J. Functional Programming 2 (4): 437–473, October 1992 © 1992 Cambridge University Press

MetaMorph – a formal methods toolkit with application to the design of digital hardware

P. J. BRUMFITT

Logica Cambridge Ltd., Betjeman House, 104 Hills Road, Cambridge, CB2 1LQ, UK

Abstract

MetaMorph is a software tool that supports transformation and proof for an equational non-strict functional language. It was developed as a vehicle for research into the synthesis of digital logic, but is equally suitable for reasoning about functional programs. The theorem prover may be used for verifying new reusable transforms and to assist the search for transformation sequences having a constrained goal. The paper provides an overview of all aspects of the project, and a brief discussion of its application to hardware.

Capsule Review

Digital circuit design is a natural application area for functional programming and formal methods, since debugging hardware is difficult and hardware errors are very costly to fix. MetaMorph is an interactive tool that helps a designer to develop functional circuit specifications and prove them correct.

MetaMorph contains a transformation tool and a theorem prover. The overall circuit development and proof is controlled by the user, with mundane details handled automatically. This approach seems more practical than 'post hoc verification', where a theorem prover attempts to verify a completed circuit design.

One of the most interesting features is the meta language, based on Prolog, for controlling the overall proof strategy. The user can paste the sequence of steps logged for a transformation into the meta language script, or edit the script directly. This increases the level of interaction between the user and the proof system.

Much of the paper is about transformation and proof techniques that are applicable to general functional programming problems. The later sections discuss the Macintosh implementation and experience with applying MetaMorph to circuit design.

1 Introduction

MetaMorph is a software tool for formal reasoning about functional programs, which was developed to support research into the design of digital hardware by transformation of specifications. However, the tool itself and many of the techniques could equally well be applied to development of software. MetaMorph brings together a number of techniques, such as transformation and theorem proving, in a novel way under the control of a graphical user interface and meta-language. The aim of this paper is to provide an overview of all aspects of the tools, including the motivation, theory, implementation and user interface, to show how these techniques have been combined. A brief discussion of hardware is included to illustrate the current application and to provide a context for discussing the results of practical work using the tool.

The work was motivated by an interest in finding better ways of applying formal methods to hardware design, since current approaches are so difficult to apply that their use is mainly confined to safety and security critical systems, where the cost can be justified. If such methods could be made easier to use they would also be of great benefit in commercial VLSI design.

One formal approach to hardware development is *verification*, which involves proving that a given circuit satisfies its specification. Unfortunately, the complexity of modern chips leads to proofs which are beyond the capabilities of current automatic theorem provers. The Boyer–Moore theorem prover for recursive function specifications (Boyer and Moore, 1979, 1988) has been used successfully for hardware design, but requires an increasing amount of guidance as proofs become more complex. Specification languages based on set theory (Spivey, 1988) and Higher-Order Logic (Gordon, 1985, 1986) allow more abstract specifications, but make automatic proof even more difficult; the proof tools typically support the user in decomposing a large proof, but require extensive guidance.

There is clearly an advantage in using the computer as design tool, so that the relationship between a circuit and its specification can be captured during the design process, in a form that can be used to guide a proof. Each step in the design process generates a proof obligation, rather than leaving one unmanageable proof until the end. Proof then becomes an integral part of the design process, rather than a *post hoc* attempt to verify a design once it has been completed.

In particular, a circuit may be derived from its specification, by a sequence of meaning-preserved *transformations*. There is an implicit proof if each transform has previously been verified and the transforms are applied by a tool which allows only correct manipulations. Transformation moves the emphasis from verifying individual designs to verifying the design process. Proofs are therefore about transforms, rather than about individual designs; the proof need be carried out only once and the proven transform can then be re-applied to other similar problems. Such transforms may form the basis of a library of reusable synthesis tools, each of which solves a particular class of problem.

Although transformation is a good approach to design, it is not ideal in situations where the goal is constrained in some way. For example, it may be necessary to express a circuit in terms of pre-defined building blocks, such as the components of a gate-array library, or to use particular instruction codes or state assignments to allow interfacing with other devices or to achieve upward compatibility with previous designs. When the circuit is highly constrained, it becomes difficult to guide the transformation towards the goal, because it is difficult to foresee all the implications of earlier steps. In these situations, proof may be more appropriate.

Proof and transformation involve essentially the same mathematical steps, but applied in a different order. For example, showing that two given expressions, A and

B, are equal involves a proof that A = B; on the other hand, if B is unknown, it may be derived from A by a sequence of transformation steps. The distinction between proof and transformation is therefore a little blurred; they may be thought of as the two ends of a spectrum, corresponding to B being completely constrained or completely unconstrained. Between these two extremes, there are interesting possibilities for a hybrid strategy, in which the result is *partially constrained.* For example, the user may guide the transformation process by specifying patterns which the result must match.

This view of proof and transformation, as the ends of a spectrum, allows the overall design process to follow a transformational style, whilst accommodating constraints on the solution by using proof on particular sub-problems. The lesser complexity of a sub-problem is often such that an automatic proof is entirely feasible.

MetaMorph is based on the philosophy outlined above. It supports a transformational design style, complemented by a theorem prover which can verify new transforms or tackle constrained sub-problems by proof. The use of partially-constrained transformations is an area of continuing research, but even a prototype implementation has led to a significant improvement in the ease and reusability of transformations.

2 The specification language

2.1 Introduction

MetaMorph provides the user with two languages. The *object* language serves the dual rôle of a specification language and a hardware description language. The *meta* language is used to manipulate the object language by performing transformations and proofs.

The object language is an equational polymorphically-typed functional language with pattern matching. It has been based on an extended subset of Miranda[1] (Turner, 1985*a*, *b*), so that the Miranda compiler can be used to execute specifications to allow experimentation at an early stage in the design cycle. It is similar to the Miranda-like language described by Bird and Wadler (1988) and enforces their constraint that definitions with multiple equations may not have overlapping patterns. The main extension to Miranda is to allow the definition of operators, so that a specification is self-contained with an explicit definition of all functions (except equality).

Specifications consist of definitions of types and functions over those types. For example, the type *stack* (of items of some generic type '$*$') and its operations *pop* and *top* may be specified as:

$$\begin{aligned}&\mathit{stack}\ * ::= \mathit{Newstack} \mid \mathit{Push}\ *\ (\mathit{stack}\ *);\\&\mathit{pop}\ (\mathit{Push}\ a\ x)\ =\ x;\\&\mathit{top}\ (\mathit{Push}\ a\ x)\ =\ a;\end{aligned}$$

This constructive style of specification is more restricted than specification languages based on set theory (Spivey, 1988) or predicate logic (Gordon, 1985, 1986). However, this makes it possible to *execute* specifications, to assist with their validation, and

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

makes proof easier to automate. Since the overall aim is not simply to produce elegant specifications, but to produce formally-proven designs from validated specifications, functional languages are considered to be a good compromise between elegance and practicality.

MetaMorph has a minimum of built-in types and functions. In particular, there are **no** built-in operators or functions, except the definition of equality. This approach results in specifications which are largely self-contained and make few assumptions about the semantics of the language. Algebraic types are predefined for lists, tuples and numbers solely so that special syntax can be used to represent them; other types (even booleans) are user defined.

The equality function is built-in so that it can be *overloaded* for different types (whereas user-defined functions may not) and so that the definition of equality can be inferred automatically for new types. Rather than force the user to accept this built-in definition, MetaMorph defines the function '*if_equal*', which the user may optionally use to define the ' = ' operator:

$$if\_equal :: * \to * \to ** \to ** \to **;$$
$$(x = y) = if\_equal\ x\ y\ True\ False;$$

The function *if_equal* returns its third or fourth argument, depending on the equality of the first two arguments, rather than returning a boolean value. This avoids the need to make *bool* a built-in type and allows the user to provide an alternative definition of *bool*.

2.2 *Defining functions*

Functions are defined as sets of equations such as:

$$map\ f\ [\,] = [\,];$$
$$map\ f\ (x{:}xs) = f\ x{:}map\ f\ xs;$$

Operators are defined similarly:

$$True \vee x\ = True;$$
$$False \vee x = x;$$

True and *False* are *constructors* which are introduced by a type definition; they may be thought of as constants which simply have no defining equations and therefore never get unfolded.

The precedences and associativities of operators are currently predefined, but include both Miranda and Bird-Wadler variants. For example, negation can be denoted by ' $\sim$ ' or ' $\neg$ '.

The original equations may be thought of as axioms, with all variables having implicit outermost universal quantification over their inferred polymorphic types; the logic has no existential quantification.

For example, the definition of ' $\vee$ ' is equivalent to the following axioms:

$$\forall \mathsf{x} :: \mathsf{bool}.\ \mathsf{True}\ \vee \mathsf{x} = \mathsf{True}$$
$$\forall \mathsf{x} :: \mathsf{bool}.\ \mathsf{False}\ \vee \mathsf{x} = \mathsf{x}.$$

The universal quantification is implemented by type checking. Each definition also generates an additional axiom for the inferred polymorphic type of the function:

$$\vee :: \mathsf{bool} \rightarrow \mathsf{bool} \rightarrow \mathsf{bool}.$$

Function definitions may be curried, allowing application one argument at a time. Constants are simply a special case of functions, with zero arguments.

When a function is defined by more than one equation, MetaMorph enforces Bird and Wadler's (1988) rule that the patterns must be non-overlapping. Each equation defines a partial function; when these partial functions are disjoint, they may be asserted as independent axioms. This allows individual equations to be used in transformation and proof, without considering whether any other equation applies.

For example, the following is valid in Miranda, but not in MetaMorph because it could be used to derive *True* = *False*:

$$nasty\ 0 \ = \ True;$$
$$nasty\ x \ = \ False;$$

Once a valid definition has been made, it is quite alright to transform it into overlapping equations, since their meanings must be the same when they overlap.

A number of tests are performed to ensure that each new function definition forms a conservative extension.[2] In particular, definitions must be well-formed (including correct typing), define a new constant which is not already defined and all variables in the right hand side must appear in parameters on the left hand side. For simplicity, variables may not be repeated on the left-hand side of a function definition.

Consider the definition

$$x \ = \ \neg x;$$

This is compiled into the following axioms:

$$\mathsf{x} :: \mathsf{bool}$$
$$\mathsf{x} \ = \ \neg \mathsf{x}.$$

Such a definition may lead to non-convergence, but cannot introduce a contradiction. In particular, the definition does **not** give rise to the axiom

$$\forall \mathsf{x} :: \mathsf{bool}.\mathsf{x} = \neg \mathsf{x}.$$

The identifier x in the definition simply has the value $\perp_{\text{bool}}$.

2.3 *Defining types*

MetaMorph supports algebraic type definitions, type synonyms and type specifications. Algebraic type definitions take the form

$$stack\ * ::= Newstack \mid Push\ *\ (stack\ *);$$

[2] A conservative extension is one in which every model of the specification up to the point of extension is also a model of the specification after the extension. This means that new definitions can be added, but extra constraints may not be placed on existing definitions.

This is compiled into the following axioms:

$$\text{Newstack} :: \text{stack} *$$

$$\text{Push} :: * \rightarrow (\text{stack} *) \rightarrow (\text{stack} *).$$

Tests are performed to ensure that the new type definition forms a conservative extension. In particular, the type constructors and type name must not have been defined previously and all the type variables in the type definition must appear as arguments.

The polymorphic type checker is implemented as a theorem prover for a formal system of type inference rules. Any correct typing, of the form $X :: \tau$, is then a theorem in that system. The theorem prover infers the type of each function (or constant) defined, so that explicit type declarations are not needed.

The formal system, which is derived from that of Cardelli (1987), has a single axiom schema:

$$X :: \tau \vdash X :: \tau$$

and a set of type inference rules. For example, the rule for a function application is

$$\frac{A \vdash f :: \sigma \rightarrow \tau \quad A \vdash x :: \sigma}{A \vdash f\,x :: \tau}.$$

The theorem prover infers the most general type for each function as a theorem. For the stack example given earlier we obtain

$$\vdash \text{top} :: (\text{stack} *) \rightarrow *$$

$$\vdash \text{pop} :: (\text{stack} *) \rightarrow (\text{stack} *).$$

The type of a function with multiple defining equations is obtained by unifying the types inferred for the partial functions given by each individual equation.

Type specifications may be used to constrain the type of a function to an instance of its inferred type. For example:

$$idn :: num \rightarrow num;$$

$$idn\ x\ =\ x;$$

This definition leads to the axiom

$$\forall x :: \text{num} \ .\ \text{idn}\ x = x$$

instead of the more general

$$\forall x :: * \ .\ \text{idn}\ x = x.$$

This is clearly valid, because any proposition p, which is true for all elements of a type t_1, must also be true for any subset t_2 of that type:

$$\forall x :: t_1 \ .\ p(x) \wedge t_2 \subseteq t_1 \Rightarrow \forall x :: t_2 \cdot p(x).$$

The type system is restricted to *shallow* types, these being ones in which quantification of the type variables occurs only at the outermost level. This restriction, which is common in functional languages (Milner, 1978), does not seem to be a significant practical limitation. However, there are certain definitions which cannot be typed, such as

$$f\, x = x\, x;$$

2.4 Numbers

MetaMorph has been designed to have a minimum of built-in types. However, some types must be built-in because they involve special syntax. Numbers are a special case which deserve further discussion.

Algebraic types in MetaMorph may be finite enumerated types or infinite recursive types. This allows a uniform approach to proof, involving case analysis and structural induction. An important consequence is that the natural numbers *cannot* be defined as an enumerated type with an infinite number of constructors (ie: $0, 1, 2, 3, \ldots$).

Natural numbers have a recursive type definition, which is equivalent to:[3]

$$num ::= \mathbf{O} \mid \mathbf{S}\ num;$$

where '**S**' is the successor function of Peano arithmetic.

MetaMorph converts between an external representation using numerals and an internal representation using successors, so that this is invisible to the user.

Functions on numbers may be defined as follows:

$$\begin{aligned} even\ 0 &= True; \\ even\ 1 &= False; \\ even\ (n+2) &= even\ n; \end{aligned}$$

This notation is compatible with both Miranda and Bird–Wadler. The definition is equivalent to:

$$\begin{aligned} even\ \mathbf{0} &= True; \\ even\ (\mathbf{S}\ \mathbf{0}) &= False; \\ even\ (\mathbf{S}(\mathbf{S}\ n)) &= even\ n; \end{aligned}$$

MetaMorph does not provide built-in definitions of even the most basic arithmetic operators. Addition and multiplication are normally defined by the user as follows:

$$\begin{aligned} 0 \quad &+ n = n; \\ (m+1) &+ n = (m+n) + 1; \\ 0 \quad &\times n = 0; \\ (m+1) &\times n = m \times n + n; \end{aligned}$$

The symbol '$+$' is overloaded, being both the addition operator and the successor

[3] Bold type is used as a reminder that '**O**' and '**S**' are not symbols of the concrete syntax.

constructor (when followed by a numeric constant). The definition of addition is therefore equivalent to:

$$\begin{aligned} \mathbf{0} \quad &+ n = n; \\ (\mathrm{S}\ m) &+ n = \mathrm{S}\ (m+n); \end{aligned}$$

A limitation of the successor representation is that it is only possible to use small constants in specifications and that evaluation of arithmetic expressions is very slow. A future enhancement could be to use integers internally, but to make them appear to the theorem prover as if they used the successor function.

The use of Peano arithmetic also has some impact on the semantics of the language. MetaMorph's type *num* is the natural numbers, whereas for Miranda (and Bird–Wadler) it is the integers. In particular, it is possible for MetaMorph to prove theorems of the form:

$$\forall n :: \mathrm{num}.p(n)$$

which are not true for negative numbers (i.e. under Miranda's definition of *num*).

Consequently, it is important to realise that the semantics of a specification relate to MetaMorph's definition of *num*. If such specifications are run as Miranda programs, they may terminate in situations where the specified value would be undefined in MetaMorph.

Another semantic difference arises from the fact that *num* is not a flat domain. For example, consider the following definition of the relational operator '>':

$$\begin{aligned} 0 \quad &> n &&= \mathit{False}; \\ (m+1) &> 0 &&= \mathit{True}; \\ (m+1) &> (n+1) &&= m > n; \end{aligned}$$

The expression '$\perp + 1 > 0$' has the value *True*, whereas in Miranda it is undefined.[4] Again, this is a minor point concerned with termination, but illustrates that the MetaMorph semantics should be treated as definitive when interpreting a specification.

Miranda's type '*num*' can represent integers and floating point numbers, whereas MetaMorph is restricted to natural numbers. There would be a problem with the semantics of equality if unwanted integer to floating-point type conversions could occur when running MetaMorph programs under Miranda. However, floating-point numbers can only be introduced in Miranda by using a floating-point constant or the division operator '/'. Since MetaMorph does not support either of these mechanisms, the problem cannot arise.

3 Transformation

3.1 Introduction

MetaMorph is implemented on the Apple Macintosh computer and makes use of an interactive style of working. A specification is initially entered in one or more source windows and then compiled into a set of axioms as described above. The specification

[4] $\perp$ is the undefined value, known as 'bottom' in denotational semantics.

may be extended by adding new definitions and compiling them, but the definitions of existing functions cannot be changed, except by transformation.

A function is transformed by first opening a *transform window*, which displays the type and defining equations for a single function; there may be any number of these open at once. Transform windows cannot be edited directly by typing into them or by cutting-and-pasting text. Transforms are applied by selecting an expression with the mouse (where appropriate) and then choosing a transform from a menu. Alternatively, the meta-language can be used to select expressions and apply transforms.

MetaMorph provides a few primitive transforms, such as folding and unfolding (Burstall and Darlington, 1977; Darlington, 1981), as built-in rules of inference. The user may build more powerful transforms either by writing meta-language programs, which invoke lower-level transforms, or by proving theorems which may be applied as transforms.

3.2 Folding and unfolding

Referential transparency allows an application of a function to be *unfolded* by replacing it with the body of the definition with the arguments substituted for the formal parameters. Conversely, a function may be introduced into an expression by *folding* the expression with the definition. In other words, a definition may be used as a left-to-right or right-to-left rewrite rule.

When definitions involve multiple equations, any equation may be used in isolation as a rewriting rule, because the equations must describe disjoint partial functions or else have been derived in such a way that they are equal where they overlap.

Type checking is needed to ensure that folding does not lead to erroneous results. For example, it is not permissible to fold the number '*7*' with the definition of the function '*pop*' as follows:

pop (*Push a x*) = *x*; || *original definitions*
y = 7;

y = *pop* (*Push a* 7) || *Invalid result of folding.*

This example also illustrates another problem with folding; it may introduce a new variable (in this case '*a*'). MetaMorph requires that such a variable is instantiated with a value of the correct type. The type constraints on folding are necessary to satisfy the quantifiers in the definition of '*pop*'

```
∀a::*.∀x::stack*.pop(Push a x) = x.
```

Unfolding does not require type-checking, since the type rules ensure that all sub-expressions of a correctly typed expression are also correctly typed. Also, unfolding cannot introduce new variables, because the definition rule requires that all variables in an equation must be introduced by its parameters.

Unfolding usually has a single solution, because function definitions must have non-overlapping equations, whereas there are many ways of folding a given expression. For example, *any* expression can be folded with the function

$$id\ x = x;$$

MetaMorph requires a pattern to be specified which is sufficient to uniquely distinguish the solution. This is normally the name of the function followed by meta-variables (denoted by β1, β2, ...) as its arguments. If more than one fold is possible, the user is prompted with a dialogue which allows a choice among them. Consequently, it is a good idea to ensure that only one fold is possible, when a fold appears in a meta-language script, as this further dialogue will interrupt the script, requiring user intervention.

An *evaluate* command is also provided, which causes a selected expression to be evaluated, using either a lazy or eager reduction strategy. Evaluation is simply equivalent to applying unfold repeatedly, until the expression can be evaluated no further.

3.3 Instantiation and enumeration

Instantiating a variable in an equation produces a new equation, which is a special instance of the original. The original equation is not deleted, since this would alter the domain of the function.

A related operation is *enumeration*, which replaces an equation by a set of equations with a specified variable replaced by terms built using each of the possible constructors for the type of the variable. Thus there are as many equations as constructors for the type and a single equation is replaced by a set of partial functions whose domains partition the original function domain.

For example, the definition

$$f\ x\ y = y : x;$$

may be enumerated with respect to x, as follows:

$$f\ [\,]\quad y = [y];$$
$$f\ (v1{:}v2)\ y = y : v1 : v2;$$

The cases are derived automatically from the algebraic type definition of the type of the selected variable; in this case, x has the type [*]. New variables, such as $v1$ and $v2$, are introduced automatically where a constructor requires arguments.

The inverse operation, called *generalisation*, combines a set of equations into one, provided that the partial functions partition the domain of the new function and that each of the equations is an instance of the new equation.

Enumeration and generalisation guarantee only *partial correctness*. In particular, enumeration can introduce non-termination and generalisation can remove it. For example

$$f\ True \;= 0$$
$$f\ False \;= 0$$

can be generalised to

$$f\,x = 0$$

which is non-strict whereas the original definition is strict. To ensure total correctness, the following additional equation would be required before generalising

$$\mathsf{f}\ \perp = 0$$

where '$\perp$' is the undefined value (bottom).

3.4 Transformation logging

Each step of a transformation or proof is logged in a window known as the *Proof Log*, if logging is turned on. The following example shows the log entries generated by the definition of a new function '*g*' and the subsequent transformation of an expression involving '*g*':

$$\begin{array}{ll} g(1)\ \ldots\ g\,f\,x\,y = f\,x\,y & \\ g\,(\times)\,x\,y & \\ = x \times y & \ldots g(1) \\ = y \times x & \ldots \mathit{comm\_times} \\ = g\,(\times)\,y\,x & \ldots g(1) \end{array}$$

Each equation is identified by the function name followed by the equation number in parentheses. Steps of the transformation are annotated with the name of the function or theorem used for rewriting. The log does not state whether the step involves folding or unfolding, as this is obvious by inspection.

Transformations and proofs are also logged in the *Command Log* window, in the form of meta-language statements, which can be executed to repeat the operations. This log is discussed further in the section on the meta-language.

3.5 Laws

The term *law* is used to refer to a theorem which states the equivalence of two expressions. For example:

$$\begin{array}{ll} \text{ß}x + \text{ß}y \approx \text{ß}y + \text{ß}x & \ldots \mathit{comm\_plus} \\ (\text{ß}a \quad \text{ß}b) \quad \text{ß}c \approx \text{ß}a \quad (\text{ß}b \quad \text{ß}c) & \ldots \mathit{assoc\_append} \\ \mathit{reverse}\ (\mathit{reverse}\ \text{ß}a) \approx \text{ß}a & \ldots \mathit{reverse\_reverse} \\ \mathit{map}\ \text{ß}f\ (\mathit{map}\ \text{ß}g\ \text{ß}x) \approx \mathit{map}\ (\text{ß}f.\text{ß}g)\ \text{ß}x & \ldots \mathit{map\_map} \end{array}$$

where the functions have their usual definitions (see appendix).

The symbol '$\approx$' denotes equivalence to avoid confusion with the equality operator '$=$'. Each theorem is given a name, introduced by an ellipsis.

Variables which start with 'ß' are called *meta-variables*. They have implicit outermost universal quantification over their inferred polymorphic type. Thus the theorem *reverse_reverse* is equivalent to:

$$\forall a :: [*].\ \mathrm{reverse}(\mathrm{reverse}\ a) = a.$$

Laws may be used as rewrite rules in a similar way to equations. This involves type checking and instantiation of any new variables with an explicit value of the correct type, to satisfy the quantifiers.

3.6 Joining

Laborious sequences of primitive fold and unfold operations can often be automated by the use of the *join* transform. Join prompts for an expression with which to replace the current selection in a transform window. MetaMorph attempts to show that the expressions are equivalent using the theorem prover. At present, only proofs by symbolic evaluation are attempted, although these may make use of theorems as well as equations.

Joining allows proof to be used on sub-problems within an overall transformation. It also supports transformation towards a partially-constrained target expression, by using meta-variables within the expression.

A simple example of joining is illustrated using the following definitions:

$$\begin{aligned} &\mathit{twice}\ x = x + x; \\ &y = \mathit{map\ twice}\ [1,2,3]; \end{aligned}$$

It is possible to select the expression *map twice* [1, 2, 3] with the mouse in a transform window and replace it, simply by typing the required expression [3, 2, 1]. This leads to the following entry in the proof log:

$$\begin{aligned}
&\mathit{map\ twice}\ [1,2,3] \\
&= \mathit{map\ twice}\ (1{:}2{:}[\,] +\!\!+ [3]) && \dots +\!\!+ (1) \\
&= \mathit{map\ twice}\ (1{:}[2] +\!\!+ [3]) && \dots +\!\!+ (2) \\
&= \mathit{map\ twice}\ (1{:}([\,] +\!\!+ [2]) +\!\!+ [3]) && \dots +\!\!+ (1) \\
&= \mathit{map\ twice}\ ((1{:}[\,] +\!\!+ [2]) +\!\!+ [3]) && \dots +\!\!+ (2) \\
&= \mathit{map\ twice}\ (([1] +\!\!+ [2]) +\!\!+ [3]) && \dots +\!\!+ (2) \\
&= \mathit{map\ twice}\ ((([\,] +\!\!+ [1]) +\!\!+ [2]) +\!\!+ [3]) && \dots +\!\!+ (1) \\
&= \mathit{map\ twice}\ (((\mathit{reverse}\ [\,] +\!\!+ [1]) +\!\!+ [2]) +\!\!+ [3]) && \dots\ \mathit{reverse}(1) \\
&= \mathit{map\ twice}\ ((\mathit{reverse}\ [1] +\!\!+ [2]) +\!\!+ [3]) && \dots\ \mathit{reverse}(2) \\
&= \mathit{map\ twice}\ (\mathit{reverse}\ [2,1] +\!\!+ [3]) && \dots\ \mathit{reverse}(2) \\
&= \mathit{map\ twice}\ (\mathit{reverse}\ [3,2,1]) && \dots\ \mathit{reverse}(2)
\end{aligned}$$

This example would be quite laborious to do manually using just folds and unfolds.

A powerful feature of joining is that it is possible to use a target pattern with meta-variables standing for arbitrary expressions. For example, the expression

$$[(1,2,3),\ (4,5,6)]$$

may be redefined using the template

$$\textit{reverse}\ [ß1, ß2]$$

instead of the full result

$$\textit{reverse}\ [(4,5,6),\ (1,2,3)].$$

In a practical example, the sub-expressions represented by ß1 and ß2 might be very large. The use of meta-variables not only saves a lot of typing, but also makes it unnecessary to know the exact result of the transformation in advance.

The target expression must contain sufficient detail that it matches the selected expression when both are symbolically evaluated. For example, the template

$$\textit{reverse}\ ß1$$

would not work, because there is no equation for *reverse* which allows '*reverse* ß1' to be unfolded so that it matches [1, 2, 3].

A further important advantage of using meta-variables is that the entry in the command log makes the *minimum commitment* to the expression to which it can be applied:

:- '*Redefine*' ("*reverse* [ß1, ß2]").

This command can be re-run on *any* list of two elements, whereas a fully-instantiated version would only be applicable for a specific problem.

4 Proof

4.1 Introduction

MetaMorph allows outermost universal quantification of variables, but no existential quantification. Universally-quantified variables are denoted by symbols with the prefix 'ß', as discussed above. For example, the theorem

$$\forall f :: * \to *.\ \forall x, y :: [*].\mathrm{map}\ f\ (x \mathbin{+\!\!+} y) = \mathrm{map}\ f\, x \mathbin{+\!\!+} \mathrm{map}\ f\, y$$

is written as

$$\textit{map}\ ßf\ (ßx \mathbin{+\!\!+} ßy) \approx \textit{map}\ ßf\ ßx \mathbin{+\!\!+} \textit{map}\ ßf\ ßy.$$

Proofs are always carried out either by symbolic evaluation or by structural induction. The principal decisions to be made in automating the proof are the lemmas needed to support the proof, how to generalise the induction goal and the choice of variable over which the induction is to be performed. Similar considerations have led to the success of the well-known Boyer-Moore (1979, 1988) theorem prover, which has been the source of many useful ideas.

4.2 Proof strategy

MetaMorph attempts to prove a theorem as follows:

(i) It checks whether the conjecture is an instance of any existing theorem and therefore does not need proving. A useful outcome of this is that a meta-language script for a sequence of proofs may be interrupted and then re-run;

the proofs which have already been carried out are skipped over quickly until the first new theorem is reached.

(ii) If this fails, a proof by symbolic evaluation is attempted, by reducing both sides to the same normal form. This reduction makes use of previously proved theorems as well as function definitions.

(iii) If this fails, a proof by structural induction is attempted (Burstall, 1969), using the type of the induction variable to determine the base case(s) and induction step(s). Simple enumerated types, such as *bool*, are treated as trivial cases of induction with multiple base cases and no induction step.

The theorem prover requires the user to prove a set of theorems in the right order, so that any theorem is preceded by proofs of any supporting lemmas. Heuristics are used to prevent application of previous theorems which may cause non-termination. In particular, permutative theorems, such as commutativity, are only applied if they improve the lexicographical ordering of terms. This follows the approach adopted by Boyer and Moore. Automation of more complex proofs will require more complex heuristics to limit the theorem prover's attention to relevant theorems.

The user must also ensure that each conjecture is sufficiently general to allow proof by induction. If the conjecture is too specific, the induction hypothesis may be too weak to support the conclusion. This has the useful side-effect of forcing us to prove theorems which are more likely to be reusable as transforms.

4.3 An example

This example illustrates the use of the theorem prover in proving the equivalence of two definitions of the factorial function.

The object of this example is to show that

$$\mathit{factorial}\ \text{ß}x \approx \mathit{fac}\ \text{ß}x\ 1$$

given the following definitions

$$\begin{aligned}
\mathit{factorial}\ 0 &= 1;\\
\mathit{factorial}\ (n+1) &= (n+1) \times \mathit{factorial}\ n;\\[1ex]
\mathit{fac}\ 0\ k &= k;\\
\mathit{fac}\ (n+1)\ k &= \mathit{fac}\ n\ ((n+1) \times k);
\end{aligned}$$

The first definition is the standard recursive definition, whilst the second is a tail-recursive version using an accumulating parameter. Elimination of non-tail recursion is a common problem in hardware synthesis, since tail-recursion (i.e. iteration) is used to describe feedback loops in circuits.

The following lemma may be proved by induction and then used to derive the required result trivially:

$$\mathit{fac}\ \text{ß}x\ \text{ß}n \approx \text{ß}n \times (\mathit{factorial}\ \text{ß}x).$$

This lemma is a generalisation of the original conjecture, with the constant '*1*' replaced by a variable. This is necessary to allow a proof by induction to succeed.

This lemma is simply typed into a dialogue box, together with a name for the theorem, such as *fac_lemma*. The lemma is proved automatically and the following entry generated in the proof log:

Proof of fac_lemma : $\mathit{fac}\ \beta 1\ \beta 2 \approx \beta 2 \times \mathit{factorial}\ \beta 1$

$$\begin{aligned}
&\mathit{fac}\ 0\ \diamond 1 && \\
&\quad = \diamond 1 && \ldots \mathit{fac}\ (1)\\
&\quad = \diamond 1 + 0 && \ldots \mathit{plus\_zero}\\
&\quad = \diamond 1 + \diamond 1 \times 0 && \ldots \mathit{times\_zero}\\
&\quad = \diamond 1 \times 1 && \ldots \mathit{times\_succ}\\
&\quad = \diamond 1 \times \mathit{factorial}\ 0 && \ldots \mathit{factorial}(1)
\end{aligned}$$

Induction case proved

$$\begin{aligned}
&\mathit{fac}\ (\diamond 1 + 1)\ \diamond 2 && \\
&\quad = \mathit{fac}\ \diamond 1\ ((\diamond 1 + 1) \times \diamond 2) && \ldots \mathit{fac}(2)\\
&\quad = (\diamond 1 + 1) \times \diamond 2 \times \mathit{factorial}\ \diamond 1 && \ldots \mathit{assumption}\\
&\quad = \mathit{factorial}\ \diamond 1 \times ((\diamond 1 + 1) \times \diamond 2) && \ldots \mathit{comm\_times}\\
&\quad = (\diamond 1 + 1) \times (\mathit{factorial}\ \diamond 1 \times \diamond 2) && \ldots \mathit{comm\_times2}\\
&\quad = \diamond 1 \times (\mathit{factorial}\ \diamond 1 \times \diamond 2) + \mathit{factorial}\ \diamond 1 \times \diamond 2 && \ldots \times (2)\\
&\quad = \diamond 1 \times (\diamond 2 \times \mathit{factorial}\ \diamond 1) + \mathit{factorial}\ \diamond 1 \times \diamond 2 && \ldots \mathit{comm\_times}\\
&\quad = \diamond 1 \times (\diamond 2 \times \mathit{factorial}\ \diamond 1) + \diamond 2 \times \mathit{factorial}\ \diamond 1 && \ldots \mathit{comm\_times}\\
&\quad = \diamond 2 \times \mathit{factorial}\ \diamond 1 + \diamond 1 \times (\diamond 2 \times \mathit{factorial}\ \diamond 1) && \ldots \mathit{comm\_plus}\\
&\quad = \diamond 2 \times \mathit{factorial}\ \diamond 1 + \diamond 2 \times (\diamond 1 \times \mathit{factorial}\ \diamond 1) && \ldots \mathit{comm\_times2}\\
&\quad = \diamond 2 \times (\diamond 1 \times \mathit{factorial}\ \diamond 1) + \diamond 2 \times \mathit{factorial}\ \diamond 1 && \ldots \mathit{comm\_plus}\\
&\quad = \diamond 2 \times (\diamond 1 \times \mathit{factorial}\ \diamond 1 + \mathit{factorial}\ \diamond 1) && \ldots \mathit{dist\_times\_plus}\\
&\quad = \diamond 2 \times ((\diamond 1 + 1) \times \mathit{factorial}\ \diamond 1) && \ldots \times (2)\\
&\quad = \diamond 2 \times \mathit{factorial}\ (\diamond 1 + 1) && \ldots \mathit{factorial}(2)
\end{aligned}$$

Induction case proved

Proof by induction successful.

The proof is by structural induction and involves a base case:

$$\mathit{fac}\ 0\ \diamond 1 \approx \diamond 1 \times \mathit{factorial}\ 0$$

and an induction step:

$$\frac{\mathit{fac}\ \diamond 1\ \diamond 2 \approx \diamond 2 \times \mathit{factorial}\ \diamond 1}{\mathit{fac}\ (\diamond 1 + 1)\ \diamond 2 \approx \diamond 2 \times \mathit{factorial}\ (\diamond 1 + 1)}$$

where the symbols $\diamond 1, \diamond 2, \ldots$ denote constants.

The theorem prover determines the cases automatically from the algebraic type definition for '*num*'.

https://doi.org/10.1017/S0956796800000502 Published online by Cambridge University Press

Having proved the lemma, it is trivial to prove the required result:

$$\textit{factorial}\ \text{ß}1 \approx \textit{fac}\ \text{ß}1\ 1.$$

The resulting proof is logged as follows:

Proof of fac_equivalence : $\textit{factorial}\ \text{ß}1 \approx \textit{fac}\ \text{ß}1\ 1$

$\textit{factorial}\ \Diamond 1$

$= \textit{factorial}\ \Diamond 1\ +\ 0$	...*plus_zero*
$= \textit{factorial}\ \Diamond 1\ +\ 0\ \times\ \textit{factorial}\ \Diamond 1$	... $\times(1)$
$=\ 0\ \times\ \textit{factorial}\ \Diamond 1\ +\ \textit{factorial}\ \Diamond 1$	...*comm_plus*
$=\ 1\ \times\ \textit{factorial}\ \Diamond 1$	... $\times(2)$
$= \textit{fac}\ \Diamond 1\ 1$	...*fac_lemma*

Proof by symbolic evaluation successful.

This proof is by symbolic evaluation, rather than structural induction. It involves instantiating ßn with 1, in the lemma, and simplifying both sides. Note that the proof log presents proofs in a transformational style, transforming the left-hand-side into the right-hand-side, rather than reducing both to a common form.

Once the theorem has been proved, it may be applied as a *transform*, to rewrite expressions. For example

$$\textit{factorial}\ (x+y)$$

may be transformed into

$$\textit{fac}\ (x+y)\ 1.$$

The above proofs make use of the following theorems from the MetaMorph standard theorems library:

plus_zero	$\text{ß}x+0 \approx \text{ß}x$
comm_plus	$\text{ß}x+\text{ß}y \approx \text{ß}y+\text{ß}x$
times_zero	$\text{ßx}\times 0 \approx 0$
dist_times_plus	$\text{ß}x\times(\text{ß}y+\text{ß}z) \approx (\text{ß}x\times\text{ß}y)+(\text{ß}x\times\text{ß}z)$
times_succ	$\text{ß}x\times(\text{ß}y+1) \approx \text{ß}x+\text{ß}x\times\text{ß}y$
comm_times	$\text{ß}x\times\text{ß}y \approx \text{ß}y\times\text{ß}x$
*comm_times*2	$\text{ß}x\times(\text{ß}y\times\text{ß}z) \approx \text{ß}y\times(\text{ß}x\times\text{ß}z)$

The meta-language start-up script typically loads a set of standard functions and proves a number of useful theorems such as these.

5 Validation

5.1 Introduction

The discussion has so far focused on deriving implementations from a specification, which is assumed to be a correct statement of requirements. However, in the overall justification of the safety or security of a system, it is just as important to be able to *validate* the specification. Unfortunately, although formal specifications should be unambiguous, they may still be quite complex and difficult to understand. The specification of a complex chip, such as a microprocessor, typically requires about ten pages of equations.

An important feature of the current approach is that the specifications are executable as Miranda programs, allowing the behaviour of the intended system to be observed. Conventional digital circuit simulation is unable to test more than a small fraction of the behaviour of a typical system. However, the transformation tools support symbolic reasoning, which may be used to explore the meaning of a specification at a much higher level of abstraction. In particular, it is possible to use the theorem prover to reason about the *properties* of a specification and to use the theorem prover to reason about the *properties* of a specification and to use the symbolic rewriting engine to *animate* specifications. These features are not yet fully implemented in MetaMorph, but are sufficiently developed to warrant discussion.

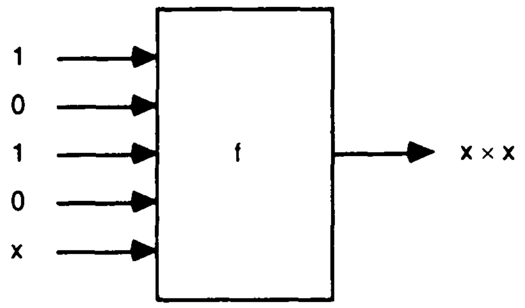
5.2 Symbolic animation

The term *animation* refers to an abstract form of simulation, which allows the operation of a system to be explored at a symbolic level, with a mixture of symbols and concrete values. It overcomes the combinatorial explosion problem of conventional simulation, by limiting the options to be considered at any one time.

Animation is achieved by *partial evaluation*, in which specified identifiers are declared as symbolic constants which are *not* to be evaluated. The animation proceeds to evaluate everything else, leaving the constants as symbols.

Another way of looking at this, is to regard certain identifiers as *pseudo-constructors*, since constructors are simply identifiers which cannot be unfolded further (because they have no definition). For partial evaluation we simply choose not to evaluate an identifier, by ignoring its definition.

As an example of animation, consider a hardware block with four boolean control inputs and a fifth numeric input 'x':



Traditional simulation would require enumerating all the possible input cases, for example 2^{36} if 'x' is a 32-bit number. Symbolic evaluation allows the numeric input to be left as a symbolic constant 'x'. The output of the animation is a symbolic expression involving 'x', such as '$x \times x$', which can be inspected for each of the values of the four control lines. Animation operates at a higher level of abstraction than conventional simulation, avoiding the need to enumerate cases which often makes exhaustive simulation infeasible.

https://doi.org/10.1017/S0956796800000502 Published online by Cambridge University Press

Another example of symbolic animation involves animating the operation of a microprocessor, with the arithmetic operations such as '*add*' treated as symbolic constants. The processor may be animated adding 2 and 2 together, leaving '*add* 2 2', in the destination register. If the result were simply '4', it would not be known whether it was obtained by adding the numbers, multiplying them or doubling one of them (see Fig. 1).

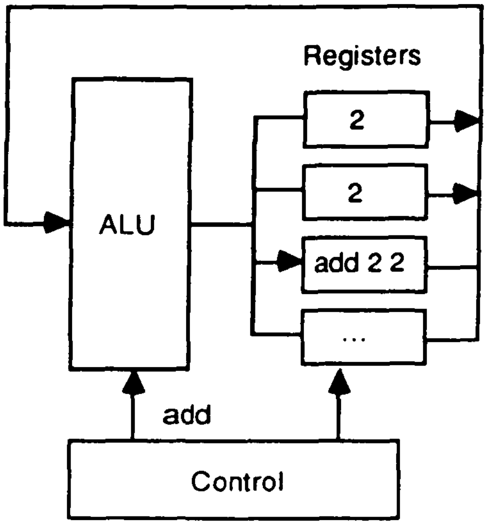


Fig. 1.

With this approach, the data movement and control mechanism may be tested in isolation, before starting to consider detailed operation of the instructions.

Symbolic animation is also of interest because it allows simulation of *incomplete* specifications. A high-level specification may be written without initially defining all of the low-level functions. Such a specification can still be animated if the undefined functions are applied symbolically. Symbolic animation may be applied top-down by progressively enabling successive layers of functionality, starting with a very abstract (and efficient) simulation and ending with a very detailed concrete simulation of the hardware.

MetaMorph allows an expression to be entered and evaluated in the environment defined by the specification, although the performance of the rewriting engine is at present only adequate to animate simple examples. Functions which are not to be evaluated are replaced by *type specifications* as place-holders. A better approach would be to associate a flag with each identifier to say whether it should be evaluated, so that functions could be enabled and disabled without recompiling the specification.

5.3 Validation by proof

The ability to prove properties of specifications can be a considerable help in validation. Such proofs can be used to show that a specification has certain expected properties; for example, that there is no state of a system such that it does not respond to a reset input. Such properties are typically propositions *about* the system, rather than specifications of its functionality.

A good illustration of this idea can be found in the specification of security-critical hardware such as encryption devices. The security properties to be satisfied are typically propositions *about* the system, rather than functionality which can be specified in an executable equational style. The specification of the system deals with its functionality, i.e. what it does. The security properties, on the other hand, are typically concerned with what it does *not* do.

Our constructive style of specification is well suited for specifying the functional properties of a computational system. It is still possible to specify non-constructive properties, albeit in a rather round-about way, using the techniques described by Boyer and Moore (1979, 1988). It may be argued that ease of specifying these properties is of secondary importance, since they are typically much shorter than the specification itself.

The chief problem arises through the lack of existential quantification. For example, it is difficult to state properties such as:

$$\forall \mathrm{d}::\mathrm{documents}.\ \exists \mathrm{u}::\mathrm{users}.\ \mathrm{can\ Access}(\mathrm{u},\mathrm{d}).$$

In some cases, the existential quantifier may be avoided simply by exhibiting a suitable value. Where this cannot be done, it can be eliminated by Skolemisation.

Another difficulty with this form of logic is the notion of infinity. For example, it is difficult to state the proposition that there are infinitely many messages. A constructive solution, based on induction, is to state the existence of at least one message and that for any given message there is a longer message:

$$\vdash \mathrm{is\_message}\ [] \wedge \forall \mathrm{m}::\mathrm{message}.(\#(\mathrm{longer\ m}) > \#\mathrm{m})$$

where a message is simply a list of characters and the function # returns the length of a list.

Notice that the *statement of the proposition* does not depend on providing a valid definition of the function *longer*. If we subsequently provide an invalid definition, we will simply not be able to prove the theorem.

In this example, we require a function which maps any message onto a longer message:

$$longer\ m = \text{‘}a\text{’} : m;$$

The transformational design process preserves the meaning of specifications. It is therefore important to consider whether transformation can violate any of the properties of a specification.

One way in which refinement may introduce problems is if the specification is a partial function. Any real implementation must clearly have *some* behaviour for all possible inputs and must be described as a total function. A specification, on the other hand, may be a partial function which says only what the system is supposed to do in normal operation. Refining such a specification involves extending the domain of a partial function; this introduces new behaviour which may violate the required properties.

The same problem can also arise through data refinement, even when the specification involves total functions. Data refinement can widen the source (type) of a function and make a total function into a partial one.

For example, the source may be an enumerated type, such as

$$colour ::= Red \mid Green \mid Blue;$$

whereas, in a hardware implementation, the corresponding concrete data type would typically be two bits, allowing four possible values. Data refinement operations should therefore generate total functions which take any required properties into account.

The MetaMorph theorem prover requires some extension to allow the proof of more general propositions than transforms, before validation by proof can be properly exploited.

6 Meta-language

The meta-language is the command language used for controlling MetaMorph. It may be used to issue simple commands, such as setting the default font, or to write complete programs for performing transformations and proofs, to extend the functionality of the system.

The meta-language is a special dialect of Prolog, which has been customised for this application. Prolog was chosen because it is well-suited for complex transformation strategies involving heuristic search. The language provides a basic set of standard Prolog predicates and a set of special Prolog predicates for controlling MetaMorph. Strings are treated in a special way, so they can be used to embed object language within the meta-language.

The following trivial example illustrates the use of the meta-language:

```
:- 'Prove' (comm_plus,  "ßx+ßy  ≈ ßy+ßx").
:- 'Prove' (comm_times, "ßx×ßy  ≈ ßy×ßx").
:- 'Prove' (comm_or,    "ßx∨ßy  ≈ ßy∨ßx").
:- 'Prove' (comm_and,   "ßx∧ßy  ≈ ßy∧ßx").

commute :- 'Use theorem' (comm_plus).
commute :- 'Use theorem' (comm_times).
commute :- 'Use theorem' (comm_or).
commute :- 'Use theorem' (comm_and).

:- 'Extend menu' ('Special', commute, "").
```

This script proves four commutativity theorems and then defines a command '*commute*' which will try each of these theorems in turn until one is applicable. Finally, the command is added to a menu called 'special'. The user may then select an expression with the mouse and then use the menu command *commute* on it.

MetaMorph is itself written in Prolog, with the meta-language implemented by an interpreter. The underlying Prolog implementation of the tools is hidden from the user, so that it is only possible to access the equations and theorems of the system through the rules of inference made available as primitive predicates in the meta-language. As a result, meta-language programs cannot introduce errors into a transformation or proof, even if they contain errors.

The MetaMorph interactive user interface is implemented as a front-end to the meta-language; menus and their associated dialogues generate meta-language commands, rather than carrying out their actions directly.[5] As a result, all commands, whether issued interactively with the mouse or textually as meta-language, can be *logged* in a common format, which can be *replayed* as meta-language script.

The *Command Log* window records each top-level meta-language goal which is executed. Nested goals are not logged, since they are automatically executed when the top-level goal is replayed; logging all the goals would result in nested goals being executed more than once. Commands can be cut-and-pasted from the log into a new meta-language window and subsequently run as a script to repeat the transformation. It is often useful to edit scripts, which have resulted from an interactive session, to introduce a hierarchical structure built from reusable Prolog predicates. These predicates may also be parameterised to increase their generality.

It is normal to define a start-up script, as a meta-language file, which customises the user interface, reads in the specification and proves useful theorems.

7 User interface

The user interface follows the usual Apple Macintosh* conventions for menus and dialogues.

The meta-language provides predicates for customising the user interface. The user may add new menus and assign command-key aliases to menu items; each menu item invokes a meta-language predicate having the same name. A dialogue builder allows the user to define dialogues which are invoked by a menu. The responses to the dialogue are passed to the meta-language predicate as arguments. The dialogue builder allows the user to define default values for the dialogue responses. These facilities make it simple to add an interactive user interface to any meta-language program.

Only a few basic menus are installed when MetaMorph is started. The rewriting rules are provided as meta-language predicates, but do not initially appear on the menus. This gives the user freedom to customise the user interface to suit individual requirements.

Because the menu interface operates via the meta-language, it is possible to log all commands to the system, including menu and dialogue selections. The log is simply a meta-language program which can be executed by the tools to replay an entire sequence of definitions, transformations and proofs. Logging mouse selections is more difficult, since simply logging the actual text selected is likely to be too context sensitive to make the log reusable. We have therefore developed an algorithm which

[5] This does not apply to certain commands, such as 'Cut' and 'Paste', which are only used interactively.
* 'Apple' and 'Macintosh' are trademarks of Apple Computer Inc.

is able to log mouse selections as patterns, in which arbitrary details are replaced by meta-variables. This is discussed below.

8 Tool implementation

The implementation of the tools is an important issue, because errors in the tools could lead to errors in the designs produced by the tools. The philosophy which we have adopted is to implement the tools as a declarative statement of the syntax, semantics and type rules of the language, expressed in Prolog. This has been reasonably successful, although pragmatic constraints, such as providing good error messages, have sometimes required a compromise between elegance and practicality.

8.1 Type checker

The polymorphic type checker is implemented as a theorem prover for a formal system of type inference rules. This leads to an elegant implementation in Prolog. Consider for example, the rule for a function application which was introduced above:

$$\frac{A \vdash f :: \sigma \rightarrow \tau \quad A \vdash x :: \sigma}{A \vdash f\,x :: \tau}.$$

This is encoded in Prolog as follows:

```
A /- F°X :: T :-
      A /- F :: U,
      A /- X :: S,
      unifies(U,S ->> T).
```

where /-, °, :: and ↠ are defined as Prolog infix operators.

The principal difference between these two versions of the rule is that the Prolog has explicit unification by the predicate *unifies*, which includes an 'occurs' check and provides error handling.

8.2 Parser

The object-language parser is also implemented in a declarative style, using a Definite Clause Grammar. Some transformation of the basic grammar was necessary to:

- generate the abstract syntax tree
- handle left recursion
- provide error handling
- reduce backtracking to allow better error messages
- include context-sensitive checks.

For example, the following fragment is taken from the grammar in the MetaMorph manual:

```
construct → "("construct")" { argtype }
| constructor { argtype }.
```

This is implemented in Prolog as follows:

```
construct(T) →
    res('('),construct(X),res(')'),argtype_list(X,T).
construct(T) → constructor(X),argtype_list(X,T).
construct(_) → parse_error('Type construct expected').

argtype_list(F,C) → argtype(A),argtype_list(F°A,C).
argtype_list(F,F) → "".
```

9 Reusable transforms

9.1 Introduction

The advantages of transformation over verification become most apparent when transforms are sufficiently general-purpose to be applied to a range of different problems. Such transforms play the rôle of synthesis tools, rather than problem-specific proofs. As well as accommodating different problems, reusable transforms make the design process more robust by being able to cope with changes to an individual specification. In particular, changes to a specification should require only minimal modifications to the transformation script.

Achieving this ideal of reusability depends upon guiding the transformation process in a sufficiently abstract way to avoid unnecessary references to the features of a particular problem. For example, meta-language scripts are not easily reusable if they make reference to variable names and expressions which are arbitrary details of a problem. One approach is to use a variable-free functional language, such as FP (Backus, 1978; Sheeran, 1984).

The situation becomes more complicated when interactive working is considered. An interactive transformation of a particular problem is likely to lead to a command log which is highly-specific to that problem. Whilst it is possible to edit such logs retrospectively, to improve their generality, it would clearly be better if the system could generate a generalised log automatically. MetaMorph incorporates such a mechanism, which is discussed below.

9.2 Generalising the problem

A common technique in solving mathematical problems is to replace the given data by variables, solve the problem and then instantiate the solution with the data. The simplest way to generalise a transformation is to replace, by a variable, any arbitrary constant, function or sub-expression, which plays no direct rôle in the transformation. When the problem has been generalised in this way, what should remain is a skeleton problem, with no irrelevant details. For a transform to be applicable, it is usually the case that the expression to be transformed must have a particular *structure*. Within that structure, there are arbitrary sub-expressions which play no rôle at all in the transformation.

The application of a transform usually involves selecting a particular expression and then applying a transform to it. If the selection is logged literally, say as '$x+4$', it is unlikely that the log can be replayed to transform other similar problems, since the variable names may not match and arbitrary expressions may be different. We have therefore developed a mechanism for logging patterns which records the structural position of a selection, without reference to context-sensitive details. Arbitrary sub-expressions are replaced by meta-variables; the pattern identifies the selection by its structure, rather than its textual content.

For example, selecting the expression in bold roman type in

$$\begin{aligned} \mathit{factorial}\ 0 &= 1; \\ \mathit{factorial}\ (n+1) &= (\mathbf{n+1}) \times \mathit{factorial}\ n; \end{aligned}$$

causes the following log entry, where ß1,ß2, ... are meta-variables used to stand for arbitrary expressions:

$$\mathit{select}(\text{“}n+1\text{”}, \text{“ß1 = ß2 ß3 ß4”}, \text{“ß3”})$$

This means that an expression was selected in the position ß3 when the pattern 'ß1 = ß2 ß3 ß4' was matched against the equation. The position is ß3, rather than ß2, because infix operators such as ' × ' are treated as (prefix) functions. The first argument '$n+1$' ensures that the right equation is selected, should the other patterns match more than one equation of the definition.

The approach is further refined so that subsequent selections on the same equation are logged as patterns *relative* to the last selection, using the predicate *relative* instead of *select*.

This selection mechanism allows the user to operate on a particular problem, but logs a generalised version of the transformation process. Whilst this is very helpful, it is not perfect. In particular, the generalisation is carried out on a step-by-step basis and is therefore too local. No generalisation is made between successive steps in a transformation; this would be difficult because the system does not know *why* a particular item has been selected. For example, it could be because it is the second sub-expression, or because it contains a specific function, or for some more complex reason. Logging the structural position of items works well in many situations, but it is not always general enough. The mechanism must therefore be complemented by other forms of generalisation.

9.3 Generalisation in proof

Definitions in MetaMorph are expressed by recursive functions. Proofs of laws therefore usually involve the use of structural induction. It is an interesting fact that proofs by induction are more likely to succeed if the conjecture to be proved is made more general. If the conjecture is too specific, the induction hypothesis is often too weak to support the conclusion. Generalising such a conjecture is very similar to the problem of generalising transformations for the purposes of logging.

For example, we may wish to prove the following theorem:

$$\#(\mathit{reverse}\ x \quad [a]) \approx \#(\mathit{reverse}\ x) + \#[a].$$

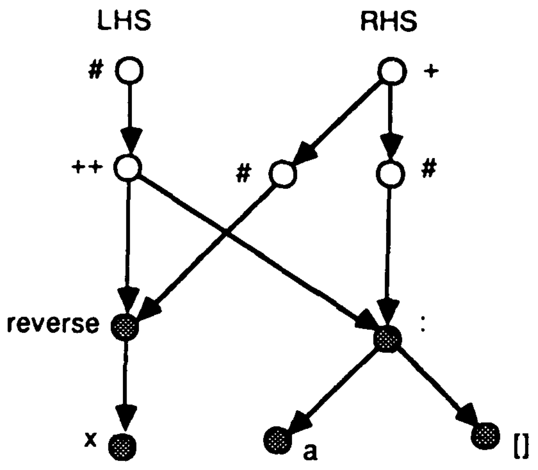


Fig. 2.

In this case, it is easier (and more useful) to prove the more general result:

$$\#(x \quad y) \approx \#x + \#y.$$

It is not necessary to instantiate this to produce the theorem originally required, since the more general result can always be used as a transform, in its place.

Looking at this problem, it is easy to guess that '*reverse x*' and [*a*] are arbitrary expressions, because they occur on both sides of the equality and none of their constituent variables (e.g. '*x*' and '*a*') are referred to in any other expression. Viewed as a directed graph (see figure 2 above), the arbitrary expressions are sub-graphs which are only connected to the main graph by their root nodes.

It is sometimes possible to over-generalise using this algorithm, in which case the sub-expressions can be progressively re-instantiated until a solution can be found. For example

$$\mathit{reverse}\ [a] = [a]$$

cannot be generalised to

$$\mathit{reverse}\ x = x,$$

because it relies on the fact the length of the list is one.

This algorithm is similar to the generalisation method used for logging transformations. It has not yet been applied to generalising goals for the MetaMorph theorem prover.

9.4 Generalisation by joining

The Join transform has proved to be particularly useful in generalising transformation scripts. A transformation is specified by giving a target pattern which makes the *minimum commitment* necessary and is therefore applicable to many similar problems.

Furthermore, only one entry is needed in the command log for what might be a variable number of steps in the proof log. In the example of joining, given earlier, the number of fold and unfold steps generated depended on the length of the list used. Carrying out these low-level operations by hand would lead to a fixed-length sequence in the command log, which could only be applied to a list of the same length. Generalising such a sequence requires re-expressing it as a recursive transform definition in the meta-language, with some termination condition. This problem is avoided using joining.

9.5 Avoiding recursion

Recursive definitions are difficult to manipulate by transformation, because the transformation needs to be equivalent to a proof by induction. This is typically carried out by using the enumeration transform to break a definition into cases, then transforming the cases individually into the same form and finally using the generalisation transform to recombine them.

Fortunately, explicit recursion can always be avoided through the use of higher-order functions, by recognising that the possible patterns of recursions depend only on the data types. These patterns may be abstracted as a set of higher-order functions for each algebraic type, leading to an elegant style of specification which is concise and which makes functions and associated theorems reusable.

For example, the higher-order function *map* provides a means of defining iterative operations on lists:

$$\begin{aligned} \mathit{map}\, f\ [\,] &= [\,];\\ \mathit{map}\, f\,(x{:}xs) &= f\,x : \mathit{map}\, f\, xs; \end{aligned}$$

Standard theorems can be proved about these higher-order functions, such as:

$$(\mathit{map}\, f) \cdot (\mathit{map}\, g) \approx \mathit{map}\,(f \cdot g).$$

Iterative functions on lists can be defined using *map* and subsequently transformed by applying such theorems as transforms. Because the definition is not recursive there is no need to use a problem-specific proof by induction, or the equivalent enumerate-generalise transform. The theorem embodies a reusable proof, which may be applied to many similar problems.

Functions on a particular type may be defined in terms of a small number of higher-order functions. Theorems about such functions describe general properties of the type, which are bound to be reusable.

When a specification contains many explicit recursions, it is possible to carry out a preliminary transformation phase, by folding the recursions with higher-order functions to hide the recursions.

9.6 Future directions

The join transform allows a transformation to be specified by partially-constraining the target expression with a pattern. Another possibility which could be explored is to partially-constrain the sequence of transforms applied. These two approaches could be used to complement one another, so that both constraints could be weaker than if either approach were used alone. Proof then corresponds to the situation where the target is fully constrained, whereas transformation corresponds to the situation where the steps are fully constrained.

Transformation patterns would constrain the search space, by controlling which transforms could be 'plugged in' to a sequence and by restricting the number of steps. To illustrate this idea with a simple example, consider the following composition of transforms:

$$F^{-1} \cdot G \cdot F$$

This is a very common pattern, in which some transformation G is sandwiched between an unfold and fold with the same function F. There are usually only a few possible choices of F for unfolding, but an exceedingly large number of possible functions for folding. With pattern matching, instantiating F in the initial unfold could constrain the subsequent fold to use the same function, thereby limiting the total search space drastically. This would allow an exploratory search to be performed, by backtracking in the meta-language.

Another technique which would be worth considering is Explanation Based Generalisation (EBG) (van Harmelen and Bundy, 1988). This attempts to find a generalised solution to a problem from one or more specific examples. One implementation of EBG involves a special Prolog interpreter which is given two copies of a goal to be solved. One copy is instantiated with the problem and drives the Prolog goal reduction in the normal way. The other copy contains variables to which identical operations are applied *symbolically*. The resulting solution is a new Prolog program which can solve problems of which the given problem was an instance. It would be interesting to investigate how this might be applied to extending the MetaMorph Prolog meta-language interpreter so that it could solve particular problems and yet log generalised Prolog code. EBG might also be considered for proving a generalised theorem, given a particular instance as an example.

10 Application to hardware

10.1 Introduction

The equational notation is used both as a specification language and as a hardware description language. The aim of the transformation process is to turn an abstract hardware specification into a concrete circuit description. The resulting set of equations has dual interpretations, both as a functional program and as a digital circuit. Hardware concepts, such as components and signals, become increasingly apparent as the specification is refined towards a realisable circuit, but they may also exist in the original specification, particularly in connection with interfaces to the outside world.

The initial specification may contain abstractions, such as higher-order functions or non-tail recursions, which have no direct interpretation as a circuit. This is a different situation from transformation of functional programs (Burstall and Darlington, 1977; Darlington, 1981), where the aim is simply to *optimise* a specification which is already an executable (albeit inefficient) program.

Our design approach ensures that the resulting circuit is logically correct viewed as an ideal synchronous system, but omits parametric timing effects such as propagation delays. Similarly, it does not address analogue electrical issues, such as loading (fanout) and noise. It is assumed that conventional CAD tools are used to address these issues after the formal design process is complete.

Circuits may be described down to the gate level, but not the transistor (switch) level, because components are represented as functions rather than predicates. However, this avoids a problem that occurs in using predicate logic to describe circuits, namely that inputs and outputs are indistinguishable, allowing incorrect circuits to be expressed. A few circuit constructs, such as tri-state buses, take a little ingenuity to describe functionally, but can be managed if necessary; for example in that case the trick is to treat a bus as a *component* with multiple inputs and outputs.

Transforms may have a number of roles. Early in the design process, we need transforms which are architectural synthesis tools; these typically take a specification and decompose it into subsystems. This leads to a piece of abstract hardware, which is decomposed into blocks, but still operating on abstract data, say numbers instead of bits. We must then introduce concrete representations for the data, usually as vectors of booleans. It is also necessary to transform arbitrary recursions into tail recursions, which have a hardware equivalent as a feedback loop. Finally, we may apply transforms which optimise a circuit or re-express it in terms of particular library components.

10.2 Time

Unlike imperative languages, functional languages have no concept of sequential execution. This gives them a simple semantics, which greatly simplifies proof or transformation. Use of a functional language to describe a sequential machine requires that the concept of time be introduced explicitly. The simplest approach is to limit attention to synchronous clocked systems with a discrete representation of time. This is a safer design style than asynchronous logic and therefore good practice for safety-critical systems.

There are several ways of representing discrete time within the functional language, which correspond to different *interpretations* of the equational specifications as circuits. MetaMorph is not restricted to any one, although the interpretation must clearly be consistent for a particular problem.

One approach is to define signals as explicit sequences and components as functions from signals to signals. The sequences can be represented as functions from time to value (Gordon, 1986). For example:

$$\begin{aligned}
&\mathit{time} \quad == \mathit{num};\\
&\mathit{signal}\ * \quad == \mathit{time}\rightarrow *;\\
&\mathit{high}\ t = \mathit{True};\\
&\mathit{inverter}\ x = \neg\ .\ x;\\
&\mathit{and\_gate}\ x\ y\ t = (x\ t) \wedge (y\ t);\\
&\mathit{register}\ x\ (n+1) = x\ n;
\end{aligned}$$

Alternatively, sequences can be represented as lists. For example:

$$\mathit{signal}\ * == [*];$$

$$\begin{aligned} high &= True : high; \\ inverter &= map\ (\neg); \\ and\_gate &= map2\ (\wedge); \\ register\ x &= undef : x; \end{aligned}$$

In both cases, the signals and components have the following types:

$$\begin{array}{ll} high & :: signal\ bool; \\ inverter & :: signal\ bool \rightarrow signal\ bool; \\ and\_gate & :: signal\ bool \rightarrow signal\ bool \rightarrow signal\ bool; \\ register & :: signal\ * \rightarrow signal\ *; \end{array}$$

In principle, the two approaches could be combined, by defining an abstract data type for sequences, implemented either as a function or as a list. However, this is an unnecessary complication and prevents the use of pattern matching and the concise syntax for lists. In practice, we normally use lists to represent signals.

Discrete time may also be described *implicitly* by the depth of a recursion. In fact, specifications typically contain recursive definitions with no explicit reference to time. The depth of a recursion may be interpreted as either time or space (hardware complexity). For example, consider the specification of factorial:

$$\begin{aligned} factorial\ 0 &= 1; \\ factorial\ (n+1) &= (n+1) \times factorial\ n; \end{aligned}$$

This function involves a *data-dependent* number of multiplications. It therefore cannot be implemented directly in hardware, since a circuit must have a fixed number of multipliers. Interpreting the depth of recursion as time allows the computation to occupy a variable number of clock cycles on a fixed amount of hardware. This is a simple example of the more general problem of *scheduling* operations. Many hardware design problems can be tackled by starting with a recursive specification, at a level of abstraction where there is no concept of time, and then implementing it in hardware using some form of sequencer (finite state machine) to control the scheduling of operations on a datapath.

10.3 Processes

The non-strict semantics of the language allows the use of *infinite sequences* to describe signals. For example:

$$data = True : False : False : data;$$

This is useful because synchronous digital circuits run indefinitely, rather than stopping with a result after a finite number of cycles. If the circuit appears to stop, it is because a finite-state-machine sticks in one state and not because the clock has been stopped.

The use of infinite sequences to represent signals leads naturally to a circuit model based on process networks (Wadge and Ashcroft, 1985). Each component is a process and the signals joining them are streams (infinite sequences). This approach enables us to handle concurrency and communications, which are essential features of any real hardware system.

The lazy evaluation of a non-strict language ensures that the functional program is *demand driven*. That is, each process generates a new element of its output sequence only when it is required by some consumer process. This automatically ensures proper scheduling of the computation. Eager evaluation, on the other hand, would lead to the computation being driven by the availability of data, resulting in *non-termination* in many cases.

The initial specification of a device typically describes a single process with a number of inputs and outputs. Refining the specification into a circuit decomposes this process into a set of communicating processes. In doing so, the global state is broken down into local components of state, for each of the processes.

10.4 Components

Processes networks are created by *lifting* constants to make signals and lifting component functions to make processes (Johnson, 1983). For example, the constant *True* becomes an infinite sequence of *True*'s:

$$high = repeat\ True;$$

Component functions, such as '$\neg$', are turned into component processes by mapping them over their input sequences. For example:

$$inverter = lift\ (\neg);$$

where $lift = map$.

Components with more than one input may be handled similarly, by defining functions *lift*2, *lift*3, *lift*4, etc:

$$and\_gate = lift2\ (\wedge);$$

There are a number of different ways in which the function *lift*2 may be defined. One possibility is to define it such that *and_gate* is a curried function, which may be partially applied to its inputs:

$$and\_gate :: signal\ bool \rightarrow signal\ bool \rightarrow signal\ bool;$$

$$\begin{aligned} lift2\ f\ [\,]\ \ ys &= [\,];\\ lift2\ f\ (x{:}xs)\ [\,] &= [\,];\\ lift2\ f\ (x{:}xs)\ (y{:}ys) &= f\ x\ y : lift2\ f\ xs\ ys; \end{aligned}$$

The disadvantage of this approach is that multiple inputs and multiple outputs cannot be treated in a consistent way, since currying is only applicable to inputs. This may be overcome by grouping multiple signals as a tuple:

$$and\_gate :: (signal\ bool,\ signal\ bool) \rightarrow signal\ bool;$$

$$\begin{aligned} lift2\ f\ ([\,],\ \ ys) &= [\,];\\ lift2\ f\ (x{:}xs,\ [\,]) &= [\,];\\ lift2\ f\ (x{:}xs,\ y{:}ys) &= f\ x\ y : lift2\ f\ (xs, ys); \end{aligned}$$

A further complication arises with circuits with multiple outputs. Applying the tupling version of *lift*2 to the function *addsub* below, results in a component whose output is a sequence of pairs, rather than a pair of sequences:

$$\textit{addsub} :: (\textit{signal num}, \textit{signal num}) \rightarrow \textit{signal}\ (\textit{num}, \textit{num});$$
$$\textit{addsub}\ (x,y)\ =\ (x+y,\ x-y);$$

This can be converted into a pair of sequences by transposition:

$$\textit{addsub}' :: (\textit{signal num}, \textit{signal num}) \rightarrow (\textit{signal num}, \textit{signal num});$$
$$\textit{addsub}'\ =\ \textit{transpose}2\ \cdot\ (\textit{lift}2'\ \textit{addsub})$$

where *transpose*2 converts a sequence of pairs into a pair of sequences.

The use of tuples makes it necessary to have many versions of *lift* and *transpose*, for different numbers of signals. This leads to the added complication that multiple versions of theorems about these functions are needed for transforming them.

One way of overcoming these problems is to group signals as *lists*, which can be manipulated by polymorphic versions of *lift* and *transpose*, which work for any number of signals:

$$\textit{and\_gate} :: [\textit{signal bool}] \rightarrow \textit{signal bool};$$
$$\textit{lift}'\ f = (\textit{map}\ f)\ \cdot\ \textit{transpose};$$

Multiple outputs can then be transposed as follows:

$$\textit{lift}''\ f\ =\ \textit{transpose}\ \cdot\ (\textit{map}\ f)\ \cdot\ \textit{transpose};$$
$$\textit{addsub}'\ =\ \textit{lift}''\ \textit{addsub};$$

However, there are two disadvantages to the use of lists. Firstly, all the signals in a group must have the same type. Secondly, the type checking is weaker since it cannot check that the bus width matches the component. We are currently using tuples, rather than lists, for our work.

10.5 Circuits

Components are cascaded by function composition, either using an argument to denote the signal, as in:

$$\textit{nandgate in}\ =\ \textit{inverter}\ (\textit{andgate in});$$

or using the composition operator:

$$\textit{nandgate}\ =\ \textit{inverter}\ \cdot\ \textit{andgate};$$

A component consisting of the composition of two functions can be converted into two cascaded processes using the theorem:

$$\textit{map}\ (\text{ß}f\ \cdot\ \text{ß}g)\ \text{ß}s\ \approx\ \textit{map}\ \text{ß}f\ (\textit{map}\ \text{ß}g\ \text{ß}s);$$

The composition operator can be thought of as a higher-order function for the serial connection of two blocks. Similar higher-order functions may be used to build arbitrarily complex circuit structures. The arguments to the HOF are the components to be substituted into the block. Such HOFs are extremely important, as they enable us to separate the structure of a circuit from the details; standard methods can then be developed for transforming common structures.

Feedback can be expressed by recursion, using either a recursive signal definition:

$$out = dtype\ (nandgate\ (in,\ out));$$

or a recursive component definition:

$$circuit\ in = dtype\ (nandgate\ (in,\ circuit\ in));$$

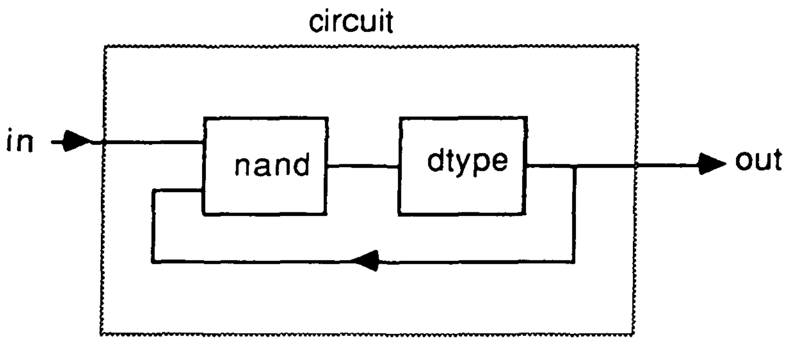


Fig. 3.

Explicit recursion can be avoided by using a higher-order function *loop* to describe the feedback loop:

$$\begin{aligned} loop\ (cct,\ in) &= cct\ in\ (loop\ (cct,\ in)); \\ circuit\ in &= loop\ (dtype.nandgate,\ in); \end{aligned}$$

10.6 Experience

The original prototype of MetaMorph was a simple fold-unfold transformation system, with pre-defined algebraic laws and no theorem prover. The first experiment with hardware design involved the design of a trivial processor with just two instructions, with the following specification:

$$instruction ::= Factorial \mid Square;$$

$$\begin{aligned} simple\ Factorial &= fac\ n; \\ simple\ Square\ n &= n \times n; \end{aligned}$$

$$\begin{aligned} fac\ 0 &= 1; \\ fac(n+1) &= (n+1) \times fac\ n; \end{aligned}$$

Despite its simplicity, this problem has a number of features characteristic of more complex hardware specifications:

- It uses natural numbers which must be refined into binary representations.
- It involves a recursive function definition, which must be made tail-recursive before it can be expressed as a hardware feedback loop.

- The factorial calculation involves a *data-dependent* number of multiplications. This cannot be implemented directly as a variable number of hardware multipliers and therefore requires the introduction of time cycles.
- If the processor is to be implemented using an ALU with multiply and subtract operations, these two operations must be *scheduled*, so that they are carried out alternately using the same ALU.

The specification was first transformed into a sequencer, which schedules operations, and a datapath, which applies the operations. This transformation involved the introduction of registers, an instruction decoder and a finite-state machine for the sequencer. Auxiliary functions were defined for these components and then folded into the specification. The next stage of the transformation refined the datapath into a block-level circuit using the ALU and multiplexers. The final stage was to refine the operations on abstract data into functions operating on concrete binary representations, using lists of booleans for words. However, no tool support was available for this last step.

This example highlighted a number of problems with the early prototype of MetaMorph. In particular, transformation involved many low-level steps and the resulting transformation scripts were not sufficiently general to be re-applied to other similar problems. Also, it was found that extensive use was made of algebraic laws, which involved the introduction of many unproven laws as *axioms*. Based on feedback from this work, a number of enhancements were made to the tools, principally the introduction of the theorem prover and improvements to the logging mechanism.

The next application to be attempted was the design of a 32-bit microprocessor, with many instructions and addressing modes. As an indication of the complexity of this problem, the specification occupied 7 pages of Miranda, plus 9 pages of auxiliary functions and 3 pages of standard library functions. The transformation took about 10 man-months, not including writing the specification, tool enhancements and other investigations.

To make the task more manageable, the processor was specified in terms of a next-state function, which mapped the current state of the processor and memory onto the next state. The processor was successfully animated running a program, by executing the specification as a Miranda program. The specification was subsequently refined to the level of electronic blocks, such as registers and ALU, operating on abstract (non-binary) data and instructions.

MetaMorph continued to evolve during the early part of this work, driven by feedback from the design problem. The theorem prover turned out to be much more useful than originally anticipated and the emphasis moved away from a transformation system, with a theorem prover to verify new laws, towards a hybrid transformation and proof system. Once the microprocessor design was well underway, it was necessary to avoid changes to the tools which would require repeating earlier work. At the end of the exercise, the MetaMorph tools had reached the state described in this paper and many useful ideas had arisen for further development. The processor design was completed apart from some minor details.

It was challenging, but useful, to apply MetaMorph to a problem of such complexity. One of the key issues in software (or hardware) engineering is how to manage the complexity of large design problems; approaches which work well for small problems often prove totally inadequate when scaled up to real-world problems. Experience from the processor design showed the feasibility of the approach, but highlighted the need to further raise the level of abstraction of the transformation process and improve the reusability of transforms. Some suggestions for achieving these objectives are given in section 9.6.

11 Summary and conclusions

In summary, we have developed a formal methods tool which supports transformation of polymorphically-typed equational specifications. The tool incorporates a theorem prover which can be used to verify new transforms or to perform proofs about specific sub-problems within the overall transformational framework. The *join* transform allows the automation of transform sequences leading to a goal which is partially-constrained by a pattern containing meta-variables.

The tool has a meta-language, which can be used to write transformation and proof programs (cf. tactics). The customisable user interface allows meta-language programs to be invoked using user-defined menus. The use of Prolog for the meta-language allows pattern matching and backtracking, which are useful in developing new generic transforms.

An important feature of the system is that all commands are logged as meta-language statements, even if they are given interactively using mouse selections and menu commands. This proof log not only provides a record of a transformation session, but can be cut-and-pasted to form new meta-language commands, which may be parameterised to make them more generic. Mouse selections are logged using patterns, rather than specific instances, so that the command log is applicable to a class of similar problems having the same structure but different details.

In addition to the machine-readable meta-language command log, there is also a proof log, which records transformations and proofs at a step-by-step level.

The tools have some capability for validating specifications, either by symbolic animation using partial evaluation or by proof of properties that a specification is expected to satisfy. These are areas identified for further development.

MetaMorph was developed as a tool for investigating the transformational design of digital hardware from functional language specifications. The approach has been applied successfully to the design of a 32-bit microprocessor. Feedback from this work has led to a number of recommendations for further development of the tools.

In conclusion, MetaMorph makes proof into an integral part of the design process, rather than leaving it as a *post hoc* verification problem. We believe that this plays an important part in managing the complexity of large proofs. The constructive nature of the logic is more limited than approaches based on set theory or higher-order logic, but leads to two important advantages. Firstly, specifications may be executed to assist with validation and, secondly, it becomes easier to automate proof. Since the overall aim is to produce formally-proven designs from validated specifications, rather than just to produce elegant specifications, we believe that this is a good compromise.

Acknowledgement

This work has been sponsored in part by the Defence Research Agency, Electronics Division (formerly RSRE), Malvern, under contract SLS42c/856.

I should especially like to thank David Brazier who assisted with the development of MetaMorph. Thanks are also due to Tony Boswell and David Brazier, who used the tools to carry out the design of a microprocessor, and to Susan Stepney and Dave Whitley for their useful comments on a draft of this paper.

Appendix

The following definitions are used in the examples in this paper:

|| Booleans

$$bool ::= False \mid True;$$

$$\neg\, True \;=\; False;$$
$$\neg\, False \;=\; True;$$

$$True \vee x \;=\; True;$$
$$False \vee x \;=\; x;$$

$$True \wedge x \;=\; x;$$
$$False \wedge x \;=\; False;$$

$$map\; f\; [\,] \;=\; [\,];$$
$$map\; f\; (x{:}xs) \;=\; f\; x : map\; f\; xs;$$

$$\begin{aligned} &map2\; f\; [\,] \quad ys &&= [\,];\\ &map2\; f\; (x{:}xs)\; [\,] &&= [\,];\\ &map2\; f\; (x{:}xs)\; (y{:}ys) &&= f\; x\; y : map2\; f\; xs\; ys; \end{aligned}$$

|| Numbers

$$\begin{aligned} 0 \quad &+ n = n;\\ (m+1) &+ n = (m+n) + 1; \end{aligned}$$

$$\begin{aligned} 0 \quad &\times n = 0;\\ (m+1) &\times n = m \times n + n; \end{aligned}$$

```
        0       > n       = False;
        (m+1) > 0         = True;
        (m+1) > (n+1) = m > n;

|| Lists

        []        ys = ys;
        (x:xs)    ys = x : (xs     ys);

reverse  []    = [];
reverse (x:xs) = reverse xs     [x];

repeat x = x : repeat x;

map f  []    = [];
map f (x:xs) = f x : map f xs;

# []      = 0;
# (x:xs) = #xs + 1;

|| Functions

        (f·g) x = f (g x);

|| Stacks

        stack * ::= Newstack | Push * (stack *);

        pop (Push a x) = x;
        top (Push a x) = a;
```

References

Backus, J. Can programming be liberated from the von Neumann style? *Comm. ACM*, **21** (8): 613–41.

Bird, R. and Wadler, P. 1988. *Introduction to functional programming*. Prentice Hall.

Boyer, R. S. and Moore, J. S. 1979. *A computational logic*. Academic Press.

Boyer, R. S. and Moore, J. S. 1988. *A computational logic handbook*. Academic Press.

Burstall, R. M. 1969. Proving properties of programs by structural induction. *The Computer J.*, **12** (1): 41–8.

Burstall, R. M. and Darlington, J. 1977. A transformation system for developing recursive programs. *JACM*, **24** (1): 44–67.

Cardelli, L. 1987. Basic polymorphic type checking. *Sci. of Computer Programming*, **8** (2): 147–72.

Darlington, J. 1981. An experimental program transformation and synthesis system. *Artificial Intelligence*, **16**: 1–46.

Gordon, M. 1985. *A Machine Oriented Formulation of Higher Order Logic.* Computer Laboratory, University of Cambridge, Technical Report 68.

Gordon, M. 1986. Why higher-order logic is a good formalism for specifying and verifying hardware. In: *Formal aspects of VLSI design*, G. J. Milne and P. A. Subrahmanyam (eds). Elsevier.

van Harmelen, F. and Bundy, A. 1988. Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, **36**: 401–12.

Johnson, S. D. 1983. *Synthesis of digital designs from recursion equations.* PhD Thesis, Indiana University.

Milner, R. 1978. A theory of type polymorphism in programming. *J. Computer and System Sci.*, **17** (3): 348–75.

Sheeran, M. 1984. muFP, a language for VLSI design. In: *1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, 104–12.

Spivey, J. M. 1988. *Understanding Z.* Cambridge University Press.

Turner, D. A. 1985*a*. Functional programs as executable specifications. In: *Mathematical Logic and Programming Languages*, C. A. R. Hoare and J. C. Shepherdson (eds). Prentice Hall.

Turner, D. A. 1985*b*. Miranda: A non-strict functional language with polymorphic types. In: *Proc IFIP International Conference on Functional Programming Languages and Computer Architecture*, Nancy, France. (Springer Lecture Notes in Computer Science, vol 201).

Wadge, W. W. and Ashcroft, E. A. 1985. *LUCID, the dataflow programming language.* Academic Press.