From ACT-ONE to Miranda, a Translation Experiment

Nathan Charles[*], Howard Bowman[+] and Simon Thompson[+]

[*]Department of Computer Science,
The University of York,
York, Y01 5DD, United Kingdom
Email: nathan@minster.york.ac.uk

[+]Computing Laboratory,
University of Kent at Canterbury,
Canterbury, Kent, CT2 7NF, United Kingdom
Email: {H.Bowman,S.J.Thompson}@ukc.ac.uk

Abstract

It is now almost universally acknowledged that the data language ACT-ONE associated with the formal description technique LOTOS is inappropriate for the purpose of OSI formal description. In response to this the LOTOS restandardisation activity plans to replace ACT-ONE with a functional language. Thus, compatibility between ACT-ONE and the replacement data language becomes an issue.

In response to this, we present an experimental investigation of backward compatibility between ACT-ONE and the new LOTOS data language. Specifically, we investigate translating ACT-ONE data types into the functional language Miranda. Miranda has been chosen as it is a widely used functional programming language and it is close in form to the anticipated new data language.

This work serves as a "verification of concept" for translating ACT-ONE to the E-LOTOS data language. It identifies the bounds on embedding ACT-ONE in a functional data language. In particular, it indicates what can be translated and what cannot be translated.

In addition, the paper reveals pertinent issues which can inform the E-LOTOS work. For example, which constructs are needed in E-LOTOS in order to support the class of data type specifications typically made in the

LOTOS setting? We conclude with a number of specific recommendations for the E-LOTOS data language.

1 Introduction

The OSI formal description technique LOTOS [13] combines a process algebraic language for describing "temporal ordering of actions" and a data description language: ACT-ONE, which is based on algebraic specification of data types [8]. It is now almost universally acknowledged that ACT-ONE is an inappropriate data language for the purpose of OSI formal description. The flaws in ACT-ONE have been extensively documented; see, for example, [12] [19] [20]. Among the limitations we particularly note the following:

- *ACT-ONE data definitions are long-winded.* Even inherently very simple data types yield a verbose description.
- *Writing ACT-ONE definitions is laborious and difficult.* Each new type has to be specified equationally and there is no built-in support for types like records and unions.
- *Type definitions are not protected.* Existing types (even those from the standard library) can be extended inconsistently, with the result that the meaning of the type is collapsed. That is, hitherto distinct elements of the type become identified. This is exacerbated by the lack of a distinction between the constructors of elements of the type and general functions defined over the type built by these constructors.
- *The algebraic style is not appreciated by industrial users.* Specifiers and tool builders that work in a declarative style are generally happier viewing definitions as rewrite systems. Although many ACT-ONE definitions can be read thus, such an interpretation is not always valid.
- *Equivalence between elements of types is undecidable.* This hinders the development of reliable verification tools.

As a reflection of these perceived flaws, one of the central objectives of the LOTOS restandardisation activity [16] is to replace ACT-ONE with a more usable data language. Although the E-LOTOS work is still in progress, it is now accepted that the replacement language will have a functional character. In fact, the language will be a derivative of the strict functional language Standard ML (SML) [18]. A clear consequence of this restandardisation activity is that compatibility between ACT-ONE and the replacement data language becomes an issue.

Both compatibility directions are of interest, namely:

- *Forward Compatibility*, meaning translating to ACT-ONE, enables the tools and techniques developed for ACT-ONE to be reused in the context of E-LOTOS.
- *Backward Compatibility*, meaning translating from ACT-ONE, enables existing LOTOS specifications, including those in the library, to be transformed into E-LOTOS specifications.

Thus, forward compatibility implies reuse of old tools, while backward compatibility implies reuse of old specifications.

This paper focuses on the latter of these: backward compatibility. The primary reason for choosing this direction is that it is intellectually more interesting. This is because algebraic specification languages, such as ACT-ONE, are broadly *more expressive* than their functional counterparts. In particular, specifications can be written in ACT-ONE that are not executable (the definition of *sets* is a classic example) and thus, can not be interpreted in their full generality in a functional setting [1].

With the broad aim of considering backward compatibility between ACT-ONE and the new LOTOS data language we have investigated translating ACT-ONE data types into the functional language Miranda [2]. Miranda is a side-effect free lazy functional language which supports higher order functions and polymorphism and has been extensively used [27]. Our preference for Miranda in this translation experiment is largely pragmatic. Miranda is the in-house functional language at the University of Kent and is the language most well understood by the authors. Furthermore, at the time of starting our work, it was not clear which variety of functional language would be adopted by E-LOTOS and so Miranda was a reasonable choice.

We believe the contribution of this experiment in translation is as follows:

- As already stated, the E-LOTOS data language will be SML based. There are some important differences between SML and Miranda, not least that the former is a *strict* language, while the latter is *lazy*. In a

[1] On the other hand, modern functional programming languages, such as Miranda, are higher order as they allow functions to be arguments or results of other functions, while most algebraic specification languages are first order. It is a matter of debate whether this difference affects expressiveness significantly: [10], [11]

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

strict language, all arguments to a function application are evaluated prior to the application itself. In a lazy language, evaluation begins with the application, and arguments are only evaluated if and when it is necessary. Moreover, in the case of structured arguments, such as lists, only those parts of the list required for computation to proceed will be evaluated. However, in terms of classes of languages (imperative, logical, functional etc.) Miranda and SML are closely related. Thus, we anticipate that the reported research will serve as a "verification of concept" for translating ACT-ONE to the E-LOTOS data language. In fact, the majority of our results are also applicable to SML.

- The experiment identifies the bounds on embedding ACT-ONE in a functional data language. In particular, it indicates what can be translated and what cannot be translated.

- In addition, such an exercise in translation reveals pertinent issues which can inform the E-LOTOS work. For example, which constructs are needed in E-LOTOS in order to support the class of data type specifications typically made in the LOTOS setting? One such requirement is the necessity to handle non-termination in the E-LOTOS data language. We will return to this topic in section 5. In addition, we summarise our suggestions for the design of the language in section 6.

- A final benefit is that the ACT-ONE to Miranda translation yields a mechanism for execution of ACT-ONE specifications, which clearly has relevance for tool construction.

As we have already emphasized, in general terms ACT-ONE is more expressive than Miranda. Thus, a completely faithful translation is not feasible. In fact, in many circumstances the resulting Miranda program can be viewed as an "implementation" of the ACT-ONE specification. For example, the embedding will impose a particular evaluation order, which, amongst other things will resolve non-determinism inherent in some ACT-ONE specifications. Thus, the spirit of our experiment is to consider how much of ACT-ONE can be faithfully captured through translation into an executable language.

In addition, we wish our translation to generate meaningful and usable Miranda code. This particularly becomes an issue when considering how to translate ACT-ONE facilities to allow types to be extended and parameterised. In both cases a solution could be devised which in-line expands all types, however, this would lead to an explosion in size of types and lose the re-use inherent in the original ACT-ONE types.

The structure of this paper is as follows. Section 2 presents background material. Both ACT-ONE and Miranda are briefly introduced and then the basic translation approach is described. Section 3 contains the main technical body of the paper; the translation of a series of increasingly more sophisticated ACT-ONE data types is considered. We show how basic specifications are translated; how translations of parts of specifications are combined; how parameterisation and actualisation are rendered and how renaming can be performed. Also discussed are formal equations and the library mechanism. The full algorithm for library translation is given in the appendix. It is through these examples that the translation algorithm is illustrated.

Section 4 discusses some techical limitations of the translation. Section 5 gives a perspective on the translation. In particular, we examine the effect of the Haskell [26] type class mechanism on translating overloading; the effect of lazy evaluation on translating particular kinds of specification and finally we look at the role of non-termination in ACT-ONE specifications.

We conclude, in section 6, with some remarks on the translation and a number of specific suggestions about the design of the E-LOTOS data language.

2 Background and Basic Approach

This section presents background material for the remainder of this paper. We give short introductions to ACT-ONE and Miranda in the next subsection and then we describe the basic translation approach.

2.1 Introductions to ACT-ONE and Miranda

We assume a certain level of familiarity with ACT-ONE and with a functional language. In particular, in order to understand the material in this paper, knowledge of one of the modern functional languages, such as Haskell or SML, should be sufficient. Our discussion of Miranda in this subsection and the examples to be found in the body of the paper should clarify the notational differences between Miranda and other modern functional languages.

ACT-ONE is an algebraic specification language whose fundamentals are fully described in [8]. A number of LOTOS oriented introductions to the language have also been given and can be found in [1], [3], [6]. The semantic model for an ACT-ONE specification is a *many sorted algebra*, data operations being defined as functions over terms in the algebra. An *initial algebra* semantics is employed.

ACT-ONE sorts are defined by a signature and a set of equations. For example, the natural numbers can be defined as follows:

```
type Nat_numbers is
    sorts nat
    opns zero: -> nat
         succ: nat -> nat
         plus: nat, nat -> nat
    eqns forall x, y:nat
         ofsort nat
         plus(x, zero) = x;
         plus(x, succ(y)) = succ(plus(x, y));
endtype
```

The sort names, here just 'nat', and operations, here 'zero', 'succ' and 'plus', together comprise the signature of the data type. The algebra of this data type contains terms constructed by composing the operations of the data type arbitrarily according to their input and output types. The equations of the data type define equality between terms generated from the signature.

In addition to basic algebraic definitions of the form above, ACT-ONE offers facilities to structure and refine specifications. Mechanisms are provided for incrementally *extending* data types (also called *combining specifications*), *renaming* data types, defining *parameterised (generic)* data types, *actualizing parameterised* data types and *reusing* data types defined in a *library*. The reader is referred to the literature [1], [3], [6] for details of these facilities. Section 3 of this paper will consider typical examples of each of these facilities.

Miranda is a lazy functional language which enables both polymorphic and higher order functions to be defined and employs lazy evaluation of expressions. The language contains a rich set of programming features, including built-in data types (numbers, characters, tuples, lists, etc.), algebraic and abstract data types and modules. Although this spectrum of features is highly relevant to the definition of the E-LOTOS data language, we only in fact use a subset of these features in our translation. In particular, the translation interprets ACT-ONE definitions using Miranda Algebraic Types (MATs) (we will refer to these explicitly as Miranda Algebraic Types in order to avoid confusion with the algebraic type concept as embodied in ACT-ONE), functions over these types and modules. We consider these constructs in turn.

Miranda Algebraic Types. As an illustration of Miranda algebraic types and functions over MATs, the following is the definition of a natural number queue in Miranda:

```
queue ::= Create | Add nat queue

first :: queue -> nat
first Create = 0
first (Add x Create) = x
first (Add x (Add y z)) = first (Add y z)

remove :: queue -> queue
remove Create = Create
remove (Add x Create) = Create
remove (Add x (Add y z)) = Add x (remove (Add y z))
```

The description is divided into a MAT definition and the definition of two functions over the MAT: `first` and `remove`. The former of these defines a recursive data type, which means that elements of the type `Queue` can either be of the form `Create` or `(Add n q)`, where `n::nat` and `q::queue`. For example, a typical value of this type is: `Add 5 (Add 6 Create)` which is a queue of two items, whose "first" element is 6. `Create` and `Add` are *constructors* for the data type and are distinguished from other identifiers by starting with a capital letter.

Each function is defined by its type and equations, but, in contrast to the situation with ACT-ONE, these equations have a clear evaluational interpretation. For example, the rules are always applied as rewrite rules from top to bottom, thus resolving any non-determinism arising from overlapping patterns. As an illustration of this, we could rewrite the above rules for `first` as follows:

```
first :: queue -> nat
first Create = 0
first (Add x Create) = x
first (Add x w) = first w
```

Although, the last two rules now have overlapping patterns, the top to bottom order of application of rules prevents non-determinism from arising.

In order to avoid confusion we will use the term *operation* to apply specifically to ACT-ONE operations, while the terms *constructors* and *functions* are used in the Miranda context. Thus, we will not speak of operations in the Miranda context.

Modules. Miranda definitions are collected together in files or *scripts* and the module mechanism supports the inclusion of one script in another. The effect of inclusion is to make the definitions in the included file visible within the including file.

The directive specifying inclusion is exemplified by:

```
%include "ant.m"
```

The basic mechanism is extended in three ways.

- On inclusion, definitions can be hidden (`- dove`) or 'renamed' (`wombat/fish`), thus,

  ```
  %include "ant.m" wombat/fish -dove
  ```

- A script can control exactly which definitions are exported, the default being only those in the script itself. For instance,

  ```
  %export + wombat
  ```

  specifies that together with the definitions in the file (`+`), `wombat` will be exported.

- Modules can be parameterised by adding a `%free` declaration. `%free` is followed by a signature containing type and function declarations; on inclusion, these parameters must be bound to actual types and values. Examples follow in section 3.

Miranda modules are sensible units of program code, which can be interpreted independently of each other. Low level textual insertion (with no syntactic or other restrictions) is provided by the directive:

```
%insert "filename"
```

For introductions to Miranda, we once again refer the reader to the literature: [23], [22].

2.2 The Basic Translation

The translation mechanism is implemented as a suite of Miranda programs. First, an abstract syntax is given for ACT-ONE as a set of Miranda algebraic types; this is a standard way to describe language syntax and closely relates to BNF. Since an abstract syntax is used, it is assumed that in order to resolve issues of precedence the ACT-ONE specifications are fully parenthesised.

The translator interprets ACT-ONE programs expressed in this abstract syntax. The heart of the translation is a set of Miranda functions which maps each ACT-ONE syntactic form into a portion of Miranda script. Thus, the translation has a denotational flavour, in which the denotation generated is a Miranda script. Implementation of the translation is fully described in [2].

Implicit in the translation is the interpretation of ACT-ONE equations as rewrite rules with a particular orientation. This immediately constrains the generality of the translation. In particular, ACT-ONE equations that do not adhere to such an orientation are not translated meaningfully.

3 Examples of translation

The aim of this section is show how LOTOS data types (represented in ACT-ONE) can be translated into Miranda. The section begins with a translation of basic data types taken from [15], then progresses on to more interesting examples found in [14]. A summary of the methods used is given at the end of the section.

Miranda algebraic types and functions are used to represent the ACT-ONE data types. To form these algebraic types it is necessary to identify the constructors of ACT-ONE types. One way to do this is to use a heuristic, such as treating all operators with no equations as constructors and the rest as non-constructors. This would not be sufficient though because, as we will see in Section 3.2 it is possible for constructors to have associated equations, so we would include operators that appear within patterns in the left hand side of an equation as constructors as well. In cases where equations for an operation are given in a different type to that of its signature, this heuristic may fail, identifying a non-constructor as a constructor. Garavel, [9], accepts that strategies exist to identify constructors, but for ease of implementation concludes that the specifier should identify the constructors themselves, by attaching a *special comment*. The specifier should know what they intend to be the constructors so this does not place any limiting constraint.

Given that ACT-ONE data types are abstract data types, it might be expected that they would translate neatly into Miranda's abstract type mechanism `abstype`, unfortunately there are some incompatibilities between their different interpretation of abstract data types. In LOTOS it is possible to distribute the operations associated to a sort across a number of data types, whereas in Miranda they must all appear in the same `abstype` definition. For this reason the ACT-ONE data types are translated into basic Miranda types; it would not be hard to convert the types into the `abstype` form.

Our translation makes some initial assumptions about the data types input:

1. The equations when used as rewrite rules are:
   (a) Confluent [5]
   (b) Terminating [5]
2. All the constructors of a sort must be defined in the same data type as the sort.
3. The constructors of a sort must be defined explicitly. This is done using a special comment (*! constructor *) immediately after the operation declaration. This is consistent with Garavel's suggestion mentioned earlier.
4. There is no overloading of sorts and operations.
5. None of the sorts or operations may be a Miranda reserved word.
6. The equations are given in a prefix form.

These constraints limit the expressiveness of ACT-ONE. However 3, 4, 5 and 6 are pragmatic constraints which do not affect the generality of translation, i.e. they are not really limiting ACT-ONE. In particular the fourth assumption is not restrictive since we assume that ACT-ONE data types have been transformed by replacing overloaded names with unique identifiers. A more general solution whereby the overloading is preserved through the translation will be discussed in Section 5 when we consider Haskell types classes.

However, assumptions 1 and 2 do restrict the class of ACT-ONE data types that can be translated. The first assumption is necessary in order to enable the data types to be viewed as rewrite rules in a Miranda setting. The second assumption is not as fundamental as the first; it can be relaxed, although any translation of an example which falls in this category would be messy and considerably more complex, requiring extensive rewriting of files. These two assumptions are in fact constraints typically applied by current LOTOS tools such as, LOLA [4] and SMILE [7], SDL tools such as, RASTA [17] and in the literature, [9].

3.1 Basic non-parameterised specifications

The first example of our translation is a specification of natural numbers.

```
type Nat_numbers is
    sorts nat
    opns zero: -> nat             (*! constructor *)
         succ: nat -> nat         (*! constructor *)
         plus: nat, nat -> nat
    eqns forall x, y:nat
         ofsort nat
         plus(x, zero) = x;
         plus(x, succ(y)) = succ(plus(x, y));
endtype
```

'succ' and 'zero' have been flagged as constructors. It is not hard to verify this is consistent with the heuristic discussed at the start of this section, and indeed we can also observe that 'plus' is a function defined over the sort 'nat'. In Miranda these defined functions are modelled using Miranda functions. The signature of plus maps neatly to a function signature and the equations to function definitions. Using these ideas the following translation is given:

```
nat ::= Zero | Succ nat

plus :: nat -> nat -> nat
plus x Zero = x
plus x (Succ y) = Succ (plus x y)
```

The type of `Succ` and `Zero` has been extracted from the signature of the ACT-ONE data type to produce a Miranda algebraic type, whereas `plus` has been constructed, as previously described, as a Miranda function.

It is usual for Miranda functions to be written in curried form. A curried function is a function that takes its arguments one at a time, so a function of two arguments would have the type:

```
t1 -> t2 -> t
```

In contrast the types of the corresponding uncurried form would be:

```
(t1, t2) -> t
```

It is trivial to convert between the two notations. However in Miranda we use curried functions as they allow partial function application, see [24, 22], hence its use in the translation despite the ACT-ONE equations being in an uncurried form.

We see that the Miranda definitions differ only syntactically from the ACT-ONE definitions, demonstrating the directness of the translation; this is true of most simple ACT-ONE data types.

Although no example has been given that includes premisses with the equations it should be clear that a premiss translates into a Miranda style `if` test (or guard) in which a list of premisses is treated as a conjunction.

3.2 Non-free constructors

Normally different constructor terms denote different values, we now meet an example where this is not the case.

```
type Switch is
    sorts switch
    opns on:->switch              (*! constructor *)
         not:switch->switch       (*! constructor *)
    eqns forall x:switch
         ofsort switch
         not(not(x)) = x;
endtype
```

This can be translated into Miranda in the usual way except the equation is translated into a law:

```
switch ::= On | Not switch

Not (Not x) => x
```

Unfortunately laws are an obsolete feature of Miranda and are likely to be unsupported in later versions, so it is necessary to find an alternative. [23] provides a way of removing laws whilst keeping the overall meaning:

1. Throughout the script (including the rhs of the laws) replace all right-hand-side occurrences of the lawful constructors by the associated function names. Only the 'left-hand-side' uses of the constructor, i.e. in pattern matching, are left alone.

2. Turn each law into a function definition, by replacing the outermost occurrence of the constructor on the lhs of the law by the associated function name, and replacing each `=>` by `=`. We must also add a last case to the function definition, stating that its result is equal to a call of its associated constructor on the same arguments if no earlier case applies.

This is in fact the way the laws were implemented in Miranda. If we perform the algorithm on the above example, we produce the following code:

```
switch ::= On | Not switch

not (Not x) = x
not = Not
```

3.3 Combination of specifications

This is a feature of LOTOS which allows data types to be formed from other data types, through inheritance. The second assumption at the beginning of the section disallows the introduction of constructors into a sort outside the type the sort is defined in, thus when inheriting a sort we can only extend the sort by adding extra non-constructor operations. For example we can create 'Enriched_nat':

type Enriched_nat **is** Nat_numbers
    opns times:nat, nat -> nat
    eqns **forall** x, y: nat
        ofsort nat
        times(x, zero) = x;
        times(x, succ(y)) = plus(x, times(x, y));
endtype

where 'Nat_numbers' has already been defined. Thus this translates into the following Miranda:

```
times :: nat -> nat -> nat
times x Zero = x
times x (Succ y) = plus x (times x y)
```

Given that the above ACT-ONE type requires 'Nat_numbers' it is self-evident that the translation will require the translation of 'Nat_numbers'. One way to implement this in Miranda is to place 'Nat_numbers' in a module, which is implemented in Miranda as a file. The file would then be `%include`d at the beginning of the 'Enriched_nat' translation. It turns out that this use of modules is convenient for other parts of the translation and so in general each LOTOS type is translated into a module of its own. This could be expected because as we have already suggested LOTOS types are very much like modules.

'Enriched_nat' extends the use of the sort 'nat', we now look at an example that *uses* 'nat' rather than extending it. A queue of natural numbers is an example of this:

```
type Nat_number_queue is Nat_numbers
    sorts queue
    opns create:-> queue               (*! constructor *)
         add: nat, queue -> queue      (*! constructor *)
         first: queue -> nat
         remove: queue -> queue
    eqns forall x, y: nat, z: queue
         ofsort nat
         first(create) = zero;
         first(add(x, create)) = x;
         first(add(x, add(y, z))) = first(add(y, z));
         ofsort queue
         remove(create) = create;
         remove(add(x, create)) = create;
         remove(add(x, add(y, z))) = add(x, remove(add(y, z)));
endtype
```

We translate this directly into the following Miranda:

```
queue ::= Create | Add nat queue

first :: queue -> nat
first Create = Zero
first (Add x Create) = x
first (Add x (Add y z)) = first (Add y z)

remove :: queue -> queue
```

```
remove Create = Create
remove (Add x Create) = Create
remove (Add x (Add y z)) = Add x (remove (Add y z))
```

where the translation of 'Nat_numbers' is an imported module.

3.4 Parameterisation of specifications

This feature of LOTOS allows polymorphic data types to be defined, for example a queue. The type of queue is parametrically polymorphic because its elements may take any type. The following ACT-ONE data type is one method to define a queue in LOTOS:

```
type Queue is
    formalsorts data
    formalopns d0: -> data
    sorts queue
    opns create:-> queue                (*! constructor *)
         add: data, queue -> queue      (*! constructor *)
         first: queue -> data
         remove: queue -> queue
    eqns forall x, y: data, z: queue
         ofsort data
         first(create) = d0;
         first(add(x, create)) = x;
         first(add(x, add(y, z)) = first(add(y, z));
         ofsort queue
         remove(create) = create;
         remove(add(x, create)) = create;
         remove(add(x, add(y, z))) = add(x, remove(add(y, z)));
endtype
```

This type is similar to the 'Nat_number_queue' but there are some important distinctions which can be highlighted:

1. The queue is now no longer of sort 'nat' but of sort 'data'. The type of data has yet to be established - this is achieved during actualization.

2. 'first(create)' is now equal to 'd0', a constant which will be instantiated during actualization, rather than to 'zero'.

This has a number of similarities to Miranda parameterised modules (see [24], 27/4), with 'data' and 'd0' declared `%free` and their bindings given at `%include` time. To take advantage of this, it is necessary to place the translation below in a module (implemented by a file), say, *queue.m*. A direct translation is then produced:

```
%free { data :: type
        d0 :: data
}

queue ::= Create | Add data queue

first :: queue -> data
first Create = d0
first (Add x Create) = x
first (Add x (Add y z)) = first (Add y z)

remove :: queue -> queue
remove Create = Create
remove (Add x Create) = Create
remove (Add x (Add y z)) = Add x (remove (Add y z))
```

Again, as with the ACT-ONE type, the type and definition of `d0` are not specified. The bindings for `d0` and `data` will have to be provided later at `%include` time, which as the next sub-section suggests, is when the type is being actualized.

In Section 5 we discuss an alternative method of translating such parameterised data types using type classes.

3.5 Actualisation of parameterised specifications

Parameterised types are instantiated through actualization. For example, the following assigns natural numbers to the items in a queue to form a queue of natural numbers:

> **type** Nat_number_queue **is**
>   Queue **actualizedby** Nat_numbers **using**
>   **sortnames** nat **for** data
>   **opnnames** zero **for** d0
> **endtype**

The mappings given are required to complement the translation of 'Queue', providing the `%free` bindings for *queue.m*. This is done by placing the translation:

```
%include "nat_numbers.m"
%include "queue.m" {data == nat; d0 = Zero;}
```

in a file *nat_number_queue.m*.

Note that the sort of `data` is provided by using a type synonym whilst the definition of `d0` is expressed using definitional equality. More generally, it is always the case that sorts have bindings provided by type synonyms and operations by definitional equalities.

The mechanism given here allows the possibility of actuals themselves depending on other formals.

3.6 Renaming of specifications

Another feature that LOTOS incorporates is the renaming of one data type to form another. The example that follows generates a type 'Numbers' which is isomorphic to 'Nat_numbers':

type Numbers **is**
  Nat_numbers **renamedby**
  sortnames numbers **for** nat
  opnnames nought **for** zero
    add **for** plus
endtype

Miranda provides no *real* renaming facility but it is possible to use the aliasing method provided in the modules system (see [24], 27/3). In this case we produce:

```
%include "nat_numbers.m" numbers/nat Nought/Zero add/plus
```

To illustrate a more complex form of translation of a renamed data type we turn to 'Connection', an example given in the tutorial [15]:

type Connection **is**
  Queue **renamedby**
  sortnames channel **for** queue
    objects **for** data

opnnames send **for** add
receive **for** first
endtype

This example is more subtle than Numbers. To import the 'Queue' module it is necessary to provide the `%free` bindings but as they are not being actualized in this data type we still require them to be `%free`, furthermore `data` needs to be renamed. We achieve these objectives with the following Miranda:

```
%include "queue.m" {data==objects; d0=d0;}
                   Send/Add receive/first channel/queue
%free {
        objects::type;
        d0::objects;
}
```

This translation has the desired effect of renaming the components of the data type, whilst keeping the formal parameters `%free`. In general a type such as this is translated by first translating the non-formal sorts and operations in the same way as 'Numbers'. Then the `%free` bindings for the formal sorts and operations are provided by setting the formal name to itself, except where a renaming occurs in which case the new name is used. The formal sorts and operations still need to be `%free`, so the `%free` declaration is copied from *Queue.m*, replacing the renamed formal sorts and operations by their new names.

3.7 Introduction of formal equations

The example given below is the first to introduce formal equations:

type Fboolean **is**
formalsorts fbool
formalopns true : -> fbool
not : fbool -> fbool
formaleqns **forall** x: fbool
ofsort fbool
not(not(x)) = x;
endtype

This type requires careful consideration. Let us first consider the translation of the above ACT-ONE type ignoring the equation; we would give the following translation:

```
%free {
        fbool::type;
        true::fbool;
        not::fbool->fbool;
}
```

This is acceptable as it stands, but how do we translate the formal equation? An initial thought may be to treat it as a normal equation but this is obviously wrong because not only has 'not' been declared `%free` (and therefore may have no associated definition in the current script) but when actualized the function would have more than one definition for the same case (one in the module and one in the formal module).

In fact, formal equations should be considered as equations to be satisfied when actualized - a proof obligation - rather than as definitions of equations. For example the following formal equation defines that when multiply is actualized it is true that whatever order its parameters are in the result is unaffected (i.e. multiply is commutative); it is not enough to specify what multiply is, after all, many mathematical operations including + and × would satisfy this constraint:

formaleqns **forall** a, b: nat
ofsort nat
multiply(a, b) = multiply(b, a);

This links well to predicate calculus where we may use the following to represent the above equation:

$$\forall x, y : nat.(\text{multiply } a\ b = \text{multiply } b\ a)$$

A proof is then required that the above equation holds when the equation is actualized. Miranda does not have this powerful system built into it and although it is possible to model the theory in Miranda, for the purpose of translation it is more sensible to place the constraint in a comment and leave it to the specifier to verify that the constraint holds when the type is actualized; this is similar to the way LOTOS deals with the formal equations - no proof is required during actualization. We do not use the built in Miranda commenting system to do this because we want the constraint to be type checked. One way to do this is to write a function that tests the constraint over all possible values in type. This function can be written using a list comprehension (see [22] for an introduction). For the 'Fboolean' example above we would add the following lines after the `%free` declaration:

```
enum_fbool :: [fbool]
test_not1 = and [not(not x) = x | x <- enum_fbool]
```

Notice that only the type of `enum_bool` is given, where we assume `enum_bool` to be a list of all the possible values of type `fbool`. For the sake of type checking there is no need to give the definition of `enum_bool`, however this can be done during instantiation, although for infinite types `test_not1` would be non-terminating if the constraint did hold.

The next example, 'Element', also has formal equations but this time the equations have premisses:

> **type** Element **is** Fboolean
>   **formalsorts** element
>   **formalopns** e_eq, e_ne: element, element -> fbool
>   **formaleqns** **forall** x, y: element
>   **ofsort** element
>   e_eq(x, y) = true => x = y;
>   **ofsort** fbool
>   x = y => e_eq(x, y) = true;
>   e_ne(x, y) = not(e_eq(x, y));
> **endtype**

The formal equations can be expressed in predicate calculus as:

$$\begin{array}{l} \forall x, y : element.(\text{e\_eq } x\ y = \text{true} \implies x = y)\ \wedge \\ \forall x, y : element.(x = y \implies \text{e\_eq } x\ y = true \wedge \\ \forall x, y : element.(\text{e\_ne } x\ y = \text{not}(\text{e\_eq } x\ y)) \end{array}$$

which, incidentally, is equivalent to:

$$\begin{array}{l} \forall x, y : element.(\text{e\_eq } x\ y = \text{true} \iff x = y)\ \wedge \\ \forall x, y : element.(\text{e\_ne } x\ y = \text{not}(\text{e\_eq } x\ y)) \end{array}$$

Again we use list comprehension to model the formal equations, but this time we add constraints, so we would represent the equations in this example as follows:

```
enum_element :: [element]
test_e_eq1
    = and [e_eq x y = true |
              x <- enum_element, y <- enum_element, x=y]
```

```
test_e_eq2
    = and [x=y | x <- enum_element;
           y <- enum_element; e_eq x y = true]
test_e_ne1
    = and [e_ne x y = not (e_eq x y) |
           x <- enum_element; y <- enum_element]
```

For 'Element' to import 'Fboolean' in the translation, the `%free` bindings need to be given. However, as the type is not being actualized at this point, it is not possible to instantiate the type, so a variant of the method given when translating 'Connection' (see Section 3.6) is used to give:

```
%include "fboolean.m" {fbool==fbool; true=true; not=not;}

%free {
       fbool :: type;
       true :: fbool;
       not :: fbool -> fbool;
       element :: type;
       e_eq :: element -> element -> fbool;
       e_ne :: element -> element -> fbool;
}

enum_element :: [element]
test_e_eq1
    = and [e_eq x y = true |
           x <- enum_element, y <- enum_element, x=y]
test_e_eq2
    = and [x=y | x <- enum_element;
           y <- enum_element; e_eq x y = true]
test_e_ne1
    = and [e_ne x y = not (e_eq x y) |
           x <- enum_element; y <- enum_element]
```

3.8 Actualising and renaming in the same data type

The next type, 'Hexstring', uses the LOTOS shortcut whereby it is possible to actualize and rename a data type in the same type:

> **type** HexString **is** NonEmptyString **actualizedby** HexDigit **using**
> **sortnames** hexdigit **for** element

bbool **for** fbool
hexstring **for** nonemptystring
opnnames hex **for** string
bnot **for** not
btrue **for** true
hex_eq **for** e_eq
hex_ne **for** e_ne
endtype

This data type renames 'nonemptystring' and 'string' whilst the remaining sorts and operations are actualized. This translates quite neatly into:

```
%include "nonemptystring.m" {element==hexdigit; fbool==bbool;
                             not=bnot; true=Btrue;
                             e_eq=hex_eq; e_ne=hex_ne;}
                             Hex/String hexstring/nonemptystring
%include "hexdigit.m"
%include "naturalnumber.m"
%include "basicnaturalnumber.m"
%include "boolean.m"
```

Note that the last three inclusions arise from the translation of the earlier inclusions in `hexdigit` and `nonemptystring`.

3.9 Libraries and modules

LOTOS allows the use of libraries to give access to pre-defined types. For example:

library Queue, Nat_Number
endlib
...
type Nat_number_queue **is** Queue, Nat_Number
...

Miranda's module system eliminates the need for a library declaration. For example, ignoring the library declaration, the above skeleton would be translated as:

```
%include "nat_number_queue.m"
%include "queue.m"
```

So far no general method has been given to import modules under all circumstances. To rectify this we give an algorithm that can be implemented. The algorithm works in three stages (**p-specificaton**, **actualization** and **renaming**), bottom-up through the dependency graph of data types. It is fully described in the appendix.

3.10 Summary of the translation

We finish this section with a brief summary of how to translate an ACT-ONE data type into Miranda using the methods discussed in this section.

Basic non-parameterised types

Translate the sorts into Miranda algebraic types with their constructors extracted from the operations marked with (*! constructor *), the types of the constructors are also found here.

Translate the equations into function definitions where the signature of the function is extracted from the corresponding operations. In the case where a constructor has an associated equation or equations, then the method given in section 3.2 is used.

All the types are translated into an individual module, which for convenience will be the name of the type appended with a *.m*.

Combination of specifications

Import the translations of the types to be inherited and their dependencies, then translate the rest of the type in the same way as a basic non-parameterised data type.

Parameterisations of specifications

Import the translations of the types to be inherited and their dependencies. The formal sorts and formal operations are translated into a `%free` declaration whilst the formal eqns are translated into a *testing* function, which should be interpreted as a constraint that has to be satisfied when actualizing the data type. The rest of the type will be translated in the same way as the basic non-parameterised type.

Actualisation of parameterised specifications

Import the translations of the types to be inherited and their dependencies, providing `%free` bindings for the translation of the type being actualized.

These bindings are extracted from the bindings given in the ACT-ONE type. Where a binding exists for a type, function or constructor not declared `%free` then the binding shall be used to create an alias (and hence rename it).

Renaming of specifications

Import the translations of the types to be inherited and their dependencies, providing alias bindings for the translation of the type to be renamed. These bindings are extracted from the bindings given in the ACT-ONE type.

Libraries and Modules

The full details of this translation can be found in the appendix.

4 Limitations of the translation

This section analyses two weaknesses of the translation presented in the previous section.

4.1 The problems of using the aliasing system to rename data types

In Section 3.6 we used the aliasing system in order to rename data types. This system does not strictly rename the components of the data type but instead provides aliases for them, although only the alias can be used in the current scope. In the majority of cases there are no problems with this, however where a type inherits another type twice, once with its contents renamed, this system fails. The type 'both' is an example of this:

```
type original
    sorts colour
    opns purple:-> colour
        inverse: colour -> colour
endtype

type new is original renamedby
    sortnames color for colour
    opnnames mauve for purple
              opposite for inverse
endtype
```

type both **is** original, new

...

endtype

Using the methods discussed thus far, we would translate this as:

original.m

```
colour ::= Purple | Inverse colour
```

new.m

```
%include "original.m" color/colour Mauve/Purple
                      Opposite/Inverse
```

both.m

```
%include "new.m"
%include "original.m"
%include "original.m" color/colour Mauve/Purple
                      Opposite/Inverse
```

The problem with this is that in Miranda it is not possible to import the same script twice, even with all its contents renamed. In fact this is not entirely true, it is possible to do this if the script contained only functions definitions, but considering we are mainly concerned with types, this will not cover a large number of translations.

We ask ourselves what limitations this has: in general, this will not affect the majority of specifications, however, the renaming system is often used to prevent the overloading of the same type inherited twice; this cannot be translated into Miranda using the aliasing system. An alternative way to translate examples that fall into this class would be to, rather than use an automatic renaming facility, copy the script and textually rename the constructors, types and functions. Of course this solution has severe drawbacks, such as the lose of the inheriting structure present in the ACT-ONE types.

This problem highlights a limitation of the implementation of Miranda rather than a limitation of the translation. It is foreseeable that a different implementation may well incorporate an improved aliasing system, which could cope with importing the same script twice, once with the elements renamed.

4.2 The duplication present when inheriting `%free` statements

The translation method given in the last section imports modules by using the `%include` directive. In some cases where a script is imported that contains a `%free` declaration [3] it is necessary to copy the whole `%free` declaration from the imported script as well. This can lead to vast amounts of (almost redundant) copying, especially if formal sorts and operations are inherited over a number of types. In Section 2.1 we identified that there is another method for importing scripts in Miranda, by use of a `%insert` directive. This can be used to reduce the amount of redundancy. To import a module containing a `%free` declaration into the current script all that is required is one `%insert` directive, so for example, to import the script *element.m* given in Section 3.7 we use:

```
%insert "element.m"
```

Note that there is no need to copy the `%include` statement from within *element.m* because this will be automatically inserted into the current script with the rest of the contents of *element.m*.

Unfortunately it is not possible to use the `%insert` method for every example. To see this we consider an example where the current script introduces new `%free` elements and also imports a script which contains a `%free` declaration. Miranda allows only one `%free` declaration per script so this example would not work. In fact, in such cases it is necessary to revert back to the `%include` method.

An advantage of the `%include` method is that as Miranda uses separate compilation of files and stores object code for each file, the speed of compilation of complete specifications is often reduced, especially in cases where a minor change is made to one data type. Most of this is lost using the `%insert` method.

In conclusion it would be sensible to adopt a heuristic that combines the two methods. The fact that there is no *clean* way to import files without a certain amount of redundancy highlights a weakness in Miranda.

5 A Perspective on the Translation

The material in this section discusses some alternative approaches to the translation, and presents a perspective on ACT-ONE brought out by the

[3] Specifically, when dummy bindings are used in the current script, see the appendix for a definition of dummy bindings.

translation. This, in turn, has implications for the design of the data part of E-LOTOS, which we enumerate in the conclusions.

5.1 Parametric specifications, polymorphism and overloading

In this section we discuss the different ways in which overloaded names or parametric specifications can be translated into the functional languages Miranda and Haskell.

5.1.1 Polymorphism and overloading

Before discussing the translation it is worth establishing some terminology and introducing some general ideas. In particular we look at what is meant by 'polymorphism'.

Parametric polymorphism – the polymorphism of the Hindley-Milner type system which underlies Miranda and other modern functional programming languages – is the feature by which a single definition can be used over different types. For instance in writing the definition of the `length` function

```
length []    = 0
length (a:x) = 1 + length x
```

over lists the type of elements in the list is immaterial: we can apply `length` to a list of any arbitrary type, and we therefore say that

```
length :: [*] -> num
```

where `*` is a *type variable*. By this means the same code is associated with the identifier `length` over a whole class of types: namely the list types.

5.1.2 Overloading and type classes

Quite distinct from polymorphism is a mechanism which allows the same name to be associated with different definitions at different types. In the literature of object-oriented programming this is often known as polymorphism. Here we use the terminology of the functional programming community and call it *overloading*.

Suppose that we overload `plus` so that it operates over both numbers and Booleans

```
plus :: num -> num -> num
plus :: bool -> bool -> bool
```

what then is its type? We cannot say

```
plus :: * -> * -> *
```

since there is no definition of `plus` over most types (such as `char`). In Haskell notation (but using the Miranda syntax for type variables) we say that

```
plus :: (Arith *) => * -> * -> *
```

so that plus has type `* -> * -> *` not for all types, but for all types `*` belonging to the *type class* `Arith`. A class is defined by a declaration, exemplified by

```
class Arith * where
  plus :: * -> * -> *
  zero :: *
```

The members of this class are exactly those types which are *instances* of `Arith`. An instance declaration contains definitions of the functions and values named in the signature of the `class` declaration, so that for example an instance making `bool` a member of `Arith` will take the form,

```
instance Arith bool where
  plus = (\/)
  zero = False
```

This mechanism is important because it allows us to give types to functions whose definitions use overloaded functions. For instance we can say

```
sum []    = zero
sum (a:x) = plus a (sum x)
```

and the type of `sum` will be

```
Arith * => [*] -> *
```

that is it takes a list of items of type `*` to a `*` if `*` is in the type class `Arith`. In particular it can be used over the type `bool` (and presumably also `num`).

We believe that this powerful mechanism is of value in our translation, but also that this suggests a sound and effective way of describing the types of overloaded operators in the data part of E-LOTOS, [16].

Note that parametric polymorphism resembles overloading, but in a strong form: there is no type class context (such as `Arith *`) in the type of a polymorphic function, and so no constraint on the type of the function.

5.1.3 Translating parametric specifications

In this section we consider the different possible approaches to translation in the light of the material in Section 5.1.2. As a running example we take the 'Queue' of Section 3.4.

As indicated in that section, we cannot give a polymorphic rendering of the 'Queue' type, depending as it does both on the type 'data' and the value 'd0'. We can however translate a similar type which depends only upon 'data'. This is given by replacing the first equations for `first` and `remove` by

first(create) = error "first"
remove(create) = error "remove"

where the 'error' function aborts execution. We can then write in Miranda

```
queue * ::= Create | Add * (queue *)
```

with

```
first  :: queue * -> *
remove :: queue * -> queue *
```

The advantage of this approach is that `first` and `remove` (and indeed the constructor functions `Create` and `Add`) have polymorphic type, thus supporting a strong form of overloading.

How is the full type 'Queue' rendered in a similar way? The answer is to use the type class mechanism of Section 5.1.2. These are a feature of Haskell (but can be simulated in Miranda[4]).

'Queues' can be created over any type ('data') which contains an element designated 'd0'. We therefore define

```
class Data * where
  d0 :: *
```

Now we have `queue *` defined above, and `first` and `remove` defined exactly as in Section 3.4 except that now their types are

```
first  :: Data * => queue * -> *
remove :: Data * => queue * -> queue *
```

This means that `first` and `remove` can be used over queues of any type in the class `Data`.

What is the advantage of this over the translation of Section 3.4? It allows full overloading, so that `first` and `remove` can be used over more than one type in a given context. This contrasts with the earlier translation in which only a single instance of the parametrised module is allowed in any context; multiple instances have to be replaced by calls to renamed functions.

To conclude this discussion we have shown how the overloading of LOTOS can be accommodated in the type system of Haskell which is essentially the type system of Miranda (or indeed Standard ML) augmented with type classes. We would recommend the inclusion of a type class mechanism in the re-designed data language of E-LOTOS, since it gives a clear and well-founded type to overloaded operators, in contrast to the current situation in ACT-ONE.

---

[4]Using the higher-order nature of Miranda functions it is possible to simulate type classes in Miranda. Suppose we want to model the class

```
class Eg * where
  f ::  * -> [*] -> *
  g ::  *
```

A function `h` of type `Eg * => ....` is now modelled by

```
h' ::  (* -> [*] -> * , *) -> ....
```

whose first argument is a pair of values whose types are those of `f` and `g`. This mechanism requires that the particular `f` and `g` for the type in question are passed as parameters to applications of `h'`. For instance we might write

```
h' (fNum,gNum) e1 ...  en
```

in place of `h e1 ...  en` in a Haskell-style class system.

The method described here is indeed that adopted in simple implementations of type classes.

5.2 Lazy evaluation and infinite objects

It has been envisaged that the new data language of LOTOS will be strict. Miranda, by contrast, is a lazy language and this has some positive benefits for the translation of ACT-ONE specifications. Take as an example the specification

> **type** infinite **is** list, nat_numbers
>   **opns** ones:->list;
>   one:->nat;
>   **eqns** **ofsort** list
>   ones = cons(1, ones);
>   **ofsort** nat
>   one = head(ones);
> **endtype**

Here we define 'one' in terms of 'ones', an infinite list. Specifically 'one' is defined as 'head(ones)' where

head(cons(a,x)) = a

In the initial algebra for the type we have

one = head(ones) = head(cons(1, ones)) = 1

and so the specification is meaningful despite the fact that the rule

ones = cons(1, ones)

does not lead to a terminating rewrite rule. Under a strict translation 'ones' and thus 'one' will be undefined. Tools for LOTOS vary in their treatment of examples such as these: Smile gives no warning that this might be problematic, and 'one' reduces to 'succ(0)'; Topo core dumps in the same situation.

It is questionable whether such features of ACT-ONE are used in day-to-day specifications. We might suggest that E-LOTOS incorporate lazy evaluation, but if this were to happen there need to be stipulations placed on the data passed between processes. In particular compound data items need to be fully evaluated before being communicated as otherwise unevaluated expressions of unbounded size can be passed from process to process.

5.3 Totality and termination in data specifications

In a functional programming language like Miranda it is quite possible to define functions which are only partial over their domains. For instance

```
head (a:x) = a                                        (1)
```

is undefined on the empty list `[]`, since the cases given in the pattern match are not exhaustive. Another example is provided by

```
fac 0 = 1                                             (2)
fac n = n * fac (n+1)
```

which when applied to any positive argument will give no result.

The non-termination of `(1)` is intended and indeed is benign, since the case(s) in which the function fails to terminate are decidible, that is they can be tested for at run-time. One can therefore complete the definition `(1)` to `(3)` in which an error is raised or an exception thrown in the empty list case.

```
head (a:x) = a                                        (3)
head [] = error ...
```

The form of non-termination evident in `(2)` is in general not decidible (this is exactly Turing's halting problem, of course). We can force our language to avoid such situations by controlling the forms of definition so that only recursive calls on structurally smaller arguments are permitted. Thus, only structural/primitive recursion is allowed [5] . Clearly `(2)` would be disallowed in this case, but the correct definition

```
fac 0 = 1                                             (4)
fac n = n * fac (n-1)
```

is permitted since the recursive call is on the smaller `(n-1)`. Such an approach has various merits.

First, and most importantly, it simplifies the meaning of the language. In this situation we can rely on a function call giving one of two outcomes

- a defined result is returned, or
- an exception is raised.

[5] This approach has also been advocated for general functional programs; see [25].

In either case the outcome is evident after a finite amount of time; it does not fall into a 'black hole' as would `fac 2` under definition `(2)` above. This form of *divergence* is problematic in describing the semantics of the language: the system appears to deadlock, but not in the same way that `Stop` deadlocks. Divergence becomes particularly difficult to treat in a timed version of the language, where it becomes necessary to decide whether expression evaluation is instantaneous or that it allows time to pass.

Related to this semantic difficulty is the problem caused for simulation of the language. In the terminating case of the language we know that expression evaluation causes a defined outcome (as explained earlier); in the general case there is a risk that a tool will fail to terminate while evaluating a data value. Non-termination is also a problem for program verification; it has been shown that non-termination can add complexity to the verification of functional programs [21], and so in the absence of non-termination we would simplify reasoning about LOTOS specifications.

From the language design point of view, one might argue philosophically that functions over data in LOTOS or E-LOTOS should be simple, with complexities of behaviour only evident in the behavioural part of the language.

Finally, it appears that in the context of LOTOS this proposal is not restrictive in practice. All the libraries we have examined use only structural forms of recursion to define the specified functions.

6 Conclusions

6.1 Overview and Summary

This paper has described an experiment in backward compatibility in the context of the LOTOS data typing language. We have described how to translate data types written in ACT-ONE into the functional language Miranda. This language has similarities to the new data language being developed in the E-LOTOS forum. We have shown that, given some constraints on the input ACT-ONE data types, translation is feasible. Furthermore, the more restrictive of these constraints (e.g. confluence and termination) are in practice already imposed on the ACT-ONE data language; they are, for example, required in order for tools to be constructed.

A technical limitation of our translation is that, as discussed in section 4, our modelling of renaming is not fully general since we have implemented ACT-ONE renaming using aliasing. However, this arises from the features of Miranda and is not a fundamental hindrance to general translation into functional languages. In particular, as discussed in section 5, this problem can be resolved using type classes.

One of the main benefits of performing such an experiment in translation is that it informs the design of the new E-LOTOS data language. We conclude this paper by, in the next subsection, summarising our recommendations in this direction.

6.2 The design of E-LOTOS

We recommend the following:

1. We would like the language to be *total*, as outlined in Section 5.3. We understand that there is some demand from the user-community for partiality, but we believe that it is of the benign form (see example `(1)` above). Specifications of this form can be seen as a shorthand form of an exception-raising version (as in `(3)` above) and so forming part of a total language. This view is supported by the document [20], which is one of the most indepth discussions of the problems of LOTOS data types. In particular, the partiality argued for by [20] is completely of the exception handling variety. Thus, we believe that full recursive definitions are unnecessary for LOTOS data specifications, and advocate the change for the advantage it brings in simplification, simulation, tool support and verification.

2. We believe that making the language polymorphic, with type classes as described in Section 5.1.2, allows the overloading required in the data language to be presented in a straightforward but well-founded form. Each item defined gets a most-general type; if it involves overloading there will be one or more constraints over the variables appearing in the type. There is an additional benefit that polymorphism allows the re-use of the same code (rather than simply the same name) over different types. In the libraries of Miranda and other languages it is plain that most of the list manipulating functions are polymorphic, and so re-usable over the whole class of list types.

   In designing a language it is possible to choose to model a particular feature in more than one way. In the case of parametric and overloaded functions it can be argued that the module system can provided the same functionality. We would agree with this, but also note that the most-general *types* given to overloaded functions can only be described using a polymorphic system with type classes, and not simply by means of modules.

3. Building upon polymorphism one can suggest that higher-order functions are added to the language. These are functions whose arguments or results are themselves functions and they add another dimension of re-use, so that we can for instance define a single operation to apply a function to every member of a list. Again in the case of Miranda and other languages it is evident that higher-order functions are in widespread use in defining reusable libraries of general functions. It is reasonable to suggest adding higher-order functions as a construct of the language since the facilities they provide can be given using a module system only in an indirect and inelegant way.

4. Another important feature of Miranda which facilitates verification is the presence of full-precision integers rather than fixed-size representations of them. The advantages of these include their accuracy, and their correspondence to the usual algebraic data type definition of natural numbers.

References

[1] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25–59, 1988.

[2] N. Charles. *Translating LOTOS Data Types into Miranda.* Final year project, University of Kent, June 1996.

[3] J. de Meer, R. Roth, and S. Vuong. Introduction to algebraic specifications based on the language ACT ONE. *Computer Networks and ISDN Systems*, 23:363–392, 1992.

[4] Departamento De Ingenieria Telematica, Universidad Politecnica de Madrid. *LOtos LAboratory*, February 1995.

[5] N. Dershowitz and J.-P. Jouannaud. Rewrite systes. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B, Formal Methods and Semantics.* MIT Press/Elsevier, 1990.

[6] K.J. Turner (editor). *Using Formal Description Techniques, An Introduction to Estelle, LOTOS and SDL.* Wiley, 1993.

[7] H. Eertink. Executing lotos specifications: the smile tool. In T. Bolognesi, J. van de Lagemaat, and C. Vissers, editors, *LOTOSphere: Software Development with LOTOS.* Kluwer Academic Publishers, 1995.

[8] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics.* EATCS, Monographs on Theoretical Computer Science. Springer-Verlag, 1985.

[9] H. Garavel. Compilation of LOTOS abstract data types. In S. T. Vuong, editor, *Formal Description Techniques, II : proceedings of the IFIP TC/WG 6.1 Second International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, FORTE '89, Vancouver, Canada, 5-8 December 1989*, pages 147–162. Elsevier Science Publishers, 1989.

[10] J.A. Goguen. Higher-order functions considered unnecessary for higher-order programming. In D.A. Turner, editor, *Research Topics in Functional Programming, University of Texas at Austin Year of Programming Series*, pages 309–351. Addison-Wesley, 1990.

[11] R.J.M. Hughes. Why functional programming matters. In D.A. Turner, editor, *Research Topics in Functional Programming, University of Texas at Austin Year of Programming Series.* Addison-Wesley, 1990.

[12] ISO. *Towards a Proposal for Datatypes in E-LOTOS.* Output Document of ISO/IEC JTC1/SC21/WG7/1.21.20.2.3, Ottawa Meeting, July 1995.

[13] ISO 8807. *LOTOS: A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*, July 1987.

[14] ISO 8807, Annex A. *Standard library of data types*, 1989.

[15] ISO 8807, Annex C. *A Tutorial on LOTOS*, 1989.

[16] ISO/IEC JTC1/SC21/WG7 N1053. *Revised Working Draft on Enhancements to LOTOS (V3)*, March 1996.

[17] N.N. Mansurov, A.S. Ragozin, A.V. Chernov, and I.V. Mansurov. Tool support for algebraic specifications of data in sdl-92. In R. Gotzhein and J. Bredereke, editors, *FORTE/PSTV 96, Formal Description Techniques, IX/ Protocol Specification Testing and Verification XVI, Kaiserslautern, Germany, 8-11 October 1996*, pages 61–76. Chapman and Hall, 1996.

[18] R. Milner, J. Parrow, and D. Walker. *The Definition of Standard ML.* MIT Press, 1990.

[19] H.B. Munster. Comments on the LOTOS standard. Technical Report DITC 52/91, National Physical Laboratory, Teddington, Middlesex, UK, September 1991.

[20] H.B. Munster. LOTOS specification of the MAA standard, with an evaluation of LOTOS. Technical Report DITC 191/91, National Physical Laboratory, Teddington, Middlesex, UK, September 1991.

[21] S.J. Thompson. A logic for Miranda. *Formal Aspects of Computing*, 1:339–365, 1989.

[22] S.J. Thompson. *Miranda, The Craft of Functional Programming.* Addison Wesley, 1995.

[23] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Proceedings of IFIP Conference on Functional Programming Languages and Computer Architecture, Nancy, France*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, September 1989.

[24] D. A. Turner. *Miranda Online Manual Pages*, 1989.

[25] D. A. Turner. Elementary strong functional programming. In *Proceedings of First International Symposium, FPLE'95, Functional Programming Languages in Education*, volume 1022 of *Lecture Notes in Computer Science*, pages 1–14, December 1995.

[26] WWW. *Haskell 1.3, A Non-strict Purely Functional Language.* World Wide Web page http://haskell.cs.yale.edu/haskell-report/haskell-report.html.

[27] WWW. *Real World Applications of Functional Programming.* WWW page http://www.dcs.gla.ac.uk/fp/realworld/.

Appendix - An Algorithm to Import Modules

This algorithm can be seen as a denotational semantics from ACT-ONE to Miranda, which we present using the usual syntax for denotational semantics, as the function:

$$[\![\,]\!] : \psi_A \longrightarrow \psi_M$$

where ψ_A is the set of all ACT-ONE data types and ψ_M is the set of all Miranda programs. Thus $[\![\text{child}]\!]$ is the translation of the ACT-ONE data type 'child', which we would place in *child.m*. It is necessary to define $[\![\,]\!]$ using structures such as, $[\![\text{child}]\!]$. Although this is recursive, ACT-ONE data types cannot be cyclic (i.e. a type can not inherit itself), so the definition of $[\![\,]\!]$ will also not be cyclic. Since the dependencies are non-cyclic we can (and do) perform the translation 'bottom-up': first we translate the files depending on nothing else, then the files depending on those files, and so on. The algorithm is presented as follows:

Definition: A *dummy binding* of identifiers i,j,... is made when i,j,... are given as values in the actualisation of a module with parameters when that module is `%include`d and then subsequently in the including script the identifiers i,j,... are declared `%free`. An instance of this is given by the binding of d0 in the final example of Section 3.6.

1. **p-specification**

type current **is** child_1, child_2, ..., child_n

...

endtype

(a) `%include` each [[child_i]] providing dummy `%free` bindings for identifiers declared `%free` in [[child_i]].

(b) Copy the `%include` statements (including any bindings) from each [[child_i]] and remove exact duplicates (i.e. `%include` statements with identical files, `%free` bindings and alias bindings).

(c) Copy the `%free` statements from each of the [[child_i]], combining them into one `%free` declaration, removing duplicate entries. Note this `%free` declaration may be extended further if the current type has formal parts.

2. **actualization**

type current **is** formal_child

actualizedby actual_child_1, actual_child_2, ..., actual_child_m

[actualization mappings]

endtype

(a) `%include` [[formal_child]], provide its `%free` bindings, using the actualization mappings, as described in Section 3.5.

(b) Copy the `%include` statements in [[formal_child]], updating any (`%free` and alias) bindings with new names given in the actualization mappings. The bindings are updated as follows:

- For each `%free` binding which has a corresponding entry in the actualization mapping replace the right hand side of the binding by the new name given in the mapping.
- For each alias binding which has a corresponding entry in the actualization mapping replace the left hand side of the binding by the new name given in the mapping.

(c) Goto step 1., treating the actual children as the children.

(d) Now for each sort and operation that has a actualization mapping but not handled in 2(a) or 2(b) find the script the operation was originally translated into (this is done by searching through all the files that have been `%included`) and place an alias binding after its corresponding `%include` declaration.

3. **renaming**

> **type** current **is** child
>   **renamedby**
>   [rename mappings]
> **endtype**

(a) First we deal with the `%include` statement, the renaming of `%free` identifiers and the renaming of identifiers that already have an alias binding.

   i. `%include` [[child]] providing dummy `%free` bindings for identifiers declared `%free` in [[child]] and where any of the identifiers have a rename mapping, use the new name as the dummy.

   ii. Copy all the `%include` statements in [[child]], including their bindings (both alias and `%free`), renaming where a rename mapping exists. The renaming is performed as follows:

      - For each `%free` binding which has a corresponding entry in the rename mapping replace the right hand side of the binding by the new name given in the mapping.
      - For each alias binding which has a corresponding entry in the rename mapping replace the left hand side of the binding by the new name given in the mapping.

   iii. Copy the `%free` statement from [[child]], renaming the dummy identifiers wherever a corresponding rename mapping exists.

(b) Next we deal with the *new* alias bindings.

   i. For each sort and operation that has a rename mapping and that has not been handled in 3(a)(i) or 3(a)(ii), find the script the operation was originally translated into (this is done by searching through all the files that have been `%include`d) and place an alias binding after its corresponding `%include` declaration. In the case that an operation was translated into both a constructor and a function it is necessary to provide alias bindings for both of them.