Formal Aspects of Computing (1994) 3: 1–000

A Logic for Miranda,[1] Revisited

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

Keywords: Functional programming; program verification; logic; Miranda; operational semantics; Isabelle

Abstract. This paper expands upon work begun in the author's [Tho89], in building a logic for the Miranda functional programming language. After summarising the work in that paper, a translation of Miranda definitions into logical formulas is presented, and illustrated by means of examples. This work expands upon [Tho89] in giving a complete treatment of sequences of equations, and by examining how to translate the local definitions introduced by `where` clauses.
The status of the logic is then examined, and it is argued that the logic extends a natural operational semantics of Miranda, given by the translations of definitions into conditional equations.
Finally it is shown how the logic can be implemented in the Isabelle proof tool.

1. Introduction

'Functional programming languages are the best hope for formally-verified programming' is article of faith for many. That it remains an article of faith, rather than a fact is because of the lack of any experimental evidence. In this paper we continue work begun in [Tho89] aimed at giving a logical translation of the Miranda lazy functional programming language, [Tur90], and we report on the implementation of the system in Isabelle, [Pau90].

The paper begins in Section 2 with a self-contained summary of the material in [Tho89]. This is followed in Section 3 by a full translation of definitions in Miranda into logical formulas. This uses the notion of the complement of a pattern, from [Tho89], but unlike that paper gives a full treatment of sequences of defining equations, as well as local definitions as given in `where` clauses.

[1] Miranda is a trade mark of Research Software Ltd.
Correspondence and offprint requests to: `S.J.Thompson@ukc.ac.uk`

The relation of the logic to the language itself is examined in Section 4. We argue that a subset of the logic provides an operational semantics of the language, and argue that this logic is, by construction, consistent. Similar arguments suggest that the full logic is also consistent.

Large portions of the logic have been implemented in Isabelle; we report on this work in Section 5. Work on the Isabelle implementation is proceeding with case studies which we hope will guide research into generating theories automatically from Miranda scripts and the design of tactics and tacticals to support reasoning about functional programs – preliminary work in this area appears to indicate that more tacticals to support forward reasoning from assumptions would be welcome.

Similar work on giving a logical formulation of Haskell is reported in [Tho93].

I am grateful to Howard Bowman, Stephen Hill, Richard Jones, Andy King, Mark Longley, Gerry Nelson and David Turner of the University of Kent, Norbert Völker of FernUniversitätHagen and the two anonymous reviewers for their helpful comments on this material and its presentation.

2. A Logic for Miranda – Summary

In this Section we use the notation '§n.m' for Section n.m of the original paper, [Tho89], reserving 'Section n.m' for sections of *this* paper.

In the original paper, §3 and §4, we present an axiomatisation of the Miranda language, based on the atomic formulas

`e` $\equiv$ `f`

where `e` and `f` are Miranda expressions; the statement is intended to assert that the expressions `e` and `f` have the same value. In addition, the logic contains the formulas `e` $\sqsubseteq$ `f`, interpreted as 'the value of `e` approximates that of `f`'; these formulas can be defined from the equality relation $\equiv$ by induction over the construction of the types, and we say nothing further about them.

The atomic formulas are embedded in a many-sorted first-order logic, the sorts of the logic corresponding to the types of Miranda. The connectives are the standard ones: conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$), negation ($\neg$), bi-implication ($\Leftrightarrow$) and the typed universal ($\forall$) and existential ($\exists$) quantifiers.

The logic is standard first-order logic with equality ($\equiv$). Because the language contains unrestricted recursion, it is possible for the evaluation of an expression to fail to terminate. Each type is therefore taken to contain a term $\perp$ to denote the undefined value; this is broadly the approach taken in LCF, [Pau87]. §3 examines some of the alternative approaches available.

In keeping with the Miranda notation, we shall in general omit the types on quantifiers. We also assume that the type checking facilities of Miranda are employed in ensuring that only equations between objects of unifiable types are well-formed.

Axiomatising the language consists of two steps.

- §4, 8, 9 contain an axiomatisation of the *types* of the language.
- §5, 6, 7 give an axiomatisation of the *definitions* which form the programs of the language.

These are examined in turn now.

2.1. Types

A description of a type has two aspects. First, we need to describe equality over the type, $\equiv$. This is, in general, done by stating which elements are unequal as well as those which are equal. In the case of compound types, equality is based on comparing the components. The general case of an algebraic type is examined in §8. Taking as an example

`tree ::= Leaf num | Node tree tree`

we have two axioms to characterise equality:

$\texttt{(Leaf m} \equiv \texttt{Leaf m)} \Leftrightarrow \texttt{(m} \equiv \texttt{n)}$

$\texttt{(Node s1 s2} \equiv \texttt{Node t1 t2)} \Leftrightarrow \texttt{((s1} \equiv \texttt{t1)} \wedge \texttt{(s2} \equiv \texttt{t2))}$

and one to give the inequalities between the different constructors:

$\texttt{(Leaf n} \not\equiv \texttt{Node t1 t2)} \wedge \texttt{(Leaf n} \not\equiv \bot\texttt{)} \wedge \texttt{(Node t1 t2} \not\equiv \bot\texttt{)}$

Similar observations apply to products in §4.4.2, 8.2.

Finally in considering equality, we look at functions. Equality between functions is *extensional*: two functions are equal if and only if they give the same results on all possible arguments; this is discussed in §4.4.3.

The second aspect of types to examine is how to prove *universal* results over a type. Clearly we can use the standard rule of universal introduction: from a proof of `P(x)` we can infer $\forall$`x.P(x)`, so long as `x` is not free in any of the assumptions, but structured types carry *induction* rules which characterise the way in which they are generated.

The induction rules for the boolean type are introduced in §4.2:

$$\frac{\texttt{P(True)}\quad \texttt{P(False)}\quad \texttt{P(}\bot\texttt{)}}{\forall\texttt{x::bool.P(x)}} \qquad \frac{\texttt{P(True)}\quad \texttt{P(False)}}{\forall_{\texttt{df}}\texttt{x::bool.P(x)}}$$

The first characterises the full domain, and the second the sub-domain of *defined* elements. To eliminate the quantifier $\forall_{\texttt{df}}$, it must be instantiated with a defined element, and the logic contains a definition of this; for the Boolean type, it is simply $\texttt{defined(x)} \Leftrightarrow \texttt{x} \not\equiv \bot$.

This approach is generalised to structured types such as lists in §9.1, where rules characterising various sub-domains of the full type of lists are given.

For example, the *finite* lists, whose structure is finite and fully defined but whose elements may be undefined (for example $\texttt{[2,}\bot\texttt{,3]}$) are characterised by the rule

$$\frac{\texttt{P([])}\quad \forall\texttt{a.}\forall\texttt{x.(P(x)}\Rightarrow\texttt{P(a:x))}}{\forall_{\texttt{fin}}\texttt{x.P(x)}}$$

As well as various degrees of defined list, the list types contain *infinite* elements. To prove a result for the full type of lists, including the infinite elements, requires more sophisticated methods, which are outlined in §9.3, together with a discussion of the equality over these objects.

Lists are taken as a paradigm for all (covariant) algebraic types in the paper. A similar approach to the numeric type `num` is discussed in §4.3.

2.2. Definitions

On first examining a Miranda function definition, it appears to be a set of equations; unfortunately, the situation is more complicated than that.

A definition consists of a number of equations; when a function is applied, the first equation whose patterns match the given arguments will be applied. Within a particular equation there can be multiple right-hand sides, which are searched through sequentially. §5 discusses the simple case of a single equation, while §6.1 gives a thorough examination of *pattern matching* upon which we build in the remainder of this paper. Miranda pattern matching is sequential, proceeding top-down and left to right.

The crucial definition in §6.1 is of the *complement* of a pattern, `p` say. This is a set of a patterns, some accompanied by guards, which characterise the cases in which a value *fails* to match the pattern `p`, and so when control 'falls thorough' to the next equation, if any.

The explanation is given by defining a Miranda type to represent patterns, and a Miranda function over that type to calculate complements. As an example, the pattern `[a,a]` is examined. Its complement will consist of

- `[]`, the empty list;
- `[a]`, a one-element list;
- `(a:b:x)`, a list with a least two elements *also satisfying* the guard `a~=b`; and
- `(a:b:c:x)`, such that the guard `a=b` holds; a list of at least three elements, the first two of which have to be equal, in other words.

This example can also be used to illustrate the fact that repeated variables in patterns cannot simply be replaced by distinct copies of the variable which are tested for equality in a guard: different complements are derived.

Finally, in §7, 10 there are discussions of the characterisation of recursion and related issues, to be taken up further in the present paper.

3. Miranda definitions

This Section builds on §5 and 6 of [Tho89], where a discussion of the translation of Miranda definitions is initiated. New to this exposition are a full description of pattern matching in the context of a sequence of equations, as well as a treatment of local definitions, introduced in `where` clauses.

Miranda definitions are, to the casual reader, equations. This section outlines a transformation from definitions to logical formulas and shows that a number of features combine to make the explanation complex. These features include the following.

- Definitions consist of a number of equations, each of which can have multiple right-hand sides. These are to be understood sequentially: equations and clauses within equations are tried one by one until an applicable case is found.
- Pattern matching is complex. Patterns may contain literals and repeated variables, as well as nested patterns. The process of pattern matching is sequential. Pattern matching has three outcomes: success, failure and divergence.
- Conditionals are expressed in Miranda by means of boolean guards. It is possible to write an equation without a final `otherwise` case, which can have the effect of allowing control to 'fall through' to the following equation.
- Definitions contain local bindings in `where` clauses; in some circumstances, such as the 'fall through' mentioned above, it is necessary to combine together

the local definitions of distinct equations when giving a logical explanation of their meaning.

- Lazy evaluation of constructor expressions has the consequence that algebraic types contain partially defined elements – that is elements which are defined, at the top level, at least, yet which contain undefined components. Describing pattern matching against such objects is tricky.

As was noted in Section 2, we shall use the notion of the *complement* of a pattern in explaining the behaviour of a function defined by means of a sequence of equations. The translation will be given in a number of stages, and will be illustrated by a series of examples. Lists are used as a typical example of a type, with constructors cons, `':'`, and the empty list, `[]`.

3.1. Single Equations

The simplest form of definition is

`f x = e`

where `x` is a variable. This translates to the formula

$\forall$`x.(f x` $\equiv$ `e)`

This is the case for any number of pattern arguments on the left hand side (LHS), so long as no variables are repeated on the LHS. For example,

$\mathtt{f}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_n\ \mathtt{=}\ \mathtt{e}$

is translated to

$\forall \mathtt{x}_1 \ldots \mathtt{x}_k.(\mathtt{f}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_n \equiv \mathtt{e})$

where it is assumed that the variables appearing in the patterns $\mathtt{p}_1$ to $\mathtt{p}_n$ are $\mathtt{x}_1$ to $\mathtt{x}_k$.

Note that throughout this account '...' is used to elide parts of the text – usually a sequence of patterns or the right hand side of a definition.

If, in a definition consisting of a single equation, one or more variables are repeated on the LHS then the effect is the same as an equality test in a guard.

$\mathtt{f}\ \mathtt{p}_1\ \ldots\ \mathtt{p}_n\ \mathtt{=}\ \mathtt{e}$

will have the same behaviour as the definition

$\mathtt{f}\ \mathtt{p}_1\mathtt{'}\ \ldots\ \mathtt{p}_n\mathtt{'}\ \mathtt{=}\ \mathtt{e'}\qquad \mathtt{,\ if\ guard} \qquad\qquad (*)$

where multiple occurrences of a variable, `x` say, are replaced by different subscripted variables, $\mathtt{x}_1$ to $\mathtt{x}_k$ to give the primed versions of the patterns. The right hand side, `e'` results from replacing a repeated variable, `x` say, by one of its subscripted instances.

In the `guard` we have the tests

$\mathtt{x}_1\mathtt{=}\mathtt{x}_2\ \&\ \mathtt{x}_1\mathtt{=}\mathtt{x}_3\ \&\ \ldots\ \&\ \mathtt{x}_1\mathtt{=}\mathtt{x}_k\ \&\ \ldots$

This in turn is translated by

$\forall \mathtt{x}_1 \ldots (\mathtt{guard} \equiv \mathtt{True} \Rightarrow\ \mathtt{f}\ \mathtt{p}_1\mathtt{'}\ \ldots\ \mathtt{p}_n\mathtt{'} \equiv \mathtt{e'})$

where the variable list $\mathtt{x}_1\ldots$ consists of all the variables free in the patterns $\mathtt{p}_1\mathtt{'}$ to $\mathtt{p}_n\mathtt{'}$.

3.2. Single equations; multiple clauses

A single equation may have multiple clauses on the right-hand side

```
f p = e1    , if g1
    = e2    , if g2
    = ...
    = en    , if gn
```

(The case that the equation has a final `otherwise` clause is equivalent to the final guard, g_n being replaced by `True`.)

In evaluating the function, the guards are evaluated in turn until one is found which returns the value `True`. In translation, we have

$$(g_1 \equiv \texttt{True} \Rightarrow \texttt{f p} \equiv e_1) \wedge$$
$$(g_1 \equiv \texttt{False} \Rightarrow g_2 \equiv \texttt{True} \Rightarrow \texttt{f p} \equiv e_2) \wedge \ldots \wedge$$
$$(g_1 \equiv \texttt{False} \Rightarrow g_2 \equiv \texttt{False} \Rightarrow \ldots \Rightarrow g_n \equiv \texttt{True} \Rightarrow \texttt{f p} \equiv e_n)$$

where we assume that implication, $\Rightarrow$, is right associative, so that

$$A \Rightarrow B \Rightarrow C$$

is shorthand for

$$A \Rightarrow (B \Rightarrow C)$$

If there are repeated variables in the pattern `p` then guarding clauses like that seen above in `(*)` need to be added to each conjunct before translation, thus:

```
f p' = e1'    , if guard & g1'
     = e2'    , if guard & g2'
     = ...
     = en'    , if guard & gn'
```

where the primed versions of the pattern, guards and result expressions result from replacing the repeated variables by their subscripted variants.

Each of the guards g_i' can evaluate to `True`, `False` or can diverge. In the latter case, the result of the function will be undefined, $\perp$. These divergent cases form part of the description of the function, and are discussed in Section 3.9.

3.3. Definitions and recursion

Operationally, a recursive definition of an object results in the *least* well-defined solution of the definition. The philosophy of the work reported here is only to reason about the properties common to all solutions of a particular equation. For instance, if

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

then the definition is translated thus:

$$(\texttt{n=0}) \equiv \texttt{True} \Rightarrow \texttt{fac n} \equiv \texttt{1} \wedge$$
$$(\texttt{n=0}) \equiv \texttt{False} \Rightarrow \texttt{fac n} \equiv \texttt{n * fac (n-1)}$$

Nothing definite can be inferred about the value of `fac` on negative integers, for instance. The least solution has the property of being $\perp$ on these values, but it would also be possible for them all to be zero.

3.4. Local definitions

The philosophy of treating equations as defining the properties common to *all* solutions of a set of equations has an effect on the treatment of local definitions. Consider

```
b = a
    where
    a = a
```

If the local definition, call it `P(a)`, is satisfied by a unique value, then the whole definition can be rendered either as

$$\forall\mathtt{a.(P(a)} \Rightarrow \mathtt{b} \equiv \mathtt{a)} \qquad (\dagger)$$

or thus

$$\exists\mathtt{a.(P(a)} \wedge \mathtt{b} \equiv \mathtt{a)} \qquad (\ddagger)$$

In the case that there are multiple solutions to `P`, the equations have different interpretations. In (†), `b` is set equal to every solution of `P`, thus generating a logical inconsistency, that is a formula from which we can prove every formula. In (‡) `b` is set equal to some (unknown) value — we can therefore deduce nothing about `a` beyond the fact that it has the property `P`, in line with our philosophy of explanation.

The example

```
f x y = e1
        where
        g x = e2
        h   = e3
```

if we make the inner definitions visible, translates to

$$(\forall\mathtt{x.(g\ x} \equiv \mathtt{e_2)} \wedge \mathtt{h} \equiv \mathtt{e_3} \wedge \mathtt{f\ x\ y} \equiv \mathtt{e_1)}$$

The variables used locally (in this case the `x` used in the definition of `g`) are hidden using the universal quantifier. At the outer level the remaining variables are similarly quantified and names are hidden by existential quantification thus:

$$\forall\mathtt{x,y.}\exists\mathtt{g,h.(}\forall\mathtt{x.(g\ x} \equiv \mathtt{e_2)} \wedge \mathtt{h} \equiv \mathtt{e_3} \wedge \mathtt{f\ x\ y} \equiv \mathtt{e_1)}$$

Since the scope of the variables `x` and `y` is the whole of the right hand side of the definition, including the right hand sides of the local definitions, the order of the quantifiers allows the values of `h` and `g` to be different for different values of `x` and `y`, which is how it should be.

In the general case,

```
f p = e1   , if g1
    = e2   , if g2
    = ...
    = en   , if gn
      where
      l1 ... = ...
      ...
      lq ... = ...
```

the translation will be

$$\begin{array}{l}
\forall\mathtt{x_1 \ldots x_k.} \\
\quad \exists\mathtt{l_1 \ldots l_q.} \\
\quad\quad \mathtt{(trans(l_1)} \wedge \ldots \wedge \mathtt{trans(l_q)} \wedge \\
\quad\quad\ \mathtt{(g_1} \equiv \mathtt{True} \Rightarrow \mathtt{f\ p} \equiv \mathtt{e_1)} \wedge \\
\quad\quad\ \mathtt{(g_1} \equiv \mathtt{False} \Rightarrow \mathtt{g_2} \equiv \mathtt{True} \Rightarrow \mathtt{f\ p} \equiv \mathtt{e_2)} \wedge \ldots \wedge \\
\quad\quad\ \mathtt{(g_1} \equiv \mathtt{False} \Rightarrow \mathtt{g_2} \equiv \mathtt{False} \Rightarrow \ldots \Rightarrow \mathtt{g_n} \equiv \mathtt{True} \Rightarrow \mathtt{f\ p} \equiv \mathtt{e_n))}
\end{array}$$

where $\mathtt{trans(l_i)}$ is the translation of the local definition of $\mathtt{l_i}$ according to the same rules as for top-level definitions. The variables $\mathtt{x_1}$ to $\mathtt{x_k}$ are precisely the variables free in the pattern `p`. The case of repeated variables in `p` is treated as above.

Another way of justifying the use of an existential quantifier to give the local scope is to examine the traditional existential elimination rule in natural deduction systems:

$$\frac{\exists \mathtt{x.P(x)} \quad \mathtt{P(y)} \vdash \mathtt{C}}{\mathtt{C}}(\exists E)$$

where `c` and any other assumptions of the derivation do not contain the variable `y` free. To infer `C`, we have to use an *arbitrary* `y` with the property `P`. That is precisely the effect we wish to achieve with our definition.

Local definitions and guards

It is worth examining one of the consequences of the translation of local definitions above.

The local definitions are given in an existential block, containing within it the case switch. For example,

```
f p = e1   , if g1
    = e2   , otherwise
      where
      b x = e
```

becomes

```
∀x1...xk.
    ∃b.( ∀x.(b x ≡ e) ∧
            (g1 ≡ True ⇒ f p ≡ e1) ∧
            (g1 ≡ False ⇒ f p ≡ e2) )
```

Note that the case switch is *within* the scope of the quantifier, so that the same value of `b` is used in both cases. This is stronger than the formula

```
∀x1...xk.
     ∃b.( ∀x.(b x ≡ e) ∧
            (g1 ≡ True ⇒ f p ≡ e1) ) ∧
     ∃b.( ∀x.(b x ≡ e) ∧
            (g1 ≡ False ⇒ f p ≡ e2) )
```

in which different values of `b` may potentially be used in the two clauses. This latter rendering is strictly weaker then the former, and is not used.

3.5. Multiple Equations

In explaining pattern matching for a single equation, it is sufficient to describe the set of values for which the pattern match succeeds. In general an attempt to match a value against a pattern can have one of three results: success, failure and divergence. For instance, matching the list `[2,⊥]` against `(a:x)` will succeed, whilst against `[a]` it will fail, and against `[a,a]` it will give divergence, since it will involve evaluating `2=⊥`.

Given two equations, the second will only be applied to values which *fail* to match against the pattern in the first equation. We call this set the *complement* of the first pattern, and we argued in Section 2 that it could be described by a finite set of patterns, perhaps with accompanying guards. The patterns, right-hand sides and guards are specialised as a result of *unifying* the patterns with the complements of earlier patterns. Section 2 described the complement of the pattern `[a,a]`, and so the definition

```
f [a,a] = e1
f y     = e2
```

will be translated thus:

$$\begin{array}{l} \forall \mathtt{a,b,c,x,z}. \\ \quad \begin{array}{llll} (\mathtt{a=a}) \equiv \mathtt{True} & \Rightarrow \mathtt{f\ [a,a]} & \equiv \mathtt{e}_1 & \wedge \\ & \mathtt{f\ []} & \equiv \mathtt{e}_2\{\mathtt{[]/y}\} & \wedge \\ & \mathtt{f\ [a]} & \equiv \mathtt{e}_2\{\mathtt{[a]/y}\} & \wedge \\ (\mathtt{a=b}) \equiv \mathtt{False} & \Rightarrow \mathtt{f\ (a:b:x)} & \equiv \mathtt{e}_2\{\mathtt{(a:b:x)/y}\} & \wedge \\ (\mathtt{a=b}) \equiv \mathtt{True} & \Rightarrow \mathtt{f\ (a:b:c:z)} & \equiv \mathtt{e}_2\{\mathtt{(a:b:c:z)/y}\} & \end{array} \end{array}$$

(It is assumed that none of `a,b,c,x,z` is free in $\mathtt{e}_2$. If this is not the case then the appropriate re-namings should first be done.) In this example, the unification with `y` is trivial. In general, the equations will be rendered thus:

A similar explanation holds for equations with multiple patterns on the left hand side: the patterns are matched left to right. Occurrences of numeric literals in patterns are treated in similar way to repeated variables: the equality tests they give rise to are performed during the course of pattern matching, and so definitions which contain numeric literals cannot in general be replaced by definitions containing only variables with an auxiliary equality test on the variables in a guard.

When a sequence of equations like

$$\begin{array}{l} \mathtt{f}\ \mathtt{p}_1 = \mathtt{RHS}_1 \\ \mathtt{f}\ \mathtt{p}_2 = \mathtt{RHS}_2 \\ \mathtt{....} \\ \mathtt{f}\ \mathtt{p}_\mathtt{k} = \mathtt{RHS}_\mathtt{k} \end{array}$$

is to be translated, the second equation is only invoked on expressions which match both the complement of $\mathtt{p}_1$, written $\sim\mathtt{p}_1$, and $\mathtt{p}_2$. We need therefore to unify the two, written $\mathtt{p}_2\ \&\ \sim\mathtt{p}_1$. (As was evident above, the complement of a pattern may be described by a set of patterns rather than a single pattern; in that case $\mathtt{p}_2$ has to be unified with each of the patterns in $\sim\mathtt{p}_1$.) The result in the case above is

$$\begin{array}{l} \mathtt{f}\ \mathtt{p}_1 = \mathtt{e}_1 \\ \mathtt{f}\ (\mathtt{p}_2\ \&\ \sim\mathtt{p}_1) = \mathtt{RHS}_2\sigma_1 \\ \mathtt{f}\ (\mathtt{p}_\mathtt{k}\ \&\ \sim\mathtt{p}_1\ \&\ldots\&\ \sim\mathtt{p}_{\mathtt{k-1}}) = \mathtt{RHS}_\mathtt{k}\sigma_{\mathtt{k-1}} \end{array}$$

where $\sigma_\mathtt{i}$ is the substitution unifying $\sim\mathtt{p}_1$ up to $\sim\mathtt{p}_\mathtt{i}$ with $\mathtt{p}_{\mathtt{i+1}}$. Each of these definitions can then be translated separately using the methods above, *assuming that the final guard in each equation is* `otherwise` *or the equation is without guards.*

An example of this is given by the definition

```
f []      (a:b:x) = e1
f (a:b:x) y       = e2
f x       (a:y)   = e3
```

which gives rise to

$$\begin{array}{l} \forall \mathtt{a,b,c,x,y}. \\ \quad \begin{array}{llll} \mathtt{f\ []} & \mathtt{(a:b:x)} & \equiv \mathtt{e}_1 & \wedge \\ \mathtt{f\ (a:b:x)} & \mathtt{y} & \equiv \mathtt{e}_2 & \wedge \\ \mathtt{f\ []} & \mathtt{[a]} & \equiv \mathtt{e}_3\{\mathtt{[]/x,[]/y}\} & \wedge \\ \mathtt{f\ [c]} & \mathtt{(a:y)} & \equiv \mathtt{e}_3\{\mathtt{[c]/x}\} & \end{array} \end{array}$$

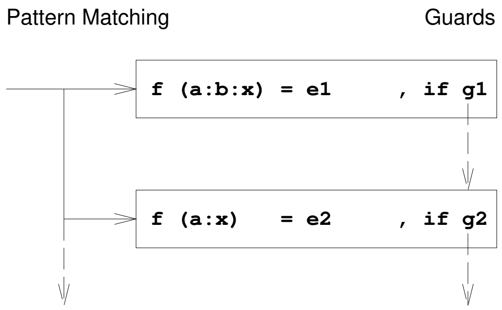
In these examples it has been evident that substitutions have to be performed on the right hand sides of equations. This remark applies equally well to guards and local definitions (`where` clauses).

3.6. Patterns and Guards: Dangling conditions

The guards of Miranda do not necessarily include an `otherwise` clause. It is therefore possible for a pattern match to succeed, only for (all) the guard(s) in an equation to fail, causing control to 'drop through' to the following equation. This mechanism is orthogonal to the pattern matching which can also cause control to pass to an subsequent equation. The example of

```
f (a:b:x) = e1        , if g1
f (a:x)   = e2        , if g2
```

can be illustrated thus:



In its translation there are *three* clauses. The first corresponds to a successful match with the first equation, followed by a `True` result of the guard; the second to the fall through from the first equation to the second, and the final clause to use of the second equation after failure to match with the first:

```
∀a,b,x.
    g1  ≡ True  ⇒ f (a:b:x) ≡ e1 ∧
    g1  ≡ False ∧ g2' ≡ True
                 ⇒ f (a:b:x) ≡ e2' ∧
    g2" ≡ True  ⇒ f [c] ≡ e2"
```

where

```
    g2' ≡ g2{(b:x)/x}
    g2" ≡ g2{c/a,[]/x}
```

and `e2'` and `e2"` are defined similarly.

Local Definitions

The difficulty of explanation is compounded by the presence of `where` clauses. For example,

```
f (a:x) y = u    , if h b
            where
            b = g a x y
f x (b:y) = v    , if w
            where
            a = s t
...
```

Control will fall through to the second equation from the first if the guard, which involves the locally defined value `b`, fails. In explaining the effect of the second equation, therefore, we will need to use this value `b`. It is as though the scope of `b` is extended to include the other defining equations of `f`. To make the explanation clearer, the variables in the second equation are renamed thus:

```
f x' (b':y') = v'   , if w'
                where
                a' = s' t'
```

The translation is then

```
∀a,b,x,y.
  ∃b.(b ≡ g a x y ∧                                    (0)
        (h b ≡ True ⇒                                  (1)
           f (a:x) y ≡ u) ∧                            (2)
        (h b ≡ False ⇒                                 (3)
           (∃x',b',y',a'.                              (4)
             ((b':y')≡y ∧ x'≡(a:x) ∧ a' ≡ (s' t') ∧    (5)
               ((w' ≡ True ⇒ f x' (b':y') ≡ v') ∧      (6)
                ... ) ) ) ) ) ∧                        (7)
  ∃a.(a ≡ (s t){[]/x} ∧                                (8)
         (w{[]/x} ≡ True ⇒ f [] (b:y) ≡ v{[]/x}) ∧     (9)
         ... ) ∧                                       (10)
  ...                                                  (11)
```

The first conjunct (lines `(0)` to `(7)`) denotes successful matching with the patterns in the first equation.

Within this conjunct, the first conjunct (lines `(1)` and `(2)`) denotes the case in which the first equation applies; the second (lines `(3)` to `(7)`) covers that in which control falls through to the second equation.

In this case, a collection of existentially quantified variables is introduced in line `(4)`; these are used to represent a successful match with the patterns in the second equation (line `(5)`), as well as the definition local to the second equation (line `(5)`). Line `(6)` contains the guard from the second equation, and the result of successful evaluation of the guard. The ellipsis in line `(7)` denotes the case in which controls falls through to the third and subsequent equations from the first and second.

The second conjunct (lines `(8)` to `(10)`) covers the case of a successful match with the second equation after failure to match the first; the ellipsis in line `(10)` allows for control to fall to the third. The final ellipsis in line `(11)` covers the case of a successful match with the third equation (after failure to match the first two).

The example shown here is representative of the most complex possible form of definitions in Miranda. Schematically, a sequence of equations will be translated as in Figure 1.

As a result of our work in this area, we would propose that the `otherwise` clause in Miranda and its analogue in Haskell, [HJE92], become compulsory – this means that once a pattern is matched, there is commitment to that equation, and that there is no need for the joining together of local definitions which makes the operation of the construct substantially more complicated than it need be. In Standard ML, [MTH90], this problem does not occur: alternatives are constructed using `if...then...else...` with no option to drop the `else` case.

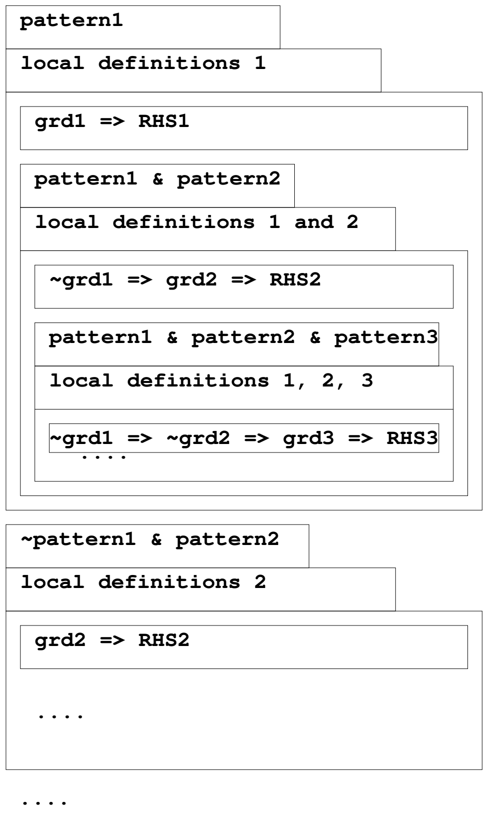


Fig. 1. Translating a sequence of equations: the general case.

3.7. Built-in functions

A number of the functions available in the Miranda system are built-in:

- The equality and ordering functions, which are defined over all ground types, that is types built from the atomic types `num`, `bool` and `char` using (covariant) algebraic type definitions, tuples and lists.
- The arithmetical functions, defined over the type `num`.
- The `show` functions, which provide printable (`[char]`) representations of objects.

The values of these functions are assumed to be given as *axioms*, so that for all `m` and `n` of type `num`, the equation

$$\mathtt{m+n} \equiv \mathtt{v}_{\mathtt{m+n}}$$

will be an axiom, where $\mathtt{v}_{\mathtt{m+n}}$ is the value of the sum of `m` and `n`.

3.8. List Comprehensions

The syntax used to describe lists in Miranda is rich. Definitions using a variety of patterns are permitted, and there is also the list comprehension notation. Intuitively, a list comprehension allows a declarative description of a list, thus

```
[ e | g1 ; g2 ; ... ; gn ]
```

where each g_i is either a *generator* of the form

```
pattern <- expression
```

or a `boolean` expression. The generators bind values to the variables in the pattern, successively binding the pattern to the values in the list denoted by the expression.

The boolean expressions are tests which filter the values chosen, so that those making the test `False` are omitted. The result of the evaluation is the list of `e`'s resulting from the successive bindings. For example,

```
[ f x | x <- l ; g x ] ≡ map f (filter g l)
[ x+y | x <- [0,1] ; y <- [4,2] ] ≡ [4,2,5,3]
```

The following axioms suffice to describe list comprehensions.

```
[ e | ] ≡ [e]
[ e | ... ; True ; ... ] ≡ [ e | ... ; ... ]
[ e | ... ; False ; ... ] ≡ []
[ e | ... ; x <- [] ; ... ] ≡ []
[ e | x <- (a:l) ; ... ]
      ≡ [ e | ... ]{a/x} ++ [ e | x <- l ; ... ]
```

In case a pattern appears in a generator, it is first translated into a simpler form. Suppose we have the generator

```
(a:x) <- l
```

make the auxiliary definitions

```
patt1 (a:x) = True
aPart (a:x) = a
xPart (a:x) = x
```

replace the generator by

```
y <- filter patt1 l
```

and replace the identifiers `a` and `x` by

```
(aPart y)        (xPart y)
```

in the remainder of the list comprehension. The translation is simpler in the case of an irrefutable pattern, such as `(x,y)`, there being no need to filter the values matching the pattern.

Miranda contains one final form of generator, which can be described by a source-to-source transformation. It is possible to write

```
x <- c, f x ..
```

this is explained in exactly the same way as the

```
x <- iterate f c
```

where `iterate f c` is the list `[c, f c, f (f c), ...]`.

3.9. Divergence information

There is further information to be gleaned from the equations about cases in which a function is undefined; this can happen because pattern matching or guard evaluation diverges.

```
True & x  = x
```

`False & x = False`

has the property that

`⊥ & x = ⊥`

since the pattern match in the first argument forces the conjunction operation to examine its first argument.

The information about this form of divergence can be derived from the definitions in a similar way to the calculation of complements above, if it is required.

3.10. Conclusion

This Section has given a translation of Miranda definitions into conditional equations of the form

$$\texttt{cond}_1 \Rightarrow \ldots \Rightarrow \texttt{cond}_k \Rightarrow \texttt{f}\ \texttt{p}_1\ \ldots\ \texttt{p}_m \equiv \texttt{v} \qquad (\dagger)$$

and more complex logical formulas in which scopes are represented by existentially quantified variables.

As is well-known (see, for example, [Pey87]) local definitions can be removed by a process of *lambda lifting*, in which a local definition of `e` say, depending upon formal parameters $\texttt{x}_1$ to $\texttt{x}_k$, is *lifted* to a top-level definition of a function of these parameters. Occurrences of `e` within its original scope are then replaced by applications

$$\texttt{e}\ \texttt{x}_1\ \ldots\ \texttt{x}_k$$

Different definitions of `e` in different scopes will have to be renamed, of course. If the process of lambda lifting is applied to a script, an equivalent script results. Moreover, the translations of a script without local definitions will consist only of conditional equations like (†). We shall use this fact below.

It should also be observed that the translation is *monotonic*, in the sense that if an extra equation is added to a definition, the translation of the augmented definition will *extend* the translation of the original. This is a consequence of the approach of Section 3.3, where we noted that we only observe properties common to all functions satisfying the defining equations, rather than the least solution. (Note also that this is true of the divergence information we infer in Section 3.9. We only axiomatise the cases in which pattern matching diverges, which will not be altered by the addition of extra equations.)

4. The status of this logic

Section 3 gives a translation of Miranda definitions into a logic; together with the rules for types in [Tho89] and reviewed in Section 2 we have a system for reasoning about Miranda programs. In this Section we look at the status of the logic, and in particular explore the twin questions of its consistency and its relation to Miranda.

4.1. An Operational Semantics for Miranda

In the absence of a published semantics for the Miranda programming language it is impossible to assess the relative soundness of the logic proposed here.

On the other hand we suggest that the system is *itself* an extension of an operational semantics for Miranda. First we make a definition.

A Miranda expression is a *value* if it contains only literals and constructors. For instance,

- the values of type `num` are the numeric literals,
- the values of `[num]` are finite lists built from numeric literals, and so on.

We claim that a Miranda expression `e` of ground type will have the value `v` if and only if

$\vdash_0 \ \mathtt{e} \equiv \mathtt{v}$

where the deduction system $\vdash_0$ consists of

- the elimination rules for $\exists$ (as given above, in Section 3.4), $\forall$ and $\Rightarrow$,

$$\frac{\forall \mathtt{x.P(x)}}{\mathtt{P(e)}}(\forall E) \qquad \frac{\mathtt{A}\Rightarrow\mathtt{B} \quad \mathtt{A}}{\mathtt{B}}(\Rightarrow E)$$

  together with
- the structural rules governing assumptions,
- the axioms stating that $\equiv$ is a congruence, and,
- the translations of the definitions of the Miranda definitions into quantified formulas built over conditional equations of the form (†), as described in Section 3.

The explanation above gives directly only values to expressions denoting hereditarily fully-defined objects; Miranda as a lazy language contains other values also. These are defined indirectly by equating two expressions $\mathtt{e}_1$ and $\mathtt{e}_2$, $\mathtt{e}_1 \cong \mathtt{e}_2$, if and only if for all contexts, `C[..]`, and values `v`,

$\vdash_0 \ \mathtt{C[e_1]} \equiv \mathtt{v}$ if and only if $\vdash_0 \ \mathtt{C[e_2]} \equiv \mathtt{v}$

4.2. Consistency of the operational semantics

As was explained in Section 3.10, without loss of generality we can consider scripts without local definitions, since the lambda-lifting transformation preserves meaning. Another way of looking at this is to see the lambda lifted definitions as *Skolem functions* witnessing the existential quantifiers in the translations of general formulas.

In this case, the translations are (the universal closures of) conditional equations of the form

$\mathtt{cond}_1 \Rightarrow \ldots \Rightarrow \mathtt{cond}_k \Rightarrow \mathtt{f} \ \mathtt{p}_1 \ \ldots \ \mathtt{p}_m \equiv \mathtt{v}$ (†)

We now claim that the set of formulas produced by the translation will be consistent *by construction*. This is because the algorithm for translation ensures that no two conditional equations for a single defined object have both

- compatible conditions — that is have the union of their sets of conditions a consistent set of formulas, and
- unifiable patterns.

As a consequence of this property, it apparent that no non-trivial equations between values can arise as a consequence of the translations of definitions, and so that in particular,

$\nvdash_0 \ 0 \equiv 1$

which is a formal statement of the consistency of the system $\vdash_0$. Another way of seeing this is to use the confluence of the corresponding semi-equational conditional term rewriting system; see, for instance [Klo93].

4.3. Consistency of the full logic

The full logic consists of the translations of Section 3 together with the rules which characterise the types, reviewed in Section 2. The induction rules, and so on, are derived from domain models in [Pau87], and so are consistent. The addition of the rules derived from the definitions should not compromise consistency for the same reasons as discussed in the previous Section.

5. Implementation in Isabelle

Isabelle is a generic proof assistant with some support for automatic theorem proving, and which supports proof in a variety of logics. We have used the version of first-order many-sorted logic in Isabelle to implement the logic for Miranda.

Isabelle supports goal-directed backwards proof construction. Inference rules are matched against goal states using higher-order unification, and in the main we have applied rules one by one when constructing proofs. The system supports tactics, and in our work we have used some of the built-in tactics: `safe-tac` applies a number of simple logical strategies, without causing any instantiation of logical variables; `fast-tac` will do instantiations as well. These tactics are often used to strip off universal quantifiers, move hypotheses from implication formulas into assumption lists, and so on. Also built into the system is a rewriting package, which uses logical equations and equivalences as directed rewriting rules. Application of tactics such as `SIMP_TAC` and `ASM_SIMP_TAC` which recursively apply all the available rewriting rules can substantially simplify proof search.

In reasoning about functional programs, it is often the case that certain conclusions are sought on the basis of equational assumptions. At certain points of proof, forward reasoning from these assumptions is required, and it took a certain amount of time (and advice) to discover how best to do this. In particular it is sometimes necessary to cut in the appropriate substitution instance of a known fact to guide search or variable instantiation.

The translation of Miranda scripts is made easier by a number of features of Isabelle.

- The types (or sorts) allowed in the system can belong to classes, similar to the type classes of Haskell. The class `mira` is defined to represent the class of Miranda types, and allows a natural representation of Miranda polymorphism. This is assisted by the ability to give type variables a default class, which we choose to be `mira` in most cases.
- Classes also give a clean treatment of the overloaded operations of Miranda, like the computational equality operation, which returns a `bool`ean result. This is denoted by `=` in Miranda; we use `=:=` for the relation in Isabelle, since `=` is used for identity, which we have denoted by $\equiv$ thus far in the paper. We also use overloading to define a predicate `def`, for the fully-defined elements of each type.
- Operators can be given mixfix syntax, so that the built-in syntax of the

operations can be made to resemble that of the application domain. Declaring values thus:

```
undef   :: "'a::mira"                     ("'_|'_")
":"     :: "['a,'a list] => 'a list"      (infixr 90)
member  :: "'a list => 'a => bool"        ("member _ _")
```

ensures that _|_ is the polymorphic undefined element, cons is denoted by an infix colon and that applications of `member` are written without brackets, as in Miranda. More complex syntactic declarations are easily available.

A selection of the rules for booleans and lists follow:

```
    disBool1        " ~ (true = false) "
    disBool2        " ~ (true = _|_) "
    disBool3        " ~ (false = _|_) "
```

These three axioms ensure that the three boolean values are distinct. The axioms are preceded by their names, which are used when writing proofs. Note that lower case `true` and `false` are used; the upper-case versions denote the true and false propositions.

```
boolInd   "[| P(false) ; P(true) ; P(_|_) |] ==> ALL x.P(x)"
```

This is the induction rule for booleans, from which can be proved

```
"ALL x . x=true |x=false|x=_|_"
```

The disjunction operation is axiomatised thus

```
orTrue          "true \\/ x = true"
orFalse         "false \\/ x = x"
orUndef         "_|_ \\/ x = _|_"
```

A typical list function is `++`

```
    appendNil       "[]++x    = x"
    appendUndef     "_|_++x   = _|_"
    appendCons      "(a:x)++y = a:(x++y)"
```

The list induction axiom is only to be applied to chain complete predicates `P`, see [Pau87] for details.

```
    listInd
    "[| P([]) ; P(_|_) ; ALL a x . P(x) --> P(a:x) |]
          ==> ALL x.P(x)"
```

Here can be seen a part of the definition of the `def` predicate, for list types. The rules for `def` are formulated as equivalences, so that they can be input to the simplification system.

```
    defNil          "def([]) <-> True"
    defUndef        "def(_|_) <-> False"
    defCons         "def(a:x) <-> def(a) & def(x)"
```

As an experiment in verification using functions involving local definitions, patterns and local definitions, we performed verification for the Miranda function

```
frontsubst :: [*] -> [*] -> [*] -> ( [*] , bool )
```

The function `frontsubst x y z` replaces the initial sublist `x` of `z` by `y`, if possible. In this case the result if paired with `True`. If the substitution fails, the unmodified string is paired with `False` to give the result.

```
frontsubst [] rep st       = ( rep++st , True )
frontsubst (a:x) rep []    = ( [] , False )
frontsubst (a:x) rep (b:y)
  = ( b:y , False ) , if a ~= b \/ ~ok
  = ( out , True )  , otherwise
    where (out,ok) = frontsubst x rep y
```

The function is translated thus:

```
frontsubst :: "['a list,'a list,'a list] => ('a list * bool)"
                ("frontsubst _ _ _" [110,110,110] 100)
```

The syntax declaration here ensures that the arguments to the function are bracketed in exactly the cases that they are bracketed in Miranda.

```
fs1     "frontsubst [] rep st       = ( rep++st , true )"
fs2     "frontsubst (a:x) rep []    = ( [] , false )"
```

The final case translates into a block, containing an inner case analysis

```
fsBlock
"EX out ok .( (out,ok) = frontsubst x rep y
    & ( not (a =:= b) \\/ not ok = true
        --> frontsubst (a:x) rep (b:y) = ( b:y , false ) )
    & ( not (a =:= b) \\/ not ok = false
        --> frontsubst (a:x) rep (b:y) = ( out , true ) )
    & ( not (a =:= b) \\/ not ok = _|_
        --> frontsubst (a:x) rep (b:y) = _|_ ))"
```

The goal is one half of the bi-implication which expresses that the function performs the substitution required:

```
"ALL x y z ans .
      def(x) --> def(y) --> def(z) --> def(ans) -->
      (frontsubst x y z = (ans,true) -->
      (EX w. x++w=z & y++w=ans))";
```

and this is proved in a proof of some 100 steps. Many of these steps are small, and could be put together by defining the appropriate tactics. In outline, the proof proceeds by induction over the first variable, `x`. Within the induction step there is a case analysis on the value `z`. Now a case analysis on the value of the guard can be used. The interesting case (`False`) requires the proof of two subsidiary lemmas.

In constructing a proof of this size, it is imperative to keep a textual log of the steps taken so that this can be modified and re-played at will. How best to *document* such proofs will form part of our future research, since if the proofs are to be re-used, at least partially, they need to be comprehensible.

References

[HJE92] Paul Hudak, Simon Peyton Jones, and Philip Wadler (Editors). Report on the Programming Language Haskell, version 1.2. *ACM SIGPLAN Notices*, 27(5), 1992.

[Klo93] J. W. Klop. Term rewriting systems. In Samson Abramsky, Tom Maibaum, and Dov Gabbay, editors, *Handbook of Logic in Computer Science, Volume II*. Oxford University Press, 1993.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Pau87] Lawrence C. Paulson. *Logic and Computation — Interactive proof with Cambridge LCF*. Cambridge University Press, 1987.

[Pau90] Lawrence C. Paulson. Isabelle: the next 700 theorem provers. In P. Oddifreddi, editor, *Logic and Computer Science*. Academic Press, 1990.

[Pey87] Simon Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International, 1987.

[Tho89] Simon J. Thompson. A Logic for Miranda. *Formal Aspects of Computing*, 1, 1989.

[Tho93] Simon J. Thompson. Formulating Haskell. In *Workshop on Functional Programming, Ayr, 1992*, Workshops in Computing. Springer Verlag, 1993.

[Tur90] David A. Turner. An overview of Miranda. In David A. Turner, editor, *Research Topics in Functional Programming*. Addison Wesley, 1990.