ON ADDING *CONSTRAINT ACCUMULATION* TO PROLOG

Wilf R. LaLonde

SCS-TR-105

January 1987

School of Computer Science
Carleton University
Ottawa, Ontario
CANADA K1S 5B6

This research was supported by NSERC, DREA, AND DREO.

On Adding *Constraint Accumulation* To Prolog

Wilf R. LaLonde

School of Computer Science
Carleton University
Ottawa, Ontario, Canada K1S 5B6.

Abstract Recently, Colmerauer[4] described a variation of Prolog that permits equality and inequality constraints to be specified between distinct logic variables. We describe a generalization of that notion by permitting arbitrary relations as constraints. The generalization is easy to retrofit into existing Prolog systems.

1. Introduction

Because logic variables can be viewed both as input and output variables in different contexts, the notion that Prolog is a practical language for solving constraint satisfaction problems is a viable one. However, Prolog is not as general as constraint languages like ThingLab[1] that provide mechanisms for satisfying violated constraints. Its main power lies in powerful backtracking mechanisms for generating solutions that can easily be tested for compliance with constraints imposed by the requirements of the solution. Even so, as a constraint testing language, it is deficient in one important respect. Constraints (relations that must be true) can only be tested for at computational points in which the relevant variables are already bound. Thus it is not possible to easily accummulate constraints during the computation; e.g., prior to binding the logic variables that must satisfy the constraints.

A special case solution to this problem was presented by Colmerauer[4]. His solution permitted both equality and inequality constraints to be associated with logic variables. Equality constraints are already accommodated directly by Prolog; e.g., by unifying variables constrained to be equal. However, inequality constraints cannot. Implementations supporting the notions were in progress. However, their special orientation to the above two specific constraints led to a computational model very different from the traditional ones used to implement Prolog.

Our approach generalizes the above to arbitrary constraints. Moreover, the implementation mechanism needed to support the generalization is easily retrofitted into existing Prolog implementations. A simple prototype of the **constraint accumulation** language has been implemented in Smalltalk to test the ideas. However, we have had little opportunity to experiment with this constraint accumulation facility. The design of useful programs using constraints is a topic for further research.

2. What A Constraint Accumulation Facility Is?

A **constraint** is a relation between arbitrary logic variables that **must hold when ALL variables**

are bound. A **constraint accumulation** facility associates the constraints with the contained variables. As soon as the last variable in the constraint is unified with a non-variable, the constraint is checked once. If the check is successful, the variable is successfully unified; otherwise, failure results and normal backtracking occurs. Constraints can therefore be viewed as a delayed evaluation mechanism whereby variables accummulate restrictions as the computation proceeds. The restrictions are checked only when all participating variables are bound. Consequently, it serves as a selection mechanism for eliminating candidate bindings far removed from the original constraint description site.

To serve as a simple example, we consider a generalization of the cryptarithmetic puzzle presented by Colmerauer[4]. In keeping with his notation, we syntactically differentiate between constraints and normal relations (structures) by enclosing constraints in special brackets "{" and "}". The **add** rules perform standard addition with carry on a list of digits in which the ordering is the reverse of the standard ordering; i.e., the leftmost digit is the low order digit and the higher order digits are to the right. The **cycle** rules simply generates all digit values between 0 and 9 (when initiated with -1); the **listCycle** rules do the same for all digits of a number (a list of variables). Rule **solveAddPuzzle** actually solves the puzzle but **test** imposes the interesting constraints.

```
add (0, [], [], []).

add (1, [], [], [1]).

add (Carry, [LowDigit1 | HigherDigits1], [], [LowDigit | HigherDigits]) :-
    Sum is LowDigit1 + Carry,
    NewCarry is Sum / 10,
    LowDigit is Sum mod 10,
    add (NewCarry, HigherDigits1, [], HigherDigits).

add (Carry, [], [LowDigit2 | HigherDigits2], [LowDigit | HigherDigits]) :-
    add (Carry, [LowDigit2 | HigherDigits2], [], [LowDigit | HigherDigits]).

add (Carry, [LowDigit1 | HigherDigits1], [LowDigit2 | HigherDigits2], [LowDigit | HigherDigits]) :-
    Sum is LowDigit1 + LowDigit2 + Carry,
    NewCarry is Sum / 10,
    LowDigit is Sum mod 10,
    add (NewCarry, HigherDigits1, HigherDigits2, HigherDigits).

cycle (OldValue, NewValue) :-
    OldValue < 9,
    NewValue is OldValue + 1.

cycle (OldValue, NewestValue) :-
    OldValue < 8,
    NewValue is OldValue + 1,
    cycle (NewValue, NewestValue).

listCycle ([]).

listCycle ([Element | RemainingElements)
    cycle (-1, Element),
    listCycle (RemainingElements).
```

```
solveAddPuzzle (Value1, Value2, Sum) :-
    listCycle (Value1), listCycle (Value2),
    add (0, Value1, Value2, Sum).

test (B,I,G,O,Y,M,A,N) :-
    {B \= I}, {I \= G}, {G \= O}, {O \= Y}, {Y \= M}, {M \= A}, {A \= N},
    solveAddPuzzle ([B,I,G], [B,O,Y], [M,A,N]).
```

In this example, solutions to the problem BIG + BOY = MAN are obtained using solveAddPuzzle. However, the solutions are constrained by the requirement that each letter be unique. For instance, the constraint {B \= I} stipulates that B must not equal I when and if B and I are both bound. This problem could be solved without constraints by moving the inequalities below solveAddPuzzle and removing the special brackets. Computationally, however, the effects are quite different. With constraints, any binding to O, for example, automatically triggers a test to determine whether the constraints associated with O are satisfied. If G is already bound, the constraint {G \= O} is tested. If I is already bound, the constraint {O \= I} is tested. If the constraint fails, the failure occurs in the context where the binding was attempted. The constraints consequently eliminate many of the search paths. The traditional approach must search all possible solutions. Only after each solution is obtained are tests performed to accept or reject the solution.

We have already mentioned that little experience has been gained in programming with constraint accumulation. Nevertheless, the ease they provide in **accumulating restrictions** is an important facility that does not yet exist in standard Prolog[3]. For instance, in a geometric setting such as Borning[1]'s, we could be computing with arbitrary quadrilaterals. Invoking **imposeMinimumRequirements** (see below) only serves to attach restrictions to the original parameters. **Only when all four points are bound** will they be tested for squareness and minimum area size. In the same way, a computation that was determining eligible employees in a special contest might require (among a large number of other restrictions) that s/he be unrelated to the company president. Constraints therefore act as demons being activated only when sufficient information is obtained to make the test possible.

```
imposeMinimumRequirements (Point1, Point2, Point3, Point4) :-
    {isQuare (Point1, Point2, Point3, Point4)},
    {area (Point1, Point2, Point3, Point4) > 100}.

unrelatedToCompanyPresident (APerson) :-
    companyOfEmployment (APerson, ACompany),
    president (ACompany, AnotherPerson),
    not (related APerson AnotherPerson).

candidateEmployee (APerson) :-
```

{unrelatedToCompanyPresident (APerson)}

The Colmerauer[4] restriction that constriants involve only = and \= is a restriction driven primarily by the implementation technique used. Given that Prolog encourages users to define their own relations, it seems only natural to permit user definable constraints. The more general case suggests an implementation technique that is a small perturbation of existing techniques.

3. Modifying An Existing Prolog Implementation

How we modify an existing interpreter will depend a great deal on the actual implementatiuon strategy currently supported. Details will differ between structure sharing implementations like Warren[8]'s and structure copying versions like Mellish[5]'s. Nevertheless, the modifications must center around two extensions:

1. **Generalizing Variables:** In addition to permitting an arbitrary value to be bound to a logic variable, we must also permit an arbitrary list of constraints to be associated with the variable.

2. **Generalizing The Unifier:** To take the above into account, the unifier must be modified as follows:
   (a) When unifying a variable A with a variable B, perform he binding in a manner that is equivalent to propagating the A-constraints into the B variable. By the A-constraints, we mean not only the constraints associated with A but also the constraints associated with other variables that previously unified with A.
   (b) When unifying a variable A with a non-variable, perform the binding and then consider each A-constraint in turn. **If all variables in a specific constraint are bound** (to non-variables), attempt to prove the constraint. If it succeeds, remove this constraint from further consideration. If it fails, the unification fails.

A structure sharing implementation, for example, could be devised based on the notion of a **prototype frame**. Prototype frames would contain a vector of **prototype variables** and all of the static information about rules including the constraints they contain along with initialization information. Rule instantiation would involve making a copy of the prototype frame that shares all of the static components and also making a copy of the prototype variables. Static information in the prototype frame that references logic variables would be in terms of **logic variable offsets.**

More specifically, adding a rule to the database would involved the creation of a prototype frame that contains the following:

(1) **prototype variables**: their structure is detailed below.
(2) **the relativized constraints**: the individual constraints with all variables replaced by offsets; associated with each constraint is a **constraint counter** and an **activation indicator**.
(3) **the relativized rule**: a modification of the original rule in interpretive systems or a compiled version; in either case, offsets are used for variable references.

If a constraint is a relation between 3 variables, for example, the constraint counter is initialized to 3. Each time a variable is bound, the constraint counters of all associated constraints are decremented. Determining when a constraint is to be tested is a matter of determining when the constraint counter reaches 0. When a constraint is encountered in the middle of a rule during execution, it is **activated** by setting the activation indicator (initially off). When the prototype frame is copied to make a new frame, only the variables, counters, and indicators are copied; the remaining information is shared.

Prototype variables contain the expected information along with special information that alleviates the need to propagate constraints (a bindee list):

(1) **binding value**: the value bound to the variable; a distinguished value is used to indicate the variable is unbound. The alternative would add an additional boolean to keep track of whether or not the variable is bound.
(2) **dynamic frame**: the frame that contains this variable. This enables the constraints and constraint counters to be accessed from the variable.
(3) **associated constraints**: a list of offsets into the dynamic frame (recall that the constraint information is shared with the prototype frame).
(4) **bindee list**: when a variable A is bound to variable B, A is added to B's bindee list by the unifier.

Since unification always dereferences variables until either a value or unbound variable is encountered, any variable X being bound to a non-variable must be the last one in a chain of variables. This fact and the use of bindee lists permits the following algorithm to be used to test the relevant constraints. After binding X to a value, process the constraints associated with X. If successful, all constraints associated with the bindee list is then processed, and their bindee lists etc. recursively. Processing the constraints amounts to decrementing the constraint counter and proving those for which the counters have reached 0. Failure at any point implies that binding X to a value has failed with the subsequent backtracking (and undoing via the trail of undo operations) being executed.

4. The Actual Implementation

Our original aim had been to add a traditional Prolog[3] programming facility to Smalltalk. We began by implementing a variant of the structure sharing interpreter based on success continuations as described

in Carlsson[2] but modified to create a new activation record or frame as in Warren[8] for each rule instantiation. Unlike the latter, however, **logic variables** and **logic variable offsets** were implemented as independent objects. Before completing this implementation, we switched over to a compiler-based implementation and incorporated the major ideas provided by a PC-based Smalltalk implementation by Digitalk[6] and by a prototype of Prolog with inheritance called Poops[7]. The current prototype does not yet make use of counters to speed up testing for applicable constraints nor is it fully integrated with the Smalltalk debugger. An LR(k) parser is used to parse the Prolog syntax.

The Digitalk implementation (based on the above Carlsson[2] model) provides a clean model for interfacing between the Smalltalk and Prolog worlds. The Poops prototype distinguishes between local rulebases for Prolog objects and inherited rules associated with the class of the objects. Prolog queries can be sent to Prolog objects from standard Smalltalk code via operations ? (to obtain all solutions) or ?? (to obtain a stream that can returns successive solutions one at a time; this situation will work for infinite solutions). Prolog variables used in the query are local to the query and implicitly declared.

```
aPrologObject ? aPrologRelation, aPrologRelation, ..., aPrologRelation.
aPrologObject ?? aPrologRelation, aPrologRelation, ..., aPrologRelation.
```

Conversely, a Prolog rule can interface to standard Smalltalk objects via the **is** operations as shown. The smalltalk code can access the logic variables and query it via simple messages such as **hasValue** and **value**.

```
aLogicVariable is (arbitrarySmalltalkCode)
```

We illustrate the above with two Prolog methods: **anInteger** and **factorial** along with two Smalltalk methods that invoke the Prolog methods (or rules). The two kinds of rules can co-exist in the same class of objects. These methods are provided in class PrologObject. Except for built-in Prolog operations like **is**, =, ~=, >, >=, <, <=, most operations must be sent to a specific Prolog object. In the examples that follow, the receiver used is **self**.

instance operations

factorial support

```
anInteger (oldValue, newValue) :-
    newValue is (oldValue value + 1).
anInteger (oldValue, newValue) :-
    anotherValue is (oldValue value + 1), self.anInteger (anotherValue, newValue).
"This rule invoked as anInteger (-1, aLogicVariable) generates all integers from 0 onward (infinite)"
```

factorial rule

```
factorial (x, y) :-
    "Illustrates the interface from Prolog to Smalltalk via is"
    self.isNonVariable (x), !, "! is the cut"
    y is (x value factorial).
factorial (x, y) :-
    self.isVariable (x), self.isVariable (y),
    self.anInteger (-1, x),
    self.factorial (x, y).
```

class methods

examples

factorialTest

```
    "Illustrates the interface from Smalltalk to Prolog via ?"
    | result |
    result ← PrologObject new ? factorial (3, b), factorial (b, c).
    ^result first at: 2 "returns 720"
    "A query with n logic variable returns all solutions as a list; if there are no solutions, the list is
    empty. Each solution consists of the n values of the logic variables (as an array) in order of
    appearance. For this example, the result obtained is the one-solution list ((6, 720)) corresponding
    to variables b and c."
```

factorialTestAgain

```
    "Illustrates interface from Smalltalk to Prolog via ??"
    | result answer |
    result ← PrologTest new ?? factorial (a, b).
    answer ← Array new: 5. "For the first five solutions"

    result atEnd ifTrue: [^#done]. "Doesn't happen"
    answer at: 1 put: result next. "Obtains (0, 1)"
    answer at: 2 put: result next. "Obtains (1, 1)"
    answer at: 3 put: result next. "Obtains (2, 2)"
    answer at: 4 put: result next. "Obtains (3, 6)"
    answer at: 5 put: result peek. "Obtains (4, 24). Illustrates peek"
    answer at: 5 put: result next. "Obtains (4, 24)"
    ^answer
```

Facts and rules associated with specific objects are added via **addFact** and **addRule** (this latter operation

is provided for but not yet implemented). Specific syntax in the Poops[7] style consists of

aRuleName (...) "Local rule invocation"
self.aRuleName (...) "Invoking a global rule associated with the receiver (self) class"
aLogicVariable.aRuleName (...) "Invoking a global rule associated with the receiver (variable) class"

Unless otherwise specified, queries via ? and ?? are implicitly prefixed by "self."; i.e., local rule invocation is possible only in a Prolog method. An example illustrative of the syntax and the facility is provided below.

class name Person
superclass PrologObject
instance variable names "None"

instance methods

ancestry operations

female () :- IAmFemale ().
"I am a female (global rule) if I contain a local fact to that effect."!
male () :- IAmMale ().
"I am a male (global rule) if I contain a local fact to that effect."!
parent (x) :- self.male (), father (x).
"I am the parent of x if I am male (global rule) and I am the father of x (local rule)."
parent (x) :- self.female (), mother (x).
"I am the parent of x if I am female (global rule) and I am the mother of x (local rule)."
setChild (aChild) :- self.male (), self.addFact (father (aChild)).
setChild (aChild) :- self.female (), self.addFact (mother (aChild)).
setFemale () :- self.addFact (IAmFemale ()). "Adds a local fact"
setMale () :- self.addFact (IAmMale ()). "Adds a local fact"

class methods

examples

personTest
| person1 person2 person3 query1 query2 query3 |
(person1 ← Person **new**) ? setMale (), setChild (#Brannon). "Discard result of query"
(person2 ← Person **new**) ? setFemale (). "Discard result"
(person3 ← Person **new**) ? setMale (), setChild (#John), setChild (#Mary). "Discard result"

```
        query1 ← person1 ? parent (ofWho). "Returns ((Brannon))"
        query2 ← person2 ? parent (ofWho). "Returns ()"
        query3 ← person3 ? parent (ofWho). "Returns ((John) (Mary))"
        ^query1, #(and), query2, #(and), query3 "Concatenate results"

class name Male
superclass Person
instance variable names "None"

instance methods

ancestry operations
    male ().
        "I am a male for sure. Specialization Male doesn't need the fact stored."

class methods

examples
    maleTest
        | person1 person2 query1 query2 |
        (person1 ← Male new) ? setChild (#Brannon). "Discard result"
        (person2 ← Male new) ? setChild (#John), setChild (#Mary). "Discard result"
        query1 ← person1 ? parent (ofWho). "Returns ((Brannon))"
        query2 ← person2 ? parent (ofWho). "Returns ((John) (Mary))"
        ^query1, #(and), query2

class name Female
superclass Person
instance variable names "None"

instance methods

ancestry operations
    female ().
        "I am a female for sure"

class methods
```

examples

femaleTest

```
| query1 query2 query3 female1 female2 male1 |
query1 ← (Female new) ? parent (ofWho). "Returns no solution; i.e., ()"
female1 ← Female new. female2 ← Female new. male1 ← Male new.
female1 ? aChild is (female2), setChild (aChild). "female1 has child female2"
male1 ? aChild is (female2), setChild (aChild). "male1 also has child female2"
query2 ← female1 ? parent (aChild), aChild.female (). "Returns one solution containing female2"
query2 ← female1 ? parent (aChild), aChild.male (). "Returns no solution (the child is not male)"
"The above illustrates a query directed at a specific prolog object"
```

We finish with a final example that illustrates the use of constraints. The methods are assumed to be in class PrologObject. The built-in arithmetic operations can equivalently be specified as relations; e.g., **prologEqual (x, y)** is equivalent to **x = y**, **prologGreater (x, y)** is equivalent to **x > y**, etc. A constraint is a relation that is specified with { } instead of (). Hence **x = y** can be specified as a constraint by writing **prologEqual {x, y}**; **factorial (x, y)** as a constraint is **factorial {x, y}**.

instance methods

arithmetic

```
even (x) :- y is (x value even), y is (true).
odd (x) :- y is (x value odd), y is (true).
bothEven (x, y) :- self.even(x), self.even(y).
bothOdd (x, y) :- self.odd(x), self.odd(y).
bothSame (x, y) :- self.bothEven (x, y).
bothSame (x, y) :- self.bothOdd (x, y).
```

class methods

testSame

```
| aPrologObject result |
aPrologObject ← PrologObject new.
result ← aPrologObject ? odd (1). "Succeeds, returns (())"
result ← aPrologObject ? odd (2). "Fails, returns ()"
result ← aPrologObject ? bothSame {x,y}, x = 1, y = 3. "Succeeds, returns ((1 3))"
result ← aPrologObject ? bothSame {x,y}, x = 1, y = 4. "Fails, returns ()"
result ← aPrologObject ? bothSame {x,y}, odd {x}, prologGreater {x, y}, x = 11, y = 1.
    "Returns ((11, 1))"
```

5. Conclusion

We presented an extension of Colmerauer's modified Prolog in which equality and inequality constraints between logic variables (not necessarily bound) are maintained. Our extension permits arbitrary relations (constraints) to be accumulated with variables during the course of exectuion. These constraints are tested once when all variables associated with the constraint are bound. Programming **by constraint accumulation** is a new paradigm that has not yet been explored by the programming community.

We described informally how an existing Prolog interpreter might be modified to accommodate constraints as we defined them. We also described a prototype implementation that demonstrates the feasibility of the approach. This paper should at least pave the way for more serious implementations.

References

1. Borning, A., *The Programming Language Aspects of Thinglab, A Constraint-Oriented Simulation Laboratory,* ACM Toplas, Vol. 3, No. 4, Oct. 1981, pp. 353-387.
2. Carlsson, M., *On Implementing Prolog In Functional Programming*, 1984 International Symposium on Logic Programming, February 1984, Atlantic City, New Jersey, pp. 154-159.
3. Clocksin, W. and Mellish, C., *Programming in Prolog*, Springer-Verlag, Berlin, 1981.
4. Colmerauer, A., *Prolog in 10 Figures*, Proceedingts of the Eight Internaltional Joint Conference on Artificial Intelligence, August 1983, Karlsruhe, West Germany, pp. 487-499.
5. Mellish, C.S., *An Alternative To Structure Sharing In The Implementation Of A Prolog Interpreter*, in Tärnlund, S.-A., Clark, K.L. (eds.), *Logic Programming*, Academic Press, London, 1982.
6. Smalltalk/V, Tutorial and Programming Handbook, Digitalk Inc., Los Angeles, California, 1986.
7. Vaucher, J.G. and Lapalme, G., Poops: Object-Oriented Programming In Prolog, Technical Report 565, Laboratoire INCOGNITO, Dept. d'Informatique et de Recherche Operationnelle, University of Montreal, March 1986.
8. Warren, D., *Implementing Prolog - Compiling Predicate Logic Programs*, Dept. of Artificial Intelligence, University of Edinburgh, 1977.

Carleton University, School of Computer Science
Bibliography of Technical Reports
Publications List (1985 -->)

School of Computer Science
Carleton University
Ottawa, Ontario, Canada
KIS 5B6

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**
out-of-print M.D. Atkinson, November 1984. See Congressus Numerantium 47 ('86), 77-88

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, T. Strothotte, March 1985.

SCS-TR-73 **Designing Communities of Data Types**
Wilf R. LaLonde, May 1985.
Replaced by SCS-TR-108

SCS-TR-74 **Absorbing and Ergodic Discretized Two Action Learning Automata**
out-of-print B. John Oommen, May 1985. See IEEE Trans. on Systems, Man and Cybernetics, March/April 1986, pp. 282-293.

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.

SCS-TR-77 **Linearizing the Directory Growth in Order Preserving Extendible Hashing**
E.J. Otoo, July 1985.

SCS-TR-78 **Improving Semijoin Evaluation in Distributed Query Processing**
E.J. Otoo, N. Santoro, D. Rotem, July 1985.

SCS-TR-79 **On the Problem of Translating an Elliptic Object Through a Workspace of Elliptic Obstacles**
B.J. Oommen, I. Reichstein, July 1985.

SCS-TR-80 **Smalltalk - Discovering the System**
W. LaLonde, J. Pugh, D. Thomas, October 1985.

SCS-TR-81 **A Learning Automation Solution to the Stochastic Minimum Spanning Circle Problem**
B.J. Oommen, October 1985.

SCS-TR-82 **Separability of Sets of Polygons**
Frank Dehne, Jörg-R. Sack, October 1985.

SCS-TR-83 **Extensions of Partial Orders of Bounded Width**
out-of-print M.D. Atkinson and H.W. Chang, November 1985. See Congressus Numerantium, Vol. 52 (May 1986), pp. 21-35.

SCS-TR-84 **Deterministic Learning Automata Solutions to the Object Partitioning Problem**
B. John Oommen, D.C.Y. Ma, November 1985

SCS-TR-85 **Selecting Subsets of the Correct Density**
out-of-print M.D. Atkinson, December 1985. To appear in Congressus Numerantium, Proceedings of the 1986 South-Eastern conference on Graph theory, combinatorics and Computing.

SCS-TR-86 **Robot Navigation in Unknown Terrains Using Learned Visibility Graphs. Part I: The Disjoint Convex Obstacles Case**
B. J. Oommen, S.S. Iyengar, S.V.N. Rao, R.L. Kashyap, February 1986

SCS-TR-87 **Breaking Symmetry in Synchronous Networks**
Greg N. Frederickson, Nicola Santoro, April 1986

SCS-TR-88 **Data Structures and Data Types: An Object-Oriented Approach**
John R. Pugh, Wilf R. LaLonde and David A. Thomas, April 1986

SCS-TR-89 **Ergodic Learning Automata Capable of Incorporating Apriori Information**
B. J. Oommen, May 1986

SCS-TR-90 **Iterative Decomposition of Digital Systems and Its Applications**
Vaclav Dvorak, May 1986.

SCS-TR-91 **Actors in a Smalltalk Multiprocessor: A Case for Limited Parallelism**
Wilf R. LaLonde, Dave A. Thomas and John R. Pugh, May 1986

SCS-TR-92 **ACTRA - A Multitasking/Multiprocessing Smalltalk**
David A. Thomas, Wilf R. LaLonde, and John R. Pugh, May 1986

SCS-TR-93 **Why Exemplars are Better Than Classes**
Wilf R. LaLonde, May 1986

SCS-TR-94 **An Exemplar Based Smalltalk**
Wilf R. LaLonde, Dave A. Thomas and John R. Pugh, May 1986

SCS-TR-95 **Recognition of Noisy Subsequences Using Constrained Edit Distances**
B. John Oommen, June 1986

SCS-TR-96 **Guessing Games and Distributed Computations in Synchronous Networks**

Carleton University, School of Computer Science
Bibliography of Technical Reports

SCS-TR-117 **Recognizing Polygons, or How to Spy**
James A. Dean, Andrzej Lingas and Jörg-R. Sack, August 1987.

SCS-TR-118 **Stochastic Rendezvous Network Performance - Fast, First-Order Approximations**
J.E. Neilson, C.M. Woodside, J.W. Miernik, D.C. Petriu, August 1987.

SCS-TR-120 **Searching on Alphanumeric Keys Using Local Balanced Trie Hashing**
E.J. Otoo, August 1987.