a definition of the form

```
head (Ocons a x) = a
```

as universally valid, or at least we do so if the constructor Ocons is *lazy*, as is the case in Miranda and Haskell. If this is a definition over the view type, then we can no longer see it as universally valid, for if so, we have

```
3 = head (Ocons 3 (Ocons 1 Onil))
  = head ⊥
  = head (Ocons 2 (Ocons 1 Onil))
  = 2
```

since all unordered lists built using the constructor Ocons are undefined. This echoes the situation for laws where we saw that such an equation was no longer universally valid, only its analogue over the associated free type was.

This situation also has something in common with a type of lists in which the constructors are *strict*. Again, in such a case we must verify the well-formedness of an argument before pattern matching against it.

13.1. Implementation

Wadler discusses this briefly in the POPL paper, [10], but we need to look in more detail at condition (14) also. Some values created using the constructors will be undefined, and the criterion for definedness uses an equality check. In terms of implementation we shall have to use the equality operation, at run-time, to verify particular expressions are defined. One consequence of this is that if the objects we are comparing are infinite or partial then the equality operation will not return a result. In turn this implies that we will be unable to give views of infinite or partial objects, if constraint (14) is to be checked at run time.

There is an alternative to run time checking, and that is to use a sophisticated compiler or a theorem prover to verify the condition (14) *statically*.

13.2. Verification

We have already seen one effect of the introduction of views: we cannot treat defining equations simply as universally quantified equalities; we must check the well-formedness of expressions before applying equations.

Wadler points to another difficulty in [10, Section 10]: defining equations must meet a *homomorphism* condition before they may be used. This is crucial since otherwise the logic becomes inconsistent. This is in contrast with the mechanism of laws, as examined earlier in the paper. We showed there that even in the difficult situation of non-faithful functions we are able to perform program verification, and in the happier situation of faithfulness we are able to transfer results from the lawless to the lawful domian.

The constraint mentioned by Wadler must presumably be verified before programs are executed. Again this will need support from a theorem proving system or a very "smart" compiler.

14. Conclusion

We have shown that the general laws mechanism is a very powerful one, and must be used with care. However, we have seen that the notion of *faithfulness* is a natural one, and in many situations we shall be able to use that criterion as a justification for the transfer of logical results from the lawless to the lawful type.

Even when we look at a function which is not faithful, we find that our proofs can be aided by choosing *faithful representatives* of general functions—this notion was suggested by the proof which we saw in Section 12, in fact.

The discipline suggested by the work we have discussed seems to be one that of "use faithful functions as much as possible"—these have the twin advantages of being independent of the laws (which is an aim of a software engineer keen to separate concerns as much as possible) and of carrying proof-theoretic information from the lawless type.

As Wadler mentions in [10], we can think of laws as providing a view of an algebraic data type, whereas the general view mechanism allows a view of any type. On the other hand, as we saw in the previous section, views are unsuitable for maintaining data in a normal form, one of the intended uses of laws which we discussed in the introduction. The two features are therefore complementary and irredundant.

As can be seen from the examples examined here and in [8, 10] both mechanisms have unforeseen features. Neither is a fundamental feature of a (lazy) functional programming language, yet we would argue that, especially in the case of laws, the techniques outlined here are ones which provide for their disciplined use.

Appendix A. The finite definiteness of olists

In this section we give the proof of

```
∀x::olist' ( findef (norm x) = True ∨ norm x = ⊥ )
```

The proof proceeds by *induction*. The body of the universal statement is syntactically directedly complete [2], and so the universal statement will follow from the result for ⊥, Onil' and the induction step. The proof is obvious in the first two cases so we aim to show that

```
( findef (norm (Ocons' a x)) = True ∨
        norm (Ocons' a x) = ⊥ )
```

on the basis of

```
( findef (norm x) = True ∨ norm x = ⊥ )
```

The cases that x is either ⊥ or Onil' are obvious, so we assume that x = Ocons' b y for some b and y.

```
norm (Ocons' a x)
  = Ocons' a (Ocons' c z),          ~(a>c)
  = norm (Ocons' c (Ocons' a z),    otherwise
    where
    Ocons' c z = norm (Ocons' b y)
```

Now, if norm x = ⊥ then c will be ⊥ and so the guard will fail, giving

```
norm (Ocons' a x) = ⊥
```

Similarly, if a is ⊥, the guard will fail. Now, suppose that norm x ≠ ⊥ and the guard is defined, then

```
findef (norm x) = True
```

with the consequence that

```
findef z = True
```

This means that, in turn,

```
findef (Ocons' a z) = True
```

If the guard is True then we have

```
findef (Ocons' a (Ocons' c z)) = True
```

which implies the result. On the other hand, if the guard is False then we need to conclude that

```
findef (norm (Ocons' c (Ocons' a z))) = True
```

For similar reasons to those above we have

```
findef (Ocons' c (Ocons' a z)) = True
```

If we define the number of *crossing points* of a list x to be the number of pairs (i, j) with i<j and

```
x!i > x!j
```

then by an induction over the number of crossing points in lists of the same length as this list we can conclude the desired result. The reduction in the number of crossing points is due to the fact that the list is making progress to becoming ordered under the action of the normalising laws.

This completes the proof.

Appendix B. Lemmas from the cardinality theorem

Lemma B.1.

```
normal y = True ⇒ normal (Add a y) = True
```

Proof. See proof of Lemma 12.1. □

Lemma B.2.

```
normal x = True ∧ sta b x = True ⇒
  Add b x = Add' b x
```

Proof. We consider two cases. If x = Empty', then

```
Add b Empty' = Add' b Empty'
```

by definition. Suppose that x = Add' a y, then

```
sta b x = True
```

implies that b<a, so

```
Add b x = Add b (Add' a y)
        = Add' b (Add' a y)
        = Add' b x
```

completing the proof. □

Lemma B.3.

```
normal (Add' b x) = True ⇒ sta b x = True
```

Proof. The proof is by induction on x. If x = Empty', then the conclusion of the implication is true. Suppose instead that x takes the form Add' a z and that the result is true for z. Assume that

```
normal (Add' b (Add' a z)) = True
```

This implies that

```
normal (Add' a z) = True                                    (B.1)
```

and that b<a. From the induction hypothesis and (B.1) we have

```
sta a z = True
```

The following lemma allows us to conclude that

```
sta b z = True
```

and combining this with b<a gives

```
sta b x = sta b (Add' a z) = True
```

giving us the desired result. □

Lemma B.4.

```
b<c ∧ sta c x = True ⇒ sta b x = True
```

Proof. The proof is a simple induction over x, using the transitivity of the "less than" relation. □

Lemma B.5.

```
normal (Add' b x) = True ∧ b<a ⇒
  Add b (Add a x) = Add' b (Add a x)
```

Proof. Lemma B.2 provides sufficient conditions for the consequent of the implication to be true. We require that, under the assumption of the hypotheses of the implication,

```
normal (Add a x) = True                    (B.2)
sta b (Add a x) = True                     (B.3)
```

Now, by the first hypothesis, we can deduce that

```
normal x = True
```

and so by Lemma B.1 we have (B.2). In order to deduce (B.3) we use that fact that sta* is a faithful function, together with the identity of sta and sta* to conclude that

```
sta b (Add a x) = sta b (Add' a x)
                = (b<a) & sta b x
                = sta b x
```

since (b<a) = True. The first hypothesis implies by Lemma B.3 that sta b x = True and so we have (B.3), completing the proof. □

Lemma B.6.

```
sta b y = True ⇒ member b y = False
```

Proof. Both the hypothesis and conclusion are true in the case that x = Empty'. Assume the result for z—we aim to prove it for Add' c z, on the assumption that

```
sta b (Add' c z) = True
```

This means that

```
sta b z
```

and b<c. By induction the former gives us

```
member b z = False
```

and since b<c implies that (b=c) = False we have the rsult we wanted, by induction. □

Acknowledgement

Thanks are due to Drew Adams, Richard Kennaway and David Turner for interesting comments on an earlier version of this paper, and to British Petroleum plc and the Alvey directorate for partial funding of the research discussed here.

References

[1] J. Backus, Can programming be liberated from the von Neumann style? A functional style and its algebra of programs, *Comm. ACM* **21** (8) (1978).

[2] R. Cartwright and J. Donahue, The semantics of lazy (and industrious) evaluation, Tech. Rept. CSL-83-9, Xerox PARC, Palo Alto, CA (1984).

[3] C.C. Chang and H.J. Keisler, *Model Theory*, Studies in Logic and the Foundations of Mathematics **73** (North-Holland, Amsterdam, 2nd ed., 1977).

[4] W. Hodges, *Logic* (Penguin, Harmondsworth, 1977).

[5] P. Hudak and P. Wadler, Report on the functional programming language Haskell (1988).

[6] D. Michie, "Memo" functions and machine learning, *Nature* **218** (1968).

[7] Y.N. Moschovakis, *Elementary Induction on Abstract Structures*, Studies in Logic and Foundations of Mathematics 77 (North-Holland, Amsterdam, 1977).

[8] S.J. Thompson, Laws in Miranda, in: *Conference Record of the 1986 ACM Conference on LISP and Functional Programming* (1986).

[9] D.A. Turner, Miranda: A non-strict functional language with polymorphic types, in: J.-P. Jouannaud, ed., *Functional Programming Languages and Computer Architecture* (Springer, Berlin, 1985).

[10] P. Wadler, Views: A way for pattern-matching to cohabit with data abstraction, in: *Proceedings 14th ACM Symposium on Principles of Programming Languages* (ACM, New York, 1987).