Specialization, Generalization and Inheritance Teaching Objectives Beyond Data Structures and Data Types

Wilf R. LaLonde
John R. Pugh

SCS-TR-57
August 1984

This research was supported by the Natural Sciences and Engineering Research Council of Canada.

Specialization, Generalization and Inheritance
Teaching Objectives Beyond Data Structures and Data Types

Wilf R. LaLonde and John R. Pugh
School of Computer Science
Carleton University
Ottawa, Ontario, Canada K1S 5B6

Abstract

This paper describes our experience in teaching a course in abstract data types over a four year period. Although abstract data types are a major improvement over traditional data structures, they provide only a portion of the concepts needed by experienced designers. These additional concepts include such notions as generalizations of data types, their specializations, relationships between data types, and the capability and necessity of abstracting common operations through inheritance mechanisms. Designing an individual abstract data type is akin to "programming in the small" but designing a library of data types introduces new problems that are best solved from the viewpoint of "programming in the large".

1 Introduction

Computer Science departments are in the process of replacing outmoded courses in data structures by more fashionable courses on abstract data types. Such courses have a strong practical flavour and place a strong emphasis on software engineering and structured design principles. Unfortunately, the introduction of such courses has been limited by two restrictions:

1. The lack of textbooks devoted to the topic. As a rule, comprehensive texts treat only data structures in detail. Nevertheless, there are some books providing an introduction to the ideas of data types; e.g., Coleman's[2] "A Structured Programming Approach to Data" and Aho, Hopcroft, and Ullman's[1] "Data Structures and Algorithms".

2. The lack of suitable language processors. The introduction of Modula[8], Ada[7] and Smalltalk[5] processors is a recent development.

These restrictions were clearly in effect when we introduced an advanced course in abstract data types four years ago. Computer Science 202 is a required course for second year students in the Computer Science honours program. Students arrive in the course with a full year of programming in Pascal behind them, and it is fair to describe them as lacking in programming experience but reasonably fluent in Pascal, structured program design, and simple data structures.

When the course was first introduced, we adopted a modified Pascal as a didactic pseudo language and gradually switched over to Ada as it became popularized. As our experience grew, so did our realization of the lack of support for fundamental notions that were better provided by a more powerful language. Although not itself the final answer, we grew to accept Smalltalk[5] (in fact, a variant with a more traditional syntax) as providing a better paradigm for unifying these important concepts.

These concepts include such interrelated notions as generalizations of data types, their specializations, relationships between data types, and the capability and necessity of abstracting common operations through inheritance mechanisms. Although we initially resisted the temptation to include these notions, it became clear that a comprehensive treatment required them. Designing an individual abstract data type is akin to "programming in the small" but designing a library of data types introduces new problems that are best solved from the viewpoint of "programming in the large"[3].

We begin by reviewing data types and then we consider each of the above notions in some detail. Finally, we discuss the evolution of the course.

2 Data Types

The use of data types as an effective way of modularizing large systems, producing reusable code, and putting into practice the principle of information hiding, has received broad acceptance by the Computer Science community. However, there is still little consensus on a suitable definition for data types. For instance, our notion differs from that

provided by Aho, Hopcroft, and Ullman[1]. Briefly, we view

1. a **data type** as a set of objects and a set of operations on the objects, and
2. a **data structure** as a data type in which the **representation** is directly manipulatable.

Data structures violate the information hiding principle by providing access and modification operations too specific to the representation. For instance, a complex number data type could provide operations to access the real part and the imaginary part without divulging the representation. A complex number data structure, on the other hand, allows the user to access the Real_Part field of the representation if it happens to be a record, or the element subscripted by Real_Part (or worse, a subscript such as 1) in the event that the representation is an array.

There is also little agreement on a definition for the term **abstract data type**. For instance, is there a difference between a data type and an abstract data type? One view is that abstraction is a summarization mechanism which allows the highlighting of some information and the suppression of less important information. Applying this view to data types, we would say that an abstract data type summarizes the external or users view of a data type while hiding the implementation details. Another approach is to view data types as a concrete realization of a more formal mathematical notion termed an abstract data type. With this distinction, an abstract data type can have many concrete realizations through different implementations. Some may argue that the distinction emphasizes the notion that a concrete data type is always an approximation to an abstract data type citing as an example, the integer abstract data type which has no bound on the size of an integer and a concrete data type, such as a Pascal integer, which does. Unfortunately, this reasoning is incorrect; corresponding to the integer abstract data type is an integer concrete data type (the Smalltalk integer is an example – it has no bound) whereas corresponding to a small integer abstract data type is a small integer concrete data type (the Pascal integer is an example). To the programmer, the distinction between a data type and an abstract data type is academic since neither provides access to the implementation (assuming, it hasn't degenerated to a data structure). For simplicity, we prefer not to distinguish the two.

3 Generalization

Data types are **generalized** by removing restrictions. Its not enough to study a data type in isolation. One must also ask whether a more general version can be created, whether this more general version (if it exists) is useful, whether or not it can be generalized along different dimensions, and whether these other generalizations can themselves be generalized. Once generalizations have been considered, appropriate implementation strategies can be contemplated.

To reconsider the above example, asking an Ada programmer if s\he can generalize Ada integers ought to provoke the obvious response: "Yes, make them unbounded" and perhaps even the suggestion to use lists of Ada integers as the representation. The natural tendency is to call them Unbounded_Integers but this only points out that Ada (and Pascal, ...) integers are misnomers. Proper name design would require a better choice of names; e.g., Small_Integer and Integer respectively. Even name design should be done "in the large".

For a more traditional example, consider the overused stack data type. Almost everyone who takes a data types course already knows what it is; they are familiar with typical stack operations. Yet, unless experienced at generalizing data types, few would conceive of a most general stack data type as one permitting both of the following:

1. An unbounded number of elements.
2. Unrestricted element types.

In terms of accessible programming languages, only Smalltalk permits the definition of such a general data type. Ada, for instance, only allows a generator of special cases to be defined (the generic facility) where each special case consists of prespecified element types. It is not possible to define a variable of type stack which can be made to contain values of **any** type.

4 Specialization

The converse of generalization is specialization. A data type can be **specialized** by imposing additional restrictions. This can be achieved, for example, by introducing new

operations, by considering only a proper subset of the instances, by providing additional space/time performance requirements.

The notion is particularly important because it emphasizes the commonality of a data type and its specialization. It also helps to highlight the differences because this issue is a central focus. In the event that distinct data types happen to have common specializations, it provides a better understanding of the relationships between the different data types.

Few books are aware of the issue. As an illustration, consider stacks once again. Almost all presentations provide a definition with the usual set of operations and then proceed to implementation issues. They highlight the fact that stacks can be implemented in two traditional ways: using arrays and using lists. This is fine as far as it goes. The error is in not pointing out that each is most naturally associated with distinct but related data types. In particular, lists provide a reasonable representation for unbounded stacks. Arrays, however, are a better representation for the specialization, bounded stacks. These are distinct data types: bounded stacks must provide an operation for determining whether or not a stack is full. Additionally, it is not incorrect to implement bounded stacks with lists but arrays are more appropriate. Representations must be chosen to best match the requirements.

To additionally illustrate the flavour of specialization and to provide a sampling of the methods for achieving it, we consider a brief series of examples:

1. Integers specialized to Small_Integers – the introduction of additional operations for determining minimum and maximum small integers (also, a different representation).

2. Arrays specialized to sparse arrays – no additional operations but a change in representation to take into account space considerations and expected usage.

3. Strings specialized to strings of characters – a change in representation to support the specified subset efficiently.

4. Polygons to regular convex polygons (equal sides) – potentially a change in representation but more important, additional operations such as an operation for obtaining the circumscribing circle (not possible for polygons in general).

5 Inheritance

By virtue of being related, different data types can share portions of their code and/or data. If A is a source of code and/or data that B wishes to share, then any mechanism that successfully realizes this sharing is termed an **inheritance mechanism. Code inheritance** denotes a code sharing mechanism whereas **data inheritance** denotes one that shares data.

The need for inheritance mechanisms (at least for expositional purposes) arises naturally from generalization and specialization considerations. If two data types are related, many operations are likely to be common, especially those operations that are **nonprimitive**; i.e., implemented in terms of a small subset arbitrarily specified to be **primitive**. If a specialization has the same representation as the more general case, simple data inheritance can provide the sharing mechanism.

More complex inheritance mechanisms can also be introduced. A data type which is the union of existing data types can share the code of the various types through **multiple inheritance**. For example, in a graphical simulation of ships passing through a canal system, data types Ship and Displayable_Object might be provided. Ships might have properties such as Weight, Length, Speed, Class, ... and Displayable_Objects might have Position, Iconic_Representation, Screen_Extent, ... along with operations such as Display_At and Translate_By. With multiple inheritance, one could easily create Displayable_Ship as a data type whereby the representation and the operations are obtained by merging the respective representations and operations of the base types.

Instances of the same type with different representations share identical operations with distinct implementations (**multiple representations**). An example of such a situation can be provided in the context of a list data type. An empty list can be represented totally differently from non-empty lists. The latter must accommodate a first element and the remaining elements. The former does not. Moreover, the First operation for an empty list should report an error (empty lists don't have a first element); the same operation for a non-empty list should provide the specified element. Traditional implementations provide a common representation with one first operation that must distinguish the two cases at execution time.

These issues ultimately point out the simplication inherent in the notion that an instance has a single unique type. For example, the value 0 can be viewed primarily as an instance of type Small_Integer. Nevertheless, it is also of type Integer, Rational, Number, ... Likewise, instances of Reals are also Numbers. More general schemes allow the instances themselves to specify their type (or types).

6 Course Evolution

To take the above notions into account, the course evolved drastically over the four years. Prior to its introduction, data structures were the focus and Pascal was a suitable didactic language. As data types became the focus, a more powerful language such as Ada became the model. However, with the introduction of the above concepts into the curriculum, Smalltalk became a better choice.

In particular, Smalltalk already provides facilities for

1. Constructing general data types such as stacks, queues, lists, sets, mappings, etc. Each of these can serve as "containers" for objects of arbitrary and often distinct types.
2. Specializing data types and providing for common operations through method inheritance (Smalltalk operations are termed **methods**).
3. Allowing data types to be organized around a type hierarchy making it obvious that instances can be simultaneous members of several types.

On the other hand, it is not a complete solution. Though available in experimental versions, multiple inheritance is not supported in standard Smalltalk. Neither is the ability to provide distinct instances with different representations. More important, the syntax is unusual (at least initially) and practical processors are only now becoming available[6].

Our current approach is to use a pseudo language with a more conventional Ada-like syntax, but considerably simplified, and also extended to include methods and distinct inheritance mechanisms. Pascal, Ada, and Smalltalk are viewed as implementation languages with different expressive powers. Due to the fact that Pascal is the primary teaching language, assignments are usually oriented to designing specializations of the more important

data types. This approach ensures that design and organization is not restricted by the peculiarities and restrictions of the available programming languages.

Overall, experience with the course is positive. Students now tend to

1. Think beyond the trappings of the available programming languages.
2. Consider design from the broader perspective of data type libraries.
3. Analyse relationships between similiar data types.
4. Consider implementation strategies to satisfy data type requirements rather than attempt to design data types that revolve around a preconceived implementation.

Two other positive by-products of the course are that students now appreciate

1. The expressive limitations of available programming languages.
2. The virtues of re-usable code.

After completing the course, students asked to define "arrays" no longer reply with the narrow view provided by typical books on Pascal programming. They tend to ask themselves questions such as

1. Do the elements types have to be the same type?
2. Must integer subscripts be contiguous?
3. Must the subscripts be integers? Are reals or strings acceptable subscripts?
4. Must the subscripts consist of instances of the same type?
5. Can the number of subscripts vary for the same array?

The fact that such questions are now asked points out the advantage over the more traditional approaches to teaching data structures or even the new approaches based on abstract data types "in the small".

7 Conclusions

We believe that a data types course is lacking if it does not discuss the notions of generalization, specialization and inheritance. Our experience is that a course dealing with these concepts can be successfully taught with existing programming languages, the lack of a suitable textbook continues to be a source of concern.

Although our notes are expanding, we found the important ideas and concepts outstripping our ability to maintain up-to-date notes. In fact, a proper book borders closely on the research fringe. The magnitude of the effort also rivals the effort needed to design and implement a sophisticated support library as has been provided in the Smalltalk environment.

8 References

1. Aho, A.V. et al,*Data Structures and Algorithms*, Addison-Wesley, Reading, Massachussettts, 1983.
2. Coleman D., *A Structured Programming Approach to Data*, Macmillan Press, 1978.
3. DeRemer F., and Kron, H., *Programming-in-the-Large Versus Programming-in-the-Small*, IEEE Transactions on Software Engineering, SE-2, June 1976, pp 80-86.
4. Feldman, M.B., *Abstract Data Types, Ada Packages, and the Teaching of Data Structures*, Proceedings of the 15th SIGSCE Technical Symposium on Computer Science Education, Philadelphia, Penn, February 1984.
5. Goldberg, A. and Robson, D., *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.
6. Krasner G., *Smalltalk-80: Bits of History, Words of Advice*, Addison-Wesley, 1983.
7. United States Department of Defense, *Ada Programming Language*, ANSI-MIL-STD-1815A, January 1983.
8. Wirth N., *Programming in Modula-2*, Springer-Verlag, New York, 1982.

CARLETON UNIVERSITY

School of Computer Science

BIBLIOGRAPHY OF SCS TECHNICAL REPORTS

SCS-TR-1
THE DESIGN OF CP-6 PASCAL
Jim des Rivieres and Wilf R. LaLonde, June 1982.

SCS-TR-2
SINGLE PRODUCTION ELIMINATION IN LR(1) PARSERS: A SYNTHESIS
Wilf R. LaLonde, June 1982.

SCS-TR-3
A FLEXIBLE COMPILER STRUCTURE THAT ALLOWS DYNAMIC PHASE ORDERING
Wilf R. LaLonde and Jim des Rivieres, June 1982.

SCS-TR-4
A PRACTICAL LONGEST COMMON SUBSEQUENCE ALGORITHM FOR TEXT COLLATION
Jim des Rivieres, June 1982.

SCS-TR-5
A SCHOOL BUS ROUTING AND SCHEDULING PROBLEM
Wolfgang Lindenberg, Frantisek Fiala, July 1982.

SCS-TR-6
ROUTING WITHOUT ROUTING TABLES
Nicola Santoro, Ramez Khatib, July 1982.

SCS-TR-7
CONCURRENCY CONTROL IN LARGE COMPUTER NETWORKS
Nicola Santoro, Hasan Ural, July 1982.

SCS-TR-8
Out of Print
ORDER STATISTICS ON DISTRIBUTED SETS
Nicola Santoro, Jeffrey B. Sidney, July 1982.

SCS-TR-9
OLIGARCHICAL CONTROL OF DISTRIBUTED PROCESSING SYSTEMS
Moshe Krieger, Nicola Santoro, August 1982.

SCS-TR-10
COMMUNICATION BOUNDS FOR SELECTION IN DISTRIBUTED SETS
Nicola Santoro, Jeffrey B. Sidney, September 1982.

SCS-TR-11
SIMPLE TECHNIQUE FOR CONVERTING FROM A PASCAL SHOP TO A C SHOP
Wilf R. LaLonde, John R. Pugh, November 1982.

SCS-TR-12
EFFICIENT ABSTRACT IMPLEMENTATIONS FOR RELATIONAL DATA STRUCTURES
Nicola Santoro, December 1982.

SCS-TR-13
ON THE MESSAGE COMPLEXITY OF DISTRIBUTED PROBLEMS
Nicola Santoro, December 1982.

SCS-TR-14 **A COMMON BASIS FOR SIMILARITY MEASURES INVOLVING TWO STRINGS**
R. L. Kashyap and B. J. Oommen, January 1983.

SCS-TR-15 **SIMILARITY MEASURES FOR SETS OF STRINGS**
R. L. Kashyap and B. J. Oommen, January 1983.

SCS-TR-16 **THE NOISY SUBSTRING MATCHING PROBLEM**
R. L. Kashyap and B. J. Oommen, January 1983.

SCS-TR-17 **DISTRIBUTED ELECTION IN A CIRCLE WITHOUT A GLOBAL SENSE OF ORIENTATION**
E. Korach, D. Rotem, N. Santoro, January 1983.

SCS-TR-18 **A GEOMETRICAL APPROACH TO POLYGONAL DISSIMILARITY AND THE CLASSIFICATION OF CLOSED BOUNDARIES**
R. L. Kashyap and B. J. Oommen, January 1983.

SCS-TR-19 **SCALE PRESERVING SMOOTHING OF POLYGONS**
R. L. Kashyap and B. J. Oommen, January 1983.

SCS-TR-20 **NOT-QUITE-LINEAR RANDOM ACCESS MEMORIES**
Jim des Rivieres, Wilf LaLonde and Mike Dixon,
August 1982, Revised March 1, 1983.

SCS-TR-21 **SHOUT ECHO SELECTION IN DISTRIBUTED FILES**
D. Rotem, N. Santoro, J. B. Sidney, March 1983.

SCS-TR-22 **DISTRIBUTED RANKING**
E. Korach, D. Rotem, N. Santoro, March 1983.

SCS-TR-23 **A REDUCTION TECHNIQUE FOR SELECTION IN DISTRIBUTED FILES : I**
N. Santoro, J. B. Sidney, April 1983.
Replaced by SCS-TR-69

SCS-TR-24 **LEARNING AUTOMATA POSSESSING ERGODICITY OF THE MEAN : THE TWO ACTION CASE**
M. A. L. Thathachar and B. J. Oommen, May 1983.

SCS-TR-25 **ACTORS - THE STAGE IS SET**
John R. Pugh, June 1983.

SCS-TR-26 **ON THE ESSENTIAL EQUIVALENCE OF TWO FAMILIES OF LEARNING AUTOMATA**
M. A. L. Thathachar and B. J. Oommen, May 1983.

SCS-TR-27 **GENERALIZED KRYLOV AUTOMATA AND THEIR APPLICABILITY TO LEARNING IN NONSTATIONARY ENVIROMENTS**
B. J. Oommen, June 1983.

SCS-TR-28 **ACTOR SYSTEMS: SELECTED FEATURES**
Wilf R. LaLonde, July 1983.

SCS-TR-29 ANOTHER ADDENDUM TO KRONECKER'S THEORY OF PENCILS
M. D. Atkinson, August 1983.

SCS-TR-30 SOME TECHNIQUES FOR GROUP CHARACTER REDUCTION
M. D. Atkinson and R. A. Hassan, August 1983.

SCS-TR-31 AN OPTIMAL ALGORITHM FOR GEOMETRICAL CONGRUENCE
M. D. Atkinson, August 1983.

SCS-TR-32 MULTI-ACTION LEARNING AUTOMATA POSSESSING ERGODICITY OF THE MEAN
B. J. Oommen and M. A. L. Thathachar, August 1983.

SCS-TR-33 FIBONACCI GRAPHS, CYCLIC PERMUTATIONS AND EXTREMAL POINTS
N. Santoro and J. Urrutia, December 1983

SCS-TR-34 DISTRIBUTED SORTING
D. Rotem, N. Santoro, and J. B. Sidney, December 1983

SCS-TR-35 A REDUCTION TECHNIQUE FOR SELECTION IN DISTRIBUTED FILES: II
N. Santoro, M. Scheutzow, and J. B. Sidney, December 1983

SCS-TR-36 THE ASYMPTOTIC OPTIMALITY OF DISCRETIZED LINEAR REWARD-INACTION LEARNING AUTOMATA
B. J. Oommen and Eldon Hansen, January 1984

SCS-TR-37 GEOMETRIC CONTAINMENT IS NOT REDUCIBLE TO PARETO DOMINANCE
N. Santoro, J. B. Sidney, S. J. Sidney, and J. Urrutia, January 1984

SCS-TR-38 AN IMPROVED AGLORITHM FOR BOOLEAN MATRIX MULTIPLICATION
N. Santoro and J. Urrutia, January 1984

SCS-TR-39 CONTAINMENT OF ELEMENTARY GEOMETRIC OBJECTS
J. Sack, N. Santoro and J. Urrutia, February 1984

SCS-TR-40 SADE: A PROGRAMMING ENVIRONMENT FOR DESIGNING AND TESTING SYSTOLIC ALGORITHMS
J. P. Corriveau and N. Santoro, February 1984

SCS-TR-41 INTERSECTION GRAPHS, {B }-ORIENTABLE GRAPHS AND PROPER CIRCULAR ARC GRAPHS
Jorge Urrutia, February 1984

SCS-TR-42 MINIMUM DECOMPOSITIONS OF POLYGONAL OBJECTS
J. Mark Keil and Jorg-R. Sack, March 1984

SCS-TR-43 AN ALGORITHM FOR MERGING HEAPS
Jorg-R. Sack and Thomas Strothotte, March 1984

SCS-TR-44 **A DIGITAL HASHING SCHEME FOR DYNAMIC MULTIATTRIBUTE FILES**
E. J. Otoo, March 1984

SCS-TR-45 **SYMMETRIC INDEX MAINTENANCE USING MULTIDIMENSIONAL LINEAR HASHING**
E. J. Otoo, March 1984

SCS-TR-46 **A MAPPING FUNCTION FOR THE DIRECTORY OF A MULTIDIMENSIONAL EXTENDIBLE HASHING**
E. J. Otoo, March 1984

SCS-TR-47 **TRANSLATING POLYGONS IN THE PLANE**
Jorg-R. Sack, March 1984

SCS-TR-48 **CONSTRAINED STRING EDITING**
J. Oommen, May 1984

SCS-TR-49 **O(N) ELECTION ALGORITHMS IN COMPLETE NETWORKS WITH GLOBAL SENSE OF ORIENTATION**
Jorg Sack, Nicola Santoro, Jorge Urrutia, May 1984

SCS-TR-50 **THE DESIGN OF A PROGRAM EDITOR BASED ON CONSTRAINTS**
Christopher A. Carter and Wilf R. LaLonde, May 1984

SCS-TR-51 **DISCRETIZED LINEAR INACTION-PENALTY LEARNING AUTOMATA**
B. J. Oommen and Eldon Hansen, May 1984

SCS-TR-52 **SENSE OF DIRECTION, TOPOLOGICAL AWARENESS AND COMMUNICATION COMPLEXITY**
Nicola Santoro, May 1984

SCS-TR-53 **OPTIMAL LIST ORGANIZING STRATEGY WHICH USES STOCHASTIC MOVE-TO-FRONT OPERATIONS**
B. J. Oommen, June 1984

SCS-TR-54 **RECTINLINEAR COMPUTATIONAL GEOMETRY**
J. Sack, June 1984

SCS-TR-55 **AN EFFICIENT, IMPLICIT DOUBLE-ENDED PRIORITY QUEUE**
M. D. Atkinson, Jorg Sack, Nicola Santoro, T. Strothotte, July 1984

SCS-TR-56 **DYNAMIC MULTIPAGING: A MULTIDIMENSIONAL STRUCTURE FOR FAST ASSOCIATIVE SEARCHING**
E. Otoo, T. H. Merrett

SCS-TR-57 **SPECIALIZATION, GENERALIZATION AND INHERITANCE**
Wilf R. LaLonde, John R. Pugh, August 1984

SCS-TR-58 COMPUTER ACCESS METHODS FOR EXTENDIBLE ARRAYS OF VARYING DIMENSIONS
E. Otoo, August 1984.

SCS-TR-59 AREA-EFFICIENT EMBEDDINGS OF TREES
J. P. Corriveau, Nicola Santoro, August 1984.

SCS-TR-60 UNIQUELY COLOURABLE m-DICHROMATIC ORIENTED GRAPHS
V. Neumann-Lara, N. Santoro, J. Urrutia, August 1984.

SCS-TR-61 ANALYSIS OF A DISTRIBUTED ALGORITHMS FOR EXTREMA FINDING IN A RING
D. Rotem, E. Korach and N. Santoro, August 1984.

SCS-TR-62 ON ZIGZAG PERMUTATIONS AND COMPARISONS OF ADJACENT ELEMENTS
M. D. Atkinson, October 1984

SCS-TR-63 SETS OF INTEGERS WITH DISTINCT DIFFERENCES
M. D. Atkinson, A. Hassenklover, October 1984.

SCS-TR-64 TEACHING FIFTH GENERATION COMPUTING: THE IMPORTANCE OF SMALL TALK
Wilf R. LaLonde, Dave A. Thomas, John R. Pugh, October 1984.

SCS-TR-65 AN EXTREMELY FAST MINIMUM SPANNING CIRCLE ALGORITHM
B. J. Oommen, October, 1984.

SCS-TR-66 ON THE FUTILITY OF ARBITRARILY INCREASING MEMORY CAPABILITIES OF STOCHASTIC LEARNING AUTOMATA
B. J. Oommen, October, 1984. Revised May 1985.

SCS-TR-67 HEAPS IN HEAPS
T. Strothotte, J.-R. Sack, November 1984. Revised April 1985.

SCS-TR-68 PARTIAL ORDERS AND COMPARISON PROBLEMS
M. D. Atkinson, November 1984.

SCS-TR-69 ON THE EXPECTED COMMUNICATION COMPLEXITY OF DISTRIBUTED SELECTION
N. Santoro, J. B. Sidney, S. J. Sidney, February 1985.

SCS-TR-70 FEATURES OF FIFTH GENERATION LANGUAGES: A PANORAMIC VIEW
Wilf R. LaLonde, John R. Pugh, March 1985.

SCS-TR-71 ACTRA: THE DESIGN OF AN INDUSTRIAL FIFTH GENERATION SMALLTALK SYSTEM
David A. Thomas, Wilf R. LaLonde, April 1985.

SCS-TR-72 MINMAXHEAPS, ORDERSTATISTICSTREES AND THEIR APPLICATION TO THE COURSEMARKS PROBLEM
M. D. Atkinson, J.-R. Sack, N. Santoro, Th. Strothotte, March 1985.

SCS-TR-73 DESIGNING COMMUNITIES OF DATA TYPES
Wilf R. LaLonde, May 1985.

SCS-TR-74 ABSORBING AND ERGODIC DISCRETIZED TWO ACTION LEARNING AUTOMATA
B. John Oommen, May 1985.

SCS-TR-75 OPTIMAL PARALLEL MERGING WITHOUT MEMORY CONFLICTS
Selim Akl and Nicola Santoro, May 1985.

SCS-TR-76 LIST ORGANIZING STRATEGIES USING STOCHASTIC MOVE-TO-FRONT AND STOCHASTIC MOVE-TO-REAR OPERATIONS
B. John Oommen, May 1985.