Value Types and Object Classes

Ole-Johan Dahl
Dept. of Informatics, University of Oslo

Talk presented at the 25 years Simula 67 Anniversary Conferences, Nordwijkerhout 11-12 June 1992 and Oslo 22 June.

1 Introduction

Simula 67 was presented for the first time at the IFIP Working Conference on Simulation Languages, Oslo, May 1967, [1]. At a subsequent private conference the Simula 67 Common Base was defined. After the addition of mechanisms for text processing and input/output, and access protection mechanisms a few years later, the language has remained remarkably stable.

At the Common Base Conference Kristen Nygaard and I proposed to enrich the language with class-like types. The proposal was turned down by the participants responsible for implementations. Today it would seem a better idea to build a programming language around a good general type concept, and subsequently add type-like classes. In the following we indicate how a generalized subset of Simula 67 could be defined in this way.

2 Data types by generator functions

The idea is to define the value set associated with a type implicitly, by listing a set of functions, the *generators*, in terms of which all such values can be expressed.

This basic idea occurs in some form in several languages for abstract specification as well as programming, notably the LARCH Shared Language (J. Guttag et al, [5]), OBJ versions (J. Goguen et al, [4]) and dialects of ML (R. Milner [7], G.Huet and others [6]). The version shown here is drawn from of a language ABEL (Abstraction Building Experimental Language) for specification and programming, developed at Oslo University, mainly by O. Owe and the author, as part of a research effort in program specification and verification (see e.g. [2]).

It is felt that a good formal language should be strongly typed; in particular, it is useful to characterize functions syntactically by *profiles* specifying their domain and codomain:

$$\mathbf{func}\ f:\ T_1 \times T_2 \times \ldots \times T_n \longrightarrow T$$

The *generator basis* for a type T is a list of functions with the codomain T, the chosen T-generators.

$$\mathbf{genbas}\ g_1, g_2, \ldots g_m$$

If types other than T occur in the generator domains, they should be previously defined. They are called *underlying types.* The set of variable-free type T expressions in generators, including those of underlying types, is called the *generator universe* of T, GU_T. Its elements, called the *basic T-expressions*, by definition express all values of the type T. (For GU_T to be non-empty, at least one generator must have no occurrence of T in its domain.)

Examples

Bool:	$\mathbf{func}\ \mathbf{f}, \mathbf{t} : \longrightarrow \mathit{Bool}$	$\mathbf{1\text{-}1\ genbas}\ \mathbf{f}, \mathbf{t}$
giving	$GU_{\mathit{Bool}} = \{\mathbf{f}, \mathbf{t}\}$	
Nat:	$\mathbf{func}\ 0 : \longrightarrow \mathit{Nat},\ \mathbf{S}\hat{}\ : \mathit{Nat} \longrightarrow \mathit{Nat}$	$\mathbf{1\text{-}1\ genbas}\ 0, \mathbf{S}\hat{}$
giving	$GU_{\mathit{Nat}} = \{0,\ \mathbf{S}0,\ \mathbf{SS}0,\ \mathbf{SSS}0,\ \dots\}$	
Int:	$\mathbf{func}\ 0\ \longrightarrow \mathit{Int},\ \mathbf{S}\hat{}, -\hat{}\ : \mathit{Int} \longrightarrow \mathit{Int}$	$\mathbf{genbas}\ 0, \mathbf{S}\hat{}, -\hat{}$
giving	$GU_{\mathit{Int}} = \{0,\ \mathbf{S}0,\ -0,\ \mathbf{SS}0,\ \mathbf{S}-0,\ -\mathbf{S}0,\ --0,\ \mathbf{SSS}0,\ \dots\}$	
Set$\{T\}$:	$\mathbf{func}\ \emptyset : \longrightarrow \mathit{Set},\ \mathit{add} : \mathit{Set} \times T \longrightarrow \mathit{Set}$	$\mathbf{genbas}\ \emptyset, \mathit{add}$
giving	$GU_{\mathit{Set}\{T\}} = \{\emptyset,\ \mathit{add}(\emptyset, a),\ \mathit{add}(\emptyset, b),\ \dots,$ $\mathit{add}(\mathit{add}(\emptyset, a), a),\ \mathit{add}(\mathit{add}(\emptyset, a), b),\ \dots,$ $\mathit{add}(\mathit{add}(\emptyset, b), a),\ \mathit{add}(\mathit{add}(\emptyset, b), b),\ \dots,\ \dots\}$	

For the types *Bool* and *Nat* the basic expressions are in a one-to-one correspondence with the intended "abstract" values, when the generator $\mathbf{S}\hat{}$ is interpreted as the successor function for natural numbers.[1] For that reason the corresponding generator bases are specified as being "**1-1**". The one-to-one property of a generator basis implies that equality over that type reduces to syntactic equality of the basic expressions, up to equality over underlying types.

The generator basis given for the type *Int* of integers does not have the one-to-one property. For instance 0 and -0 are both intended to represent the value zero, and $--e$ is intended to be equal to e and to $-\mathbf{S}-\mathbf{S}e$, etc., for any basic *Int* expression e. In this case the intended abstract values correspond to *sets* of basic expressions, rather than to individual ones. There are several ways of defining these subsets of the generator universe, one is to give an explicit definition of the equality relation for the type in question, see e.g. [3]. On the other hand, logic requires equality to have the property that "equal" expressions can substitute for one another. This results in certain mathematical difficulties in the treatment of types with many-to-one generator bases. Unfortunately, in the case of integers there does not seem to be any simple and natural way of providing a one-to-one generator basis.

Also the generator basis specified for the type of finite sets of T-values is many-to-one, when the *add* function is supposed to extend a given set by another element, if it is not already in the set. Thus the expressions $\mathit{add}(\emptyset, a)$ and $\mathit{add}(\mathit{add}(\emptyset, a), a)$ are both intended to represent the singleton set $\{a\}$. Also the expressions $\mathit{add}(\mathit{add}(\emptyset, a), b)$ and $\mathit{add}(\mathit{add}(\emptyset, b), a)$ are intended to be semantically equal. Unfortunately there exists no one-to-one generator basis at all for the concept *Set*$\{T\}$ (for unspecified underlying type T).

It turns out that the concept of *subtypes* can sometimes be used to provide one-to-one generator bases in a practical way.

3 Subtypes

We may define the type *Int* of integers as the union of three disjoint *syntactic subtypes*: *Zero* with the single value 0, *Nat*1 corresponding to the strictly positive integers, and *Neg*1 representing the set of strictly negative ones. Let *Nat* correspond to the non-negative

[1]The symbol $\hat{}$ is used to specify an argument position in a "mixfix" notation, where the operator symbols are sequences of special characters like $+$ and $=$, as well as boldface text.

integers, $\mathit{Nat} = \mathit{Zero} \cup \mathit{Nat}1$. Then a one-to-one basis may be defined by restricting the domain of the successor function $\mathbf{S}\hat{}$ to *Nat*, and using a negation function $\mathbf{N}\hat{}$ with the domain *Nat*1. The codomain of each generator then will be one of the three "basic" subtypes.

type *Int* **by** *Zero*, *Nat*1, *Neg*1 **with** $\mathit{Nat} = \mathit{Zero} \cup \mathit{Nat}1$:
func $0 : \longrightarrow \mathit{Zero}$, $\mathbf{S}\hat{} : \mathit{Nat} \longrightarrow \mathit{Nat}1$, $\mathbf{N}\hat{} : \mathit{Nat}1 \longrightarrow \mathit{Neg}1$
1-1 genbas 0, $\mathbf{S}\hat{}$, $\mathbf{N}\hat{}$
giving $GU_{\mathit{Int}} = \{0;\ \mathbf{S}0,\ \mathbf{SS}0,\ \mathbf{SSS}0,\ \ldots;\ \ \mathbf{NS}0,\ \mathbf{NSS}0,\ \mathbf{NSSS}0,\ \ldots\}$

The generator universe consists of those basic expressions which are consistent with strong typing. They are clearly in a one-to-one correspondence with the integers. In addition to *Nat*, another two "intermediate" syntactic subtypes of *Int* may be defined: $\mathit{Neg} = \mathit{Zero} \cup \mathit{Neg}1$ and $\mathit{Nzo} = \mathit{Neg}1 \cup \mathit{Nat}1$. They are useful for characterizing functions partial on *Int*, but total on a restricted domain. For instance, the division operator $\hat{}/\hat{}$ is total on the domain $\mathit{Int} \times \mathit{Nzo}$.

Whereas syntactic subtypes, together with their main type, form a hierarchy of types defined simultaneously by a single generator basis, *semantic* subtypes are formed by explicitly restricting given types. For instance, the set of integers implemented on a given computer might be the semantic subtype

$$\textbf{type } \mathit{ImpInt} == x : \mathit{Int} \textbf{ where } -32768 \le x \le 32767$$

It is possible to simulate an one-to-one generator basis for $\mathit{Set}\{T\}$ in a semantic subtype consisting of unique "canonic forms", e.g. basic expressions of the form

$$\mathit{add}(\ldots \mathit{add}(\mathit{add}(\emptyset, t_1), t_2), \ldots, t_n), \quad n \ge 0,$$

where $t_1 < t_2 < \ldots < t_n$, assuming that $\hat{} < \hat{}$ is a total order on the otherwise unspecified type T. In the subtype the generator *add* will be replaced by a defined function generating canonic forms. Since all set values are then canonic, equality on sets is reduced to syntactic equality (up to T-equality) on the representations.

4 Function definition by generator induction

The subterm relation provides a well-founded partial order on any generator universe: e is "less" than $g(\ldots, e, \ldots)$, for basic T-expressions e and $g(\ldots, e, \ldots)$. This order gives rise to an induction principle, so called *generator induction*, used to cover the entire generator universe. This principle is useful for the purpose of defining functions, as well as for proving theorems.

In a function definition

$$\textbf{def } f(x_1, x_2, \ldots, x_n) == \text{expression in } x_1, x_2, \ldots, x_n$$

generator induction with respect to an argument of type T may be expressed by a "**case**-construct" branching on the value of that argument, with one branch for each T-generator. If all the generators are constants, the construct is analogous to that of Pascal; for instance, branching on the value of an argument of type *Bool* gives one branch for the value **f** and one for **t**. A value of type *Nat*, defined as above, must either be 0 or it must be of the form $\mathbf{S}e$ where e is a *Nat* value. A corresponding **case**-construct must be a pattern matching mechanism. The "discriminator" heading the second branch must introduce a new variable as the argument of $\mathbf{S}\hat{\ }$.

An alternative to a function definition with a **case**-construct in the right hand side, branching on a left hand argument, is to give one defining equation for each branch with the discriminator thrown back into the left hand side. Such equations are referred to as **case**-free (inductive) axioms.

case-constructs may be nested to any depth, where inner ones may branch on variables introduced in outer discriminators. (Branching on expressions other than variables corresponds to equational axioms guarded by explicit conditions.)

Examples

$$
\begin{array}{ll}
\textbf{func} & \hat{\ }\wedge\hat{\ } : \mathit{Bool} \times \mathit{Bool} \longrightarrow \mathit{Bool} \\
\textbf{def} & x \wedge y == \textbf{case}\ y\ \textbf{of}\ \mathbf{f} \to \mathbf{f} \mid \mathbf{t} \to x\ \textbf{fo} \\
\text{or:} & x \wedge \mathbf{f} == \mathbf{f}, \quad x \wedge \mathbf{t} == x
\end{array}
$$

$$
\begin{array}{ll}
\textbf{func} & \hat{\ }+\hat{\ } : \mathit{Nat} \times \mathit{Nat} \longrightarrow \mathit{Nat} \\
\textbf{def} & x+y == \textbf{case}\ y\ \textbf{of}\ 0 \to x \mid \mathbf{S}y' \to \mathbf{S}(x+y')\ \textbf{fo} \\
\text{or:} & \text{A1: } x+0 == x, \quad \text{A2: } x+\mathbf{S}y == \mathbf{S}(x+y)
\end{array}
$$

$$
\begin{array}{ll}
\textbf{func} & \hat{\ }<\hat{\ } : \mathit{Nat} \times \mathit{Nat} \longrightarrow \mathit{Bool} \\
\textbf{def} & x<y == \textbf{case}\ y\ \textbf{of}\ 0 \to \mathbf{f} \mid \mathbf{S}y' \to \\
 & \qquad\qquad \textbf{case}\ x\ \textbf{of}\ 0 \to \mathbf{t} \mid \mathbf{S}x' \to x'<y'\ \textbf{fo}\ \textbf{fo} \\
\text{or:} & \text{L1: } x<0==\mathbf{f}, \quad \text{L2: } 0<\mathbf{S}y==\mathbf{t}, \quad \text{L3: } \mathbf{S}x<\mathbf{S}y==x<y
\end{array}
$$

Notice that, using generator induction and recursion when necessary, we are able to define these functions on the basis of generators alone. It is easy to see that the recursion in the axioms A2 and L3 must terminate, for in either case there is an argument which is textually simpler in the recursive application than in the left hand side. Such recursion is said to be *guarded by induction.*

5 Term Rewriting

The **case**-free axioms of a function definition may be taken as rewrite rules for term rewriting, read from left to right, which may be applied to arbitrary well-defined expressions without changing their meaning. Provided that any recursion is guarded by induction, these rules form a so called *convergent* system, which means that the rewriting process must necessarily terminate with a unique final result independent of the rewriting order.

If the given initial expression is variable-free, the final result is a basic expression, i.e. *the value* of the given one:

$$
\begin{array}{l}
\mathbf{SS}0+\mathbf{SS}0 \xrightarrow{\text{A2}} \mathbf{S}(\mathbf{SS}0+\mathbf{S}0) \xrightarrow{\text{A2}} \mathbf{SS}(\mathbf{SS}0+0) \xrightarrow{\text{A1}} \mathbf{SSSS}0 \\
\mathbf{SS}0<\mathbf{SSS}0 \xrightarrow{\text{L3}} \mathbf{S}0<\mathbf{SS}0 \xrightarrow{\text{L3}} 0<\mathbf{S}0 \xrightarrow{\text{L2}} \mathbf{t}
\end{array}
$$

If the given expression contains variables the rewriting process will often amount to a *simplification*:

$$\mathbf{S}x<y+\mathbf{S}0 \xrightarrow{\text{A2}} \mathbf{S}x<\mathbf{S}(y+0) \xrightarrow{\text{A1}} \mathbf{S}x<\mathbf{S}y \xrightarrow{\text{L3}} x<y$$

If a Boolean expression can be rewritten to $\mathbf{t}$, it is thereby proved as a theorem. It is sometimes possible to strengthen a convergent rewriting system, by including well chosen lemmas as additional rewrite rules, without losing convergence. In general, however, term rewriting is not sufficient for proof purposes; one needs the additional power provided by an induction principle.

Generator induction in particular combines very nicely with generator inductive definition techniques. The structure of a generator inductive proof is determined by the relevant generator basis: there must be one separate proof for each generator, in which the theorem is modified by replacing the induction variable by the generator applied to fresh variables. Induction hypotheses are allowed for those generator arguments which are of the same type as the induction variable (or of a subtype). In typical cases most of the work can be done by term rewriting based on (inductive) function definitions.

Example

> The expression $x < \mathbf{S}x$ is irreducible by the inductive axioms above. In order to prove that it is true for all *Nat*-values x we may use generator induction on x. Given a generator basis for *Nat* consisting of 0 and $\mathbf{S}\hat{}$, the structure of the proof is similar to that of ordinary mathematical induction:
>
> $\underline{0:}$ $\quad 0 < \mathbf{S}0 \xrightarrow{\text{L2}} \mathbf{t}$
>
> $\underline{\mathbf{S}y:}$ $\quad$ Assuming IH: $y < \mathbf{S}y$, prove $\mathbf{S}y < \mathbf{SS}y$.
>
> $\quad \mathbf{S}y < \mathbf{SS}y \xrightarrow{\text{L3}} y < \mathbf{S}y \xrightarrow{\text{IH}} \mathbf{t}$

In summary, generator inductive definition techniques provide a very general framework for concept modelling. Term rewriting based on such definitions, combined with generator inductive proof techniques, provide powerful and efficient means of semi-mechanical reasoning about properties of functions and of programs using them.

6 Type modules

The ideas of the last sections may be seen as techniques of abstract specification, but they can also be taken as an *applicative programming language*. Once a representation of expressions has been chosen, convenient for internal machine manipulation, say list structures, a language interpreter could essentially take the form of a term rewriting algorithm. Another possibility is to view function definitions as algorithms working on values which are basic expressions.

We present a **type module** construct, in some ways comparable to Simula classes. Values of the type defined correspond to class objects, and the functions introduced in the module (generators as well as defined functions) correspond to procedure attributes. They might even be made to look "local to" the individual values if some special syntax, say dot notation, is used to identify a distinguished argument of the type in question (if there is such an argument).

It is convenient also to include inductively defined types and functions into an imperative language permitting assignable program variables of such types. The assignment operations should have the usual meaning of value copying.

Example

> **type** $\mathit{Stack}\{T\}$ **by** Estack, $\mathit{Stack}1$ ==
> **module**
> **func** $\mathit{nil} : \longrightarrow \mathit{Estack}$, $\quad \mathit{push} : \mathit{Stack} \times T \longrightarrow \mathit{Stack}1$
> **1-1 genbas** *nil*, *push*

func $top : Stack1 \longrightarrow T$ **def** $top(push(S,x)) == x$
func $pop : Stack1 \longrightarrow Stack$ **def** $pop(push(S,x)) == S$
endmodule

var $S : Stack\{Int\} = push(push(nil,3),5)$;
var $T := S$; $T := push(T,17)$; $T := pop(T)$; $T := pop(T)$;

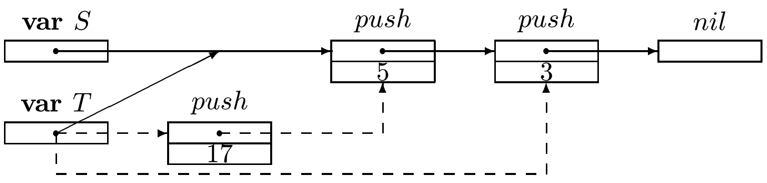

Figure 1: Pushing and popping stacks.

Figure 1 shows the data configuration after variable initialization and after the second assignment (unbroken lines), respectively after the first and third assignments (dashed lines). The assignments to T do not change the value of S, although these variables share data structure. Thus, implementing value assignments and parameter transmission by pointer copying does not harm the program logic. This is true as long as values are represented by *immutable* data structures. Notice that the list structures directly represent basic $Stack\{Int\}$ expressions (but with an ad-hoc representation of integers, rather than a list structure representing **SSS**0 for 3, and so forth).

It should be noticed that the functions *pop* and *top* are total functions on the domain of non-empty stacks, but are undefined for empty stacks. This is the reason why the syntactic subtypes *Estack* and *Stack*1 have been introduced. (*Stack*1 has only one generator, therefore one **case**-free inductive axiom is sufficient for either function.) Whereas values of a subtype are also values of its supertype, the reverse is not generally true. However, it may be practical to include a mechanism for "coercion" to subtypes (as for reference types in Simula). For instance, behind the scenes application of the function

func $StackToStack1 : Stack \longrightarrow Stack1$
def $StackToStack1(S) ==$ **case** S **of** $nil \rightarrow$ **error** | **others** $\rightarrow S$ **fo**

would make it legal to apply *pop* and *top* to values of type *Stack*, i.e. to stacks not syntactically identifiable as being non-empty.

Unfortunately the requirement that values be immutable in general leads to gross inefficiency when manipulating high volume data, having to recreate the whole thing in order to make a small local change. In fact, the efficiency of the operations of pushing and popping stacks is atypical. A more normal situation occurs in the following example of adding new elements at the far end of a stack.

```
func  $enq : Stack \times T \longrightarrow Stack1$
def   $enq(S,x) ==$ case $S$ of $nil \rightarrow push(S,x)$
                                  | $push(S',y) \rightarrow push(enq(S',x),y)$ fo
```

var $S : Stack\{Int\} = push(push(nil,3),5)$;
var $T := S$; $T := enq(T,17)$;

Notice that each *push* operation, invoked recursively in *enq*, must generate a new record in the list structure. As a result the whole stack is copied.

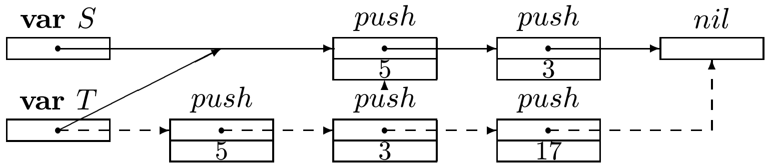

Figure 2: Enqueuing.

To summarize, values are timeless concepts, like the number 3, and must be represented in a computer by immutable data structures. Values of high volume, such as large arrays or voluminous list structures, therefore must be updated by (more or less) complete recreation, which may entail much inefficiency. On the other hand, whereas assignment and parameter transmission of values conceptually are by copying, they may be realized efficiently through pointer copying at the discretion of a compiler without harming the program logic.

7 Object classes

Efficiency considerations show that there is a need for locally updatable data structures accessed through pointers. By introducing a local update mechanism for structures defined inductively as above, we obtain a kind of simplified and generalized Simula objects. We choose here, however, to let the pointers remain implicit in order to emphasize the similarity between values and objects. Thus, we may define a class of objects in the same way as a value type, simply by replacing the keyword **type** by the keyword **class**.

A local update of an object is achieved by updating a component of the object. Thus, for an object variable X the component names introduced in **case** discriminators for X should count as assignable program variables:

$$\textbf{case}\ X\ \textbf{of}\ \ldots \mid g(\ldots, y, \ldots) \to y := \ldots \mid \ldots\ \textbf{fo}$$

In order to protect the integrity of class objects it is reasonable to allow local updates only textually inside the class in question (and in subclasses).

class modules may have the same kind of contents as **type** modules, but they may in addition contain *procedure declarations* with imperative bodies and assignable variable parameters. Thus, in order to define a class of "stack" objects with a more efficient mechanism for entering elements at the far end, we may make a copy of the module part of figure 1 and add a procedure for the "enqueuing" of elements. However, rather than copying all that program text we may introduce a Simula inspired prefixing mechanism and define a "subclass" of the stack type of figure 1.

```
class $STACK\{T\}$ by $ESTACK$, $STACK1$ == $Stack\{T\}$
module
proc $Enq($var $S : STACK$, val $x : T)$ ==
case $S$ of $nil \to S := push(S, x) \mid push(S', y) \to$ call $Enq(S', x)$ fo
endmodule

var $S : STACK\{Int\} = push(push(nil, 3), 5)$;
var $T := S$; call $Enq(T, 17)$;
```

It is still necessary to search for the far end of the stack, but at least we do not have to copy any part of the old stack in order to insert the new element. On the other hand, updating

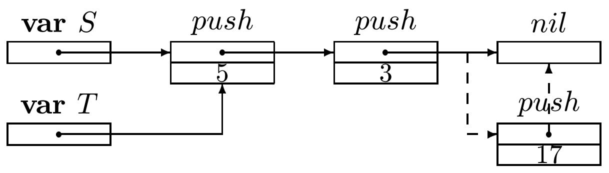

Figure 3: More efficient enqueuing.

inside the stack object has made the program logic suffer: changing T has a side effect on S. There is nothing in the statement **call** $Enq(T, 17)$ to indicate that the object called S should be modified, but it is! The reason is that the preceding statement has made S and T become *aliases.*

In general, simultaneously accessible converging pointers in the run time data structures represent aliases and provide a danger of side effects to component assignments. In the example the alias problem was easy to identify, but in general it may be *very* difficult, especially when reasoning about incorrect programs, as we have to do some of the time.

Fortunately there exist styles of programming which will guarantee the absence of aliasing, and which may be enforced by syntactic checks on the program text. For instance, the following requirement on object assignments is sufficient:

- The right hand side must contain no object variable outside value subterms, except at most a single occurrence of the left hand variable or a component of that object. (In this context variable subscripts must be ignored, so that for instance $A[i]$ and $A[j]$, where A is an object array, are regarded as the same variable, which they are if $i{=}j$.)

This restriction is too hard to live with, but it may be relaxed in several ways. One important idea is to introduce multiple assignments in the language. Each object variable of the left hand list (or a component of that object) is then allowed to occur once in the right hand list. So, for instance the statement $X, Y := Y, X$ would be allowed. Obviously, it cannot cause aliasing.

Alias-free programming with object classes is a way to combine different important considerations which are otherwise in conflict:

- simplicity of the program logic, in particular with respect to assignment operations, and consequent ease of reasoning about programs,
- the flexibility and computational efficiency that may be obtained using updatable data structures accessed and related through assignable pointers.

8 Simula-like classes

The class/object mechanisms as exemplified so far may look more like a list processing facility than object orientation in the sense of Simula. In particular, objects in Simula have local variable attributes. We now show how a little special purpose syntax may close that gap. We first introduce the concept of "labelled Cartesian products". A type definition of the form

$$\textbf{type}\ T == \{a_1{:}T_1 \times a_2{:}T_2 \times \ldots \times a_n{:}T_n\}$$

is shorthand for:

```
type $T$ ==
module
func $(\hat{~},\hat{~},\ldots,\hat{~}) : T_1 \times T_2 \times \ldots \times T_n \longrightarrow T$          ($n$-tuple)
1-1 genbas $(\hat{~},\hat{~},\ldots,\hat{~})$
func $\hat{~}.a_1 : T \longrightarrow T_1,\ \ldots,\ \hat{~}.a_n : T \longrightarrow T_n$          (component selectors)
def $(x_1, x_2, \ldots, x_n).a_1 == x_1$
– – – – – – – –
def $(x_1, x_2, \ldots, x_n).a_n == x_n$
endmodule
```

So, if t is a value of type $\{a_1{:}T_1 \times a_2{:}T_2 \times \ldots \times a_n{:}T_n\}$, the expression $t.a_k$, where k is an integer in the range $1..n$, evaluates to the k'th component of t.

Now (more) Simula-like classes, say a class C, may be defined as a subclass of a labelled Cartesian product, where the labels are made to play the roles of assignable local variables within the module body, as follows. Any function or procedure whose name is of the form *.name* has a distinguished class C parameter in front of the dot. Each occurrence of *.name* within the class module is interpreted as $(a_1, a_2, \ldots, a_n).name$ (in particular any left hand occurrence). For a procedure the distinguished formal parameter is a **var** parameter, which implies that its components a_k are assignable variables.

Let X be an object of class C. It follows from the above trickery that any invocation $X.p(\ldots)$ of a procedure or function $.p$, whose body contains occurrences of $a_1, a_2, \ldots, a_n$, will access the components of X. Notice that $X.a_k$ is an application of a selector function, not an updatable variable. It follows that any invariant established initially and is maintained by the C-procedures is guaranteed to hold for all C-objects. The invariant may be expressed as the restricting predicate of a *semantic subtype* of a labelled Cartesian product, occurring as the prefix of the C-module.

No predefined "reference-to-nothing" is needed in the language, provided that explicit initialization of (object) variable declarations is possible, and preferably mandatory. On the other hand, any generator basis must contain at least one relative constant, i.e. a function with no occurrence in its domain of the type(s) being defined, in order to terminate data recursion. For instance, the *Stack* value *nil* plays the role of recursion terminator. A Cartesian product must be non-recursive, thus, the tuple generator is already a relative constant.

9 Aliasing for efficiency

It happens that data structures with converging pointers are necessary in order to obtain good efficiency. For instance, in order to program the enqueuing and popping operations of a FIFO list in constant time, pointers to both ends of the list are required. Consequently there will be pointer convergence at the "far" end of the list and a corresponding alias problem. We show an efficient version of a FIFO queue based on the *STACK* class of section 7, as a Simula-like class.

```
class $FIFOQ\{T\} == \{front{:}STACK\{T\} \times rear{:}STACK\{T\}\}$
                                                   where $last(front) = rear$
module
func $empq : \longrightarrow FIFOQ$                         def $empq == (nil, nil)$
proc $.Enq($val $x) ==$ var $new : STACK = push(nil, x);$
```

```
                                   case $rear$ of $nil \to front, rear := new, new$
                                       | $push(S,y) \to S, rear := new, new$ fo
proc $.Pop$ == case $front$ of $nil \to$ abort
                     | $push(S,x) \to front := S$; if $front{=}nil$ th $rear := nil$ fi fo
endmodule
```

var $Q\!:\!FIFOQ\{Int\}{=}empq$; **call** $Q.Enq(5)$; **call** $Q.Enq(3)$:

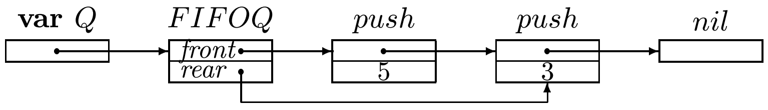

Figure 4: Efficient FIFO queue.

It is assumed that the *STACK* module has been extended by the function *last* in order that its occurrence in the object invariant be meaningful:

```
func $last : STACK \longrightarrow STACK$
def  $last(S)$ == case $S$ of $nil \to S \mid push(S',x) \to$
                          case $S'$ of $nil \to S \mid$ others $\to last(S')$ fo
```

Reasoning about assignments in the context of pointer aliases is not at all trivial. For instance, it must be proved that the occurring pointer assignments do not create pointer circles, in which case the *last* function might be undefined due to non-terminating recursion, and that would make the object invariant undefined as well. (Actually, the invariant has to be strengthened in order to be provable.) The "correctness" of *FIFOQ* may be defined and proved by comparison with a corresponding *type* (an extended *Stack* as in section 6 would do) expressing the purpose of *FIFOQ* in a way suitable for easy reasoning while disregarding computational efficiency.

Fortunately, having verified the correctness of *FIFOQ* the users will not be troubled by the internal aliasing; in fact, he may reason as if the *FIFOQ* operations were replaced by corresponding *Stack* operations. For instance, a statement **call** $Q.Enq(x)$ would correspond to $Q := enq(Q,x)$, interpreting Q as a variable of type *Stack* rather than *FIFOQ*.

10 Conclusions

Elements of a language for programming and "abstract" specification has been explained and exemplified. The language mechanisms are drawn from a language ABEL developed at Oslo University. The techniques of generator inductive type and function definition provides an easy to use style of applicative programming, and at the same time lends itself to efficient reasoning aids based on term rewriting and generator induction. Important aspects of object orientation may be included in the applicative part of the language: natural grouping of functions into modules, associated rules of function overloading (not discussed here), and subtyping with inheritance.

For the purposes of efficient usage of computer resources, as well as interactions with users and other programs, imperative facilities are often useful. In addition to standard mechanisms such as assignment and explicit sequencing of operations, an object class concept is introduced analogous to value type modules. The difference is a mechanism for internal updating of objects and mandatory use of (implicit) pointers. This in general may give rise to pointer aliases which to a large extent complicates the reasoning about assignment operations. There exist, however, restricted programming styles, syntactically enforceable, which guarantee the absence of pointer aliases, and which make it possible to combine easy reasoning with efficiency up to a point.

A general advice to programmers, borne out by experience, is to put as much of the program logic as is feasible into the applicative part, i.e. into types and functions. It is important to keep the applicative part "clean", avoiding functions with side effects, in order to enhance perspicuity and easy reasoning.

11 Acknowledgements

Bjørn Kristoffersen and Olaf Owe have contributed to the readability of this paper by providing non-trivial feedback.

References

[1] J. Buxton, ed.: *Simulation Programming Languages*, North Holland, 1967.

[2] O. Owe, O.-J. Dahl: Formal Development with ABEL. VDM'91, *Springer Lecture Notes 552*, pp. 320-362. Revised version: Research Report 159, Dept. of Informatics, University of Oslo.

[3] O.-J. Dahl: *Verifiable Programming.* Prentice Hall, 1992.

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

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

[6] G. Huet, ed.: *Logical Foundations of Functional Programming*, Addison Wesley 1990.

[7] A.J.R.G. Milner: A Proposal for Standard ML. *Proceedings of the ACM Symposium on LISP and Functional Programming*, Austin, 1984.

Contents