Technical Report

UCAM-CL-TR-221
ISSN 1476-2986

Number 221

Transforming axioms for data types into sequential programs

Robert Milne

ISSN 1476-2986

Transforming axioms for data types into sequential programs

Robert Milne*

BNR Europe Limited
London Road, Harlow, Essex, CM17 9NA
United Kingdom

Abstract

A process is proposed for refining specifications of abstract data types into efficient sequential implementations. The process needs little manual intervention. It is split into three stages, not all of which need always be carried out. The three stages entail interpreting equalities as behavioural equivalences, converting functions into procedures, and replacing axioms by programs. The stages can be performed as automatic transformations which are certain to produce results that meet the specifications, provided that simple conditions hold. These conditions describe the adequacy of the specifications, the freedom from interference between the procedures, and the mode of construction of the procedures. Sufficient versions of these conditions can be checked automatically. Varying the conditions could produce implementations for different classes of specification. Though the transformations could be automated, the intermediate results, in styles of specification which cover both functions and procedures, have interest in their own right and may be particularly appropriate to object-oriented design.

*This work was partly supported by a Royal Society / SERC Industrial Fellowship held on leave at the Cambridge University Computer Laboratory. An earlier version of this paper was presented at the Fourth BCS - FACS Refinement Workshop, 1991.

1 Introduction

1.1 Scope

Most work on specifying data types abstractly emphasises applicative constructs, without any notion of store. Most work on implementing software emphasises imperative constructs; for instance, informal structured analysis and design methods use data stores instead of parameters, and programming methods use object classes instead of abstract data types. There are both cultural and technical reasons for this difference of emphasis: applicative constructs are regarded by their supporters as being easier to understand and to manipulate formally, whilst imperative constructs are regarded by their supporters as being easier to write and to execute efficiently. Indeed, to people accustomed to data stores and object classes, the benefits offered by applicative constructs are rarely self-evident. The work in this paper explores ways of reducing this difference between theory and practice. The ways involve providing imperative specifications of software, relating applicative and imperative specifications, and converting specifications into implementations systematically; here an 'applicative specification' is one which specifies applicative functions, whilst an 'imperative specification' is one which specifies imperative procedures.

This work contributes to the assembly of techniques for developing specifications by stages into implementations that guarantee correctness by construction. The implementations are adequately efficient for conventional computers and readily expressible in commonplace programming languages, such as C and C++. They could nonetheless be generated automatically for a very wide range of specifications. The stages of development are distinguished from one another partly so that alternative implementations may be devised if necessary and partly so that the corresponding stages of maintenance, which in practice often involve moving from implementations to specifications, may be understood more fully.

This work also assists with the validation of specifications, by identifying conditions which specifications must meet in order to be acceptable for abstract data types. These conditions could be checked automatically.

In this paper the techniques discussed allow equalities to be interpreted as equivalences of observable behaviour, applicative functions to be converted into imperative procedures, and axioms to be replaced by programs using assignable store. Each of these three techniques requires that specifications be validated against some associated conditions. These conditions ensure that the functions in applicative specifications are adequately defined (and, in particular, are sufficiently complete), that the procedures in imperative specifications do not interfere with one another, and that the functions or procedures are constructed in certain ways. The first and second of these conditions could reasonably be imposed on finished specifications of all abstract data types. The third is specific to a particular technique for replacing axioms by programs and to a particular class of abstract data types; other techniques and classes are also important and should be considered. (In fact rather few techniques and classes require consideration: many useful abstract data types can be collected and organised simply, by relying on the ways in which their members are constructed and observed, and these ways need to differ only in very constrained manners.)

The emphasis in the paper is on sequential implementations which copy data structures. It can be extended to cover sequential implementations which share data structures. Some work has been done on handling concurrent implementations in a similar manner [12], but more needs to be done to provide techniques for such implementations which satisfy enough compositionality conditions.

In order to convey the ideas, the presentation in this paper is fairly informal; the formal definitions, assumptions, and proofs should appear elsewhere.

1.2 Structure

As motivation, in §2 there is a simple application of the techniques, an argument for the correctness of the application, a discussion of the limitations of the application, and an outline of how the techniques are generalised to circumvent the limitations. The notation used by the imperative specifications is explained in §3; it is very closely related to that for the RAISE specification language. The concepts used for the applicative specifications are discussed in §4, but some of them are not used until much later in the paper. The separate stages of the general techniques are described in some detail in §5, §6 and §7; they are illustrated using modest extensions of the simple example, but the stages have been tested on several realistic abstract data types. For ease of reference, the specifications considered are collected together, in §8.

2 Motivation

2.1 Survey

The techniques considered in this paper permit applicative specifications to be transformed into imperative implementations. To demonstrate the techniques in their simplest forms the obvious example (of lists or stacks) is discussed in 2.2. An argument for the correctness of the techniques in this case is sketched in 2.3. The reasons why the simplest forms cannot be generalised immediately are given in 2.4. The appropriately general techniques are outlined in 2.5.

2.2 A simple illustration

In 8.1 there is an applicative specification of 'List' which largely takes a familiar form. The axioms in it consist of equations and an induction rule. The equations are actually *unconditional*, in that they hold irrespective of any pre-conditions; in other specifications the equations may need to be *conditional*. The induction rule quantifies over predicates, which are treated as functions having result type **Bool** (as is usual in systems based on higher order logic). (Many languages provide short ways of writing induction rules, but in this paper the long way exhibited in 8.1 is adopted, in order to expose the concepts more clearly.) In the specification in 8.1 (and in every other specification in this paper) 'Element' is taken to be a type which is given in advance; it might be **Bool** or **Int**, for instance. The general forms of types and specifications are analysed in 4.1 and 4.3.

An intuitive conversion of this applicative specification into an imperative specification is given in 8.2, using notation which is explained in 3.1, 3.2, 3.3 and 3.4. (Again the notation is not shortened, in order to expose the concepts.) Among the axioms are ones asserting that 'empty ( )' and 'add (e)' are always 'deterministically convergent' in the sense discussed in 3.4. The presence of these particular axioms is justified informally because the types of 'empty' and 'add' convey less information in 8.2 than in 8.1 and formally because such axioms are needed for the argument in 2.3.

Before the imperative procedures can be compared directly with the applicative functions their names must be changed to avoid clashes. A specification which changes the names is provided by 8.3. It has the same properties as the specification in 8.2, except that it uses the names 'empty_', 'add_', 'head_', 'tail_' and 'is_empty_' instead of 'empty', 'add', 'head', 'tail' and 'is_empty'.

The imperative specification in 8.2 does not mention the type 'List' and is not immediately susceptible to being proved correct using abstraction functions [7] or simulation relations [13]. To establish that it is correct, versions of 'List' and the applicative functions acting on lists are defined in terms of the imperative procedures by extending the specification in 8.3 without giving the procedures new properties. The relevant definitions are given in 8.4. These definitions are examined at some length in 2.3, which sketches an argument that they have the properties laid down in 8.1. This argument relies heavily on the notation explained in 3.1, 3.2, 3.3 and 3.4. It demonstrates that the specification in 8.4 is a 'refinement' of that in 8.1, in the sense discussed in 4.2; this particular refinement effectively uses an abstraction function which is the identity.

It is straightforward to implement the procedures specified in 8.2 by defining them fully. This is done in the specification in 8.5, on the assumption that the implementation language provides variables capable of holding finite lists. The procedures in 8.5 evidently have the properties laid down in 8.2, so they can be interpreted as in 8.4 to provide functions having the properties laid down in 8.1.

2.3 The correctness of the illustration

In 8.4 the members of 'List' are taken to be procedures; the application of one of these procedures terminates with a store which embodies the fact that 'add_' has been applied some number of times since the most recent application of 'empty_'. Consequently,

$$\lambda\,(\,)\bullet \text{empty\_}\,(\,)$$

must be such a procedure, and if 'l' is such a procedure then

$$\lambda\,(\,)\bullet l\,(\,)\,;\,\text{add\_}\,(\,e\,)$$

(which signifies what happens when an application of 'l' is followed by one of 'add_') must also be such a procedure. However, not every member of the type

$$\textbf{Unit} \overset{\sim}{\to} \textbf{write any Unit}$$

can be a member of 'List': to be so, it must represent a sequence of applications of 'empty_' and 'add_'. There is therefore a representation invariant which restricts attention to the subtype of

$$\textbf{Unit} \overset{\sim}{\to} \textbf{write any Unit}$$

comprising members of 'List'. This representation invariant is provided by the predicate 'is_list'. It could be defined by primitive recursion; however, in practice it is more convenient to ignore 'is_list' and use instead an induction rule to describe all the possible members of 'List', as in 8.4. This induction rule is in fact that provided in 8.1, but relies on the definitions

$$\text{empty} = \lambda\,(\,)\bullet \text{empty\_}\,(\,)$$

and

$$\text{add}\,(\,e\,,\,l\,) = \lambda\,(\,)\bullet l\,(\,)\,;\,\text{add\_}\,(\,e\,)$$

Here 'add' is defined to be a function which, when applied to an element and a procedure representing a list, returns a procedure representing a list. It is therefore a higher order function.

Similarly 'tail' is defined to be a function which, when applied to a procedure representing a list, returns a procedure representing a list. It satisfies

$$\text{tail}\,(\,l\,) = \lambda\,(\,)\bullet l\,(\,)\,;\,\text{tail\_}\,(\,)$$

When it is applied to non-empty lists, 'tail' cannot construct any lists that cannot already be constructed using 'empty' and 'add', owing to the assertion

$$\Box\,(\,\text{add\_}\,(\,e\,)\,;\,\text{tail\_}\,(\,)\equiv(\,)\,)$$

inherited by the specification in 8.4 from the specification in 8.3 which changes the names of the procedures in 8.2. This assertion ensures that given the functions in 8.4, for every element 'e' and for every list 'l',

$$
\begin{aligned}
&\text{tail} (\text{add} (e , l)) = \\
&(\lambda ( ) \bullet \text{add} (e , l) ( ) ; \text{tail\_} ( )) = \\
&(\lambda ( ) \bullet l ( ) ; \text{add\_} (e) ( ) ; \text{tail\_} ( )) = \\
&(\lambda ( ) \bullet l ( ) ; ( )) = \\
&(\lambda ( ) \bullet l ( )) = \\
&l
\end{aligned}
$$

However, 'head' is applied to lists but does not return lists. In effect it must execute a procedure representing a list and then execute 'head_' in order to extract information from the store. The information extracted should depend only on the list; in other words, it should depend only on the effect of executing, in any store, the procedure representing the list. Consequently 'head' should satisfy

$$\text{head} (l) = \textbf{let } e : \text{Element} \bullet \Box (e \equiv \textbf{result} (l ( ) ; \text{head\_} ( ))) \textbf{ in } e \textbf{ end}$$

It is not immediately obvious that this assertion is appropriate, because it is not immediately apparent that

$$\textbf{result} (l ( ) ; \text{head\_} ( ))$$

represents the element at the head of the list independently of the store. Indeed, if 'l' is 'empty' then nothing in the axioms of 8.2 ensures that this is so; also, if the execution of 'l ( )' did not terminate then

$$\textbf{result} (l ( ) ; \text{head\_} ( ))$$

could be defined arbitrarily. However, there is no need for 'head (l)' to be useful for every 'l'; all that is necessary is that it be useful for non-empty lists. In fact induction demonstrates that 'l ( )' is always deterministically convergent for every list 'l', because the specification in 8.4 inherits the assertions

$$\Box\ \textbf{definite}\ \text{empty\_} ( )$$

and

$$\Box\ \textbf{definite}\ \text{add\_} (e)$$

from the specification in 8.3. Furthermore, as 'l ( )' is deterministically convergent, the inherited assertion

$$\Box (\text{add\_} (e) ; \text{head\_} ( ) \equiv \text{add\_} (e) ; e)$$

for every element 'e' guarantees that

$$
\begin{aligned}
&\text{head} (\text{add} (e , l)) \equiv \\
&\textbf{result} (\text{add} (e , l) ( ) ; \text{head\_} ( )) \equiv \\
&\textbf{result} (l ( ) ; \text{add\_} (e) ; \text{head\_} ( )) \equiv \\
&\textbf{result} (l ( ) ; \text{add\_} (e) ; e) \equiv \\
&\textbf{result}\ e \equiv \\
&e
\end{aligned}
$$

The connection between 'is_empty' and 'is_empty_' is similar to that between 'head' and 'head_'.

2.4 The limitations of the illustration

The transformation of an applicative specification of lists into an imperative one illustrated in 2.2 appears simple. However, it cannot instantly be formalised in a way which covers types other than lists as specified in 8.1. The reason for this is that these lists are subject to various limitations and treating them offers no hints about how to deal with the following problems.

Equalities satisfied only as behavioural equivalences

The proof that

$$\text{tail} (\text{add} (e , l)) = l$$

for the specification in 8.4 depends crucially on the assertion

$$\Box (\text{add\_} (e) ; \text{tail\_} ( ) \equiv ( ))$$

However, this assertion may be invalid: in an implementation an application of 'add_' may modify the store in a way which is not reversed by 'tail_' but which is irrelevant to the observable behaviour of lists. If this happens, the correctness proof outlined in 2.3 breaks down. This can happen in an implementation of lists like that in 8.12 which, by contrast with that in 8.5, does not assume that lists are available as members of a concrete data type which can be held in single variables.

Specifications containing inadequately defined functions

The proof that

$$\text{head} (\text{add} (e , l)) = e$$

for the specification in 8.4 depends crucially on the fact that an application of 'head_' returns a result which is independent of the store, provided that, immediately before, a procedure representing a non-empty list is executed; as 'add_ (e)' is always deterministically convergent this fact is evidently ensured by the assertion

$$\Box (\text{add\_} (e) ; \text{head\_} ( ) \equiv \text{add\_} (e) ; e)$$

If now the specification in 8.1 is extended with the declaration

$$\text{unhead} : \text{List} \overset{\sim}{\rightarrow} \text{Element}$$

and the assertion

$$\sim (\text{unhead} (\text{add} (e , l)) = e)$$

(with the intention, perhaps, that 'unhead' is a choice function of some kind when 'Element' contains at least two members), then the specification in 8.2 must be extended in a corresponding manner, which when inherited in turn by the specifications in 8.3 and 8.4 gives the declaration

$$\text{unhead\_} : \textbf{Unit} \overset{\sim}{\rightarrow} \textbf{read any} \ \text{Element}$$

and the assertion

$$\Box \sim (\text{add\_} (e) \, ; \, \text{unhead\_} (\,) \equiv \text{add\_} (e) \, ; \, e)$$

This assertion does not at all guarantee that an application of 'unhead_' ever returns a result which is independent of the store. The problem arises not because of the use of '$\sim$' but because in a certain sense 'unhead' is inadequately defined. Realistic examples where the problem arises are provided by name generators (which allow the generation of hitherto unused names when a particular function is applied) and, in a less troublesome manner, by maps and queues (which allow the extraction of elements other than the ones most recently added).

Assertions about more than one object

The specification in 8.2 relies on the restriction that only one list need ever be identified in any of the assertions in 8.1: none of the functions acting on lists ever acts on more than one list at once. This restriction in the specifications is quite common and is ruthlessly exploited in object-oriented programming languages, which typically declare the procedures acting on the objects in a class alongside the variables private to an object. However, sometimes the restriction is violated; for example, the specification in 8.1 might be extended with the declaration

$$\text{join} : (\text{List} \times \text{List}) \overset{\sim}{\rightarrow} \text{List}$$

and with assertions which mention more than one list, such as

$$\text{join} (\text{add} (e , l_1) , l_2) = \text{add} (e , \text{join} (l_1 , l_2))$$

When a class definition must consider more than one object in the class at once, object-oriented programming languages typically resort to a syntactic device (**self** or **this**) to indicate the instance with which the procedures are associated. This device is both ugly and irrelevant to the issue for specifications, which involves finding ways of providing imperative implementations for functions like 'join'. Other examples where the issue manifests itself are provided by trees (as a tree is usually constructed from more than one tree at once) and by equality functions implemented using equality procedures; this latter case is the one handled in the specification in 8.9.

Relations between constructed values

The specification in 8.1 contains an induction rule which indicates that all the members of 'List' can be constructed using 'empty' and 'add'. However, it contains no axioms relating an application of 'add' to another application of 'add' or to 'empty': the equations in it just define functions ('head', 'tail' and 'is_empty') in terms of 'empty' and 'add'. A specification concerned with sets instead of lists might contain axioms relating applications of 'constructor' functions like 'add'. The first and second of the transformation techniques discussed in this paper do deal with such axioms. The third does not do so in the form presented here; however, it can be adapted to apply to specifications which do not require such axioms but which can nevertheless treat members of types as having different constructions but the same observable behaviour.

2.5 The general techniques

In order to handle the problems mentioned in 2.4, it is necessary to impose constraints on the specifications to be transformed and to introduce extra functions. The resulting process is sketched below. It is split into three stages, involving different transformation techniques, because departing from the process at some stages can allow implementations to be optimised or specifications to be implemented despite violating the assumptions underlying subsequent stages. The stages are discussed in more detail in §5, §6 and §7.

Interpreting equalities as behavioural equivalences

When verifying that a refinement of a specification is correct, '=' in the specification may need to be interpreted as *behavioural equivalence* (equality of observable behaviour) rather than as equality in the refined specification. To achieve this, given the view of refinement outlined in 4.2, the specification must be transformed so that '=' is replaced by a conventional operation. If it is to embody behavioural equivalence, this operation should distinguish between the values of two expressions if and only if observations of behaviour can distinguish between the expressions. When this operation is itself intended to be implemented conditions must be imposed on the applicative specification; these ensure that the functions are adequately defined, in that (in a certain sense) they depend only on the construction of their parameters. The transformation of the equalities into equivalences, and a sufficient check on the adequacy of the definitions of the functions, could be automated.

Converting functions into procedures

An applicative specification can be transformed into an imperative specification, provided that the functions in the applicative specification are adequately defined; if the functions are adequately defined the procedures in the imperative specification are influenced only by relevant aspects of objects. If the functions in the applicative specification depend on more than one member of the types to be implemented, then the procedures must satisfy conditions which ensure that they do not interfere with each other when they act on different objects; only if this is so does the imperative specification give rise to a 'refinement' of the applicative one, in the sense discussed in 4.2. The transformation of the functions into procedures, and a sufficient check that the procedures in a pre-existing specification do not interfere with one another, could be automated.

Replacing axioms by programs

The axioms in a specification can be replaced by programs, using a technique which is specific to a given class of abstract data types; the class considered in this paper is quite wide. The transformation of the axioms into programs, and a sufficient check that a given abstract data type is in this class, could be automated.

3 Notation

3.1 Functions and procedures

For types 't_1' and 't_2',

$$t_1 \overset{\sim}{\rightarrow} t_2$$

signifies the type of applicative partial functions which take members of the parameter type, 't_1', as parameters and which may return members of the result type, 't_2', as results. By contrast,

$$t_1 \overset{\sim}{\rightarrow} \textbf{write any } t_2$$

signifies the type of imperative partial procedures which take members of 't_1' as parameters, which may read from or write to any accessible variable, and which may return members of 't_2' as results. In addition,

$$t_1 \overset{\sim}{\rightarrow} \textbf{read any } t_2$$

signifies the subtype of this type of procedures comprising those procedures which may read from, but not write to, any accessible variable.

If types 't_1' and 't_2' have subtypes 't_3' and 't_4' respectively then

$$t_3 \overset{\sim}{\rightarrow} t_4$$

consists of those applicative partial functions ('f', say) in

$$t_1 \overset{\sim}{\rightarrow} t_2$$

such that when the execution of an application of 'f' to a parameter in 't_3' terminates then the result is in 't_4'; in other words,

$$\begin{aligned}
&t_3 \overset{\sim}{\rightarrow} t_4 = \\
&\{ f \mid \\
&\quad f : t_1 \overset{\sim}{\rightarrow} t_2 \bullet \\
&\qquad \forall x_3 : t_3 \bullet (\exists x_2 : t_2 \bullet x_2 = f (x_3)) \Rightarrow (\exists x_4 : t_4 \bullet x_4 = f (x_3)) \}
\end{aligned}$$

3.2 Units

The type **Unit** has a unique (trivial) member. Accordingly,

$$t_1 \overset{\sim}{\rightarrow} \textbf{write any Unit}$$

is the type of procedures which take members of the type 't_1' as parameters, which may read from or write to any accessible variable, and which may return trivial results; such procedures are effectively "without results" and are used only because they modify the store. Also,

$$\mathbf{Unit} \overset{\sim}{\rightarrow} \mathbf{write\ any}\ t_2$$

is the type of procedures which take trivial parameters, which may read from or write to any accessible variable, and which may return members of the type 't_2' as results; such procedures are effectively "without parameters".

The unique member of **Unit** can be written as '( )'. The application of a function or procedure ('f', say) to a parameter of type **Unit** can be shortened to 'f ( )'. A function abstraction 'λ x : **Unit** • e' such that 'x' is not mentioned in 'e' can be shortened to 'λ ( ) • e'.

3.3 Specifying the effects of expressions

For any expression 'e' having type **Bool**

$$\Box\ e$$

indicates that 'e' is equal to **true** for every store, no matter what values have been written to the variables.

For any expressions 'e_1' and 'e_2' with the same types evaluating the equivalence

$$e_1 \equiv e_2$$

returns **true** for the current store if and only if the executions of 'e_1' and 'e_2' in the current store have identical effects. These effects may be to modify the store and to return results. The evaluation of the equivalence itself does not modify the store; it merely returns a member of **Bool** by comparing the modifications to the store and the results returned. Moreover, the executions of 'e_1' and 'e_2' do not have to terminate in order to make the evaluation of the equivalence return **true**; hence a specification which relies on '$\equiv$' (as opposed to one which relies on '=') typically needs to include explicit assertions to ensure that executions terminate.

As with '$\forall$' and 'λ' in higher order logic,

$$\begin{aligned}&(\ \Box\ (\ e_1 \equiv e_2\ )\ ) = \\ &(\ (\ \lambda\ (\ )\ \bullet e_1\ ) = (\ \lambda\ (\ )\ \bullet e_2\ )\ )\end{aligned}$$

For example, the assertion

$$\text{empty} = \lambda\ (\ )\ \bullet \text{empty\_}\ (\ )$$

in 8.4 is equivalent with

$$\Box\ (\ \text{empty}\ (\ ) \equiv \text{empty\_}\ (\ )\ )$$

or indeed with

$$\text{empty} = \text{empty\_}$$

In this paper '$\Box$' is given the same precedence as the quantifiers (which are given higher precedence than implication and conjunction). Also, '$\equiv$' is given higher precedence than ';'.

3.4 Describing the results of expressions

For any expression 'e' without input and output, the assertion that in the current store 'e' is *deterministically convergent*, in that its execution terminates with a unique store and with a unique result, is written in this paper as

$$\textbf{definite}\ \mathrm{e}$$

In the RAISE specification language this is

$$\mathrm{e}\ \textbf{post}\ \textbf{true}$$

The result returned by the execution of 'e' is signified by

$$\textbf{result}\ \mathrm{e}$$

Provided that 'e' is deterministically convergent, in the RAISE specification language this is

$$\textbf{let}\ \mathrm{i} : \mathrm{t} \bullet (\ \mathrm{e} \equiv (\ \mathrm{e}\ ;\ \mathrm{i}\ )\ )\ \textbf{in}\ \mathrm{i}\ \textbf{end}$$

Here executing the expression 'e ; i' involves executing 'e', discarding its result, and executing 'i' (which simply returns as its result the value of the constant 'i'). (Also, **let** is used to define 'i' because the language does not provide 'ϵ'.)

Much as with 'ι' and '$\exists!$', '**result** e' is usable if '**definite** e' evaluates to **true**:

$$\textbf{definite}\ \mathrm{e} \Rightarrow \exists!\ \mathrm{i} : \mathrm{t} \bullet (\ \mathrm{e} \equiv (\ \mathrm{e}\ ;\ \mathrm{i}\ )\ )$$

For example, in the imperative specification in 8.2 if the assertion

$$\Box\ \textbf{definite}\ \mathrm{add}\ (\ \mathrm{e}\ )$$

holds, then

$$\Box\ (\ \mathrm{add}\ (\ \mathrm{e}\ )\ ;\ \mathrm{head}\ (\ )\ \equiv \mathrm{add}\ (\ \mathrm{e}\ )\ ;\ \mathrm{e}\ )$$

is equivalent with

$$\Box\ \textbf{definite}\ (\ \mathrm{add}\ (\ \mathrm{e}\ )\ ;\ \mathrm{head}\ (\ )\ )\ \wedge\ (\ \textbf{result}\ (\ \mathrm{add}\ (\ \mathrm{e}\ )\ ;\ \mathrm{head}\ (\ )\ ) \equiv \mathrm{e}\ )$$

or indeed with

$$\Box\ \textbf{definite}\ \mathrm{head}\ (\ \mathrm{add}\ (\ \mathrm{e}\ )\ )\ \wedge\ (\ \textbf{result}\ \mathrm{head}\ (\ \mathrm{add}\ (\ \mathrm{e}\ )\ ) \equiv \mathrm{e}\ )$$

In 2.3 and elsewhere it is necessary to require that the sequential composition of deterministically convergent expressions be deterministically convergent and that the results of the expressions be properly related; in other words, for all expressions 'e_1' and 'e_2',

$$\begin{array}{l}(\ \textbf{definite}\ \mathrm{e}_1 \wedge (\ \textbf{result}\ (\ \mathrm{e}_1\ ;\ \textbf{definite}\ \mathrm{e}_2\ ) \equiv \textbf{true}\ )\ ) \Rightarrow \\ (\ \textbf{definite}\ \ (\ \mathrm{e}_1\ ,\ \mathrm{e}_2\ ) \wedge \\ \quad (\ \textbf{result}\ \ (\ \mathrm{e}_1\ ,\ \mathrm{e}_2\ ) \equiv (\ \textbf{result}\ \mathrm{e}_1\ ,\ \textbf{result}\ (\ \mathrm{e}_1\ ;\ \textbf{result}\ \mathrm{e}_2\ )\ )\ )\ )\end{array}$$

If there are non-deterministic constructs in the language, this requirement is not met just by demanding that the results of the expressions be unique; they must also modify the store in ways that are deterministic. As an illustration of this, in the RAISE specification language

$$(n := 0 \sqcap n := 1) ; n$$

does not have a unique result, though both

$$(n := 0 \sqcap n := 1)$$

and

$$n$$

do have unique results.

Both **definite** and **result** have precedence no higher than the precedence of '$\equiv$'.

4 Concepts

4.1 The form of types

The types declared in a specification may be either *sorts* or abbreviations for other types. Sorts are not interpreted further in the specification but may be constrained by axioms. Types may be pre-defined types (such as **Bool**), sorts or composite types composed by applying pre-defined operators (such as $\times$) to other types.

A *flat* type is one that can be composed without the use of function type operators like those in 3.1.

A subset of a type may be a type. Hence possible declarations of types include

List

and, when 'is_list' is a predicate defined on members of 'List_',

List = { l | l : List_ • is_list (l) }

The range of types permitted allows specifications to be either model-oriented (as in VDM [9]) or property-oriented (as in OBJ [1]). However, the only types that may be refined are sorts. Where a type is expected to be refined, it should be treated as a sort; a concrete data type can be used to model this sort by introducing an 'observer' function which has the concrete data type as its result type. An example of this is provided by extending the specification of lists in 8.1 with the declaration

gather : List $\overset{\sim}{\rightarrow}$ Element-**list** and the assertions

$$\text{gather (empty)} = \langle\,\rangle\ ,$$
$$\text{gather (add (e , l))} = \langle\text{ e }\rangle \mathbin{^\frown} \text{gather (l)}$$

In fact, for all 'l_1' in 'List' and for all 'l_2' in 'List',

$$\text{gather (} l_1 \text{)} = \text{gather (} l_2 \text{)} \Rightarrow l_1 = l_2$$

but 'List' is not identified with 'Element-**list**', just as an abstract data type of trees is not identified with the set of encodings of trees as lists.

Behavioural abstraction requires that the members of a sort can be distinguished from one another by observations. It therefore requires the selection of certain *observable* types; all other types are *unobservable*. The observable types are those composed from pre-defined types and observable sorts by applying pre-defined operators. A sort which is regarded as observable at one stage in a development may be regarded as unobservable at a subsequent stage.

4.2 The nature of refinement

In this paper, a refinement of a specification is another specification which allows one to make the same assertions (and possibly more besides). This view of refinement (as theory extension) seems common to HOL [2], Larch [4] and RAISE [11] (though the emphasis on refinement, and the underlying logic, differ between the cases cited). It requires behavioural abstraction from model-oriented specifications to be treated explicitly (as exemplified in 4.1). A more general view, adopted for Extended ML [14], permits behavioural abstraction to be treated more succinctly but needs reasoning about model classes when there are no suitable proof rules; such proof rules are beginning to emerge, but they are of little help if the equivalence induced by the behavioural abstraction is itself intended to be implemented.

Here, as for the RAISE specification language, a specification can contain declarations (of types, constants and functions, among other things) and axioms, and refining a specification can involve turning sorts into abbreviations for other types, adding extra declarations or adding extra axioms.

4.3 The form of specifications

Applicative specifications are often expressed as conditional equations plus induction rules. Each conditional equation may have *quantified names*, which are named members of types bound by universal quantifiers, a *premise*, which is a finite conjunction of equations between expressions having observable types, and a *consequence*, which is a finite conjunction of equations between expressions having observable or unobservable types. Each induction rule takes a standard form, with a finite conjunction of *hypotheses* that themselves have quantified names. premises and consequences; the premises help to indicate when the functions occurring subsequently in the premises or the consequences are applicable. In order to let behavioural equivalences take simple forms, either premises should be allowed to include universal quantifiers binding members of observable types or types consisting of finite sets (or lists) of members of observable types should be observable.

In practice certain abbreviations to conditional equations are permitted. For instance, an equation taking the form '$e = \textbf{false}$' may be shortened to '$\sim e$'.

It is not a severe practical restriction to require that applicative specifications of abstract data types rely on conditional equations and induction rules. In particular, specifications of applicative functions using pre-conditions and total correctness post-conditions, as in VDM, are equivalent with ones using conditional equations. For instance,

$$\text{add} (e , l) \ \textbf{as} \ l' \ \textbf{post} \ (\text{head} (l') = e \wedge \text{tail} (l') = l)$$

is equivalent with

$$\text{head} (\text{add} (e , l)) = e \wedge \text{tail} (\text{add} (e , l)) = l$$

In fact specifications of imperative procedures using pre-conditions and total correctness post-conditions can also be turned into things akin to conditional equations by using '$\equiv$' and **definite** (discussed in 3.3 and 3.4).

Hereafter the definitions assume that applicative specifications are expressed as conditional equations plus induction rules.

4.4 Constructors, inspectors and observers

The functions in applicative specifications can usually be defined in terms of ***constructor*** constants and functions, ***inspector*** functions and ***observer*** functions. The types of the constants, and the parameter and result types of the functions, are flat. (In the specification of lists in 8.1, 'empty' and 'add' are constructors, 'tail' is an inspector, and 'head' and 'is_empty' are observers.)

The constructors are the constants and functions used in the (consequences of hypotheses in the) induction rules: every consequence applies a predicate to a constructor constant or function. Each constructor constant has an unobservable type and each constructor function has an unobservable result type. (A stronger version of this assumption, to the effect that the types are actually sorts, would preclude treating procedures which both return results and have effects on the store.) Though in principle this use of the term 'constructor' is different from that in term rewriting theory [8], in practice it is very similar.

The inspectors and observers are the functions used to distinguish between different members of sorts. An inspector function serves to extract other members of an unobservable sort from a given member, so it has an unobservable result type. An observer function serves to observe the members of an unobservable sort, so it has an observable result type.

These terms need to be generalised somewhat (though the generalisation can largely be ignored in the reading of this paper). Loosely, a *generalised* constructor is defined using conditional equations formed from suitable expressions using constructors; a generalised inspector or observer is defined using conditional equations formed from suitable expressions using inspectors and observers (in such a way that the result type is unobservable for an inspector and observable for an observer). A *suitable* expression is a constant, a quantified name, an application of a function to a suitable expression, a product of suitable expressions, or a component of a product. (For the specification of lists in 8.1, 'λ l : List • (head (l) , tail (l))' is a generalised inspector.)

In any application of a generalised constructor function there is a greatest depth of nesting of constructor functions. As the constructors constitute the constants and functions used in the induction rules, every member of an unobservable sort can be expressed in terms of them (though the means of expressing it may not be apparent in the absence of extra conditions). The *construction level* of a member of an unobservable type is the greatest depth of nesting of constructor functions needed by the "shallowest" means of constructing the member by applying generalised constructor functions to generalised constructor constants.

4.5 Applicability functions

Associated with every generalised constructor, inspector or observer function ('f', say) there is its *applicability* function, '*applicable*[[f]]'. This is a generalised observer which provides the pre-condition indicating when 'f' is applicable in the conditional equations and induction rules; it is a total function. If 'f' has type '$t_1 \overset{\sim}{\rightarrow} t_2$' then '*applicable*[[f]]' has type '$t_1 \overset{\sim}{\rightarrow}$ **Bool**'. The applicability function for 'f' can (for the present) be any total function which, when applied to any parameter to which 'f' is applied in the specification, evaluates to **true**. (In fact the applicability functions used in this paper can be devised by a systematic process, the details of which are omitted.)

4.6 Destructors

Associated with a constructor function ('c', say) there may be a *destructor* function, '*destructor*[[c]]'. This is a generalised inspector which acts as a left inverse for ‘c’ when ‘c’ is applicable. If ‘c’ has type ‘$t_1 \xrightarrow{\sim} t_2$’ then when ‘*destructor*⟦c⟧’ exists it has type ‘$t_2 \xrightarrow{\sim} t_1$’ and

$$\begin{array}{l} \forall\ x_1 : t_1 \bullet \\ \quad \mathit{applicable}[\![c]\!]\ (\ x_1\ ) \Rightarrow \\ \quad \mathit{destructor}[\![c]\!]\ (\ c\ (\ x_1\ )\ ) = x_1 \end{array}$$

For example, for the lists specified in 8.1, ‘λ l : List • (head (l) , tail (l))’ is a destructor function for ‘add’. There may not be any destructor functions: a specification of sets, for instance, would not have them (because set union is commutative) but could have inspectors and observers.

4.7 Discrimination functions

Associated with a constructor constant (‘k’, say) there may be a *discrimination* function, ‘*discriminant*⟦k⟧’. This is a generalised observer which indicates when something must be ‘k’; it is a total function. If ‘k’ has type ‘t_2’ then when ‘*discriminant*⟦k⟧’ exists it has type ‘$t_2 \xrightarrow{\sim}$ **Bool**’ and, for every generalised constructor function ‘f’ of type ‘$t_3 \xrightarrow{\sim} t_2$’ such that ‘k’ is not mentioned in ‘f’ but another constructor is mentioned in ‘f’,

$$\begin{array}{l} \forall\ x_3 : t_3 \bullet \\ \quad \mathit{applicable}[\![f]\!]\ (\ x_3\ ) \Rightarrow \\ \quad \mathit{discriminant}[\![k]\!]\ (\ k\ ) \wedge \sim \mathit{discriminant}[\![k]\!]\ (\ f\ (\ x_3\ )\ ) \end{array}$$

For example, for the lists specified in 8.1, ‘is_empty’ is a discrimination function for ‘empty’.

Associated with a constructor function (‘c’, say) there may be a discrimination function, ‘*discriminant*⟦c⟧’. This is a generalised observer which indicates when something can only be constructed by an application of ‘c’; it is a total function. If ‘c’ has type ‘$t_1 \xrightarrow{\sim} t_2$’ then ‘*discriminant*⟦c⟧’ has type ‘$t_2 \xrightarrow{\sim}$ **Bool**’ and, for every generalised constructor function ‘f’ of type ‘$t_3 \xrightarrow{\sim} t_2$’ such that ‘c’ is not mentioned in ‘f’ but another constructor is mentioned in ‘f’,

$$\begin{array}{l} \forall\ x_1 : t_1\ ,\ x_3 : t_3 \bullet \\ \quad \mathit{applicable}[\![c]\!]\ (\ x_1\ ) \wedge \mathit{applicable}[\![f]\!]\ (\ x_3\ ) \Rightarrow \\ \quad \mathit{discriminant}[\![c]\!]\ (\ c\ (\ x_1\ )\ ) \wedge \sim \mathit{discriminant}[\![c]\!]\ (\ f\ (\ x_3\ )\ ) \end{array}$$

For example, for the lists specified in 8.1, ‘λ l : List • ∼ is_empty (l)’ is a discrimination function for ‘add’. There may not be any discrimination functions: an assertion such as

$$\text{join}\ (\ \text{add}\ (\ e\ ,\ l_1\ )\ ,\ l_2\ ) = \text{add}\ (\ e\ ,\ \text{join}\ (\ l_1\ ,\ l_2\ )\ )$$

(which is discussed in 2.4) precludes ‘add’ and ‘join’ from having discrimination functions if both ‘add’ and ‘join’ are classified as constructors. (However, in the case discussed in 2.4, ‘join’ would be not be classified as a constructor, because the induction rule would use only ‘empty’ and ‘add’.)

5 Interpreting equalities as behavioural equivalences

5.1 Survey

The first stage in refining applicative specifications into imperative implementations entails interpreting equalities as behavioural equivalences. The technique needed replaces tests for equality between the members of unobservable types by applications of functions having result type **Bool**. The specification must obey conditions which ensure that no implicit uses of equality in it cannot be made explicit. These conditions are rather weaker than 'stability' requirements [15], which also prohibit the use of set and map operators involving abstract data types.

The technique is described in 5.2 and illustrated in 5.3. Assumptions that it requires are mentioned in 5.4. Variants of it are discussed in 5.5. The main assumption about the applicative specification is formalised in 5.6.

5.2 Tasks

5.2.1 Transforming the specification to interpret equality

- For each unobservable sort with name 'S', a function

  $$\mathrm{eq\_s} : (\ \mathrm{S} \times \mathrm{S}\ ) \xrightarrow{\sim} \mathbf{Bool}$$

  is introduced. This function is intended to replace '=' for 'S'.

- For each such 'S', 'eq_s' is postulated to be an equivalence (which is reflexive, symmetric and transitive). In fact this amounts to asserting that 'eq_s' is symmetric and is a congruence for the function 'eq_s', in the sense that when two members of 'S' are related by 'eq_s' then one may be substituted for the other in parameters of the function.

- The explicit uses of '=' in the specification are replaced by uses of 'eq_s' (with appropriate modifications when the types of the left and right hand sides of the equality are not unobservable sorts).

- The implicit uses of '=' which exploit cardinality assertions are replaced by uses of 'eq_s'. For example,

  $$\exists!\ \mathrm{i} : \mathrm{t} \bullet \mathrm{e}$$

  needs to be regarded as

  $$(\ \exists\, \mathrm{i} : \mathrm{t} \bullet \mathrm{e}\ ) \wedge (\ \forall\, \mathrm{i}_1 : \mathrm{t}\ ,\ \mathrm{i}_2 : \mathrm{t} \bullet (\ \mathrm{e}[\mathrm{i}_1/\mathrm{i}] \wedge \mathrm{e}[\mathrm{i}_2/\mathrm{i}]\ ) \Rightarrow \mathrm{i}_1 = \mathrm{i}_2\ )$$

  Cardinality assertions also arise from size functions (such as **card** for sets).

- The implicit uses of '=' which exploit congruence properties are replaced by uses of 'eq_s'. The congruence properties of '=' are used in the application of functions to parameters, so there must be assertions to ensure that 'eq_s' is a congruence for all the functions declared in the specification (including the functions which are used as predicates in the induction rules); for instance, if 'f' is a function having type '$S \xrightarrow{\sim} S$' then it must satisfy

  $$\begin{aligned}&\forall\, s_1 : S\ ,\ s_2 : S \bullet\\ &\quad \mathrm{eq\_s}\,(\,s_1\ ,\ s_2\,) \Rightarrow \mathrm{eq\_s}\,(\,f\,(\,s_1\,)\ ,\ f\,(\,s_2\,)\,)\end{aligned}$$

  This style of assertion can be used whatever the parameter and result types of the function may be.

- Given that the functions introduced in the specification satisfy the conditions outlined in 5.6, there is no need to postulate that 'eq_s' is an equivalence and a congruence for all the functions declared in the specification. Indeed, 'eq_s' can be defined as a total function using just the observers and inspectors, applied when their applicability functions permit; for instance, if there is only one inspector function 'i' and only one observer function 'o' and if they both have 'S' as parameter type then 'eq_s' can satisfy

  $$\begin{aligned}&\forall\, s_1 : S\ ,\ s_2 : S \bullet\\ &\quad \mathrm{eq\_s}\,(\,s_1\ ,\ s_2\,) =\\ &\quad (\,(\,\mathit{applicable}[\![i]\!]\,(\,s_1\,) \wedge \mathit{applicable}[\![i]\!]\,(\,s_2\,) \Rightarrow\\ &\qquad\quad \mathrm{eq\_s}\,(\,i\,(\,s_1\,)\ ,\ i\,(\,s_2\,)\,)\,) \wedge\\ &\qquad (\,\mathit{applicable}[\![o]\!]\,(\,s_1\,) \wedge \mathit{applicable}[\![o]\!]\,(\,s_2\,) \Rightarrow\\ &\qquad\quad o\,(\,s_1\,) = o\,(\,s_2\,)\,)\,)\end{aligned}$$

  This style of definition can be used whenever the parameter types of the inspector and observer functions each contain at most one occurrence of an unobservable sort; however, when these types are not sorts, the premises in a specification must include universal quantifiers binding members of observable types if the definition is to be turned into conditional equations.

5.2.2 Renaming the specification to distinguish names

- The specification, thus transformed, is subject to renaming by changing the names declared in it to avoid clashes with the names declared in the original specification. (Below, the name of an unobservable sort 'S' is taken to be changed by this means to 'S_'.)

5.2.3 Extending the specification to show refinement

- For each hitherto unobservable sort with name 'S', the declaration

  $$\begin{aligned}&S =\\ &\quad \{\,s\ |\\ &\qquad s : S\_\textbf{-infset} \bullet\\ &\qquad\quad s \neq \{\,\} \wedge\\ &\qquad\quad \forall\, s_{\_1} : S\_\ ,\ s_{\_2} : S\_ \bullet s_{\_1} \in s \Rightarrow \mathrm{eq\_s}\,(\,s_{\_1}\ ,\ s_{\_2}\,) = (\,s_{\_2} \in s\,)\,\}\end{aligned}$$

is provided. The type 'S' comprises the equivalence classes of members of 'S_' with respect to 'eq_s', so '=' for it is the equality between equivalence classes.

- For each such 'S', functions

  $$\mathrm{abs\_s} : \mathrm{S\_} \xrightarrow{\sim} \mathrm{S} ,$$
  $$\mathrm{rep\_s} : \mathrm{S} \xrightarrow{\sim} \mathrm{S\_}$$

  are introduced. These functions constitute the main use of abstraction and representation functions required by the view of refinement adopted in this paper.

- For each such 'S', 'abs_s' and 'rep_s' are made to satisfy

  $$\forall\ \mathrm{s\_1} : \mathrm{S\_} \bullet \mathrm{abs\_s}\ (\ \mathrm{s\_1}\ ) = \{\ \mathrm{s\_2} \mid \mathrm{s\_2} : \mathrm{S\_} \bullet \mathrm{eq\_s}\ (\ \mathrm{s\_1}\ ,\ \mathrm{s\_2}\ )\ \}\ ,$$
  $$\forall\ \mathrm{s} : \mathrm{S} \bullet \mathrm{rep\_s}\ (\ \mathrm{s}\ ) \in \mathrm{s}$$

  Consequently

  $$\forall\ \mathrm{s\_1} : \mathrm{S\_} \bullet \mathrm{eq\_s}\ (\ \mathrm{s\_1}\ ,\ \mathrm{rep\_s}\ (\ \mathrm{abs\_s}\ (\ \mathrm{s\_1}\ )\ )\ )\ ,$$
  $$\forall\ \mathrm{s} : \mathrm{S} \bullet \mathrm{s} = \mathrm{abs\_s}\ (\ \mathrm{rep\_s}\ (\ \mathrm{s}\ )\ )$$

  Here the abstraction function, 'abs_s', maps the members of equivalence classes to the equivalence classes, and the representation function, 'rep_s', maps the equivalence classes to representative members.

- Versions of the constants and functions declared in the original specification are defined using 'abs_s' and 'rep_s' according to the usual 'homomorphic' approach; for instance, if 'f_' is a function having type '$\mathrm{S\_} \xrightarrow{\sim} \mathrm{S\_}$' then the defined function 'f' must satisfy

  $$\forall\ \mathrm{s} : \mathrm{S} \bullet \mathrm{f}\ (\ \mathrm{s}\ ) = \mathrm{abs\_s}\ (\ \mathrm{f\_}\ (\ \mathrm{rep\_s}\ (\ \mathrm{s}\ )\ )\ )$$

  This style of definition can be used whatever the parameter and result types of the function may be.

- Given that the functions introduced in the specification satisfy the conditions outlined in 5.6, there must be induction rules. In this situation the constructor constants and functions can construct all the members of each unobservable sort, so the specification thus extended provides a refinement of the original specification in which '=' is replaced by a function which can be implemented further.

5.3 Example

5.3.1 Transforming the specification to interpret equality

The applicative specification of 'List' given in 8.1 can be transformed by applying the technique described in 5.2. The outcome of doing so is presented in 8.6.

5.3.2 Renaming the specification to distinguish names

The type, constant and functions declared in 8.6 must have their names changed before they can be compared directly with those in 8.1. This is done in 8.7.

5.3.3 Extending the specification to show refinement

To establish that the specification in 8.6 is an appropriate transformation of that in 8.1, versions of the type, constant and functions declared in the specification in 8.1 are defined by extending the specification in 8.7. The outcome of doing so is presented in 8.8.

5.4 Assumptions

In order to exploit the explicit construction of 'eq_s' in terms of inspectors and observers, the applicative specification must take the form described in 4.3 and the functions introduced in it must satisfy the conditions outlined in 5.6.

The parameter type of each inspector or observer function must contain only one occurrence of an unobservable sort. (This assumption is not merely convenient; it is central to defining 'eq_s'.)

In this situation induction on construction levels can demonstrate that 'eq_s', defined according to the scheme in 5.2, is a total function, an equivalence and a congruence for all the functions (including the constructors); accordingly it is sufficient to define 'eq_s', as is done in 8.6, and it is unnecessary to postulate that 'eq_s' is an equivalence and a congruence for the functions. In fact 'eq_s' is the interpretation of '=' which distinguishes between two members of 'S' if and only if they can be distinguished using applicable observations. (However, 'eq_s' depends on the applicability functions, which can be devised in various ways.)

The specification must contain no constructs that may read from or write to variables having unobservable types.

Every construct in the specification which implicitly makes a cardinality assertion must be capable of being rewritten to make the assertion explicit through the use of '='. In particular, if the language provides 'ι' it must provide 'ϵ' also.

Every construct in the specification which implicitly exploits a congruence property of '=' must be capable of being rewritten to make the property explicit through the use of '='. In particular, if the language provides types which are not flat then it must provide subtypes also.

5.5 Variants

The technique described in 5.2 is applicable even when the conditions outlined in 5.6 are not satisfied. When this happens, it is necessary to postulate that the constructor constants and functions can construct all the members of each unobservable sort (perhaps owing to induction rules) and that 'eq_s' is an equivalence and a congruence for all the functions. (For lists this is done in 8.13; the specification in 8.13 can be related to the specification in 8.1 by the same renamings and extensions as are adopted in 5.3 for the specification in 8.6.) It may still be necessary to ensure separately that 'eq_s' distinguishes members of 'S' only if they can be distinguished using observations.

The technique can be specialised as well as generalised. When the functions satisfy the conditions outlined in 5.6 and for each constructor function there is a corresponding destructor function, the only generalised inspectors that need be considered are the destructors. In particular, the scheme for the definition of equality in 5.2 need mention only the destructors instead of the inspectors. This can be established by induction on construction levels.

5.6 The adequacy of the definitions

The functions declared in a specification can be related in varied ways. However, they generally obey conditions that enable them to induce behavioural equivalences that can even be used in imperative specifications. These conditions ensure that applications of functions to parameters produce results which are determined solely by the way in which their parameters are constructed; they thereby facilitate induction on construction levels.

Induction on construction levels, as practiced in this paper, requires essentially that for any inspector 'i' and for any appropriate parameter 'x' the construction level of 'i (x)' is less than that of 'x'. To bring this about it is necessary to impose conditions on the constructors, inspectors and observers. The condition for the inspectors and observers is that they all be adequately defined, in a sense to be clarified below. The condition for the constructors is that no constructor function can produce a constructor constant as a result; in other words for any generalised constructor constant 'k' with type 't_2' (and with greatest depth of nesting of constructor functions 0) and for any generalised constructor function 'c' with type '$t_1 \xrightarrow{\sim} t_2$' (and with greatest depth of nesting of constructor functions 1) there must be a generalised observer 'o' with type '$t_2 \xrightarrow{\sim}$ **Bool**' such that

$$\begin{array}{l} \forall\, x_1 : t_1 \bullet \\ \quad \textit{applicable}[\![c]\!]\,(\,x_1\,) \Rightarrow \\ \quad o\,(\,k\,) \wedge \sim o\,(\,c\,(\,x_1\,)\,) \end{array}$$

An inspector 'i', with type '$t_2 \xrightarrow{\sim} t_3$', is *adequately defined* by a specification if the following conditions hold:

- for any generalised constructor constant 'k' with type 't_2' (and with greatest depth of nesting of constructor functions 0) it is the case that

  $$\sim \mathit{applicable}[\![i]\!] \ (\ k \ )$$

  so a constructor constant is never extracted by applying an inspector function, just as it is not constructed by applying a constructor function;

- for any generalised constructor function 'c' with type '$t_1 \xrightarrow{\sim} t_2$' (and with greatest depth of nesting of constructor functions 1) there can be inferred $m \geq 0$ conditional equations, each with the form

  $$\begin{aligned} &\forall \ x_1 : t_1 \ \bullet \\ &\quad \mathit{applicable}[\![c]\!] \ (\ x_1 \ ) \wedge p_l \ (\ x_1 \ ) = q_l \ (\ x_1 \ ) \Rightarrow \\ &\quad i \ (\ c \ (\ x_1 \ ) \ ) = c_l \ (\ i_l \ (\ x_1 \ ) \ ) \end{aligned}$$

  where

  $$\begin{aligned} &\forall \ x_1 : t_1 \ \bullet \\ &\quad \mathit{applicable}[\![c]\!] \ (\ x_1 \ ) \wedge \mathit{applicable}[\![i]\!] \ (\ c \ (\ x_1 \ ) \ ) \Rightarrow \\ &\quad p_1 \ (\ x_1 \ ) = q_1 \ (\ x_1 \ ) \vee \ldots \vee p_m \ (\ x_1 \ ) = q_m \ (\ x_1 \ ) \end{aligned}$$

  and where (for all l having $m \geq l \geq 0$) 'p_l' is a generalised observer function, 'q_l' is a generalised observer function, 'c_l' is a generalised constructor function and 'i_l' is a generalised inspector function such that the greatest depth of nesting of constructor functions in 'c_l' is no more than the least depth of nesting of inspector functions in 'i_l'.

For example, for the lists specified in 8.1, if

$$\sim \mathit{applicable}[\![tail]\!] \ (\ empty \ )$$

'tail' is adequately defined as, for all 'e' in 'Element' and for all 'l' in 'List',

$$tail \ (\ add \ (\ e \ , l \ ) \ ) = l$$

An observer 'o', with type '$t_2 \xrightarrow{\sim} t_3$', is *adequately defined* by a specification if the following conditions hold:

- for any generalised constructor constant 'k' with type 't_2' (and with greatest depth of nesting of constructor functions 0) there can be inferred an equation with the form

  $$\mathit{applicable}[\![o]\!] \ (\ k \ ) \Rightarrow o \ (\ k \ ) = b$$

  where 'b' is a constant which has an observable type and which can be defined without mentioning members of unobservable types;

- for any generalised constructor function 'c' with type '$t_1 \xrightarrow{\sim} t_2$' (and with greatest depth of nesting of constructor functions 1) there can be inferred $m \geq 0$ conditional equations, each with the form

$$\begin{array}{l} \forall\ x_1 : t_1 \bullet \\ \quad \mathit{applicable}[\![c]\!]\ (\ x_1\ ) \land p_l\ (\ x_1\ ) = q_l\ (\ x_1\ ) \Rightarrow \\ \quad o\ (\ c\ (\ x_1\ )\ ) = f_l\ (\ o_l\ (\ c_l\ (\ i_l\ (\ x_1\ )\ )\ )\ ) \end{array}$$

where

$$\begin{array}{l} \forall\ x_1 : t_1 \bullet \\ \quad \mathit{applicable}[\![c]\!]\ (\ x_1\ ) \land \mathit{applicable}[\![o]\!]\ (\ c\ (\ x_1\ )\ ) \Rightarrow \\ \quad p_1\ (\ x_1\ ) = q_1\ (\ x_1\ ) \lor \ldots \lor p_m\ (\ x_1\ ) = q_m\ (\ x_1\ ) \end{array}$$

and where (for all l having $m \geq l \geq 0$) 'f_l' is a function which has observable parameter and result types and which can be defined without mentioning members of unobservable types, 'p_l' is a generalised observer function, 'q_l' is a generalised observer function, 'o_l' is a generalised observer function, 'c_l' is a generalised constructor function and 'i_l' is a generalised inspector function such that the greatest depth of nesting of constructor functions in 'c_l' is no more than the least depth of nesting of inspector functions in 'i_l'.

For example, for the lists specified in 8.1, if

$$\sim \mathit{applicable}[\![\mathrm{head}]\!]\ (\ \mathrm{empty}\ )$$

'head' is adequately defined as, for all 'e' in 'Element' and for all 'l' in 'List',

$$\mathrm{head}\ (\ \mathrm{add}\ (\ e\ ,\ l\ )\ ) = e$$

However, an assertion such as

$$\sim (\ \mathrm{unhead}\ (\ \mathrm{add}\ (\ e\ ,\ l\ )\ ) = e\ )$$

(which is discussed in 2.4) is not enough to make 'unhead' adequately defined, even when it appears as the more orthodox conditional equation

$$\mathrm{unhead}\ (\ \mathrm{add}\ (\ e\ ,\ l\ )\ ) = e \Rightarrow \mathbf{true} = \mathbf{false}$$

The function 'unhead' could be made adequately defined in several ways; the simplest of them, which ensures that 'unhead (add (e , l))' depends only on 'e', entails introducing the declaration

$$\mathrm{reject} : \mathrm{Element} \xrightarrow{\sim} \mathrm{Element}$$

and the assertions

$$\begin{array}{l} \mathrm{unhead}\ (\ \mathrm{add}\ (\ e\ ,\ l\ )\ ) = \mathrm{reject}\ (\ e\ )\ , \\ \sim (\ \mathrm{reject}\ (\ e\ ) = e\ ) \end{array}$$

In order to ensure that the definitions of constants like 'b' above and functions like 'f_l' above do not depend on members of unobservable types, such constants and functions can be taken to be *fully defined* in a different specification.

A specification may introduce other functions beside the constructors, inspectors and observers; for instance, the function 'join' described in 2.4 is likely not to be a constructor or an inspector, at least if it is being defined for lists. These *primitively defined* functions are all taken to be 'primitive recursive', in that if 'f' is such a function having type '$t_1 \xrightarrow{\sim} t_2$' it must satisfy equations taking the general form

$$\begin{aligned}&\forall\, x_1 : t_1 \bullet\\ &\quad p_l\ (\ x_1\ ) = q_l\ (\ x_1\ ) \Rightarrow\\ &\quad f\ (\ x_1\ ) = g_l\ (\ f_l\ (\ i_l\ (\ x_1\ )\ )\ )\end{aligned}$$

where

$$\begin{aligned}&\forall\, x_1 : t_1 \bullet\\ &\quad \mathit{applicable}[\![f]\!]\ (\ x_1\ ) \Rightarrow\\ &\quad p_1\ (\ x_1\ ) = q_1\ (\ x_1\ ) \vee \ldots \vee p_m\ (\ x_1\ ) = q_m\ (\ x_1\ )\end{aligned}$$

and where (for all l having $m \geq l \geq 0$) 'p_l' is a generalised observer function, 'q_l' is a generalised observer function, 'g_l' is a generalised constructor, inspector or observer function, 'f_l' is a generalised primitively defined function and 'i_l' is a generalised inspector function such that the greatest depth of nesting of primitively defined functions in 'f_l' is no more than both 1 and the least depth of nesting of inspector functions in 'i_l'. (In fact this condition can be weakened.) Here a generalised primitively defined function is defined by conditional equations formed from suitable expressions using primitively defined functions.

Induction on construction levels can demonstrate that, if the observer and inspector functions in the specification are adequately defined and the functions other than the constructor, inspector and observer functions are primitively defined, then the specification is *sufficiently complete*: every suitable expression in which there are no occurrences of quantified names and in which the functions are applied only to parameters which satisfy their applicability functions can be reduced to a term which does not mention observer or inspector functions. (This notion of sufficient completeness is the counterpart for partial functions of the notion for total functions [3].)

6 Converting functions into procedures

6.1 Survey

The second stage in refining applicative specifications into imperative implementations entails converting functions into procedures. The technique needed turns declarations of, and assertions about, functions into declarations of, and assertions about, procedures. The functions must be defined so adequately that the procedures are influenced only by relevant aspects of the objects on which they act. Moreover, if the functions depend on more than one member of the types being implemented, then the procedures acting on different objects must not interfere with one another.

The technique is described in 6.2 and illustrated in 6.3. Assumptions that it requires are mentioned in 6.4. Variants of it are discussed in 6.5. The main constraint on the imperative specifications is formalised in 6.6.

6.2 Tasks

6.2.1 Transforming the specification to convert functions

- Any function having an unobservable parameter type is given the right to read from any accessible variable. Any function having an unobservable result type is given the right to read from or write to any accessible variable; however, any such function which is used in the definition of a function having an observable result type (such as 'eq_s') may only be given the right to read from variables, not to write to them. The functions are thereby replaced by procedures.

- For each unobservable sort with name 'S', a procedure

  $\text{has\_s} : \text{S} \overset{\sim}{\rightarrow} \textbf{read any Bool}$

  is introduced. This procedure is intended to provide a test that in a given store a member of a type with name 'S' gives access through the store to a representation of a member of the unobservable sort 'S'.

- For each such 'S', 'has_s' is postulated to be deterministically convergent for every store and for every parameter.

- For each such 'S', 'has_s' is postulated to return the result **true** when applied to a constructor constant.

- For each such 'S', 'has_s' is postulated to provide results that are not interfered with by the application of any procedure declared in the specification for every store and for every parameter which satisfy both the applicability procedure and 'has_s' (with appropriate modifications when the parameter type or the result type is not an unobservable sort); for instance, if 'f' is a procedure having type 'S $\overset{\sim}{\rightarrow}$ **write any** S' then it must satisfy

  $$\begin{array}{l} \forall\, s_1 : S\ ,\ s_2 : S \bullet \\ \quad \textit{applicable}[\![f]\!]\ (\ s_1\ ) \wedge \text{has\_s}\ (\ s_1\ ) \wedge \text{has\_s}\ (\ s_2\ ) \Rightarrow \\ \quad (\ \textbf{result}\ (\ f\ (\ s_1\ )\ ;\ \text{has\_s}\ (\ s_2\ )\ ) \equiv \textbf{true}\ ) \end{array}$$

  This style of assertion can be used whenever the parameter types of the procedures are flat.

- After every universal quantifier binding members of unobservable types, '$\Box$' is inserted and an extra premise is added, to test that each quantified name of sort 'S' satisfies 'has_s' (with appropriate modifications when the types of the quantified names are not unobservable sorts).

- Any equation '$e_1 = e_2$' is treated as follows. If 'e_1' and 'e_2' have unobservable sorts the equation is replaced by '$e_1 \equiv e_2$'. If 'e_1' and 'e_2' have observable types the equation is replaced by '**result** $e_1 \equiv$ **result** e_2' (though '$\equiv$' is used instead of '$=$' purely for clarity). If 'e_1' and 'e_2' have unobservable types that are not sorts the equation is regarded as a conjunction of equations between the components of 'e_1' and 'e_2'.

- Each procedure declared in the specification is postulated to be deterministically convergent and to provide results that satisfy 'has_s' for every store and for every parameter which satisfies both its applicability procedure and 'has_s' (with appropriate modifications when the parameter type or the result type is not an unobservable sort).
- Each inspector or observer procedure is postulated to provide results that are not interfered with by the application of any declared in the specification for every store and for all parameters which satisfy both the applicability procedures and 'has_s' (with appropriate modifications when the parameter type or the result type is not an unobservable sort). The need for, and nature of, the postulates depends on the forms taken by the conditional equations in the specification and is discussed in 6.6.
- The induction rules are removed.

6.2.2 Renaming the specification to distinguish names

- The specification, thus transformed, is subject to renaming by changing the names declared in it to avoid clashes with the names declared in the original specification. (Below, the name of an unobservable sort 'S' is taken to be changed by this means to 'S_'.)

6.2.3 Extending the specification to show refinement

- For each hitherto unobservable sort with name 'S', the declaration

$$\mathrm{S} = \{\ \mathrm{s} \mid \mathrm{s} : \mathbf{Unit} \xrightarrow{\sim} \mathbf{write\ any}\ \mathrm{S\_} \bullet \mathrm{is\_s}\ (\ \mathrm{s}\ )\ \}$$

  is provided. The type 'S' comprises procedures members of 'S_' with respect to 'eq_s', which may read from or write to any accessible variable and which may return members of 'S_'.

- For each such 'S', a function

$$\mathrm{is\_s} : (\ \mathbf{Unit} \xrightarrow{\sim} \mathbf{write\ any}\ \mathrm{S\_}\ ) \xrightarrow{\sim} \mathbf{Bool}$$

  is introduced. This function is used merely to indicate that 'S' may not comprise all the members of '**Unit** $\xrightarrow{\sim}$ **write any** S_'.

- The induction rules are added from the original specification to delimit 'S'.
- Versions of the constants and functions declared in the original specification are defined; for instance, if 'k_' is a constant having type 'S_' then the defined constant 'k' must satisfy

$$\mathrm{k} = \lambda\ (\ )\ \bullet\ \mathrm{k\_}$$

  if 'f_' is a procedure having type '**Bool** $\xrightarrow{\sim}$ **write any** S_' then the defined function 'f' must satisfy

$$\forall\, b : \mathbf{Bool} \bullet f(\, b\,) = \lambda\, (\,)\bullet f\_(\, b\,)$$

if 'f_' is a procedure having type 'S_ $\overset{\sim}{\rightarrow}$ **write** any S_' then the defined function 'f' must satisfy

$$\forall\, s : S \bullet f(\, s\,) = \lambda\, (\,)\bullet f\_(\, s(\,)\,)$$

and if 'f_' is a procedure having type 'S_ $\overset{\sim}{\rightarrow}$ **read any Bool**' then the defined function 'f' must satisfy

$$\forall\, s : S \bullet f(\, s\,) = \mathbf{let}\ b : \mathbf{Bool} \bullet \Box\, (\, b \equiv \mathbf{result}\ f\_(\, s(\,)\,)\,)\ \mathbf{in}\ b\ \mathbf{end}$$

This style of definition can be used whenever the types of the constants and the parameter and result types of the procedures are flat (though it requires extension if the types of the constants and the result types of the procedures may contain more than one unobservable sort occurrence).

- Given that the functions introduced in the specification satisfy the conditions outlined in 5.6, the specification thus extended provides a refinement of the original specification in which functions are interpreted as acting on procedures.

6.3 Example

6.3.1 Transforming the specification to convert functions

The applicative specification of 'List' given in 8.6 can be transformed by applying the technique described in 6.2. The outcome of doing so is presented in 8.9.

6.3.2 Renaming the specification to distinguish names

The type, constant and functions declared in 8.9 must have their names changed before they can be compared directly with those in 8.6. This is done in 8.10.

6.3.3 Extending the specification to show refinement

To establish that the specification in 8.9 is an appropriate transformation of that in 8.6, versions of the type, constant and functions declared in the specification in 8.6 are defined by extending the specification in 8.10. The outcome of doing so is presented in 8.11.

6.4 Assumptions

The applicative specification must take the form described in 4.3 and the inspectors and observers must satisfy the conditions outlined in 5.6. (In fact the inspectors and observers may satisfy any conditions which make the specification sufficiently complete and which allow the behaviour of expressions to be characterised using generalised observers.)

In any equation '$e_1 = e_2$' the expressions 'e_1' and 'e_2' must be 'suitable' in the sense explained in 4.4.

Functions must be able to take procedures as parameters.

6.5 Variants

Very often the applicative specification which is to be transformed does not make assertions about more than one object (where the term 'object' is used in the sense of object-oriented programming). This is so if each equation in the applicative specification takes a form which, according to the rules in 6.6, needs no assertions about freedom from interference. When this is so, each equation mentions at most one quantified name having an unobservable type.

For instance, the equations in 8.6 do not take such forms, because the assertion about 'eq_list' mentions two members of the sort 'List'. By contrast, the equations in 8.1 do take such forms (as a constant having an observable type needs no inspector or observer procedures to characterise its behaviour).

When the applicative specification needs no assertions about freedom from interference, the technique given in 6.2 can be modified to produce less intricate imperative specifications. Doing this involves:

- replacing in the applicative specification each constructor constant, having type 't', by a constructor function having type '**Unit** $\overset{\sim}{\rightarrow}$ t', so that when its type is altered by the transformation it is given the right to read from or write to any accessible variable;
- taking 'has_s' in the imperative specification to be 'λ ( ) • **true**' (or omitting its declaration entirely);
- taking 'S' in the imperative specification to be **Unit** (or omitting its declaration entirely);
- removing superfluous components of type **Unit** from the parameters and results of procedures in the imperative specification.

The specification in 8.2 results from applying this modified transformation to the specification in 8.1, so the modified transformation formalises the process described in 2.2.

The trick of replacing each constructor constant by a constructor function can be employed quite generally, whenever creating a member of an abstract data type ought to have an effect on the store. It is usually appropriate if the abstract data type is implemented using static storage allocation and is frequently appropriate if the abstract data type is implemented using dynamic storage allocation.

6.6 Freedom from interference between procedures

In order to make sure that the procedures in imperative specifications do not interfere with one another, it is necessary to introduce certain assertions into the specifications. The form of these assertions is influenced by the form of the specifications; in the interests of simplicity the versions presented here are fairly insensitive to the form of the specifications. The assertions may relate to 'absorption', 'commutativity' or 'idempotence'. The absorption assertions imply the idempotence assertions; moreover, for an equation in the specification which equates expressions having observable types, general absorption assertions are equivalent with general commutativity assertions.

When in the specification there is an equation '$e_1 = e_2$' between two expressions such that a quantified name having an unobservable type is mentioned in one and only one of the expressions ('e_1', say), then absorption assertions are added for applications of the procedures used in the other expression. Here the 'procedures used in the other expression' include the constructor procedures if the other expression mentions any quantified names having unobservable types. If 'e_1' and 'e_2' have unobservable types then the absorption assertions take the following form: if 'f_1' and 'f_2' are such procedures having types '$t_1 \stackrel{\sim}{\rightarrow}$ **write any** t_3' and '$t_2 \stackrel{\sim}{\rightarrow}$ **write any** t_4' respectively (and used in 'e_1' and 'e_2' respectively) then

$$\begin{array}{l} \forall\, x_1 : t_1 \,,\, x_2 : t_2 \bullet \\ \quad \textit{applicable}[\![f_1]\!]\,(\, x_1 \,) \wedge \textit{applicable}[\![f_2]\!]\,(\, x_2 \,) \Rightarrow \\ \quad (\, (\, f_1 \,(\, x_1 \,) \,;\, f_2 \,(\, x_2 \,)\,) \equiv f_2 \,(\, x_2 \,)\,) \end{array}$$

If 'e_1' and 'e_2' have observable types then the absorption assertions are typified as follows: if 'f_1' and 'f_2' are as above except that 'f_2' is an inspector or observer procedure needed for characterising the behaviour of 'e_2', then, when 't_1' and 't_2' are both 'S',

$$\begin{array}{l} \forall\, s_1 : S \,,\, s_2 : S \bullet \\ \quad \textit{applicable}[\![f_1]\!]\,(\, s_1 \,) \wedge \text{has\_s}\,(\, s_1 \,) \wedge \textit{applicable}[\![f_2]\!]\,(\, s_2 \,) \wedge \text{has\_s}\,(\, s_2 \,) \Rightarrow \\ \quad (\, \textbf{result}\,(\, f_1 \,(\, s_1 \,) \,;\, f_2 \,(\, s_2 \,)\,) \equiv \textbf{result}\ f_2 \,(\, s_2 \,)\,) \end{array}$$

(with appropriate modifications when a parameter or result type is not an unobservable sort). This version of the absorption assertions is essentially that adopted for the conditions imposed on 'has_s' in 6.2.

When in the specification there is an expression which is a product, then commutativity assumptions are added for the applications of the procedures used in different components of the product. Here the 'procedures used in different components of the product' include the constructor procedures if the components mention any quantified names having unobservable types. If the expression occurs in an equation between expressions having unobservable types then the commutativity assertions take the following form: if 'f_1' and 'f_2' are such procedures having types '$t_1 \tilde{\rightarrow}$ **write any** t_3' and '$t_2 \tilde{\rightarrow}$ **write any** t_4' respectively (and used in different components of the product) then

$$\begin{array}{l} \forall\, x_1 : t_1 \,,\, x_2 : t_2 \bullet \\ \quad \mathit{applicable}[\![f_1]\!]\,(\,x_1\,) \wedge \mathit{applicable}[\![f_2]\!]\,(\,x_2\,) \Rightarrow \\ \quad (\,(\,f_1\,(\,x_1\,)\,;\,f_2\,(\,x_2\,)\,) \equiv \textbf{let}\ x_3 = f_2\,(\,x_2\,)\ \textbf{in}\ f_1\,(\,x_1\,)\,;\,x_3\ \textbf{end}\,) \end{array}$$

If the expression occurs in an equation between expressions having observable types then the commutativity assertions reduce to the absorption assertions. (The commutativity assertions also have longer forms which are symmetrical in their uses of 'f_1' and 'f_2'.)

When in the specification there is an expression which has an unobservable type containing more than one unobservable sort occurrence, then idempotence assumptions are added for the applications of the procedures used in the expression. Here the 'procedures used in the expression' include the constructor procedures if the expression mentions any quantified names having unobservable types. If the expression occurs in an equation between expressions having unobservable types then the idempotence assertions take the following form: if 'f' is such a procedure having type '$t_1 \tilde{\rightarrow}$ **write any** t_2' then

$$\begin{array}{l} \forall\, x_1 : t_1 \bullet \\ \quad \mathit{applicable}[\![f]\!]\,(\,x_1\,) \Rightarrow \\ \quad (\,(\,f\,(\,x_1\,)\,;\,f\,(\,x_1\,)\,) \equiv f\,(\,x_1\,)\,) \end{array}$$

If the expression occurs in an equation between expressions having observable types then the idempotence assertions again reduce to the absorption assertions.

In an implementation it is difficult to satisfy the absorption and commutativity assertions needed by equations between expressions having unobservable types. Hence all such equations tend to be replaced by behavioural equivalences using 'eq_s'. In this case it is enough to have general absorption assertions; in the notation adopted above, these should let 'f_1' range over all constructor procedures and let 'f_2' range over all inspector and observer procedures needed in the characterisation of behaviour using 'eq_s'.

7 Replacing axioms by programs

7.1 Survey

The third stage in refining applicative specifications into imperative implementations entails replacing axioms by programs. The technique needed supplies the procedures declared in a specification with executable definitions. The specification must obey conditions which ensure that no member of the sorts can be constructed in two distinct ways by using constructors alone. These conditions therefore allow lists and queues, but not sets and maps, to be implemented using this technique. In fact they permit many of the flat "recursive" types that can be defined briefly in RAISE [6] and all of those that can be defined briefly in Standard ML [5], HOL [10] and Z [16].

The technique is described in 7.2 and illustrated in 7.3. Assumptions that it requires are mentioned in 7.4. Variants of it are discussed in 7.5.

7.2 Tasks

7.2.1 Transforming the specification to replace axioms

- For each hitherto unobservable sort with name 'S', the declaration

  S = **Int**

  is provided. The type 'S' comprises the possible indexes into an array. It induces a representation of the members of any flat, hitherto unobservable, type. (If the induction rules indicate that there is a bound on the permitted depth of constructor functions, then a finite range of integers can be used instead of **Int**; this range must contain one entry per constructor constant and one entry per permitted depth of nesting.)

- For each such 'S', an index variable

  index_s : S

  is introduced. This index variable is intended to indicate where unused array elements lie.

- For each such 'S', an array object

  array_s : ...

  is introduced. An element of this array is brought into use by the application of a constructor procedure; it must be capable of storing information which can identify the constructor procedure and represent the parameter in the application of the constructor procedure. Consequently the array object must provide, for every possible index, a finite collection of variables which suffices to store both a tag (drawn from an observable type in which every constructor procedure is allocated a distinct tag) and the representation of any parameter in a constructor procedure application. The tag can be omitted if there is only one constructor procedure.

- For each such 'S', each application of 'has_s' tests whether its integer parameter lies between the value most recently written to the index variable and the lowest representation of a constructor constant. (A more modular treatment would combine the array, the index and the definition of 'has_s' in the implementation of a name generator; it is omitted from this paper in order to concentrate on the systematic development of just one module.)
- For each such 'S', each constructor constant is represented by a different integer; the representations are arranged consecutively.
- For each such 'S', each application of a constructor procedure increments the value in 'index_s' (making sure that it is above the highest representation of a constructor constant) and stores in the corresponding element of 'array_s' the information noted above.
- For each such 'S', each application of a destructor procedure uses its parameter to determine an array element from which it then extracts the representation of the parameter in a constructor procedure application.
- For each such 'S', each application of a discrimination procedure checks whether its parameter is a constructor constant; if so, it checks which constructor constant is its parameter, but, if not so, it uses its parameter to determine an array element which identifies a constructor procedure.
- Given that the procedures introduced in the specification are derivable from functions which satisfy the conditions outlined in 5.6, the remaining procedures are defined recursively in terms of the constructor procedures, the destructor procedures, the discrimination procedures, and the observations on the constructor constants: some of the axioms are replaced by definitions, in which first the discrimination procedures and then the appropriate constructor, inspector and observer procedures are applied.
- Any axioms that are not replaced in the above manner should be proved to be consequences of the definitions of the procedures. If they are proved to be consequences, they can be deleted from the specification; if they are not proved to be consequences, they must be retained in the specification (which may well be inconsistent). Generally the interpretation of equalities as equivalences of observable behaviour, according to the scheme laid down in 5.2, ensures that these axioms are indeed consequences of the definitions.

7.3 Example

7.3.1 Transforming the specification to replace axioms

The imperative specification of 'List' given in 8.6 can be transformed by applying the technique described in 7.2. The outcome of doing so is presented in 8.12. No renaming or extending is needed in order to establish refinement: the procedures in 8.12 have already all the properties specified for those in 8.9.

7.4 Assumptions

The imperative specification must be derivable by applying the transformation described in 6.2 to an applicative specification satisfying the conditions in 6.4.

For each constructor function in the applicative specification there must be destructor and discrimination functions obeying the rules in 4.6 and 4.7. There is then a unique way of constructing each member of a sort using constructors alone.

For every observable type which can appear as part of the parameter type for a constructor function there must be a way of declaring a finite collection of variables which can hold any member of the type.

7.5 Variants

An extension to the technique considered in 7.2 allows certain inspector procedures to be implemented without the use of constructor procedures (which write to variables) and destructor procedures (which need to be applied recursively). It applies if there is a unique constructor constant 'k' and if for all constructor and inspector functions ('c' and 'i' respectively), in the notation of 5.6,

$$\begin{array}{l} \forall\, x_1 : t_1 \bullet \\ \quad \textit{applicable}[\![c]\!]\,(\,x_1\,) \wedge \textit{applicable}[\![i]\!]\,(\,c\,(\,x_1\,)\,) \Rightarrow \\ \quad i\,(\,c\,(\,x_1\,)\,) = k \vee i\,(\,c\,(\,x_1\,)\,) = c\,(\,i\,(\,x_1\,)\,) \end{array}$$

In this case, each array element must store, besides the information noted in 7.2, an "upwards-pointing" array index indicating which array element (if any) is constructed from the given one; also, the representation of a member of 'S' must provide, beside the array index for its "latest" constructor, array indexes for the results of applying the inspector procedures. This extension to the technique can be used to implement queues for which applications of the constructor procedures take linear time (unless there is only one object, which is defined according to 6.5 and which then does not need to be copied) but applications of the inspector and observer procedures take constant time.

8 Specifications

8.1 First applicative specification

```
FIRST_APPLICATIVE_LIST =
  class
    type
      List
    value
      empty : List ,
      add : ( Element × List ) ~→ List ,
      head : List ~→ Element ,
      tail : List ~→ List ,
      is_empty : List ~→ Bool
    axiom forall e : Element , l : List •
      is_empty ( empty ) = true ,
      head ( add ( e , l ) ) = e ,
      tail ( add ( e , l ) ) = l ,
      is_empty ( add ( e , l ) ) = false
    axiom forall p : List ~→ Bool •
      ( p ( empty ) ∧
        ( ∀ e : Element , l : List • p ( l ) ⇒ p ( add ( e , l ) ) ) ) ⇒
      ( ∀ l : List • p ( l ) )
  end
```

8.2 First imperative specification

```
FIRST_IMPERATIVE_LIST =
  class
    value
      empty : Unit ~→ write any Unit ,
      add : Element ~→ write any Unit ,
      head : Unit ~→ read any Element ,
      tail : Unit ~→ write any Unit ,
      is_empty : Unit ~→ read any Bool
    axiom forall e : Element •
      □ definite empty ( ) ,
      □ ( empty ( ) ; is_empty ( ) ≡ empty ( ) ; true ) ,
      □ definite add ( e ) ,
      □ ( add ( e ) ; head ( ) ≡ add ( e ) ; e ) ,
      □ ( add ( e ) ; tail ( ) ≡ ( ) ) ,
      □ ( add ( e ) ; is_empty ( ) ≡ add ( e ) ; false )
  end
```

8.3 Renamed first imperative specification

```
RENAMED_FIRST_IMPERATIVE_LIST =
  use
    empty_ for empty ,
    add_ for add ,
    head_ for head ,
    tail_ for tail ,
    is_empty_ for is_empty
  in FIRST_IMPERATIVE_LIST
```

8.4 Extended renamed first imperative specification

```
EXTENDED_RENAMED_FIRST_IMPERATIVE_LIST =
  extend RENAMED_FIRST_IMPERATIVE_LIST with
  class
    type
      List = { l | l : Unit ~→ write any Unit • is_list ( l ) }
    value
      empty : List ,
      add : ( Element × List ) ~→ List ,
      head : List ~→ Element ,
      tail : List ~→ List ,
      is_empty : List ~→ Bool
    value
      is_list : ( Unit ~→ write any Unit ) ~→ Bool
    axiom forall e : Element , l : List •
      empty = λ ( ) • empty_ ( ) ,
      add ( e , l ) = λ ( ) • l ( ) ; add_ ( e ) ,
      head ( l ) = let e : Element • □ ( e ≡ result ( l ( ) ; head_ ( ) ) ) in e end ,
      tail ( l ) = λ ( ) • l ( ) ; tail_ ( ) ,
      is_empty ( l ) = let b : Bool • □ ( b ≡ result ( l ( ) ; is_empty_ ( ) ) ) in b end
    axiom forall p : List ~→ Bool •
      ( p ( empty ) ∧
        ( ∀ e : Element , l : List • p ( l ) ⇒ p ( add ( e , l ) ) ) ) ⇒
      ( ∀ l : List • p ( l ) )
  end
```

8.5 Completed imperative specification

```
COMPLETED_IMPERATIVE_LIST =
  class
    value
      empty : Unit ⥲ write any Unit ,
      add : Element ⥲ write any Unit ,
      head : Unit ⥲ read any Element ,
      tail : Unit ⥲ write any Unit ,
      is_empty : Unit ⥲ read any Bool
    variable
      list : Element-list
    axiom
      empty = λ ( ) • list := ⟨ ⟩ ,
      add = λ e : Element • list := ⟨ e ⟩ ^ list ,
      head = λ ( ) • hd list ,
      tail = λ ( ) • tl list ,
      is_empty = λ ( ) • list = ⟨ ⟩
  end
```

8.6 Second applicative specification

```
SECOND_APPLICATIVE_LIST =
  class
    type
      List
    value
      empty : List ,
      add : ( Element × List ) $\xrightarrow{\sim}$ List ,
      head : List $\xrightarrow{\sim}$ Element ,
      tail : List $\xrightarrow{\sim}$ List ,
      is_empty : List $\xrightarrow{\sim}$ Bool
    value
      eq_list : ( List × List ) $\xrightarrow{\sim}$ Bool
    axiom forall e : Element , l : List •
      is_empty ( empty ) = true ,
      head ( add ( e , l ) ) = e ,
      eq_list ( tail ( add ( e , l ) ) , l ) = true ,
      is_empty ( add ( e , l ) ) = false
    axiom forall p : List $\xrightarrow{\sim}$ Bool •
      ( ∀ $l_1$ : List , $l_2$ : List • eq_list ( $l_1$ , $l_2$ ) ⇒ p ( $l_1$ ) = p ( $l_2$ ) ) ⇒
      ( p ( empty ) ∧
        ( ∀ e : Element , l : List • p ( l ) ⇒ p ( add ( e , l ) ) ) ) ⇒
      ( ∀ l : List • p ( l ) )
    axiom forall $l_1$ : List , $l_2$ : List •
      eq_list ( $l_1$ , $l_2$ ) =
      ( is_empty ( $l_1$ ) = is_empty ( $l_2$ ) ∧
        ( ~ is_empty ( $l_1$ ) ∧ ~ is_empty ( $l_2$ ) ⇒
          head ( $l_1$ ) = head ( $l_2$ ) ∧ eq_list ( tail ( $l_1$ ) , tail ( $l_2$ ) ) ) )
  end
```

8.7 Renamed second applicative specification

```
RENAMED_SECOND_APPLICATIVE_LIST =
  use
    List_ for List ,
    empty_ for empty ,
    add_ for add ,
    head_ for head ,
    tail_ for tail ,
    is_empty_ for is_empty
  in SECOND_APPLICATIVE_LIST
```

8.8 Extended renamed second applicative specification

```
EXTENDED_RENAMED_SECOND_APPLICATIVE_LIST =
  extend RENAMED_SECOND_APPLICATIVE_LIST with
  class
    type
      List =
        { l |
          l : List_-infset •
            l ≠ { } ∧
            ∀ l_1 : List_ , l_2 : List_ • l_1 ∈ l ⇒ eq_list ( l_1 , l_2 ) = ( l_2 ∈ l ) }
    value
      empty : List ,
      add : ( List × List ) →̃ List ,
      head : List →̃ Element ,
      tail : List →̃ List ,
      is_empty : List →̃ Bool
    value
      abs_list : List_ →̃ List ,
      rep_list : List →̃ List_
    axiom forall e : Element , l : List •
      empty = abs_list ( empty_ ) ,
      add ( e , l ) = abs_list ( add_ ( e , rep_list ( l ) ) ) ,
      head ( l ) = head_ ( rep_list ( l ) ) ,
      tail ( l ) = abs_list ( tail_ ( rep_list ( l ) ) ) ,
      is_empty ( l ) = is_empty_ ( rep_list ( l ) )
    axiom forall l : List , l_1 : List_ •
      abs_list ( l_1 ) = { l_2 | l_2 : List_ • eq_list ( l_1 , l_2 ) } ,
      rep_list ( l ) ∈ l
  end
```

8.9 Second imperative specification

```
SECOND_IMPERATIVE_LIST =
  class
    type
      List
    value
      empty : List ,
      add : ( Element × List ) →̃ write any List ,
      head : List →̃ read any Element ,
      tail : List →̃ read any List ,
      is_empty : List →̃ read any Bool
  value
    eq_list : ( List $\times$ List ) $\overset{\sim}{\to}$ read any Bool
  value
    has_list : List $\overset{\sim}{\to}$ read any Bool
  axiom forall e : Element , l : List •
    $\Box$ has_list ( empty ) ,
    $\Box$ definite is_empty ( empty ) $\wedge$
      ( result is_empty ( empty ) $\equiv$ true ) ,
    $\Box$ has_list ( l ) $\Rightarrow$
      definite add ( e , l ) $\wedge$
    $\Box$ has_list ( l ) $\Rightarrow$
      definite head ( add ( e , l ) ) $\wedge$
      ( result head ( add ( e , l ) ) $\equiv$ e ) ,
    $\Box$ has_list ( l ) $\Rightarrow$
      definite tail ( add ( e , l ) ) $\wedge$
      ( result has_list ( tail ( add ( e , l ) ) ) $\equiv$ true ) $\wedge$
      ( result eq_list ( tail ( add ( e , l ) ) , l ) $\equiv$ true ) ,
    $\Box$ has_list ( l ) $\Rightarrow$
      definite is_empty ( add ( e , l ) ) $\wedge$
      ( result is_empty ( add ( e , l ) ) $\equiv$ false )
  axiom forall l : List •
    $\Box$ has_list ( $l_1$ ) $\wedge$ has_list ( $l_2$ ) $\Rightarrow$
      ( eq_list ( $l_1$ , $l_2$ ) =
        ( is_empty ( $l_1$ ) = is_empty ( $l_2$ ) $\wedge$
          ( $\sim$ is_empty ( $l_1$ ) $\wedge$ $\sim$ is_empty ( $l_2$ ) $\Rightarrow$
            head ( $l_1$ ) = head ( $l_2$ ) $\wedge$ eq_list ( tail ( $l_1$ ) , tail ( $l_2$ ) ) ) ) )
  axiom forall l : List •
    $\Box$ definite has_list ( l )
  axiom forall e : Element , $l_1$ : List , $l_2$ : List •
    $\Box$ has_list ( $l_1$ ) $\wedge$ has_list ( $l_2$ ) $\Rightarrow$
      ( result ( add ( e , $l_1$ ) ; has_list ( $l_2$ ) ) $\equiv$ has_list ( $l_2$ ) )
  axiom forall e : Element , $l_1$ : List , $l_2$ : List •
    $\Box$ has_list ( $l_1$ ) $\wedge$ has_list ( $l_2$ ) $\Rightarrow$
      ( result ( add ( e , $l_1$ ) ; head ( $l_2$ ) ) $\equiv$ head ( $l_2$ ) ) ,
    $\Box$ has_list ( $l_1$ ) $\wedge$ has_list ( $l_2$ ) $\Rightarrow$
      ( result ( add ( e , $l_1$ ) ; tail ( $l_2$ ) ) $\equiv$ tail ( $l_2$ ) ) ,
    $\Box$ has_list ( $l_1$ ) $\wedge$ has_list ( $l_2$ ) $\Rightarrow$
      ( result ( add ( e , $l_1$ ) ; is_empty ( $l_2$ ) ) $\equiv$ is_empty ( $l_2$ ) )
end
```

8.10 Renamed second imperative specification

```
RENAMED_SECOND_IMPERATIVE_LIST =
  use
    List_ for List ,
    empty_ for empty ,
    add_ for add ,
    head_ for head ,
    tail_ for tail ,
    is_empty_ for is_empty ,
    eq_list_ for eq_list
  in SECOND_IMPERATIVE_LIST
```

8.11 Extended renamed second imperative specification

```
EXTENDED_RENAMED_SECOND_IMPERATIVE_LIST =
  extend RENAMED_SECOND_IMPERATIVE_LIST with
  class
    type
      List = { l | l : Unit ~→ write any List_ • is_list ( l ) }
    value
      empty : List ,
      add : ( Element × List ) ~→ List ,
      head : List ~→ Element ,
      tail : List ~→ List ,
      is_empty : List ~→ Bool
    value
      eq_list : List × List ~→ Bool
    value
      is_list : ( Unit ~→ write any List_ ) ~→ Bool
    axiom forall e : Element , l : List •
      empty = λ ( ) • empty_ ,
      add ( e , l ) = λ ( ) • add_ ( e , l ( ) ) ,
      head ( l ) = let e : Element • □ ( e ≡ result head_ ( l ( ) ) ) in e end ,
      tail ( l ) = λ ( ) • tail_ ( l ( ) ) ,
      is_empty ( l ) = let b : Bool • □ ( b ≡ result is_empty_ ( l ( ) ) ) in b end
    axiom forall p : List ~→ Bool •
      ( ∀ l₁ : List , l₂ : List • eq_list ( l₁ , l₂ ) ⇒ p ( l₁ ) = p ( l₂ ) ) ⇒
      ( p ( empty ) ∧
        ( ∀ e : Element , l : List • p ( l ) ⇒ p ( add ( e , l ) ) ) ) ⇒
      ( ∀ l : List • p ( l ) )
    axiom forall l₁ : List , l₂ : List •
      eq_list ( l₁ , l₂ ) = let b : Bool • □ ( b ≡ result eq_list_ ( l₁ ( ) , l₂ ( ) ) ) in b end
  end
```

8.12 Completed second imperative specification

```
COMPLETED_SECOND_IMPERATIVE_LIST =
  class
    type
      List = Int
    value
      empty : List ,
      add : ( Element × List ) ⥲ write any List ,
      head : List ⥲ read any Element ,
      tail : List ⥲ read any List ,
      is_empty : List ⥲ read any Bool
    value
      eq_list : ( List × List ) ⥲ read any Bool
    value
      has_list : List ⥲ read any Bool
    variable
      index_list : Int
    object
      array_list [ l : List ] : class variable h : Element , t : List end
    axiom
      empty = 0 ,
      add =
      λ ( e , l ) : Element × List •
        index_list := if 0 ≥ index_list then 1 else index_list + 1 end ;
        array_list [ index_list ] . h := e ;
        array_list [ index_list ] . t := l ;
        index_list ,
      head = λ l : List • array_list [ l ] . h ,
      tail = λ l : List • array_list [ l ] . t ,
      is_empty = λ l : List • array_list [ l ] . t = empty
    axiom
      eq_list =
      λ ( l₁ , l₂ ) : List × List •
        is_empty ( l₁ ) = is_empty ( l₂ ) ∧
        ( ∼ is_empty ( l₁ ) ∧ ∼ is_empty ( l₂ ) ⇒
          head ( l₁ ) = head ( l₂ ) ∧ eq_list ( tail ( l₁ ) , tail ( l₂ ) ) )
    axiom
      has_list = λ l : List • index_list ≥ l ∧ l ≥ 0
  end
```

8.13 Third applicative specification

```
THIRD_APPLICATIVE_LIST =
  class
    type
      List
    value
      empty : List ,
      add : ( Element × List ) $\xrightarrow{\sim}$ List ,
      head : List $\xrightarrow{\sim}$ Element ,
      tail : List $\xrightarrow{\sim}$ List ,
      is_empty : List $\xrightarrow{\sim}$ Bool
    value
      eq_list : ( List × List ) $\xrightarrow{\sim}$ Bool
    axiom forall e : Element , l : List •
      is_empty ( empty ) = true ,
      head ( add ( e , l ) ) = e ,
      eq_list ( tail ( add ( e , l ) ) , l ) = true ,
      is_empty ( add ( e , l ) ) = false
    axiom forall p : List $\xrightarrow{\sim}$ Bool •
      ( ∀ $l_1$ : List , $l_2$ : List • eq_list ( $l_1$ , $l_2$ ) ⇒ p ( $l_1$ ) = p ( $l_2$ ) ) ⇒
      ( p ( empty ) ∧
        ( ∀ e : Element , l : List • p ( l ) ⇒ p ( add ( e , l ) ) ) ) ⇒
      ( ∀ l : List • p ( l ) )
    axiom forall $l_1$ : List , $l_2$ : List , $l_3$ : List •
      eq_list ( $l_1$ , $l_1$ ) ,
      eq_list ( $l_1$ , $l_2$ ) ⇒ eq_list ( $l_2$ , $l_1$ ) ,
      eq_list ( $l_1$ , $l_2$ ) ∧ eq_list ( $l_2$ , $l_3$ ) ⇒ eq_list ( $l_1$ , $l_3$ )
    axiom forall $e_1$ : Element , $e_2$ : Element , $l_1$ : List , $l_2$ : List •
      eq_list ( empty , empty ) ,
      $e_1$ = $e_2$ ∧ eq_list ( $l_1$ , $l_2$ ) ⇒
      eq_list ( add ( $e_1$ , $l_1$ ) , add ( $e_2$ , $l_2$ ) ) ,
      eq_list ( $l_1$ , $l_2$ ) ∧ ~ is_empty ( $l_1$ ) ∧ ~ is_empty ( $l_2$ ) ⇒
      head ( $l_1$ ) = head ( $l_2$ ) ,
      eq_list ( $l_1$ , $l_2$ ) ∧ ~ is_empty ( $l_1$ ) ∧ ~ is_empty ( $l_2$ ) ⇒
      eq_list ( tail ( $l_1$ ) , tail ( $l_2$ ) ) ,
      eq_list ( $l_1$ , $l_2$ ) ⇒
      is_empty ( $l_1$ ) = is_empty ( $l_2$ )
  end
```

9 References

[1] Goguen, J.A., and Winkler, T., "Introducing OBJ3", Report SRI-CSL-88-9, SRI International (1988).

[2] Gordon, M.J.C., "HOL: A Proof Generating System for Higher-Order Logic", in *VLSI Specification, Verification and Synthesis*, edited by Birtwistle, G., and Subrahmanyam, P.A., (Kluwer, 1988) 73-128.

[3] Guttag, J.V., and Horning, J.J., "The Algebraic Specification of Abstract Data Types", *Acta Informatica*, 10 (1978), 27-52.

[4] Guttag, J.V., Horning, J.J., and Wing, J.M., "Larch in Five Easy Pieces", Report 5, DEC System Research Center (1985).

[5] Harper, R., Milner, A.J.R.G., and Tofte, M., "The Definition of Standard ML", Report ECS-LFCS-88-62, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh (1988).

[6] Havelund, K., "An RSL Tutorial", Report LACOS/CRI/DOC/1/V3, Computer Resources International A/S (1991).

[7] Hoare, C.A.R., "Proof of Correctness of Data Representations", *Acta Informatica*, 1 (1972), 271-281.

[8] Huet, G., and Hullot, J.-M., "Proofs by induction in equational theories with constructors", *Journal of Computer and System Sciences*, 25, 2 (October 1982), 239-266.

[9] Jones, C.B., *Software Development: A Rigorous Approach* (Prentice-Hall, 1980).

[10] Melham, T.F., "Automating Recursive Type Definitions in Higher Order Logic", in *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by Birtwistle, G., and Subrahmanyam, P.A., (Springer-Verlag, 1989) 341-386.

[11] Milne, R.E., "The proof theory for the RAISE specification language", Report RAISE/STC/REM/12/V3, STC Technology Limited (1990).

[12] Milne, R.E., "Specifying and refining concurrent systems", Report RAISE/STC/REM/13/V1, STC Technology Limited (1990).

[13] Milner, A.J.R.G., "An Algebraic Definition of Simulation Between Programs", *Proceedings of the Second Joint International Conference on Artificial Intelligence* (British Computer Society, 1971) 481-489.

[14] Sannella, D., and Tarlecki, A., "Toward formal development of ML programs: foundations and methodology", Report ECS-LFCS-89-71, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh (1989).

[15] Schoett, O., "Data Abstraction and the Correctness of Modular Programming", Report ECS-LFCS-87-19, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh (1987).

[16] Spivey, J.M., *Understanding Z* (Cambridge University Press, 1988).