EQUATIONAL REASONING SUPPORT FOR ORWELL

by

Stephen Paul Wilson

Technical Monograph PRG-104
ISBN 0-902928-81-3

March 1993

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD
England

Copyright © 1993 Stephen Paul Wilson

Oxford University Computing Laboratory
Programming Research Group
11 Keble Road
Oxford OX1 3QD
England

Equational Reasoning Support for Orwell

Stephen Paul Wilson

Abstract

This report describes the development of an interactive equational reasoning assistant, ERA, for an Orwell type notation (Orwell is a functional programming language which is very similar to Miranda; Miranda is a trademark of Research Software Ltd). It is designed to support both proof and program synthesis, using either the inductive or purely calculational styles. Aspects of the tool include pretty printed output, proof schemas, induction hypothesis generation, multiple current proofs, rewriting modulo associativity, and the ability to save proofs mid-stream.

Acknowledgements

I would especially like to thank Richard Bird for his enthusiastic supervision and much needed guidance, throughout this project. Thanks to Mark Jones for his kind advice and also for allowing me to use his prototype rewriting system as a starting point for the proof tool we developed. I'm also grateful to Bernard Sufrin, Mike Spivey and Wayne Luk for our discussions. Thanks to Jeff Sanders for his helpful supervision during term time.

I know that there are more romantic things in life than dissertations, but I would like to dedicate this project to my wife Claire for her support and understanding throughout the year.

Contents

Chapter 1

Introduction

1.1 Equational Reasoning

An important aspect of functional programs is that their properties can be established using the simple process of equational reasoning. Consider a very simple example. Suppose we know the definition of the square function

```
square x = x * x
```

and we have an expression

```
1 + square (x+y)
```

tben we can use the definition to rewrite the expression as follows

```
1 + (x+y) * (x+y)
```

Here we applied the law left to right, by matching the left hand side against a subexpression of tbe original expression; this is sometimes known as **reduction** or **unfolding** with a law. We can also apply a law right to left, by matching the right hand side with a subexpression; this is known as **folding** or **unreducing** with a law. Above, the left hand side of square matched the subexpression

```
square(x+y)
```

So, given some definitions of functions and laws about them, we can take an expression and transform it over a number of steps by applying

laws and definitions, through the substitution of equals for equals. We use the terms law, lemma and equation interchangeably throughout the text to mean an equation relating two Orwell expressions. Such laws may be conditional on certain assumptions.

Proof and Synthesis

We can use this process of expression transformation not only to verify a given equation, but also to derive new laws and definitions, a process referred to as program **synthesis** or program **calculation**. We can use synthesis to calculate more efficient definitions of functions, or perhaps alternative recursive definitions.

To illustrate the style of proofs we wish to support, we give two examples and make some comments about them.

An Example Inductive Proof

The purpose of the first proof is to illustrate how such proofs are discovered: we want a tool to help us calculate proofs that we haven't done before, not one which will allow us to type in a proof after we've calculated it. The example states that the `foldr` operator is equivalent to the `foldl` operator on monoids.

First of all we need some definitions:

```
foldr f a []     = a                          (foldr.1)
foldr f a (x:xs) = f x (foldr f a xs)         (foldr.2)

foldl f a []     = a                          (foldl.1)
foldl f a (x:xs) = foldl f (f a x) xs         (foldl.2)
```

Our theorem is

```
foldr f a xs = foldl f a xs
```

provided that `f` and `a` form a monoid, that is,

```
f x a       = x                               (left-id)
f a x       = x                               (right-id)
f x (f y z) = f (f x y) z                     (assoc)
```

We wish to prove that the theorem holds for all finite lists **xs**. For this we use finite list induction and show that $P([\,])$ and $P(xs) \Rightarrow P(x : xs)$.

We use a simple convention to show the substitutions we make. The notation

```
P (x:xs / xs)
```

means that we replace all occurrences of **xs** with **x:xs**, in equation P.

We work on left and right hand sides separately.

Base Case

```
lhs ([] / xs)
    = { definition }
      foldr f a []
    = { foldr.1 }
      a

rhs ([] / xs)
    = { definition }
      foldl f a []
    = { foldl.1 }
      a
```

So the left and right hand sides can be rewritten to a common form, which establishes the case.

Inductive Case

We have the following inductive hypothesis:

```
foldr f a xs = foldl f a xs    (inductive-hypothesis)
```

Again we work on both sides separately:

```
lhs (x:xs / xs)
  = { definition }
    foldr f a (x : xs)
  = { foldr.2 }
    f x (foldr f a xs)
  = { inductive-hypothesis }
    f x (foldl f a xs)

rhs (x:xs / xs)
  = { definition }
    foldl f a (x : xs)
  = { foldl.2 }
    foldl f (f a x) xs
  = { left-id }
    foldl f x xs
```

We cannot get any further with our definitions or inductive hypothesis, so we are left with proving

```
f x (foldl f a xs) = foldl f x xs
```

We prove this equation by induction on xs, again using the assumption that f and a form a monoid.

Base Case

```
lhs (□ / xs)
   = { definition }
     f x (foldl f a □)
   = {foldl.1}
     f x a
   = {right-id}
     x

rhs (□ / xs)
   = { definition }
     foldl f x []
   = { foldl.1 }
     x
```

This establishes the case.

Inductive Case

The inductive hypothesis is that for all `x`

```
f x (foldl f a xs) = foldl f x xs  (ind-hyp)
```

In the proof of the inductive case we use a fresh variable `x'`.

```
lhs (x':xs / xs)
   = { definition }
     f x (foldl f a (x' : xs))
   = { foldl.2 }
     f x (foldl f (f a x') xs)
   = { left-id }
     f x (foldl f x' xs)
   = { ind-hyp (RIGHT TO LEFT) }
     f x (f x' (foldl f a xs))
   = { assoc }
     f (f x x') (foldl f a xs)
   = { ind-hyp }
     foldl f (f x x') xs

rhs (x':xs / xs)
   = { definition }
     foldl f x (x' : xs)
   = { foldl.2 }
     foldl f (f x x') xs
```

So we have established the case, and the proof of the lemma. We can now go back to the original proof, and apply this lemma to the last step of the inductive case, completing the proof.

Commentary

Firstly, note the form of the theorem to be proved, with all variables present. Sometimes this style of proof is known as the **pointed** style, as distinct from the **pointless** style shown in the next example. This style of definition normally means that proofs about functions are carried out using induction.

To prove an equation by equational reasoning, we worked on left and right hand sides separately. For each side we tried to **reduce**

them as far as possible using our laws and definitions (reduction means applying laws left to right), the idea being to bring them to a common form. This makes many proofs just a simple exercise in simplification. The conventional style for presenting proofs is as a single chain of transformations from the left hand side, but the above can easily be converted to this form simply by reversing the right-hand sides's steps and appending to the left-hand side's steps. See Bird and Wadler [1], p111, for a simple example of this. Any proofs which succeed using this process alone can easily be automated. However, certain properties are required of our set of laws. The order of rewriting should not affect the final result; any rule set with this property is said to be Church–Rosser, or confluent. The rules should also be terminating; any rule set with these properties is said to be canonical.

Reduction on its own is insufficient to carry out the above proof: indeed we applied the induction hypothesis right to left when proving the lemma. But since we know that theorem proving in general cannot be automated, it was unlikely that reduction would be sufficient for our purposes.

When proving by induction we have to formulate a suitable induction hypothesis. For example in the proof of the lemma, we had

```
f x (foldl f a xs) = foldl f x xs
```

as the induction hypothesis. It is important to note that the lemma is asserted to hold for any `x`. For example, in the last step of the left hand side we instantiated `x` to `(f x x')`. We cannot however generalize `f` or `a` as we make assumptions about them, and we cannot generalise `xs`, as that is the variable being induced upon.

In the above proof we were very loose about quantifiers, but any machine support would have to be exact about them. The variables of a law without context conditions (or hypotheses) are implicitly universally quantified:

```
forall a x xs : foldr f a (x:xs) = f x (foldr f a xs)
```

However, with hypotheses (including induction hypotheses) this is not the case. The statement that `a` is a left identity for `f` does not hold for all `f` and `a`, it holds only for the `f` and `a` used in our consequent.

```
forall x : f a x = x
```

and the lemma's induction hypothesis would therefore be quantified as follows:

```
forall x : f x (foldl f a xs) = foldl f x xs
```

where f, a, and xs are specific. If we did not generalize x then the proof would not go through. We return to the problems about quantification in the next chapter.

In the above example we proved two theorems, both of which were conditional on f and a forming a monoid. When we apply such laws these proof obligations must be discharged. For example, in the step

```
  foldr (++) [] [x0,x1,x2]
= { first-duality }
  foldl (++) [] [x0,x1,x2]
```

the following side conditions will have to be discharged:

```
[] ++ x  = x
x  ++ [] = x
x ++ (y ++ z) = (x ++ y) ++ z
```

An Example Calculational Proof

As another example, consider the theorem

```
filter p . map f = map f . filter (p . f)
```

It is possible to prove this law using induction, but by characterizing filter in a certain way, we can prove this conjecture by purely equational reasoning at the pointless level. Define

```
filter p = concat . map (if p (one, none))
```

One and none can be defined by their application to variables:

```
one x  = [x]
none x = []
```

We will not use these definitions, but rather some laws about them (these laws can of course be proved by induction):

```
map f . one  = one . f                    (map.one)
none . f     = none                       (none-fun)
map f . none = none                       (map.none)
```

We do not need the recursive definitions of map and concat either, just the following:

```
map f . map g = map (f . g)               (map-distrib)
map f . concat = concat . map (map f)     (map-promotion)
```

We use a special conditional form, characterized by the following laws:

```
if p (f, g) . h  = if (p.h) (f.h, g.h)    (if-fun)
h . if p (f , g) = if p (h.f, h.g)        (fun-if)
```

Once again, we can take the left and right hand sides separately:

```
lhs
   = { definition }
     filter p . map f
   = { filter }
     concat . map (if p (one, none)) . map f
   = { map-distrib }
     concat . map (if p (one, none) . f )
   = { if-fun }
     concat . map (if (p . f) ((one . f), (none . f)))
   = { none-fun }
     concat . map (if (p . f) ((one . f), none))
```

```
rhs
  = { definition }
    map f . filter (p . f)
  = { filter }
    map f . concat . map (if (p . f) (one, none))
  = { map-promotion }
    concat . map (map f) . map (if (p . f) (one, none))
  = { map-distrib }
    concat . map (map f . if (p . f) (one, none))
  = { fun-if }
    concat. map (if (p.f) ((map f.one), (map f.none)))
  = { map-none }
    concat . map (if (p . f) ((map f . one), none))
  = { map-one }
    concat . map (if (p . f) ((one . f), none))
```

So left and right hand sides have been reduced to equivalent forms, completing our proof.

Commentary

This time we cast our theorem as the composition of a number of higher order functions. We did not use the definitions at all, just properties of the functions involved. These were expressed using functional composition. This is known as the **pointless** or **calculational** style of programming, and it gives rise to a dual style of proof. It is a very concise style of programming and proof.

The most important point about such calculations is the reliance on the associativity of composition. It is never given as a step in the proof, indeed it would tediously painful to do so. A list of terms composed together is not written with brackets, such as

```
((f . g) . h) . i)
```

but always written as

```
f . g . h . i
```

Any segment of this list is a valid sub–expression, and therefore suitable for rewriting. Bird [2] gives a calculation of an efficient solution to the

maximum segment sum problem, and relies heavily on the associativity of composition. At one point in the calculation we have five terms composed together:

```
max . map sum . concat . map tails .inits
```

It would be terrible if we had to apply associativity of composition at every step in this proof. There are other algebraic properties which are sometimes assumed in proofs, such as commutativity, identities, and idempotence. Of the other three, commutativity is probably the most important.

There are a number of calculational style laws given in [2], and some of them make use of **where** clauses for auxiliary definitions. When we apply such laws, our new expression is qualified by a new definition. To be formal we should augment each step in the proof with these definitions, but this would clutter a simple style of proof. Most laws using **where** clauses can be written in an alternative form, and as we shall see later this is what we propose to do.

We also note that a preferahle notation for this style of proof is Squiggol [3, 4], which defines special symbols for functions such as we have used here. But to provide this in a satisfactory way would require special fonts for the operators.

1.2 Some Core Requirements

We are now in a position to state some high level requirements for a tool to support equational reasoning about Orwell functions.

We wisb to provide support for both proof and synthesis using equational reasoning in either of the inductive or purely calculational styles. We noted that we cannot hope to automate all proofs, and so what we need is a machine assistant which can faithfully apply the commands we give it. It should be useful as a teaching aid so that novice functional programmers can learn very quickly how to conduct proofs and calculations, without making the sort of mistakes inherent with the manual approach.

We want help to discover new proofs of the type illustrated above. We do not want just to be able to type proofs into the tool after we have done them with pen and paper - we wish to replace the pen and paper.

It is to be an experimental tool, where the user does not necessarily know beforehand how the proof is to proceed. It is possible that a user may make a mistake, and so must be able to remove erroneous steps from the proof, and even construct alternative proofs.

We want an equational reasoning assistant and so for convenience we suggest the name ERA, as an abbreviation.

Laws

Firstly, we need to be able to specify named definitions and laws about Orwell functions. The form of these laws should allow side conditions to be attached, as in our first example. When conditional laws are applied, side conditions arise and so we must cater for these too.

On start-up, the tool should read in a set of standard definitions, just as Orwell reads in its standard prelude on start-up. The user should be able to specify definitions in other files and read these in too. It is very important that laws can be both browsed and added to during a proof.

Rewriting with Laws

For expressions of a reasonable size, it is very common for a human prover to make mistakes when manually rewriting them with a law. It is very easy to write x instead of x' and even this minor error can cause problems. Rewriting expressions with laws is a well defined process and so at the very least we want a term rewriting system to do this for us. It should be possible to re-write expressions by folding or unfolding with a named law. If it is possible to apply the law in more than one place in the expression, then all of the options should be offered to the user, who can then select the appropriate one.

We noted that some proofs can be completed by reduction alone, or even partially done, and so we should provide a facility to reduce with the laws, or a subset of them. This we shall call our **go button**.

As we saw, calculational style proofs rely on the associativity of composition, and so support for such proofs must take this into account. Since associativity is a property of operators other than just composition, it should be possible to specify any binary operator to have this property. There are other properties such as commutativity, identity and idempotence. Perhaps the most important of these is

commutativity. It could be provided as a law as follows

```
x + y = y + x        (+.commutes)
```

but this obviously gives rise to a non-terminating rule set, and it would be dangerous to use our **go button** if this was a law. One approach would be to provide rewriting modulo commutativity too, but we know that associative/commutative matching is an NP-hard problem. What we propose is that equations can be disabled, so that our **go button** will not try and use such laws. We do not propose to provide rewriting modulo identity either, as this can easily be added as an equation (likewise for idempotence).

Splitting into Sub-proofs

We need to be able to break derivations down into sub-derivations, using either simple case analysis or induction. There should be a general facility for stating induction rules, or schemas.

Multiple Derivations

In the first example we proved and used a lemma, so we need to be able to work on more than one proof at a time, and allow the user to work on each at will. We must also allow the results of derivations to be *learnt* for use as lemmas in other calculations. Having multiple proofs means that we must also provide good navigation facilities to move between the proofs and sub-proofs.

Mode of Working

We must allow the user to work on left and right-hand sides of proofs separately and attempt to bring them to a common form. They should be able to switch between sides at will. The tool should be able to recognize when left and right hand-sides have been reduced to common forms. Ideally the user should be able to see both sides at the same time, but this is not possible with a simple teletype interface.

Input and Output

We should provide presentable output of proofs, with the the left and right-hand side transformations combined to form a single chain of reasoning.

The user may not complete a proof in one session with the tool and so it should also be possible to save an incomplete proof to be finished at a later date.

1.3 Other Tools

Lindsay et al, [10], surveys some of the existing machine support for theorem proving and identifies the following key features which characterize the systems:

1. Object language and underlying logic,
2. Theory building facilities,
3. Automated deduction facilities,
4. Emphasis on user interaction,
5. Tactical facilities,
6. Proof Management support.

The object language, underlying logic and the theory building facilities have a great affect on the type of proofs which can be performed, although some systems are able to simulate logics other than their native one.

Most tools emphasize either automation or user interaction, and Lindsay claims that "the degree of user interaction is inversely proportional to the sophistication of a notional **estab** function." Here **estab** is a function to establish the truth of the current goal, and can return a verdict of true, false, or open, or even fail to terminate. Lets now go on and look at some existing systems.

The Boyer-Moore Theorem Prover

Boyer and Moore's theorem prover, [6], BM for short, is an automatic one: the user supplies a theorem and BM tries to prove it from its axioms and already proved results. No user interaction is possible during a proof but a brief commentary is given as it proceeds. It is possible that BM will not terminate for some proofs, and when this happens, it is possible to stop it, add a lemma and start it off again. An overview of its proof procedure is given in [10]. It is a prime example of a system with a sophisticated **estab** function.

Its object language is a Lisp type notation, which can be complicated to read and write. Types must be defined in a certain way, and be countable - BM requires a count function from the new type to the natural numbers - and recursive definitions are accepted only if the user can identify a measure function (a partial order) which will decrease on recursive calls. Building definitions in this way allows BM to formulate suitable induction hypotheses for use during theorem proving.

BM, because of its lack of user interaction, and minimal commentary, is not suited for novice functional programmers to try out new proofs. It is however one of the most sophisticated automatic tools around and has some very clever heuristics for induction and generalization.

OBJ

OBJ [8] is a wide spectrum functional programming language based upon order sorted equational logic. Its rigorous semantics permits a declarative style of programming, but most importantly to us, it also supports theorem proving. It provides sophisticated hierarchical theory building, through the use of "objects" and "theories".

The operational semantics of OBJ is reduction: programs are executed (i.e. theorems are proved) by term rewriting. This rewriting can be done modulo associativity, commutativity and identity. Modulo identity rewriting is done by a process of equation completion, but this can generate a large number of extra equations and even non terminating rule sets. Associativity and commutativity are done using special data structures, but Goguen [8] notes the implementation of this rewriting is inefficient. The problem of associative, commutative matching is an NP-complete one.

As an example, we can request OBJ to prove a theorem with its reduce command:

```
red 1 ** (m * 0) == (1 ** m) ** 0
```

This will cause OBJ to reduce the left and right hand sides of the equality to normal form, using the equations in the natural numbers object. Then if both sides are equal it will return the value true. Other examples are given in Goguen [8].

By default, OBJ will just give an answer of *true* or *false*. The user can request running commentary which gives all of intermediate steps in the rewrite process. They are not justified by the equations used, and so this commentary is not as suitable for presentable output as that produced by, say, WWW (see below).

It would be possible to use OBJ for conducting proofs about Orwell programs, we could simply define the necessary equations use the red command above to request proof. This means the user would have to learn OBJ, and be forced to update OBJ objects and theories.

WWW

WWW (We Wite Wules) is a tool local to Oxford, currently under development by Mike Spivey. (Since it is very recent there is no published documenation, but [13] describes rewriting with functional languages.) It is aimed principally at automatic proof in the Bird–Meertens formalism. It is written in Orwell, and indeed its interface is a set of powerful higher-order functions that provide a very flexible approach to proof. It is best run inside an Emacs Shell because of this functional interface. It is worth looking at an example of its use.

```
> theory = map parse_eqn [
>     "Length-def:    Length = Sum . (Const 1) *",
>     "Sum-def:       Sum = (+) /",
>     "red-promote:   f / . (++) / = f / . (f /) *",
>     "map-compose:   (g . f) * = g * . f *",
>     "map-promote:   f * . (++) / = (++) / . (f *) *"]

?prove(reduce theory)"lemma:Length.(++) / = Sum.Length *"

    Length . (++) /
=     {Length-def}
    Sum . Const 1 * . (++) /
=     {Sum-def}
    (+) / . Const 1 * . (++) /
=     {map-promote}
    (+) / . (++) / . (Const 1 *) *
=     {red-promote}
    (+) / . ((+) /) * . (Const 1 *) *
=     {Sum-def}
    (+) / . Sum * . (Const 1 *) *
=     {map-compose}
    (+) / . (Sum . Const 1 *) *
=     {Length-def}
    (+) / . Length *
=     {Sum-def}
    Sum . Length *
```

So, we set up some equations as the function theory, and then request proof of the lemma using those equations. WWW reduces left and right hand sides to normal form, which should be equal, and then joins the two derivations together to give a well presented proof. The notation above is that used in [3, 4].

The most important thing to note is that WWW performs the re–writing of expressions taking into account the associativity of composition. It does this by representing composition as a special data structure, which in Orwell notation is

```
expr ::= Const string     |
         Var   string     |
         Apply expr expr  |
         Compose [expr]
```

Composition is represented as a list of composed expressions, so that all segments form sub-expressions and potential matches. Because composition is treated as a special operator, it does mean that we cannot declare other operators to be associative.

WWW's emphasis is very much on automatic proofs, and the user cannot intervene once a proof has been started. Its output is very suitable for presentation and we will do well to mimic it.

Provewell

Provewell [11], is an equational reasoning tool for Orwell, with a simple tele-type interface, developed by Andy Sphyris an M.Sc. student at Oxford in 1989.

Provewell consists of a law compiler and a core reasoner. The compiler is used to convert a list of named equations into a large static function to be included with the reasoner at run time. This means that the laws are fixed, but there is a limited facility for adding other axioms.

To use the reasoner the user can first input any extra named laws, and once the reasoning begins no more can be added. An expression is input, and then the user transforms it over a number of steps by naming laws to apply. If a law is applicable in more than one place in the current expression, the user is offered each in turn. It is possible to delete steps and also replay the proof to the screen (user messages often clutter the screen), or to a file. Once finished it is not possible to do any more calculations without restarting the whole tool.

The tool is written in Orwell, and is best viewed as a useful specification of a rewriting engine for expressions, although WWW supercedes it. It has a very crude interface which makes it difficult to use.

EQR

EQR [9] is another tool, developed by Mark Jones, also an M.Sc. student in 1989 as a small part of his project. He used it to help do

equational reasoning about an Orwell implmentation of a Prolog Complier. Since this was a minor part of his project, there is only a brief one page description of its facilities.

It allows the definition of laws and the transformation of a single expression by naming laws to apply. At the end of a transformation a new law can be learnt. Laws are read in from a file at start up and can be extended at anytime during a derivation. New constant symbols and operators can also be defined. All definitions can be browsed during a calculation, which is very important. Laws can be applied in both directions and there is a facility to repeatedly apply laws until no further reductions are possible.

Again only one fragment at a time is allowed, no conditional laws are allowed, it is not possible to do structured proofs (only one side of a law can be worked on), and all laws are implicitly universally quantified.

Jape

Yet another tool recently developed at Oxford, is Jape, written by Sufrin and Bornat [5]. It is an X-Windows based proof utility and is used to calculate proofs interactively. It is based on hypothetical deduction and has a notion of safe substitution. It can be tailored to deal with most formalisms, and this has been done for equational logic, and then used to carry out proofs about Orwell programs.

The user edits a proof as a tree structure, by clicking on expressions that require rewriting. The main goal appears at the root of the tree and the user bnilds the tree by splitting this into simpler and simpler goals. It provides very good tactical facilities, and it is possible to attach proof rules and tactics to mouse buttons to allow painless proving. Induction hypotheses can also be stated for theories.

There is also a facility for performing proofs modulo associativity. This is done by normalizing expressions with the associativity rnle so they do not require brackets when printed. Then the user can select sub-expressions for transformation. The sub-expression may not currently be a valid sub-expression but Jape will use the associativity rule in the reverse direction to shuffle the expression into a suitable form. Two special tactics are provided to do this.

Calculation by Computer

In Chisolm [7], the author describes a tool for interactive proof in the Bird-Meertens style. He notes there are a number of problems with tools based on automation, and support intended to mimic the pen and paper approach is preferable. It is windows based and, like Jape, allows the user to select sub-expressions to work on. It can be parameterized to most formalisms.

Chisolm argues that it is impossible to automate all proofs, and so support should be provided for experimentation, where the user guides the development of a proof interactively. For example the user should be able to replace a subexpression manually, rather than be restricted to automatic instantiation of laws at all times. This is referred to as *syntactic editing*. The replacement must preserve the correct syntax of the original expression. The aim is to use automated transformation as much as possible, but if necessary do a syntactic edit. So the ability to validate proofs is sacrificed, and the user decides the level of formality and what is important in a proof.

In general an automatic proof engine will generate a number of unproven sub-goals necessary to the proof of the original goal. The user then finds it very difficult to relate these sub-goals back to the original goal, and difficult to decide what is important and what is not. If done interactively, the user controls sub-goal generation and, provided good proof management facilities are provided, the problem goes away.

A similar problem is that results of calculations are often left in an expanded form, so it is difficult to relate the result back to the starting point. We require notational abbreviations to be used when displaying results.

This tool is very good, but we prefer to insist upon automatic instantiation of laws, and a simpler object language.

Conclusions

Of the tools reviewed Jape seems to offer the best approach to interactive proof. It is very easy for the user to select a sub-expression to re-write, and the tree structure of the proof is clear. However, we wish to provide a more flattened style of proof like those in [1], with induction and case analysis treated differently to rewriting with equations.

We also want to reason in a forwards, rather than backwards, direction. Also, since we are to build a tele-type interface, the JAPE approach to modulo associative re-writing would be inappropriate. There are no means to select sub-expressions from the terminal, so we need to take an approach similar to that used by WWW or OBJ.

Can any of the other tools form a useful basis for our new tool? WWW carries out proofs with the approach we require and its output is in a very suitable form. It does not have an interface however and we note that it is difficult to build effective interfaces with a functional language.

EQR, which is written in C, is also a candidate: it has a reasonably good user interface and could be extended quite easily. We would have to modify EQR to provide proof as well as synthesis, multiple derivations, conditional laws, case analysis and induction, and rewriting modulo associativity as well as provide a facility to disable dangerous laws such as commutativity when appropriate.

Chapter 2

Object Language

We base the object language of ERA on simple untyped Orwell expressions. This will allow us to do most types of proofs, and more importantly means that we can use EQR [9] as a starting point. We make the assumption that all expressions and definitions are built from functional application. Orwell does have features which do not fit neatly into this scheme. List comprehensions, conditional definitions and where clauses are the three main problems.

Where Clauses

Where clauses are often used in function definitions, as a modularising facility. It is always possible to convert them to a global form. For example, consider the definition of list difference in BW [1]:

```
xs -- []     = xs
xs -- (y:ys)
   = remove xs y -- ys
     where
      remove [] y = []
      remove (x:xs) y = xs,                 if x = y
                      = x : remove xs y, otherwise
```

Tbe subsidiary function remove can be lifted to top level:

```
xs -- []     = xs
xs -- (y:ys) = remove xs y -- ys

remove [] y     = []
remove (x:xs) y = xs,              if x = y
                = x : remove xs y, otherwise
```

However, sometimes the where clauses use variables that appear as arguments to the function being defined. For example, consider the fold-map fusion law:

```
foldl f a . map g = foldl h a
                    where h b x = f b (g x)
```

Here, h, has f and g as implicit parameters. To get rid of the where clause we must make these parameters explicit by introducing an extra combinator:

```
foldl f a . map g = foldl (fmfuse f g) a

fmfuse f g b x = f b (g x)
```

Here we define a special combinator, fmfuse, which takes f and g as parameters. The readability of the law is compromised, but we feel this is an acceptable approach. We could define such laws using a set of general combinators, but we feel the specially named combinators are more instructive.

Sometimes, however, where clauses are used to isolate common subexpressions, in a definition, so that they need only be calculated once. This can not be handled by this expansion scheme. For example, consider the fold-scan fusion law:

```
foldl f a . scanl g b = fst . foldl h (f a b, b)
                        where
                            h (u, v) x = (f u w, w)
                            w = g v x
```

We can make f and g explicit parameters, but cannot avoid w being calculated twice. We can rewrite this law in the form

```
foldl f a.scanl g b = fst.foldl (fsfuse f g) (f a b,b)

fsfuse f g (u, v) x = (f u (g v x), g v x)
```

This problem will prevent ERA from supporting derivations such as the calculation of a fast fibonacci function, see BW [1], p131. On the other hand this does not affect the correctness, only the efficiency of the final result.

List Comprehensions

List comprehensions are very useful syntactic sugar, but they can always be converted into a form involving `map`, `filter` and `concat`. Becaise they use bound variables, we will not provide them.

On the other hand, there is a useful special form for lists that improves readability. We set up the correspondences between the two forms:

```
[x]   = x:[]
[x,y] = x:y:[]
```

Conditional Definitions

With Orwell, conditional definitions are specified with a special syntax as follows:

```
filter p (x:xs) = x : filter p xs, if p x
                = filter p xs,     otherwise
```

Since we do not wish to provide this special form, we define a special conditional function:

```
if (True)  t f = t
if (False) t f = f
```

Then the above definition can be written as

```
filter p (x:xs) = if (p x) (x:filter p xs)(filter p xs)
```

This idea is not entirely satisfactory for definitions involving multiple cases, such as

```
member x (Bin y t1 t2) = member x t1,   if x < y
                       = True,          if x = y
                       = member x t2,   if x > y
```

This would have to be rewritteu as

```
member x (Bin y t1 t2) = if (x < y) (member x t1)
                         (if (x = y) True (member x t2))
```

Types

ERA is based on untyped expressions, and this does mean that it is possible to derive incorrectly typed expressions from valid ones, for example applying a law such as

```
fst (x, y) = x
```

right to left. The variable `x` matches all possible sub-expressions, and so is very dangerous. In fact this is a problem whenever no constants appear on the side of the equation being used as a match.

New variables can also be introduced when applying laws, and this happens when there are less variables on the matching side than on the instantiating side. Again this is a source of type violations.

To solve this problem all laws should satisfy the following definition principle:

$$vars(lhs) \supseteq vars(rhs)$$

where `vars` is a list of all of the distinct variables occurring in an expression.

Then the user should always try and apply laws left to right. If the starting expression is well typed, and we apply a well typed law left to right, satisfying this principle, then we are guaranteed a well typed answer. This is known as the Subject Reduction Theorem, see [12] for more details.

I can however think of one problem, the law relating map and id:

```
id = map id
```

Which applies when

```
id :: [a] -> [a]
```

If the law is written in a reverse form, the problem goes away.

User Defined Types

If no type system is provided, then it is not possible to provide user defined types in a sound way. We cannot have definitions such as

```
tree a ::== Nil | Bin a (tree a) (tree a)
```

Of course, it is still possible to reason with expressions involving new types, but the burden of ensuring only legally typed expressions are written is again placed on the user. The user should just define `Nil` and `Bin` as function constants, but give no definitions for them:

```
function Nil Bin
```

2.1 Laws and Definitions

We need to be able to specify named laws and definitions, and for simple laws a suitable form is

```
law <name> : <expr> = <expr>
```

For example the map distributivity law can be stated as follows:

```
law map.distrib : map f (xs ++ ys) = map f xs ++ map f ys
```

This type of law is implicitly universally quantified over its variables:

```
forall f,xs,ys : map f (xs ++ ys) = map f xs ++ map f ys
```

where map and ++ are constants which will have been pre-declared.

This simple scheme is insufficient to state laws such as the first duality theorem which depends on certain assumptions or hypotheses. For example, in order to apply the first duality theorem we must prove the three equations necessary to show we have a monoid. For this kind of scheme we propose a Horn Clause Form, which is sufficiently powerful for the sort of laws we expect to work with. A law is in Horn clause form if it consists of a set of hypotheses, and a consequent. We make the added assumption that the hypotheses and the consequents are just equations. Horn clauses are normally written

$$h_1 \wedge h_2 \wedge \cdots \wedge h_n \Rightarrow C$$

but we prefer to state the consequent first and then the hypotheses:

```
condlaw <name> : <expr> = <expr>
    law <name> : <expr> = <expr>
    ...
    law <name> : <expr> = <expr>
end
```

For example, the first duality theorem can be stated as follows:

```
condlaw FirstDuality : foldr f a xs = foldl f a xs
    law leftid  : f a x = x
    law rightid : f x a = x
    law assoc   : f x (f y z) = f (f x y) z
end
```

We assume that any variable occurring in a hypothesis which also occurs in the consequent refers to that variable specifically, and so a hypothesis would not be universally quantified over such variables. All other variables in the hypotheses are universally quantified. This assumption leads to the following, correct interpretation of the duality law:

```
forall f a xs :
   for all x :        f a x = x
   for all x :        f x a = x
   for all x, y , z : f x (f y z) = f (f x y) z
infer
   foldr f a xs = foldl f a xs
```

Note, however, that this does place a burden of care on the user. For example, if the user wrote `x` instead of `xs` in the consequent, then we would not get the correct interpretation of the law at all.

What happens when we apply such a law? Let us take the example of the first duality theorem again, if we have the following step in a derivation

```
  foldr (++) [] xs
= {  First Duality }
  foldl (++) [] xs
```

then the following obligations will be generated:

```
[] ++ x = x
x ++ [] = x
x ++ (y ++ z) = (x ++ y) ++ z
```

Note that when the law was applied we generated these three side conditions by making the same substitutions necessary to unify the left hand side of the consequent with the starting expression. They were generated as follows (using the square bracket convention for substitutions)

```
f a x = x [(++)/f, []/a]
f x a = x [(++)/f, []/a]
f x (f y z) = f (f x y) z [(++)/f, []/a]
```

2.2 Proofs and Syntheses

Statement of *proofs* are very similar to the the statement of laws. Simple proofs can be stated as follows:

```
proof map.distrib : map f (xs++ys) = map f xs ++ map f ys
```

The more general Horn Clause Form for such proofs is

```
proof <name>  : <expr> = <expr>
   hyp <name> : <expr> = <expr>
   ...
   hyp <name> : <expr> = <expr>
```

We do not provide an end to complete the statement of proofs, because we prefer to allow the user to be able to add hypotheses during a proof. As we shall see, this is necessary for some types of case analysis and induction.

We make the same assumptions for proofs that we made for laws. All variables occurring in the consequent belong to a set called the **fixed variables** of the proof, and any occurrences of these variables in the hypotheses mean that the hypotheses only apply for that specific variable. All other variables in a hypothesis belong to the set of **free variables** for that hypothesis, and are implicitly universally quantified. So for a proof we must always maintain the following condition:

$$freevars(proof) \cap fixedvars(proof) = \emptyset$$

Syntheses can be stated in a similar way, but require an initial expression rather than an equation:

```
synth <name>  : <expr>
   hyp <name> : <expr> = <expr>
   ...
   hyp <name> : <expr> = <expr>
```

Before we can go on to describe the facilities required to provide induction and case analysis, we must be more precise about what we mean by a proof. There are two aspects to a proof, its statement, and the work to carry it out.

```
proof == (statement, work)
```

The statement of a proof can be expressed as

```
statement  == (name,lhs,rhs,hypotheses,fixedvariables)
lhs, rhs   == expr
hypotheses == [namedeq]
namedeq    == (name, lhs, rhs, freevariables)
```

Note that each hypothesis has a list of its own free variables, but all hypotheses refer to the same fixed list of variables.

From our examples we know that there are two ways in which a proof can proceed, either by transforming left and right-band sides with equations, or by breaking into sub-proofs:

```
work      == (leftfrag, rightfrag, cases)
leftfrag  == frag
rightfrag == frag
cases     == [proof]
```

A derivation fragment is simply a list of steps, where a step consists of an expression and the name of a justifying law (note that because some laws can have side conditions attached we have to be able to attach obligations to derivation steps in a proof):

```
frag      == [step]
step      == (expr, law, sideconds)
sideconds == [proof]
```

The first step is taken simply from the original proof. The last step in a fragment does not have a justifying law, as we take the convention that the law refers to the next step in a fragment. Syntheses have exactly the same structure, except that we have no `rhs` in the `statement`, and no `rightfrag` in the `work` component.

2.3 Case Analysis

We must be able to perform case analysis on both proofs and syntheses. A case can be specified with a list of substitutions, or perhaps by adding new assumptions. We can define the syntax for a list of substitutions as:

```
substitutions = [substitution]
substitution  = ((<expr>)/(<var>))
```

The `hyp` command can be used to add new assumptions for a case. We provide a command to form a new case specified by a list of substitutions, for example

```
case Inductive : ((x:xs)/(xs)) ((y:ys)/(ys))
```

This would add another case to the cases of the current proof, building it as follows:

```
name = Inductive
lhs=fst(last(proof.work.lfrag))[(x:xs)/(xs),(y:ys)/(ys)]
rhs=fst(last(proof.work.rfrag))[(x:xs)/(xs),(y:ys)/(ys)]
hyp = proof.statement.hyps
fixedvariables=proof.statement.fixedvariables and x and y
```

The new proof is formed from the last steps of the derivation fragments; this allows us to do some equational reasoning before invoking case analysis. Once case analysis has been invoked, the user should be prevented from modifying the derivation fragments, as the cases rely on them staying the same.

Note that `x` and `y` would be added to the `fixedvariables` of the new proof. Normally `x` and `y` should be fresh variables, but the case command will allow already existing variables to be used. For a safe case analysis the user should always use schemas, which we describe below. Note also that we do not perform the substitutions on the hypotheses, as we expect substitutions to be made to variables about which we make no assumptions.

If we require case analysis on the value of an expression, we can do this by specifying an empty list of substitutions, and adding a hypothesis for each case.

To leave a case we provide a command up which will return the user to the parent proof. We use this command when we give a method for describing proof schemas.

2.4 Induction Hypotheses

The main emphasis in [1] is on recursive definitions and inductive derivations. We therefore need a simple facility for performing induction. It is no good hardwiring one specific form of induction into the tool, as we may wish to perform natural number induction, list induction, double list induction, or induction on trees.

For example, consider the induction needed to prove the theorem:

```
f x (foldl g y xs) = foldl g (f x y) xs
```

subject to the proviso

```
f u (g v w) = g (f u v) w
```

Here the induction hypothesis we require is just the original theorem, with x and y generalized, and all other variables fixed. Deciding which variables to generalize is done as follows:

1. The variables on which the induction is being performed are not generalized.

2. Any variables which are qualified in the hypotheses are not generalized.

3. All other variables are generalized.

So if we perform induction on xs, to prove the above theorem, then

1. we do not generalize **xs**;
2. we do not generalize **f** and **g**;
3. we generalize **x** and **y**.

To generalize **x** and **y** we will have to rename them and make them into a list of free variables for the generated hypothesis – since we noted that the set of free variables and fixed variables for a proof are non-intersecting sets. So ERA should produce the following induction hypothesis, where **x0** and **y0** are free variables.

```
f x0 foldl g y0 xs = foldl g (f x0 y0) xs
```

Note that we prefer to rename variables to something based on the original names, so we adopt the scheme of appending the smallest natural number possible to obtain a fresh variable. This is perhaps a little "unfriendly" and we could perhaps dash the variables instead. The general command takes the form:

```
ihyp <hyp-name> : <list of substitutions>
```

This command generates an induction hypothesis from its parent proof, using generalization and substitution.

For example, to generate the above induction hypothesis we first invoke case analysis

```
case Inductive : ((x':xs)/(xs))
```

and when inside this case we need the induction hypothesis, we invoke the command:

```
ihyp ind-hyp : ((xs)/(xs))
```

It may seem redundant to specify that **xs** remains the same but the tool needs to know which variables are being induced upon to perform generalization and also we sometimes require to replace the variable induced upon with another value - see for example induction on trees in a later section.

2.5 Schemas

Using the **case** and ihyp commands we now have the ability to carry out induction on our proofs. Since it would be tedious to type these commands each time, we provide a facility for eucapsulating the sequence of commands as a schema.

Since we do not always want to perform our induction on the same variables our schemas must are parameterized. A suitable scheme is as follows:

```
schemadef <name> <parameters>
   comment <text>
   <command>
   ...
   <command>
end
```

For example a simple induction schema for lists could be stated as follows:

```
> schemadef ListInduction xs x
> comment (P([]) & (P(xs) => P(x:xs))) INFERS  P(xs)
>    on xs
>    fresh x
>    case Base : ((□)/(xs))
>       up
>    case Inductive : ((x:xs)/(xs))
>       ihyp ind-hyp : ((xs)/(xs))
>       up
> end
```

When generating a new case and an induction hypothesis the variables whicb are substituted must exist in the fixed variables of the proof. Also we need to check that certain variables are fresh.

1. *On* checks that the variables on which the induction is to take place actually exist within the current proof. If they do not then the schema will fail and not be invoked.

2. *Fresh* checks that the new variables used in substitutions are indeed fresh, that is they do not occur in the list of fixed variables for the proof, or occur freely in any of the hypotheses.

   Then the induction is done by splitting the proof into two cases, and once inside the inductive case we add the appropriate inductive hypothesis.

This example uses a command called up, which leaves the current case and returns us to the parent proof. ERA commands will be described in the next Chapter. Note that schemas are macros which can contain other commands and so permit a flexible approach to proof.

This form for specifying schemas is reasonably general and allows various forms of induction. For example, assuming we have defined the Node and Tip functions, here is an induction principle on trees:

```
> schemadef TreeInduction t x lt rt
>   comment (P(Tip x) & (P(lt) & P(rt)=>P(t))) => P(t)
>   on t
>   fresh lt rt x
>   case Base : ((Tip x) / (t))
>       up
>   case Inductive : ((Node lt rt) / (t))
>       ihyp ind-hyp-left  : ((lt) / (t))
>       ihyp ind-hyp-right : ((rt) / (t))
>       up
> end
```

Here is an induction principle for reasoning about pairs of lists:

```
> schemadef DoubleListInduction xs ys x y
>     comment (P(□,ys) & P(x:xs,□) &
>              (P(xs,ys) => P(x:xs,y:ys)))
>             INFERS P(xs,ys)
>     on xs ys
>     fresh x y
>     case Base       : ((□)/(xs))
>         up
>     case Base2      : ((x:xs)/(xs)) ((□)/(ys))
>         up
>     case Inductive : ((x:xs)/(xs)) ((y:ys)/(ys))
>         ihyp ind-hyp : ((xs)/(xs)) ((ys)/(ys))
>         up
> end
```

A number of standard induction schemas are provided in the ERA prelude.

Chapter 3

User Interface

3.1 Preliminaries

To start up ERA and read in a file of definitions, type the following command at the Unix prompt.

```
era prelude.era
```

This will read in the named files and prompt you for a command. Files adopt the literate script convention, so only lines prefixed with a greater than sign are interpreted by ERA, all other lines are ignored and treated as comments. As we shall see, files can contain commands as well as definitions. We suggest that, as a convention, you prefix ERA input files with a ".era" suffix. If you wish to read in further files then use the `read` command:

```
read <filename> ...
```

To leave ERA simply type

```
quit
```

This will quit immediately, so ensure you have saved the system state if you wish to continue the work of the current session at a later date, or have at least obtained output of your proofs.

3.2 Definitions

Constants

All constant symbols must be defined before they are used otherwise they will be interpreted as variables and not constants. For example, the following declares functions, prefix operators, and left and right associative operators. Note the higher the number accompanying an infix declaration, the higher precedence the operator has when being parsed. There is currently no scope for declaring mixfix or non associative operators, as is possible with Jape and other tools.

```
> function  map filter concat
> prefix    ~ #
> left  110 .
> left  50  +
> right 40  &
```

Properties of Operators

It is possible to declare binary operators to be associative. If this is done, the user should **not** add equations for associativity, otherwise, a non terminating rule set will be generated. For example, we can declare composition, addition and concatenation to be associative operators:

```
> assoc     .
> assoc     +
> assoc     ++
```

If any of these declarations are made at the command line, then they are logged to the logfile **era.log**. We can list the constants and their properties with the following command:

```
consts <pattern>
```

A pattern can contain the wildcard character * which will match any string. For example to list all constants beginning with a type:

```
consts a*
```

Laws

Named laws can be defined as follows

```
> law map.concat : map f . concat = concat.map (map f)

> condlaw FirstDuality : foldr f a xs = foldl f a xs
>    law left-id  : f a x       = x
>    law right-id : f x a       = x
>    law assoc    : f x (f y z) = f (f x y) z
> end
```

Any laws which are added at the command line are again logged. It is not possible to define conditional laws at the command line; instead the user is expected to define them in a file and read them in. You can list from the current laws as follows:

```
laws <pattern>
```

Enabling and Disabling

Experience shows that is sometimes desirable to disable some laws during a calculation. We may wish to prove a theorem that is already installed as a law, and arrange that our **go button** will not use them. For example the standard prelude contains

```
> law +.commutes : x + y = y + x
> disable +.commutes
```

Laws can easily be re-activated with the `enable` command:

```
enable +.commutes
```

3.3 Stating Proofs and Syntheses

Proof

Proofs are started with the `proof` command, as follows:

```
> proof MapDist : map f (xs++ys) = map f xs ++ map f ys

> proof FirstDuality : foldr f a xs = foldl f a xs
>    hyp left-id  : f a x       = x
>    hyp right-id : f x a       = x
>    hyp assoc    : f x (f y z) = f (f x y) z
```

The `hyp` command adds hypotheses to the current proof. Proofs are an open structure so that hypotheses can be added at any time to the current proof - this is necessary for an interactive style proof assistant. In the above example, the hypotheses only apply for the specific `f` and `a`. We can use hypotheses to rewrite expressions just as we can with laws. We can list the hypotheses of the current derivation as follows:

```
hyps <pattern>
```

and we can ask what we are currently proving with

```
proving
```

Syntheses

Syntheses are started with the `synth` command as follows:

```
> synth NewMaxSegSum : mss
```

As for a proof, hypotheses can be stated for a synthesis.

Case Analysis

You can invoke case analysis on the current proof through use of the `case` command. For example

```
case Third : ((x:xs)/(xs)) ((y:ys)/(ys))
```

This will perform the substitutions to the current proof, and generate a new sub-proof. Any hypotheses, or assumptions which the parent proof used are copied to the sub-proof. This will then become the new current proof.

Induction Hypotheses

Once inside a sub-proof it is possible to generate an induction hypothesis, and add this to the list of current hypotheses. This is done by specifying a list of substitutions to make to the parent theorem.

```
ihyp ind-hyp-left : ((lt)/(t))
```

The new hypothesis will be generated, with appropriate generalization of variables. Even if we do not require any substitutions, we must say that the variable we are inducing upon is replaced by itself. You can list the top level proofs and syntheses by simply typing

```
proofs
```

and if you are in a proof with case analysis then you can list the cases by typing

```
cases
```

3.4 Moving Around

If you have entered a sub-proof by use of the **case** command, you can return to the parent proof by typing

```
up
```

If the current proof has no parent proof, that is, it is a top level proof, then you will now be free to select a new proof from those at the top level. Otherwise you will be returned to the parent proof. If you are not currently editing a proof, then you choose one with the **select** command:

```
select FilterMap
```

and if you are in a proof you can select from the sub-proofs with the same command, for example:

```
select Base
```

3.5 Using the Laws

Switching Sides

On entry to a proof the user is expected to work on left and right hand sides separately. If you are in a synthesis, then of course there is only a left hand side. Two commands are provided to select sides to work on:

```
lhs
```

will switch to the left hand side, and

```
rhs
```

will switch to the right hand side.

Applying Laws

To reduce with a law we use the `red` command, or `r` for short, together with a law name, or pattern for a law name. For example the following are both valid commands:

```
red map.1
r filter*
```

If no laws apply then nothing happens, if one rewrite is possible this is made the next step in the proof, and if more than one rewrite is possible, you can select from them by typing the appropriate option number. Unreduction, or folding with a law, is very similar; again we have two commands:

```
unred foldr.1
u fold
```

Because laws applied backwards can often apply to an expression in a lot of ways, it is recommended that this command is not used with a pattern for laws. Note that if any new variables are introduced as a consequence the user will be asked to supply a value for them. For example unreducing with the `map.1` law would require a value to be supplied for `f`. `Map.1` is defined as follows:

```
map f □ = □
```

The value supplied will have to be an expression involving constants and variables already occurring in the proof. We can try to apply a law in either way by use of the `by` command:

```
by foldr.1
b  foldr.1
```

To repeatedly apply laws, left to right, there is a `go` command which will try and reduce an expression as much as possible, with laws matching the supplied pattern:

```
go *
```

This will apply laws as much as possible using outermost reduction, giving a commentary of steps and justifications. If a non-terminating set of laws are provided then the tool may cycle forever. So you should be careful when supplying laws, and if necessary make use of the `disable` feature we described above.

Deleting Steps

We can remove steps from the current side's derivation with the `delete` command.

```
delete
d 3
```

If the number of steps is not specified then only one step is deleted.

Queries

We can list the steps of for the current side with

```
steps
```

After you done some work on the current proof, you can ask what remains to be proven with

```
remaining
```

If case analysis has been invoked any cases which have not been proved will be listed, if not then we still need to prove left hand side equals right hand side. Conditional laws generate side conditions, which are attached to steps in a derivation fragment. The following command will list all of these, if any, and ask the user to select one:

```
sideconds
```

Again the up command can be used to return to the parent proof.

Learning Results

After you have proved a theorem you may wish to add it to the current rule set so that you can use it in other derivations. For example,

```
learn FirstDuality
```

will add a new law, with the name `FirstDuality`, based on the current derivation. If the user is currently in a synthesis, then it will take the starting expression and equate it to the last step in the transformation. The resultant law is also logged.

3.6 Schemas

The schema command can be used to invoke an induction rule on the current proof. If the user tries to apply it to a synthesis, then the cases will be generated, but obviously the induction hypotheses will not be. A schema is invoked as follows:

```
schema ListInduction xs x
```

Schemas provide a facility for specifying induction rules, they are parameterized to allow them to be used for any variable. They work like macros and we can do more than just create cases and new hypotheses, we can put commands in to actually apply laws. For example the list induction schema we gave in Chapter 2 can be augmented to go with the available laws for each side of each case generated.

```
> schemadef ListInduction xs x
>    comment [P([]) & P(xs) => P(x:xs)] IMPLIES P(xs)
>    on    xs
>    fresh x
>    case Base : (([])/(xs))
>        go *
>        rhs
>        go *
>        lhs
>    up
>    case Inductive : ((x:xs)/(xs))
>        ihyp ind-hyp : ((xs)/(xs))
>            go *
>            rhs
>            go *
>            lhs
>    up
> end
```

This schema specifies that to perform a proof by simple list induction, we break it down into two cases, and for each one we try to rewrite each side as far as possible using the available laws.

We can list the schemas with the following command:

```
schemas
```

There are a number of schemas defined in the standard prelude, given in Appendix A. When you list them, the first comment line is displayed on the screen.

3.7 Other Commands

Help

The help command can be used to get help on various commands. You simply type

```
help
```

and you will see the following menu

```
ERA HELP MENU
-------------
1 : Starting and Quitting
2 : Definitions
3 : Starting Proofs and Syntheses
4 : Navigating the System
5 : Expression Transformation
6 : Querying
7 : Input and Output
```

you then chose an option which will give you details on all of the commands in the selected category.

Listing the Fixed Variables of the Current Calculation

The command `fixed` will list all of the fixed variables for the current proof.

```
fixed
```

This is sometimes useful before adding a new hypothesis, to ensure that we achieve the correct quantifications.

Deleting Proofs

There is a command to delete cases, by specifying the name of the case to delete:

```
delcase <name>
```

and to delete top level proofs we have

```
delproof <name>
```

3.8 Input Output

Reading Files

As we saw we can read files in as follows

```
> read <file-1> <file-2> ....
```

Saving the Session

This command will save the entire state of ERA, and allow you to restart ERA with those details at a later date, by simply reading the file in.

```
> save <file>
```

Outputting Proofs for Presentation

This command will output the current proof, or subproof to a file, in a pretty printed style. Left and right hand side derivations in proofs will be glued together where appropriate. For example:

```
output FilterMap.out
```

Chapter 4

Using ERA

We set out with the aim of developing a tool to be able to support proofs and syntheses of the type given in the Bird and Wadler text book [1]. In this chapter we take some example proofs from the book, and hopefully demonstrate how easy they are to do using ERA. We also give some other examples.

4.1 Inductive Examples

Proof

As an example of an inductive proof, consider the proof of the following equation relating **subs** and **map**.

```
subs (map f xs) = map (map f) (subs xs)
```

The function **subs** returns a list of the subsequences of a list and is defined by

```
subs □ = [□]
subs (x:xs) = subs xs ++ map (x:) (subs xs)
```

Using ERA we would type

```
function subs
law subs.1 : subs □ = [□]
law subs.2 : subs (x:xs) = subs xs ++ map (x:) (subs xs)
```

Now we can state the proof, naming it **SubsMap**:

```
proof SubsMap : subs (map f xs) = map (map f) (subs xs)
```

Looking at the definition of subs we can see that structural induction over lists is required, so we type

```
schema ListInduction xs x
```

Invoking the schema for simple list induction (in the standard prelude) will break the proof into two subproofs and reduce left and right hand sides as far as possible. After ERA has read in the proof, and started doing it using list induction, we will see some commentary of the steps, and then, assuming we have a terminating rule set, it will finish. We can ask ERA if the proof succeeded by typing

```
remaining
```

We will get the answer that the inductive case has not been proven yet:

```
Still to prove :
Inductive : subs (map f (x:xs)) = map (map f)(subs (x:xs))

Given that
law  ind-hyp : subs (map f0 xs) = map (map f0) (subs xs)
```

So we select that case and again use remaining to see what left to be done:

```
select Inductive
remaining
```

and ERA will give us the following message

```
Still to prove ::
 (map (map f) (subs xs))++(map (((f x):).map f ) (subs xs))=
 (map (map f) (subs xs)) ++ (map (map f . (x:) ) (subs xs))
```

By inspection, we can see that this equation requires the law

```
map (((f x):) . map f ) = map (map f . (x:))
```

This law can easily be established by applying both sides to `xs`

```
proof maplemma : map (((f x):).map f)xs=map (map f.(x:))xs
```

and invoke list induction again, hut this time we use `x'` as we already have `x` in our expression.

```
schema ListInduction xs x'
```

We will then see some commentary and ERA will stop. We ask ERA if it succeeded by typing:

```
remaining
```

and it will say

```
Proof is complete
```

Out of interest if we examine our eventual output for the proof of this lemma, we see that the inductive hypothesis was not used, and case analysis would have been sufficient. So now lets learn this lemma as a new law:

```
learn maplemma
```

Now let's return to the inductive case of our original proof and see if we can get any further:

```
up
select SubsMap
select Inductive
```

Rather than use the new lemma explicitly lets just say go again (we are on the left hand-side)

```
go *
```

We will see one more step performed and if we type

```
remaining
```

we see that the proof is complete.

Now that we have finished we can output the proof in a presentable form to a file. So let's leave the inductive case and do this as follows:

```
up
output SubsMap.out
```

The file will look as follows:

```
SubsMap : subs (map f xs) = map (map f) (subs xs)

Splits into case analysis

Base : subs (map f []) = map (map f) (subs [])

   lhs
 = {definition}
   subs (map f [])
 = {map.1}
   subs □
 = {subs.1}
   [□]
 = {singleton}
   □ : □
 = {map.1}
   □ : (map (map f) □)
 = {map.1}
   (map f □) : (map (map f) □)
 = {map.2}
   map (map f) (□ : □)
 = {singleton}
   map (map f) [□]
 = {subs.1}
   map (map f) (subs □)

Inductive : subs (map f (x:xs))=map (map f) (subs (x:xs))

Given that
law   ind-hyp : subs (map f0 xs) = map (map f0) (subs xs)

   lhs
 = {definition}
   subs (map f (x : xs))
 = {map.2}
   subs ((f x) : (map f xs))
 = {subs.2}
```

```
  subs (map f xs) ++ map (((f x):)) (subs (map f xs))
= {ind-hyp}
  subs (map f xs) ++ map (((f x):)) (map (map f) (subs xs))
= {ind-hyp}
  map (map f) (subs xs) ++
  map (((f x):)) (map (map f) (subs xs))
= {map.5}
  map (map f) (subs xs) ++
  map (((f x):) . map f ) (subs xs)
= {maplemma}
  map (map f) (subs xs) ++ map (map f . (x:) ) (subs xs)
= {map.5}
  map (map f) (subs xs) ++
  map (map f) (map ((x:)) (subs xs))
= {map.3}
  map (map f) (subs xs ++ map ((x:)) (subs xs) )
= {subs.2}
  map (map f) (subs (x : xs))
```

Even though the proof was calculated by working on left and right hand sides, we can get a conventional presentation of the proof. We can easily do the same for the proof of the lemma.

Note that ERA could do this proof automatically if we read the following specification in from a file:

```
> function subs
> law subs.1 : subs [] = [[]]
> law subs.2 : subs (x:xs) = subs xs ++ map (x:) (subs xs)
> proof SubsMap subs (map f xs) = map (map f) (subs xs)
 We need this lemma
> law maplemma : map (((f x):) . map f) = map (map f.(x:))
> schema ListInduction xs x
  > output SubsMap.out
```

Synthesis

For a good example of program synthesis we can borrow another exercise from [1], exercise 5.5.4. The task is to synthesize recursive definitions of list subtraction and **remove** from the specification

```
law --      : xs -- ys = foldl remove xs ys
law remove : remove xs y = takewhile (=/= y) xs ++
                           drop 1 (dropwhile (=/= y) xs)
```

The definitions of takewhile, drop etc., can be seen in the standard prelude for ERA in Appendix A. I approached this task by getting the tool to do as much of the work as possible - seeing how far it can get on its own, doing steps it cannot do automatically and setting it off again. I did not try to plan this proof out before doing it, and as a consequence it is not presented in an ideal way. We can start new syntheses as follows:

```
synth New-- : xs --  ys
synth NewRemove : remove xs y
```

Let's consider `remove` first. Synthesis is done by instantiation on `xs`, the two cases are `[ ]` and `(x:xs)`.

To start off the base case we type

```
case Base : ((□)/(xs))
```

We are now inside the base case, that is, we are in a sub-synthesis with the starting expression

```
remove □ y
```

We expect this case to be simple so we try the go command:

```
go *
```

ERA then produces the following set of steps:

```
= {remove}
  (takewhile ((=/=y)) □)++(drop 1 (dropwhile ((=/=y)) □))
= {dropwhile.1}
  (takewhile ((=/=y)) □) ++ (drop 1 □)
= {takewhile.1}
  □ ++ (drop 1 □)
= {++.1}
  drop 1 □
```

Now we want to use the fact that

```
drop (n+1) □ = □
```

However, ERA does not know that 1 is of the form $n + 1$. We can convert into this form by unreducing with the equation

```
law 0+ : 0 + x = x
```

So we type

```
u 0+
```

and select the appropriate one from the list of options. We now have as the next step

```
= {0+}
drop (0 + 1) □
```

It would be nice to be able to type go again now, but this would just apply the same equation in the opposite way, something we do not want. So we just reduce with drop (note how we use the pattern so we don't have to remember the exact name of the equation):

```
r drop*
```

As our result we now have

```
remove □ y = □
```

Next we consider the inductive case, but first we must leave the current case.

```
up
case Inductive : ((x:xs)/(xs))
```

Once again, let us get ERA to do the work:

```
go *
```

We get the following sequence of steps

```
= {remove}
  (takewhile ((=/=y)) (x : xs)) ++
  (drop 1 (dropwhile ((=/=y)) (x : xs)))
= {dropwhile.2}
  (takewhile ((=/=y)) (x : xs)) ++
  (drop 1 (if ((=/=y) x) (dropwhile ((=/=y)) xs) (x : xs)))
= {Swap}
  (takewhile ((=/=y)) (x : xs)) ++
  (drop 1 (if (x =/= y) (dropwhile ((=/=y)) xs) (x : xs)))
= {takewhile.2}
  (if ((=/=y) x) (x : (takewhile ((=/=y)) xs)) []) ++
  (drop 1 (if (x =/= y) (dropwhile ((=/=y)) xs) (x : xs)))
= {Swap}
  (if (x =/= y) (x : (takewhile ((=/=y)) xs)) []) ++
  (drop 1 (if (x =/= y) (dropwhile ((=/=y)) xs) (x : xs)))
```

What we have left is quite a complicated expression, but we can see the following is present as a subexpression:

```
(x =/= y)
```

We can perform case analysis on the value of this expression and see how far we can get.

First Sub-case

We do not need to make any substitutions so give a null list as an argument to the case command. So we type the following sequence of commands

```
case X=Y : ()
hyp assumption : (x =/= y) = False
go *
```

and ERA does the following steps

```
  =  {assumption}
     (if False (x : (takewhile ((=/=y)) xs)) []) ++
     (drop 1 (if (x =/= y) (dropwhile ((=/=y)) xs) (x : xs)))
  =  {assumption}
     (if False (x : (takewhile ((=/=y)) xs)) []) ++
     (drop 1 (if False (dropwhile ((=/=y)) xs) (x : xs)))
  =  {if.False}
     [] ++ (drop 1 (if False (dropwhile ((=/=y)) xs) (x:xs)))
  =  {++.1}
     drop 1 (if False (dropwhile ((=/=y)) xs) (x : xs))
  =  {if.False}
     drop 1 (x : xs)
```

We have the same problem with drop as we had before and so we unreduce with the law relating + and 0, and select the appropriate option.

```
u 0+
```

which gives us

```
= {0+}
  drop (0 + 1) (x : xs)
```

Now we just say go with the drop laws

```
go drop*
```

which gives us the final steps for this case.

```
  =  {drop.3}
     drop 0 xs
  =  {drop.1}
     xs
```

Second Sub-case

The other sub-case is done as follows (first we must leave this one):

```
up
case (X=/=Y) : ()
hyp assumption : (x =/= y) = True
go *
```

This gives us the first sequence of steps

```
= {assumption}
  (if True (x : (takewhile ((=/=y)) xs)) []) ++
  (drop 1 (if (x =/= y) (dropwhile ((=/=y)) xs) (x : xs)))
= {assumption}
  (if True (x : (takewhile ((=/=y)) xs)) []) ++
  (drop 1 (if True (dropwhile ((=/=y)) xs) (x : xs)))
= {if.True}
  (x : (takewhile ((=/=y)) xs)) ++
     (drop 1 (if True (dropwhile ((=/=y)) xs) (x : xs)))
= {++.2}
  x : ((takewhile ((=/=y)) xs) ++
  (drop 1 (if True (dropwhile ((=/=y)) xs) (x : xs))))
= {if.True}
  x : ((takewhile ((=/=y)) xs) ++
  (drop 1 (dropwhile ((=/=y)) xs)))
```

Looking at the expression in the outer parenthesis we see that it matches the right hand side of the definition of remove, and so we can unreduce with that law:

```
u remove
```

which gives us the final step

```
= {remove}
  x : (remove xs y)
```

So collecting our definitions into a an Orwell form we have that

```
remove [] y = []
remove (x:xs) y = xs,                    if x = y
                = x : (remove xs y) , otherwise
```

This example can be quite time consuming to do by hand, but here we can do it very quickly, without much thought.

List Subtraction

Now let us turn our attention to synthesising a new definition of list subtraction. Synthesis is by instantiation on `xs` and `ys`, the cases being `([], ys)`, `((x:xs),[])` and `((x:xs),(y:ys))`. So lets take each case in turn.

```
case 1 : (([])/(xs))
go *
```

gives us the following sequence of steps:

```
= {--}
  foldl remove [] ys
```

We cannot get any further with our available laws, although we can see that this simplifies to `[]` – because `[]` is a left zero of `remove`, and there is a special law about `foldl` and left zeros:

```
condlaw foldl.leftzero : foldl f a xs = a
   law leftzero : f a x = a
end
```

This law holds only for finite lists. If we reduce with this law:

```
r foldl.leftzero
```

we get the following as our final step

```
= {foldl.leftzero}
  []
```

But since we applied a conditional law, there is a side condition attached to this step, namely

```
remove [] y = []
```

We can either prove this fact all over again, or ignore it because we know it holds from the synthesis of `remove`, or we could go back and learn the new definition of `remove`, and then the proof would require only one step to be completed. We ignore it in this example. So that completes our first case.

We note that introducing the new law about left zeros, introduces a problem when using our go button. Since the conditions are not checked before application, they have to be checked later, the law can be applied when its not strictly applicable. So we should disable it with

```
disable foldl.leftzero
```

and our go command will not use it: we have to ask for it explicitly.

Second Case

Now let us consider the second case

```
case 2 : ((x:xs)/(xs)) (([])/(ys))
go *
```

This gives us the following steps and completes the case.

```
  (x : xs) -- []
= {--}
  foldl remove (x : xs) []
= {foldl.1}
  x : xs
```

Third Case

Now for the last case, again we must leave the current case:

```
up
case 3 : ((x:xs)/(xs)) ((y:ys)/(ys))
go *
```

This gives us the following steps

```
  (x : xs) -- (y : ys)
= {--}
  foldl remove (x : xs) (y : ys)
= {foldl.2}
  foldl remove (remove (x : xs) y) ys
= {remove}
  foldl remove ((takewhile ((=/=y)) (x : xs)) ++
    (drop 1 (dropwhile ((=/=y)) (x : xs)))) ys
= {dropwhile.2}
  foldl remove ((takewhile ((=/=y)) (x : xs)) ++ (drop 1
  (if ((=/=y) x) (dropwhile ((=/=y)) xs) (x : xs)))) ys
= {Swap}
  foldl remove ((takewhile ((=/=y)) (x : xs)) ++ (drop 1
  (if (x =/= y) (dropwhile ((=/=y)) xs) (x : xs)))) ys
= {takewhile.2}
  foldl remove ((if ((=/=y) x)
  (x : (takewhile ((=/=y)) xs)) []) ++ (drop 1
  (if (x =/= y) (dropwhile ((=/=y)) xs) (x : xs)))) ys
= {Swap}
  foldl remove ((if (x =/= y)
  (x : (takewhile ((=/=y)) xs)) []) ++ (drop 1
  ( if (x =/= y) (dropwhile ((=/=y)) xs) (x : xs)))) ys
```

Again it looks as though we should perform case analysis on the value of `(x =/= y)`.

First Sub-case

Here, we type

```
case X=Y : ()
hyp assumption : (x =/= y) = False
go *
```

which gives us the following steps

```
= {assumption}
  foldl remove ((if False
  (x : (takewhile ((=/=y)) xs)) []) ++ (drop 1
  (if (x =/= y) (dropwhile ((=/=y)) xs) (x : xs)))) ys
= {assumption}
  foldl remove ((if False
  (x : (takewhile ((=/=y)) xs)) []) ++ (drop 1
  (if False (dropwhile ((=/=y)) xs) (x : xs)))) ys
= {if.False}
  foldl remove ([] ++
  (drop 1 (if False (dropwhile ((=/=y)) xs) (x : xs)))) ys
= {++.1}
  foldl remove (drop 1 (if False (dropwhile ((=/=y)) xs)
  (x : xs))) ys
= {if.False}
  foldl remove (drop 1 (x : xs)) ys
```

Once again we have the problem with drop, so we do the following and select the appropriate option

```
u 0+
go drop*
```

and we get the following steps

```
= {0+}
  foldl remove (drop (0 + 1) (x : xs)) ys
= {drop.3}
  foldl remove (drop 0 xs) ys
= {drop.1}
  foldl remove xs ys
```

Now we can unreduce with the definition of --, giving the final step for this case

```
= {--}
  xs -- ys
```

Second Sub-case

The case for `((x=/=y) = True)` is done in the same way:

```
up
case X=/=Y : ()
hyp assumption : (x =/= y) = True
go *
```

and we get the following series of steps:

```
  foldl remove ((if (x =/= y)
  (x : (takewhile ((=/=y)) xs)) []) ++ (drop 1
  (if (x =/= y) (dropwhile ((=/=y)) xs) (x : xs)))) ys
= {assumption}
  foldl remove ((if True
  (x : (takewhile ((=/=y)) xs)) []) ++ (drop 1
  (if (x =/= y) (dropwhile ((=/=y)) xs) (x : xs)))) ys
= {assumption}
  foldl remove ((if True
  (x : (takewhile ((=/=y)) xs)) []) ++ (drop 1
  (if True (dropwhile ((=/=y)) xs) (x : xs)))) ys
= {if.True}
  foldl remove ((x : (takewhile ((=/=y)) xs)) ++
  (drop 1 (if True (dropwhile ((=/=y)) xs) (x : xs)))) ys
= {++.2}
  foldl remove (x : ((takewhile ((=/=y)) xs) ++
  (drop 1 (if True (dropwhile((=/=y)) xs) (x : xs))))) ys
= {if.True}
  foldl remove (x : ((takewhile ((=/=y)) xs) ++
  (drop 1 (dropwhile ((=/=y)) xs)))) ys
```

Now we try and unreduce with our definitions to get a recursive characterization:

```
u remove
```

which gives us

```
= {remove}
  foldl remove (x : (remove xs y)) ys
```

and this matches the right hand side of the definition of --, so we type

```
u --
```

and we get the final step

```
= {--}
  (x : (remove xs y)) -- ys
```

To summarize we have the following new definition of remove:

```
□       -- ys      = □
(x:xs) -- □        = (x:xs)
(x:xs) -- (y:ys) = xs -- ys, if (x = y)
                   = (x : (remove xs y)) -- ys
```

Again this is a fairly complicated example to construct by hand, but it quite easy to do with ERA.

4.2 Calculational Examples

Proof

To demonstrate a calculational example, we consider again the law relating filter and map given in the introductory chapter. We can specify the laws and proof in a file as follows:

```
> left  110 .
> assoc .
> function filter map concat if
> function one none

> law filter  : filter p = concat.map (if p (one,none))
> law map-distrib : map f . map g = map (f . g)
> law map-prom    : map f . concat = concat.map (map f)
> law map-one     : map f . one  = one . f
> law none-f      : none . f     = none
> law map-none    : map f . none = none
> law if-f : if p (f, g) . h  = if (p.h) (f.h, g.h)
> law f-if        : h . if p (f,g) = if p (h . f, h . g)

> proof FilterMap : filter p.map f = map f.filter (p.f)
```

This file is self contained and so we invoke ERA without the standard prelude. After reading this in, we use ERA to reduce left and right side as far as possible. After typing

```
go *
```

we get the following steps

```
    filter p . map f
  = {filter}
    concat . map (if p (one,none)) . map f
  = {map-distrib}
    concat . map (if p (one,none) . f )
  = {if-f}
    concat . map (if (p.(f)) ((one . (f)),(none . (f))))
  = {none-f}
    concat . map (if (p . f ) ((one . f ),(none )))
```

If we then we do the same for the right hand side,

```
rhs
go *
```

we get the following steps

```
    map f . filter (p . f)
  = {filter}
    map f . concat . map (if (p . f ) (one,none))
  = {map-prom}
    concat . map (map f) . map (if (p . f ) (one,none))
  = {map-distrib}
    concat . map (map f . if (p . f ) (one,none) )
  = {f-if}
    concat . map (if (p.f) (((map f).one ),((map f).none)))
  = {map-none}
    concat . map (if (p . f ) ((map f . one ),(none )))
  = {map-one}
    concat . map (if (p . f ) ((one . f ),none))
```

If we now type

```
remaining
```

we see that the proof has succeeded, and ERA did all of the work.

Synthesis

The example in [2] of the maximum segment sum can also be calculated by ERA, by just using the go command. We set up the appropriate definitions from this paper in a file as follows:

```
Define the following operators
> left  110 .
> assoc .
> left 100 +
> right 10 MIN MAX

Define the following functions
> function foldl map concat sum fst min max
> function segs tails inits

Define the following constants for pos infinity and neg infinity
respectively
> function posinf neginf

Negative infinity is a left zero for MAX
> law MAX.1 : neginf MAX x = x
> law MAX.2 : x MAX neginf = x

We have the following laws about map, foldl and concat
> law map.distrib : map f . map g      = map (f.g)
> law map.concat  : map f . concat     = concat . map (map f)
> law foldprom    : foldl f a . concat = foldl f a . map (foldl f a)

Define sum, min and max as follows
> law sum : sum = foldl (+) 0
> law min : min = foldl (MIN) posinf
> law max : max = foldl (MAX) neginf
> function scanl
> law scanlemma : map (foldl f a) . inits = scanl f a
> function fsfuse
> law foldscanfuse : foldl f a . scanl g b =
                     fst . foldl (fsfuse f g) (f a b, b)
> law fsfuse : fsfuse f g (u, v) x = (f u (g v x), g v x)
> function mss
> law mss : mss = max . map sum . segs
> law segs : segs = concat . map tails . inits

Horners rule, defined with a special combinator horn
> function horn
> condlaw horner : foldl f a . map (foldl g b) . tails =
                   foldl (horn f g b) b
```

```
>    law distributes : g (f x y) z = f (g x z) (g y z)
>    law left-id     : f a x = x
> end
> law horn    : horn f g b x y = f (g x y) b
```

This file is self contained, although we could have made use of the definitions in the standard prelude.

We can start a synthesis as follows:

```
synth NewMaxSegSum : mss
```

and we can just instruct ERA to simplify as far as possible with the available laws:

```
go *
```

and we get the following steps in the calculation

```
  mss
= {mas}
  max . map sum . segs
= {max}
  foldl (MAX) neginf . map sum . segs
= {segs}
  foldl (MAX) neginf . map sum . concat . map tails . inits
= {map.concat}
  foldl (MAX) neginf . concat . map (map sum) . map tails . inits
= {foldprom}
  foldl (MAX) neginf . map (foldl (MAX) neginf) . map (map sum) .
  map tails . inits
= {map.distrib}
  foldl (MAX) neginf . map (foldl (MAX) neginf) .
  map (map sum . tails) . inits
= {map.distrib}
  foldl (MAX) neginf .
  map (foldl (MAX) neginf . map sum . tails ) . inits
= {sum}
  foldl (MAX) neginf .
  map (foldl (MAX) neginf . map (foldl (+) 0) . tails ) . inits
= {horner}
  foldl (MAX) neginf . map (foldl (horn (MAX) (+) 0) 0) . inits
= {scanlemma}
  foldl (MAX) neginf . scanl (horn (MAX) (+) 0) 0
= {foldscanfuse}
  fst . foldl (fsfuse (MAX) (horn (MAX) (+) 0)) (neginf MAX 0,0)
= {MAX.1}
  fst . foldl (fsfuse (MAX) (horn (MAX) (+) 0)) (0,0)
```

This is the same answer as given in [2], except for the special combinators that replace where clauses. Horners rule is conditional, so we can ask ERA if we have any side conditions to discharge:

```
sideconds
```

and we get the following

```
1 SideCondition-1 : (x MAX y) + z = (x + z) MAX (y + z)

2 SideCondition-2 : neginf MAX x = x

Enter choice (1-2) :
```

The second one is easy, so type

```
2
```

and type

```
go *
remaining
```

and we see the proof is complete. The proof of the second condition is a little more involved, and we do not give this here.

4.3 Proof about Trees

Given the definition of `size` and `nsize` on trees as defined in the standard prelude we can prove the following law:

```
size t = 1 + nsize t
```

We state the proof and invoke tree induction as follows:

```
proof SizeNsize : size t = 1 + nsize t
schema TreeInduction t x lt rt
```

We see some commentary and ERA stops. We ask what is left to prove:

```
remaining
```

We see that the inductive case was not completed, so we select it and ask again:

```
select Inductive
remaining
```

and we are told:

```
Still to prove ::
  1 + nsize lt + 1 + nsize rt  =
  1 + 1 + nsize lt + nsize rt
```

So all we need to do is apply the law about commutativity of addition. Note because this law is dangerous it is disabled from use with our go button, so we must type

```
r +.com*
```

and then select the appropriate option, and the proof will be complete.

4.4 Infinite Lists and the Bottom Element

When executing an Orwell program, any function that examines the bottom element must return the bottom element. For example the following are true statements about Orwell computations:

```
(bot = x)     = bot
(x = bot)     = bot
(bot = bot )  = bot
if bot t f    = bot
```

These facts should be used in proofs about partial and infinite lists, and so we need to be able to supply laws about equality, such as

```
(x = x) = True
```

But as we saw above this does not hold when `x` equals `bot`. So how do we supply a law about equality? What is suggested is that we have the following laws about equality:

```
(bot = x)          = bot
(x = bot)          = bot
(x = x)            = True
((x:xs) = [])      = False
((x:xs) = (y:ys)) = (x = y) & (xs = ys)
```

The laws involving bot should always be used in preference to the more general ones.

Pattern Matching and Case Exhaustion

ERA needs to be told about the strictness of pattern matching. Consider the definition of zip:

```
zip ([],     ys)     = []
zip ((x:xs), [])     = []
zip ((x:xs), (y:ys)) = (x,y) : zip (xs,ys)
```

The value of zip(bot, ys) is bot, but it is not possible for ERA to deduce this fact, so we should add the extra laws

```
zip (bot,  ys)  = ys
zip (x:xs, bot) = bot
```

Let us illustrate the above with a brief proof about partial lists. The theorem states that if we catenate a partial list to any other list, we just get the partial list. We state the theorem and invoke a schema for partial list induction as follows:

```
> proof Partial : xs ++ ys = xs
> schema PartialListInduction xs x
```

This will invoke the schema for induction on partial lists defined in the prelude. The base case has the following steps:

```
  bot ++ ys
= {++.0}
  bot
```

and the inductive case is

```
   (x : xs) ++ ys
 = {++.2}
   x : (xs ++ ys)
 = {ind-hyp}
   x : xs
```

as required.

Chapter 5

Aspects of the Design

One of the first decisions we had to make was which programming language we should use to implement our system. Functional languages are excellent for symbolic computation, such as parsing, pattern matching, and term re-writing and since these components are needed to build an equational reasoner, a functional language seems an ideal candidate for this. They are not so good, however, for building user interfaces. Provewell is a case in point. Jape and Chisholm's tool have excellent interfaces and the authors used the language C, together with a user interface library. So a compromise would he to write the engine in a functional language and the interface in C, connecting the two components by a pipe. This is the approach used for implementing Jape.

The alternative is to write the whole system in C, and avoid having to interface two different languages. We chose to do this and use EQR as a basis for our implementation. EQR is a well written, structured, system but unfortunately is not documented, and the source code minimally commented. We still feel, however, that it is worth the effort to try and use it as a basis for our new system ERA.

5.1 Data Representation

We can see from our requirements for ERA that we need to be able to maintain a table of constants, laws, schemas, and proofs. We give the representation for these structures in an Orwell type notation, and these definitions map very neatly onto the actual implementation in C.

Constants

We define a constant table to be a list of constants, where a constant consists of a name, a type and an identifier. Note that infix operators also have an associated precedence and an associativity flag.

```
consttbl  == [constant]

constant  ==  (name, type, id)

type      ::= Function |
              Left precedence  property |
              Right precedence property |
              Prefix

property  ::= Associative | NonAssociative
id        ==  num
name      ==  string
```

We initialize the constant table to have equality and a swap function, the later is used in the representation of right sections.

Laws

We provide the table of laws as a list of laws, where a law consists of a list of hypotheses (antecedents) and a consequent. Hypotheses and consequents are just named equations, and a named equation consists of a name, a left hand side, a right hand side and a list of variables. We note that the name for the whole law, is the name attached to the consequent.

```
lawlist    == [law]

law        == (sequent, hypotheses)
sequent    == namedeq
hypotheses == [namedeq]

namedeq    == (name, lhs, rhs, vars)
lhs        == expr
rhs        == expr
vars       == [(name, num, binding)]
binding    == expr
```

Variables are represented as a triple of name, an index, and a binding. When we unify two terms, then bindings are used to capture the substitutions made. We will see below that variables in expressions are indices into these lists.

Proofs

In Chapter two we introduced a little of the structure of proofs in a form convenient for our explanation of case analysis and induction. We give the full representation here:

```
proofs == [proof]
```

A proof consists of a reference to its parent proof, its statement, and the work to carry it out. We need the parent proof to allow user movement through the proof structure.

```
proof == (parent, statement, work)
```

A statement consists of a name for the proof, left and right hand sides of the equality, a list of hypotheses, and lists to maintain the fixed and free variables for the proof:

```
statement == (name, lhs, rhs, hypotheses, fixedvars, freevars)
fixedvars == vars
freevars  == vars
```

The work for a proof consists left and right derivation fragments, some sub-proofs if case analysis has been performed, the current side on

which the user is working, and True or False value depending upon whether the proof has been established or not. Derivation fragments are built from a list of steps. Steps consist of the expression rewritten to, a justifying law, and any side conditions that have to be proved.

```
work        == (lfrag, rfrag, cases, currentside, proven)

lfrag       == [step]
rfrag       == [step]
step        == (expr, law, sideproofs)
cases       == [proofs]
sideproofs  == [proofs]
currentside == Left | Right
proven      == bool
```

`Proven` will be True if left and right hand sides have been rewritten to a common form, or if case analysis has been invoked then each case should be proven. When the user alters a proof in any way, or types the `remaining` command, this value will be recalculated.

This structure is also used to represent syntheses, but left and right hand sides are ignored throughout, and `currentside` and `proven` have no meaning.

Schemas

We noted before that schemas are treated just like macros. We represent our schemas in a list, where a schema consists of a name, a list of parameters, and a list of commands.

```
schemas == [schema]

schema  == (name, [parameters], [command])
```

When we invoke a schema we supply actual values for the parameters, and so we represent a parameter as pair : its name in the definition, and the value its bound to when invoked. The bind value will be updated each time it is used. We represent a command as a string, which has the appropriate substitutions made before being passed to our command interpreter.

```
parameter  == (name, value)
command    == string
```

Expressions

EQR has a fairly complex structure for representing expressions, which in Orwell notation is as follows:

```
expr ::= Const     con
       | Var       num
       | FixedVar num
       | Apply   expr expr
       | Infix   con  expr expr
       | Prefix  con  expr
       | Prefixsection con
       | Infixsection  con
       | Leftsection   con expr
       | Rightsection  con expr
       | Tuple [expr]
       | List [expr]
```

Note that the representation of a variable and fixed variable is an index into the list of variables and fixed variables respectively. So infix and prefix operators, and sections are all represented in a special way

```
x + y = Infix + (Var x) (Var y)
~ p   = Prefix ~ (Var p)
(~)   = Prefixsection ~
(+)   = Infixsection +
(x+)  = Leftsection  + (Var x)
(+x)  = Rightsection + (Var x)
```

This means that printing expressions is simple, but does cause us some problems when stating laws about them. For example we know the following laws about sections,

```
(+) x y = x + y
(x+) y  = x + y
(+x) y  =  y + x
```

but these laws must be repeated for every distinct operator, since we cannot have an operator appearing as a variable. A solution to this is to adopt a simpler representation of expressions, as is done with Provewell:

```
expr ::= Con con
       | Var var
       | Apply expr expr
```

and have the following representations

```
x + y = Apply (Apply (Con +) (Var x)) (Var y)
- p   = Apply (Con -) (Var p)
(-)   = Con -
(+)   = Con +
(x+)  = Apply (Con +) (Var x)
(+x)  = Apply (Apply (Con swap) (Con +)) (Var x)
```

So now our first two laws about sections hold by default, and we can state the third one in a general way so that it applies for all operators.

```
swap f x y = f y x
```

We modified the parser to parse expressions into this form, removed the special cases from the matching algorithm, and made the expression printing routine recognize these special forms and print them in the correct way. Provewell formed a useful specification for this task.

5.2 Rewriting Modulo Associativity

Provewell provides a useful specification of how to do rewriting of expressions with laws. Briefly it is to generate all subexpressions from the current expression, try to unify the matching side of the law with each subexpression to obtain a list of substitutions, apply these substitutions to the other side of the law, and finally replace the original subexpression.

We have a requirement to provide rewriting modulo associativity, which means, for example, that terms composed together should be treated as though they have no bracketing, and any segment from a list of terms composed together is a legal subexpression. There are two main approaches to this.

The approach we adopt is to modify our representation of expressions to represent applications of associative operators as a list structure rather than a tree structure. This we feel is the most natural approach as it matches our intuitions well.

An alternative is to provide the associativity laws as special simplification rules. The tool would then apply these laws as much as possible to the starting expression to bring it to a normalized form. Then when we wish to apply a law, we generate all possible rewrites of the current expression using the simplification laws in the other direction, and all of these are tried to be rewritten. This approach gets very complicated when there are a number of associative operators involved. A similar approach is adopted by Jape, as we described in Chapter 1.

Normalization

We could just treat composition as an associative operator and so just have

```
expr ::= ...
       | Compose [expr]
```

This is the approach adopted for WWW, but there is no need to restrict ourselves to composition and we choose augment our representation of expressions as follows:

```
expr ::= Con con
       | Var  num
       | Fixed num
       | Apply expr expr
       | List  [expr]
       | Tuple [expr]
       | ApplyList con [expr]
```

Any constant (infix operator) can be treated as an associative operator. The user must declare associative operators before using them:

```
assoc .
assoc ++
```

Such declarations will ensure that use of composition and catenation are always treated as special operators. We convert to this form using a normalization process, which we can define using Orwell. Constants and variables are in normalized form, and tuples and lists are easily defined:

```
norm :: expr -> expr
norm (Const c)  = (Const c)
norm (Var v)    = (Var v)
norm (Tuple es) = Tuple (map norm es)
norm (List es)  = List  (map norm es)
```

Applications of an associative operator are not in normalized form and must be converted into the list form. The two terms which are arguments to the associative operator themselves must also be normalized.

```
norm (Apply (Apply (Const c) e1) e2)
    = join head e2',                    if associative c
    = Apply (Apply (Const c) e1') e2',  otherwise
      where
          e1'  = norm e1
          e2'  = norm e2
          head = join (Alist c []) e1'
%else
norm (Apply e1 e2)      = Apply (norm e1) (norm e2)
```

Note that if after applying a law we are left with an applylist of one element then that term is just removed from its encapsulating apply list.

```
norm (Alist c [e])   = e
```

Application lists themselves must be normalized: every element in the list must be normalized, and then if any element is itself an application list of the same operator, we just flatten it out.

```
norm (Alist c (e1:e2:es)) =
  Alist c (flatten c (map norm (e1:e2:es)))
```

Flatten checks for elements which are themselves application lists involving the same operator, and just removes the encapsulating application list.

```
flatten :: con -> [expr] -> [expr]
flatten c []  = []
flatten c ((Alist c' es'):es)
              = es' ++ flatten c es ,          if c = c'
```

```
                = (Alist c' es') : flatten c es, otherwise
%else
flatten c (e:es) = e : flatten c es
```

We made use of a join function which attempts to join two application lists. If the lists are for the same operator then we can merge the lists, otherwise we insert the second list into the first list.

```
join :: expr -> expr -> expr
join (Alist c1 es1) (Alist c2 es2)
       = (Alist c1 (es1 ++ es2)),                  if c1 = c2
       = (Alist c1 (es1 ++ [Alist c2 es2])),       otherwise
%else
join (Alist c1 es1) e = Alist c1 (es1 ++ [e])
```

This completes our specification of the normalization process. It is used whenever expressions are read in, and also after an expression has had a law applied to it.

Subexpressions

So now any segment of an application list forms a valid subexpression. For example if we have

```
f . g . h
```

then the following are legal subexpressions

```
f , g , h , f.g, f.g.h, g.h
```

Unification

We must also consider the unification of terms. Suppose we have generated

```
map f (xs ++ ys ++ zs)
```

as a subexpression, and we are trying to apply the law

```
map f (xs ++ ys) = map f xs ++ map f ys
```

How can we unify the left hand side of the law with our subexpression? By inspection we can see that there are two ways in which the above can be matched. We can convert the original expression into two forms:

```
map f ((xs++ys) ++ zs)
map f (xs ++ (ys ++ zs))
```

and for each we have the following unifying substitutions

```
xs == (xs ++ ys) , ys == zs
xs == xs , ys == (ys ++ zs)
```

It follows that we have to try all rearrangements of apply lists, and this amounts to calculating all of the partitions. A definition of the partitions of a list is given in [1]:

```
parts :: [a] -> [[[a]]]
parts []        = [[]]
parts [x]       = [[[x]]]
parts (x:x':xs) = map (glue x) (parts (x':xs)) ++
                  map ([x]:) (parts (x':xs))
glue x xss = (x : hd xss) : tl xss
```

We can describe the unification algoritbm in an Orwell type notation, but we cheat a little and allow some state and side effects. It is not intended to be a formal description, but rather an intuitive description. We provide a function to unify the matching side of a law with the current expression:

```
unify :: expr -> expr -> bool
```

This function takes the matching side of the law as its first parameter and the current expression as its second. We assume the existence of some free variables for the law we are using, which we call

```
subs
```

and which has type `vars`. As we unify terms we build up the substitutions, by binding to these variables. So our `unify` function has the side effect of building up a list of substitutions in subs. We proceed by case analysis on the value of `e1` and `e2`. Constants, applications, lists, and tuples, and fixed variables can only unify as follows:

```
unify (Const c) (Const c') = (c = c')

unify (Apply e1 e2) (Apply f1 f2) = unify (e1, f1) &
                                    unify (e2, f2)

unify (List es) (List es') = (#es = #es') &
                             (zipwith unify (es, es'))

unify (Tuple es) (Tuple es') = (#es = #es') &
                               (zipwith unify (es, es'))

unify (Fixed n) (Fixed n) = (n = n)
```

A variable which is unbound, is bound to the value we are matching against. If the variable is bound then its binding must unify with the value we are matching against. `Boundval` returns the value of a substitution for a variable, `boundin` returns True only if the variable is bound, and `bind` will bind a variable to a value.

```
unify (Var n) f
   = unify (boundval subs n, f), if (boundin subs n)
     bind subs n f,              otherwise
```

An apply list only unifies with another apply list of the same operator, and the two lists of terms must be able to be `matched`. This `match` function takes into account that variables in apply lists can unify with any non-empty segment of the list being matched against, provided the remainder of the two apply lists also `match`.

```
unify (Alist c es) (Alist c' es') = (c=c') &
                                    match c es es'
```

We define match to have type

```
match :: con -> [expr] -> [expr] -> bool
```

Two empty lists of expressions match up

```
match c [] [] = True
```

If the first term of the matching list is not a variable then it must unify with the first term of the list being matched against.

```
match c (e:es) (e':es') = (unify e e') &
                          (match c es es'), if notvar e
                        = matchasegment c (e:es) (e':es')
```

If the last term is a variable we must try and unify the variable with all non empty segments of the list we are matching against and also unify the remainder of the matching list with the remainder of list being matched against. Once we have a successful match we return it, and do not try any more segments. We give a pseudo code description of this as it would radically depart from our design to use a strictly Orwell style. Note comments are prefixed with an asterix

```
matchasegment c (e:es) (e':es') is defined as

* remember the old substitutions *
oldsubs := subs

* i holds the length of the initial segment and
* ranges of the length of es
i := 1

* initially we assume we have failed to match
failed := True

* try all non empty segments until we succeed or
* exhaust the list
while (i < #(e':es') & failed)

   * we fail if we cannot unify e with the initial
   * segment or we cannot unify es with the remaining
   * segment
   failed := not (unify e (Alist c (take i (e':es))) &
                  unify es (Alist c (drop i es)))

   * if we fail then we must backtrack and use the old
   * substitutions
   if failed
      subs := oldsubs

   * try a longer segment
   i := i + 1
endwhile
```

```
* if failed is False we succeed with the unifying
*  substitution in subs as required.
```

5.3 Implementation

Modules

We implemented ERA as the following modules (C source files), and provide a Makefile to build the system. All source code is under the unix SCCS facility.

era.c is the main program, and provides the interface to ERA

express.c is the module for manipulating expressions

laws.c is the module for manipulating laws

const.c is the module for manipulating constants

proof.c is the module for maintaining the current proofs and subproofs

lexparse.c is the module which provides a function to parse expressions

vars.c is the module for manipulating variables

app.c is a module with some general purpose routines used by the other modules.

C Style

All of source code modules come with a header file describing the interface provided. We adopt a standard layout for these source files, and also follow a consistent layout convention. All files pass through lint with only trivial warnings resulting.

The contents of C header files are ordered as follows

1. constant definitions,
2. exported type declarations,
3. exported function declarations.

The C source code files are ordered as follows

1. included header files,
2. constant definitions,
3. internal type declarations,
4. global variable declarations,
5. forward declarations of static functions,
6. externally visible function definitions,
7. static function definitions.

Chapter 6

Conclusions

Successes

We have taken EQR, a system for rewriting expressions, and adapted it into a useful tool to support program calculation and proof. We have followed the original brief for the support and provided something simple but usable. A large number of the exercises and examples given in [1] can now be done with the aid of ERA. Indeed a lot of them can be done automatically with no user intervention.

We believe that beginners to functional programming could very quickly learn how to use the tool, and initially check some of the given example proofs in BW. Then, as they become more proficient, they can progress to the exercises for which there are no solutions.

There are very few calculational style proofs in BW, but we have provided support for such a style through rewriting modulo associativity. This is a much more concise style of proof, and is preferred by more advanced functional programmers.

Allowing multiple proofs, and permitting the user to work on each one at will permits a very flexible approach to proof, not often found in other tools. This is essential if a user is actually going to discover proofs with ERA.

Limitations

There are limitations to ERA, and we highlight a of few of these here. We have essentially provided a rewrite engine, with no underlying logical system. So the tool can not make simple inferences. For example,

it cannot conclude `x <= y` from `x < y`. We have not provided type checking or types and this allows rewrite rules to be applied in contexts where they shouldn't be. We have also omitted *where* clauses and tbis means that some laws have to be converted to a less readable form.

All laws are based on equality, and it is not possible to state laws about inequalities. If inequalities were to be allowed it is not so simple to work on left and right hand sides of a proof.

If this tool is extended in the future, we recommend that *where* clauses and type checking be added. There are perhaps more efficient methods of implementation, but because the tool is written in C and is for interactively proving modest theorems, this is not a major concern.

We also note that there is a small amount of "space leakage" in the program because a full implementation of garbage collection is not given. This is not a serious problem, because *most* freed memory *is* collected, and the tool is only expected to be used for relatively small proofs. The necessary extra C functions can easily be added.

Graphical User Interface

At the very outset of the project we considered building a graphical user interface, but due to time constraints this was not attempted. We can now however suggest a simple design which would allow a very user friendly approach to proofs. We can make a number of observations about our proofs with ERA.

At each step in a proof there only a few possibilities for rewriting with different laws. It would nice if the user could cycle through these in simple way, before accepting a particular step. This would allow the user to easily see the different ways in which a proof could progress.

Good style is to reduce with laws as much as possible, and only unreduce with a law when absolutely necessary. So reduction should be a **default option** which is easy to do. Unreduction should be something the user has to specifically select.

It is very nice to be able to use the **go button**; it relieves the user from the tedium of a proof and allows concentration on strategy.

It would be nice if the user could see left and right hand sides at the same time, which means we really need a window for each.

Taking these points into consideration we suggest a graphical user

interface as illustrated in Figure 6.1. There is a window each for left and right hand sides, with a set of buttons for each. Each window shows the derivation fragment for each side. The tool will automatically calculate all of the possible rewrites from the current expression. One will be on display at all times. The user could move through these possibilities using the left and right arrows. When happy with the current step the user presses the down arrow to accept it, and the up arrow to return to the previous step. It should also be possible to use the cursor keys for these tasks, and they will apply to the side the user last worked on. We can say **go** for each side. So this makes reducing with laws very easy to do.

The bottom window lists all of the available laws, and the user has to request a specific law to unreduce with, by clicking on the appropriate law with the right mouse button. It should also be possible to click with the left button to reduce with a specific law.

An input line allows the user to type commands that would normally be available from the textual interface.

A button for invoking schemas is also suggested.

All other commands normally available from ERA could be provided at the menu bar at the top of the window.

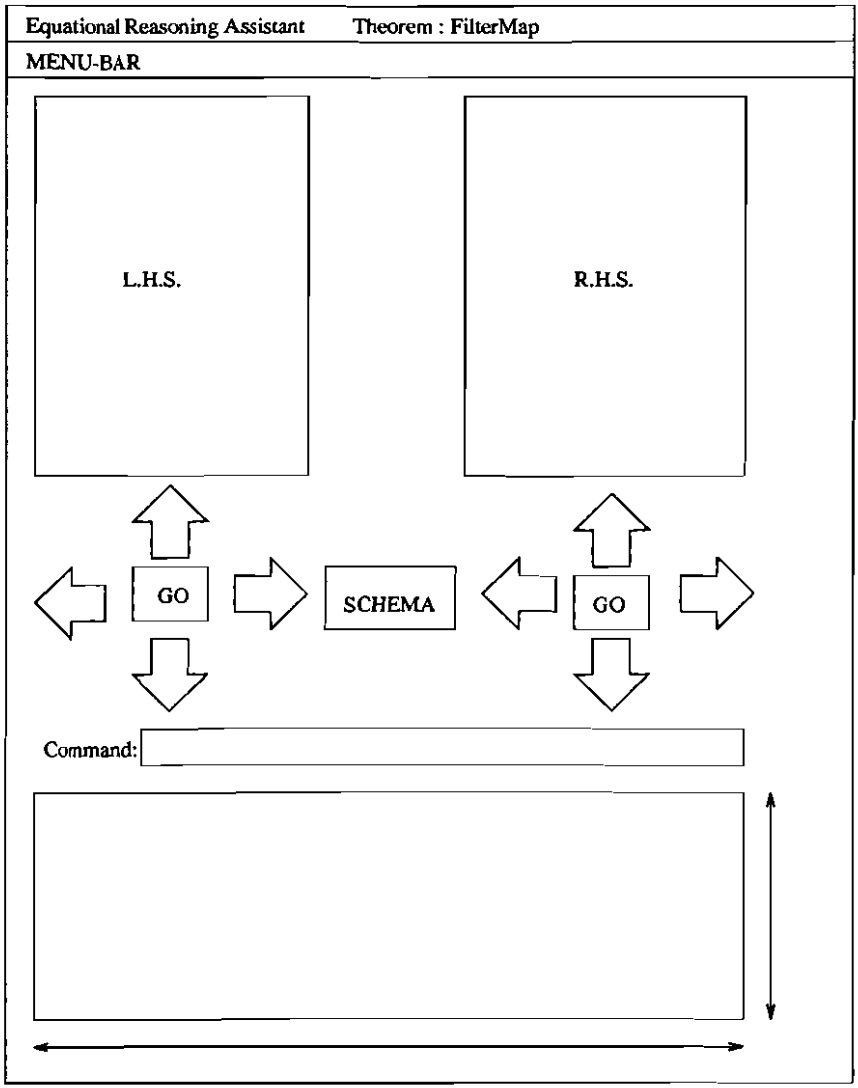

Figure 6.1: A Suggestion for a Graphical User Interface

Bibliography

[1] R. S. Bird and P. L. Wadler, *Introduction to Functional Programming*, Prentice Hall International 1988.

[2] R. S. Bird, *Algebraic Identities for Program Calculation* The Computer Jounrnal, Vol. 32, No. 2, 1989.

[3] R. S. Bird, *A Calculus of Functions for Program Derivation* Technical Monograph PRG-64, Oxford University Computing Laboratory, Programming Research Group, 1987.

[4] R. S. Bird, *Lectures on Constructive Functional Programming* Technical Monograph PRG-69, Oxford University Computing Laboratory, Programming Research Group, 1987.

[5] R. Bornat and B. Sufrin, *The Gist of Jape (draft)* Programming Research Group Oxford

[6] R. S. Boyer and J. S. Moore, *A Computational Logic* Academic Press, London.

[7] P. Chisolm, *Calculation by Computer* Department of Computer Science, University of Groningen.

[8] J. A. Goguen and T. Winkler, *Introdcing OBJ3* Computer Science Laboratory, SRI International, August 1988.

[9] M. Jones, *An Implementation of Prolog* Thesis (M.Sc.), Programming Research Group, Oxford University, 1989.

[10] P. A. Lindsay, R. C. Moore and B. Ritchie, *A Review of Existing Theorem Provers* Department of Computer Science,

University of Manchester Technical Report Series (UMCS-87-8-2)

[11] A. A. Sphyris, *Support for Equatinal Reasoning in Orwell* Thesis (M.Sc.), Programming Research Group, Oxford University, 1989.

[12] J. R. Hindley and J. P. Seldin *Introduction to Combinators and λ-Calculus.* London mathematical society student texts 1. Cambridge University Press.

[13] M. Spivey, *A Functional Theory of Exceptions* Science of Computer Programming 14 (1990) 25–42. North Holland.

[14] P. Wadler and Q. Miller, *An Introuction to Orwell 6* Oxford University Computing Laboratory, Programming Research Group 1990.

Appendix A

Standard Prelude for ERA

```
                    prelude.era

     Standard ERA definitions equational reasoning


OPERATOR DECLARATIONS :
(the higher the number the greater the precedence)
> left   110 .
> right  40 &
> right  30 \/
> right  20 : ++
> prefix ~ #
> left 100 + -
> left 101 * /
> left 110 =/=

> assoc .
> assoc ++
> assoc +

> law +.commutes : x + y = y + x
> disable +.commutes

LAWS AND FUNCTION DECLARATIONS:

> function bot

> law . :          (f.g) x   = f (g x)


> function False True
```

```
> function and or
> function if

> law and : and = foldr (&)  True
> law or  : or  = foldr (\/) False

> law if.bot   :    if bot   x y = bot
> law if.True  :    if True  x y = x
> law if.False :    if False x y = y

> law ~.bot    :    ~ bot   = bot
> law ~.True   :    ~ True  = False
> law ~.False  :    ~ False = True

> law \/.0     :    bot   \/ y = bot
> law \/.1     :    True  \/ y = True
> law \/.2     :    False \/ y = y

> law &.0      :    bot &   y = bot
> law &.1      :    False & y = False
> law &.2      :    True  & y = y

> function id const swap
> law id    :       id x       = x
> law const :       const k x  = k
> law swap  :       swap f x y = f y x

> function fst snd
> law fst   :       fst (x,y) = x
> law snd   :       snd (x,y) = y

> function foldl foldr
> law foldr.0  :    foldr f a bot    = bot
> law foldr.1  :    foldr f a []     = a
> law foldr.2  :    foldr f a (x:xs) = f x (foldr f a xs)

> law foldl.0  :    foldl f a bot    = bot
> law foldl.1  :    foldl f a []     = a
> law foldl.2  :    foldl f a (x:xs) = foldl f (f a x) xs

> function zip
> law zip.00   : zip (bot, y)         = bot
> law zip.01   : zip (x:xs, bot)      = bot
> law zip.1    : zip ([], ys)         = []
> law zip.2    : zip ((x:xs), [])     = []
```

```
> law zip.3     : zip ((x:xs), (y:ys)) = (x, y):zip (xs, ys)

> function hd tl init last Nil
> law Nil.[]    :   Nil      = []
> law singleton :   [x]      = x : []
> law list2     :   [x,y]    = x : [y]
> law list3     :   [x,y,z]  = x : [y,z]
> law list4     :   [w,x,y,z] = w : [x,y,z]

> law hd.00     :   hd bot         = bot
> law hd.0      :   hd []          = bot
> law hd        :   hd (x:xs)      = x

> law tl.00     :   tl bot         = bot
> law tl.0      :   tl []          = bot
> law tl        :   tl (x:xs)      = xs

> law last.00   :   last bot       = bot
> law last.0    :   last []        = bot
> law last.1    :   last [x]       = x
> law last.2    :   last (x:y:ys)  = last (y:ys)

> law init.00   :   init bot       = bot
> law init.0    :   last []        = bot
> law init.1    :   init [x]       = []
> law init.2    :   init (x:y:ys)  = x : init (y:ys)

> function map filter
> law map.0     :   map f bot         = bot
> law map.1     :   map f []          = []
> law map.2     :   map f (x:xs)      = f x : map f xs
> law map.3     :   map f (xs++ys)    = map f xs ++ map f ys
> law map.4     :   map f . map g     = map (f . g)
> law map.5     :   map f (map g xs)  = map (f . g) xs


> law filter.0  : filter p bot      = bot
> law filter.1  : filter p []       = []
> law filter.2  : filter p (x:xs)   = if (p x) (x:filter p xs) (filter p xs)
> law filter.3  : filter p (xs++ys)     = filter p xs ++ filter p ys
> law filter.map : map f . filter (p.f) = filter p . map f

> function concat
> law concat.0    :  concat bot              = bot
> law concat.1    :  concat []               = []
> law concat.2    :  concat (xs:xss)         = xs ++ concat xss
```

```
> law concat.3    :  concat (xs ++ ys)       = concat xs ++ concat ys
> law map.concat  :  map f . concat          = concat . map (map f)
> law concat.conc :  concat . map concat     = concat . concat
> law filter.conc :  concat . map (filter p) = filter p . concat


> law ++.0        : bot ++ ys        = bot
> law ++.1        : [] ++ ys         = ys
> law ++.2        : (x:xs) ++ ys     = x : (xs ++ ys)
> law ++.3        : xs ++ []         = xs


> law =.bot1      : (bot = x)         = bot
> law =.bot2      : (x = bot)         = bot
> law =           : (x = x)           = True
> law =.1         : ([] = [])         = True
> law =.2         : ([] = (x:xs))     = False
> law =.3         : ((x:xs) = [])     = False
> law =.4         : ((x:xs) = (y:ys)) = if (x = y) (xs = ys) False


> function takewhile
> law takewhile.1 : takewhile p [] = []
> law takewhile.2 : takewhile p (x:xs) = if (p x) (x:takewhile p xs) ([])

> function dropwhile
> law dropwhile.2 : dropwhile p (x:xs) = if (p x) (dropwhile p xs) (x:xs)
> law dropwhile.1 : dropwhile p [] = []

> function drop
> law drop.1 : drop 0 xs     = xs
> law drop.2 : drop (n+1) [] = []
> law drop.3 : drop (n+1) (x:xs) = drop n xs

> function take
> law take.1 : take 0 xs     = []
> law take.2 : take (n+1) [] = []
> law take.3 : take (n+1) (x:xs) = x : take n xs

> function reverse prefix
> law prefix      : prefix xs x = x:xs
> law reverse.0       : reverse bot = bot
> law reverse.1       : reverse [] = []
> law reverse.2       : reverse (x:xs) = reverse xs ++ (x:[])

> function rev shunt
```

```
> law rev : rev xs = shunt [] xs
> law shunt.1 : shunt ys [] = ys
> law shunt.2 : shunt ys (x:xs) = shunt (x:ys) xs
> law aux : shunt ys xs = reverse xs ++ y

> function sum
> law sum : sum = foldl (+) 0

> function product
> law product : product = foldl (*) 1

> law #.1 :  # [] = 0
> law #.2 :  # (x:xs) = 1 + # xs


> function Tip Node
> function size nsize depth

> law size.0 : size (bot) = bot
> law size.1 : size (Tip x) = 1
> law size.2 : size (Node lt rt) = size lt + size rt

> law nsize.0 : nsize (bot)   = bot
> law nsize.1 : nsize (Tip x) = 0
> law nsize.2 : nsize (Node lt rt) = 1 + nsize lt + nsize rt


> law +0 :  x + 0 = x
> law 0+ :  0 + x = x

> law foldprom : foldl f a . concat = foldl f a . map (foldl f a)

> function length
> law length : length = sum . map (const 1)

> function tails
> law tails.1 : tails [] = [[]]
> law tails.2 : tails (x:xs) = map (++ [x]) (tails xs) ++ [[]]


INDUCTION SCHEMAS FOR LISTS AND ONE FOR TREES

> schemadef ListInduction xs x
> comment P([]) & (P(xs) => P(x:xs)) => P(xs)
>    on    xs
>    fresh x
```

```
>    case Base : ((□)/(xs))
>       go *
>       rhs
>       go *
>       lhs
>    up
>    case Inductive : ((x:xs)/(xs))
>       ihyp ind-hyp : ((xs)/(xs))
>          go *
>          rhs
>          go *
>          lhs
>    up
> end


> schemadef DoubleListInduction xs ys x y
> comment P([],ys)&P(x:xs,[])&(P(xs,ys)=>P(x:xs,y:ys)) => P(xs,ys)
>    on    xs ys
>    fresh x  y
>    case Base1 : ((□)/(xs))
>          go *
>          rhs
>          go *
>          lhs
>    up
>    case Base2 : ((x:xs)/(xs)) ((□)/(ys))
>          go *
>          rhs
>          go *
>          lhs
>    up
>    case Inductive : ((x:xs)/(xs)) ((y:ys)/(ys))
>       ihyp ind-hyp : ((xs)/(xs)) ((ys)/(ys))
>          go *
>          rhs
>          go *
>          lhs
>    up
> end


> schemadef PartialListInduction xs x
> comment P(bot) & (P(xs) => P(x:xs)) => P(xs)
> on    xs
> fresh x
```

```
> case Base : ((bot)/(xs))
>   go *
>   rhs
>   go *
>   lhs
>   up
> case Inductive : ((x:xs)/(xs))
>    ihyp ind-hyp : ((xs)/(xs))
>       go *
>       rhs
>       go *
>       lhs
>       up
> end

> schemadef TreeInduction t x lt rt
> comment P(Tip x) & (P(lt) & P(rt) => P(Node lt rt)) => P(t)
>     on    t
>     fresh x lt rt
>     case Base : ((Tip x)/(t))
>            go *
>            rhs
>            go *
>            lhs
>     up
>     case Inductive : ((Node lt rt)/(t))
>         ihyp ihl : ((lt)/(t))
>         ihyp ihr : ((rt)/(t))
>            go *
>            rhs
>            go *
>            lhs
>     up
> end
```