# A First Implementation of ProSet

Ernst-Erich Doberkat, Wolfgang Franke, Ulrich Gutenbeil
Wilhelm Hasselbring, Ulrich Lammers, Claus Pahl

University of Essen
Computer Science / Software Engineering
Schützenbahn 70, 4300 Essen 1, Germany
pst@informatik.uni-essen.de

**Abstract**

The VHLL ProSet (Prototyping with Sets) has been designed at the University of Essen for the support of experimental and evolutionary prototyping. The language is based on set theory augmented by bits and pieces from λ-calculus. ProSet provides, among other features, higher order data types, flexible exception handling, generative communication, and the integration of persistence. This note provides a short overview of the first ProSet implementation. Included are both the compiler construction, being mostly supported by the compiler construction system Eli, and a description of the runtime system. Finally, we conclude with a brief glimpse at our current work constructing tools for a prototyping environment.

## 1 Review of ProSet

ProSet is a procedural, set-oriented very high level language (VHLL). Beneath the usual primitive data types *integer*, *real*, *boolean*, *string*, and *atom*, the language provides the compound data types *set* and *tuple*. Both have their usual mathematical semantics and are accompanied by the usual operations (intersection, union, concatenation, etc.). Elements or components of a compound data type need not have the same type. The control structures are similiar to those found in SETL [SDDS86] or SETL2 [Sny90]. The following features distinguish ProSet from other set-oriented languages:

- *Data abstraction* is supported uniformly by the data types *function*, *module*, and *instance*.
- *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 [Gel85] serve as a basis for some primitive operations in our language.

A detailed discussion of the design and motivation for ProSet can be found in [DFG+92].

## 2 Implementation

The first version of the ProSet compiler is written in highly portable Ansi C on Sun Sparcs. Portability, efficiency, and wide availability suggest using Ansi C also as target language being further translated by a C compiler and linked with the *runtime library*. 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.

## 2.1 Compiler

The compiler generation is supported by the compiler construction system Eli [GHL+92]. It integrates off-the-shelf tools and libraries with specialized language processors to provide a system for generating a complete compiler. The system derives an executable compiler from specifications of the structure of different program representations.

**Analysis** The first compilation task, the analysis of the source program, consisting of the usual subproblems lexical, syntactical, and semantical analysis, is very well supported by Eli. This enabled us to generate an executable front end at a very early phase of the implementation. Additionally, the front end could be easily extended by a translation of a subset of ProSet into SETL2. To a large extend the specifications for the analysis task are based on an attribute grammar written in LIDO [GHL+92]. This grammmar is derived from the abstract grammar of ProSet and also implicitly defines a decorated abstract syntax tree being the input of the next step.

**Transformation** To facilitate the mapping to C as well as to obtain a better modularization of the compiler, we perform as a next step a transformation into an intermediate language. ProSet is a wide spectrum language which allows programming close to the conceptual ideas of a solution and therefore on a high level as well as programming quite close to the machine comparable to Ada or C. This property is exploited by the definition of the intermediate language, called $\mu$ProSet, being essentially a subset of ProSet containing only the low level constructs of ProSet. Eli supports the translation to a target tree by providing tree building functions derived from the attribute grammar of the target language. These functions are used in the attribution of the source tree to build the target tree in a bottom up way. For our purposes, a more powerful transformation mechanism (e.g. transformation of attributed trees using pattern matching) would have been desirable, because the attribution mechanism is not flexible enough.

**Code Generation** The last task in the translation of ProSet to C is the generation of an Ansi C program. The generation of structured text is supported by the Eli tool PTG [GHL+92]. Since many of the high level constructs are transformed into equivalent lower level ones, the mapping to C is essentially one to one for $\mu$ProSet's statements and expressions. The main task of the code generation consists of the translation of nested procedures, modules, and exception handlers to the flat structure of C functions, preserving the scope rules. We have developed a *contour model* providing a conceptual basis for this task (cfg. [DFG+92], appendix A). The model reflects that ProSet as a block structured language is well suited to a stack implementation technique. However the availibility of higher order types and copy semantics require rather a kind of block retention strategy (notice that in ProSet procedures, modules and instances may access nonlocal objects; applying the **closure** construct on them freezes the bindings to these objects). For the sake of efficiency we have integrated both strategies into our model. Whenever possible we use a stack and only do retention, i.e. holding the freezed bindings in the heap, when values of higher order type are involved. This strategy is sometimes called *mixed mode strategy*. The model is feasibly implementable with techniques known as *structurization* and *extended parameter list* [Sun90].

## 2.2 Runtime System

The runtime system supporting the execution of ProSet programs consists of four components: the runtime library, an object management system implementing the persistent store, the transaction manager, and the tuple space manager. The implementation of concurrency, comprising the latter two components, is currently in progress.

**Runtime Library** The final executable output of the compiler is simply produced by linking the translated C programs and the runtime library with the standard link-editor. The current version of the runtime library provides only standard representations for ProSet's data objects. For example sets are implemented as hash tables. When data structure selection (section 3.1) will be integrated, specialized representations, e.g. bitvectors for sets, will be added. Furthermore the library contains functions corresponding to the predefined operations and some auxiliary functions supporting more or less technical particulars, e.g. the iteration over composite objects.

**Persistent Store** The implementation of the persistent store is based on an object management system called H-PCTE [Kel92]. The system is a simplified version of ECMA-PCTE and a high performance implementation 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 structurally 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.

## 3 Further Developments

Our work has concentrated on language design and the implementation proper. We estimate that the implementation will be mostly complete by the late summer of 1992, so that we may concentrate on the design and realization of tools for the programming environment. Section 3.1 details the work on type inference and data structure selection. In closing the overview 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.

### 3.1 Type Inference and Data Structure Selection

A consequence of ProSet's intent to be a language for the support of rapid software prototyping on a high expressive level is a weak type system: ProSet programs are dynamically checked for type correctness such that even erroneous programs can partially be executed until the control flow reaches the type error.

The advantages of static type checking, i.e. more efficiency w.r.t. both storage and execution time and error detecting early in the programming process will partially be retained by an automatic type inference mechanism including two steps: type inference on applicative language constructs is done by an unification algorithm, which had to be extended for union types, since in some situations more than one type leads to a correct typing. Afterwards the types inferred for expressions are minimized by propagating them through the program flow graph until a fixed point is reached.

When type inference leads to a monotype for a variable, all overloading operations can be resolved and more optimizations like data structure selection for that particular data object in that particular program can be performed. The inferred type together with the operations applied to that object, the frequency of those operations, and the relationship of that object to other objects in the program will be used to to select an efficient data structure from a library of predefined implementations. Notice that in contrast to other approaches to VHLL, ProSet gets along without a declaration mechanism like the Data Representation Sub Language in SETL [SSS81] or the Constraints in V [GK83] — all informations needed to choose an efficient representation for data objects are analyzed from the program text [Dob91].

## 3.2 Further Work

**Transformational issues** Using finite differencing as a transformational paradigm [PK82] has shown convincing 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 specifying and implementing ADTs, supporting in this way semantic data modeling.

**Support for persistence** Persistent values are maintained in a structure called $P-File$. Working with these archives should be supported through tools for *browsing* (display the contents of a $P-File$ graphically, show the interconnections and interdependencies between various items in a $P - File$, modify items in a $P - File$, etc.).

**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.

**Support for reusing program parts** Prototypes written in 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 [PDF87]. 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 [Vos90] which allows specifying user interfaces on a sufficiently high level.

## References

[DFG+92] E.E. Doberkat, W. Franke, U. Gutenbeil, W. Hasselbring, U. Lammers, and C. Pahl. ProSet — Prototyping with Sets: Language Definition. Informatik-Bericht 02-92, University of Essen, April 1992.

[Dob91] E.E. Doberkat. Type Checking and Data Structure Selection for Mini-ML with Sets. Informatik-Bericht 04-91, University of Essen, September 1991.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[GHL+92] R.W. Gray, V.P. Heuring, S.P. Levi, A.M. Sloane, and W.M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–131, February 1992.

[GK83] A. Goldberg and G. Kotik. Knowledge based programming: An overview of data structure selection and control structure refinement. In H.L. Hausen, editor, *Symp. Software Validation*, pages 287–309, Darmstadt, West Germany, 1983. North Holland.

[Kel92] U. Kelter. H-PCTE – A High Performance Object Management System for System Development Environments. Internal report 128, University of Hagen, FernUniversität-GHS-Hagen, Feithstr. 140, D-5800 Hagen, May 1992.

[PDF87] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 4(1):6 – 16, 1987.

[PK82] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Trans. Prog. Lang. Syst.*, 4(3):402 – 454, 1982.

[SDDS86] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets – An Introduction to SETL*. Graduate Texts in Computer Science. Springer-Verlag, 1986.

[Sny90] W.K. Snyder. The SETL2 programming language. Technical Report 490, Courant Institute, New York University, September 1990.

[SSS81] E. Schonberg, J.T. Schwartz, and M. Sharir. An Automatic Technique for Selection of Data Representations in SETL Programs. *ACM Trans. on Prog. Lang. and Systems*, 3(2):126–143, Apr. 1981.

[Sun90] N. Sundaresan. Translation of Nested Pascal Routines to C. *SIGPLAN Notices*, 25(5):69–80, 1990.

[Vos90] J. Voss. *Entwurf und Implementierung von graphischen Benutzeroberflächen: Ein integrierter, objektorientierter Ansatz*. PhD Thesis, University of Hagen, Hagen, Germany, 1990.