Miranda in Isabelle

Steve Hill & Simon Thompson
Computing Laboratory
University of Kent at Canterbury
{S.A.Hill,S.J.Thompson}@ukc.ac.uk

July 26, 1995

Abstract

This paper describes our experience in formalising arguments about the Miranda functional programming language in Isabelle. After explaining some of the problems of reasoning about Miranda, we explain our two different approaches to encoding Miranda in Isabelle. We conclude by discussing some shorter examples and a case study of reasoning about hardware.

Miranda[1][Turner, 1990, Thompson, 1995b] is a modern functional programming language, allowing type polymorphism and higher-order functions in a similar way to ML[Milner *et al.*, 1990]. It differs from ML in being lazy — arguments to functions are only evaluated when and to the extent that they are needed — and in being side-effect free. It has long been an article of faith in the functional programming community that languages like this are ideal candidates for program verification because of their 'declarative' nature. This is clearly true for idealised languages, but real languages like Miranda bring their own complexities which we have discussed in the past[Thompson, 1989, Thompson, 1995a].

In this paper we discuss our approaches to formalising proof about Miranda in Isabelle, specifically Isabelle92, after a brief description of the language and how it is given a logical description.

1 Miranda

In this section we give a short survey of the main features of Miranda, and how we translate the definitions into logical statements. Full details of a translation can be found in [Thompson, 1989, Thompson, 1995a].

Equations

The simplest definitions in Miranda resemble equations. In defining a constant function we say

```
con :: * -> num                                              (1)
con x = 14
```

where the `*` is a type variable in the type of the function, indicating that the function can be given an argument of any type, and `=` is used to give a definition. If we write $\equiv$ for logical equality, then `con e` $\equiv$ `14` for *any* expression `e`, including an expression whose value is undefined. We therefore translate the definition `(1)` by the equation

```
con x ≡ 14
```

[1] Miranda is a trademark of Research Software Limited

Sequencing

In general Miranda definitions are more complex than the equation we have just seen. The definitions are written in sequence, and this ordering is significant.

We distinguish between different cases using guards, giving an equation multiple clauses on the right hand side. For instance in comparing two numeric lists element by element we might say

```
compare :: [num] -> [num] -> [num]

compare (a:x) (b:y) = a : compare x y   , if a <= b                (2)
                    = b : compare x y   , if True
```

The second right hand side does not hold in *all* circumstances, despite having the guard `True`; the clause applies only if the first guard is `False`. In general a clause applies only if all the preceding guards are `False`. Logically we have

((a<=b)≡True ⇒ compare (a:x) (b:y) ≡ a : compare x y) ∧
((a<=b)≡False ⇒ compare (a:x) (b:y) ≡ b : compare x y)

A definition may well consist of more than one equation; `(2)` only applies when the two list arguments to `compare` are non-nil. We complete the definition by giving the result in case either list is empty:

```
compare x y = []                                                   (3)
```

This equation will only apply if the preceding equations *fail* to apply; in terms of patterns, it only applies to the *complement* of the preceding patterns. In this case, we have

compare [] y ≡ [] ∧
compare (a:x) [] ≡ []

When combined with complex guards and repeated variables in expressions, the translation of definitions can become complex; we give a complete treatment in [Thompson, 1995a].

Local definitions

Each equation can carry with it a collection of local definitions, whose scope is restricted to the right hand side of the equation. For example, to substitute at the front of a list we can write

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

so that

```
frontSubst "cat" "dog" "catalyst" = ("dogalyst",True)
frontSubst "bat" "dog" "catalyst" = ("catalyst",False)
```

The first element of the result is got by substituting the second argument (`"dog"`) for the first argument (`"cat"`) when it occurs at the front of the third (`"catalyst"`); the second component of the result is a flag signalling whether the substitution has been successful. The definition of the function follows:

```
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 local definition of `(out,ok)` is used to make a recursive call to `frontSubst` and to select its components.

In translating the final equation, we introduce the locally defined objects by means of an existential quantifier and so translate it thus:

```
(∀a,x,rep,b,y).(∃out,ok).
  (out,ok) ≡ frontSubst x rep y ∧
  ( (a ~= b \/ ~ok) ≡ True ⇒
      frontSubst (a:x) rep (b:y) ≡ ( b:y , False ) ∧
    (a ~= b \/ ~ok) ≡ False ⇒
      frontSubst (a:x) rep (b:y) ≡ ( out , True ) )
```

The order of the quantifiers in the logical translation shows that `out` and `ok` depend on the parameters of the function `a`, `x`, `rep`, mi b and `y` as would be expected.

When local definitions combine with the sequential features above, translation becomes complicated; see [Thompson, 1995a] for further details.

Types

Miranda types include characters, booleans, numbers (integers and floats combined into a single type) and algebraic types. Because Miranda is a lazy language, the structured types (like lists) contain partial elements such as `[2,⊥,3]` and 'infinite' objects defined as follows:

```
ones = 1 : ones
primes = sieve [2..]
         where
         sieve (a:x) = a : sieve [ b | b<-x ; b mod a > 0 ]
```

and we therefore have to be careful in stating the exact rules for induction over algebraic types. Details of the various approaches can be found in [Paulson, 1987].

2 Miranda in Isabelle

We have given a translation of Miranda into Isabelle92, and in this section we comment on how the translation uses some of the features of the system.

- The Miranda logic is defined to be an extension of first-order logic; Miranda functions are taken to be Isabelle functions. This has some advantages: type checking and other facilities are inherited from Isabelle, but also drawbacks which we come to presently.
- Miranda is a polymorphic language. We have an Isabelle class `mira` which is defined to represent the class of Miranda types. We are also assisted by being able to declare types as belonging to a default class, in this case the class `mira`.
- Miranda also contains some built-in overloaded operations, in particular the boolean operations of equality, ordering and so on as well as the printing functions. The classes are again useful here; for example we define `=:=` for the Miranda equality operation in Isabelle, since `=` is used for identity, which we have denoted by ≡ thus far in the paper. We also use overloading to define a predicate `def`, for the fully-defined elements of each type.
- The syntax of Miranda differs from that of Isabelle. Function application is denoted by juxtaposition, with function application binding most tightly among the operations. We use the mixfix facility to give expressions the same appearance that they have in Miranda. For instance, in translating the `frontSubst` function we declare

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

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

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

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 ) )                    (4)
    & ( 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) = _|_ ))"
```

Figure 1: Translation of the function `frontSubst`

The full translation of the `frontSubst` function appears in Figure 1.

Note from figure 1 that some minor syntactic changes have to be made. We use the prefix type constructor `list` rather than the Miranda square brackets, and we have to use `true` and `false` for the Boolean constants since their capitalised counterparts are used for the valid and contradictory propositions.

What are the drawbacks of this approach? Principally, we are unable to reflect the fact that Miranda functions are *curried*, so that a function of two (or more) arguments like

```
mult :: num -> num -> num
mult a b = a*b
```

can legitimately be given a single argument, returning a function:

```
mult 34 :: num -> num
```

To model these partial applications in Isabelle, we need to write a lambda term

```
λ b. mult 34 b
```

which is rather more unwieldy than the original.

A Second Approach

In this section we explore a second approach to coding Miranda in Isabelle.

The Basic Theory

For the final case-study, an alternative approach was adopted addressing the concerns regarding curried functions. Again, the theory is based on the theory of first-order logic provided as a standard component of the Isabelle system. In a departure from the previous study, a new type constructor and constant `app` are introduced to support the Miranda function space.

```
types
  "->"    2   (infixr 50)
arities
```

```
  "->" :: (mira,mira)mira
consts
  app :: "[('a -> 'b),'a] => 'b" ("_ _" [100,101] 100)
```

This facilitates reasoning about higher-order Miranda terms. For example the rule for extensional equality might be couched as:

```
ALL x. f x = g x ==> f = g
```

However, one drawback (with Isabelle92) is that the parser is not able in all circumstances to parse function applications correctly. In particular, if a function is applied to an expression in parentheses, the parse will fail. To circumvent this problem, an explicit infix application operator, denoted `$` is provided and must be inserted in all places where the problem would arise. A parse-translator converts these to the standard application operator, so they never appear in printed terms.

For convenience, each built-in Miranda operator is described via two Isabelle constants. For example, for function composition we have:

```
  "."     :: "[('a->'b),('c->'a)] => 'c->'b"      (infixl 70)
  Dot_op  :: "('a->'b) -> ('c->'a) -> ('c->'b)"   ("'(.')")
```

The former allows expressions to be written in the familiar Miranda syntax, whereas the second can be used if it is ever necessary to reason with a curried operator. Two rules are given: the first defines the operator, and the second relates the two constants:

```
    comp    "(f . g) x = f $ (g x)"
    Dot_op  "(.) f g = f . g"
```

The core theory is extended to provide support for the fundamental Miranda datatypes. With each new type we introduce:

- a type constructor,
- constants representing the constructors,
- a set of standard functions,
- proof rules, including rules for definedness and uniqueness,
- where appropriate, rules defining a computational equality.

For example, the theory of lists, given in Figure 2, defines:

- the type constructor `list`,
- the constructors : and `[]`,
- standard functions `hd`, `tl`, `++` and `map`,
- an induction rule for lists (the rule presented is only sound for chain-complete predicates); rules for definedness of lists; rules asserting the uniqueness of the constructors,
- a definition of computational equality for lists.

Other standard theories include the following:

- Booleans: `true`, `false`, `cond` the usual operators and computational equality,
- Natural numbers: `succ`, `zero`, `+`, `1`, `2` etc.,

```
types
    list 1
arities
    list :: (mira)mira
consts
    ":"  :: "['a, 'a list] => 'a list"   (infixr 52)
    nil  :: "'a list"                    ("[]")
    hd   :: "'a list -> 'a"
    tl   :: "'a list -> 'a list"
    "++" :: "['a list, 'a list] => 'a list" (infixr 52)
    map  :: "('a -> 'b) -> 'a list -> 'b list"
rules
    listInd
        "[| ALL a x. P(x) --> P(a:x); P([]); P(_|_) |] ==>
         ALL x::'a list.P(x)"
    nilCons "[] = (a:x)  <-> False"
    nilBot  "[]    = _|_ <-> False"
    consBot "(a:x) = _|_ <-> False"
    defNil  "def([]) <-> True"
    defCons "def(a:x) <-> def(a) & def(x)"
    eqList0 "[] === [] = true"
    eqList1 "(a:x) === [] = false"
    eqList2 "[] === (b:y) = false"
    eqList3 "(a:x) === (b:y) = a === b && x === y"
    eqList4 "_|_ === y = _|_"
    eqList5 "x === _|_ = _|_"
    hd0     "hd []    = _|_"
    hd1     "hd $ (a:x) = a"
    hd2     "hd _|_   = _|_"
    tl0     "tl []    = _|_"
    tl1     "tl $ (a:x) = x"
    tl2     "tl _|_   = _|_"
    conc0   "[] ++ y    = y"
    conc1   "(a:x) ++ y = a : (x ++ y)"
    conc2   "_|_ ++ y   = _|_"
    map0    "map f []      = []"
    map1    "map f $ (a:x) = f a : map f x"
    map2    "map f _|_     = _|_"
```

Figure 2: The Theory of Lists

- Tuples (currently up to 6-tuples),
- Association lists derived from the theory of lists.

The theory of natural numbers represents our first departure from the Miranda system. No attempt has been made to account for the Miranda `num` type which is a conflation of arbitrary precision integers and floating point numbers.

Translation to Isabelle

In this exercise, no attempt has been made to address the whole of the Miranda language. In particular, the following restrictions have been introduced:

- definitions are restricted to non-overlapping patterns,
- guards must be converted to conditional expressions,
- local definitions must be lifted to the top level.

These restrictions are intended to bring the language closer to a logic. In particular definitions can be converted directly to equations without the complications described in the previous section. Extra rules covering the case of undefined arguments are required for functions that perform pattern matching.

Algebraic types are translated according to the scheme described for lists. Synonym and abstract data types seem to be most conveniently represented as a one-constructor type in Isabelle. An alternative scheme would have been to expand synonyms, but in practice this leads to an unwieldy theory.

The translation of function type signatures is straightforward simply requiring the replacement of Miranda's star notation for type variables with Isabelle's more conventional identifier names, for example:

```
consts
    id      :: "'a -> 'a"
    const   :: "'a -> 'b -> 'a"
    apply   :: "('a -> 'b) -> 'a -> 'b"
```

Currently, the translation process is done by hand. However, the method is entirely mechanical and could be automated.

3 Examples

We have developed a series of smaller examples, and a larger case study which we explore in the next section. In developing these smaller examples, we have often had to develop supporting libraries of proofs concerning the behaviour of elementary operations over simple data types. As we look at the examples we make some observations about our approach and the Isabelle system.

The second and third of the examples here were developed in collaboration with Gerald Nelson of the University of Kent.

Substitution

We chose the `frontSubst` function as an example since has many of the features of Miranda definitions, including pattern matching, guards and a `where` clause. We can specify one aspect of its behaviour in a high-level way, thus:

```
"ALL x y z ans .
      def(x) --> def(y) --> def(z) --> def(ans) -->                    (5)
      (frontsubst x y z = (ans,true) -->
      (EX w. x++w=z & y++w=ans))";
```

and we chose to prove this using Isabelle. The proof takes some 100 elementary steps, and proceeds by induction over finite lists. Much of the proof involves reasoning *forward* from comparatively large sets of assumptions. Some of the assumptions come from stripping off the definedness hypotheses from `(5)`, and others from opening up the existentially quantified formula in Figure 1,`(4)`.

Using these assumptions and a case analysis on the result of comparing elements under the ordering we apply *modus ponens* to close this assumption set. It was our experience that this needed hand guidance, and that we would find it difficult using the available tools to automate this 'closure under modus ponens' as a tactic. Clearly this would be desirable to support larger-scale proof development in a context like this.

Sorting

We have automated a proof of correctness of insertion sort,

```
sort []     = []
sort (a:x) = insert a (sort x)

insert a []     = [a]
insert a (b:x) = a:b:x           , if a<=b
               = b : insert a x  , otherwise
```

by proving the following two propositions for all finite lists `x`

```
sorted (sort x) ≡ True
perm (x , sort x) ≡ True
```

where `sorted` expresses the fact that its argument is sorted, and `perm` the fact that its arguments are permutations of each other. The proof proceeds by induction at the top level, but also uses some twenty lemmas about elementary properties of orderings. We also have to introduce a function `smallest` which takes the smallest element of a list, and many of the lemmas involve proving simple properties of this function.

Simulation

Our third case study concerns a simulation of a bank, in which on arrival customers are placed in a single queue. A customer goes to a clerk when the clerk becomes free. Our proof shows that increasing the number of clerks will reduce the total waiting time of the customers, if it is initially non-zero. Details of the simulation can be found in Chapter 13 of [Thompson, 1995b].

The proof involves manipulating sums of lists of numbers; as in the previous example, it was necessary to develop a substantial foundation in order to build the required proof.

We also tried to prove that under a round-robin scheduling mechanism the total waiting time was reduced, but discovered that this was not the case. Increasing the number of clerks in this case can increase the total waiting time. The scenario in which this happens is when there are two customers requiring a long time to be served; with two queues they are allocated to one server, whereas with three they will be allocated to different servers, this means that they delay more people, and so increase the overall delay.

4 Case study – Hardware Description

This case study describes an experiment in verifying the refinement of a processor description/simulation written in Miranda which is described more fully in [Hill, 1994]. The subject of this experiment is a step in the design of a simple microprocessor. There are two executable descriptions of the machine addressing two levels of abstraction. The aim of the verification is to show that these two descriptions behave in essentially the same way.

The first machine, dubbed `m0`, has the following components:

- a memory – implemented as an association between locations and contents
- a register set – implemented as an association between register number and contents
- a statistics field
- a halt flag

The operation of the machine is described by transitions from one state to another, similar in style to that used in [Peyton Jones, 1992]. For example the transition that reads data from memory into a register is given by:

```
memToReg :: regnum -> regnum -> m0 -> m0
memToReg rs rd (m, r, s, h)
  = (m,
     aBind rd (aLookup (aLookup rs r) m) r,
     s, h)
```

The second machine, dubbed `m1`, is more explicit about some of the internal structure. Its state is given by the following:

- a memory – as in `m0`,
- a memory interface – implemented as a pair of values corresponding to a Memory Data Register (MDR) and Memory Address Register (MAR).
- a register set – as in `m0`,
- a set of four buses – implemented as a quadruple of values,
- a statistics field,
- a halt flag.

The transitions of this machine are more restricted. Data must pass from a register to a bus (A or B), and thence via the ALU to another bus (C). Data on the C-bus may be placed in a register or into the memory interface which is the only route to the memory. So, a typical transition might be:

```
regToAbus n (m, i, r, (a, b, c, d), s, h) =
    (m, i, r, (aLookup n r, b, c, d), s, h)
```

The machine is depicted in Figure 3.

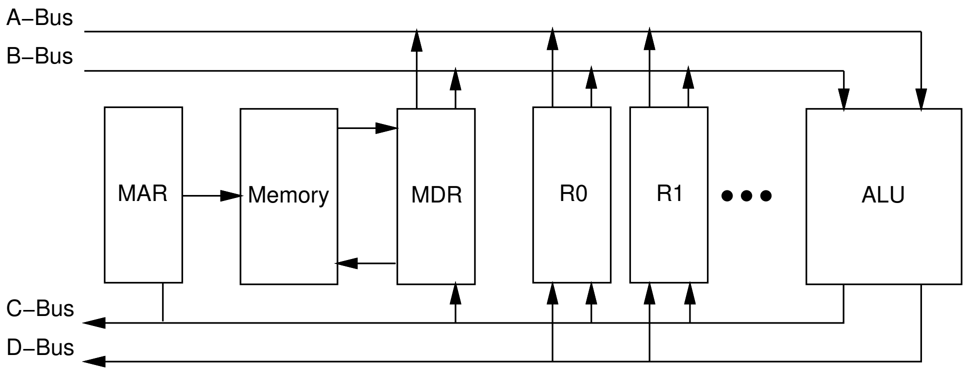

Figure 3: Machine Architecture

Both machines define three combinators to construct compound transitions from the basic operations.

- `comma` – combines two transitions sequentially; a derived combinator `do` takes a list of transitions and combines them sequentially.
- `switch` – selects a transition according to the contents of a register
- `passReg` – passes the contents of a register to a transition

The behaviour of the full machine is ultimately implemented in terms of the basic transitions. The simulation has been used to describe a simple register machine, a memory-based stack machine and a register-based stack machine. The register machine is the subject of this verification, and supports an instruction set of some eight instructions supporting four addressing modes. A flavour of the implementation is given in Figure 4.

Verification

The aim of the verification is to show that `m1` is a faithful refinement of `m0`. The relationship that we wish to hold is depicted in Figure 5 and is rendered in Isabelle as:

```
t1 refines t0 ==
  (ALL m. spec (t0 m) =m1 t1 (spec m))
spec (m0 $ (m,r,s,h)) =
  m1 (m, _|_, r, _|_, _|_, h)
m1 $ (mm0,i0,rr0,b0,s0,h0) =m1
m1 $ (mm1,i1,rr1,b1,s1,h1) <->
  mm0 = mm1 & rr0 = rr1 & h0 = h1"
```

The specialisation function `spec` takes an `m0` state and creates an `m1` state, placing undefined values in the new fields. The predicate `=m1` tests for equality of the `m0` components of two `m1` states.

The following simple example shows that the halt instruction in `m1` is indeed a refinement of the equivalent `m0` transition. It gives a flavour of the style of the goal directed proofs within this framework.

```
goal Machine01.thy "halt1 refines halt0";
by (rewrite_goals_tac [refines]);
br m0 1;
```

```
fetch
 = do [
     regToMar pc,
     memRead,
     mdrToReg ir,
     op1 pc AluIncA pc
   ]
execute
 = switch ir [
     (moveW,   moveI),
     (addW,   addI),
     (subW,   subI),
     (jumpW,   jumpI),
     (jumpeqW, jumpeqI),
     (jsrW,   jsrI),
     (rtsW,   rtsI),
     (haltW,   haltI)
   ]
moveI
 = do [
     srcOpTo tmp1,
     dbusToReg ccr,
     destOpFrom tmp1
   ]
srcOpTo r
 = do [
     fetch,
     switch ir
     [
         (litW, do [fetch, regToReg ir r]),
         (absW, do [fetch, regToMar ir, memRead, mdrToReg r]),
         (regW, do [fetch, passReg ir ((flip regToReg) r)]),
         (indW, do [fetch, passReg ir regToMar, memRead, mdrToReg r])
     ]
   ]
```

Figure 4: Sample of Machine Implementation

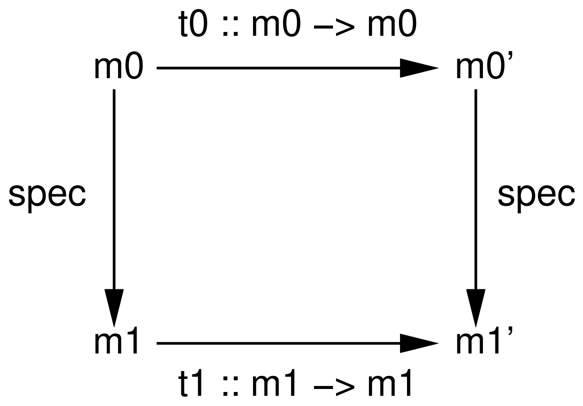

Figure 5: Refinement

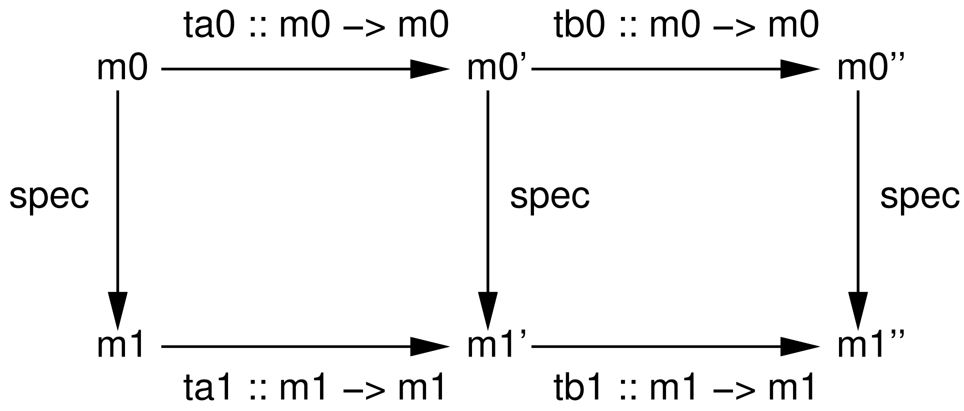

Figure 6: Composing Transitions

```
br tup4Ind 1;
by (REPEAT (SIMP_TAC machine01_ss 1));
```

The first step expands the definition of `refines`. The next two steps apply the appropriate rule for reasoning about the type of the leading quantified variable. Finally, the simplification rules are sufficient to complete the proof.

Figure 6 shows the situation when two transitions are composed using the combinator `comma`. The corresponding theorem is:

```
ALL ta0 tb0 ta1 tb1.
  ta1 refines ta0 --> tb1 refines tb0 --> respect tb1 -->
  (comma1 ta1 tb1) refines (comma0 ta0 tb0)
```

that is, if we have two transitions in the `m1` world that are refinements of transitions in the `m0` world, then the composition of the two should be a refinement of the composition in the `m0` world. We have one further hypothesis which is required to ensure that the functions are well-behaved which is formulated as:

```
respect t1 == (ALL x y. x =m1 y --> t1 x =m1 t1 y)
```

This guarantees that the second machine can make no distinctions between states on the basis of information held in the fields not present in `m0`.

The proof of the composition theorem makes use of the first-order logic theorem prover and an elimination tactic, but follows broadly the same shape as the halt proof given earlier:

```
goal Machine01.thy
"ALL ta0 tb0 ta1 tb1.
   ta1 refines ta0 --> tb1 refines tb0 --> respect tb1 -->
   (comma1 ta1 tb1) refines (comma0 ta0 tb0)";
by (rewrite_goals_tac [refines, respect]);
by (MACH_TAC [comma0, comma1] 1);
by (step_tac FOL_cs 1);
by (REPEAT (etac allE 1));
br eq1_trans 1;
be impE 2;
by (REPEAT (assume_tac 1));
```

The work has reached a point where the proof is nearly complete. The proof is modular, and in most parts quite tedious. Fortunately, most proofs follow a simple pattern which can almost be cut and pasted to produce the next one.

The rewriting tactics of Isabelle are sufficiently powerful, that many of the larger proofs could be conducted for the most part automatically. However, Isabelle is too slow on our systems too make this practical. Instead the proof is decomposed into smaller elements which are combined using tools such as the composition theorem. This also has the benefit of offering theorems which might be re-used later in the verification.

During this exercise, it has proved useful to provide some tactics to support rewriting. These are built on top of the existing rewriting mechanism. The proof of the composition theorem uses one of these called `MACH_TAC` defined as:

```
fun MACH_TAC rules
  = SIMP_TAC (holmAll_ss addcongs machine01_congs
               addrews rules);
```

It is useful when a proof requires rewriting with only a small number of equations. A similar tactic is provided to use the rules in the opposite direction:

```
fun RMACH_TAC rules
  = MACH_TAC (map (fn r => r RS sym) rules);
```

More work could be done in this area, for example to select induction principles according to the type of a quantified variable.

5 Conclusions

Our experiments with Isabelle have proved to be successful. The proof of the machines' equivalence is almost complete. The extension of Isabelle to support reasoning about Miranda was straightforward, requiring little expert assistance. The support for an extensible parser/unparser makes it possible to express terms using the Miranda syntax directly. This makes the process of translation less prone to error, and makes the job of the verifier simpler since the theorems and goals are presented in a familiar style. In the case study, there were few creative steps in the proofs. Much of the "hard" work is managed by use of the FOL theorem prover and the simplification package.

The marriage, however, is not perfect. We found that the treatment of synonyms was somewhat clumsy and would advocate the inclusion of some sort of type synonym mechanism in Isabelle. In some cases, there seems to be a tension between the object and meta levels. Often, in different parts of a proof a theorem may be needed in either the FOL form, or in the meta form. Whilst there is no technical difficulty in moving between the two, there can be a confusing increase in the number of theorems.

Reasoning about Miranda programs often involves very simple rewriting of terms. Although the simplification mechanism provides tools to support rewriting, it is sometimes not possible to obtain the desired effect, for example rewriting one instance of a pattern whilst leaving a second. There is scope here for development of our own tactics.

We were gratified that the experiment showed that even if one were neither an experienced verifier nor logician one would still be able to render a significant Miranda script as an Isabelle theory and to construct a reasonably large proof.

However, in our case study we have dealt with a quite small subset of the Miranda language and have chosen a regular and flat problem. The proofs to-date are mostly goal directed, with little cause for forwards reasoning. This was exercised more in the substitution example, which raised the issue of how to 'close up' a set of hypotheses under deduction.

Isabelle's generality makes our experiments possible, but can also make Miranda-specific reasoning more complex that one might hope of a tailor made tool. It is to be hoped that appropriate tactics should bridge the gap.

References

[Hill, 1994] Steve Hill. The functional simulation of a simple microprocessor. Technical Report 17-94, UKC Computing Laboratory, 1994. Available by ftp from `unix.hensa.ac.uk` in the directory `pub/misc/ukc.reports/comp.sci/reports` as the file `17-94.ps.Z`.

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

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

[Peyton Jones, 1992] Simon L. Peyton Jones. *Implementing Functional Languages*. Prentice-Hall, 1992.

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

[Thompson, 1995a] Simon J. Thompson. A Logic for Miranda, Revisited. *Formal Aspects of Computing*, to appear, 1995.

[Thompson, 1995b] Simon J. Thompson. *Miranda The Craft of Functional Programming*. Addison-Wesley, 1995.

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