Formal Development with ABEL *

Ole-Johan Dahl and Olaf Owe

Department of Informatics
University of Oslo

*In S. Prehn, W.J. Toetenel (Eds.): *VDM'91: Formal Software Development Methods, Vol. 2: Tutorials, LNCS 552*, Springer Verlag, 1991, pp. 320-362. (Revised 1992)

Contents

1 Introduction

ABEL is a language together with a formal logic for use in program development. The overall goal has been to support specification and program development through semi-mechanical aids for reasoning and verification. It has been of major concern that the language and the associated reasoning formalism are such that consistency requirements and other proof obligations are as simple and manageable as possible. In particular we have sought to:

- offer language constructs well suited for mechanical aids to reasoning, with some emphasis on mechanisms for constructive specification,
- encourage modularization and abstract specification of interfaces,
- facilitate reusability of modules through the use of parameters,
- offer powerful ways of putting modules together, including inheritance, restriction, inclusion, extension, as well as assumption specification and checking, and
- enable simple and manageable proof obligations, including those related to the composition and (internal) consistency of modules, in the form of first order formulas.

A major idea behind ABEL has been to allow the same language cover all stages of program development from abstract requirement specifications to efficient low level programming. Thus ABEL includes facilities for non-constructive requirements specification, constructive specification, applicative programming, and object oriented, imperative programming. (We let the term "specification" cover all of these.) We have tried to build the language around a few concepts which may be applied at all levels. Different stages may be related to each other through a concept of module simulation. If a low level module is proved to simulate an abstract one, then the latter may be used as an abstract specification of the low level module.

In the sequel we emphasize an applicative language level called TGI, which stands for *terminating generator induction*. TGI specifications give rise to convergent rewrite rules, which enable efficient manipulation of formulas and other expressions for purposes of simplification and proof. At the TGI level all proof obligations in connection with the composition of modules and module simulation are quantifier-free formulas in constructively defined functions, provided that user specified axioms are quantifier-free. ABEL includes logic for partial functions, also at the TGI level.

The ABEL language has been developed at the University of Oslo over a period of more than 15 years, mainly by the authors, and in close interaction with a regular student course on program specification and verification. The most important sources of ideas have been as follows: SIMULA 67 (classes and subclasses), the LARCH activity (generator induction), [7], and OBJ (order sorted algebras), [5].

2 The Formula Language

The syntactic core of ABEL is a strongly typed first order expression language, whose main elements are *variables*, *functions*, and *types*. As we shall see elements of all three categories may be introduced and named by the ABEL user. In the complete language we therefore distinguish between *defining* occurrences and *applied* occurrences of such named elements.

A type T represents a set of *values*, V_T, of that type. The type of any variable, say x, is specified by syntactically associating the defining occurrence with a type, say T, usually by writing $x:T$. The variable is thereby restricted to range over values of type T. Similarly the defining occurrence of a function f is associated with the *profile* of the function by writing

$$f : T_1 \times T_2 \times \ldots \times T_n \longrightarrow T$$

where $n \geq 0$, and $T_1, \ldots, T_n, T$ are types. $T_1 \times T_2 \times \ldots \times T_n$ is the *domain* of the function, and T is the *codomain*. The number n is sometimes called the *arity* of the function. A *constant* is a function with arity zero. The word *signature* is our term for a set of function profiles.

The strong typing of ABEL ensures that the semantic definition of a function respects the profile in the following sense: for given argument values in the declared domain the function either has a value of the codomain type, or it has no value for these arguments. It can only be applied to arguments in the declared domain. An application of the function to given argument values is said to be *well-defined* if the function has a value for these arguments, otherwise it is said to be *ill-defined*.

ABEL uses the standard notation for function applications, $f(e_1, e_2, \ldots, e_n)$, where the parentheses are omitted if $n=0$. In addition infix, prefix, and postfix operators consisting of special symbols and boldface script may be used, as well as other "mixfix" notations where the operator may consist of more than one symbol. In these cases the function "name" is formed by writing the symbol $\hat{}$ in every argument position. Examples:

$$\neg\hat{} : \mathit{Bool} \longrightarrow \mathit{Bool}$$
$$\hat{}+\hat{} : \mathit{Int} \times \mathit{Int} \longrightarrow \mathit{Int}$$
$$\mathbf{if}\ \hat{}\ \mathbf{th}\ \hat{}\ \mathbf{el}\ \hat{}\ \mathbf{fi} : \mathit{Bool} \times T \times T \longrightarrow T$$

where *Bool* is the type of truth values, $\mathbf{f}$ and $\mathbf{t}$, *Int* is the type of integers, and T is an arbitrary type. Now for instance $x+y$ and $\hat{}+\hat{}(x,y)$ are alternative notations for an application of the addition function to the arguments x and y. The notation $x<y=z$, is shorthand for $(x<y) \wedge (y=z)$, and similar usage of other infix relational operators is allowed.

The set of Boolean, or logical, operators is standard: $\neg\hat{}$, $\hat{}\wedge\hat{}$, $\hat{}\vee\hat{}$, $\hat{}\Rightarrow\hat{}$, $\hat{}\Leftrightarrow\hat{}$, listed in the order of precedence. In addition $\hat{}=\hat{}$ is an alternative equality operator for Booleans whose precedence is that of relational operators, i.e. higher than the logical ones. The ABEL syntax for quantified formulas is as follows: $\forall x : T \bullet P$, and $\exists x : T \bullet P$, where the bound variable ranges over values of the indicated type. Syntactically a quantifier $\forall x : T \bullet$ or $\exists x : T \bullet$, is a unary (prefix) operator which binds less strongly than the other logical operators, except $\hat{}\Leftrightarrow\hat{}$.

In general a *partial order*, $\prec$, is syntactically defined for the introduced types. If $T \prec U$ holds for types T and U, then T is said to be a (proper) *subtype* of U. $T \preceq U$ means $T \prec U \vee T = U$. A subtype relationship has inclusion of the associated value sets as a semantic consequence: $T \preceq U \Rightarrow V_T \subseteq V_U$. Notice that the subtype relation is a syntactic notion; the value set inclusion is a consequence of the type definition conventions of ABEL. The reverse implication does not in general hold (however, an ABEL implementation might make it possible to establish certain subtype relations through semantic proofs).

The concept of subtypes is introduced for several purposes, one is to make strong typing of expressions more *flexible*. For example, let $\mathit{Nat} \prec \mathit{Int}$, where *Nat* is the type of natural

numbers (non-negative integers). Then the operator ˆ+ˆ would accept operands of either type, whereas for instance a square root function, $sqrt : Nat \longrightarrow Nat$ would require natural numbers.

At the same time typing may become *stronger*. As we shall see, it may be possible to discover "syntactic theorems" in the form of additional function profiles, like $\hat{}+\hat{} : Nat \times Nat \longrightarrow Nat$. This would make it possible to identify the expression $u+v$ as being of type *Nat* (rather than *Int*) if $u, v : Nat$. Two profiles for the same function are said to be *synonymous*.

We are now in a position to define the concept of *well-formedness* of expressions and at the same time describe the typing algorithm of ABEL, somewhat simplified. The following information is assumed to be available:

1. a set of types, partially ordered by the subtype relation,
2. a set of typed variables, and
3. a signature containing function profiles, including a user provided one for each function introduced.

The signature must satisfy certain restrictions, like not containing redundant profiles. This means that for any two synonymous profiles $f : D \longrightarrow C$ and $f : D' \longrightarrow C'$, the implication $D' \prec D \Rightarrow C' \prec C$ should hold. For type products $D = T_1 \times \ldots \times T_n$ and $D' = T'_1 \times \ldots \times T'_m$ the expression $D \prec D'$ means: $m = n \wedge T_1 \preceq T'_1 \wedge \ldots \wedge T_n \preceq T'_n \wedge D \neq D'$.

- A well-formed expression of type T is either a variable of type T, or it is a function application, say $f(e_1, \ldots, e_n)$, where e_i is a well-formed expression of type T_i, $i = 1, 2, \ldots, n$, and there is a f-profile with domain part D such that $T_1 \times \ldots \times T_n \preceq D$, and T is the "smallest" codomain part of such profiles (assumed to be unique).

The signature will contain one profile $\mathbf{if}\ \hat{}\ \mathbf{th}\ \hat{}\ \mathbf{el}\ \hat{}\ \mathbf{fi} : Bool \times T_1 \times T_2 \longrightarrow T$ for every triple of types T_1, T_2, T, not necessarily distinct, such that $T_1 \preceq T$ and $T_2 \preceq T$ and T is minimal. If there is such a type T, the two other types are said to be *related*. This implies that an **if**-construct can only be well-formed if the two alternatives are of related types, and if well-formed its type is the smallest common supertype. (The type conventions of ABEL ensure that the latter is unique.)

Only well-formed expressions are part of the ABEL expression language. Thereby the language, or rather a type checking device, helps the user by preventing a large class of mistakes. Notice that well-formedness is a syntactic property of expressions, entirely independent of the semantics of the occurring functions. In the sequel expressions are assumed to be well-formed.

Let an expression e be of type T. If e only contains applications of *total* functions, i.e. functions which have values on the entire user defined domain, then e is necessarily *well-defined* for arbitrary (type correct) interpretation of its free variables, and its value must be of type T. This is true although the well-definedness of an expression in general does depend on the semantics of the functions involved.

In computer programming one frequently has to deal with partial functions, i.e. functions which do not have values on the entire domain. A suitable subtype mechanism can be of some help in this connection, since any partial function will be total on some subdomain. For instance integer division, which has no function value for the denominator 0, can be

defined as a total function with the profile $\hat{}/\hat{} : Int \times Nzo \longrightarrow Int$, where Nzo is the type of non-zero integers. Thus, x/y is well-formed and well-defined for $x : Int$ and $y : Nzo$.

The last example suggests that there is a need for so-called *coercion* functions for checking values of a given type for membership in a subtype. For instance x/y would give a well-defined result for $x, y : Int$, whenever $y \neq 0$. The coercion functions are expressed using a generalized mixfix format: $\hat{}\,\mathbf{as}\,T : U \longrightarrow T$, where the type is considered part of the function name, and $T \prec U$. The function is partial; the function value is that of the argument if the latter belongs to the subtype T, otherwise the function has no value. For e of type U the expression $e\,\mathbf{as}\,T$ is a well-formed expression of type T, not necessarily well-defined.

The notation $e\,\mathbf{qua}\,T$ can be used for changing the type from U to T without applying the coercion function. Use of this notation entails an obligation for the user to prove that the occurrence of the expression e in the given context does have a value of type T whenever well-defined. For example, the expressions

$$x/(y\,\mathbf{as}\,Nzo) \qquad \text{and} \qquad x/\mathbf{if}\ y{\neq}0\ \mathbf{th}\ y\,\mathbf{qua}\,Nzo\ \mathbf{el}\ \perp\ \mathbf{fi}$$

are semantically equivalent for $x, y : Int$ and $\hat{}/\hat{}$ as above. The notation $\perp$ stands for a predefined constant with no value, sometimes pronounced "error". Its type is the predefined "empty" type $\oslash$, which by definition is a proper subtype of all other types. It follows that the denominator of the second expression is of type Nzo, which makes the expression well-formed (but not necessarily well-defined).

It is considered practical to have coercions to subtypes inserted automatically by the typing algorithm, possibly as an option controlled by the user.

3 Semantic Definitions

Having identified a set of type names and the signature of a set of functions it remains to define associated semantics:

1. a set of values for each type, and
2. the semantics of the identified functions.

In the approach of abstract algebra the semantics consist of a set of equational axioms which may define both aspects of semantics indirectly through the notion of an initial (mixed) algebra. While this approach is very flexible and leads to a very abstract, i.e. implementation independent, notion of types, it may be quite demanding mathematically. For instance, difficult questions about logical consistency and ground completeness arise. Furthermore, when semantics are based on the concept of initial algebra, the meaning of subspecifications may depend in non-obvious ways on its specification environment. ABEL avoids this by following the LARCH approach of being more constructive and somewhat less abstract (to an even greater extent than LARCH itself).

3.1 Generators

In particular a so-called *generator basis* (also called a constructor set) is required for each type. The generator basis, G_T, for a type T consists of a chosen subset of T-*producers*, i.e. functions with the codomain T. The generator basis G_T by definition spans the entire

value set associated with the type T, in the sense that any T-value is expressible in terms of generators alone, possibly including generators of other types.

This idea can be seen as a generalization of the concept of enumeration types first introduced in PASCAL. Whereas the generator basis of an enumeration type consists of constants, a generator basis G_T in general may contain T-producers other than constants. A ground term (variable-free expression) of type T consisting exclusively of generator applications is called a *basic T-expression*. The set of basic T-expressions is called the *generator universe*, GU_T, of the type T. In order that GU_T be non-empty, G_T must contain at least one *relative T-constant*, i.e. a T-producer with no occurrence of T in its domain. If T occurs in the domain of any T-generator, then GU_T is infinite. Notice that a generator universe is at most countably infinite; therefore a concept like real numbers is not expressible as an ABEL type.

In ABEL one writes **genbas** $g_1, \ldots, g_n$ in order to specify the generator basis $G_T = \{g_1, \ldots, g_n\}$, where the type T is identified by context. For the moment we may assume that types are defined one by one, so that types other than T occurring in the domains of T-generators are previously defined, "underlying" types.

Examples:

1. The type *Bool* (predefined) has a generator basis consisting of the constants **f** and **t**: $G_{Bool} = GU_{Bool} = \{\mathbf{f}, \mathbf{t}\}$.

2. A natural generator basis for the type *Nat1* of non-zero natural numbers would consist of the constant 1 and a successor function:

   func $1 : \longrightarrow$ *Nat1*
   func $\mathbf{S}\hat{} : Nat1 \longrightarrow Nat1$
   genbas $1,\ \mathbf{S}\hat{}$
   giving $GU_{Nat1} = \{1,\ \mathbf{S}1,\ \mathbf{SS}1,\ \ldots\}$.

3. The type *Int* of integers may be spanned by a generator basis consisting of 0, a successor function, and negation:

   func $0 : \longrightarrow$ *Int*
   func $\mathbf{S}\hat{} : Int \longrightarrow Int$
   func $-\hat{} : Int \longrightarrow Int$
   genbas $0,\ \mathbf{S}\hat{},\ -\hat{}$
   giving $GU_{Nat} = \{0,\ \mathbf{S}0,\ -0,\ \mathbf{SS}0,\ \mathbf{S}-0,\ -\mathbf{S}0,\ --0,\ \mathbf{SSS}0,\ \ldots\}$

4. Whereas real numbers are not expressible in ABEL, rationals are. They may be defined on top of the types *Int* and *Nat1*:

   func $div : Int \times Nat1 \longrightarrow Ratn$
   genbas div
   giving $GU_{Ratn} = \{div(0,1),\ div(0,2),\ \ldots,\ div(1,1),\ div(1,2),\ \ldots,\ \ldots\}$
   using standard representations (overloaded) of values of the underlying types.

5. The type of finite sequences of T-values, denoted $Seq\{T\}$, may be spanned as follows, for $V_T = \{a, b, \ldots\}$:

   func $\varepsilon : \longrightarrow Seq\{T\}$ — empty sequence
   func $\hat{}\vdash\hat{} : Seq\{T\} \times T \longrightarrow Seq\{T\}$ — append right
   genbas $\varepsilon,\ \hat{}\vdash\hat{}$
   giving $GU_{Seq\{T\}} = \{\varepsilon,\ e \vdash a,\ \varepsilon \vdash b,\ \ldots,\ (\varepsilon \vdash a) \vdash a,\ (\varepsilon \vdash a) \vdash b,\ \ldots,\ \ldots\}$

6. The type of finite sets of T-values, $Set\{T\}$, may be spanned in a similar way:

$$\begin{array}{ll}
\textbf{func } \emptyset \ : Set\{T\} & \text{— empty set} \\
\textbf{func } add : Set\{T\} \times T \longrightarrow Set\{T\} & \text{— add an element} \\
\textbf{genbas } \emptyset,\ add & \\
\text{giving } GU_{Set\{T\}} = \{\emptyset,\ add(\emptyset, a),\ add(\emptyset, b),\ \ldots, & \\
\qquad add(add(\emptyset, a), a),\ add(add(\emptyset, a), b),\ \ldots,\ \ldots\} &
\end{array}$$

Three of these generator bases, nos. 1, 2 and 5, are such that the basic expressions are in a one-to-one relationship with the intended abstract values. Therefore these bases are said to have the *one-to-one property*. Thus, specifying a one-to-one generator basis for a type defines the associated values to be (represented by) the corresponding basic expressions (using unique denotations for values of underlying types, if any).

For the examples 3, 4 and 6 the abstract values must be identified with certain equivalence classes of basic expressions. For instance, the expressions 0, –0, – –0, . . . , all represent the value zero, and $add(\emptyset, a)$, $add(add(\emptyset, a), a)$, $add(add(add(\emptyset, a), a), a), \ldots$, all represent the singleton set $\{a\}$. The need to define a corresponding equivalence relation on the generator universe represents a considerable complication of the mathematical treatment of a type, and we shall see how the one-to-one property can sometimes be obtained through the use of subtypes.

Any generator universe is partially ordered by the subterm relation, and this order is well-founded. It therefore gives rise to an induction principle, called *generator induction*. As we shall see, the principle of generator induction is useful for purposes of function definition as well as theorem proving.

3.2 Function definition

The semantics of functions (other than generators) are given by *axioms* or by explicit definitions. We may exemplify the three styles of axiomatization in ABEL through a simple example, the subtraction function for natural numbers, *Nat*, including zero, $\hat{}-\hat{}$: $Nat \times Nat \longrightarrow Nat$.

3.2.1 Arbitrary first order axioms

Consider the axioms $\text{A1} : x-0=x$ and $\text{A2} : \neg\exists y : Nat \bullet x-1<y<x$, for $x : Nat$. Are they consistent with the standard interpretation of the other occurring functions? Yes, they are, for instance by interpreting $\hat{}-\hat{}$ as addition or subtraction. Thus, they do not define the intended function completely. What if we add $x-1<x$ as a third axiom? Unfortunately the latter is inconsistent, because no natural number is less than 0. However, a slightly weaker one preserves consistency; $\text{A3} : x \neq 0 \Rightarrow x-1<x$. We leave it to the interested reader to discuss whether A1–3 define subtraction on natural numbers completely. (The answer is no, even if V_{Nat} is completely specified.)

The example shows that arbitrary first order axioms are not always easy to reason about (or with). This does not mean that such axioms have no place in ABEL specifications; they are indeed useful for specifying minimal requirements of functions, but not for giving complete definitions.

3.2.2 Recursive definitions

Using a successor function $\mathbf{S}\hat{}$ for natural numbers we may provide the following recursive definition:

$$\mathbf{def}\ x - y == \mathbf{if}\ x = y\ \mathbf{th}\ 0\ \mathbf{el}\ \mathbf{S}(x - \mathbf{S}y)\ \mathbf{fi}$$

The double equality sign stands for so-called "strong" equality, which expresses that the two operands are equally well-defined, and equal whenever well-defined. (The standard, or "weak" equality is *strict*, i.e. has no value for ill-defined operands.) The left hand arguments are defining occurrences of distinct, so-called *formal variables*, whose number and types are determined by the function profile and whose scope is the right hand side. The latter must contain no other free variables. The definition is said to be *constructive* if it is quantifier-free and contains only generators and constructively defined functions. If the right hand side is recursive, it should be considered well-defined only for arguments such that the recursion terminates. This is the usual "fixed point semantics", which implies that the definition is useful for bottom-up evaluation of ground terms. It is fairly easy to see that the recursion will terminate in the example definition if and only if $y \leq x$. Since evaluation of $x - y$ would not terminate for ground terms such that $x < y$, our function has no value in that case, which is reasonable in view of the required codomain *Nat*.

An advantage of explicit function definitions, compared to the use of arbitrary axioms, is that logical consistency as well as ground completeness are ensured by syntactic checks: there must be exactly one definition of every non-generator function.

3.2.3 Terminating generator induction

Definition by generator induction means using induction on an argument with respect to the syntactic structure of basic expressions (which in turn stand for abstract values). Thereby we obtain a definition of the function over the entire generator universe, which means for all values of the inductive argument. The generator induction can be expressed using a generalization of the **case**-construct of PASCAL for discriminating on values of enumeration types. Assume that the generator basis of natural numbers is $\{0, \mathbf{S}\hat{}\}$. The corresponding **case**-construct has one alternative for either generator:

$$\begin{aligned}\mathbf{def}\ x - y == \ & \mathbf{case}\ y\ \mathbf{of}\ 0 \to x \mid \mathbf{S}y' \to \\ & \mathbf{case}\ x\ \mathbf{of}\ 0 \to \bot \mid \mathbf{S}x' \to x' - y'\ \mathbf{fo}\ \mathbf{fo}\end{aligned}$$

The expression heading each alternative of a **case**-construct is called a *discriminator*. Notice that a discriminator corresponding to a non-constant generator has variables as arguments. These are defining occurrences of variables whose scope is the corresponding alternative expression, and serve to name the actual arguments of the leading generator application of the discriminated value. (Any variable name of the left hand side or of an outer discriminator may be reused; if so that old variable is inaccessible in the alternative.) The expression whose value is tested is called the *discriminand*. If all discriminands in the right hand side are variables, there is an alternative set of **case**-free defining equations, "Guttag-style" axioms, one for each innermost alternative, which are directly usable as term rewriting rules. In our case they are:

$$\begin{aligned}& x - 0 == x \\ & 0 - \mathbf{S}y == \bot \\ & \mathbf{S}x - \mathbf{S}y == x - y\end{aligned}$$

There is no restriction on depth of nested **case**-constructs, and the discriminand of an inner one may well be a variable introduced in an outer discriminator. This implies that the nesting of generators in the left hand sides of the **case**-free inductive axioms may be arbitrarily deep. If a discriminand of a **case**-construct is an expression other than a variable, the corresponding **case**-free axioms are *conditional*. (**if**-constructs may be seen as **case**-expressions with discriminand of type *Bool*, however, if they are treated as functions with respect to term rewriting, conditional rewrite rules can in most cases be avoided.)

An important advantage of generator inductive definitions is that there exist powerful syntactic checks which provide sufficient conditions for the termination of recursion. For instance, the third of the **case**-free axioms is recursive, but it obviously terminates since the arguments of the recursive application are subterms of those of the left hand side. The definition is therefore said to be by *terminating generator induction*, TGI. Notice that partial functions may be defined within a TGI framework, by explicit use of the ill-defined constant $\bot$ (error). The evaluation of any application of $\hat{}-\hat{}$ to values of type *Nat*, i.e. to expressions of the form $\mathbf{S}\ldots\mathbf{S}0$, either terminates with a resulting *Nat* value or with the symbol $\bot$ which is an explicit indication that the application is ill-defined.

A definition is said to be TGI if the right hand side is quantifier-free, and all occurring functions, except generators and $\bot$, are TGI defined, and, if recursive, the recursion is "guarded" by generator induction in some textually defined sense which ensures termination. Termination checks of different complexity and strength are possible; the following one is fairly general: each recursive application must be "smaller" than the corresponding left hand side, according to the lexicographic order induced on the list of arguments by the monotonic extension of the subterm relation, for each defined function according to a fixed permutation of its arguments.

Example

The check is strong enough to permit the following inductive definition of the Ackermann function (which does not belong to the class "primitive recursive" functions):

$$\begin{array}{l}\textbf{func}\ Ack : Nat \times Nat \longrightarrow Nat\\ \textbf{def}\quad Ack(x,y) == \textbf{case}\ x\ \textbf{of}\ 0 \rightarrow \mathbf{S}y \mid \mathbf{S}x \rightarrow\\ \qquad\qquad\qquad\qquad \textbf{case}\ y\ \textbf{of}\ 0 \rightarrow Ack(x,\mathbf{S}0) \mid \mathbf{S}y \rightarrow Ack(x,Ack(\mathbf{S}x,y))\ \textbf{fo fo}\end{array}$$

which corresponds to the following set of **case**-free axioms:

$$\begin{array}{l}Ack(0,y) == \mathbf{S}y\\ Ack(\mathbf{S}x,0) == Ack(x,\mathbf{S}0)\\ Ack(\mathbf{S}x,\mathbf{S}y) == Ack(x,Ack(\mathbf{S}x,y))\end{array}$$

There are three recursive applications to consider: $Ack(x,\mathbf{S}0)$, $Ack(x,Ack(\mathbf{S}x,y))$, and $Ack(\mathbf{S}x,y)$, where the last one is an argument of the second. We check the arguments from left to right: In the two first cases the left argument x is a subterm of that of the corresponding left hand side, $\mathbf{S}x$. In the third case the first argument is identical to that of the left hand side, but the second argument, y, is a subterm of the left hand one, $\mathbf{S}y$. This shows that the definition is TGI.

TGI definitions have several important advantages compared to general recursive definitions:

1. They permit a mechanical derivation of definedness predicates, see section 4.

2. They represent a *convergent set of rewrite rules*. Thus, TGI definitions are not restricted to bottom-up evaluation of ground terms, but may be used for the purpose of simplifying arbitrary expressions.

3. They are well suited for semi-mechanical proofs by generator induction, as explained in the subsection 4.4. In fact, TGI term rewriting and derived techniques are powerful proof generators for quantifier-free theorems.

3.3 Equality

For a type to have a fixed semantics, independent of its specification environment, its associated value set must be fully specified. As we have seen, however, a generator basis only determines a generator universe whose elements are not necessarily in a one-to-one correspondence with the intended abstract values.

The concept of equality on abstract values has so far been taken for granted, as it would be for any type T whose value space V_T is identified as a set of specified elements. In our approach, however, since the starting point is a generator universe, the T-values must be specified indirectly, as equivalence classes induced by defining an equivalence relation on GU_T. (If the generator basis is one-to-one these equivalence classes should be singleton sets.) Then, turning the table upside down, this equivalence relation on GU_T can be taken to be the *equality relation on* T, which can thus be defined as any other non-generator function, possibly by TGI technique.

In particular the one-to-one property of a generator basis, say $\{g_1, g_2, \ldots, g_n\}$, can be specified by defining an equality function which amounts to syntactic equality of basic T-expressions, up to equality on arguments of underlying types:

$$\textbf{def } t{=}t' == \textbf{case } (t,t') \textbf{ of } \overset{n}{\underset{i=1}{|}} (g_i(\bar{x}_i), g_i(\bar{x}'_i)) \to \bar{x}_i{=}\bar{x}'_i \mid \textbf{others } \to \textbf{f fo}$$

where nested **case** levels have been combined and all "off-diagonal" alternatives could be collected in a final **others** clause. Notice that the definition is recursive (but TGI) for those generators which are not relative constants. We may note that the TGI definition of syntactic equality of basic expressions can be constructed mechanically for any given generator basis. In ABEL one can therefore use a shorter syntax for specifying the one-to-one property:

1–1 genbas $g_1, g_2, \ldots, g_n$

In cases where a many-to-one generator basis must be used, one possibility is to define the desired equality relation explicitly.

Example 1

Consider the the type $Set\{T\}$ of finite sets of T-values, with the generator basis specified in example 5 of section 3.1. The equality may be TGI defined using the set membership and set inclusion relations as stepping stones.

$$\begin{array}{l}
\textbf{func } \hat{}\in\hat{} : T \times Set\{T\} \longrightarrow Bool \\
\textbf{def } \quad t \in s == \textbf{case } s \textbf{ of } \emptyset \to \textbf{f} \mid add(s,t') \to t{=}t' \vee t \in s \textbf{ fo} \\
\textbf{func } \hat{}\subseteq\hat{} : Set\{T\} \times Set\{T\} \longrightarrow Bool \\
\textbf{def } \quad s \subseteq s' == \textbf{case } s \textbf{ of } \emptyset \to \textbf{t} \mid add(s,t) \to t \in s' \wedge s \subseteq s' \textbf{ fo} \\
\textbf{def } \quad s{=}s' == s \subseteq s' \wedge s' \subseteq s
\end{array}$$

Having defined an equality relation explicitly there is an obligation to prove that it is in fact an equivalence relation, for instance by proving the standard axioms of reflexivity, commutativity, and transitivity for $s, s', s'' : Set\{T\}$:

$$s = s, \quad s = s' \Rightarrow s' = s, \text{ and } \quad s = s' = s'' \Rightarrow s = s''$$

Equality must in addition be such that meaning is preserved by substitution of equals for equals in expressions. This means that the relation consisting of the equalities for all types must be a so-called *congruence relation* respecting axioms of the form

$$y = y' \Rightarrow (f(\bar{x}, y, \bar{z}) == f(\bar{x}, y', \bar{z}))$$

for every function including generators and every argument position. (The strong equality is necessary for partial functions. It may be simulated using definedness predicates.) Fortunately all of these axioms are respected if all generator bases are one-to-one. But otherwise the total proof burden associated with explicit equality definition is rather formidable.

Another possibility is to define a so-called *observation basis* consisting of functions with one or more arguments of the type in question. The members of an observation basis are usually "observer" functions, i.e. functions with codomains which are underlying types. The members of the observation basis are called *basic observers*. They by definition observe "all there is to see" in the abstract values of the type under definition. Thus, two basic expressions of this type are to be considered equal if and only if all possible observations on them by basic observers are (strongly) equal. An observation basis is specified by the ABEL statement **obsbas** $h_1, \ldots, h_m$, listing the chosen basic observers by name.

For example, an observation basis may be specified for $Set\{T\}$ consisting of the single function $\hat{} \in \hat{}$. This implicitly defines equality on sets as:

$$\textbf{def} \;\; s = s' == \forall t : T \bullet \;\; (t \in s) = (t \in s')$$

In general the right hand side is a conjunction of equalities, one for each argument of the type under definition in the list of basic observers. If partial functions occur, then definedness predicates are also needed in order to simulate strong equalities. Unfortunately equality definition through observation basis is only constructive if all member functions are unary; otherwise quantifiers will occur in the right hand side, as in the example.

On the other hand, an equality defined through an observation basis is necessarily an equivalence relation, and it satisfies congruence with respect to generators and basic observers. The associated proof burden is thus considerably less.

Example 2

We define a type $IMap\{X, Y\}$ of "initialized maps", which simulate total functions with domain X and codomain Y. The function values are equal to a default Y-value, identified initially, except at a finite number of X-values where the map has been updated. Two *IMap* objects should be considered equal if and only if they have the same "function value" for all "arguments". Hence the specified observation basis.

```
func init : Y ⟶ IMap{X,Y}                                  — initial map
func ^[^↦^] : IMap{X,Y} × X × Y ⟶ IMap{X,Y}                — update map
genbas init, ^[^↦^]
func ^[^] : IMap{X,Y} × X ⟶ Y                              — apply map
def  m[x] == case m of init(y) → y | m1[x1↦y1] →
                 if x=x1 th y1 el m1[x] fi fo
obsbas ^[^]
```

If more than one update occurs for the same argument value, the last one takes precedence. Thus, for instance: $init(y_0)[x_1 \mapsto y_1][x_1 \mapsto y_2][x_1] = y_2$.

There is no proof obligation associated with these specifications since the only occurring function definition is for a basic observer.

It follows from the above that TGI function definitions preserve logical consistency as long as generator bases are one-to-one. There is, however, a danger of losing consistency when defining functions by generator induction over fully defined types with many-to-one bases. The reason for this may be explained by noting that generator induction in that case reveals the entire structure of the generator universe, including details which ought to be hidden inside equivalence classes. Thus, any use of such generator induction entails an obligation to prove one or more congruence axioms.

Example 3

We define a function counting the "multiplicity" of elements in finite sets by generator induction over $Set\{T\}$.

func $mpc : Set\{T\} \times T \longrightarrow Nat$
def $mpc(s,t) ==$ **case** s **of** $\emptyset \rightarrow 0 \mid add(s_1,t_1) \rightarrow$
if $t = t_1$ **th** $\mathbf{S}mpc(s_1,t)$ **el** $mpc(s_1,t)$ **fi fo**

Now consistency is lost, and this becomes clear when trying to carry out the required proof of the congruence axiom $s = s' \Rightarrow mpc(s,t) = mpc(s',t)$. Counterexample: Take $add(\emptyset,a)$ for s and $add(add(\emptyset,a),a)$ for s'. They are equal sets according to the defined equality (both represent the same singleton set), but the multiplicity of a is 1 in s and 2 in s'. (The loss of consistency shows that the concept of element multiplicity has no place in connection with sets; it belongs to the type of *multisets*, also called *bags*.)

The partial lack of syntactic consistency control of TGI function definitions is an obstacle to the use of types with many-to-one generator bases. There are, however, ways of achieving one-to-one-ness through the use of subtypes. See sections 5.2 and 5.4.

4 Logical Foundation

A function is said to be *strict* in an argument if ill-definedness in that argument propagates. A function is said to be strict if it is strict in all arguments. A function is said to be *total* if it is well-defined for all well-defined arguments. A function is said to be *monotonic* if replacing an argument in an application an ill-defined one makes the ill-definedness propagate or leaves the function value unchanged. An expression is said to *approximate* another if they are equivalent (in all respects) whenever the former is well-defined.

We introduce two non-monotonic logical operators: the definedness operator Δ, and non-strict or "strong" equality $==$. Thus, Δe expresses that the expression e is well-defined, and $e_1 == e_2$ is true iff either the expressions e_1 and e_2 are well-defined and equal, or they are both ill-defined. Thus, for instance $e == \bot$ expresses that the left hand side is ill-defined and is equivalent to $\neg\Delta e$. The definedness operator is defined inductively below. Definedness conditions and strong equalities are always well-defined: $\Delta(\Delta e) == \Delta(e_1 == e_2) == \mathbf{t}$. The non-monotonic operators are not part of the ABEL expression language.

We interpret the logical operators in analogy with the three valued logic originating from Kleene, [9], (although an ill-defined Boolean expression is considered to have no

value, rather than a third one). For instance, a conjunction is interpreted as false if either argument is false, regardless of the well-definedness of the other argument: $\mathbf{f} \wedge \bot == \bot \wedge \mathbf{f} == \mathbf{f}$. When one argument is true, ill-definedness in the other one propagates: $\mathbf{t} \wedge \bot == \bot \wedge \mathbf{t} == \bot$. The other logical connectives follow by the standard equivalences, with negation strict ($\neg\bot == \bot$):

$$\begin{array}{ll} a \vee b & == \neg(\neg a \wedge \neg b) \\ (a \Rightarrow b) & == \neg a \vee b \\ (a \Leftrightarrow b) & == (a \Rightarrow b) \wedge (b \Rightarrow a) \end{array}$$

A universal quantification may be seen as a generalized conjunction, in the standard way; and $\exists$ is equivalent to $\neg\forall\neg$. It follows that the logical connectives as well as the quantifiers are monotonic, and that they satisfy the classical distribution laws and deMorgan laws, and negations may be moved innermost in the classical way:

$$\begin{array}{ll} (a \vee b) \wedge c & == a \wedge c \vee b \wedge c \\ (a \wedge b) \vee c & == (a \vee c) \wedge (b \vee c) \\ \neg(a \Rightarrow b) & == a \wedge \neg b \\ \neg(a \vee b) & == \neg a \wedge \neg b \\ \neg(a \wedge b) & == \neg a \vee \neg b \\ \neg\forall x : T \bullet a & == \exists x : T \bullet \neg a \\ \neg\exists x : T \bullet a & == \forall x : T \bullet \neg a \end{array}$$

letting x denote a variable, and a, b and c formulas. Furthermore, the following equivalences may be added for convergent rewriting when extended with capabilities for handling $\wedge$ and $\vee$ as associative, commutative operators:

$$\begin{array}{ll} \neg\neg a & == a \\ a \wedge a & == a \\ a \vee a & == a \\ \\ (a \vee b) \wedge a & == a \\ (a \wedge b) \vee a & == a \\ \\ a \wedge \mathbf{t} & == a \\ a \wedge \mathbf{f} & == \mathbf{f} \\ a \vee \mathbf{t} & == \mathbf{t} \\ a \vee \mathbf{f} & == a \end{array}$$

The classical law of the excluded middle and its variants only hold for well-defined formula a. This reflects the fact that ill-definedness is a third possibility in addition to the two truth values.

$$\begin{array}{lll} \Delta a \Rightarrow a \vee \neg a & \text{or} & \neg(a \vee \neg a == \mathbf{f}) \\ \Delta a \Rightarrow \neg(a \wedge \neg a) & \text{or} & \neg(a \wedge \neg a == \mathbf{t}) \\ \Delta a \Rightarrow (a \Rightarrow a) & \text{or} & \neg((a \Rightarrow a) == \mathbf{f}) \end{array}$$

The **if**- and **case**-constructs, both strict in the leftmost argument, satisfy the following laws:

$$\begin{array}{ll} \textbf{case } g(t) \textbf{ of } .. \mid g(y) \rightarrow e \mid .. \textbf{ fo} & == e^y_t \\ \textbf{case } u \textbf{ of } .. \mid g(y) \rightarrow e^x_u \mid .. \textbf{ fo} & == \textbf{case } u \textbf{ of } .. \mid g(y) \rightarrow e^x_{g(y)} \mid .. \textbf{ fo} \\ \textbf{if } a \textbf{ th } e \textbf{ el } e' \textbf{ fi} & == \textbf{case } a \textbf{ of } \mathbf{t} \rightarrow e \mid \mathbf{f} \rightarrow e' \textbf{ fo} \end{array}$$

where y denotes a list of variables, and t a list of expressions, and where a^x_e denotes a with all free occurrences of x replaced by the expression e (renaming bound variables in a when needed), and a^y_t denotes simultaneous substitution.

It follows that $\textbf{if } a \textbf{ th } e^x_a \textbf{ el } {e'}^x_a \textbf{ fi} == \textbf{if } a \textbf{ th } e^x_{\mathbf{t}} \textbf{ el } {e'}^x_{\mathbf{f}} \textbf{ fi}$

Validity

A formula a is said to be *valid* iff it is well-defined and true, i.e. $a == \mathbf{t}$, for all possible well-defined interpretations of the free variables ("strong" interpretation). It follows that $a \wedge c$ is valid if and only if both a and c are valid, and that $a \vee c$ is valid if either a or c is valid.

In order to formalize the use of assumptions, we introduce sequents of the form $A \rightsquigarrow c$ where c, the conclusion, is a formula, and A, the assumption part, is a list of formulas. The sequent $A \rightsquigarrow c$ is said to be valid iff $A \Rightarrow c$ is valid, taking commas in the assumption part as $\wedge$'s. Thus, the sequent expresses that the conclusion must be well-defined and true unless (at least one formula in) the assumption part is well-defined and false, i.e. $\neg(A == \mathbf{f}) \Rightarrow (c == \mathbf{t})$. This is called WS logic since the assumptions have "weak" interpretation and the conclusion has "strong" interpretation.

Provability

In WS logic, we may derive $A \rightsquigarrow \Delta c$ from $A \rightsquigarrow c$, where Δc expresses that c is well-defined, due to the strong interpretation of the conclusion. Furthermore, the classical introduction and elimination rules of natural deduction [13] are sound; in particular, we have $A, a \rightsquigarrow c$ if and only if $A \rightsquigarrow a \Rightarrow c$, we have $A \rightsquigarrow \neg c$ if and only if $A, c \rightsquigarrow \mathbf{f}$, and we have $A, a \rightsquigarrow c$ if and only if $A, \neg c \rightsquigarrow \neg a$, which means that special rules introducing and eliminating symbols in the assumption part are not needed. The classical structural rules of sequent calculus are also sound, when the instantiation rule is restricted to well-defined substitutions, i.e. a sequent may be instantiated by replacing all occurrences of the same variable in both the assumption part and the conclusion by the same well-defined term, as formalized by the rule:

$$\frac{A \rightsquigarrow c \qquad A^x_e \rightsquigarrow \Delta e}{A^x_e \rightsquigarrow c^x_e}$$ instantiation rule

(If A and c are monotonic, it suffices that the conclusion is well-defined.)

The logical axiom $c \rightsquigarrow c$, which is trivial in classical sequent calculus, is not sound in WS logic. Instead we have $c \rightsquigarrow c$ if and only if c is well-defined. This means that a trivial sequent requires a proof of well-definedness; thus in WS logic nothing can be proved from meaningless assumptions, not even meaningless conclusions. This seems to be a healthy principle in computer science applications, and it fits well with proof by generator induction, see below.

By the below formalization of the well-definedness operator Δ, one may prove well-definedness requirements in a straightforward way, since $\Delta\Delta a$ is true.

Equality

Strong equality is a congruence relation satisfying the axiom $e == e$ and the rule

$$\frac{A \rightsquigarrow e == e' \qquad A \rightsquigarrow a^x_e}{A \rightsquigarrow a^x_{e'}}$$ substitution rule

Notice that with $==$ as a logical symbol, all the strong equations stated above may be taken as logical axioms.

The strict restriction of strong equality is called weak equality. The relationship between strong and weak equality may be formalized as follows by means of the well-definedness operator: $(e == e')$ is equivalent to $(\Delta e = \Delta e') \wedge (\Delta e \Rightarrow e = e')$. And notice that $a = b$ is equivalent to $(a \Rightarrow b) \wedge (b \Rightarrow a)$.

4.1 Definedness

The well-definedness of the logical operators is axiomatized as follows:

$\Delta(\bot) == \mathbf{f}$
$\Delta(\mathbf{t}) == \mathbf{t}$
$\Delta(\mathbf{f}) == \mathbf{t}$
$\Delta(x) == \mathbf{t}$ (except when x is a formal variable of a definition)
$\Delta(\neg a) == \Delta a$
$\Delta(a \wedge b) == (\Delta a \wedge (\neg a \vee \Delta b)) \vee (\Delta b \wedge (\neg b \vee \Delta a))$
$\Delta(a \vee b) == (\Delta a \wedge (a \vee \Delta b)) \vee (\Delta b \wedge (b \vee \Delta a))$
$\Delta(a \Rightarrow b) == (\Delta a \wedge (\neg a \vee \Delta b)) \vee (\Delta b \wedge (b \vee \Delta a))$
$\Delta(\forall x : T \bullet a) == (\forall x : T \bullet \Delta a) \vee (\exists x : T \bullet \Delta a \wedge \neg a)$
$\Delta(\exists x : T \bullet a) == (\forall x : T \bullet \Delta a) \vee (\exists x : T \bullet \Delta a \wedge a)$
Δ**if** a **th** e **el** e' **fi** $== \Delta a \ \wedge$ **if** a **th** Δe **el** $\Delta e'$ **fi**
Δ**case** e **of** ... $\mid g_i(x_i) \rightarrow e_i \mid$ .. **fo** $== \Delta e \ \wedge$ **case** e **of** ... $\mid g_i(x_i) \rightarrow \Delta e_i \mid$ .. **fo**
$\Delta(e == e') == \mathbf{t}$
$\Delta\Delta e == \mathbf{t}$

Quantifiers range over defined values only, thus bound variables are well-defined; and so are variables introduced in a case-construct. Formal variables of a function definition, however, may not be considered always well-defined.

Notice that the equations above may be used to calculate the well-definedness of a formula such that the resulting formula is without occurrences of Δ, except for applications to formal variables and non-logical functions — the well-definedness of the latter are given below. It follows from the above definitions that $(\Delta a) \wedge a$ and $(\Delta a) \Rightarrow a$ are well-defined for arbitrary formula a.

The left-strict versions of $\wedge, \vee, \Rightarrow$, denoted **and**, **or**, **implies**, respectively, are practically useful, giving more efficient execution and simpler definedness analysis. For instance,

$\Delta(a$ **and** $b) == \Delta a$ **and** $(\neg a$ **or** $\Delta b)$

In fact all occurrences of $\wedge$ and $\vee$ in the above right hand sides could be replaced by **and** and **or**, respectively.

4.2 Non-logical functions

For total and strict functions, such as generators and weak equality, we define

$$\Delta g(e) == \Delta e$$

letting Δ of a list be the conjunction of Δ of each list member, and letting Δ of an empty list be $\mathbf{t}$. Thus, generator constants such as $\mathbf{t}$, $\mathbf{f}$, 0 are well-defined. A constructive function definition, say

$$\textbf{def} \quad f(y) == e \qquad \text{(non-generator)}$$

where y is a list of formal variables, is logically understood as the axiom schema

$$f(t) == e^y_t \qquad (f\text{-axiom})$$

for any list of terms t (of the appropriate types), well-defined or not. It follows that

$$\Delta f(t) == \Delta e_t^y \qquad (\Delta f\text{-lemma})$$

which defines the well-definedness of f if the equation has only one fix-point. For non-TGI, recursively defined functions it is possible to provide proof rules corresponding to least fix-point semantics [12]. For a TGI definition, however, there is only one fix-point, and the well-definedness of f is implicitly defined by the Δf-lemma.

However, since the non-monotonic Δ-operator is outside the ABEL language, it may not be used inside definitions. For each defined function f, we therefore introduce a monotonic definedness predicate $\mathbf{d}f$ with the same domain as f, and such that $\mathbf{d}f(t)$ approximates $\Delta f(t)$. Since f in general may be non-strict the definition of $\mathbf{d}f$ is not straight forward: From a TGI definition of f as above, we first define a temporary definedness predicate, denoted $\mathbf{d}'f$, with domain as f but extended with one boolean argument for each argument of f:

$$\textbf{def} \quad \mathbf{d}'f(..,y_i,d_i,..) == \Delta' e \qquad (\mathbf{d}'f\text{-definition})$$

where Δ' is Δ as defined above, but replacing $\Delta h(..,t_i,..)$ by $\mathbf{d}'h(..,t_i,\Delta t_i,..)$ for each non-logical function h, and replacing occurrences of Δy_i by d_i. It follows that the right hand side is without occurrences of Δ, and that it is monotonic and TGI if the definition of f was TGI. (For non-TGI f, $\mathbf{d}'f$ is the maximal fix-point of the above equation when taking $\mathbf{f}$ as the greatest boolean value.) By induction on the nesting of functions, it follows that $\Delta f(..,t_i,..) == \mathbf{d}'f(..,t_i,\Delta t_i,..)$.

We then define the definedness predicate of f as follows:

$$\textbf{def} \quad \mathbf{d}f(..,y_i,..) == \mathbf{d}'f(..,y_i,\delta y_i,..) \qquad (\mathbf{d}f\text{-definition})$$

where δy is strongly equal to $y = y$, i.e. $\mathbf{t}$ if y is well-defined and ill-defined otherwise (thus $\Delta\delta t == \Delta t$). For TGI defined f it follows that the definition of $\mathbf{d}f$ is TGI as well. Furthermore, $\mathbf{d}f$ is total and $\mathbf{d}f(t) == \mathbf{t}$ is equivalent to $\Delta f(t) == \mathbf{t}$; and therefore $\mathbf{d}f(t)$ approximates $\Delta f(t)$.

For each non-logical function f, we let $\mathbf{d}f$ be part of the ABEL language, but not $\mathbf{d}'f$. For any generator g, $\mathbf{d}g(t)$ is δt. The well-definedness of a non-constructive function f may be characterized through non-logical axioms about $\mathbf{d}f$, or indirectly through axioms about f (because of the underlying strong interpretation). And its definedness predicate may be used to ensure well-definedness of f-applications in axioms — and also in constructive definitions of other functions if f is introduced in an assumed property.

Notice that the formula $f(x) == h(x')$ where x and x' are lists for free variables, is equivalent to $\mathbf{d}f(x) = \mathbf{d}h(x') \wedge (\mathbf{d}f(x) \Rightarrow f(x) = h(x'))$ (since free variables range over defined values). Note that from $\rightsquigarrow \mathbf{d}f(t)$ we may derive $\rightsquigarrow \Delta f(t)$ and vice versa, and that from $\rightsquigarrow \delta t$ we may derive $\rightsquigarrow \Delta t$ and vice versa.

Example

From the above TGI definition of the minus-function on natural numbers, we derive the following definition of its definedness predicate, $\mathbf{d}\hat{}-\hat{}$, letting $\mathbf{d}$ extend the mixfix notation:

$$\begin{aligned}\textbf{def} \quad \mathbf{d}\,x - y == \ & \textbf{case } y \textbf{ of } 0 \to \delta x \\ & \mid \mathbf{S}y' \to \textbf{case } x \textbf{ of } 0 \to \mathbf{f} \\ & \mid \mathbf{S}x' \to \mathbf{d}\,x' - y' \textbf{ fo fo}\end{aligned}$$

The right hand side simplifies to δx **and** δy **and** $y \leq x$ by inductive reasoning.

Axioms and lemmas

A user defined axiom **axm** a is understood as a non-logical axiom $\rightsquigarrow a$ taking commas in a as $\wedge$'s. Notice that this gives a strong interpretation of axioms. Free variables are implicitly universally quantified, and outermost universal quantifiers may be omitted.

A user defined lemma **lma** a states that $\rightsquigarrow a$ can be proved in WS logic extended with the introduced non-logical axioms and generator induction rules. There is an obligation to prove all stated lemmas.

Example

The following lemmas may be proved about the minus-function defined above:

lma $x, y, z : Nat \bullet$
d $x - y = y \leq x$,
$(x + y) - y = x$,
$y \leq x \Rightarrow x - (x - y) = y$

The first lemma follows from the results above and the fact that free variables are well-defined (thus δy is true). The condition of the last axiom is needed to ensure well-definedness.

4.3 Rewriting

Each step in a term rewriting process consists in first instantiating a rewrite rule so that the left hand side matches a subterm of the expression being processed, and then replacing that subterm by the instantiated right hand side. Notice that strong equations, proved or given as axioms, may not be used unconditionally as rewrite rules because of the definedness premise of the instantiation rule, restricting the instantiation of free variables. However, such a strong equation may be used unconditionally if both sides are strongly equal when the free variables are taken as formal variables (which may be instantiated unconditionally).

In a Guttag axiom, say $f(x, g(y), ..) == rhs$, the variable x is a formal variable of the corresponding **def**-item, and may therefore be instantiated to arbitrary (type correct) expressions, well-defined or not. The variable y on the other hand corresponds to one introduced in a discriminator, which is by definition well-defined. Unfortunately, this distinction is not recognized in ordinary rewriting, and for that reason there is a subtle difference between the semantics based on unconditional rewriting with Guttag axioms and that defined for function definitions with **case**-constructs in the right hand side.

However, it turns out that the left hand side of a Guttag axiom approximates the right hand side, when all variables are taken as formal ones. And the Guttag style axioms form a convergent (unconditional) rewrite system. Without loss of convergence the system may be enriched with the strong equations given above for the logical operators (but not generator strictness rules). It follows that a monotonic expression approximates (in the sense of ABEL semantics) its rewrite result with this system. In particular, a well-defined, monotonic expression strongly equals the result from rewriting with Guttag axioms [3].

It is possible to generate a set of convergent rewrite rules consistent with the ABEL semantics of the **case**-definitions, and generator strictness, by mechanically modifying right hand sides in certain Guttag axioms (by means of a definedness operator) [10]. No such modification would be necessary, however, in the Guttag rules for minus or plus on *Nat*.

4.4 Induction Proof

The usual rule for generator induction is sound in WS logic:

$$\frac{.., a^x_{y_i}, .. \ \rightsquigarrow \ a^x_{g(y)}}{\rightsquigarrow \ \forall x : T \bullet a} \qquad \text{generator induction on } T$$

where there is one premise for each T-generator g, each premise with one assumption for each argument y_i of type T. Each argument list y must consist of fresh variables correctly typed.

Example

For natural numbers (with 0 and **S** as generators) we get the following induction rule:

$$\frac{\begin{array}{c} \rightsquigarrow \ a^x_0 \\ a \ \rightsquigarrow \ a^x_{\mathbf{S}x} \end{array}}{\rightsquigarrow \ \forall x : Nat \bullet a} \qquad \text{generator induction on } Nat$$

As an example, we prove the second lemma above $(x + y) - y = x$ (*) by induction on y. The first premise is rewritten to true with the rules $(x + 0) == x$ and $(x - 0) == x$. The second premise becomes:

$$(x + y) - y = x \ \rightsquigarrow \ (x + \mathbf{S}y) - \mathbf{S}y = x \qquad (**)$$

With the rules $(x + \mathbf{S}y) == \mathbf{S}(x + y)$ and $\mathbf{S}x - \mathbf{S}y == x - y$, (**) rewrites to $(*) \rightsquigarrow (*)$ which is trivial since $(*)$ is well-defined. ∎

5 Modules

The module concept is essentially a mechanism for the encapsulation of specifications. A module is said to be *constructive* if the semantics of the specified functions are given by explicit definitions. In ABEL there are four different kinds of modules:

- A *type module* serves to define a type (possibly with subtypes) including associated functions.
- A *function module* defines a collection of functions.
- A *property module* specifies a set of minimal requirements on type parameters.
- *Class modules* are analogues of type modules for imperative, object oriented programming.

Module definitions have the following general format:

<module kind><module name> {<formal type parameters>}
    <optional clauses> == <right hand side>

where < module kind> is one of **type**, **funcs**, **property**, and **class**. In the following explanations any non-exceptional statement about types is valid for classes as well.

The module name of a type module is at the same time the name of the (main) type defined by the module. For the purpose of the present review we may assume that all module (and subtype) names are distinct. The list of formal type parameters is optional. The <optional clauses> of the left hand side include syntax for the introduction of *syntactic*

subtypes of the main type in a type module, see section 5.2, as well as assumption and inclusion clauses, see below.

The right hand side of a module definition is a non-empty text consisting of an optional module expression, called a *module prefix*, followed by an optional *module body* of the form:

module <list of module items> **endmodule** <optional satisfaction clause>

where the final optional clause may be used to express syntactically that the module satisfies a list of properties.

A module expression consists of the module name followed by a list of actual type parameters (if any) enclosed in braces, which are type module expressions. A module expression represents an *instance* of the named module obtained by substituting the actual parameters for the occurrences of the formal ones in its right hand side. A module prefix is an instance of a module of the same kind as the one being defined. (Exception: the module prefix of a class may be a type module instance, fully constructive.) In the case of type modules the one under definition is said to be a *semantic subtype* of the module prefix. In that case the latter may contain a predicate which restricts the value space. See section 5.3 for examples of semantic subtypes.

Module items of the following kinds can, with the mentioned exceptions, occur in all kinds of modules:

- function profiles, described earlier,
- function definitions, described earlier (not allowed in property modules),
- axiom items of the form: **axm** <list of variable declarations><list of formulas>, where the formulas may only have free occurrences of the declared variables, (not allowed in class modules), and
- lemma items headed by the keyword **lma**, and otherwise of the same format as axiom items. The contents of a lemma item can also be an entire module item (other than lemma or axiom item).

In addition any type module shall have exactly one **genbas** statement and at most one **obsbas** statement (possibly inherited). Procedure declarations, as well as function declarations in imperative style, are allowed in class modules.

A module is said to be the *owner* of the module items listed in its module body, if any. In addition it *inherits* those owned by the prefixing module instance, if any, and is thereby made an owner of these items too. For that reason the new module is sometimes said to be an *extension* of the prefix module. The owner relationship is of particular importance for functions, represented by their profiles and semantic specifications, since it gives rise to possibilities for a controlled kind of function overloading, see below. A function owned by a module M is also said to be *associated with* M, and to be a M-function. The set of functions owned by a module must have distinct names, thus no redefinition of inherited functions is allowed, with the single exception of function redefinition in subtypes and subclasses, essentially retaining the original semantics on the reduced domain.

Any non-property module, say $M\{T_1, T_2 \ldots\}$, can introduce requirements on its formal type parameters by an *assumption clause* in its left hand side. The clause lists one or more property module expressions, each with one or more formal M-parameters as actual parameters. The requirements expressed by a property expression, say $P\{T_1, T_2\}$, are that

the actual parameters for T_1 and T_2 of any M-instance, say $M\{U_1, U_2, \ldots\}$, must own functions satisfying the profiles that would be owned by the property instance $P\{U_1, U_2\}$. A function $f : D \longrightarrow C$ is said to *satisfy a profile* $f : D' \longrightarrow C'$ if and only if $D' \preceq D$, and $\forall x : D' \bullet f(x) \in C'$. Notice that profile satisfaction may sometimes be determined by type analysis.

In addition the associated functions must satisfy the axioms of $P\{U_1, U_2\}$. Thus, there is in general a proof obligation associated with any instantiation of M.

Any module can, by an *inclusion clause* listing one or more function module expressions, cause these module instances to be *included*, i.e. they (or more precisely their contents) are made *available* to the module under definition. At the same time those available to the former are also included. Type modules need no explicit inclusion clause in order to become available; an occurrence of a type expression anywhere in the right hand side, or as an actual parameter of a module expression in an assumption or inclusion clause, causes the inclusion of the corresponding type module. The same is true for class modules.

A module M under definition is allowed to refer to any module previously defined. (Exception: no type module may refer to any class module, directly or indirectly.) Thereby a *definition hierarchy*, a partial order, is defined for the set of modules (not module instances) consisting of M and those which have instances available to M, where M is the single maximal element and *Bool* is the single minimal one.

Ownership of functions is not altered by assumptions or inclusions. That makes it possible to avoid name conflicts between functions associated with different modules. It is considered practical to permit function overloading in the sense that the number and types of arguments of a function application can influence the binding of the applied function, with priority for binding to locally owned functions.

Assuming that most functions are owned by type modules, then, in the majority of cases of no local match, it is sufficient to search for a matching profile in modules which are *maximal elements* in the definition hierarchy, among those which occur in the argument types. In practical cases at most one match will be found in these modules (which in general depends on the actual parameters of the occurring module instances as well). If none is found, the function may be defined in a function module, or it may be a relative constant (i.e. the owning type module does not occur in the function domain). In these cases a wider search is required with a correspondingly greater danger of ambiguity especially for non-local constants. If several functions redefined in semantic subtypes can match, the one with the smallest domain which does not require argument coercion, is to be chosen.

As a means to resolve ambiguities, and to override the given overloading rules, ABEL provides notations for identifying the intended owner. (Usually the module name is sufficient, but there exist cases where the actual instance would have to be specified.) An owner M may be specified either by using the notation M' as a prefix or the construct **at** M as a suffix. For ordinary functional notation the prefix notation, $M'f(\ldots)$, reads well, but for mixfix notations ending with an operator symbol the **at**-construct is usually better, as in the examples of section 5.4. For operators and other mixfix notations starting and ending with operands good constructs are difficult to find; in ABEL one can e.g. choose between $M'(\ldots \mathbf{op} \ldots)$, $(\ldots \mathbf{op} \ldots)$ **at** M, and $M'\hat{}\mathbf{op}\hat{}(\ldots, \ldots)$ for specifying the owner of an infix operator.

The advantage of function overloading is illustrated by the fact that any type T, even a formal one, owns the following predefined module items:

func $\hat{} = \hat{} : T \times T \longrightarrow Bool$

func $\hat{}\neq\hat{}$: $T \times T \longrightarrow Bool$
def $x \neq y == \neg x = y$
func if $\hat{}$ **th** $\hat{}$ **el** $\hat{}$ **fi** : $Bool \times T \times T \longrightarrow T$
def **if** x **th** y **el** z **fi** == **case** x **of** **t** $\rightarrow y \mid$ **f** $\rightarrow z$ **fo**

It is required that any actual type module specifies the semantics of the equality relation either by a **def** item or by an **obsbas** statement.

It follows from the above rules for allowed contents of different kinds of modules that classes are necessarily fully constructive, whereas property modules are non-constructive since they do not contain **def** items. Type and function modules are primarily intended to be constructive, but semantics through **axm** items are not forbidden. The intention is, however, that the consistency of modules shall be established through later module extension by constructive function definitions satisfying the stated axioms. Thus, there is in general a proof obligation associated with module extension to prove any inherited axiom referring to defined functions only. The axiom is thereby redefined as a **lma** item. For any property module P consistency is established through proofs required in connection with instantiation of modules assuming P.

As an example of a type module we define the sequence type and a few non-generator functions which are used in the sequel.

type $Seq\{T\}$ ==
module
func ε : $\longrightarrow Seq$
func $\hat{}\vdash\hat{}$: $Seq \times T \longrightarrow Seq$ — right append
1–1 genbas ε, $\hat{}\vdash\hat{}$
func $\hat{}\dashv\hat{}$: $T \times Seq \longrightarrow Seq$ — left append
def $t \dashv q ==$ **case** q **of** $\varepsilon \rightarrow \varepsilon \vdash t \mid q' \vdash t' \rightarrow (t \dashv q') \vdash t'$ **fo**
func $\hat{}\mapsto\hat{}$: $Seq \times Seq \longrightarrow Seq$ — concatenate
def $q \mapsto q' ==$ **case** q' **of** $\varepsilon \rightarrow q \mid q'' \vdash t \rightarrow (q \mapsto q'') \vdash t$ **fo**
func $\hat{}\uparrow\hat{}$: $T \times Nat \longrightarrow Seq$ — repetition
def $t\uparrow n ==$ **case** n **of** $0 \rightarrow \varepsilon \mid \mathbf{S}n' \rightarrow (t\uparrow n') \vdash t$ **fo**
endmodule

Notice that the parameter part is omitted for occurrences of the type under definition. This is in order to prevent a certain class of meaningless type definitions.

Parameterized type modules may be seen as higher order functions, giving new types when applied to type arguments. As such they are *monotonic* with respect to the subtype relation: $T_1 \preceq T_1' \ \wedge \ \ldots \ \wedge \ T_n \preceq T_n' \ \Rightarrow \ U\{T_1, \ldots, T_n\} \preceq U\{T_1', \ldots, T_n'\}$.

5.1 Some standard constructs

We consider the problem of finding a one-to-one generator basis for the type *Int* of integers. In example 3 of section 3.1 a basis consisting of zero, successor, and negation functions was proposed. It is many-to-one, however, since $-0 = 0$ and $--x = x$ for $x : Int$. An alternative way to span the negative integers is to replace the unary minus by a predecessor function $\mathbf{S}\hat{}$, but now $\mathbf{PS}x = \mathbf{SP}x$ for $x : Int$. A third possibility is to define the integers as pairs of sign and absolute value:

type Int == $sgn : \{pos, neg\} \times abs : Nat$

where the right hand side contains some useful special notations. An *enumeration type*, say $\{a, b, \ldots, c\}$, is defined in the obvious way, by a one-to-one generator basis consisting of the listed constants. A *labeled Cartesian type product*, $a_1 : T_1 \times a_2 : T_2 \times \ldots \times a_n : T_n$, $n \geq 0$, is shorthand for a type, *Prodn*, defined as follows:

type $Prodn\{T_1, T_2, \ldots, T_n\}$ ==
module
  func $(\hat{}, \hat{}, \ldots, \hat{}) : T_1 \times T_2 \times \ldots \times T_n \longrightarrow Prodn$
  1–1 genbas $(\hat{}, \hat{}, \ldots, \hat{})$
  func $\hat{}.a_i : Prodn \longrightarrow T_i$ for $i = 1, 2, \ldots, n$ — component selectors
  def $(x_1, x_2, \ldots, x_n).a_i == x_i$ for $i = 1, 2, \ldots, n$
endmodule

where the **def** item is short for a definition by a **case** with a single branch. Notice that the case $n = 0$ is useful; *Prod0* is a kind of null type whose only abstract value is the empty tuple.

However, the one-to-one property of the above labeled product is not quite right, since $(pos, 0)$ and $(neg, 0)$ should be considered equal. In order to arrive at a definition of the *Int* type by conventional means we may use another standard construct, a *disjoint union*:

type *Int* == $0 : Prod0 + pos : Nat1 + neg : Nat1$

where *Nat1* is the type of non-zero natural numbers, **type** *Nat1* == $1 : Prod0 + \mathbf{S}\hat{} : Nat1$, and the "labels" in this case represent generators, sometimes called "injector functions":

func $0 : \longrightarrow Int$
func $pos : Nat1 \longrightarrow Int$
func $neg : Nat1 \longrightarrow Int$

A one-to-one generator basis for *Int* may consist of these three functions.

We may notice in passing that the constructs of type products and labeled disjoint unions are type forming mechanisms of an expressiveness directly comparable to that of type modules with one-to-one generator bases (and without syntactic subtypes). In particular, a recursive type definition corresponds to a generator basis containing generators other than relative constants. Notice that at least one generator must be a relative constant for a basis to be meaningful. Similarly, at least one component of a disjoint union must be non-recursive.

5.2 Syntactic subtypes

The above generator basis for integers, although correct, is not very practical, considering the need for explicit injector functions. Thus, if x is a *Nat1* then the corresponding integer must be written $pos(x)$. A better idea is to define the *Int* type as the head of a family of *syntactic subtypes*, all defined simultaneously. The types *Zero*, *Nat1* and *Neg1* shall be the minimal or *basic* subtypes, where $V_{Zero} = \{0\}$, $V_{Nat1} = \{1, 2, \ldots\}$, and $V_{Neg1} = \{-1, -2, \ldots\}$. By taking the generator domains as suitable subtypes we may obtain a one-to-one basis. Their codomains should be basic types, which are thereby pairwise disjoint by definition: $Zero \sqcap Nat1 = \ldots = \oslash$. Intermediate subtypes may be user defined as indicated. (If some are left out they will be added behind the scenes in order to make the family into a lattice with *Int* as the maximal element and $\oslash$ as the minimal one.)

type *Int* **by** *Zero*, *Nat1*, *Neg1*
and $Nat = Zero \sqcup Nat1$,
$Neg = Zero \sqcup Neg1$,
$Nzo = Nat1 \sqcup Neg1$ ==
module
func $0 : \longrightarrow Zero$
func $\mathbf{S}\hat{} : Nat \longrightarrow Nat1$ — basic successor
func $\mathbf{N}\hat{} : Nat1 \longrightarrow Neg1$ — basic negation
1–1 genbas 0, $\mathbf{S}\hat{}$, $\mathbf{N}\hat{}$
func $-\hat{} : Int \longrightarrow Int$ — negation
def $-x ==$ **case** x **of** $0 \rightarrow 0 \mid \mathbf{N}x' \rightarrow x' \mid$ **others** $\rightarrow \mathbf{N}x$ **fo**
func $pred : Int \longrightarrow Int$ — predecessor
def $pred(x) ==$ **case** x **of** $0 \rightarrow \mathbf{NS}0 \mid \mathbf{S}x' \rightarrow x' \mid \mathbf{N}x' \rightarrow \mathbf{NS}x'$ **fo**
func $succ : Int \longrightarrow Int$ — successor
def $succ(x) ==$ **case** x **of** $\mathbf{N}x' \rightarrow -pred(x') \mid$ **others** $\rightarrow \mathbf{S}x$ **fo**
func $\hat{}+\hat{} : Int \times Int \longrightarrow Int$ —addition
def $x+y ==$ **case** y **of** $0 \rightarrow x \mid \mathbf{S}y' \rightarrow succ(x+y') \mid \mathbf{N}y' \rightarrow -(-x+y')$ **fo**
... ...
endmodule

The following sets of (additional) profiles may be constructed automatically by first typing the right hand side of each function for all possible combinations of (sub-)types for the arguments, including the empty one, and then removing the redundant resulting profiles. For a recursively defined function a profile set may be obtained by iteration, starting with all codomains equal to $\oslash$. In each step the previous profile set is used in the typing algorithm to obtain the next set, and the iteration terminates when two successive sets are equal. Notice that the codomain of a profile represents an "upper type bound" for applications of the function to arguments in the given domain. An actual application either has a value of a basic subtype included in the codomain, or is ill-defined. Profiles with occurrences of $\oslash$ in the domain provide strictness information.

$-\hat{}$: $\oslash \rightarrow \oslash$, $Zero \rightarrow Zero$, $Neg1 \rightarrow Nat1$, $Nat1 \rightarrow Neg1$, $Neg \rightarrow Nat$, $Nat \rightarrow Neg$, $Nzro \rightarrow Nzro$, $Int \rightarrow Int$

pred : $\oslash \rightarrow \oslash$, $Nat1 \rightarrow Nat$, $Neg \rightarrow Neg1$, $Int \rightarrow Int$

succ : $\oslash \rightarrow \oslash$, $Nat \rightarrow Nat1$, $Neg1 \rightarrow Neg$, $Int \rightarrow Int$

$\hat{}+\hat{}$: $\oslash \times Int \rightarrow \oslash$, $Int \times \oslash \rightarrow \oslash$, $Nat1 \times Nat \rightarrow Nat1$, $Nat \times Nat1 \rightarrow Nat1$, $Neg1 \times Neg \rightarrow Neg1$, $Neg \times Neg1 \rightarrow Neg1$, $Zero \times Nzro \rightarrow Nzro$, $Nzro \times Zero \rightarrow Nzro$, $Zero \times Zero \rightarrow Zero$, $Nat \times Nat \rightarrow Nat$, $Neg \times Neg \rightarrow Neg$, $Int \times Int \rightarrow Int$

5.3 Semantic subtypes

Computers work with numbers of limited size, which can be identified as a semantic subtype of the corresponding unrestricted type. Thus, the following subtype of natural numbers would be appropriate for ten-bit number representation:

type *TenBitNat* == $n : Nat$ **where** $n \leq 1023$
module
  func $\mathbf{S}\hat{}$: *TenBitNat* $\longrightarrow$ *TenBitNat*
  def $\mathbf{S}n == (Nat\text{'}\mathbf{S}n)$ **as** *TenBitNat*
  ... ...
endmodule

In redefining an inherited function it is permitted to modify the profile by replacing any occurrence of the prefix type (or any supertype of it) by the subtype under definition. The redefined function must behave as the inherited one, except possibly for a possible error caused by a final coercion.

In order to give a more substantial example we define a fragment of a type module of binary trees of nodes containing values of some type T, to be subsequently restricted to the subtype of "search trees".

type *BinTree*$\{T\}$ ==
module
  func *nil* : $\longrightarrow$ *BinTree* — empty tree
  func *tree* : *Bintree* $\times$ T $\times$ *Bintree* $\longrightarrow$ *Bintree* — non-empty tree
  1–1 genbas *nil*, *tree*
  func *infix* : *Bintree* $\longrightarrow$ *Seq*$\{T\}$
  def *infix*(b) == **case** b **of** *nil* $\rightarrow \varepsilon$ | *tree*$(l, t, r) \rightarrow$ *infix*$(l) \vdash t \dashv$ *infix*(r) **fo**
endmodule

The *infix* function computes the sequence of node values taken in infix order. Search trees are binary trees whose infix sequences are *sorted*. In order to define that notion we first have to introduce a suitable concept of *ordering relation*.

property *SortOrd*$\{T\}$ ==
module
  func $\hat{} < \hat{}$: $T \times T \longrightarrow$ *Bool*
  axm $x, y, z : T \bullet$
    d $x < y$,
    $x < y < z \Rightarrow x < z$,
    $x < y \Rightarrow x < z \vee z < y$
  lma $x, y, v, w : T \bullet$
    $x < w < y \wedge \neg v < w \wedge \neg w < v \Rightarrow x < v < y$
endmodule

where the first axiom specifies $\hat{} < \hat{}$ to be a total function.

When defining a general concept of sortedness of sequences an ordering relation must be assumed for the element type, and for maximum generality the assumption should be as weak as possible. The property module *SortOrd* expresses the weakest notion of order which permits a concept of sorted sequences, such that sortedness is maintained by element removal and correct insertion. As indicated by the notation, a "strong" ordering relation

is intended, like $\hat{}<\hat{}$ on integers. It may be noticed, however, that $\hat{}\leq\hat{}$, $\hat{}>\hat{}$, and many other binary relations over different types satisfy the same axioms. This indicates that a mechanism for function identifier substitution will be useful when instantiating modules. In ABEL a notation exemplified as follows is used: $\mathit{SortOrd}\{U\}$ **with** $\hat{}\leq\hat{}$ **for** $\hat{}<\hat{}$.

funcs $\mathit{SeqSort}\{T\}$ **assuming** $\mathit{SortOrd}\{T\}$ ==
module
  func $/<\hat{}:\ \mathit{Seq}\{T\} \longrightarrow \mathit{Bool}$ — sorted wrt. $<$
  def $/< q ==$ **case** q **of** $\varepsilon \rightarrow$ **t** $\mid q' \vdash x \rightarrow$
    case q' **of** $\varepsilon \rightarrow$ **t** $\mid q'' \vdash y \rightarrow /< q' \wedge y < x$ **fo fo**
  lma $q_1, q_2, q_3 : \mathit{Seq}\{T\},\ x : T \bullet$
    $/< (q_1 \mapsto q_2 \mapsto q_3) \Rightarrow /< q_2,$
    $/< (q_1 \vdash x) \wedge /< (x \dashv q_2) \Rightarrow /< (q_1 \vdash x \dashv q_2)$
endmodule

The fact that *SortOrd* is assumed entails the following consistency requirements for any actual parameter U for T: that the function profile $\hat{}<\hat{} : U \times U \longrightarrow \mathit{Bool}$ is satisfied by a U-function (or is assumed for U if U is in turn a formal parameter, as in the example below), and that this function satisfies the axioms of that property module.

We are now in a position to define the concept of search trees as a semantic subtype of binary trees, using the two auxiliary modules above.

type $\mathit{SearchTree}\{T\}$ **assuming** $\mathit{SortOrd}\{T\}$
    including $\mathit{SeqSort}\{T\}$ ==
    $b : \mathit{Bintree}\{T\}$ **where** $/< \mathit{infix}(b)$ **convex**
module
  func $\mathit{lkp} : \mathit{SearchTree} \times T \longrightarrow T$ — look up
  def $\mathit{lkp}(s,t) ==$ **case** s **of** $\mathit{nil} \rightarrow \perp \mid \mathit{tree}(l, t', r) \rightarrow$
    if $t < t'$ **th** $\mathit{lkp}(l,t)$ **el**
    if $t' < t$ **th** $\mathit{lkp}(r,t)$ **el** t' **fi fi fo**
  func $\mathit{add} : \mathit{SearchTree} \times T \longrightarrow \mathit{SearchTree}$ — add or replace
  def $\mathit{add}(s,t) ==$ **case** s **of** $\mathit{nil} \rightarrow \mathit{tree}(\mathit{nil}, t, \mathit{nil}) \mid \mathit{tree}(l, t', r) \rightarrow$
    if $t < t'$ **th** $\mathit{tree}(\mathit{add}(l,t), t', r)$ **el**
    if $t' < t$ **th** $\mathit{tree}(l, t', \mathit{add}(r,t))$
    el $\mathit{tree}(l,t,r)$ **fi fi fo qua** $\mathit{SearchTree}$
endmodule

The consistency requirements on the included *SeqSort* instance are validated syntactically by the assumption in its environment. Notice that the function value of the look-up function may contain non-redundant information, since $\neg t < t \wedge \neg t' < t$ does not imply $t = t'$ for the weak ordering relation assumed. For the search tree property to be maintained by *add* the node value t' must in that case be replaced by t.

The keyword **convex** asserts an important property of the defined subtype: that any subtree of a search tree is itself a search tree. In general the formulas which must be proved in order to establish the convexity property are determined mechanically by the generator basis and the restricting predicate. In this case there is an obligation to prove: $/< \mathit{infix}(\mathit{init})$, and $/< \mathit{infix}(\mathit{tree}(l,w,r)) \Rightarrow /< \mathit{infix}(l) \wedge /< \mathit{infix}(r)$. The proofs follow using the definition of the *infix* function in module *BinTree* and lemma 1 of module *SeqSort*.

The convexity has the syntactic consequence that the variables l and r introduced in discriminators on search trees are of type *SearchTree* (not *BinTree*), thereby avoiding

coercions of arguments to *lkp* and *add* in the recursive definitions of these functions. An important semantic consequence is that induction hypotheses are justified for subtrees in proofs by generator induction over search trees.

The notation **qua** *SearchTree* in the *add* definition serves to avoid coercion of the function body which is syntactically of the type *BinTree*. There is an associated obligation to prove that the function value is in fact a search tree. That can be achieved by proving

$$/<(q_1 \vdash \mathit{infix}(s) \dashv q_2) \wedge /<(q_1 \vdash v \dashv q_2) \Rightarrow /<(q_1 \vdash \mathit{infix}(\mathit{add}(s,t)) \dashv q_2)$$

by generator induction on $s : \mathit{SearchTree}$ for arbitrary node value sequences q_1 and q_2. In the non-trivial case that s is of the form $\mathit{tree}(l, t', r)$ the proof goes through using induction hypotheses for l and r, as well as lemma 2 of the *SeqSort* module and the lemma of module *SortOrd*. Then, by taking $q_1 = q_2 = \varepsilon$ we obtain the desired result: $/<\mathit{infix}(s) \Rightarrow /<\mathit{infix}(\mathit{add}(s,v))$. (Notice that the restricting predicate may be assumed for any search tree.)

5.4 Many-to-one generator bases

We define a type of abstract finite maps from a "domain type" X to a "codomain type" Y. A map may be compared to a partial function from X to Y, which is defined for at most a finite number of arguments.

type $\mathit{Map}\{X,Y\}$ ==
module
  func *init* : $\longrightarrow \mathit{Map}$ — empty map
  func ^[^↦^]: $\mathit{Map} \times X \times Y \longrightarrow \mathit{Map}$ — update map
  genbas *init*, ^[^↦^]
  func ^[^] : $\mathit{Map} \times X \longrightarrow Y$ — apply map
  def $m[x]$ == **case** m **of** $\mathit{init} \rightarrow \bot \mid m'[x' \mapsto y] \rightarrow$
    if $x = x'$ **th** y **el** $m'[x]$ **fi fo**
  obsbas ^[^]
endmodule

The **obsbas** statement defines the equality relation on maps as the strict restriction of:

$$(m_1 = m_2) \;==\; \forall x : X \bullet \; (m_1[x] == m_2[x])$$

which may be useful for proof purposes, but is not a constructive definition. The definedness predicate **d**^[^] of the map application function may be derived as explained in section 4. The result is the following TGI definition:

def **d** $m[x]$ == **case** m **of** $\mathit{init} \rightarrow$ **f** $\mid m'[x' \mapsto y] \rightarrow x = x'$ **or** **d** $m'[x]$ **fo**

By means of the definedness predicate equality on maps may be defined without the use of strong equality in the right hand side:

$$(m_1 = m_2) \;==\; \forall x : T \bullet \; \mathbf{d}\, m_1[x] = \mathbf{d}\, m_2[x] \wedge (\mathbf{d}\, m_1[x] \Rightarrow m_1[x] = m_2[x])$$

The equality definition still is not constructive, due to the quantifier in the right hand side. It is possible, however, to give a definition of the *Map* type entirely within the TGI framework in terms of a semantic subtype. The idea is to restrict the generator universe to a set of *canonic forms*, one for each equivalence class. Then, redefining the generators

so that they generate canonic maps, the subtype is made to appear to have a one-to-one generator basis. This in turn makes it possible to give a simple TGI redefinition of the equality relation. Assuming that there is a *total order* $\hat{}<\hat{}$ on the "argument" type X of *Map*, we can identify canonic representatives of the form:

$$init[x_1 \mapsto y_1][x_2 \mapsto y_2] \ldots [x_n \mapsto y_n], \quad \text{for} \quad x_1 < x_2 < \ldots < x_n \quad \text{and} \quad n \geq 0.$$

The notion of total order may be expressed as follows:

property $TotOrd\{X\} == SortOrd\{X\}$
module
  axm $x, y : X \bullet$
    $\neg(x < y \wedge y < x)$,
    $x < y \vee x = y \vee y < x$
endmodule

Since *TotOrd* is defined as a extension of *SortOrd*, it inherits the contents of the latter. Seeing a property as the conjunction of its axioms, the inheritance implies that the extended property is *stronger*. Thus $TotOrd\{T\} \Rightarrow SortOrd\{T\}$ holds for arbitrary type T, and the implication is established syntactically.

Assuming this property for the map argument type we can express formally the concept of map canonicity:

funcs $MapCan\{X, Y\}$ **assuming** $TotOrd\{X\}$ $==$
module
  func $canonic : Map\{X, Y\} \longrightarrow Bool$
  def $canonic(m) ==$ **case** m **of** $init \rightarrow$ **t** $\mid m_1[x_1 \mapsto y_1] \rightarrow$
    case m_1 **of** $init \rightarrow$ **t** $\mid m_2[x_2 \mapsto y_2] \rightarrow$
    $x_2 < x_1 \wedge canonic(m_1)$ **fo fo**
endmodule

Finally we may define a convex subtype of *canonic maps*:

type $CanMap\{X, Y\}$ **assuming** $TotOrd\{X\}$
    including $MapCan\{X, Y\}$ $==$
    $m : Map\{X, Y\}$ **where** $canonic(m)$ **convex**
module
  func $init : \longrightarrow CanMap$
  func $\hat{}[\hat{}\mapsto\hat{}]$: $CanMap \times X \times Y \longrightarrow CanMap$
  def $m[x \mapsto y] ==$ **case** m **of** $init \rightarrow m[x \mapsto y]$ **at** $Map \mid m_1[x_1 \mapsto y_1] \rightarrow$
    if $x_1 < x$ **th** $m[x \mapsto y]$ **at** Map **el**
    if $x < x_1$ **th** $m_1[x \mapsto y][x_1 \mapsto y_1]$ **at** Map
    el $m_1[x \mapsto y]$ **at** Map **fi fi fo qua** $CanMap$
  func $\hat{} = \hat{} : CanMap \times CanMap \longrightarrow Bool$
  def $m_1 = m_2 ==$ **case** (m_1, m_2) **of** $(init, init) \rightarrow$ **t**
    $\mid (m_1[x_1 \mapsto y_1], m_2[x_2 \mapsto y_2]) \rightarrow m_1 = m_2 \wedge x_1 = x_2 \wedge y_1 = y_2$
    $\mid$ **others** $\rightarrow$ **f fo**

  func $crep : Map \longrightarrow CanMap$ — canonic repr.
  def $crep(m) ==$ **case** m **of** $init \rightarrow m \mid m'[x \mapsto y] \rightarrow crep(m')[x \mapsto y]$ **fo**
endmodule

In order to establish the convexity there is a obligation to prove: $canonic(init)$ and $canonic(m[x \mapsto y]) \Rightarrow canonic(m)$, which is easy. In the semantic redefinition of the generator $\hat{}[\hat{}\mapsto\hat{}]$ there is a need to refer to the original generator. Therefore the standard rule of function overloading must be overruled. It is practical to use the **at**-construct in this case because the mixfix notation of the function begins with an operand and ends with an operator symbol. Notice that the innermost (i.e. leftmost) generator application of $m_1[x \mapsto y][x_1 \mapsto y_1]$ **at** Map refers to the redefined one (since the variable m_1 is of type $CanMap$).

The **qua**-construct introduces an obligation to prove that the redefined function generates canonic maps: $canonic(m[x \mapsto y])$ for $m : CanMap$. The latter proof goes through by generator induction on m, using the lemma $canonic(m[x' \mapsto y']$ **at** $Map) \wedge x < x' \Rightarrow canonic(m[x \mapsto y][x' \mapsto y']$ **at** $Map)$, for $m : CanMap$, also provable by induction on m.

Both redefined functions must behave as the original ones on the restricted domains. So one has to prove in addition:

$$\begin{array}{ll} \forall x : X \bullet\ m[x' \mapsto y][x] == m[x' \mapsto y]\ \textbf{at}\ Map[x] & \text{and} \\ m = m' \Leftrightarrow \forall x : X \bullet\ (m[x] == m'[x]), & \text{for}\ \ m, m' : CanMap. \end{array}$$

The proofs, which are by generator induction on m and m', are easy. The redefined equality relation corresponds to that of a one-to-one generator basis. This implies that consistency can not be violated through the definition of functions by means of generator induction over $CanMap$.

The function $crep$ computes the canonic representative of an arbitrary map, and can thus be seen as a mechanism of "unfailing" coercion to the subtype. Notice that function overloading implies that the generator applied to $crep(m')$ in the $crep$ definition is the one redefined in $CanMap$. Also the type (codomain) of $init$ has been redefined locally. Therefore type analysis shows that the body of $crep$ is of type $CanMap$, which implies that the function value is indeed a canonic map. It is fairly easy to see that it is idempotent, and that $crep(m) = m$ for $m : CanMap$, and thus for $m : Map$ and Map equality.

It may be noticed that the predicate $canonic$ and the corresponding redefined generator $\hat{}[\hat{}\mapsto\hat{}]$ are the only non-trivial parts of the construct $CanMap$. In particular the redefined equality relation and the $crep$ function could be specified automatically, given a syntactic indication of the purpose of the subtype. Also all the proof obligations may be identified mechanically. The proof concerning the redefined equality will in general only go through if the canonic forms described by the restricting predicate are in fact in a one-to-one correspondence with the abstract values defined for the supertype.

The $CanMap$ type invites function specifications that exploit of the properties of canonic representations in order to improve the efficiency of expression evaluation, seeing TGI specifications as applicative programs. This, however, is likely to make the function definitions look more like algorithmic implementations than abstract mathematical specifications, which is not always an advantage for logical reasoning and easy understanding. For instance, let the composition operator $\hat{}\oplus\hat{} : Map \times Map \longrightarrow Map$ denote the "union" of two maps, where components of the right argument override corresponding components of the left argument. The following semantic definition is easy to understand:

$$\textbf{def}\ \ m \oplus m' == \textbf{case}\ m'\ \textbf{of}\ init \rightarrow m \mid m_1[x \mapsto y] \rightarrow (m \oplus m_1)[x \mapsto y]\ \textbf{fo}$$

A more execution efficient version can be made for canonic maps, using an algorithm similar to that of merging two sorted sequences (in order to establish the TGI property of this definition one must consider the combined complexity of both operands):

$$\begin{array}{l}\textbf{def}\ \ m \oplus m' == \textbf{case}\ m\ \textbf{of}\ init \to m' \mid m_1[x_1 \mapsto y_1] \to \\ \qquad\qquad\qquad\quad \textbf{case}\ m'\ \textbf{of}\ init \to m \mid m_2[x_2 \mapsto y_2] \to \\ \qquad\qquad\qquad\quad \textbf{if}\ x_1 < x_2\ \textbf{th}\ (m \oplus m_2)[x_2 \mapsto y_2]\ \textbf{at}\ Map\ \textbf{el} \\ \qquad\qquad\qquad\quad \textbf{if}\ x_2 < x_1\ \textbf{th}\ (m_1 \oplus m')[x_1 \mapsto y_1]\ \textbf{at}\ Map \\ \qquad\qquad\qquad\quad \textbf{el}\ (m_1 \oplus m_2)[x_2 \mapsto y_2]\ \textbf{at}\ Map\ \textbf{fi fi qua}\ CanMap\ \textbf{fo fo}\end{array}$$

This indicates that specifications optimized with respect to ease of reasoning and understanding naturally belong to the *Map* module, (although there is no consistency guarantee for functions defined by generator induction over the *Map* type), whereas a redefinition with better execution efficiency may be given in *CanMap*. As usual, one would be obliged to prove the strong equality of the two definitions as applied to canonic maps.

The following stronger result also proves the consistency of the definition of $\hat{}\oplus\hat{}$ in *Map*: $(crep(m) \oplus crep(m'))[x] == (m \oplus m')[x]$, for $m, m' : Map$, $x : X$, where the *crep* function computes the canonic representative of an arbitrary map, and the definition of equality on *Map* through the observation basis has been used. Notice that the redefined $\hat{}\oplus\hat{}$ operator is applied in the left hand side. If the operator is not redefined then proof of $(m \oplus crep(m'))[x] == (m \oplus m')[x]$ is sufficient to show consistency of the definition in *Map*, because generator induction is only applied to the second argument. Again, these proof obligations may be identified mechanically. Notice that they are formulas at the TGI level since the universal quantifier occurring in the equality definition could be made implicit. That is not the case for a proof obligation in the form of a congruence axiom.

Also the map application function $\hat{}[\hat{}]$ could be redefined with better efficiency for canonic maps, however, we leave it to the reader to provide a more execution efficient version, and to prove it correct.

5.5 Type simulation

The fact that types easy to reason about are often impractical for computational purposes implies a dilemma as far as program development is concerned; one may have to choose between easy reasoning and computational efficiency. Fortunately, however, there is a way of achieving both ends: we may reason in terms of an easy type T and compute in terms of an efficient type T', provided that T' *simulates* T in a certain formal sense. We shall explain the concept of type simulation using an example.

More efficient versions of several functions on (canonic) maps may be obtained (on the average) by representing maps as search trees. We can make use of the type *SearchTree* defined above by first introducing a type of *nodes* consisting of a *key* part and a *data* part:

$$\begin{array}{l}\textbf{type}\ Node\{Key, Data\}\ \textbf{assuming}\ SortOrd\{Key\} == key : Key \times data : Data \\ \textbf{module} \\ \qquad \textbf{func}\ \hat{}<\hat{} : Node \times Node \longrightarrow Bool \\ \qquad \textbf{def}\ \ x < y == x.key < y.key \\ \textbf{endmodule satisfying}\ SortOrd\{Node\}\end{array}$$

A proof that the *Node* type satisfies the *SortOrd* property consists in proving that the *SortOrd* axioms, $\neg(x < y \wedge y < x)$ and $x < y \vee x = y \vee y < x$, hold for $x, y : Node$. This is trivial on the assumption that they are satisfied for *Key* values. Given these proofs it is established that $SortOrd\{X\}$ implies $SortOrd\{Node\{X, Y\}\}$ for arbitrary types X and Y.

The type $SearchTree\{Node\{X, Y\}\}$ *simulates*, in the sense defined below, the type $CanMap\{X, Y\}$, for arbitrary types X, Y which satisfy the assumption of *CanMap*. (This

assumption, $TotOrd\{X\}$, implies $SortOrd\{X\}$, which in turn implies the property assumed by the *SearchTree* module for its actual parameter, $SortOrd\{Node\{X,Y\}\}$. We may thus conclude, without additional proof obligations, that the actual parameter of the *SearchTree* module satisfies the property assumed for the formal parameter.)

1. There is a total function, often called an "abstraction function", transforming "concrete" search trees to "abstract" canonic maps:

   $$\begin{array}{l}\textbf{func } \mathcal{A} : SearchTree\{Node\{X,Y\}\} \longrightarrow CanMap\{X,Y\}\\ \textbf{def } \ \mathcal{A}(t) == \mathcal{A}'(infix(t)), \quad \text{where}\\ \textbf{func } \mathcal{A}' : \{q : Seq\{Node\{X,Y\}\} \textbf{ where } /\!<\! q\} \longrightarrow CanMap\{X,Y\}\\ \textbf{def } \ \mathcal{A}'(q) == \textbf{case } q \textbf{ of } \varepsilon \to init \mid q' \vdash n \to \mathcal{A}'(q')[n.key \mapsto n.data] \textbf{ fo}\end{array}$$

   (The fact that the argument q is a sorted sequence shows that it would be sufficient to use the simple version of the generator $\hat{}[\hat{}\mapsto\hat{}]$.)

2. Let $m = \mathcal{A}(t)$ and $m' = \mathcal{A}(t')$. Then the functions of *SearchTree* simulate those of *CanMap* as follows:

   $$\begin{array}{rl} m = m' & == infix(t) = infix(t'),\\ init & == \mathcal{A}(nil),\\ m[x \mapsto y] & == \mathcal{A}(add(t,(x,y))), \text{ and}\\ m[x] & == lkp(t,(x,\hat{}))\,.data,\end{array}$$

   where the second component of the second argument to *lkp* is redundant. Since the abstract generators are simulated (strongly) by concrete functions, we may conclude that all abstract values have concrete representatives, i.e. that the abstraction function is "onto" its codomain.

The simulation relationship can be established by proving the criteria 2 as they stand, for the given abstraction function. An alternative, which is somewhat simpler, is to prove the "abstract" axioms of the *CanMap* module, **case**-free versions, translated in terms of *SearchTree* functions:

$$\begin{array}{rll}
1. & infix(nil) = infix(nil) & == \mathbf{t}\\
2. & infix(add(t,(x,y)) = infix(nil) & == \mathbf{f}\\
3. & infix(nil) = infix(add(t,(x,y)) & == \mathbf{f}\\
4. & infix(add(t_1,(x_1,y_1)) = infix(add(t_2,(x_2,y_2)) & == t_1 = t_2 \wedge x_1 = x_2 \wedge y_1 = y_2)\\
5. & infix(add(nil,(x,y))) & == infix(add(nil,(x,y)))\\
6. & infix(add(add(t,(x_1,y_1)),(x,y))) & == infix(\textbf{if } x_1 < x \textbf{ th } add(add(t,(x_1,y_1)),(x,y))\\
 & & \quad \textbf{el if } x < x_1 \textbf{ th } add(add(t,(x,y)),(x_1,y_1))\\
 & & \quad \textbf{el } add(t,(x,y)) \textbf{ fi fi})\\
7. & lkp(nil,(x,\hat{})) & == \bot\\
8. & lkp(add(t,(x_1,y)),(x,\hat{})) & == \textbf{if } x = x_1 \textbf{ th } y \textbf{ el } lkp(t,(x,\hat{})) \textbf{ fi}
\end{array}$$

The formulas 1-4 are the translations of equality axioms, 5 and 6 correspond to the redefined generator axioms, and the last two are the translations of the inherited axioms for map application. Proofs of no. 4 and no. 8 follow from the observation that the *add* applications occurring in the left hand sides are translations of canonic maps and therefore right linear search trees. No. 6 requires a proof of the lemma $infix(add(add(t,(x_1,y_1)),(x,y)) == infix(add(add(t,(x,y)),(x_1,y_1))$. The remaining proofs are entirely trivial. It can be shown

that the truth of 1-8 implies the existence of a total abstraction function satisfying the criteria 2 above.

According to the definition of the simulation relation the type $CanMap\{X, Y\}$ simulates its supertype $Map\{X, Y\}$, where the abstraction function is the identity, and the redefined functions in *CanMap* directly simulate those of *Map*. Since the simulation is a transitive relation on types, it follows that $SearchTree\{Node\{X, Y\}\}$ also simulates $Map\{X, Y\}$.

A type is said to simulate another type *partially* if the simulating functions represent approximations to the abstract ones. In particular, a semantic subtype whose abstract value space is a proper subset of that of the supertype simulates the latter partially if the generators and other functions are redefined. In that case redefined producer functions, and generators, may be less defined on the subdomain than the original functions. For instance the type *TenBitNat* defined at the beginning of section 5.3 is a partial simulation of *Nat*.

5.6 Classes

A typed value, like the number 3, is a mathematical object represented by an immutable data structure in a running program. The same is true for values representing large volumes of data, like long sequence values or trees with many nodes. In an applicative environment there is no such thing as making changes to existing data structures, so, instead one has to create new values, may be from scratch, even for high volume structures equal to existing ones except for small changes. There are cases where one can not afford the luxury to ignore inefficiencies like that, but has to use an imperative approach in order to assume more direct responsibility for the use of storage space and computing time. In particular, it may be necessary to manipulate high volume data structures by incremental updating.

This motivates the introduction of *classes*, which are similar to types, except for the following differences:

1. Whereas the generator functions of type modules give rise to immutable *values*, those of a class give rise to *objects* whose contents or state may be subsequently changed.

2. Imperative style procedure declarations are allowed as module items of a class, with the syntax:

   proc <name> (**var** <varpar>, **val** <valpar>) ==<body>

   where <varpar> is a list of typed formal *variable parameters* (or in/out-parameters) and <valpar> is a typed list of formal *value parameters* (or in-parameters). The latter is the default parameter kind. <body> is a list of imperative style statements, which may include assignment operations, procedure invocations, alternatives by **if**- or **case**-constructs, and loop constructs.

3. assignment operations as well as parameter (argument) transmission is by *pointer copying* for class objects in order to avoid unnecessary copying of high volume structures, whereas these mechanisms conceptually are by value copying for typed values.

A procedure invocation is a statement of the form **call** <name> (v, e), where v is a list of variables, the actual variable parameters, and e is a list of expressions, the actual value parameters. The net effect of this call will be a simultaneous assignment to the actual variable parameters, $v := f_{\text{name}}(v, e)$, where f_{name} is called the *effect function* of the

procedure. Its function value is a tuple whose components correspond to the individual variable parameters.

The state space of an object of a given class C is equal to the value space of a corresponding type T_C. The latter may be obtained by modifying the class module definition as follows:

1. The initial keyword **class** is replaced by the keyword **type**.
2. All references to classes in the definition are replaced by references to the corresponding types.
3. Procedure declarations are replaced by the introduction and definition of corresponding effect functions.

A class C is said to (partially) simulate a type T if its associated type T_C is a (partial) simulation of T. In that case T can be taken as a (partial) specification of C, i.e. an abstract interface.

ABEL provides a mechanism for the internal updating of objects by defining the variables introduced by discriminators to be *assignable* whenever the discriminand of the **case**-construct is a class object (other than a formal value parameter). The combination of pointer copying and internal object updates may result in a quite complicated program logic, unless the use of object expressions and assignments to object variables are restricted. The problem is that confluent accessible pointers are object aliases which may result in difficult side effects in updating operations. There do exist restrictions, syntactic except for the use of subscripted variables, which are sufficient to prevent aliasing by pointers, and still allow efficient programs up to a point, see [3].

Freedom from pointer alias implies that program semantics are exactly as if classes are replaced by their corresponding types (provided that internal updates are interpreted as wholesale object assignments).

As an example we first define a concept of "lists" of T-elements in the form of a type (which is nothing but our old sequence concept in disguise):

type $List\{T\}$ $==$
module
  func $nil : \longrightarrow List$
  func $extend : List \times T \longrightarrow List$
  1–1 genbas $nil,\ extend$
  func $append : T \times List \longrightarrow List$
  def $append(t, l) ==$
  case l **of** $nil \rightarrow extend(l, t) \mid extend(l', t') \rightarrow extend(append(l', t), t')$ **fo**
endmodule

Notice that the *append* operator extends a list at the "wrong" end, and for that reason a whole new list value must be created by the repeated use of the *extend* generator.

As a contrast, let us define the list concept as a class. Then the append operation can be defined in the form of a more efficient procedure, which generates only one new *extend*-object:

class $LIST\{T\}$ $==$
module

```
        func nil : ⟶ LIST
        func extend : LIST × T ⟶ LIST
        1–1 genbas nil, extend
        proc Append(val t : T, var l : LIST) ==
        case l of nil → l := extend(l, t) | extend(l′, t′) → call Append(l′, t) fo
    endmodule
```

The type associated with this class is exactly the type *List*; in particular the *append* function is the effect function of the *Append* procedure. That can be proved using standard Hoare Logic on the body of the latter.

Although the *Append* procedure need not regenerate the old list, it does have to search for the far end of it. For maximum efficiency to be obtained a pointer to the last list element would have to be stored as part of the data structure, for instance as follows:

```
    class FIFOLIST{T} == (first : LIST{T} × last : LIST{T}) where
                         last = case first of nil → nil | others → end(first) fo
    module
        proc .APPEND(val t : T) == const new := extend(nil, t);
                                   case last of nil → first, last := new, new
                                   | extend(l, t′) → l, last := new, new fo
    endmodule
```

where *end* is a function locating the last element of a non-empty list. Whenever a class is a subclass of a labeled product, we use the ad-hoc notation of a dot to the left of a local function or procedure name to indicate an implied parameter of the class under definition. And since a product has only one generator, we omit the case construct as usual and let the implied argument be a tuple whose components are named as indicated by the labels. Thus *.APPEND* stands for *(first, last).APPEND* in the procedure declaration. The implied parameter of a procedure is by definition a **var** parameter; thus by this syntactic trick the labels of the class prefix are made to behave as assignable variables, so-called "representation variables". The dot notation is used in procedure calls as well. Thus, for *FL* : *FIFOLIST* the statement **call** *FL.APPEND*(*a*) would append the *T*-value *a* at the far end of the list *FL*. In this way subclasses of labeled type or class products are made to resemble classes as in SIMULA 67.

Individual assignments to representation variables will in general violate the restricting predicate of the class, if any, the so-called "representation invariant". Therefore there is an obligation to prove that any updating procedure maintains the invariant, and that it is established by any generating expression. By restricting **case**-constructs on objects to occur textually within the corresponding class body, the maintenance of the representation invariant is ensured by proofs of local procedures. Notice that a notation like *FL.last* is an application of a selector function, not an updatable variable. If a class is intended to simulate an "abstract" type, it would be reasonable to disallow all external access to the internal data structure of the objects, in order to prevent confusion of different levels of abstraction.

Unfortunately the pointer *last* of our final example causes an alias on the end element of a nonempty list. Consequently the verification of this class can not be made using ordinary Hoare Logic. However, the user of any instance of the class *FIFOLIST* will be protected from the difficulties caused by the internal alias if external access to the representation variables *first* and *last* is impossible.

6 A case study

The following requirements to a lift control system with n lifts in a building with m floors are formulated by Neil Davis of STL, England:

1. Each lift has a set of buttons, one button for each floor. These illuminate when pressed and cause the lift to visit the corresponding floor. The illumination is canceled when the corresponding floor is visited (i.e. stopped at) by the lift.

2. Each floor has two buttons (except low and high), one to request an up-lift and one to request a down-lift. These buttons illuminate when pressed. The buttons are canceled when a lift visits the floor and is either traveling in the desired direction, or visiting the floor with no requests outstanding. In the latter case if both floor requests are illuminated, only one should be canceled. The algorithm used to decide which to serve should minimize the waiting time for both requests.

3. When a lift has no requests to service, it should remain at its final destination with its doors closed and await further requests (or model a "holding" floor).

4. All requests for lifts from floors must be serviced eventually, with all floors given equal priority (can this be proved or demonstrated)?

5. All requests for floors within lifts must be serviced eventually, with floors being serviced sequentially in the order of travel (can this be proved or demonstrated)?

It is difficult to check whether the lift system is over- or under-specified. For instance, the following two requirements are not expressed above, but seem quite obvious:

- A lift should not make unrequested stops.

- A lift should not pass a floor if there is a request on that floor for a lift in the direction of travel.

The informal description may be understood in many ways. One may ask whether the minimization of waiting time, stated in requirement 2, should imply the following requirement:

- A lift may leave a floor only when approaching another floor with an outstanding (internal or external) request. (No unuseful moves)

These observations motivate the need for a formal treatment.

We shall present an abstract specification of the lift control system described above. All requirements (including the three additional ones) will be formalized, except that our specification will not be concerned with priorities or minimization of waiting time. In order to demonstrate object oriented programming with the ABEL class concept, we present an imperative implementation of the lift system with class objects corresponding to the hardware components of a real lift system, such as doors and buttons with associated signals. The implementation may be improved by adding emergency buttons. Other aspects of real lifts like "fullness" may also be added.

6.1 Abstract specification

In order to formalize the above requirements, we shall use (time) sequences of events, considering events caused by a user of the lift system (*InEvents*) and observable events caused by the lift system (*OutEvents*). An execution may be seen as a possibly infinite sequence of events. However, in order to avoid infinite sequences, our specification will be expressed by means of finite, initial parts of these sequences, called *histories*, in the style of [1]. Safety is expressed by specifying that each possible history r satisfies some predicate on r. It is not obvious how to specify liveness by means of finite sequences, and neither how to give a complete specification of the lift system.

However, it is possible to give an abstract specification of the lift system by its *ready set*, i.e. the set of events which the system is ready to accept next (at a given point in an execution) [8]. The ready set is formalized as a relation, **ready**, between an event-sequence and an event, such that $r\,\mathbf{ready}\,e$ expresses that e is in the ready set when r is the event history [14]. In order to avoid non-determinism in the specification we consider a fixed but arbitrary execution (i.e., there is an implicit outermost universal quantifier on the event history). For instance, $(r\,\mathbf{ready}\,a \wedge \neg r\,\mathbf{ready}\,b) \vee (r\,\mathbf{ready}\,b \wedge \neg r\,\mathbf{ready}\,a)$ expresses that the lift system, after history r, may either choose (perhaps due to "internal non-determinism") to be ready for an a-event or a b-event, but not both.

Safety properties are expressed by specifying non-readyness; in particular P is an *execution invariant* if $P(\varepsilon)$ holds and if $P(r) \wedge r\,\mathbf{ready}\,e \Rightarrow P(r \vdash e)$. (When rewritten as $P(r) \wedge \neg P(r \vdash e) \Rightarrow \neg r\,\mathbf{ready}\,e$, the non-readyness becomes evident.) For instance, the predicate **hist** ˆ defined by

$$\mathbf{def}\ \ \mathbf{hist}\ r == \mathbf{case}\ r\ \mathbf{of}\ \varepsilon \rightarrow \mathbf{t} \mid r' \vdash e \rightarrow \mathbf{hist}\ r' \wedge r'\,\mathbf{ready}\,e\ \mathbf{fo}$$

is an execution invariant, identifying the possible lift system histories.

Formulas of the form $Q(r,e) \Rightarrow r\,\mathbf{ready}\,e$ express basic (one-step) liveness properties. For instance, $e \in InEvent \Rightarrow r\,\mathbf{ready}\,e$ expresses that the lift system is always ready to accept an input event. Similarly, we may express that if there are unserved requests, there will be at least one out-event in the ready set. Other liveness properties may be proved by well-founded induction. For instance, we may prove the liveness properties of requirements 4 and 5 as follows:

Let $m(r,z)$ be a (worst case) progress measure of how many lift-moves may be needed to serve a particular request z, at a point r in an execution, such that $m(r,z)$ is always non-negative, and $m(r,z) = 0$ implies that the request is served. By means of the ready-set, we may prove that the request eventually will be served as follows:

$$\begin{aligned} m(r,z) = N > 0 \Rightarrow\ & (\forall e : Event \bullet r\,\mathbf{ready}\,e \Rightarrow m(r \vdash e, z) \leq N) \wedge \\ & (\forall e : Event' \bullet r\,\mathbf{ready}\,e \Rightarrow m(r \vdash e, z) < N) \wedge \\ & (\exists e : Event' \bullet r\,\mathbf{ready}\,e) \end{aligned}$$

where $Event'$ is a subset of the outevents — assuming underlying fairness.

We shall below give a specification of the ready set and define a measure which may be used to prove requirement 5. The safety requirements stated in requirements 1 to 3 are proved by an execution invariant (OK).

6.1.1 Description of events

It seems reasonable to consider the following input events for the lift system (caused by the users):

- $bl(x, y)$ — push the *button* for floor y in *lift* x. The button light will be turned on unless already lit.

- $bf(y, d)$ — push the *button* for direction d on *floor* y, where d is one of $\uparrow$, $\downarrow$. The button light will be turned on unless already lit. A $bf(y, d)$-event will satisfy $y = lo \Rightarrow d = \uparrow$ and $y = hi \Rightarrow d = \downarrow$ where lo and hi are two integer constants identifying the low and high floor, respectively.

As observable output events for the lift system, it seems reasonable to consider all lift movements, as follows:

- $v(x, y, d, o)$ — lift x *visits* floor y and indicates (by appropriate lamps) that it will move in direction d. Here, d may have the value $\updownarrow$, which means that the lift is free (to move up or down). The doors are open (or opening) if o is true, otherwise they are closed (or closing). The doors will be open(ed) if either button $bl(x, y)$ or $bf(y, d)$ is lit, and the two buttons will be turned off (if lit).

From the given informal specification, it is not clear whether a visiting lift should indicate its planned direction. Without any indication, it is not always possible to determine if a visiting lift is an up-lift or a down-lift, and a person entering will not know whether the lift will continue in his direction. We have therefore chosen to include a parameter for the direction in v-events.

The openness of a door is clearly observable, and is therefore included in a v-event. It may be argued whether passing a floor without stopping and opening doors should be an abstractly visible event. However, such an event is observable if there are lamps, either inside lifts or outside lifts, indicating the floor position of each lift. It seems useful that each lift indicates the current floor position so that a person pushing a button requesting a stop at a certain floor knows whether that floor is passed already. The chosen event language is rich enough to fully express the abstract behavior of such lift systems.

To illustrate how sequences of events can describe lift behavior, we point out some essential ideas. A $v(x, y, d, o)$-event will only be allowed to follow another $v(x, y', d', o')$-event when y is the next floor after y' in direction d', or y equals y' and o' is true. (When d is $\updownarrow$, y must equal y'.) For instance the history

$$bf(5, \downarrow), ..., v(x, 5, \downarrow, \mathbf{t}), v(x, 5, \updownarrow, \mathbf{f})$$

indicates that the person pushing the down-button on floor 5 had left when lift x stopped. Since no request appears inside the lift, it waits with closed doors. It is here assumed that the $bf(5, \uparrow)$-button is not alight. Otherwise, the history

$$bf(5, \downarrow), ..., bf(5, \uparrow), ..., v(x, 5, \downarrow, \mathbf{t}), v(x, 5, \uparrow, \mathbf{t}), bl(x, 8), v(x, 5, \uparrow, \mathbf{f})$$

indicates that the person waiting for an up-lift was served after no one took advantage of the down-lift (according to the third additional requirement).

This seems to imply a time-out mechanism, and consequently histories, which only express linear order in time, may seem too weak. However, by assuming that there always is some minimal amount of time between v-events involving the same lift, histories suffice. Without such an assumption lift users may not have any real chance to enter or exit a lift before its door closes. The implementation will indicate where time delays are appropriate, such that the assumption is satisfied.

6.1.2 The specification

We now specify the ready set of the lift system, using a type *ESeq* encapsulating some helpful sequence observers, which will be defined further below. First the types of the events are defined:

type *Lift* $== \{1..n\}$
type *Floor* $== \{lo..hi\}$
type *Dir* $== \{\downarrow, \updownarrow, \uparrow\}$
type *Pos* $== (f : Floor \times d : Dir)$ **where** — Lift Positions
$\neg(f = lo \wedge d = \downarrow \vee f = hi \wedge d = \uparrow)$
type *BF* $== Pos$ **where** $d \neq \updownarrow$ — *bf*-event
type *BL* $== l : Lift \times f : Floor$ — *bl*-event
type *V* $== l : Lift \times f : Floor \times d : Dir \times o : Bool$ — *v*-event
type *InEvent* $== bl : BL + bf : BF$
type *Event* $== bl : BL + bf : BF + v : V$

type *ESeq* $== Seq\{Event\}$
module
func $\hat{}@\hat{}$: $ESeq \times Lift \longrightarrow V$ — the last *v*-event involving the lift
func *Rbl* : $ESeq \times BL \longrightarrow Bool$ — is there a *bl*-request?
func *Rbf* : $ESeq \times BF \longrightarrow Bool$ — is there a *bf*-request?
func *OK* : $ESeq \longrightarrow Bool$ — safety requirements
endmodule

type $LiftHistory == r : ESeq$ **where** $OK(r)$
module
func $\hat{}$ **ready** $\hat{}$: $LiftHistory \times Event \longrightarrow Bool$ — ready relation
def r **ready** $e == OK(r \vdash e)$
lma $i : InEvent,\ r : LiftHistory,\ bl : BL,\ bf : BF \bullet$
r **ready** i, — always ready to take input
$Rbf\ (r, bf) \Rightarrow \exists v : V \bullet r$ **ready** v, — external liveness
$Rbl\ (r, bl) \Rightarrow \exists v : V \bullet r$ **ready** $v \wedge v.l = bl.l$ — internal liveness
endmodule

The lemmas can be proven from the below definition of *ESeq*. The last two lemmas express that *v*-events will re-occur as long as there are outstanding requests; and if there are outstanding requests inside a lift, *v*-events for that lift will re-occur. From this one-step liveness one may prove that any request eventually will be served, by the inductive technique explained above. One may use the following progress measure for a *bl*-request:

func $m : LiftHistory \times BL \longrightarrow Nat$
def $m(r, (x, y)) ==$ **case** $r@x$ **of** $(x, f, d, o) \rightarrow$ **if** $f = y \wedge o$ **th** 0 **el**
$2 * (abs(f - y) +$ **if** $dir(f, y) = d$ **th** 0 **el**
$2 *$ **if** $d = \uparrow$ **th** $hi - f$ **el** $f - lo$ **fi fi**$) -$ **if** o **th** 0 **el** 1 **fi fi fo**

where *dir* is a function defined in type *Dir* as follows:

func $dir : Floor \times Floor \longrightarrow Dir$ — the direction from one floor to another
def $dir(y1, y2) ==$ **if** $y1 < y2$ **th** $\uparrow$ **elif** $y1 > y2$ **th** $\downarrow$ **el** $\updownarrow$ **fi**

letting **elif** ... **fi** denote **el if** ... **fi fi**. The definition of m expresses that the worst case number of moves is the distance from where the lift is, f, to where the request is, y, plus the distance from f to the top or high floor, and back, if the lift is moving away from y; this sum is multiplied by two since it may (in the worst case) take two v-events to move the lift one floor, and we subtract one if the door is closed (then the lift reaches the next floor in only one move). The measure for external requests is more difficult when there are several lifts.

We next define the functions in the type *ESeq* constructively, except the *nd*-function, which is characterized by axioms expressing minimal requirements corresponding to the informal specification:

type $ESeq \quad == Seq\,\{Event\,\}$
module
  func $\hat{}@\hat{}$: $ESeq \times Lift \longrightarrow V$ — the last v-event involving the lift
  def $r@x \quad ==$ **case** r **of** $\varepsilon \to (x, startfloor, \updownarrow, \mathbf{f}) \mid r \vdash e \to$ **case** e **of**
    $\mid v(z) \to$ **if** $x = z.l$ **th** z **el** $r@x$ **fi**
    $\mid$ **others** $\to r@x$ **fo fo**
  func $nd : ESeq \times Lift \longrightarrow Dir$ — the next direction of the lift
  func $nf : ESeq \times Lift \longrightarrow Floor$ — the next floor the lift will be at
  def $nf\,(r,x) \quad == (r@x).f +$ **if** $(r@x).o$ **th** 0
    elif $(r@x).d = \uparrow$ **th** 1 **elif** $(r@x).d = \downarrow$ **th** -1 **el** 0 **fi**
  func $Rbl : ESeq \times BL \longrightarrow Bool$
  def $Rbl\,(r,b) \quad ==$ **case** r **of** $\varepsilon \to \mathbf{f} \mid r \vdash e \to$ **case** e **of**
    $\mid v(l,f,d,o) \to b \neq (l,f)$ **and** $Rbl(r,b)$
    $\mid bl(b') \to b = b'$ **or** $Rbl(r,b)$
    $\mid$ **others** $\to Rbl(r,b)$ **fo fo**
  func $Rbf : ESeq \times BF \longrightarrow Bool$
  def $Rbf\,(r,b) \quad ==$ **case** r **of** $\varepsilon \to \mathbf{f} \mid r \vdash e \to$ **case** e **of**
    $\mid v(l,f,d,o) \to b \neq (f,d)$ **and** $Rbf\,(r,b)$
    $\mid bf\,(b') \to b = b'$ **or** $Rbf\,(r,b)$
    $\mid$ **others** $\to Rbf\,(r,b)$ **fo fo**
  func R: $ESeq \times Lift \times Floor \times Dir \longrightarrow Bool$ — any outstanding requests?
  def $R(r,x,y,d) == (Rbl(r,(x,y)) \vee Rbf(r,(y,d))) \wedge r@x \neq (x,y,d,\mathbf{t})$
  func $OK : ESeq \longrightarrow Bool$ — safety requirements
  def $OK\,(r) \quad ==$ **case** r **of** $\varepsilon \to \mathbf{t} \mid r \vdash e \to OK\,(r) \wedge$ **case** e **of**
    $\mid v(x,y,d,o) \to y = nf\,(r,x) \wedge d = nd(r,x) \wedge o = R(r,x,y,d) \wedge r@x \neq e$
    $\mid$ **others** $\to \mathbf{t}$ **fo fo**
  axm $r : ESeq,\ x : Lift,\ f,y : Floor,\ d,d' : Dir,\ o : Bool \bullet$
    $r@x = (x,f,d,o) \wedge nd(r,x) = d' \Rightarrow$
      $(Rbl(r,(x,y)) \wedge y \neq f \Rightarrow d' \neq \updownarrow \wedge (dir(f,y) = d \Rightarrow d' = d)) \wedge$
      $(d' \neq \updownarrow \Rightarrow Rbf\,(r,(f,d')) \vee \exists (y,d) : BF \bullet dir(f,y) = d' \wedge R(r,x,y,d))$,
    $(\forall x : Lift \bullet nd(r,x) = \updownarrow) \Rightarrow \forall x : Lift \bullet \neg R(r,x,y,d)$
  lma
    $OK\,(r) \Rightarrow (nf(r,x), nd(r,x)) \in Pos$
endmodule

where the constant *startfloor* defines the starting floor of the lifts. The axioms express that a non-free direction may not be changed as long as there are internal requests in this direction, that a free direction may only be chosen when there is no internal request on

other floors, that a chosen non-free direction must be useful wrt. some outstanding request, and that not all lifts may choose free when there are outstanding requests. Notice that the axioms do not require that all free lifts must move towards a *bf*-request. As stated by the lemma, *OK* ensures that no physically incorrect move may occur.

The *ESeq* module is non-constructive because *nd* is non-constructive. Internal consistency of the module is easy to show since all other functions are constructively defined, for instance it suffices to define a constructive lift strategy as follows:

type *LiftStrategy* == *ESeq*
module
  def $nd(r, x)$ == <keep a non-$\updownarrow$ direction when useful (wrt. internal or external requests),
    otherwise take a useful direction, if any, and $\updownarrow$ if none.>
endmodule

(Such a strategy will be formalized in the implementation.) As a proof obligation one must prove that the redefined *nd*-function satisfies the axioms in *ESeq*.

Observe that the functions ^@^, *Rbl*, and *Rbf* depend on the history r; the other functions depend on the history only through these. Thus in an implementation consisting of the current position and openness of each lift, *Rbl*, and *Rbf* as (array) variables, the history variable may be eliminated. Even though such an implementation gives a reasonable data structure, it is too high level in the sense that it does not produce output which is related to the hardware of a real lift system, and it does not describe how to operate the lifts in parallel.

6.2 Implementation outline

According to object oriented principles, we model the lifts as objects, each with its own activity. Each lift should have a door, internal buttons (one for each floor), and a "lift control unit" which gives signals to the lift engine and takes care of displaying the position (floor and direction) of the lift by appropriate indicators located inside and outside the lift. In addition, the lifts share a "floor control object", controlling the external up- and down-buttons.

This structure can be modeled straightforward in ABEL. We first give a sketch of the ABEL implementation: The classes and procedures are motivated by the hardware components indicated above, and the (observing) functions correspond to what is directly visible for persons using the lift system. Module prefixes are not shown in order to focus on module interfaces rather than their insides.

We next show how ABEL classes may be used to describe the hardware components, and then build an imperative implementation of the lift system on top of these classes.

class *Button* ==
module
  proc *.push* — available for users
  proc *.reset* — turns off the light
  func *.ison* : *Bool* — is the button pushed?
endmodule

class *Door* ==
module

```
  proc .open
  proc .close
  func .isopen : Bool
endmodule

class LiftCtrl ==
module
  proc .newdir(d : Dir)                    — involves displaying the new direction
  proc .move                                                  — involves moving the lift
  func .f : Floor
  func .d : Dir                                  — (.f, .d) is the indicated position,
endmodule                                        — it must always be a legal Pos.

class FloorCtrl == M : IMap{BF, Button}

class Lift == d : Door × i : LiftCtrl × B : IMap{{lo..hi}, Button}
module
  func init(startfloor : Floor) : Lift
  proc .start(var c : FloorCtrl)
endmodule
```

A definition of the type *IMap*, initialized maps, occurs as example 2 of section 3.3. We may now program the whole lift system:

```
class LiftSystem == c : FloorCtrl × L : Array{Lift}
module
  func init(n : Nat1) : LiftSystem == (init(off), Lift'init(lo)↑n)
  proc .start ==    < for each i, simultaneously do  call L[i].start(c)  >
  proc .bf (p : BF)                                  — results in a push-call on c.M[p]
  proc .bl (p : BL)                          — results in a push-call on L[p.l].B[p.f]
endmodule
```

where an *array* is a sequence of fixed length; abstractly defined as a subclass of the sequence type, but implemented efficiently by ad hoc means. The array length is given by the initializing sequence expression.

The ABEL language mechanisms for parallel computation are not yet fully decided. However, objects are naturally turned into concurrent processes by having a means to start them simultaneously. (For instance by a cobegin-construct.) Interaction is done by calling procedures and functions local to other (non-active) objects.

It is understood that the body of a procedure local to an object has an implicit critical region locking the object. In the lift example, the shared c object is passive and acts like a monitor, in the sense of Hoare. When there is more than one lift, the lift system has non-trivial aliasing to c, and global reasoning about the state of c (except what follows from representation invariants) can not be done by simple Hoare logic.

The users of the lift system could be modeled as user-objects calling the *bl*- and *bf*-procedures (resulting in appropriate *push*-calls). Such calls correspond to abstract *bl*- and *bf*-events. The protection rules prohibit objects using the lift system to interfere with its internal components. Thus, the given interface of the lift system allows natural usage and prohibits misuse.

6.2.1 Hardware specification

A very simple piece of hardware is a flip-flop, i.e. a finite state machine with two states, *on* and *off*, and with a flip operation changing the state. By extending the flip-flop with *turn_on* and *turn_off* operations, both idempotent, we obtain a more high level concept, a switch. We may see a button as a switch, letting the turn on and -off operations be software operations and letting the flip-operation correspond to hardware signals. (Alternatively, with somewhat more advanced hardware, the turn on and -off operations could correspond directly to hardware signals.) A door may also be described as a switch, letting *isopen* correspond to *ison, close* to *turn_off, open* to *turn_on*, provided its operations are taken as signals to the door mechanism.

class *FlipFlop* == {*off*, *on*}
module
  proc *x.flip* == *x* := **case** *x* **of** *on* →*off* | *off*→ *on* **fo**
  func *x.ison* : *Bool* == (*x* = *on*)
endmodule

class *Switch* == *x* : *FlipFlop*
module
  proc *.turn_off* == **if** *.ison* **th call** *.flip* **fi**
  proc *.turn_on* == **if** ¬ *.ison* **th call** *.flip* **fi**
endmodule

class *Button* == *Switch* **with** *push* **for** *turn_on, reset* **for** *turn_off*
class *Door* == *Switch* **with** *open* **for** *turn_on, close* **for** *turn_off, isopen* **for** *ison, closed* **for** *off*

The lift control may be modeled as a subclass of *Pos* (thus implementing *f* and *d*) letting *newdir* and *move* update the *f*- and *d*-components appropriately.

6.2.2 Implementation

The following structure is an expanded version of that above, defining all remaining procedures and functions:

type *Dir* == {↓, ↕, ↑}
module
  func ^ + ^ : *Dir* × *Dir* → *Dir*
  def *d*1 + *d*2 == **if** *d*1 = ↕ **th** *d*2 **el** *d*1 **fi**
  func −^ : *Dir* → *Dir*
  def −*d* == **case** *d* **of** ↕ → ↕ | ↑ → ↓ | ↓ → ↑ **fo**
endmodule

type *Pos* == (*f* : *Floor* × *d* : *Dir*) **where** ¬(*f* = *lo* ∧ *d* = ↓ ∨ *f* = *hi* ∧ *d* = ↑)
module
  func *pos* : *Floor* × *Dir* → *Pos*
  def *pos*(*nf*, *nd*) == **if** (*nf*, *nd*) ∈ *Pos* **th** (*nf*, *nd*) **el** (*nf*, ↕) **fi**
  func *nextpos* : *Pos* → *Pos*
  def *nextpos*((*f*, *d*)) == *pos*(*f* + **case** *d* **of** ↕ → 0 | ↑ → 1 | ↓ → −1 **fo**, *d*)
endmodule

class *LiftCtrl* == $i : Pos$
module
  func *startpos*$(f : Floor) : LiftCtrl == (f, \updownarrow)$
  proc *.newdir*$(nd : Dir) == i := pos(i.f, nd);$ < display >
  proc *.move* == **if** $i.d \neq \updownarrow$ **th** < move one floor > ; $i := nextpos(i);$ < display > **fi**
endmodule

class *FloorCtrl* == $M : IMap\{BF, Button\}$
module
  func *.ison*$(p : Pos) : Bool == p.d \neq \updownarrow$ **and** $M[p].ison$
  proc *.reset*$(p : Pos)$ == **if** $p.d \neq \updownarrow$ **th call** $M[p].reset$ **fi**
  func *.isanyon*$(f : Floor) : Bool ==$ *.ison*$(pos(f, \uparrow))$ **or** *.ison*$(pos(f, \downarrow))$

  func *.useful*$(p : Pos) : Bool == p.d \neq \updownarrow$ **and**
      (*.isanyon*$[nextpos(p).f]$ **or** *.useful*$(nextpos(p)))$

  func *.search*$(p : Pos) : Dir ==$ **if** *.ison*(p) **or** *.useful*(p) **th** $p.d$ **el** $\updownarrow$ **fi**

  func *.nextdir*$(p : Pos) : Dir ==$ **if** $p.d = \updownarrow$
      th *search*$(pos(p.f, \downarrow)) +$ *search*$(pos(p.f, \uparrow))$
      el *search*$(p) +$ *search*$(pos(p.f, -p.d))$ **fi**
  lma
      if $M.nextdir(p) = \updownarrow$ **th** $M = init(off)$ **el**
      $\exists p' : BF \bullet M.nextdir(p) = dir(p.f, p'.f) \wedge M[p'].ison$ **fi**
endmodule

class *Lift* == $d : Door \times i : LiftCtrl \times B : IMap\{\{lo..hi\}, Button\}$
module
  func *init*$(startfloor : Floor) : Lift == (closed, startpos(startfloor), init(off))$

  func *.useful*$(p : Pos) : Bool == p.d \neq \updownarrow$ **and**
      $(B[nextpos(p).f].ison$ **or** *.useful*$(nextpos(p)))$

  func *.search*$(d : Dir) : Dir ==$ **if** *.useful*$(pos(i.f, d))$ **th** d **el** $\updownarrow$ **fi**

  func *.nextdir*$(c : FloorCtrl) : Dir == search(i.d) + c.nextdir(i) +$
      if $i.d = \updownarrow$ **th** $search(\uparrow) + search(\downarrow)$ **el** $search(-i.d)$ **fi**

  proc *.start*(**var** $c : FloorCtrl$) == **loop**
      if *.useful*(i) **or** $c.useful(i)$
       th call $d.close$; **call** $i.move$
       el call $i.newdir(.nextdir(c))$ **fi**;
      if $B[i.f].ison$ **or** $c.ison(i)$
       th call $d.open$; **call** $B[i.f].reset$; **call** $c.reset(i)$; < *wait* > **fi endloop**
      — the waiting (some amount of time) is needed to let people enter or exit the lift
endmodule

The lemma about *nextdir* in the floor control class corresponds to the abstract minimal requirements. The *start*-procedure repeatedly moves the lift if the direction is useful wrt. internal or external requests, and otherwise searches for a new direction; in either case, the door is opened when there are relevant outstanding requests.

Notice that the ABEL protection rules do not allow assignment to the components of a class object outside the class, thus the only way to update the floor component of a lift control object is by calling the move procedure. If correctly initialized, the lift control will therefore always indicate the position where the lift actually is.

Also notice that the floor control contains terminating operations only. When it is shared by several lifts (as described in the lift system class defined further above) each call on a floor control operation must therefore terminate.

Correctness

The abstract v-events are not directly visible in the implementation. However, we may add mythical statements updating a mythical "concrete" history, from which the abstract history can be derived. The interesting concrete history is the sequence of signals to the lift hardware components, i.e. the open- and close-calls to doors, the reset-calls to buttons, as well as the push-calls caused by lift users. This history captures the lift system's interaction with the hardware components, and contains valuable information not present in the abstract history.

type $LPos \quad == l : Lift \times i : Pos$
type $Signal == push : InEvent + reset : InEvent + open : LPos + close : LPos$

The concrete history may be recorded in a mythical history variable q of type $Seq\{Signal\}$ added as a mythical component of the lift system class. For each of the mentioned calls we extend q appropriately. For instance, the call *d.close* inside class lift has the mythical effect

$$q := q \vdash close(x, i)$$

where the lift identification x must be given as a mythical parameter to the *start*-procedure, since it is not known inside the lift class. The *start* call inside class *LiftSystem* must then have the form: **call** $L[i].start(c, i)$. Similarly, the user call $bl(b)$ has the mythical effect

$$q := q \vdash push(bl(b))$$

We define the abstract history as a function A of the concrete history:

func $A : Seq\{Signal\} \longrightarrow Seq\{Event\}$
def $A(q) ==$**case** q **of** $\varepsilon \to \varepsilon \mid q' \vdash e \to$ **case** e **of**
$\mid push(e) \to A(q') \vdash e$
$\mid reset(e) \to A(q')$
$\mid open(x, (f, d)) \to A(q') + v(x, f, d, \mathbf{t})$
$\mid close(x, (f, d)) \to A(q') + v(x, f, d, \mathbf{f})$ **fo fo**

func $\hat{} + \hat{} : Seq\{Event\} \times V \longrightarrow Seq\{Event\}$
def $r + e ==$ **if** $r@(e.l) = e$ **th** r **el** $r \vdash e$ **fi**

All abstract v-operations are generated by the door-signals. The reset-signals have no abstract effect since there are no other abstract events turning off button lights.

We must prove that $A(q)$ satisfies the abstract specification. The safety part of the abstract specification is proved by partial correctness (using Hoare logic and only invariant information about shared objects) showing that the state assertion $OK(A(q))$ holds immediately after each mythical assignment to q. Thus, $OK(A(q))$ must be a loop invariant. We use

$$\forall b : BF \bullet Rbf(A(q), b) = c.M[b] \ \wedge$$
$$\forall (x, y) : BL \bullet Rbl(A(q), (x, y)) = L[x].B[y]$$

as representation invariant of the lift system.

The proof of the liveness part of the abstract specification requires total correctness reasoning: To prove that for a given lift there is a v-event in the ready set, when its B is not all off, it suffices to prove that each of its mythical q-assignments will lead to another which will extend $A(q)$ (assuming B is not all off). Since it is obvious that no lift can lock the floor control, we may assume that it never is deadlocked.

The input events correspond to *push*-calls on the buttons. Such a call can always be performed expediently, since all button-operations are quick. In particular, no lift object may cause a button to deadlock. Thus, the system is always ready to accept input-events.

Acknowledgements

Stein Krogdahl and Anne B. Salvesen have given valuable comments through careful reading of an earlier version.

References

[1] O.-J. Dahl: " Can Program Proving be Made Practical?" In *Les Foundements de la Programmation*, M. Amirchahy and D. Néel, Ed., INRIA, 1977

[2] O.-J. Dahl: "Object Oriented Specification." In *Research Directions in Object-Oriented Programming*, B. Shriver and P. Wegner, Ed., MIT Press, 1987.

[3] O.-J. Dahl: *Verifiable Programming.* To appear in The Hoare Series, Prentice Hall.

[4] O.-J. Dahl, D.F. Langmyhr, O. Owe: "Preliminary Report on the Specification and Programming Language ABEL." Research Report 106, Dept. of Informatics, University of Oslo, 1986.

[5] K. Futasugi, J.A. Goguen, J.-P. Jouannaud, J. Meseguer: "Principles of OBJ2." In *Proceedings, 1985 Symposium on Principles of Programming Languages and Programming*, Association for Computing Machinery, 1985, pp. 52-66. W. Brauer, Ed., Springer-Verlag, 1985. Lecture Notes in Computer Science, Volume 194.

[6] J.V. Guttag: "The Specification and Application to Programming of Abstract Data Types." Ph. D. Thesis, Computer Science Department, University of Toronto, 1975.

[7] J.V. Guttag, J.J. Horning, J.M. Wing: "Larch in Five Easy Pieces." Digital Systems Research Center, Palo Alto, California, July 1985.

[8] C.A.R. Hoare: *Communicating Sequential Processes.* The Hoare Series, Prentice Hall, 1985.

[9] S.C. Kleene: *Introduction to Metamathematics.* North-Holland, 1952.

[10] O. Lysne, O. Owe: "Definedness and Strictness in Generator Inductive Definitions." Research Report 161, Dept. of Informatics, University of Oslo, 1991.

[11] O. Owe, O.-J. Dahl: "Generator Induction in Order Sorted Algebras." *Formal Aspects of Computing*, 3:2-20, 1991

[12] O. Owe: "Partial Logics Reconsidered: A Conservative Approach." Research Report 155, Dept. of Informatics, University of Oslo, 1991.

[13] D. Prawitz: *Natural Deduction.* Almquist & Wiksell, Stockholm, 1965.

[14] N. Soundararajan: Personal communication.