A Proposal for Integrating Persistence into the Prototyping Language SETL/E

Ernst-Erich Doberkat
Informatik/Software Engineering
Universität - Gesamthochschule - Essen

April 18, 1990

Abstract

SETL/E is a prototyping language based on SETL having sets, maps, tuples, and procedures as basic data types. We propose introducing a mechanism for making data persistent into SETL/E, thus creating the possibility of working with data that outlive the execution of programs which created them. This makes SETL/E resemble a data base programming language. Since procedures are first class objects in SETL/E, this applies in particular to modules as collections of procedures, hence we propose a mechanism for separate compilation for the language. The corresponding linguistic mechanisms are discussed.

Contents

1 Introduction

The classical model of software production using the *life cycle* approach has severe deficiencies indicating the desirability of complementing this model by other approaches. One of the more recent approaches for this is *rapid prototyping.* Having a look at the literature it seems that this term is used as an umbrella notion for a multitude of activities, and it is not always too easy to find some sort of common denominator, see [18, 7, 13]. We stick here to Christine Floyd's definition given in [16], according to which prototyping refers to the welldefined phase in the production process of software in which a model is constructed which has all the essential properties of the final product, and which is taken into account when properties have to be checked, and when further steps in the development have to be determined.

This does not only apply to programs, but also to data: in the process of developing an application not only the algorithms have to be explored, but the data and data structures on which the algorithms are to work may emerge from this explorative activity as well. Semantic data models working with objects, attributes, and *ISA*-relationships investigate ways of modelling data according to their semantic contents (cp. [19]). They are used for designing record-oriented schemata where the approach is somewhat similar to the one used in software prototyping, but rather than modelling programs high-level representations of data are modelled. This model is mapped into a lower-level structure (see [19], 1.4). Khoshafian and Briggs point out that data modelling should accomodate the user by making the representation and manipulation as close as possible to the user's perception of the problem (cf. [21], p. 606). Hence it is desirable to

- model data according to the user's needs,
- iteratively refine data representations (which requires access to previously formulated data models),
- reuse patterns or templates of previously formulated data models,
- share data either between different users and different prototyping sessions.

We see that there are in fact striking similarities between prototyping programs and modelling data. Both construct a model to be experimented with and eventually to be transformed into a production version. Thus it would be valuable to have a programming language which is able to serve both sides,

- the software engineer who wants to model programs
- the data engineer who wants to construct a semantic model of her data.

Persistence is interesting when considered in the context of software prototyping. Since prototyping combined with support for a semantic data model allows formulating data on a very high-level for modelling purposes, it is simply a matter of economy to make data persistent: once data are modelled it is not necessary to compute them each time they are used. Hence reusing data in a program does not necessarily mean recomputing them. A related concern for reusing data comes from the observation that more than one program may want to access them. Thus one program may generate data and another one may want to access these data. Consequently one may have to face a situation where programs communicate through persistent data. We do not address the problem of concurrent access to these data in this paper, but the reader should have in mind that such a communication is possible and bears specific problems.

Once the protoype becomes stable, it may be transformed into a production program, see e.g. [14]. The data which have been modelled using the prototype, however, are usually not affected by this transformation. Thus we may experience the situation that we have high level data structures formulated in a prototyping language, following its data structuring principles and accessible in binary form in it, but not acccessible in the production language. Consequently, reusability of code may be intertwined with reusability of data. Reusing code by program transformations[1] ought to be complemented by a method of making data reusable by transformations. A first attempt to solving this problem in the context of transforming **SETL** programs to **Ada** may be found in [26].

Now consider a prototyping language like **SETL**; it has proven its ability to model programs in many applications, but the ability to provide a semantic data model with persistent data is confined to write binary data to external files. Although the possibilities of representing data in SETL are powerful because sets, tuples, and maps are available, binary files are somewhat insufficient to serve the purpose of data modelling sufficiently. What is missing is

- the ability to use procedures as first-class objects, thus making procedures available for data modelling purposes,
- a mechanism for accessing persistent data,
- a mechanism for controlling the namespace of persistent data.

All this requires a mechanism making data outlive the program that generated them. Persistence has been discussed mainly in the data base community (e.g. [2, 8, 21, 6]), but research in programming languages becomes increasingly aware of this problem, see e.g. [4, 5, 9].

SETL/E[2](see [15]) is a prototyping language supporting sets, maps, tuples, and procedures as the primitive data structuring facilities, the primitive datatypes being integer, real, strings, boolean, and atoms. It is based on **SETL** ([27, 13]). The compound data types sets, tuples, maps, and procedures can be freely mixed, in particular sets or tuples need not be homogeneous, thus we may construct e.g. sets which contain procedures, integers, strings, tuples, or maps as elements. This is the basic language we are working with.

We propose extending the basic language to support persistence. Persistence is intended to come as an orthogonal property, hence each and every datatype may be made persistent. In conformance with the idea of **SETL** not to bother the user with low-level details and to have the machine care about these details, we have made an attempt to make use of persistent structures as unobtrusive as possible. Thus there is not an explicit operation of transferring persistent data from some external medium into an executing program. Hence persistent values are available whenever the user needs them. Since the system cannot determine, however, whether the user wants to make use of a value later on in a persistent fashion there has to be an indication for storing the value; we have tried to make this as easy as possible.

[1] "... programs in any concrete high level programming language are the result of a mapping from some conceptual or abstract specification of what is to be accomplished into various specific data representations and algorithms which provide an efficient means for accomplishing the task at hand", see [11]

[2] Set-Theoretic Language/Essen

The containers for persistent values are called $P - files$; they are abstract data types supporting among others the straightforward operations of inspecting, inserting, and deleting values, and they form a separate scope. These $P - files$ are modelled somewhat after UNIX's archives.

A special case deserves separate discussion: procedures are first-class objects in SETL/E, thus procedures may be made persistent. As Atkinson et al (see [5]) have shown persistent procedures may be used for introducing modules and consequently for introducing separate compilation and (dynamic) binding. There is a small catch, though: SETL/E uses static binding for its procedures, modules use some weak form of dynamic binding. This is so since all the variables which are visible to the procedures of a module should be static: entering a module should find all these values unchanged from the last time the module was left. Hence we have to introduce with our module facility a possibility of dynamic binding in the sense just indicated. Using this facility, it may be shown that the basic principles of object-oriented programming (in particular inheritance) may be implemented using the augmented language.

Organization of the paper The next section will give a brief introduction to SETL/E mainly by discussing an example. We then discuss in section 3 the basic mechanisms for persistent data structures in SETL/E, and apply these considerations in section 4 to issues of separate compilation; this section discusses also an example for object-oriented programming. Section 5 indicates some open problems, and gives further directions for research, and appendix A displays a solution to a test case for persistent programming languages provided by Atkinson and Buneman.

Acknowledgements The author would like to thank J. Biskup, U. Gutenbeil, W. Hasselbring, and M. Schunk for some useful discussions, and Ingrid Kleinstoll for typesetting this manuscript.

2 A Brief Introduction to SETL/E

SETL/E is a decendant of the procedural language SETL ([27, 13]). It makes sets, maps and tuples available; these structures do not have to be homogeneous. In addition, procedures are first class objects, and a mechanism for exceptions is provided. The control structures show that the language has ALGOL as one of its ancestors; both SETL and SETL/E are weakly typed, freeing the user from specifying the data structure respresentations of the objects used in a program. SETL provides a *Data Representation Sublanguage* which allows giving hints to the compiler as how to represent the data in a program; since this sub-language is seldom used, SETL/E does without it. SETL provides a mechanism for separate compilation using modules and libraries. This mechanism is conceptually quite elegant (e.g. it allows circular dependencies) but is implemented in a rather impractical way. Its successor SETL/E merely provides a mechanism for monolithic programs, so a mechanism for specifying modules still has to be provided. SETL/E is described in [15]; the language is currently being implemented.

The program in Figure 1 for topologically sorting a directed graph provides an example. The graph is input by reading the set *edges* which contains pairs, i.e. tuples of length 2. An edge between the nodes x and y is indicated by listing the pair $[x, y]$ in the set *edges*. The variables *nodes* and *edges* are declared as **visible**, hence are accessible in all scopes subordinate to the one containing the declaration (variables and constants are by default local to the scope in which they occur). The set *edges* may be interpreted as a set valued map, assigning each node x the set $edges\{x\}$ of its neighbors. The set *nodes* is the domain of *edges*, i.e. the set of all first components of tuples in

```
program TopSort;
visible nodes, edges;
get("%s", edges);
nodes := domain(edges);
 -- $\boxed{\text{here}}$
if is_dag() then
        SortTup := [ ];
        while nodes <> { } do
           x := select y in nodes | (notexists z in nodes | [z, y] in edges);
           x into SortTup;
           nodes less x; edges lessf x;
        end while;
end if;
put("%x\n", SortTup[#SortTup .. 1]);
 -- define the procedure is_dag
procedure is_dag;
 -- returns true iff the graph does not contain a cycle
S := nodes;
shrink_S: loop
          z := select y in S | edges{y} * S = {  };
          if z = om then quit shrink_S; end if;
          S less z;
end shrink_S;
return (S = { });
end is_dag;
end TopSort;
```

Figure 1: Sorting a graph topologically

```
is_dag :=
    lambda:
     -- returns true iff the graph does not contain a cycle
    S := nodes;
    shrink_S: loop
            z := arb y in S | edges{y} * S = { };
            if z = om then quit shrink_S; end if;
            S less z;
    end shrink_S;
    return (S = { });
    end lambda;
```

Figure 2: Alternative definition of procedure is_dag

edges (we could alternatively have defined *nodes* as $\{e(1) : e \texttt{ in } edges\}$). The procedure *is_dag* tests whether or not the graph contains any cycles by repeatedly selecting and removing nodes from the candidate set S. If there is no longer any z to be removed from S, the variable z gets the value om, indicating that it is no longer defined. The procedure *is_dag* returns true iff the graph does not contain a cycle. The main program repeatedly selects a node x without a predecessor, puts x into the tuple $SortTup$ of nodes already sorted, removes x from the set of all nodes as well as all edges emanating from x. This is done until there are no longer any nodes to be processed. The program terminates after writing the nodes in reverse order in which they have been found.

Since procedures are first class objects, they may be assigned as values, appear as elements in tuples or sets, and they may occur in the domain or the range of a map. SETL does not allow for nested procedures, but being first class in SETL/E, procedures may be nested to any depth. Procedures may be anonymous akin to LISP's λ. We feel that this is a useful device since it allows procedures to be read in, and to be written out to external devices. The procedure *is_dag* could have been defined in the line marked with the comment $\boxed{\text{here}}$ as indicated in Figure 2. Parameters may be passed by value (rd parameters), by result (wr parameters), and by value/result (rw parameters). Non-local variables in local procedures are bound statically to the innermost static predecessor in which they occur.

SETL/E provides exceptions along the lines of Ada's model for exceptions: if an exception is not handled in the scope in which it was activated, the scope is left and the dynamic predecessor is searched for a handler. This is done until either a handler is found or the exception's name is no longer visible (in which case the exception UnDef_Exception is activated). Exceptions may be parametrized, though. In contrast to procedures, however, exceptions are no first class objects, hence they may not be passed as parameters or returned as results.

3 The Basic Mechanism for Persistence

In this section we describe the structure of persistent objects and of their containers, the abstract data type $P - file$. We then will discuss practical issues of handling persistent data. These considerations will be applied in the next section when we are going to discuss modules.

3.1 Persistent values

A persistent value p has the following structure: it consists of

- name
- type
- time stamp
- condition
- lock
- file (name of the container)
- the value itself.

We are going to discuss these data in turn.

Name. The name for the persistent value is syntactically an identifier, and the value will be identified by its name. Access mechanisms will be discussed below.

Type. Before discussing the type attribute of a persistent value, we introduce a total order among all values on which the representation of the attribute will be based. Define first an order relation $\preceq$ on the set of predefined types (hence the range of the **type** operator) by saying $\alpha \preceq \beta$ iff the name for type α is lexicographically smaller than the name for type β (thus `map` $\preceq$ `set`). Let $\sqsubseteq$ be the lexicographic order on the set of strings. Now define for two arbitrary SETL/E-values a and b the order relation $a \leq b$ iff either one of the following cases applies

- $a =$ `om`
- **type** $a \leq$ **type** b
- **type** $a =$ **type** b and
  - **type** $a =$ `tuple`: a is lexicographically smaller than b taking $\leq$ as the order relation on the components
  - **type** $a =$ **set**: $\tilde{a} \leq \tilde{b}$, where $\tilde{a}$ is a tuple, the elements of which are arranged in ascending order given by $\sqsubseteq$ on the **str**-values[3] of the elements of a (i.e. if $a = \{x_1, \ldots, x_n\}$, then $\tilde{a} = [x_{i_1}, \ldots, x_{i_n}]$ with $\{x_{i_1}, \ldots, x_{i_n}\} = a$, and **str** $x_{i_k} \sqsubseteq$ **str** $x_{i_{k+1}}$ for $1 \leq k < n$). Similar for $\tilde{b}$.
  - **type** $a =$ `map`: maps are treated as sets of pairs
  - **type** $a =$ `proctype`: **str** $a \sqsubseteq$ **str** b
  - **type** $a =$ `optype`: similarly using **str** and $\sqsubseteq$
  - in all other cases, the "natural" order on the type of a is applied.

[3]for each SETL/E value x, **str** x is a string containing the print image of x

This order relation is rather crude and looks somewhat arbitrary (it is, in fact). It basically follows the type structure of the values, gluing components together whenever necessary (an old trick in order theory). It is not difficult to show that $\leq$ defines a total order on the set of all SETL/E-values.

The *type tree* for the SETL/E-value a is an ordered tree having the type tag for a as the label for its root. If a is a simple value, then the root has this value as its only offspring. If a is a compound value with offsprings $a_1, \ldots, a_n$, however, order the components obtaining the chain $a_1 \leq a_2 \leq \ldots \leq a_n$, then the type tree for a_i is the i^{th} offspring of the root. This yields a unique description of a's type, and the type tree is the value for the type attribute (actually, a linearized version is stored).

Time stamp. The time stamp is a string indicating either the date of the persistent value's creation or of the last update for the value. The time stamp is a string of the form

$$YY : MM : DD : hh : mm : ss : \mu$$

It may be used e.g. for version control purposes.

Condition. The condition may be used for formulating integrity checks on the value it refers to or it may be used for formulating the relationship between the value under consideration and other values. It is formulated as an anonymous function (`lambda`), which may have arbitrary parameters. The value under consideration is available through the preserved identifier `TheValue`. Binding of non-local identifiers is done relative to the scope which is provided by the $P - file$. This will be discussed below.

Lock. The lock is an indicator whether or not the value may be overwritten.

File. $p.file$ indicates the identifier for the $P - file$ in which the value is stored; it is set when the persistent value is actually written to the $P - file$, and it may be inspected by the user of the persistent value.

The value. The value itself is stored in a binary format which allows fast and easy access in a SETL/E-program.

Given a persistent value p, we indicate its name by *p.name*, its type by *p.type*, its time stamp by *p.time*, the corresponding condition by *p.cond*, the default value of which is set to

```
    lambda: return true; end lambda;
```

The value *p.lock* is initially set to `false` indicating that the value may be overwritten.

3.2 The ADT *P-file*

Persistent values will usually be stored in files, but this may be implementation dependent, e.g. it may happen that cache memory is large enough to hold the contents of a small file at least partially at run time. Thus instead of describing the operations on such a file it is more adequate specifying only the operations one wants to perform with persistent values and to leave the concrete realization of these operations together with these specific management of (internal oder external) storage to an implementation.

$P - files$ may roughly be compared to *archives* under UNIX, see [20], Sec. 3.8. An archive consists of a table of contents and of the files (mostly binaries) which are stored in it. Archives may be accessed in a number of ways: one can read the table of contents, one may insert or delete an element from an archive and one may extract named elements from it. In addition, the archive is a UNIX-file, thus it may be identified through an identifier which is admissible under a particular shell.

The abstract data type $P - file$ is represented to the outside world by a string as an identifier. This identifier is used to access the $P - file$ in the same way we gain access to a file using its name. Let x be an identifier denoting a $P - file$. Then the following operations are defined for x:

1. $Create(x)$, creating a $P - file$ with the name x; initially the table of contents of x is empty,
2. $Discard(x)$, removing the $P - file$ with the name x,
3. $TableOfContents(x)$, yielding a set with all the identifiers for the persistent values which are stored in x.

These procedures may be accessed in a SETL/E-program. The ADT $P - file$ requires some other operations which may be used only implicitly by the programmer. We will discuss using these operations in a program in a moment, right now we will just make a list of these operations. Let x be a $P - file$, then these operations may be applied:

- Insert an element into x
- Remove an element from x
- Compress x, i.e. collect the garbage in x.
- $Dismantle(x, y, z)$: if y in $TableOfContents(x)$ holds, create a new $P - file$ named z with an initially empty table of contents, instert y into it, and remove y from x; otherwise, do nothing.

Other operations may be necessary, and it is desirable to have a box of tools allowing to access $P - files$.

Scoping Each $P - file$ is regarded as a scope of its own. This means that all the names contained in the table of contents are visible throughout the $P - file$ and that each value stored in it may entertain its own namespace. This applies of course to all the λ's that are stored in a $P - file$. These functions have access to the names stored in the table of contents as if these names would be visible throughout this scope of the λ.

3.3 Handling and Practical Issues

Atkinson and Buneman coin the phrase *persistent programming language* for "languages that provide for longevity for values of all types and that do not require explicit organization of, or even mention of, data movement by the programmer" ([3], p.110). In designing the persistent mechanism for SETL/E, we hold that this maxim is in particular important for those programming languages which deal with persistence in the context of prototyping issues. Since one tends to neglect details of data representation or declarations of variables in a prototyping language, one should not have to take care of explicit data movement to and from external files. With this in mind we are going to discuss handling of persistent values and other practical issues now.

A persistent value is used in SETL/E just as every other value is, in particular it is not necessary to declare this value as persistent. When the value is needed, it is mentioned, and consequently the value is retrieved from a suitable $P - file$. The names of the eligible $P - files$ are stored in a tuple called $@SearchPath$, which is initialized upon program start as $[\$StdLib]$. $\$StdLib$ denotes the $P - file$ for the standard library provided with the SETL/E-system. It contains among others the operations for input and output, and the operations relevant for handling $P - files$ ($Commit$, $TableOfContents$, $Create$, and $Discard$) may be found there, too. The tuple $@SearchPath$ is declared in the program's environment, hence it may be accessed throughout a program; its actual value is visible in each $P - file$, thus persistent procedures may use it. In addition, this tuple may be manipulated by the programmer. This is necessary when names for $P - files$ are inserted or deleted or when the programmer wants to change the order in which the $P - files$ are searched.

When the program encounters a name y on the left-hand side of an expression or a statement, and this name is not yet associated with a value, then by default the value of y would be set to om, the undefined value in SETL/E. In the presence of persistent values this strategy is modified as follows: The $P - files$ associated with the identifiers in $@SearchPath$ are searched in order for the occurrence of a persistent value associated with the identifier under consideration. The first value found is then bound to y, in particular the corresponding value is loaded from the $P - file$ into memory.

Since the search path may be set and modified by the programmer this is a rather flexible way of making persistent values available to a program. If the programmer insists on taking a persistent value from a particular $P - file$, then she may use qualified notation: $P.x$ indicates that the $P - file$ represented by P is searched for a value associated with name x.

If searching for a value does not succeed in any of the $P - files$ given in $@SearchPath$, only then the variable is indicated as being undefined, and its value is set to om. It should be noted that this handling of persistent value generalizes the canonical approach used by SETL/E in which a value is undefined if it did not receive a value either by an explicit assignment or by being an actual parameter written by a procedure. Variables in SETL/E are by default local in the scope in which they occur. Thus a variable which is defined in an outer scope is not visible in an inner scope unless explicitly declared as **visible** in the outermost scope. An attempt to access the variable in an inner scope ends up in setting the variable's value to the undefined value om. In the presence of persistent values om is taken only as a value if a corresponding persistent value cannot be found. Hence looking up the symbol table for checking the scope of a variable is augmented by looking up the table of contents of all the $P - files$ which are listed in $@SearchPath$.

This may result in inefficiencies, and the alternative here is either to resort to setting the global tuple $@SearchPath$ to the empty tuple (thus preventing any search beyond the program's symbol table), or to setting a particular compiler option when compiling the program (thus prohibiting the use of $P - files$ and hence the use of persistent values altogether). It will have to be seen which approach is more practical — for "small" programs it may be practical not having to bother with the management of persistence, and the explicit manipulation of the tuple $@SearchPath$ may turn out to be clumsy.

This approach to persistence (which may be called *silent persistence*) is appropriate to using a prototyping language e.g. for purposes of exploring a situation: the leitmotif here is helping the system to take care of as much as possible without the programmer's intervention. Our approach is in line with this philosophy.

Names destined to hold persistent values are treated as any other identifier in SETL/E, hence such a name may be declared as `visible` or as being a constant. Constants in SETL/E are dynamic (and not manifest as in the predecessor SETL). Their value may be determined dynamically, so a constant may have different values in different invocations of a procedure. Being a constant in SETL/E does not mean that its value may be determined at compile time but rather that the value is protected against changes. Normally a constant has to be given its value in the `constant`-declaration, but persistent values will be taken from the environment when such an initialization is missing.

Accessing a persistent value can be done silently, but we have more than one option for what to do with a value when leaving a program. One possibility is discarding the value as ephemeral, just as one would do in the absence of persistence. If, however, a value is to be saved as persistent, it has to be moved from the program to a $P-file$. Supposed we want to make p persistent. First we may want to set the attributes *p.time*, *p.cond* or *p.lock* if we do not want to rely on the default values. Let q be a string valued identifier representing a $P - file$, then a call to $commit(p, q)$ transfers the actual value of p to the $P - file$ associated with q. If something goes wrong (e.g. if the value already contained in q must not be changed, or if the $P - file$ does not exist, could not be opened for writing etc.), suitable exceptions are activated.

We prefer this archival approach to approaches like the one used in `Napier` (cp. [6]), or the one used in `Galileo` (cp. [1]). In those approaches a value is considered to be persistent iff it is reachable through a path from the persistent root. This means that a value is made persistent by connecting it to a value which is already persistent, thus putting p into the *persistent store* (as the analoga to $P - files$ are called) amounts to finding an already persistent element and connecting p to it.

Conceptually our approach is not too far away from this since the program making use of persistent values may be considered the persistent root having all the $P-files$ in $@SearchPath$ as offsprings, which have all the objects displayed in the table of contents as offsprings in turn. In this sense a $P - file$ may be considered as a flat tree. Conversely a tree in the sense of `Napier`'s persistent store may be turned into a flat tree by path compression. We feel that the approach used here is more natural to SETL/E, in particular it is easier to deal with questions of scoping in a more natural way. Our approach seems to be more flexible when more than one repository for persistent objects is to be used. The counterpart of a set of $P - files$ in `Napier`'s model would be a set of persistent stores amounting to a forest of rooted trees. This forest would have to be made into one single tree. In addition, making $@SearchPath$ available as a global value to the programmer provides additional flexibility which in the persistent store model would have to be achieved by manipulating the persistent tree as a whole.

4 Modules

The introduction of persistent structures allows introducing modules, thus making separate compilation of larger program units and hence *programming in the large* feasible. The relationship between persistence of procedures as first class objects and modules has been pointed out e.g. in [5] from which a line of development may be traced to the language `Napier` (see [6]) paralleling the approach found in `ML` (see [10]).

```
module
  gensym := lambda (symb):
    visible i := 0;
    return g;
     -- define the routine g
    procedure g;
     i + := 1;
     return str symb + str i;
    end g;
  end lambda;
end gensym;
```

Figure 3: Module *gensym*

Capitalizing on the simplicity of persistent structures for making modules available avoids introducing a separate mechanism for describing modules and separate compilation. That was done e.g. in Ada, where a package is told explicitly which other packages to use, and in SETL, where interactions between modules have to be described separately in a *directory*. Similarly, using packages in Ada usually requires a particular library format and a separate mechanism for binding, all outside .the language itself, hence dynamic loading of packages is not possible. In addition, packages suffer from other drawbacks that are implied by this approach, e.g. they must not be circular with respect to their import/export behavior. SETL on the other hand allows circularity, since modules are linked early enough to the programs using them, but there are some other drawbacks, e.g. changing the externally visible behavior of a module requires changes in the directory (where this behavior ist posted) and thus recompilation of the entire program.

The straightforward way of making a procedure persistent and loading it when it is required does not work in SETL/E since the intent of a module is not fully in accordance with this approach. A module is usually thought of as a collection of routines having access to common data structures. This requires static variables, i.e. variables maintaining their value between different invocations to a routine in the module from the outside. SETL/E binds statically, thus always the value from the static environment at definition time is taken. Hence we have to expand the binding mechanism by introducing dynamic binding.

4.1 Defining Modules

A module is defined between **module** and **end** followed by the module name. This is the simplest case, we will discuss particular cases shortly. The definition proper looks like the assignment of a **lambda** to an identifier. So upon defining *gensym* as in Figure 3, we make *gensym* as a module available; with $h := gensym(\text{"}g.\text{"})$ the invocations $x := h()$; and $y := h()$; generate the strings $g.1$ for x and $g.2$ for y, resp.

```
module
   stack(create, is_empty, push, pop);
   stack.specification {reads} := { };
   stack.specification {writes} := {create, is_empty, push, pop};
   profile(create) := [ ];
   profile(is_empty) := [ ];
   profile(push) := [rd];
   profile(pop) := [wr];
end stack;
```

Figure 4: Stack specification

Usually one is interested in the interface of a module expressed in the specification part. Since the implementation of the module is given explicitly, the specification part may be derived from the implementation. The specification part indicates which items are imported (or read) or exported (i.e. written) by the module; this is indicated by the set of the corresponding formal parameters. Thus

$$\begin{aligned} gensym.specification\{reads\} &= \{symb\} \\ gensym.specification\{writes\} &= \{\}. \end{aligned}$$

Conceptually, *gensym.specification* is a relation, relating each formal parameter to the way it communicates with the caller. Thus it is represented in SETL/E as a (multivalued) map. For each procedure read or written by a module the built-in map `profile` indicates the way parameters are passed: $profile(a)$ yields a tuple the ith component of which gives `rd`, `rw`, or `wr` depending on how the ith parameter is transmitted.

Modules may be specified in two parts, as usual: first the specification is given, and the implementation may be defined at a later time (but in the same scope) as in Figure 4. This specifies a module *stack*, and makes it usable right away: the invocation

```
stack(ThisCreate, IsEmpty, ThisPush, ThisPop);
```

defines the corresponding operations, hence

```
ThisCreate();
```

creates and initializes a stack, and

```
for i in [1...10] do ThisPush(i); end for;
```

pushes the elements 1,...,10 onto the stack. Finally,

```
while not IsEmpty() do ThisPop(k); end while;
```

pops the elements off the stack. Note that *IsEmpty*() is supposed to return a Boolean value, but that this is not visible from the specification (in accordance with SETL/E's philosophy of typechecking at runtime).

The implementation of this module is done in a straightforward way, see Figure 5.

Note that this mechanism displays all the properties usually associated with modules:

```
module
  stack.implementation :=
   lambda (create, is_empty, push, pop):
    visible LocStack;  -- Thus LocStack is visible throughout this λ
    create := lambda:
     LocStack := [ ];
    end lambda;
     --
     -- similar for is_empty and push
    pop := lambda (wr t):
     if LocStack = [ ] then
      raise Stack_Underflow;
     else
      t frome LocStack;
     end if;
    end lambda;
     --
    exception Stack_Underflow;
      -- whatever has to be done
    end Stack_Underflow;
  end lambda;
end stack;
```

Figure 5: Stack implementation

1. a module has a specification and an implementation part; both parts may be separated from each other, in particular is it possible to access the specification independently of the implementation,
2. a module may have variables which are global to all routines provided by it, but not visible to the outside. In particular, a module may have local routines not visible to the caller,
3. items may be imported and exported from a module,
4. a module may execute initialization code. This happens when the corresponding `lambda` is executed,
5. a module may interact with other modules, and this may be done dynamically.

The usual mechanisms for persistent values apply: a module is loaded silently after its name is mentioned from the first $P - file$ on the search path having the module name in its table of contents.

The specification of a module may be used as any other SETL/E-value. Suppose we have set

stack.cond := **lambda** : **return** *"purpose : maintains stacks"*; **end lambda**;

Assume further that we want to identify a module for maintaining stacks in a set of persistent values stored in the $P - files$ or the search path. This module should not read anything, but write four different procedures. The following piece of code displayed in Fig. 6 performs this task. A module m may be distinguished from a persistent routine r by the fact that in the former case specification and implementation are defined, and that neither of these items is defined for r. Thus we have e.g. $r.specification$ = om, and trying to access it will usually result in a run time error (if it is not detected at compile time).

```
for Pf in @SearchPath do
  for t in TableofContents (Pf) do
    if t.cond() = "purpose: maintains stacks" then
     if t.specification {reads} = { }
       and -- short circuit
       #t.specification{writes} = 4
       and
       forall q in t.specification {writes} | type q = proctype
     then -- whatever needs to be done
     end if; -- innermost if
    end if;
  end for; -- innermost for
end for;
```

Figure 6: Searching for a module

4.2 Other Approaches

PS-algol seems to have been the first language to implement separate compilation using persistence. The approach used here is similar to the one outlined in [5], although there are differences, some of a syntactic nature, some not. PS-algol first defines a structure in which the signatures of the objects involved are specified; then a *let*-environment is used to establish the respective functionalities. There is a more serious difference, however, in the way dynamic binding is achieved: in the present proposal dynamic binding is confined to the module-environment, and the rest of the language is statically bound, whereas PS-algol binds statically without any exceptions. The effect of dynamic binding, however, is achieved by the pointer type available there: the type of an object being pointed at is not determined at compile time but rather at run time; pointer types are used heavily in PS-algol's module facility.

Napier (cp. [6, 24]) introduces a construct called a *namespace* for controlling bindings. A namespace is introduced as the abstraction of "store of arbitrary permanence" ([6], p. 8). Binding of names to objects (rather than to values) may be static or dynamic in a namespace. Since SETL/E binds names to values (and the question of object identity does not arise), the binding strategy is somewhat different to the one used in Napier — a first approximation would classify static binding in SETL/E as a special case to static binding in Napier. Since namespaces are a type of their own, the approach used in Napier seems to be more general than the one used here.

ML uses structures and functors as basic linguistic units for separate compilation (see [17]). A *structure* provides an encapsulated environment and is the building block for a ML program. Structures may be composed using functors, so a functor may be used to build a new structure from its arguments. Functors can be made persistent.

4.3 Dr. King's Cat is Object-Oriented

The discussion in [22] focusses on different aspects of object orientation vs. semantic modelling in data bases. King distinguishes the structural abstractions provided by semantic models from the behavoiral abstractions provided by object-oriented models. We are working here in the context of set theory, thus structural abstractions (which are achieved through *grouping*, i.e. set valued properties, ***attributes***, i.e. named properties, and *aggregations*, i.e. building up components) may be described easily in this context. Since this requires a more careful discussion than the author feels appropriate undertaking here, we move to the problem of object orientation.

There seems to be agreement that an object-oriented approach to problem solving requires the programming language to support at least

- object creation
- encapsulation
- inheritance
- message passing

We claim the SETL/E supports these properties. To substantiate this claim, consider the following example from geometry. An *ellipse* is characterized by its semiaxes a, b (with $a \geq b$), analytically, it is described by

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

Such an ellipse has the area

$$S(a,b) := \pi \cdot a \cdot b$$

and the circumference

$$C(a,b) := 4 \cdot a \cdot \int_0^{\pi/2} \sqrt{1 - e^2(a,b)\sin^2\varphi\, d\varphi},$$

where $e(a,b)$ denotes the numerical excentricity

$$\sqrt{1 - \frac{b^2}{a^2}}$$

The case $a = b = r$ specializes the ellipse to a *circle* with radius r that has the area $S(r,r) = r^2 \cdot \pi$ and the circumference $C(r,r) = 2 \cdot r \cdot \pi$, since $e(r,r)$ vanishes. When specializing one may want to inherit the computation for the area, but rather not the one for the circumference, since it is apparently impractical to approximate the integral in this case.

Since objects are supposed to have a local state which may change as the object interacts with its environment, we formulate the objects in question as modules in Fig. 7 Thus invoking *Ellipse*(fl, ci), we have created procedures fl and ci such that $fl(a,b)$ yields the area of an ellipse with semiaxes a and b. In the same way we may create an ellipse by saying yo := *Ellipse*. Then $yo(f,c)$ will produce the same functions f and c.

A *circle* is a special case, see Fig. 8. Thus *Circle* inherits ar from *Ellipse* and turns it by specialization into *area*; the computation of the circumference for *Ellipse* is discarded and replaced by a local procedure *circumf*. If we would be willing to pay the price for computing an elliptic integral, we could modify the call to *Ellipse* by saying *Ellipse*$(ar, circumf)$ and remove the λ.

Hence *inheritance* may be represented through passing parameters: an object $\mathcal{A}$ inherits δ from an object $\mathcal{B}$ if $\lambda_{\mathcal{A}}$ invokes $\lambda_{\mathcal{B}}$, and δ is one of the parameters to this call. *Message passing* is implemented by invoking procedures: putting ka := *Circle*(ar, ci) and invoking ar by the assignment fl := $ar(17.4)$ may be thought of as passing the message *area* with parameter 17.4 to the object *Circle*. *Object creation* is done by copying prototypes rather than by sending the message *new* to a class.

Some models for object oriented programming allow for multiple inheritance. So does SETL/E. Since the name of a method in the present model is determined by the caller and not by the callee, naming and the possible duplication of names does not present a problem.

```
module
  Ellipse = lambda (wr area, wr circum):
    visible constant $\pi$ = 3.14159;
     -- $\pi$ is global throughout this scope
    procedure excen(x,y);
     return $\sqrt{1 - \frac{x^2}{y^2}}$;
    end excen;
     -- this is a local routine
     --
    area := lambda (a,b):
      return $\pi * a * b$;
     end lambda
    circum := lambda (a,b):
    return
    $4 * a * \int_0^{\pi/2} \sqrt{1 - (excen(a,b) * \sin(\varphi))^2}\, d\varphi$;
  end lambda;
end lambda;
end Ellipse;
```

Figure 7: Object ellipse

```
module
  Circle := lambda (wr area, wr circumf):
    visible constant $\pi$ := 3.14159;
    Ellipse (ar, ci);
    area := lambda (r):
      return ar (r,r);
     end lambda;
    circumf := lambda (r):
      return $2 * r * \pi$;
    end lambda;
  end lambda;
end Circle;
```

Figure 8: Object circle

5 Further Research

The previous sections laid out the basic mechanisms for persistence of data and for the handling of separate compilation using modules. This is a rather complex area, and only a first step could be done. We give a list of some of the research issues which further work will have to address.

Roughly three areas where more work is needed can be identified:

- questions concerning database issues,
- the proposed module structure for SETL/E,
- problems concerning the transformational paradigm which is directly related to the usefulness of SETL/E as a prototyping language.

We are going to discuss each of these points in turn, but the reader should be aware of the fact that not all questions can be answered in an isolated way, because some of them are heavily intertwined.

5.1 Database Issues

The ADT $P - file$ has to be represented efficiently both in terms of space utilized and in terms of time making use of persistent structures. A first approximation to the representation of this ADT is an archive under UNIX. Although this representation may serve the immediate needs for making these ideas work, it is insufficient for obvious reasons. Thus for making things work out smoothly and efficiently it may be necessary to find an effective balance between storing persistent structures on an external device, and caching portions of a $P - file$ in primary memory. Since fast local interconnections between machines become more and more feasible the question of maintaining a $P - file$ in a distributed way has to be investigated.

It may well be possible that two or more persistent values share a common substructure. Thus it is natural to ask for a possibility of isolating that substructure, storing it separately and giving the superstructure access to the substructure using a surrogate. This question arises also in object-oriented databases and shows a connection between persistence in our context of prototyping and these databases. In this context the problem of object identity arises.

From a practical point of view it is desirable to visualize the data sitting in a $P - file$. This may e.g. entail the representation of mutual dependencies and the like. Such a visual tool may be complemented by a visual editor allowing to browse, modify, and manipulate the data represented on the screen.

Is an SQL-interface desirable?

5.2 Modules

Modules may be represented within SETL/E, hence it may be practical describing module interconnections in this same language, too. Thus it may be desirable to develop a *module interconnection language* as a level of description which would minimally allow (cp. [25], p. 119):

- specify the way modules are composed to form larger structures

- describe the dependency structure between modules
- make sure that the proper versions for the modules are used
- perform checks concerning types and signatures (note that since SETL/E is weakly typed a strong type checking in the sense of Ada is not feasible).

This means that a module interconnection language may help in the composition process of a complex program and may in this way support programming in the large.

Quite related to that is the question of *version control*; here a mechanism needs to be implemented which checks the version of persistent data and makes sure that compatible versions are used. This does not need to be confined to modules. In particular, substructures may have to be checked, and it may be useful to have the possibility of storing a sequence of Δs.

5.3 Transformations

As indicated in the Introduction the transformation of programs has to go hand in hand with the transformation of data when it comes to derive a production efficient version of a prototype. This means in particular that for the persistent values of SETL/E a semantically equivalent counterpart has to be found. A first step in this direction is described and implemented in [26]. This tool addresses a subset of the data which may be described in SETL. For a full treatment a proper type-theoretical foundation is necessary. Thus we need an adequate type system both for SETL/E and the production language (e.g. Ada), and in addition we need a semantics preserving transformation between these type systems. Since recursive data structures are involved it may be attractive to use the *MacQueen-Plotkin-Sethi* model of σ-ideals as types (cp. [23]) together with a suitable topological or uniform structure which necessarily would have to impose some continuity assumptions on the transformation (cp. [12]).

References

[1] *Albano, A., Giannotti, F., Orsini, R., Pedreschi, D.:* The Type System of Galileo. In [4], 101 - 120

[2] *Atkinson, M. P., Bailey, P. J., Chisholm, K. J., Cockshott, P. W., Morrison, R.:* An Approach to Persistent Programming. The Computer Journal 26, 4, 1983, 360 - 365

[3] *Atkinson, M. P., Buneman, P.:* Types and Persistence in Database Programming Languages. ACM Computing Surveys 19, 2, 1987, 105 - 191

[4] *Atkinson, M. P., Buneman, P., Morrsion, R. (Eds.):* Data Types and Persistence. Springer-Verlag, Berlin, 1988

[5] *Atkinson, M. P., Morrsion, R.:* Procedures as Persistent Data Objects. ACM Trans. Prog. Lang. Syst. 7, 4, 1985, 539 - 559

[6] *Atkinson, M. P., Morrsion, R.:* Types, Bindings, and Parameters in a Persistent Environment. In [4], 3 - 20

[7] *Budde, R., Kuhlenkamp, K., Mathiassen, L., Züllighoven, H. (Eds.):* Approaches to Prototyping. Springer-Verlag, Berlin, 1984

[8] *Buneman, P., Atkinson, M. P.:* Inheritance and Persistence in Database Programming Languages. In Proc. ACM SIGMOD '86, International Conference on Management of Data, Washington D.C., 1986, 4 – 15

[9] *Buneman, P.:* Data Types in Database Programming. In [4], 91 – 100

[10] *Cardelli, L., Wegner, P.:* On Understanding Types, Data Abstraction and Polymorphism. ACM Computing Surveys 17, 4, 1985, 471 – 522

[11] *Cheatham, T.:* Reusability Through Program Transformations. IEEE Trans. Softw. Eng. 10, 5, 1984, 589 – 594

[12] *Doberkat, E.-E.:* Topological Completeness in an Ideal Model for Recursive Polymorphic Types. SIAM J. Comput. 18, 5, 1989, 977 – 989

[13] *Doberkat, E.-E., Fox, D.:* Software Prototyping mit SETL. Teubner-Verlag, Stuttgart, 1989

[14] *Doberkat, E.-E., Gutenbeil, U.:* SETL to Ada — Tree Transformations Applied. Information and Software Technology 29, 1987, 548 – 557

[15] *Doberkat, E.-E., Gutenbeil, U., Hasselbring, W.:* SETL/E Sprachbeschreibung. Essener Informatik-Berichte, Universit"at — Gesamthochschule — Essen, April 1990

[16] *Floyd, Ch.:* A Systematic Look at Prototyping. In [7], 1 – 18

[17] *Harper, R.:* Modules and Persistence in Standard ML. In [4], 21 – 30

[18] *Hekmatpour, S., Ince, D.C.:* Rapid Software Prototyping. Oxford Surveys in Information Technology 3, 1986, 37 – 76 (an expanded version has been published in 1988 under the title *Software Prototyping, Formal Methods and VDM* by Addison-Wesley)

[19] *Hull, R., King, R.:* Semantic Database Modelling: Survey, Applications and Research Issues. ACM Computing Surveys 19, 3, 1987, 201 – 260

[20] *Kernighan, B. W., Plauger, P. J.:* Software Tools in Pascal. Addison-Wesley Publishing Company, Reading, MA, 1981

[21] *Khoshafian, S., Briggs, T.:* Schema Design and Mapping Strategies for Persistent Object Models. Information and Software Technology 30, 1988, 606 – 616

[22] *King, R.:* My Cat is Object-Oriented. In Kim, W., Lochovsky, F. (Eds.): Object-oriented concepts, databases, and applications. Association for Computing Machinery, New York, 1989, 23 – 30

[23] *MacQueen, D., Plotkin, G., Sethi, R.:* An Ideal Model for Recursive Polymorphic Types. Information and Computation 71, 1986, 95 – 130

[24] *Morrison, R., Brown, A. L., Carrick, R., Connor, R. C. H., Dearle, A., Atkinson, M. P.:* Polymorphism, persistence and software re-use in a strongly typed object-oriented environment. Software Engineering Journal, November 1987, 199 – 204

[25] *Prieto-Diaz, R., Neighbors, J.:* Module Interconnection Languages. J. of Systems and Software 6, 1986, 307 – 334

[26] *Schunk, M.:* Austausch persistenter Datenstrukturen zwischen Ada und SETL. M.S. Thesis, Dept. of Computer Science, University of Hildesheim, 1990

[27] *Schwartz, J.T., Dubinsky, E., Dewar, R., Schonberg, E.:* Programming With Sets, An Introduction to SETL. Springer-Verlag, New York, 1986

A Appendix: The Atkinson & Buneman Test Case

In their survey on persistence in database programming languages Atkinson and Buneman present a test case ([3], p. 115f) for illustrating some issues. This test case consists of a fragment of a manufacturing company's parts data base. The data base represents the inventory consisting of parts. Parts may be basic or composite; if they are basic, they are not manufactured out of other parts. This information is supplied for basic parts:

- the name,
- the supplier and the cost of purchasing.

If parts are composite, they are manufactured out of other parts, and for each part it should be recorded

- the subparts that are involved in its manufacture,
- the cost of manufacturing a parts from its subparts,
- the mass increment that occurs when the parts are assembled.

The following tasks are presented ([3], p. 115f):

1. Describe the database.
2. Print the names, cost and mass of all imported parts that cost more than \$ 100.
3. Print the total mass and total cost of a composite part.
4. Record a new manufacturing step in the database, that is, how a new composite part is manufactured from subparts.

We describe a solution to the problem and to the four tasks now in SETL/E. Parts are represented as atoms, and *partSet* is the corresponding set. The following maps are defined:

$$\begin{aligned} P\_name: &\ partSet \rightarrow string \\ P\_simple: &\ partSet \rightarrow boolean \end{aligned}$$

$P\_name(x)$ is the name of part x, and $P\_simple(x)$ indicates whether or not x is composite. Abbreviating

$$\begin{aligned} partSet_{\mathtt{true}} &:= \{x \in partSet; P\_simple(x) = \mathtt{true}\} \\ partSet_{\mathtt{false}} &:= \{x \in partSet; P\_simple(x) = \mathtt{false}\} \end{aligned}$$

($partSet_{\mathtt{true}}$ and $partSet_{\mathtt{false}}$ is the set of all basic and composite parts, resp.), we define

$$\begin{aligned} P\_supplier: &\ partSet_{\mathtt{true}} \rightarrow string \\ P\_cost: &\ partSet_{\mathtt{true}} \rightarrow real \\ P\_smass: &\ partSet_{\mathtt{true}} \rightarrow real \\ P\_suppcost: &\ partSet_{\mathtt{false}} \rightarrow real \\ P\_mass: &\ partSet_{\mathtt{false}} \rightarrow real \\ P\_subparts: &\ partSet_{\mathtt{false}} \rightarrow \mathcal{F}(partSet) \end{aligned}$$

```
lambda:
  return
    type TheValue = set
    and  -- short circuit
    forall x in TheValue |
      if P_simple(x) then SimplePart(x)
      else CompositePart(x) end;
  --
  -- local functions
  procedure SimplePart(t);
   return
    [type(P_name(t), type P_cost(t), type P_smass(t)]
    =
    [string, real, real];
  end SimplePart;
  procedure CompositePart(t);
   return
    [type P_suppcost(t), type P_mass(t);type P_subparts(t)]
    =
    [real, real, map]
    and
    forall q in P_subparts(x) |
      q in TheValue
      and
      type (P_subparts(t)(q)) = integer
      and
      P_subparts(x)(t) > 0;
  end CompositePart;
end lambda;
```

Figure 9: Condition for *partSet*

Thus *P_supplier*(x) yields the name of the supplier for the basic part x, it may be purchased for a \$-price of *P_cost*(x), and has a weight of *P_smass*(x) grams. The composite part x incurs a \$-price of *P_suppcost*(x) for manufacturing it, and assembly increases its weight by *P_mass*(x) grams; finally, *P_subparts*(x) is a partial map from *partSet* to the naturals indicating that subpart y is needed *P_subparts*(x)(y) times in the assembly of x.

The set *partSet* and the maps *P_name*, *P_simple*, *P_supplier*, *P_cost*, *P_smass*, *P_suppcost*, *P_mass*, and *P_subparts*(x) are made part of the database, thus need to be made persistent. The condition *partSet.cond* for the persistent value *partSet* is described by the λ displayed in Fig. 9 This yields the description of the database and solves the first task. The second task is solved by the following straightforward code, which would look exactly the same if the data would be ephemeral: we iterate over *partSet*, select the simple parts and check the condition: see Fig. 10 Note that *partSet* and the *P_∗ ∗∗*-maps are loaded into the program if they are not already there.

The third task requires interleaving arithmetic with recursion, since we have to descent the subparts hierarchy. Its solution is displayed in Fig. 11 The procedure *MassAndCost*(x) returns a pair with the mass as a first, and the cost as the second component. The case of a composite part first computes *MassAndCost*(t) for each component t and collects then the intermediate results in a map *CompTup*; the component t contributes *P_subparts*(x)(t) $*$ *CompTup*(t)(1) to the mass, and *P_subparts*(x)(t) $*$ *CompTup*(t)(2) to the cost (remember that *P_subparts*(x)(t) indicates the number of times t is a subpart of x). The compound operator $+/$ applied to a tuple sums its

```
for x in partSet | P_simple(x) do
  if P_cost(x) > 100.00 then
     put("name = %s, cost = %s, mass = %f\n", P_name(x), P_cost(x), P_smass(x));
  end if;
end for;
```

Figure 10: Solution to the second task

```
procedure MassAndCost(x);
 -- We raise an exception if the data is not appropriate
if x notin partSet then
  raise partSet_in_Error;
else
  if P_simple(x) then  -- that is easy
   return [P_smass(x), P_cost(x)];
  else  -- composite part
   -- we collect mass and cost for the components in a separate map CompTup
   CompTup := { };
   for t in domain P_subparts(x) do
    CompTup(t) := MassAndCost(t);
   end for;        -- note that CompTup(t) is a pair
   mss := P_mass(x) + (+/[P_subparts(x)(t) * CompTup(t)(1): t in domain P_subparts(x)];
   cst := P_cost(x) + (+/[P_subparts(x)(t) * CompTup(t)(2): t in domain P_subparts(x)];
   return [mss, cst];
   end if;  -- innermost if
   end if;  -- outer if

 exception partSet_in_Error;
  -- whatever has to be done here
 end partSet_in_Error;

 end MassAndCost;
```

Figure 11: Solution to the third task

components.

For a description of the solution to the fourth problem, we assume that a new part is given as a tuple, the first component of which is its name, and the second component is a Boolean indicating whether or not it is compound. Depending on this value, we assume that in the next components the following values are stored in the order below:

simple = `true`: supplier, cost and mass

simple = `false`: subparts, cost of manufacturing and mass increment; subparts is given as a map, i.e. as a set of tuples

The solution to this task is then described by the following **`lambda`** (Fig. 12), which is made persistent, and which has to be invoked whenever a new part is to be recorded in the data base. It takes two arguments: the first, *ThisPart*, one is a tuple in the format just described, the second one, *Pf*, is a string representing a $P - file$ to which the corresponding values will be committed.

```
MakeEntry := lambda(ThisPart, Pf):
   -- We first check some types in the first argument
  if ThisPart assert tuple then
   if not ThisPart(2) then
     ThisPart(3) assert map;
   end if; -- inner if
  end if; -- outer if
   -- having survived these typechecks, we may go on
   -- create a new part and insert it into the set of all parts
  NewPart := newat(); NewPart into partSet;
  P_name(NewPart) := ThisPart(1);
  P_simple(NewPart) := ThisPart(2);
   -- commit the common parts
  commit(partSet, Pf); commit(P_name, Pf); commit(P_simple, Pf);
  if P_simple(NewPart) then
   [P_supplier(NewPart), P_cost(NewPart), P_mass(NewPart)] := ThisPart(3 ..5);
   commit(P_supplier, Pf); commit(P_cost, Pf); commit(P_mass, Pf);
  else
   [P_subparts(NewPart), P_suppcost(NewPart), P_mass(NewPart)] := ThisPart(3 ..5);
   commit(P_subparts, Pf); commit(P_suppcost, Pf); commit(P_mass, Pf);
  end if;
   --
   -- assert needs to be defined
  operator assert(obj, tpe);
   -- checks obj's type against tpe
  if type(obj) <> tpe then
   raise Arg_Error;
  else
   return true;
  end if;
  end assert;
   --
  exception Arg_Error;
     -- whatever has to be done here
  end Arg_Error;

  end lambda;
```

Figure 12: Solution to the fourth task