Abstract

We discuss the prototyping language PROSET (Prototyping with Sets) as a language for experimental and evolutionary prototyping, focusing its attention on algorithm design. Some of PROSET's features include generative communication, flexible exception handling and the integration of persistence. A discussion of some issues pertaining to the compiler and the programming environment conclude the paper.

1 Introduction and Overview

This paper discusses the programming language PROSET which is an acronym for PROTOTYPING WITH SETS [1]. This language has been defined and is currently being implemented at the University of Essen; it is a descendant of the set-oriented prototyping language SETL [2].

This introductory section is intended to provide some thoughts regarding the definition of this language. In particular we will have a brief look at prototyping and how it might influence language design. We discuss the relationship of PROSET to SETL and demonstrate the flavor of the language in a brief example.

1.1 Prototyping

It is well-known that the classical software life cycle has some drawbacks which suggest that it should be complemented by some auxiliary activities. This is true in particular for the early phases. One of the main drawbacks is the lack of support to experimental or exploratory programming. Somewhat related to this problem is the observation that the user's involvement in designing a program is kept to a minimum. Basically the user is only involved during the very early phases when it comes to more or less informally stating the requirements, and at a rather late phase when it comes to acknowledge the functionality of the program. This observation is particularly striking when modeling user interfaces, but it is not restricted to that area. Prototyping tries to find a way out of these problems by assigning the user a more active role during requirements elicitation, and by making experimental and exploratory programming part of the activities related to program design. This approach to program construction may complement the life cycle approach by incorporating a prototype subphase between planning and requirements definition during the analytic phase. Boehm's spiral model [3] also takes prototyping into account by proposing prototyping phases to be carried out after risk analysis and assessment.

Having a look at the literature it is difficult to find a concise definition of software prototyping since this is really some sort of umbrella term, covering a multitude of activities more or less related to each other. We stick to the description given by Christiane Floyd [5]: "Prototyping . . . refers to a well-defined phase in the production process where a model is produced in advance, exhibiting all the essential features of the final product for use as test specimen and guide for further production." This description emphasizes that prototyping really means modeling of software, it implies that the model itself should be an executable program. Moreover, it is seen from this description that prototyping should be an activity aiming at the
rapid production of a piece of software, since otherwise the effects of modeling would be lost. This in turn implies that a language for the support of software prototyping should provide powerful features, in particular versatile data structuring facilities together with convenient control structures operating on these complex data structures.

Consequently we need powerful mechanisms based on a somewhat natural formal calculus. We emphasize a natural approach here since it should be possible to express one’s thoughts for constructing a program in a programming language rather close to the way one does express things mathematically. Finite set theory provides such a way of cleanly expressing one’s thoughts, and our proposal for a prototyping language is based on set theory augmented by bits and pieces from $\lambda$-calculus.

1.2 SETL as an Ancestor

Using set theory for the purpose of formally describing program designs is by no means new, and the most prominent programming language making finite sets available has been SETL. This venerable language was designed during the seventies at New York University’s Courant Institute of Mathematical Sciences by J.T. Schwartz and his group. The late seventies, and the early eighties saw implementations of this language on a variety of machines ranging from mainframes to work stations. Subsequently, the language has been used, and has proven the modeling capacities of the language in a convincing way. Highlights are

- the development of the first Ada compiler (certified in April 1983) [6],
- the SETL optimizer (which really was an encompassing prototype of optimization techniques for procedural languages) [7],
- the Rutgers Abstract Program Transformation System RAPTS [8],
- WAA, a tool for analyzing Pascal program fragments with respect to their potential for reuse [9].

The day-to-day use of SETL, however, indicated that the language is not free of problems since it displayed some very baroque features, sometimes more hindering the use of the language than supporting it. This applies particularly to programming in the large, the organization of separately compiled components was felt to be rather awkward. In addition, the arsenal of data structures was considered incomplete since functions as citizens with first class rights are missing, the possibilities of making values persistent are felt as a lack and parallel programming is not possible at all. The programming environment was the subject of the ESPRIT project SED during 1986 to 1989. Some progress has been made here, but regretfully the goal of integrating all the results into a coherent and uniform programming environment could not be achieved.

When we had a look at SETL we decided that we wanted to reimplement it, clean up some of the features and incorporate constructs we felt would be helpful. Reimplementation occurred to be necessary since SETL was originally implemented in a little known systems implementation language called LITTLE.

When working on the new language design and observing the design of SETL2 proposed by Kirk Snyder of Courant Institute [10] we decided to incorporate some features into our new language. The following features distinguish our language proposal from both SETL and SETL2:

- Data abstraction is supported uniformly by the data types function, module, and instance (note that SETL2 also provides anonymous functions).
- Control abstraction is supported by a variety of constructs for exception handling.
- Data modeling is supported by persistence; each and every value having first class rights in the language may be made persistent.
- Parallel programming is supported by features for generative communication; the control primitives provided by the LINDA model for concurrent programming [11] serve as a basis for some primitive operations in our language.

To avoid confusion between SETL and its variants, and to add a stone to the Tower of Babel we decided to give the language a new name.

1.3 An Introductory Example: A Solution for the Queens’ Problem

We shall now present an introductory example to give the reader a first impression of the language. In Fig. 1 a ProSet-solution for the so-called queens’ problem is given. Informally, the problem may be stated as follows:
program Queens;
constant N := 4;
persistent constant npov, abs: "StdLib";
begin
  fields := [{x,y}: x in [1..N], y in [1..N]];
pus ((NextPos: NextPos in npov(N, fields) | NonConflict(NextPos)));

procedure NonConflict (Position);
begin
  return forall F1 in Position, F2 in Position |
  ((F1 /= F2) implies
   (F1(1) /= F2(1) and F1(2) /= F2(2) and
    abs(F2(1)-F1(1)) /= abs(F2(2)-F1(2))));
end NonConflict;
end Queens;

Figure 1: Solution for the queens' problem.
The predefined function npov(k, s) yields the set of all subsets of the set s which contain exactly k elements. The predefined function abs returns the absolute value of its argument. These functions are loaded from the persistence store. NonConflict checks whether the queens in a given position do not attack each other. It is possible to use procedures with appropriate parameters as user-defined operators by prefixing their names with the "!" symbol. This is done here with the procedure \texttt{implies}. \texttt{T(i)} selects the \texttt{i}th element from tuple \texttt{T}.

\textbf{Is it possible to place} \texttt{N} \textbf{queens} \texttt{(N \in \mathbb{N})} \textbf{on an} \texttt{N \times N} \textbf{chessboard in such a way that they do not attack each other?}

Anyone familiar with the basic rules of chess also knows what "attack" means in this context: in order to attack each other, two queens are placed in the same row, the same column, or the same diagonal.

The program in Fig. 1 does not solve the above problem directly. It prints out the set of all positions in which the \texttt{N} queens do not attack each other. If it is not possible to place \texttt{N} queens in non-attacking positions, this set will be empty. We denote positions on the chessboard by pairs of natural numbers for convenience (this is unusual in chess, where characters are used to denote the columns). \texttt{[i,j]} denotes the lower left corner. This program with \texttt{N=4} produces the following set as a result:

\[
\{\{(1, 3), (2, 1), (4, 2), (3, 4)\},
\{(3, 1), (1, 2), (2, 4), (4, 3)\}\}
\]

As sets are unordered collections, the program may print the fields and positions in a different order. Note that there are no explicit loops and that there is no recursion in the program. All iterations are done implicitly. One may regard this program also as a specification of the queens' problem.

1.4 \textbf{Overview}

Now that we have discussed general aspects of prototyping, and have given an example for the modeling capacities of \texttt{PROSET}, we will discuss the language in greater detail. This happens in section 2, where first some salient linguistic features of \texttt{PROSET} are discussed (2.1) and illustrated by a set of examples (2.2). Section 3 outlines the compiler and indicates some points deserving further discussion. Finally section 4 sketches some of the work being done currently in the areas of type inference and data structure selection (4.1) and outlines further work to be done for the programming environment into which the language system will be embedded.
2 The Prototyping Language ProSet

2.1 Salient Features

This section provides a brief discussion of some of ProSet's features which might be of interest. We briefly look into issues pertaining to the type system, to making values persistent, and to programming parallel applications.

Data Types ProSet makes the data types from finite set theory available. The primitive data types provided by the language are of course integer, real, boolean, string, and atom. Compound data types include finite sets and finite tuples. They have their usual mathematical semantics, in particular we note that we deal with value semantics rather than with pointer semantics. Consequently, copying a compound value and modifying the copy will not affect the original. Sets may be described as familiar in mathematics, viz., by enumerating the elements and by describing their elements through properties. The same applies to tuples. Having these types available makes it easy to construct mathematical maps and relations by simply forming subsets of a Cartesian product. All these data types are accompanied by the usual operations (intersection, union, concatenation etc.). Thus the convenience of using finite set theory for describing solutions to problems is fully available.

Control Structures The control structures are rather canonic: we provide the usual arsenal of control structures deriving from e.g. ALGOL and define some operations which take the available compound data types into account. It is for example possible to iterate over a set and perform an operation for each element of this set, or to test whether or not some property is true for each element of a tuple. Having these types available makes it easy to construct mathematical maps and relations by simply forming subsets of a Cartesian product. All these data types are accompanied by the usual operations (intersection, union, concatenation etc.). Thus the convenience of using finite set theory for describing solutions to problems is fully available.

Procedures Procedures are polymorphic and return a value. This is parametric polymorphism in contrast to predefined operators, which are just overloaded. Parameters may be passed by value, by result, and by value/result. This is very similar to SETL, in addition it is possible to define anonymous functions (As). Procedures and As may be converted into values of type function using a closure operator. The closure of a procedure freezes bindings to the values of all non-local objects. Functions (i.e. the respective results of applying the closure operator) obtain an identity in a rather straightforward way, consequently these values may be handled as any other value with an identity. In particular these values may be elements of sets, arguments to procedures and functions, and they may be returned by them. This is quite similar to, but subtly different from, the way things are done in the SETL2 programming language.

Exception Handling Through an exception handling mechanism we integrate a device for dealing gracefully with errors in the program. For dealing flexibly with a large class of situations we extend the notion of error handling to exception handling. An exception is a non-normal situation occurring in the course of executing a program unit which has to be handled by the invoking unit. Thus exception handling is also a device for structuring and modeling, i.e. a device for concisely formulating the algorithm and for separating exceptional conditions and their handling from the algorithm. An important improvement to early approaches to exception handling is the distinction of exceptions and their handling units which may be associated dynamically with each other. Thus, the course of actions when handling an exception is as follows:

1. If an exceptional condition is detected, an exception is raised, i.e. this event is signaled to the immediate caller (and only to the immediate caller).
2. The caller reacts by invoking a previously associated handler (implicitly a default handler is associated, if no handler has been associated explicitly by the user).
3. The purpose of such handlers is diagnosing and handling the situation, and finally determining the subsequent flow of control.

In handling exceptions ProSet supports both a termination and a resumption model, i.e. the execution of the exception raising unit may be terminated or resumed by the handler. This is determined dynamically. Exception handling introduces a new principle of responsibility. Exceptions should not be handled by the detecting unit, but by the superior one.

The clean separation of the specification of the algorithm under normal conditions from the description of exceptional situations together with a specification of how to handle them is an aspect of prototyping which helps the modeler as well as the prospective user to better understanding the application to be prototyped. Thus the system to be constructed becomes clearer, the communication between user and modeler is enhanced, and the construction of the production system is less error prone.
Modules and Instances

Modules and instances are used for the support of programming in the large. Modules are templates describing the operation of functions around a common data structure. The objects imported to, and those exported from a module are described in the interface to a module by giving its name and the way the module treats the corresponding object. Modules have to be instantiated before the services they provide may be used. The result of such an instantiation is a value of type instance. In accordance with the philosophy of the language we do not specify the type of the imported or the exported values, so the polymorphism of procedures is carried a step further. Modules are somewhat similar to generic packages in ADA. Only after having instantiated a module the values being exported from a module may be used.

Modules and instances provide a data type of their own. This has as a consequence that modules may serve as parameters to procedures and may be returned from them as values. Since values of each type may be made persistent it is possible to deal with separate compilation as well as loading and binding of program units in a very flexible way. So a module is separately compiled by making it persistent, and an instance of a separate compiled and instantiated module is used by fetching the instance from the persistent store.

Modules provide a basis for applying evolutionary prototyping techniques like horizontal or vertical refinement, i.e. by completing existing procedures to their full functional extent or by adding further functions, respectively.

Persistence

Modeling 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 modeling data according to their semantic content; again, a set-oriented approach appears to be most natural: it is well accepted in the data base research community that data modeling should accommodate the user by making the representation and manipulation as close as possible to the user's perception of the problem.

Software prototyping will be most effective and have maximal impact when it caters for the modeling of programs and of data. Consequently we propose a facility for handling persistent data in ProSet. Persistence comes as a property orthogonal to types, so each and every value having a legal type in ProSet may be made persistent.

Programming Parallel Applications

The concept for process creation via Multilisp's futures [12] is adapted to set-oriented programming and combined with the concept for synchronization and communication via Linda's tuple space [11] in ProSet. Communication in Linda is based on the concept of tuple space, i.e. a virtual common data space, and also called generative communication. Reading access to tuples in tuple space is associative and not based on physical addresses — in fact, the internal structure of tuple space is hidden from the user. Reading access to tuples is based on their expected content described in so-called templates. Each component of a tuple or template is either an actual, i.e. holding a value of a given type, or a formal, i.e. a placeholder for such a value. A formal is prefixed with a question mark. Tuples in tuple space are selected by a matching procedure, where a tuple and a template are defined to match, iff they have the same structure (corresponding number and type of components) and the values of their actuals are equal to the values of the corresponding tuple fields. ProSet provides three tuple-space operations. The deposit operation deposits new tuples into tuple space, the fetch operation fetches and removes a tuple from tuple space according to a specified template, and the most operation meets and leaves a tuple in tuple space. It is possible to change the tuple's value while meeting it. There is no difference between ProSet-tuples and Linda-tuples. Linda and ProSet both provide tuples thus it is quite natural to combine them on the basis of this common feature.

Putting a communication language like Linda on top of a prototyping language like ProSet permits modeling parallel applications in an appropriate way. This approach has the advantage that no particular parallel or distributed architecture is assumed, and that no particular specific model of parallel processing is assumed. Linda is powerful enough to simulate the major current paradigms in parallel processing, so a language for prototyping rather includes such a general model than indulging in a specific approach to parallel processing which later on cannot be realistically reproduced when it comes to transform the prototype to a real program.
2.2 Examples

2.2.1 Depth First Search in Graphs

The following example will show some of the features mentioned above. We formulate a non-recursive algorithm for depth first search (dfs) in graphs.

A graph \( G = (V, E) \) is defined by a set \( V \) of nodes and a set \( E \) of (directed) edges. An edge \( e \in E \) has a starting node \( v \in V \) and a destination node \( w \in V \), i.e. a natural representation for edges is a pair \( (v, w) \). We will use this representation in our algorithm, too.

Algorithm dfs Figure 2 shows the algorithm for dfs. The input to the algorithm is a starting node \( s \) and the set \( E \) of nodes. It constructs as it goes the map \( \text{pred} \) mapping a node to is predecessor, and produces the tuple \( \text{order} \) which contains the nodes ordered in one of the possible depth first visiting orders. Inspection of the code shows that the program uses a set \( \text{used} \) of edges, and some auxilar nodes.

The tuple \( \text{order} \) is used as a queue: insertion is done at the end. The initialization in the first state- ment creates a tuple with the single element "a" and assigns this tuple to order. In the loop, (second line of if) the with:= assignment adds new elements at the end. A consumer may use the fromb-assignment \( \text{order} \); to dequeue elements from the beginning or otherwise iterate over the tuple to obtain its elements.

The set \( E \) of edges mentioned in the description and used in the code is a set of pairs. In \( \text{ProSet} \), a set \( s \) of pairs may be regarded as a finite single- or multi-valued map. A multi-valued map is a relation where a single element of the domain is related to an arbitrary number of elements in the range. Single-valued maps behave like discrete (and finite) functions: for each element \( x \) in the domain, there is at most one pair \( (x, y) \) contained in the set. The set of adjacent nodes to a node \( v \), i.e. the set \( \{w: (v, w) \in E\} \) is formulated as \( E(v) \).

The map \( \text{pred} \) is single-valued, i.e. all pairs in \( \text{pred} \) must have different first components. To indicate this, the syntax \( \text{pred}(v) \) is used to denote that a single image element is expected. Maintaining the map \( \text{pred} \) by initializing it to the empty set and assigning the predecessor of a node \( w \) to be \( v \) (first line of if-statement) ensures this integrity.

Another language construct visible here is the whilefound-loop. Its semantics is straightforward. Since in every step of the loop the iterator is evaluated newly, the predicate (making use of a set \( \text{used} \)) may yield different results. The altering of \( v \) will yield different values for \( E(v) \) to be used in the loop. This behavior makes depth first search possible. The \( \text{whilefound} \)-loop makes \( v \) going into depth, whereas the outer repeat-loop goes the opposite direction (by using the map \( \text{pred} \)).

Note that nothing is said about the representation of nodes — they can be of any type. The only restriction is that no node may be represented by the undefined value \( \text{om} \).

2.2.2 The Module Stack

We illustrate the above abstract ideas on modules, procedures, and exception handling with the well-known example of a stack. A module defining stacks and a selection of operations on them (functions top and is_empty are omitted to keep the example small) is shown in Fig. 3. Although it is possible to implement stacks of unbounded depth using \( \text{ProSet}'s \) data type \( \text{tuple} \), we will introduce here a boundary to obtain an overflow, in order to model a situation for exploring our exception handling mechanism.

Fig. 4 contains an instantiation of the stack module, i.e. the "holes" in the stack template are filled by providing the import parameters of the stack and an instance of the module is generated. Then we apply the \( \text{push} \) routine to put an element onto the stack. In terms of \( \text{ProSet} \) the stack is now an instance, i.e. an object with a state representing the stack itself. If in procedure \( \text{push} \) a stack overflow is detected, an exception is signaled to the actual routine in which that exception has been associated with a handler which is defined below. The handler resumes the \( \text{push} \) routine: executing \( \text{push} \) will be continued by substituting the top of the stack by the actual argument of \( \text{push} \). The second exported routine \( \text{pop} \) also uses exception handling but prevents itself from resuming execution by escape: invoking \( \text{pop} \) on an empty stack terminates popping in every case, here caused by an implicitly associated default handler.

Note that our stack operations work on every arbitrary element type. The stack may be heterogeneous, due to our polymorphic typing principles.

2.2.3 Persistence

Now let us consider a little bit closer the integration of persistence into \( \text{ProSet} \), guided by some examples. Persistent values are kept in data structures called \( \text{P-files} \), faintly resembling archives under UNIX, which in turn are identified in a program through strings. In addition to the value itself a \( \text{P-file} \) stores further
program dfs;
constant a := "a", b := "b", c := "c";  -- We use 3 nodes a, b, c
constant E := {{a,b}, {b,c}, {a,c}};  -- a is the starting node
begin
  order := [a];  -- the starting node is visited first
  used := { };  
  pred := { };  
  v := a;  
  repeat
    whilelfound w in E(v) | {v,w} notin used do
      used with := {v,w};  -- mark edge as used
      if vnotin order then
        pred(v) := v;  
        order with := w;  -- append at the end
        v := w;  -- go into depth
      end if;
    end whilelfound;
    v := pred(v);  -- if all edges are used, 
    continue with predecessor
  until v = omend repeat;
  put("order: ");
  put(order);
end dfs;

Figure 2: Depth First Search in Graphs

information, e.g. access rights and time stamps, and a name for the identification of the value.

The first example (Fig. 5) deals with making a value persistent. We declare the value accessed through the identifier Sort as persistent together with an indication that it will be taken from the P-file denoted by the string literal "MyProject.Utilities". Suppose the P-file exists, the user has the appropriate access rights, and there exists no value with name Sort in this P-file. Then a write lock is set to prevent any other program from accessing this P-file entry and a signal exception MissingPersistentValue is raised. Therefore, we have to associate a handler with this exception, which terminates with resume (otherwise the predefined default handler would abort the program). Then Sort is inserted into the P-file as having the undefined value om. When leaving the program Demo, the value of Sort is written to the P-file. Note that we cannot directly make the procedure MySort persistent, since procedures do not have first class rights. The example demonstrates also the possibility of separate compilation.

The next example (Fig. 6) illustrates the use of some existing persistent values. In addition to the function accessed through Sort, we use a set being stored in the P-file University.Home under the name PhoneList to model a simple phone-book. When the program is executed the persistent values are loaded as a constant and a variable, respectively. The program reads a name and a phone number from standard input, adds them to PhoneList and sorts this list. It makes use of the persistent sorting routine Sort, demonstrating the use of persistent functions. Note that Sort is passed the (closure of the) comparison procedure lt as a parameter. Upon program exit, i.e when leaving the range within which the persistent declarations are encountered, the modified value of PhoneList is written back to the persistent store.

This provides only a very sketchy picture of persistence. Combined with the powerful facilities of finite set theory, persistence will permit an adequate semantic modeling of data, thus providing a link between software prototyping and semantic data modeling. Consequently, data base issues may be formulated in ProSet, provided the data base is not too large (this limitation is due to performance restrictions). This applies in particular to program components like procedures, modules, and instances which may be in-
module stack (rd max, wr push, pop);
visible LocalStack;
begin
  LocalStack := [];  -- Imports a maximal depth
  -- The data structure stack.
  -- Initialization part.
procedure push(x);
begin
  if #LocalStack = max then
    signal StackOverflow();
  pop(LocalStack);
  end if;
  LocalStack with:= x;
end push;

procedure pop(wr t);
begin
  if LocalStack = [] then
    escape EmptyStack();
  else
    t frome LocalStack;
  end if;
end pop;
end Stack;

Figure 3: Module Maintaining Stacks

terchanged freely among applications through a persistent store. In addition, libraries of PROSET components become feasible and may be accessed through the language itself, hence the architecture of programs may easily be described in PROSET. This is an added benefit in prototyping, helping to elucidate the structure of a particular software solution.

2.2.4 The Queens' Problem Revisited

In section 1.3 the queens' problem was introduced together with a sequential solution. In Fig. 7 a parallel solution based on the master-worker model is given. It is recommended to examine the sequential solution in Fig. 1 again.

Our program employs the master-worker model where one master process interacts with a collection of identical workers. The master generates task tuples (positions) and collects results while the worker processes repeatedly grab tasks from tuple space and perform the required actions.

The master program uses explicit loops, whereas in the sequential program no explicit loops are needed and thus the parallel solution seems to have a lower level of abstraction than the sequential one. Such observations are often made in a wide spectrum language like PROSET, where programs may be transformed within the language using lower-level constructs to increase efficiency.

3 Implementation

The first version of the PROSET compiler is written in highly portable ANSI C on a Sun Sparc machine. Portability, efficiency, and wide availability suggest using ANSI C also as target language being further translated and linked with the runtime library by a C compiler. Since PROSET's powerful concepts and constructs are not supported directly by C or the target platform, the design of the runtime system is a non-trivial task.

Organisation of the Compiler The overall structure of the compiler is shown in Fig. 8. The implementa-

---

1 In a wide spectrum language it is possible to program on a high level of abstraction as in Fig. 1 as well as on a level of e.g. PASCAL.
my-stack := instantiate closure stack
rd max := 10;
wr push;
wr pop;
end instantiate;

my-stack.push(stack-elem) when StackOverflow use Substitute;

my-stack.pop();

handler Substitute;
begin
  resume;
end Substitute;

Figure 4: Using the Stack Module
program Demo;
  visible persistent Sort : "MyProject.Utilities"
  -- declaration of a persistent value accessed via Sort.

  when MissingPersistentValue use NewMember;
  -- association of the handler NewMember to the exception
  -- MissingPersistentValue: the non-existence of a value
  -- identified by Sort in the P-file results in raising the
  -- signaled exception MissingPersistentValue.

begin
  Sort := closure MySort; -- yields a value of type function having.
  -- first class rights.

  handler NewMember(); -- Handling the exception MissingPersistentValue
  begin
    -- with resume leads to an initialization of
    resume; -- Sort to omega.
  end NewMember;

procedure MySort(rd Compound, rd LessThan);
begin
  -- your favorite sorting routine returning the result as a sorted tuple.
  end MySort;
end Demo;

Figure 5: An example for separate compilation.

auxiliary functions supporting more or less technical
particulars, e.g. the iteration over composite objects.

Persistent Store The implementation of the persis-
tent store is based on an object management system
H-PCTE [14]. This system is a simplified version of
ECMA-PCTE and a high performance implementa-
tion of it. The goal to maximize performance leads to
an implementation as main (or virtual) memory object
base.

Currently we have implemented on this basis a
single-user persistent store. Since H-PCTE is struc-
tural object oriented, we use a dynamic link editor
to load values of higher order type. The next steps of
our implementation deal with the extension to a multi-
user store, the distribution on a LAN, as well as the
construction of a graphical browser for the persistent
store.

4 Further Developments

We want to briefly sketch our plans for the program-
ing environment which is a necessary addition to any
language supporting software prototyping. We outline
some components of the environment (most of which
are already under construction) in 4.2, and it should
become clear from this discussion that the language
itself is an interesting object of study. One particular
aspect is type inference and data structure selection.
Both problems are somewhat intertwined, and we dis-

cuss issues pertaining to these questions in 4.1.

4.1 Type Inference and Data Structure
Selection

Weak Type System The intention of ProSet to
be designed as a language for the support of rapid
software prototyping is also reflected in the under-
lying type system. The software developer does not
have to declare the types of the objects used in a
ProSet program. This makes him free of burden-
some routine work and also gives him the necessary
flexibility needed during the process of modeling al-
gorithms and data. In addition to declaration free-
ness, the iteration over non-homogeneous compound
data objects and the desirable feature of allowing user-
defined polymorphic procedures in an imperative pro-
gramming language may change the type of variables
during runtime.

A consequence of this flexibility in the usage of the
program AddPhone;
persistent constant Sort : "MyProject.Utilities";
visible persistent PhoneList : "University.Home";
begin
  putf("Enter name: "); -- read the first name as string
  Name := getf("\n");
  putf("Enter phone: "); -- read the phone as integer
  Phone := getf("%d");
  PhoneList with := [ Name, Phone ]; -- add the new item to the list

  for p in Sort(PhoneList, closure lt) do -- print the entries of the
    -- list sorted by name.
    putf("Name: %20s Phone:%d\n", ~(1), ~(2));
  end for;
procedure lt(x, y);
begin
  return x(1) < y(1);
end lt;
end AddPhone;

Figure 6: An example for the use of persistent values.

prototyping language is a weak type system. Hence ProSET programs cannot statically be checked for type correctness in general and even erroneous programs can partially be executed until the flow reaches the type error. It follows that the user has to pay for this

- in less efficiency w.r.t. both storage and execution time — the possibility of changing the type of a variable during runtime forces the implementation to use union types for ProSET values and to add code for dynamic type checks to the executing program.
- in more expensive during testing phases — in larger programs there are often parts which are rarely executed, so the program may be in use for a long time before dynamic checking detects a type error.

This dilemma can be softened by a static type inference mechanism.

Type Inference Type Inference in applicative programming languages is a well known problem, and there are efficient unification algorithms to detect type errors in the structure of types. In ProSET the following situations are observed:

- Since type changing during runtime is allowed, there are situations in which more than one type leads to a correct typing. So type inference must be extended to assign a whole set of correct typings to expressions. Sets of types are not unifiable because the members may appear in any arbitrary order. Thus unification has been extended by a more general algorithm for set intersection, which supplies the program with a weak principle type, i.e. the set of correct typings.
- Since ProSET is an imperative programming language, side effects of procedures to global variables are possible. This implies that type inference cannot be performed on the syntax tree of the source program alone. It has to take the data flow into account, and propagate the analyzed type information of expressions through the flow graph until a fixed point is reached. This data flow oriented type inference algorithm is more flexible but less efficient than unification.

Data Structure Selection for Compound Data Objects The compound data types set and tuple in ProSET are supported by many operations, but efficient support for all operations using only one representation is not possible. For instance a linear list implementation of the data type tuple efficiently supports the insertion of an element into the tuple, but
program ParallelQueens;
  constant N := 8, NumWorker := argv(2), -- program argument
  TS := createTS(om); -- dynamically created tuple space
begin
  Positions := npow(N, {[x,y]: x in [1..N], y in [1..N]});

  for i in [1 .. NumWorker] do
    || Worker(TS); -- spawn the worker processes
  end for;

  deposit [ {} ], [ 0 ] at TS end deposit; -- initialize the result set
  for NextPosition in Positions do
    deposit [ NextPosition ] at TS end deposit;
  end for;

  fetch ( #Positions ) at TS end fetch;
  fetch ( ? NonConflict |(type $ = set) ) at TS end fetch;
  put (NonConflict);

  procedure Worker (TS); begin
    loop
      fetch ( ? MyPosition ) at TS end fetch;
      if NonConflict (MyPosition) then -- add to the result
        meet ( ? |(type $ = set) into ($ with MyPosition) )
        at TS
        end meet;
      end if;

      meet ( ? |(type $ = integer) into ($ + 1) ) at TS end meet;
      end loop;
    end Worker;
end ParallelQueens;

Figure 7: Parallel solution for the queens' problem.
See Fig. 1 for the procedure NonConflict. The resulting set of non-conflicting positions is built in tuple
space via changing meet operations. The master program spawns NumWorker worker processes. This
number is an argument to the main program. The resulting set of non-conflicting positions is built in
tuple space via changing meet operations. The counter in tuple space is necessary to let the master wait
until all positions are evaluated. The unary operator # returns the number of elements in a compound
data structure. The symbol $ is a placeholder for the corresponding tuple-component in tuple space.

Figure 8: The structure of the compiler.
a sublinear membership test for a special element or an efficient access to the \(i\)th component is not possible with this data structure. This implies that the default data structures used for the implementation of the data types \texttt{set} and \texttt{tuple} should balance the requirements for all operations. When type inference is performed and leads to exactly one correct typing for a variable, overloading can be resolved and the usage of this variable can be analyzed to select a more efficient data structure from a library of predefined implementations for this particular program. This selection process is based on the following informations:

- the operations which are applied to that object in this particular program.
- the frequency of those operations: If it is not possible to support efficiently all operations, the support of operations used in deeply nested loops should be preferred.
- the inclusion relations between structured objects: With these information the universe of all objects related to each other can be chosen as a base for the implementation. Each object can be represented over this base as a bit vector, list of pointers to the base, etc. This prevents multiple storage of the members and allows to support different operations by different implementations (see [15]).

It should be noticed that the whole program has to be analyzed before data structure selection can be performed. Whatever two structured objects are used as operands of one and the same binary operation, the same implementation has to be selected for these objects — otherwise combinatorial explosion is inevitable.

4.2 Programming Environment and Further Work

Our work has concentrated mostly on language design and the implementation proper. We estimate that the implementation will be complete by the end of 1992, so that we may concentrate on the design and realization of tools for the programming environment. In closing the paper we will provide a brief glimpse at some of the tools to be constructed for technically supporting the programmer’s task of creating a satisfactory model of an application.

Transformational issues Using finite differencing as a transformational paradigm [8] has shown surprising results in particular with respect to improving the performance of programs asymptotically. We are in the process of adopting the approach to our current environment.

Abstract data types The designer cannot use abstract data types for protecting data against unsuitable operations, since abstract data types cannot be formulated. We need a powerful mechanism for formulating ADTs, supporting in this way semantic data modeling.

Support for persistence Persistent values are maintained in a structure called \(P\text{-File}\). Working with these archives should be supported through tools for

- Browsing: display the contents of a \(P\text{-File}\) graphically, show the interconnections and interdependencies between various items in a \(P\text{-File}\) etc.
- Module interfaces: modules lack an interface allowing a static check of consistency with respect to import and export. This is due to the language’s philosophy of not restricting the user through the necessity of type declarations. This violates type security considerably, and we currently investigate ways of providing weak type checks eliminating most typing errors at run time.

Architectural description Since functions, modules and instances are first class citizens of the language, and since binding as well as loading can be described in the language itself via the persistence mechanism, the design of an application may be described in the language. We will investigate which process models are appropriate in the context we are envisioning, and how to formulate them in PROSET; our first impression is that an approach like MERLIN [16] using backward chaining and forward propagation over a persistent knowledge base might be suitable.

Support for reusing program parts Prototypes written in \texttt{PROSET} provide their functionality on a very high semantic level, hence it is more feasible to recognize what they are doing than for programs written in a production language like C. We are currently gaining some experience with tools permitting the identification of program components based on Prieto-Diaz’s faceted classification scheme [17]. These tools will be incorporated into the prototyping environment.
Literal programming  Documentation usually serves as the token that is passed between the designer and the user of a system, describing the intended functionality. Literal programming unifies documentation and code in one single document from which either may be generated. This may be a way of demonstrating to the customer where the desired functionality is realized in a program system, enabling her (or him) to communicate in a more versatile and competent way with the system designer. We plan to experiment with this approach to communication in the context of prototyping.

The user interface  The user interface requires particular attention, and its construction should be supported by suitable tools. We consider currently constructing an interface to the toolkit DIWA [18] which allows specifying user interfaces on a sufficiently high level.

References


