Proceedings of the International Workshop on

Implementation Technologies for Programming Languages based on Logic

Konstantinos Sagonas (editor)

held in conjunction with the

Joint International Conference and Symposium on Logic Programming

Manchester, U.K.

Saturday 20th June, 1998

Preface

This workshop is a continuation of the series of workshops on Implementations of Logic Programming Systems, previously held with considerable success in Budapest (1993), Santa Margherita (1994) and Ithaca (1994), as well as the Compulog Net workshops on Parallelism and Implementation Technologies held in Madrid (1993-4), Utrecht (1995), Bonn (1996), and Port Jefferson (1997).

This time, the workshop happily coincided with an important historic event for computer science: the 50th Anniversary of the World's First Stored-Program Computer — built in the Department of Computer Science at the University of Manchester in 1948.

The main intent of this workshop series is to bring together, in an informal and friendly setting, key researchers on implementation technologies for logic-based languages and systems, in order to promote a much needed exchange of ideas and feedback on recent developments. To make the workshop more attractive to some participants, the workshop also serves as the annual Compulog Net Area meeting on Parallelism and Implementation Technologies (ParImp).

The workshop is traditionally focused on design and implementation of logic-based programming languages and systems whether sequential, parallel, or concurrent. Preference is usually given to the analysis and description of implemented systems (or systems currently under implementation) and their associated techniques, problems found in their development or design, and steps taken towards the solution of these problems. The Call For Papers listed the following as suggested (but not exclusive) topics of interest:

- standard and non-standard implementation schemes
  e.g. abstract machines, modification of WAM, translation to C, native-code generation etc.
- optimizing compilers and static analysis
- parallelism and concurrency
- techniques for the implementation of different extensions of logic programming
  e.g. constraints, multi-paradigm languages, tabulation etc.
- partial evaluation and program specialisation systems
- benchmarking and performance evaluation
- garbage collection
- programming environments
- logic-based tools for internet applications
- distributed and coordination logic programming systems
- experiences from using systems in real-life applications

All submitted papers were reviewed either by members of the program committee or by outside reviewers (see below). The help of the latter is especially appreciated and acknowledged. Each paper received at least three reviews. These proceedings include revised versions of the eight regular contributions which the program committee selected for presentation at the workshop — mainly due to the limited time allocated to the workshop not all submitted papers could be accepted — and an extended abstract of an invited talk by Paul Tarau titled *"Jinni: a Lightweight Java-based Logic Engine for Internet Programming"*.

Besides Paul for accepting this invitation, the workshop coordinator would like to extend his thanks to all authors that submitted papers to the workshop (especially those that followed the submission guidelines), the members of the PC committee for their help in the organization of

the workshop and their thorough reviews (the award went to 160-line review where 80 lines were occupied by a complete and working C program!), and Pat Hill (the Tutorial and Workshop Chair of JICSLP'98) for her help in the preparation of the workshop and of its hard-copy proceedings.

Workshop Coordinator

Konstantinos Sagonas

Program Committee

Fergus Henderson, University of Melbourne, Australia
Enrico Pontelli, New Mexico State University, U.S.A.
Konstantinos Sagonas, Catholic University of Leuven, Belgium
Vítor Santos Costa, University of Porto, Portugal
Kish Shen, IC-Parc, U.K.
Terrance Swift, University of Maryland, College Park, U.S.A.
Paul Tarau, Univerité de Moncton & Louisiana Tech University, Canada/U.S.A.

List of Additional Reviewers

Inês Dutra, Michel Ferreira, Luis Miguel Barros Lopes, Ricardo Rocha, Joachim Schimpf

Workshop WWW Site:

`http://www.cs.nmsu.edu/lldap/jicslp98`

Workshop Schedule

8.00 – 9.00	Registration
9.00 – 9.05	Welcome and introduction *Konstantinos Sagonas*
Section I	*Compilation/Implementation*
9.05 – 9.35	A choice-point library for backtrack programming *Pierre-Etienne Moreau*
9.35 – 10.05	Compiling Resources in a Linear Logic Programming Language *Mutsunori Banbara and Naoyuki Tamura*
10.05 – 10.35	Tracing Prolog Programs by Source Instrumentation is Efficient Enough *Mireille Ducassé and Jacques Noyé*
10.35 – 11.00	Coffee/tea break
Section II	*Multi-Threaded Implementations*
11.00 – 12.00	**Invited Talk** Jinni: a Lightweight Java-based Logic Engine for Internet Programming *Paul Tarau*
12.00 – 12:30	SICStus MT - Multithreaded Execution Environment for SICStus Prolog *Jesper Eskilson and Mats Carlsson*
12.30 – 14.00	Buffet Lunch
Section III	*Constraints and Applications*
14.00 – 14.30	Extension of the Revised Simplex Method for Detecting Implicit Equalities in Constraint Logic Programming *Philippe Refalo*
14.30 – 15.00	Parallel Arc-Consistency for functional constraints *A. Ruiz-Andino, L. Araujo, F. Sáenz and J. Ruz*
15.00 – 15.30	Natural Language Multiprocessing: A Case Study *Enrico Pontelli, Gopal Gupta, Janyce Wiebe and David Farwell*
15.30 – 16.00	Pseudo-Naive Evaluation: Mixing Top-down and Bottom-up Execution for Strongly Stratified Logic Programs *Donald A. Smith and Mark Utting*
16.00	End

Contributions

Invited Paper

- Jinni: a Lightweight Java-based Logic Engine for Internet Programming **1**
  Paul Tarau

Regular Papers

- A choice-point library for backtrack programming **16**
  Pierre-Etienne Moreau
- Compiling Resources in a Linear Logic Programming Language **32**
  Mutsunori Banbara and Naoyuki Tamura
- Tracing Prolog Programs by Source Instrumentation is Efficient Enough **46**
  Mireille Ducassé and Jacques Noyé
- SICStus MT - Multithreaded Execution Environment for SICStus Prolog **59**
  Jesper Eskilson and Mats Carlsson
- Extension of the Revised Simplex Method for Detecting Implicit Equalities in Constraint Logic Programming **72**
  Philippe Refalo
- Parallel Arc-Consistency for functional constraints **86**
  A. Ruiz-Andino, L. Araujo, F. Sáenz and J. Ruz
- Natural Language Multiprocessing: A Case Study **101**
  Enrico Pontelli, Gopal Gupta, Janyce Wiebe and David Farwell
- Pseudo-Naive Evaluation: Mixing Top-down and Bottom-up Execution for Strongly Stratified Logic Programs **113**
  Donald A. Smith and Mark Utting

Jinni: a Lightweight Java-based Logic Engine for Internet Programming

Paul Tarau
Université de Moncton and Louisiana Tech University
E-mail: taraup@umoncton.ca, tarau@coes.latech.edu

Abstract

We introduce JINNI (**J**ava **IN**ference engine and **N**etworked **I**nteractor), a new, lightweight, multi-threaded, pure logic programming language, intended to be used as a flexible scripting tool for gluing together knowledge processing components and Java objects in networked client/server applications. The last version of Jinni is available at
`http://clement.info.umoncton.ca/~tarau/netjinni/Jinni.html` .

Keywords: Java based Logic Programming languages, remote execution, Linda coordination, blackboard-based distributed logic programming, mobile code through first order continuations, mobile agents

1 Introduction

Jinni is a new, lightweight, pure logic programming language, intended to be used as a flexible scripting tool for gluing together knowledge processing components and Java objects in networked client/server applications and thin client environments. By supporting multiple threads, control mobility and inference processing, Jinni is well suited for quick prototyping of intelligent mobile agent programs.

Jinni supports multi-user synchronized transactions and interoperates with the latest version of BinProlog [16] , a high performance robust Prolog system with ability to generate C/C++ code and standalone executables.

For acronym lovers it can be read as: **J**ava **IN**ference engine and **N**etworked **I**nteractor, although its wishmaker status (high level, dense, network ubiquitous etc. glue language) is the real reason why it was called Jinni.

2 The world of Jinni

Jinni is based on a simple **Things**, **Places**, **Agents** ontology, borrowed from MUDs and MOOs [14, 1, 3, 9, 19, 15].

Things are represented as Prolog terms, basically trees of embedded records containing constants and variables to be further instantiated to other trees.

Places are processes running on various computers with a server component listening on a port and a blackboard component allowing synchronized multi-user Linda [6, 10] and remote predicate call transactions.

Agents are collections of threads executing a set of goals, possibly spread over a set of different Places and usually executing remote and local transactions in coordination with other Agents. Jinni

does not provide at this time a single abstract data type for agents or places because it is intended to be an infrastructure on top of which they are built in an application specific way. As crisp abstractions will emerge through development of libraries and applications, a hierarchy of Places and Agents will be built. Place and Agent prototypes will be clonable, support inheritance/sharing of Things and will be easily editable/configurable using visual tools. Agent threads moving between places and agents moving as units will be supported. Places will be used to abstract away language differences between processors, like for instance Jinni and BinProlog. Mobile code will allow very high speed processing in Jinni by delegating heavy inference processing to high-performance BinProlog components.

3 Jinni as a Logic Programming Java component

Jinni is intended to be a lightweight, thin client logic programming component, based as much as possible on fully portable, vendor and version independent Java code. Its main features come from this architectural choice:

- a trimmed down, simple, operatorless syntactic subset of Prolog,
- pure Prolog (Horn Clause logic) with leftmost goal unfolding as inference rule,
- multiple asynchronous inference engines each running as a separate thread,
- a shared blackboard to communicate between engines using a simple Linda-style subscribe/publish (in/out in Linda jargon) coordination protocol based on associative search,
- high level networking operations allowing code mobility [2, 12, 11, 4, 20, 13] and remote execution,
- a straightforward Jinni-to-Java translator allows packaging of Jinni programs as Java classes to be transparently loaded over the Internet
- backtrackable assumptions [18, 8] implemented through trailed, overridable undo actions

Jinni's spartan return to (almost) pure Horn Clause logic does not mean it is necessarily a weaker language. Expressiveness of full Prolog (and beyond:-)) is easily attained in Jinni by combining multiple engines. For theoretical CS lovers, the magic is similar to just adding another stack to a Push Down Automaton: this morphs it into a Turing machine! Engines give transparent access to the underlying Java threads and are used to implement local or remote, lazy or eager findall operations, negation as failure, if-then-else, etc. at source level. Inference engines running on separate threads can cooperate through either predicate calls or through an easy to use flavor of the Linda coordination protocol.

Remote or local dynamic database updates (with redesigned, non-backtracking synchronized transactions with immediate update semantics) make Jinni an extremely flexible Agent programming language. Jinni is designed on top of dynamic, fully garbage collectible data structures, to take advantage of Java's automatic memory management.

4 Basic agent programming with Jinni

Two simple agents are shown in file lib.pro:
Window 1: a reactive channel listener

```
?-listen(fun(_)).

Window 2: a selective channel publisher

?-talk(fun(jokes)).
```

They implement a front end to Jinni's associative publish/subscribe abilities. The more general pattern `fun(_)` will reach all the users interested in instances of `fun/1`, in particular `fun(jokes)`. However, someone publishing on an unrelated channel e.g. with `?-talk(stocks(nasdaq))`. will not reach fun/1 listeners because stocks(nasdaq) and fun(jokes) channel patterns are not unifiable.

5 Application domains

Jinni's client and server scripting abilities are intended to support platform and vendor independent Prolog-to-Java and Prolog-to-Prolog bidirectional connection over the net and to accelerate integration of the effective inference technologies developed the last 20 years in the field of Logic Programming in mainstream Internet products.

The next iteration is likely to bring a simple, plain English scripting language to be compiled to Jinni, along the lines of the LogiMOO prototype, with speech recognizer/synthesizer based I/O. A connection between Jinni and its Microsoft Agent counterpart Genie are among the high priority tasks likely to be left to the growing community of Jinni co-developers[1].

Among the potential targets for Jinni based products: lightweight rule based programs assisting customers of Java enables appliances, from Web based TVs to mobile cell phones and car computers, all requiring knowledge components to adjust to increasingly sophisticated user expectations.

A stock market simulator is currently on the way to be implemented based on Jinni, featuring user programmable intelligent agents. It is planned to be connected to real world Internet based stock trade services.

6 What's new in Jinni

6.1 Engines

Jinni inherits from BinProlog's design the ability to launch multiple Prolog engines having their own stack groups (heap, local stack and trail). An engine can be seen as an abstract data-type which produces a (possibly infinite) stream of solutions as needed. To create an new engine, we use:

```
new_engine(Goal,Answer,Handle)
```

The `Handle` is a unique Java Object denoting the engine, assigned to its own thread, for further processing.

To get an answer from the engine we use:

```
ask_engine(Handle,Answer)
```

[1] Jinni's sustained growth is insured through a relatively unconventional *bazaar* style development process, similar to Linux and more recently Netscape client products.

Each engine has its own virtual garbage collection process and backtracks independently using its choice-point stack and trail during the computation of an answer. Once computed, an answer is copied from an engine to its "master".

When the stream of answers reaches its end, `ask_engine/2` will simply fail. The resolution process in an engine can be discarded at any time with `stop_engine/1`. This allows avoiding the cost of backtracking, for instance in the case when a single answer is needed.
The following example (see more in file `tarau/jinni/lib.pro` in the Jinni distribution shows how to extract one solution from an engine:

```
one_solution(X,G,R):-
  new_engine(G,X,E),
  ask_engine(E,Answer),
  stop_engine(E),
  eq(Answer,R).
```

Note that new_engine/3 speculatively starts execution of Goal on a new thread and that either a term of the form **the(X)** or **no** is returned by ask_answer. Synchronization with this thread is performed when asking an answer, using a special monitor object. It is quite surprising how simply everything is built on top of this one_solution/3 primitive[2].

```
if(Cond,Then,Else):-
  one_solution(successful(Cond,Then),Cond,R),
  select_then_else(R,Cond,Then,Else).

select_then_else(the(successful(Cond,Then)),Cond,Then,_):-Then.
select_then_else(no,_,_,Else):-Else.

once(G):-one_solution(G,G,the(G)).

not(G):-one_solution(G,G,no).

copy_term(X,CX):-one_solution(X,true,the(CX)).

spawn(Goal):-new_engine(Goal,_,_). % spawns a new background thread
```

Similarly, findall/3 is emulated easily by iterating over ask_engine/2 operations. Note that lazy variants of findall can be implemented as well.
Multiple 'orthogonal engines' enhance the expressiveness of Prolog by allowing an AND-branch of an engine to collect answers from multiple OR-branch of another engine. They give to the programmer the means to see as an abstract sequence and control, the answers produced by an engine, in a way similar to Java's `Enumeration` interface.

6.2 Coordination and remote execution mechanisms

Our networking constructs are built on top of the popular Linda [6, 10, 7] coordination framework, enhanced with unification based pattern matching, remote execution and a set of simple client-server

[2]In fact, for efficiency reasons, now Jinni has a first_solution/3 builtin, which does not require a separate engine. It is implemented simply by throwing a special purpose exception when the first solution is found.

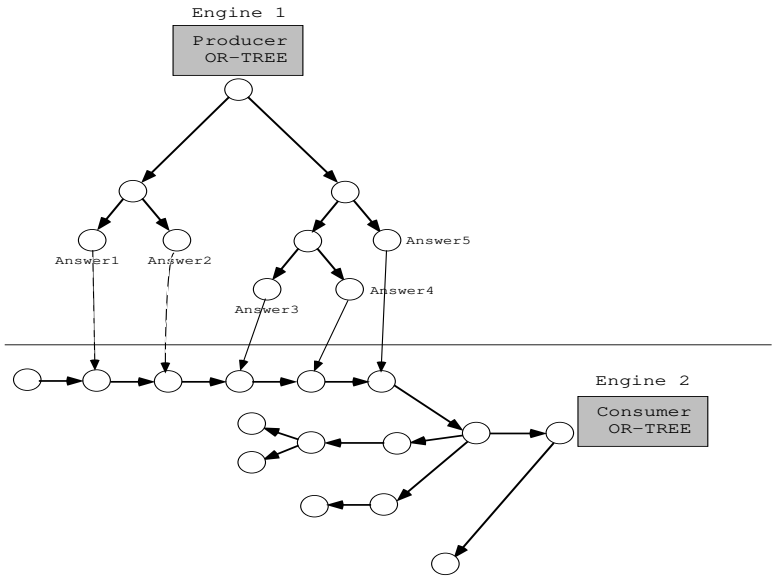

Figure 1: Orthogonal Engines

components melted together into a scalable peer-to-peer layer, forming a 'web of interconnected worlds':

```
out(X): puts X on the server
in(X):  waits until it can take an object matching X from the server
all(X,Xs): reads the list Xs matching X currently on the server
the(Pattern,Goal,Answer): starts a thread executing Goal on server
```

The presence of the `all/2` collector compensates for the lack of non-deterministic operations. Note that the only blocking operation is `in/1`. Blocking `rd/1` is easily emulated in terms of `in/1` and `out/1`, while non-blocking `rd/1` is emulated with `all/2`.

6.3 Server-side constraint solving

A natural extension to Linda is to use constraint solving for selection matching terms, instead of plain unification. This is implemented in Jinni through the use of 2 builtins:

Wait_for(Term,Constraint): waits for a Term such that Constraint is true on the server, and when this happens, it removes the result of the match from the server with an in/1 operation. Constraint is either a single goal or a list of goals [G1,G2,..,Gn] to be executed on the server.

Notify_about(Term): notifies the server to give this term to any blocked client which waits for it with a matching constraint i.e.

```
notify_about(stock_offer(nscp,29))
```

would trigger execution of a client having issued

```
wait_for(stock_offer(nscp,Price),less(Price,30)).
```

The use of server side execution was in fact suggested by a real-life stock market application. It turned out that is just too tricky in pure Linda to do something as simple as triggering an

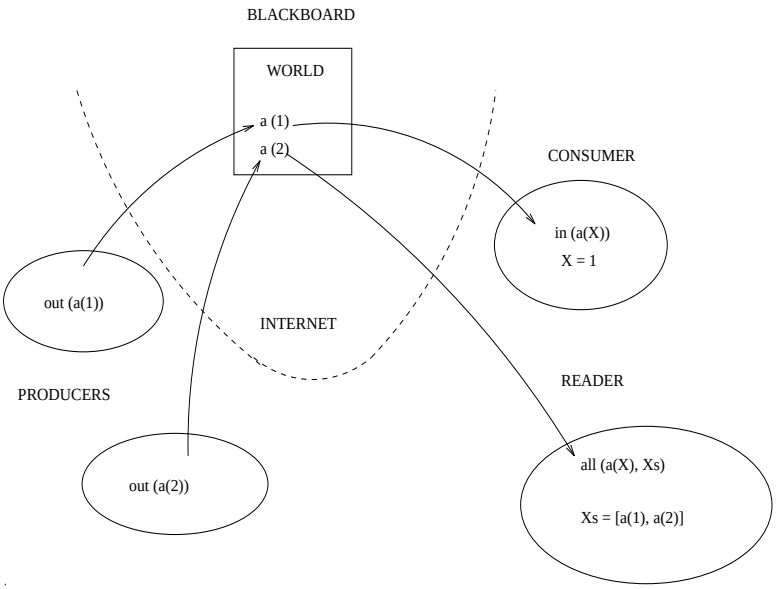

Figure 2: Basic Linda operations

atomic transaction when data verifying a simple arithmetic inequality becomes available. We plan to incorporate in the near future a simple server-side symbolic constraint reducer (FD or interval based).

6.4 Mobile Code: for expressiveness and for acceleration

An obvious way to accelerate slow Prolog processing for a Java based system is through use of native (C/C++) methods. The simplest way to accelerate Jinni's Prolog processing is by including BinProlog through Java's JNI.

However, a more general scenario, also usable for applets not allowing native method invocations is use of a remote accelerator. This is achieved transparently through the use of *mobile code*.

The Oz 2.0 distributed programming proposal of [20] makes *object mobility* more transparent, although the mobile entity is still the state of the objects, not "live" code.

Mobility of "live code" is called *computation mobility* [5]. It requires interrupting execution, moving the state of a runtime system (stacks, for instance) from one site to another and then resuming execution. Clearly, for some languages, this can be hard or completely impossible to achieve.

Telescript and General Magic's new Odissey [11] agent programming framework, IBM's Java based *aglets* [12] as well as Luca Cardelli's Oblique [2] have pioneered implementation technologies achieving *computation mobility*.

In the case of Jinni, computation mobility is used both as an *accelerator* and an *expressiveness lifting* device. A live thread will migrate from Jinni to a faster remote BinProlog engine, do some CPU intensive work and then come back with the results (or just sent back results, using Linda coordination). A very simple way to ensure atomicity and security of complex networked transactions is to have the agent code move to the site of the computation, follow existing security rules, access possibly large databases and come back with the results.

As continuations (encapsulating the future of the current computation) are first-order objects

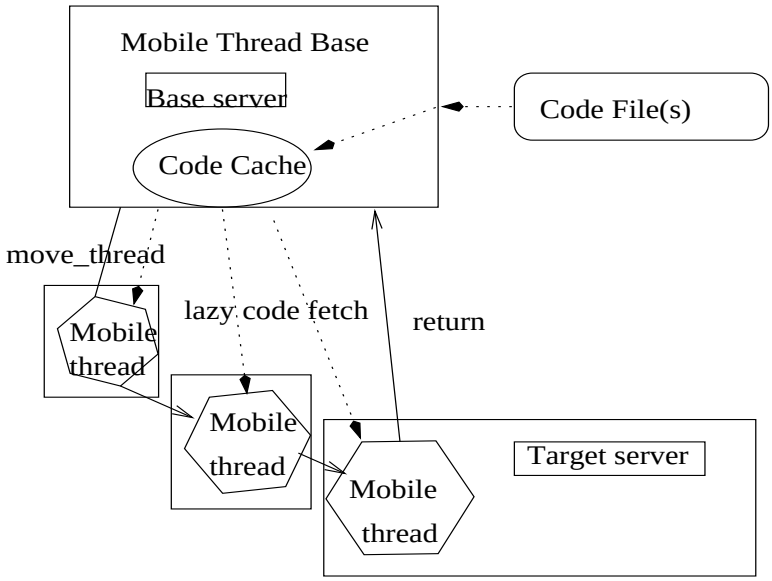

Figure 3: Launching a mobile thread from its base

both in Jinni and BinProlog, the implementation is straightforward [17].

Initially the target side waits in server mode. Once the continuation is received on the target side, the source side switches in server mode ready to execute code fetching and persistent database update requests from its mobile counterpart on the target side.

Fig. 3 shows the connections between a mobile thread and its base.

Note that our continuation moving protocol expresses *computation mobility* in terms of "client-server role alternation", i.e. by specifying which end plays which of the roles at a given time. In principle, with some added complexity, our target-side `remote_call` operation can be replaced with equivalent target side code (with clients pulling commands to be run locally from a server) if security restrictions or unreachability of the target through firewalls gets in the way.

7 Conclusion

The Jinni project shows that Logic Programming languages are well suited as the basic glue so much needed for elegant and cost efficient Internet programming. The ability to compress so much functionality in such a tiny package shows that building logic programming components to be integrated in emerging tools like Java might be the most practical way towards mainstream recognition and widespread use of Logic Programming technology.

Acknowledgments

We thank for support from NSERC (grants OGP0107411), the Université de Moncton, Louisiana Tech University as well as from E-COM Inc. and the Radiance Group Inc. Special thanks go to Veronica Dahl, Bart Demoen, Koen De Boschere, Ed Freeman, Don Garrett, Stephen Rochefort and Yu Zhang for fruitful interaction related to the design, implementation and testing of Jinni.

References

[1] The Avalon MUD. http://www.avalon-rpg.com/.

[2] K. A. Bharat and L. Cardelli. Migratory applications. In *Proceedings of the 8th Annual ACM Symposium on User Interface Software and Technology*, Nov. 1995. http://gatekeeper.dec.com/ pub/DEC/SRC/research-reports/ abstracts/src-rr-138.html.

[3] BlackSun. CyberGate. http://www.blacksun.com/.

[4] L. Cardelli. Mobile ambients. Technical report, Digital, 1997. http://www.research.digital.com/ SRC/personal/Luca_Cardelli/Papers.html.

[5] L. Cardelli. Mobile Computation. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems - Towards the Programmable Internet*, pages 3–6. Springer-Verlag, LNCS 1228, 1997.

[6] N. Carriero and D. Gelernter. Linda in context. *CACM*, 32(4):444–458, 1989.

[7] S. Castellani and P. Ciancarini. Enhancing Coordination and Modularity Mechanisms for a Languag e with Objects-as-Multisets. In P. Ciancarini and C. Hankin, editors, *Proc. 1st Int. Conf. on Coordination Models and Languages*, volume 1061 of *LNCS*, pages 89–106, Cesena, Italy, April 1996. Springer.

[8] V. Dahl, P. Tarau, and R. Li. Assumption Grammars for Processing Natural Language. In L. Naish, editor, *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 256–270, MIT press, 1997.

[9] K. De Bosschere, D. Perron, and P. Tarau. LogiMOO: Prolog Technology for Virtual Worlds. In *Proceedings of PAP'96*, pages 51–64, London, Apr. 1996.

[10] K. De Bosschere and P. Tarau. Blackboard-based Extensions in Prolog. *Software — Practice and Experience*, 26(1):49–69, Jan. 1996.

[11] GeneralMagicInc. Odissey. 1997. available at http://www.genmagic.com/agents.

[12] IBM. Aglets. http://www.trl.ibm.co.jp/aglets.

[13] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-Grained Mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[14] T. Meyer, D. Blair, and S. Hader. WAXweb: a MOO-based collaborative hypermedia system for WWW. *Computer Networks and ISDN Systems*, 28(1/2):77–84, 1995.

[15] P. Tarau. Logic Programming and Virtual Worlds. In *Proceedings of INAP96*, Tokyo, Nov. 1996. keynote address.

[16] P. Tarau. BinProlog 5.75 User Guide. Technical Report 97-1, Département d'Informatique, Université de Moncton, Apr. 1997. Available from *http://clement.info.umoncton.ca/BinProlog*.

[17] P. Tarau and V. Dahl. Mobile Threads through First Order Continuations. 1997. submitted, http://clement.info.umoncton.ca/ html/tmob/html.html.

[18] P. Tarau, V. Dahl, and A. Fall. Backtrackable State with Linear Affine Implication and Assumption Grammars. In J. Jaffar and R. H. Yap, editors, *Concurrency and Parallelism, Programming, Networking, and Security*, Lecture Notes in Computer Science 1179, pages 53–64, Singapore, Dec. 1996. Springer.

[19] P. Tarau and K. De Bosschere. Virtual World Brokerage with BinProlog and Netscape. In P. Tarau, A. Davison, K. De Bosschere, and M. Hermenegildo, editors, *Proceedings of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP'96, Bonn, Sept. 1996. http://clement.info.umoncton.ca/ lpnet.

[20] P. Van Roy, S. Haridi, and P. Brand. Using mobility to make transparent distribution practical. 1997. manuscript.

Appendix: A Quick Introduction to Jinni

Getting started

Using Jinni through an applet

You can try out the latest version of Jinni through an applet at:

`http://clement.info.umoncton.ca/~tarau/jinni/Jinni.html`
or
`http://www.latech.edu/~tarau/jinni/Jinni.html`

Enter for instance the query:

```
append(Xs,Ys,[1,2,3]).
```

You will see the applet displaying the results.

Using Jinni in command line mode:

Consulting a new program:

```
?-co(<myprog>).
```

will read in memory the file `<myprog>.pro` program replacing similarly named predicates with new ones. It is actually a shorthand for reconsult/1. To accumulate clause for similarly named predicates, use consult/1.

```
?-co.
```

will reconsult again the the last reconsulted file.

Client/server interaction

To try out Jinni's client/server abilities, open 3 shell windows:

```
Window 1

java  Jinni
.............
?-run_server.


Window 2

?-there.
?-in(a(X)).


Window 3

?-there.
?-out(a(hello)).
```

When entering the out command in Window 3 you will see activity in Window 2. Through the server in Window 1, Window 3 has communicated the word "hello" returned as a result of the in query in Window 2!

Try out some more commands in Window 2 or Window 3:

```
?-listen.
?-talk.
?-for(I,1,5),out(a(I)),write(I),nl,fail.
?-all(a(X),Xs).
?-cin(a(X)).
?-the(X,eq(X,1),R).
?-the(Xs,ints(1,10,Xs),R).
?-the(_,listing,_).
?-all(I,for(I,1,3),Is).
```

Bi-directional Jinni / BinProlog talk

As client, it talks to BinProlog 6.33 and newer servers with "out", "cin" and "all" commands and with "the(Answer,Goal,Result)" or "all(Answer,Goal,ListOfResults)" remote execution queries. To try this out, start an unrestricted BinProlog server with:

```
?-trust.
```

As a server, Jinni understands out, all, cin, rd, in commands coming from BinProlog clients and uses multiple threads to synchronize them as well as "the(Answer,Goal,Result)" or "all(Answer,Goal,ListOfResults)" remote execution queries. The most natural use is a Java server embedded into a larger application which communicates with Prolog clients. Jinni-aware BinProlog clients or servers are available from

```
http://clement.info.umoncton.ca/BinProlog/CURRENT
```

Jinni's syntax

Jinni's syntax is based on standard Prolog terms, with the infix operators :- accepted at top level :

```
Clause --->              Head :- Body.
Clause --->              Head.
Head --->                Term
Body --->                CommaSeparatedTerms
CommaSeparatedTerms-->   Term {,Term}*
Term --->                Const | VARIABLES | CompoundTerm
CompoundTerm--->         IDENTIFIER(CommaSeperatedTerms)
Const--->                IDENTIFIER  |  INTEGER  |  REAL
```

IDENTIFIERs starting with lower case characters do not require quoting. If starting with upper case or containing non-alphabetical character they require single quotes. Variable start with upper case characters. Both Terms sent over the network follow the same syntactic convention and can be read/written by any Prolog or C program using BinProlog's socket communication protocol: messages contain a 4 byte header indicating their length, followed by a human readable Prolog term. Currently, Unicode zealous Java is forced into talking bytes (but this might change if requested by

users). A way to run under Jinni full Prolog programs with trickier operator syntax is to write them to a file using plain term syntax. A Prolog to Jinni converter is planned in the near future as well as a BinProlog based Jinni emulator.

As Jinni is heading towards plain Natural Language input through standard speech recognition software, its current internal syntax will be normally generated through a preprocessor. Jinni's parser and tokenizer are very small and based on Java's own parsing/tokenizing libraries. This allows relying on code already in browsers instead of having to load classes over the network.

Some of Jinni's built-in predicates

Jinni's built-in predicates follow the spirit of Prolog while simplifying when possible and adjusting to Jinni's syntax. We list here only those that are in some ways unusual for Prolog programmers.

- if(Cond,Then,Else): If Cond succeeds, executes Then with possible bindings provided by Cond. If Cond fails, the bindings it produces are undone and Else is executed.
- compute(Operation,Operand1,Operand2,Result): raw arithmetic processor.
  Used as in: compute('+',2,3,R) with R=5 or as in in compute('?',1,2,R),with R=-1 (with R=0 if Operand1==Operand2 and R=1 if Operand1>Operand2).
- new_engine(Goal, Answer, Handle): creates a new engine and returns a handle to it
- ask_engine(Handle,Answer): asks an engine given as a Handle for a new Answer
- stop_engine(Handle): makes sure the thread used by the engine is stopped. No further answers are available from the engine
- the(X,Goal,Result): executes Goal and returns the(A) where A is the first answer of Goal based on pattern X or no if no such answer is found. Note that the/3 starts execution speculatively before the first answer is actually required, so that this answer is usually ready by the time it is asked for. In particular, this allows the(_,Goal,_) to be used to run a Goal in background. The/3 is the unique primitive needed to define various constructs like if-then-else, not etc. See the file lib.pro for details. Negation as failure, for instance, is simply: not(G):-the(G,G,no). Although provided as a built-in, the/3 could have been defined solely in terms of new_engine/ask_answer/stop_engine. When the built-in there has set the focus of execution to a remote server thread, the/3 will actually act as a remote predicate call primitive. It has the same semantics but it will run the query on a remote processor. If different clauses are contained in the remote database, results returned might be different as well. If in doubt, use where(X) to check where the focus of execution currently is.
- all(X,Goal,Xs): is similar to the/3, except that the list of all answers X is collected in Xs, as Prolog's findall would do. When focus of execution is set to a remote server with "there", all/3 acts as a remote predicate call mechanism, similar in a way similar to the/3, except that it collects all the answers to the list Xs.
- enumerator(Goal,Handle): variation on the theme of engines. Returns a Handle which can be queried for successive elements. The last 2 arguments of Goal represent the state before and the state after the application of enumerate(Handle,Element). For instance, if decons(X,[X—Xs],Xs) is defined as a predicate and the Goal is decons(A,[1,2,3],[]), enumerate/2 will return the([2,3]), the([3]), the([]) and finally no if queried further

- enumerate(Handle,Element): companion to enumerator/2, returns the(X) if X is an element of the enumeration and no otherwise
- `load(<FILE>)`: loads ¡FILE¿.class Java class generated with `to_java(<FILE>)` from `<FILE>.pro` and previously compiled with the java compiler (e.g. javac). Mostly used to have Prolog code loaded from a server, transparently. As any Java class, such code can be compressed in the same jar file together with the Jinni kernel itself
- `to_java(<FILE>)`: converts `<FILE>.pro` to `<FILE>.java`.
- run_server: starts a server thread listening on current default port. Although technically possible, there's no obvious reason to start more than one sever inside a Jinni process. When interacting with clients, a Jinni server spawns separate threads for each interaction. Failure of one of them does not jeopardize the server as such or its global state. The server manages a synchronized blackboard which stores code in the form of dynamic prolog clauses and special terms to be used by in/1 and out/1 in Linda transactions
- set_host(HostIPAddressOrName): sets the name of the remote host
- set_thisPort(PortNumber): sets the port number (default 7001) where this server will run if started with run_server
- set_thatPort(PortNumber): sets the port number to the server to which execution will focus if there is used
- here: sets the focus of execution local to this process: the/3, all/3 will spawn local threads and in/1, out/1 etc. will now synchronize through the local blackboard
- there: sets the focus of execution to a remote server. the/3, all/3 will spawn remote threads and in/1, out/1 etc. will now synchronize through the remote blackboard. Note that set_host and set_that_port can be used to target a specific server. here and there are anaphoric constructs intended to avoid the need to use complex server addresses. Inspired by good old Natural Language: the best programming language ever invented:-) where(Place): returns "here" or "there" indicating local or remote mode
- out(Term): puts Term on a local or remote blackboard and resumes execution of threads waiting for a matching term
- cout(Term): the same as out(Term) but does nothing if a matching term is already on the (local or remote) blackboard
- in(Term): waits until a pattern unifiable with Term is put on the (remote or local) blackboard by an out/1 or cout/1 operation. The suspended thread (kept on the blackboard as a wrapped JavaObject) is resumed at that point, when the argument of out/1 or cout/1 is removed from the blackboard. If a matching term is already available, it is simply removed from the blackboard
- all(X,Xs): returns the list Xs of terms currently on the (remote or local) blackboard matching X
- rd(X): returns a term currently on the (remote or local) blackboard matching X and fails if no such term exists

- cin(X): removes and returns a term currently on the (remote or local) blackboard matching X and fails if no such term exists
- when(X): waits until a pattern unifiable with X is put on the (remote or local) blackboard by an out/1 or cout/1 operation. The suspended thread (kept on the blackboard as a wrapped JavaObject) is resumed at that point and returns the matching term, without removing it from the blackboard.
- retract1(Head): remove a matching clause from the (remote or local) blackboard, using cin/1
- `'+'(Assumption)`: adds a backtrackable assumption to the store associated to the current AND branch
- `'-'(Assumption)`: selects and removes a matching backtrackable assumption from the store associated with the current AND branch
- `'#'(Assumption)`: consumes the head of the current list of assumptions: useful as a DCG grammar connect/3 operation, in conjunction with dcg_def/1 and dcg_val/1 which respectively set and consult the state of the current DCG stream

From Jinni to Java and back

The simplest way to call Java from Jinni and back is using java Strings for information exchange (see file Jinni.java).

```
/**
  Example of Jinni calling a Java based
  user defined predicate of arity 2,
  which, in turn, calls back Jinni,
  to consult a program and run a goal.

  Try it with: ?-nqueens(5,Answer).
*/
class nqueens extends UserFunBuiltin {
  nqueens() {super(2);}

  public int exec(Prog p) {
    long n=((Int)getArg(0)).longValue();
    IO.println("Got "+n+ " queens!\n Consulting q8.pro");
    Top.askJinni("co(q8)");
    String result=Top.askJinni("queens("+n+",X)");
    IO.println("Result:"+result);
    return putArg(1,new Const("finished"),p);
  }
}
```

A more interesting example, with data structure exchanges follows:

```
/**
  Example of Jinni calling a Java based
```

```
  user defined predicate of arity 3,
  which, in turn, calls back Jinni,
  to consult a program and run a goal.

  <p>
  Try it with:
  ?- sumvec2(v(1.1,2.2),v(10,20),R).
  R=v(11.1,22.2)
  </p>
*/
class sumvec2 extends UserFunBuiltin {
  sumvec2() {super(3);}

  public int exec(Prog p) {
    Fun v1=(Fun)getArg(0);
    Fun v2=(Fun)getArg(1);
    double v1x=((Num)v1.getArg(0)).getValue();
    double v1y=((Num)v1.getArg(1)).getValue();
    double v2x=((Num)v2.getArg(0)).getValue();
    double v2y=((Num)v2.getArg(1)).getValue();
    double rx=v1x+v2x;
    double ry=v1y+v2y;
    Fun r=new Fun("v",new Real(rx),new Real(ry));
    return putArg(2,r,p);
  }
}
```

A choice-point library for backtrack programming

Pierre-Etienne Moreau
LORIA-CNRS & INRIA-Lorraine
BP 239
54506 Vandœuvre-lès-Nancy Cedex, France
E-mail: Pierre-Etienne.Moreau@loria.fr

Abstract

Implementing a compiler for a language with nondeterministic features is known to be a difficult task. This paper presents two new functions setChoicePoint and fail that extend the C language to efficiently handle choice point management. Originally, these two functions were designed to compile the ELAN strategy language. However, they can be used by programmers for general programming in C. We illustrate their use by presenting the classical 8-queens problem and giving some experimental results. Algorithms and implementation techniques are sufficiently detailed to be easily modified and re-implemented.

1 Introduction

In the area of formal specifications, rewriting techniques have been developed for two main applications: prototyping algebraic specifications of user-defined data types and theorem proving related to program verification. In this context we are interested in nondeterministic computation and deduction.

Term rewriting is nondeterministic in the sense that there may be several reductions starting from one initial term and producing different results. Rewriting logic [12] gives a logical background and raises new interesting problems concerning the efficient implementation of nondeterministic rewriting which needs backtracking. This is similar to the implementation of logic programming languages, but a significant difference is the fact that rewriting rules can be applied inside the terms. Moreover the formalism used to prune the search space is different from that of logic programming languages.

In this paper we present a new technique for compiling the specific control flow in programs during the backtracking. Our method preserves the efficiency of deterministic computations and is of more general interest; it could be used in implementations of constraint solvers, imperative languages with backtracking such as Alma-0 [14, 2], the WAM [18, 1] and Prolog-like languages.

A first implementation of our techniques has been done by Marian Vittek in 1996. The experimental results, presented in [16], show that nondeterministic rewriting can be implemented as efficiently as the best current implementations of functional and logic programming languages. This paper presents a formalisation of the implementation and gives detailed algorithms to re-use, adapt and improve the proposed method. It took great benefit from the idea and comments of Marian Vittek.

Section 2 illustrates the behaviour of usual functions used to implement backtracking in nondeterministic computations. Section 3 gives a brief overview of existing techniques for implementing

choice points and compiling languages with nondeterministic features into C. Section 4 presents algorithms for the two new proposed functions that implement an efficient backtracking control flow: setChoicePoint and fail. Then Section 5 illustrates on one example how the use of setChoicePoint and fail can help in solving in a natural way algorithms that involve search. Some experimental results show that the proposed method can be a good alternative to compile in an efficient way languages that involve nondeterministic features.

2 Basic choice point primitives

Backtracking is a well-known approach to implement nondeterministic computations. In compilation techniques, two functions are usually needed: the first one to create a choice point and save the execution environment. The second one to backtrack to the last created choice point and restore the saved environment. Many languages that offer nondeterministic capabilities provide similar functions: for instance world+ and world- in Claire [6], try and retry in the WAM, onfail, fail, createlog and replaylog in the Alma-0 Abstract Machine [14], setChoicePoint and fail in ELAN [16]. Recently, a new approach to the implementation of tabling for prolog [9] has been proposed. The authors suggest to extend a prolog implementation by adding some new built-in predicates. For their purpose, a similar idea as the one presented here has been explored in a different context.

We propose to extend the C language by adding two control flow functions: setChoicePoint and fail. setChoicePoint returns the integer 0 when setting a choice point, and the computation goes on. When the function fail is called, it performs a jump into the last call of setChoicePoint and it returns the integer 1. These functions can remind the pair of standard C functions `setjmp` and `longjmp`. However, the `longjmp` can be used only in a function called from the function setting `setjmp`. Functions setChoicePoint and fail do not have such a limitation.

The following program, written in C, illustrates the behaviour of these two new functions. The **Output** column shows the result obtained when executing the program:

Program	**Output**

`static int counter=0;`
`main() {`
`  if(setChoicePoint()!=0) exit(0);`
`  f();`
`  fail();`
`}`
`f() {`
`  int result, locvar=0;`
`  result=setChoicePoint();`
`  printf(result,locvar,counter);`
`  locvar++; counter++;`
`  printf(locvar,counter);`
`}`
	`result=0, locvar=0, counter=0`
	`locvar=1, counter=1`
	`result=1, locvar=0, counter=1`
	`locvar=1, counter=2`

When setting a choice point, only local variables are saved. If a failure occurs, only local variables are restored to the value they had when setting the choice point. When executing the example program, a first choice point is created and the computation goes on. The function `f()` is called and `locvar` is initialised to 0. Then a second choice point is created, `result` and `locvar` are saved, printed, incremented and printed again. Before executing the first fail, `counter=1` and

`locvar=1`. Then a backtrack is performed: the function fail restores the last saved environment (`f` is re-activated, `locvar=0`) and transfers the control to setChoicePoint function which returns the integer 1. This explains why the third line is `result=1`, `locvar=0`, `counter=1`. The function fail is called again: a backtrack to the first set choice point is performed; the conditional test is evaluated to true and the program stops.

3 Known choice point implementations

The implementation of choice point management most often involves two mechanisms: first, an environment stack, called the *trail*, to save local variable values and a continuation address; second, a control flow handler to perform the jump to the saved continuation address when backtracking.

A number of techniques for implementing branching schemes have been proposed over the years, especially in the functional and logic programming communities. Several languages use C as target language such as Cg, Icon, Janus, Erlang, KL1, and Mercury. Their different compilation schemes are presented in [5, 17, 8, 7, 10].

Among these techniques, the simplest method implements branching using a C `goto` statement. However problems arise because indirect branching is not available in standard C and also because a `goto` instruction can only do a jump into its function scope. This leads to a C program composed of a unique huge function with a `switch` statement to simulate indirect gotos. This compilation scheme is unrealistic since it makes impossible separate compilation. Moreover, collecting all codes into one C function affects compilation time and compiler's ability to perform register allocation.

The second method consists in translating each labelled block by a C function that returns a continuation address. Those functions are managed by a driver function that does the necessary dispatching to transfer control from one function to another. Consequently, this method is not the most optimised one but is suited for standard C and separate compilation.

A third well-known existing scheme consists in using non standard C features that are supported by the GNU C compiler [15]. The `gcc` compiler makes it possible to take the address of labels, and later on to jump to those addresses. It also offers the possibility to insert inline assembly code, and to specify the assembly name of a function. With those extensions it is now possible to translate any branching by a `goto` statement.

When using one of the three presented schemes, the structure of a program is not taken into account. Parameter passing cannot be done in a natural way. Instead, global variables are used to communicate arguments from caller to callee. That is why local variables have to be saved before doing a jump and classical function calls have to be simulated. As a matter of fact, it is very difficult for a human to write a program in these conditions and the presented compilation schemes can only be used in automatically generated programs.

Even if the three presented methods are efficient and well-designed to implement a WAM-like abstract machine, it is still difficult to use them to design new compilation schemes because resulting programs are often difficult to read.

4 New choice point management

In this section we present algorithms and implementation techniques of the two new functions setChoicePoint and fail.

The key idea of our approach is to use the system stack and only one environment stack to store values that have to be saved. Consequently, there is no restriction on the usage of local variables

and parameter passing when programming with setChoicePoint and fail.

We first present an approach which consists in extending the two standard C functions `setjmp` and `longjmp`. Then we define some notations in order to give detailed algorithms of the second method which minimises the size of memory blocks that have to be saved (resp. restored) when setting a choice point (resp. performing a failure).

4.1 setJump: an extension of setjmp

The standard C library defines two low level functions `setjmp` and `longjmp`. The first one saves the current execution context (machine registers and a return address) in a `jmp_buf` structure. The second one can restore any stack context that has been saved in a `jmp_buf` structure by `setjmp`. After the `longjmp` runs, the program execution continues as if the corresponding call to the `setjmp` function had just returned the value specified in the `longjmp` call. The result of `longjmp` is undefined if the function that made the corresponding call to the `setjmp` has already returned.

We propose first to extend `setjmp` and `longjmp` into setJump and longJump to suppress such undefinedness: the whole stack system (memory block between the *base pointer* and the *stack pointer*) has to be saved in a Jump_buf structure when calling setJump.

Given an integer different from 0 and a valid Jump_buf structure, the longJump function restores registers and the whole system stack, and then the integer parameter is returned.

Depending on the architecture, these two functions may be implemented in C: `setjmp` and `longjmp` are used to save and restore registers[1] and `memcpy` is used to copy memory blocks. This approach was successful on a PC under Linux and a DEC Alpha-Station but the result is not really safe: as mentioned previously, `longjmp` is used in a non-standard way; since in is not possible in C to get the *base pointer* and the *stack pointer*, some heuristics are used; and furthermore, the behaviour is not stable when using advanced optimisation options such as `gcc -O6`. This why we recommend this approach only to get a first easy implementation, but to get a safe implementation, setJump and longJump have to be re-implemented in assembly language. There is nothing surprising, because `setjmp` are `longjmp` are themselves implemented in assembly language.

Some processors, such as Sparc, are based on a window register architecture. In this case, it is not possible to save and restore the window position: the corresponding assembly instruction must be executed in privileged mode (not accessible by users). This restriction makes impossible the implementation of setJump and longJump on such architecture, however, in Section 4.4, we present a general algorithm for setChoicePoint and fail that can be implemented on almost any architectures.

4.2 A first implementation of setChoicePoint and fail

setChoicePoint and fail are somehow restrictions of setJump and longJump because the fail function always restores the context of the last set choice point. This special case allows us to design smarter and more efficient algorithms. A naive implementation of setChoicePoint and fail consists in reusing setJump, longJump implementations and storing Jump_buf structures in a LIFO data structure (the *trail* itself).

Let us remark that saving the whole system stack is too expensive. In average, only a small part of the system stack needs to be changed when a failure occurs. For example, let us consider the following program, where dots denote irrelevant instructions:

[1] a similar idea is used in the BDW Garbage Collector [3]

```
void main() {
  ...
  g(i);
  ...
  fail();
  ...
}
```

```
void g(int arg) {
  ...
  setChoicePoint();
  ...
}
```

An execution of the main function creates the stack frame of main in the system stack. Then main calls g, this pushes the stack frame of g onto the stack. So, when the choice point is set, two stack frames (main and g) are on the stack (see Figure 1). After this, when leaving g, its stack frame is freed and execution continues in main by fail. But, at this moment, the stack contains the stack frame of main. So, only the stack frame of g has to be restored onto the current system stack to reconstitute the stack as it was at the moment of the choice.

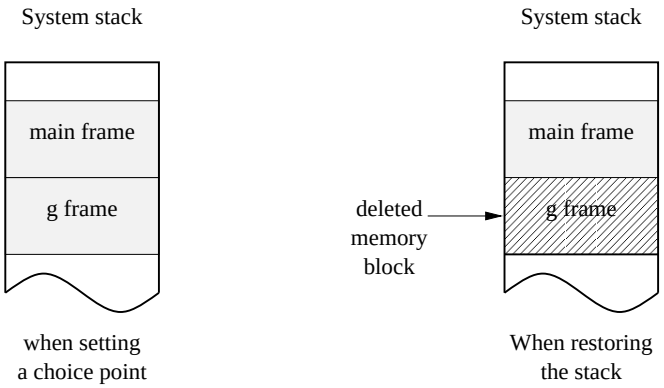

Figure 1: It is useless to copy the whole system stack

This example illustrates the fact that this is useless to copy the whole system stack because only the stack frames of some functions are concerned. The next implementation of setChoicePoint uses this idea.

4.3 Notations

In order to give a detailed algorithm several notations are needed. They are useful to clearly compute memory blocks that have to be saved and restored.

Let us first consider a simple execution model that consists in viewing a program execution as a sequence of instruction executions, function calls, and function returns.

Let us define watch points $\tau_1, \ldots, \tau_{n_\tau}$ as the first executed instruction after a function call or a function return.

When running the program presented in Section 4.2, the function main calls g which calls setChoicePoint. The first executed instruction when entering main, g and setChoicePoint corresponds to watch points τ_1, τ_2 and τ_3 respectively. The first executed instruction after setChoicePoint corresponds to τ_4. Other watch points τ_5, τ_6 and τ_7 are defined similarly. The program execution and watch-points are illustrated in Figure 2.

When executing a function, there is a corresponding environment called ϵ_j that contains information about the current executing function. This information is available through different functions:

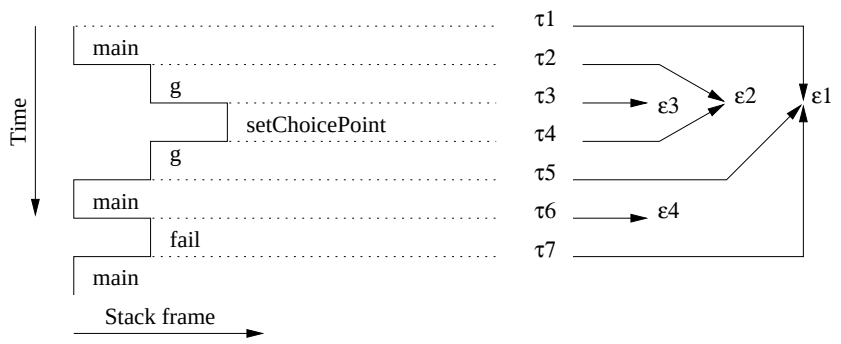

Figure 2: Setting watch points and environments

- $fp(\epsilon_j)$ to get the frame pointer value, i.e. the address which indicates the beginning of the stack frame;
- $ra(\epsilon_j)$ to get the return address value, i.e. the address from the program to which the program counter should be restored;
- local variables are also saved in an environment.

To each watch point τ_i is associated an environment ϵ_j (i and j are not equal in general because several watch-points may be associated to a given environment). The notion of environment and the correspondences between (τ_i) and (ϵ_j) are illustrated in Figure 2.

A stack pointer value $sp(\tau_i)$, which is the current top of the system stack, is associated to each watch point τ_i.

Let us define Env_at as the surjective function from $\{\tau_1, \ldots, \tau_m\}$ to $\{\epsilon_1, \ldots, \epsilon_n\}$ that maps each (τ_i) to its environment (ϵ_j). The set $\{\tau_1, \ldots, \tau_m\}$ is totally ordered by indices values: $\tau_1 < \cdots < \tau_m$.

In the previous example we have:

- $Env_at(\tau_1) = Env_at(\tau_5) = Env_at(\tau_7) = \epsilon_1$;
- $Env_at(\tau_2) = Env_at(\tau_4) = \epsilon_2$;
- $Env_at(\tau_3) = \epsilon_3$;
- $Env_at(\tau_6) = \epsilon_4$.

Environments are organised as in a block structure language, thus, the notion of *embedded* environment can be defined: for a given ϵ_j, the embedded environment $emb(\epsilon_j)$ is the environment in which the function associated to ϵ_j is called.

In the previous example, we have:

- $emb(\epsilon_3) = \epsilon_2$
- $emb(\epsilon_2) = emb(\epsilon_4) = \epsilon_1$
- $emb(\epsilon_1)$ is not defined

Let us define for $\epsilon \in \{\epsilon_1, \ldots, \epsilon_n\}$ the function that returns the minimum element of the inverse image of Env_at: $Min(\epsilon) = min(Env_at^{-1}(\epsilon))$

From a practical point of view, τ_i is a program's address that contains an assembly instruction. ϵ_j is a stack frame associated to the current executed C function and $Min(\epsilon)$ is the first executed instruction when entering a C function.

Let $<$ be a total ordering on addresses such that the base pointer bp is the minimum. Let l_1, l_2 be two addresses of the system stack. If $l_1 < l_2$, l_1 is said to be closer to bp than l_2. When $l_1 < l_2$, $[l_1, l_2[$ is the memory block between those two addresses.

4.4 Advanced algorithm for setChoicePoint and fail

The goal of the algorithm sketched in Section 4.2, is to minimise the number of saved stack frames. The main idea consists in implementing a special *handle function* which saves the top stack frame of the system stack.

Each time a nondeterministic function[2] returns to the caller function, the last jump is redirected to this *handle function*: return addresses of nondeterministic functions are successively (first by setChoicePoint and then by the handle function itself) modified to point to this handle function. This guarantees that the handle function is called each time a nondeterministic function executes the return instruction. These calls save the corresponding stack frames which are then used to recover the original system stack by the fail function. The setChoicePoint algorithm performs two actions: save machine registers and activate the handle function.

Let τ_i be a choice point: setChoicePoint saves registers, which include $ra(Env_at(\tau_i))$, $sp(\tau_i)$ and $fp(Env_at(\tau_i))$, pushes the special mark endReg into the trail and then jumps into the handle function: saveFrame.

Let ϵ_j be the environment of the function which did the branching to saveFrame (when creating the choice point, $Env_at(\tau_i) = \epsilon_j$):

- saveFrame saves the frame of the function that called the function associated to ϵ_j. Let us call $emb(\epsilon_j)$ its environment. The saved frame is $[fp(emb(\epsilon_j)), sp(Min(\epsilon_j))[$. Then saveFrame pushes the special mark endFrame into the trail,
- saveFrame replaces the caller's return address (saved in $emb(\epsilon_j)$) by the saveFrame procedure's address. Thus, each time a function returns, the save frame handler is activated.

The handle function saveFrame may be called several times before a fail occurs. In this case, several frames are saved into the trail. Figure 3 illustrates this possibility: two choice points have been created and three frames have been saved (two of them are associated to the first choice point). This situation occurs when the function calling setChoicePoint had returned.

The fail algorithm rebuilds the system stack with saved frames until the endReg code is found. Then, the registers and the stack pointer are restored, and the integer 1 is returned.

Note that fail removes the last created ChoicePoint and restores the system stack to its initial state. Assuming that a return address is saved in the system stack, the return address (that was modified to saveFrame) is restored to its initial value by recovering the stack. Consequently, the save frame handler is no longer active.

[2] a function is said to be nondeterministic if a choice point is set or a nondeterministic function is called during its execution

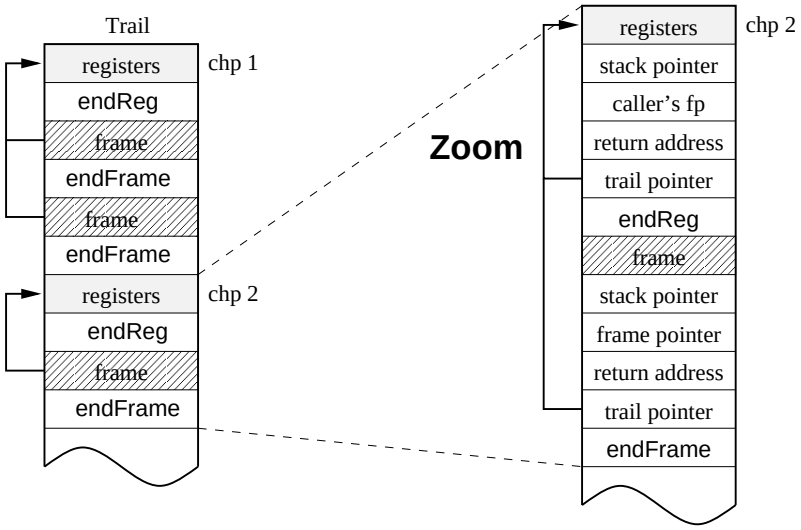

Figure 3: Trail stack state after setting a choice point

4.5 Detailed implementation

In this section we give a low level description that corresponds to the assembly language implementation. The next algorithm describes the implementation of setChoicePoint:

Algorithm 1 setChoicePoint

load the caller's frame pointer ($fp(Env_at(\tau_i))$) into reg_0
load the trail pointer into reg_1 and reg_2
push non specific registers into the trail (referenced by reg_2)
push the stack pointer (sp), the caller's frame pointer reg_0, the return address (ra) and the initial trail pointer reg_1 into the trail
push the endReg code
prepare the return value: 0
do a jump to saveFrame

The following figure illustrates the state of the trail stack after executing setChoicePoint. This corresponds to the top level of the zoom part given in Figure 3.

reg_1	$\rightarrow$	... saved registers ...
		stack pointer
		caller's frame pointer: reg_0
$reg_2 - 3$	$\rightarrow$	return address
$reg_2 - 2$	$\rightarrow$	initial trail pointer: reg_1
		endReg code
reg_2	$\rightarrow$	

This saved information is used by the handle function saveFrame to save stack frames and update

links to the corresponding choice point. Note that the notation $reg_2 - 3$ is used to specify a base address. In this example, the value 3 is subtracted from the base register reg_2 value to denote a new address whose contents is the saved return address.

The following algorithm describes a pseudo assembly code of the saveFrame handle function:

Algorithm 2 saveFrame

load the trail pointer into reg_2
load the caller's frame pointer into reg_0
restore the saved return address $(reg_2 - 3)$ into reg_4
restore the saved trail pointer $(reg_2 - 2)$ into reg_5
push the frame (memory block $[reg_0, sp[$) into the trail
push the stack pointer and caller's frame pointer reg_0
load the frame pointer of the previous memory block $(reg_5 - 4)$ into reg_3
if $reg_3 = reg_0$ **then**
  {the return address is already into the trail. The link to the previous block has to be followed}
  load the old return address $(reg_5 - 3)$ into reg_4
  load the beginning of the previous block $(reg_5 - 2)$ into reg_5
end if
push reg_5 and reg_4 and the endFrame code into the trail
update the trail pointer
modify the caller's return address to the save frame handler saveFrame

The state of the trail stack after executing saveFrame is described in Figure 3. Let us notice that top and bottom parts of the zoom stack have similar structures: values are saved in the same order.

The following pseudo assembly code describes the implementation of fail:

Algorithm 3 fail

load the trail pointer into reg_2
`recoverFrame`: load the current code $(reg_2 - 1)$
if it is a endReg code **then**
  branch to `returnInCP`
end if
restore the stack pointer $(reg_2 - 5)$
restore the frame pointer $(reg_2 - 4)$
compute the saved frame size and restore it
branch to `recoverFrame`
`returnInCP`: update the trail pointer
restore saved registers, including the return address register
return the value 1

4.6 Portability

The two functions setChoicePoint and fail are implemented with a 200 lines assembly library. This could be a source of difficulty for portability. However, in practice, the library can be easily implemented on a new architecture. In fact, only three specific access functions are required: $fp()$, $sp()$ and $ra()$ which return respectively the current frame pointer, the current stack pointer and

the return address. The nondeterministic library[3] has been implemented on three different architectures: Sparc, Intel and DEC-Alpha. It has not been ported yet to HP-PA or MIPS processors, but it should not be a problem even if the stack grows from low to high addresses.

It is well-known that efficient implementations of nondeterministic library usually take benefit of gcc extensions and use inline statements to add assembly labels in the generated code. Therefore, our approach is not less portable than classical implementations of choice point managements.

5 Imperative programming with backtracking

Several languages such as 2LP [11] and Alma-0 [14, 2] have been developed to combine advantages of logic and imperative programming in order to deal in a natural way with algorithmic problems that involve search. Alma-0 extends imperative programming with some features inspired by the logic programming paradigm. For instance:

- use of boolean expressions as statements and vice versa;
- the ORELSE statement starts by proceeding through the first branch. If the computation eventually fails, backtracking takes place and the computation resumes with the next branch in the state in which the previous branch was entered;
- the SOME statement extends the ORELSE construction to generate a number of choice points determined only at run-time;
- the FORALL statement introduces a controlled form of iteration over the backtracking.

With these extensions, the program that computes all solutions of the N-queens problem can be expressed in a natural way. Let us remind that the N-queens problem consists in placing N queens on the chess board so that they do not attack each other.

```
FOR column := 1 TO N DO
  SOME row := 1 TO N DO
    FOR i := 1 TO column-1 DO
      x[i] <> row;
      x[i] <> row+column-i;
      x[i] <> row+i-column
    END;
    x[column] = row
  END
END
```

```
BEGIN
(* print all solutions *)
  nrSols := 0;
  FORALL queens(b);
    DO
      (* print solution: b *)
    END;
END queens.
```

In the Alma-0 environment, the program is first translated into the Alma-0 abstract machine language (AAA), and then, each AAA instruction is compiled into C statements. The used backtracking mechanism corresponds to the first method described in Section 3. Thus, modular compilation is not possible and optimisations performed by the C compiler are not optimal.

We think that setChoicePoint and fail could be a good alternative to implement AAA backtracking operations. Even more, setChoicePoint and fail seem to be well-suited to compile directly an Alma-0 program to C without any intermediate abstract machine. Indeed, the previous Alma-0 program fragment can be easily translated in C extended with our two functions setChoicePoint

[3] available at **http://www.loria.fr/equipes/protheo/PROJECTS/ELAN**

and fail. The SOME statement is replaced by the call of the from1ton function and the boolean expressions are translated into conditions and failures.

```
void main() {
 backTrackInit();
 queens(12);
 fail();
}

int from1ton(int n) {
 int i;
 for(i=1;i<n;i++) {
  if(setChoicePoint()==0) {
   /* set a choice point
      and return i */
   return(i);
  }
 }
 return(i);
}
```

```
queens(int n) {
 int col,row,i;
 for(col=1;col<=n;col++) {
  row=from1ton(n);
  for(i=1;i<col;i++) {
   if(array[i]==row ||
      array[i]==row+col-i ||
      array[i]==row+i-col) {
       /* choose another row */
       fail();
   }
  }
  array[col] = row;
 }
 /* print solution: array[] */
}
```

Remark that the library is initialised by a call to the backTrackInit function which creates an initial choice-point in order to prevent using fail before setChoicePoint.

In order to show the potential of the approach, the N-queens solving procedure has been implemented in several languages: standard C, C extended with setChoicePoint and fail, Mercury, Wamcc, Alma-0 and 2LP. In each case, even if it is not the best one, the same searching strategy has been used: a position for the current queen is chosen and is not removed from the list of remaining positions. The experimental results presented in Figure 4 show that the presented approach is as efficient as Mercury. But, you should not deduce that replacing its backtracking management by the proposed library should improve the efficiency. We can only deduce, from these benchmarks, that the proposed approach is a good compromise between simplicity and efficiency.

Let us mention that the use of setChoicePoint and fail is not restricted to the implementation of toy examples such as the N-queens problem: they are intensively used in a more general context related to the compilation of rewriting systems with strategies. In this case, they are frequently used in generated programs that may consist of several thousands lines of C code.

6 Concluding Remarks

The previous section has shown that setChoicePoint and fail are designed to extend the C language in order to deal in a natural way with problems that involve search. setChoicePoint and fail can also be used to compile languages with some features inspired by the logic programming paradigm. This is a good alternative to avoid using an abstract machine and to get interesting performances. The two functions were originally designed to compile the ELAN language. Some examples of compilation schemes are given in [13] (see appendix A).

Their "plug-in" designs make them easy to use: conventional programming techniques such as function calls, local variables, parameters passing, modular compilation are compatible with the proposed C language extension. Having a readable code is a key point to be able to design new complex compilation schemes.

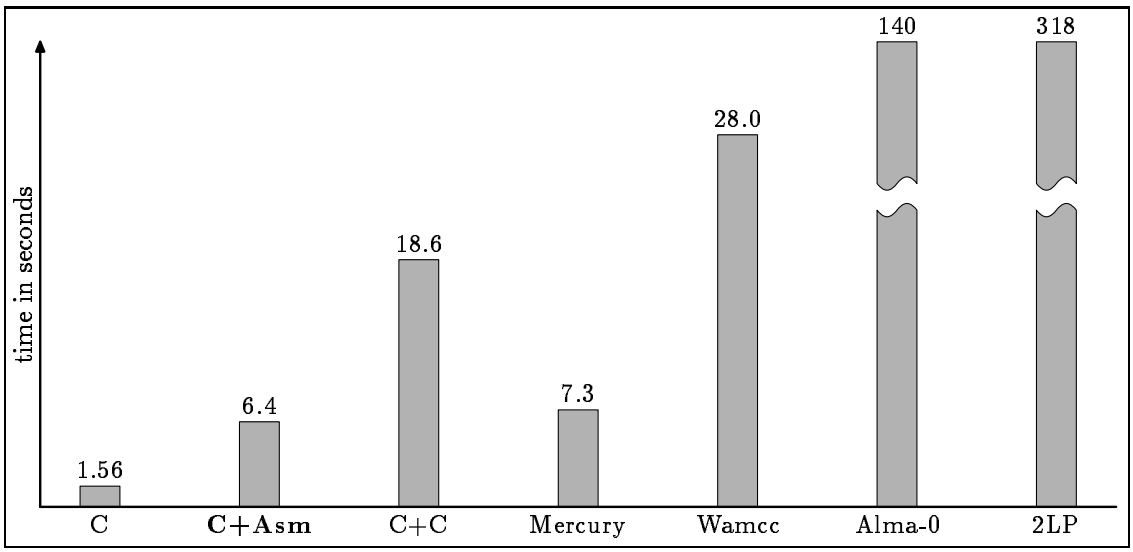

Figure 4: Several implementations of the same algorithm are compared in order to show the potential of the approach. The fastest implementation of the N-queens solving procedure is implemented in standard C with a recursive function call. Two others implementations have been done in C extended with setChoicePoint and fail: the first one `C+Asm` use the assembly version of the proposed library and the second one `C+C` use the C version presented in 4.1.

When designing compilation schemes for a new language, the backtracking management, if any, is always a difficult task to solve. The proposed approach seems to be a good compromise between simplicity and efficiency: it provides both high level concepts such as choice points and backtracking, and a low level implementation (in assembly) to get good performances. Experimental results show that the proposed implementation is comparable with an ad-hoc approach like the one used in optimised WAM implementations.

Acknowledgements

I would like to thank the Protheo group and especially Peter Borovanský, Claude Kirchner, Hélène Kirchner and Laurent Vigneron for helpful discussions and comments. I would especially like to thank Marian Vittek and Bart Demoen for their support and enlightening conversations.

References

[1] H. Aït-Kaci. The WAM: a (real) tutorial. Technical report 5, Digital Systems Research Center, Paris (France), Jan. 1990.

[2] K. R. Apt and A. Schaerf. Search and imperative programming. In *24th POPL*, pages 67–79, 1997.

[3] H. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18:807–820, 1988.

[4] P. Borovanský, C. Kirchner, H. Kirchner, P.-E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In J. Meseguer, editor, *Proceedings of the first international workshop on rewriting logic*, volume 4 of *Electronic Notes in TCS*, Asilomar (California), Sept. 1996.

[5] T. Budd. An implementation of generators in C. *Computer Languages*, 7:69–87, 1982.

[6] Y. Caseau and F. Laburthe. Introduction to the CLAIRE programming language. Technical report 96-15, LIENS Technical, September 1996.

[7] P. Codognet and D. Diaz. wamcc : Compiling Prolog to C. In *Proceedings of International Conference on Logic Programming*, Tokyo, Japan, 1995. The MIT press.

[8] B. Demoen and G. Maris. A comparison of some schemes for translating logic to C. In *Workshop on Implementations of the 11th International Conference of Logic Programming*, Santa Margherita, Italy, 1994. MIT Press.

[9] B. Demoen and K. Sagonas. CAT: the Copying Approach to Tabling. Personal Communication, 1998.

[10] F. Henderson, T. Conway, and Z. Somogyi. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29:17–54, October-December 1996.

[11] K. McAloon and C. Tretkoff. 2LP: Linear programming and logic programming. In P. Hentenryck and V. Saraswat, editors, *Principles and Practice of Constraint Programming*, pages 101–116. The MIT press, 1995.

[12] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[13] P.-E. Moreau. Compiling nondeterministic computations. Technical Report 98-R-005, CRIN, 1998.

[14] V. Partington. Implementation of an Imperative Programming Language with Backtracking. Technical Report P9714, University of Amsterdam, Programming Research Group, 1997. Available by anonymous ftp from ftp.wins.uva.nl, file pub/programming-research/reports/1997/P9712.ps.Z.

[15] R. Stallman. *Using and porting the GNU CC compiler*, 1995.

[16] M. Vittek. A compiler for nondeterministic term rewriting systems. In H. Ganzinger, editor, *Proceedings of RTA'96*, volume 1103 of *Lecture Notes in Computer Science*, pages 154–168, New Brunswick (New Jersey), July 1996. Springer-Verlag.

[17] S. Wampler and R. Griswold. The implementation of generators and Goal-Directed Evaluation in Icon. *Software-Practice and Experience*, 13:495–518, 1983.

[18] D. H. D. Warren. An abstract Prolog instruction set. Technical Report 309, SRI International, Artificial Intelligence Center, 1983.

A Appendix: Compiling nondeterministic rewriting

A.1 Short presentation of ELAN

Originally setChoicePoint and fail were designed to compile the ELAN language. In ELAN, a logic can be expressed by specifying its syntax and its inference rules described by conditional rewrite rules. In order to control the execution, the notion of strategy [4] has been introduced.

ELAN provides the following main features: a semantics based on many-sorted rewriting logic; a powerful language to express strategies of rewrite rule application, including dont-care and dont-know choices on strategies; a general pre-processor making easier the translation of a logic into rewriting logic; a standard library to facilitate user developments; some modular constructions via local or global importations as well as parametric modules; a generic mixfix and user-definable syntax; associative commutative (AC) operators; and a very efficient compiler for ELAN programs.

A.2 Rewrite rule

In ELAN, a rewrite rule is a pair of terms with some conditions and local assignments:

[rule name] < left-hand side term > ⇒ < right-hand side term >
(**if** < boolean term >
| **where** < variable > := (strategy) < term >)*

Applying a rule on a term consists in matching the left-hand side, then verifying that conditions are satisfied and evaluating where statements. A special backtracking mechanism exists: if a condition is not satisfied or a local assignment leads to a *failure*, the previous nondeterministic where (if there is one) has to select another solution. If there is no way to satisfy conditions and compute local assignments, the rule cannot be applied.

A.3 Strategies

Strategies is one of the main originality of ELAN. In practice, a strategy is a way to describe which computations the user is interested in. The application of a strategy to a term results in the (possibly empty) set of all terms that can be derived from the starting term using this strategy. When a strategy returns an empty set of terms, we say that it *fails*.

Thus the language provides a way to handle this non-determinism. This is done using the basic strategy operators: dc (dont care choose) and dk (dont know choose). For a labelled rewrite rule ℓ, the strategy dk(ℓ) returns *all* possible results. The implementation handles these several results by an appropriate back-chaining operation.

Two strategies can be concatenated: this means that the second strategy will be applied on all results of the first one. In order to allow the automatic concatenation of the same strategy, ELAN offers the two iterators iterate and repeat. The strategy iterate corresponds to applying zero, then one, then two,..., n times the strategy to the starting term, until the strategy fails. Thus (iterate(s))t returns $\bigcup_{n=0}^{\infty}(s^n)t$. The strategy repeat iterates the strategy until it fails and returns only the terms resulting of the last unfailing call of the strategy.

In order to illustrate how strategies work, let us consider the example consisting of the extraction of the constituents of a list:

```
rules for elem
   e : elem;
   l : nelist;
bodies
   [extract]   element(e.l)    => e              end
   [extract]   element(e.l)    => element(l)     end
end
```

If we assume furthermore that the constants `a,b,c` are of sort `elem`, then:

- repeat(dk(extract)) (a.b.c) yields the set {a, b, c},
- iterate(dk(extract)) (a.b.c) yields the set
  {element(a.b.c), a, element(b.c), b, element(c), c}.

A.4 Compilation of strategies

The ELAN compiler transforms each strategy s into a C function `str_s` with one argument. An application of the strategy s to the term t is compiled into a call to `str_s`, whose argument is a pointer to t. When the strategy is deterministic, i.e. has at most one result, the function returns a pointer to the resulting term. Otherwise, backtracking is needed to implement nondeterministic computations.

Let us illustrate the use of setChoicePoint and fail on a few compilation schemes of nondeterministic computations in ELAN.

Any computation in ELAN begins with a call to setChoicePoint to set an initial choice point and ends with a call to the fail function. So coming back to the initial choice point with a failure means that the computation has terminated and all results have been returned.

A.4.1 Nondeterministic rule application.

This strategy tries all rules and returns all possible results; local assignments and conditions have to be evaluated with the special back-chaining convention.

For each rule, a choice point is created and matching, checking conditions and local affectations are attempted. If there is a failure in one of these steps, the choice point is removed and the next rule is tried. If the rule is applied, the choice point is kept for further backtracking. When the control comes back to the initial choice point, all possible results have been returned. This compilation scheme generalises in a straightforward way to the compilation of nondeterministic application of strategies in a dk($S_1, \ldots, S_n$).

Let us remind that the "**if** setChoicePoint=1 **then**" statement means "if the control comes from a fail, delete the ChoicePoint and do the *then* part".

Algorithm 4 dk($r_1, \ldots, r_n$)

```
if setChoicePoint=1 then
   try next rule
end if
compilation of conditions and local affectations
right-hand side construction
```

A.4.2 Nondeterministic iterator.

Iterating a strategy has a simple compilation scheme. In an infinite loop, a choice point is created and an exit from the loop (the **break** statement) is done. The current term is returned. So applying iterate(S) on a term t returns t first (zero application of S). When coming back with a failure which means that the computation of the n-th iteration is finished, if the strategy S can be applied once on the result, a new choice point is created, and $S(S^n)(t)$ is returned. Else the failure is propagated.

Algorithm 5 iterate(S)

```
loop
  if setChoicePoint=0 then
    break
  end if
  code of S
end loop
```

Compiling Resources in a Linear Logic Programming Language

Mutsunori Banbara
Department of Mathematics,
Nara National College of Technology,
22 Yata, Yamatokoriyama,
639-1080, Japan
banbara@libe.nara-k.ac.jp

Naoyuki Tamura
Department of Computer and Systems Engineering, Kobe University
1-1 Rokkodai, Nada, Kobe,
657-8501, Japan
tamura@kobe-u.ac.jp

Abstract

There have been several proposals for logic programming language based on linear logic: Lolli [6], Lygon [5], LO [3], LinLog [2], Forum [8]. In addition, BinProlog allows the use of linear implications of affine logic (a variant of linear logic) [12, 13]. In these languages, it is possible to create and consume resources dynamically as logical formulas. The efficient handling of resource formulas is therefore an important issue in the implementation of these languages.

In [10], N. Tamura and Y. Kaneda proposed an abstract machine called LLPAM which is an extension of the standard WAM [14, 1] for a linear logic programming language called LLP. LLP is a superset of Prolog and a subset of Lolli. However, in the original LLPAM design, a resource formula was stored as a term in a heap memory and was not compiled into LLPAM code.

In this paper, we describe an extension of LLPAM for compiling resource formulas. In our extension, resources are compiled into *closures* which consist of compiled code and a variable binding environment.

1 Introduction

Linear logic, a new logic proposed by J.-Y. Girard [4], is drawing attention for applications in various fields of computer science. There have been several proposals for a logic programming language based on linear logic: Lolli [6], Lygon [5], LO [3], LinLog [2], Forum [8]. In addition, BinProlog allows the use of linear implications of affine logic (a variant of linear logic) [12, 13]. In these languages, since it is possible to create and consume resources dynamically as logical formulas, the efficient handling of resource formulas is an important issue.

In [10], N. Tamura and Y. Kaneda proposed an abstract machine called LLPAM which is an extension of the standard WAM [14, 1] for a linear logic programming language called LLP.[1] LLP is a superset of Prolog and a subset of Lolli.[2] The LLPAM resource management method and its extension for $\top$ goals are formally described in the paper [7] by J. Hodas et al.

The creation of resources is, in some sense, similar to the `assert` of clauses (but it is canceled by backtracking). For example, in the goal $(q \multimap p) \Rightarrow G$, creation of the resource $q \multimap p$ is just like

[1] `http://bach.seg.kobe-u.ac.jp/llp/`
[2] `http://www.cs.hmc.edu/~hodas/research/lolli/`

asserting a usual Prolog clause p :- q and it can be called during the execution of G. Therefore, in this case, resources can be compiled in the same way as the compilation of clauses.

However, in the previous LLPAM design, a resource formula was stored as a term in a heap memory and was not compiled into LLPAM code. The reason why resources were not compiled is the existence of free variables in the resources. For example, in the goal $(q(X) \multimap p(X)) \Rightarrow G$, X is a free variable which might be instantiated after the creation of the resource. Therefore, it is not possible to compile the resource as an independent clause. Resource formulas with free variables require a variable binding environment.

In this paper, we discuss an extension of LLPAM for compiling resource formulas. To compile resources with free variables, we introduce a new data structure which consists of compiled code and a variable binding environment. This is well known as a *closure* and is widely used in functional programming languages.

2 The LLP Language

The LLP language discussed in this paper is based on the following fragment of linear logic, where A is an atomic formula, $\vec{x}$ represents all free variables in the scope, and $\Rightarrow$ means intuitionistic implication (that is, $A \Rightarrow B$ is equivalent to $!A \multimap B$):

$$\begin{array}{lcl} C & ::= & !\forall \vec{x}.A \mid !\forall \vec{x}.(G \multimap A) \\ G & ::= & \mathbf{1} \mid \top \mid A \mid G_1 \otimes G_2 \mid G_1 \,\&\, G_2 \mid G_1 \oplus G_2 \mid !G \mid R \multimap G \mid R \Rightarrow G \\ R & ::= & A \mid R_1 \,\&\, R_2 \mid G \multimap R \mid G \Rightarrow R \mid \forall x.R \end{array}$$

The letters C, G and R stand for "clause", "goal" and "resource" respectively. Compared with the fragment used in the previous papers concerning LLPAM [10, 7], newly added operators are implications and universal quantifiers in resources (that is, $G \multimap R$, $G \Rightarrow R$, and $\forall x.R$).

The definition of R can be replaced with the following:

$$\begin{array}{lcl} R & ::= & R' \mid R_1 \,\&\, R_2 \\ R' & ::= & A \mid G \multimap A \mid G \Rightarrow A \mid \forall x.R' \end{array}$$

This is because the following logical equivalences hold in linear logic:

$$\begin{array}{rcll} G_1 \multimap (G_2 \multimap R) & \equiv & (G_1 \otimes G_2) \multimap R & \\ G \multimap (R_1 \,\&\, R_2) & \equiv & (G \multimap R_1) \,\&\, (G \multimap R_2) & \\ G \multimap (\forall x.R) & \equiv & \forall x.(G \multimap R) & (\text{where } x \text{ is not free in } G) \\ \forall x.(R_1 \,\&\, R_2) & \equiv & (\forall x.R_1) \,\&\, (\forall x.R_2) & \end{array}$$

We also allow $\otimes$-product of resource formulas because $(R_1 \otimes R_2) \multimap G$ is equivalent to $R_1 \multimap (R_2 \multimap G)$.

We will refer to the resource formulas defined by R' as *primitive resources*, and the formula A in a primitive resource as its *head* part. We will also refer to the resource formulas occurring in R of $R \multimap G$ and $R \Rightarrow G$ as *linear resources* and *exponential resources* respectively.

We use the following notation to write LLP programs corresponding to the above definition.

$$\begin{array}{lcl} C & ::= & A. \mid A\texttt{:-}G. \\ G & ::= & \texttt{true} \mid \texttt{top} \mid A \mid G_1\texttt{,}G_2 \mid G_1\texttt{\&}G_2 \mid G_1\texttt{;}G_2 \mid \texttt{!}G \mid R\texttt{-<>}G \mid R\texttt{=>}G \\ R & ::= & A \mid R_1\texttt{\&}R_2 \mid G\texttt{-<>}R \mid G\texttt{=>}R \mid \texttt{forall}\ x\texttt{\textbackslash}\ R \end{array}$$

The order of the operator precedence is "`forall`", "`\`", "`:-`", "`;`", "`&`", "`,`", "`-<>`", "`=>`", "`!`" from wider to narrower.

LLP covers a significant fragment of Lolli. The principal limitations are: limited-use clauses in the initial program are not allowed; universal quantifiers in goal formulas are not allowed; higher-order quantification and unification of λ-terms are not allowed.

3 Programming in LLP

In this section, we give an intuitive explanation of the resource programming features of LLP. Compared with Prolog, the biggest difference of LLP is its resource consciousness. The LLP system maintains a resource table to which resources can be dynamically added or deleted (consumed) during the execution.

3.1 Resource Addition

Resources are added by the execution of a goal formula R`-<>`G. For example, the following query adds a resource `r(1)` to the resource table and then executes a goal `r(X)` which consumes `r(1)` by letting `X` = 1.

```
?- r(1) -<> r(X).
```

In the execution of the goal R`-<>`G, all resources in R should be consumed during the execution of G. For example, the following query fails since `r(1)` is not consumed.

```
?- r(1) -<> true.
```

The resource formula G`-<>`A is used to represent a rule-type resource, In which the goal G is executed on resource consumption. So the following query displays `1`.

```
?- (write(X) -<> r(X)) -<> r(1).
```

The resource formula R_1`,`R_2 is used to add multiple resources. The following query adds resources `r(1)` and `r(2)` and then consumes both of them by letting `X` = 1 and `Y` = 2, or `X` = 2 and `Y` = 1.

```
?- (r(1), r(2)) -<> (r(X), r(Y)).
```

Note that the following does the same thing.

```
?- r(1) -<> r(2) -<> (r(X), r(Y)).
```

Resource formulas on the left-side of $\Rightarrow$ mean infinite resources; that is, they can be consumed arbitrarily many times (including zero many times). The following query succeeds by letting `X` = 1 or `X` = 2.

```
?- r(1) => r(2) => (r(X), r(X)).
```

The resource formula R_1`&`R_2 is used to represent a selective resource. For example, when `r(1)&r(2)` is added as a resource, either `r(1)` or `r(2)` can be consumed, but not both of them. The following query succeeds by letting `X` = 1 or `X` = 2.

```
?- (r(1) & r(2)) -<> r(X).
```

Executing a goal with universally quantified resource on the left-hand side of $\Rightarrow$ is similar to `asserting` a clause. For example, in the following query, the effect during the execution of `r` is as if a clause `p(X) :- q(X)` is `asserted`.

```
?- (forall X\ (q(X) -<> p(X))) => r.
```

3.2 Resource Consumption

An atomic goal formula A means resource consumption and program invocation. All possibilities are examined by backtracking. For example, the following program displays 1 and 2.

```
r(2).
?- r(1) => r(X), write(X), nl, fail.
```

In the goal formula $G_1 \& G_2$, resources are copied before execution, and the same resources should be consumed in G_1 and G_2. The following query succeeds by letting X = Y = 1 and Z = 2, or X = Y = 2 and Z = 1, because r(X) and r(Y) should consume the same resources.

```
?- (r(1), r(2)) -<> ((r(X) & r(Y)), r(Z)).
```

The goal formula $!G$ is just like G, except that only infinite resources may be consumed during the execution of G. The following query succeeds by letting X = 1 and Y = 2.

```
?- r(1) => r(2) -<> (!r(X), r(Y)).
```

The goal formula top means the consumption of some consumable resources. Of the following queries, the first one succeeds, but the second one fails because there is a remaining resource.

```
?- (r(1), r(2)) -<> (r(X), top).
?- (r(1), r(2)) -<> r(X).
```

3.3 LLP Example Programs

We show some example programs of LLP here. Other example programs of LLP are described in [11].

The following program finds a path through a directed graph. Since the arcs are added as rule-type linear resources, the conditions can be elegantly expressed.

- Each arc can be used at most once.
- A path is the transitive closure of the arc connected relation.

```
% Path Finding Program
path :-
    (a -<> b) -<>    % arc a -> b
    (b -<> c) -<>    % arc b -> c
    (c -<> a) -<>    % arc c -> a
    (c -<> d) -<>    % arc c -> d
    (d -<> b) -<>    % arc d -> b
    a -<> (d, top).  % find path from a to d
```

The following program temporarily defines a predicate test/1 by adding a resource. The predicate is used in the filter program to test whether the element satisfies the condition or not.

```
% choose(Xs, Y, Zs)
% Zs is a list of elements greater than Y in Xs.
choose(Xs, Y, Zs) :-
    (forall X \ ( X>Y -<> test(X) )) =>   % define test/1
```

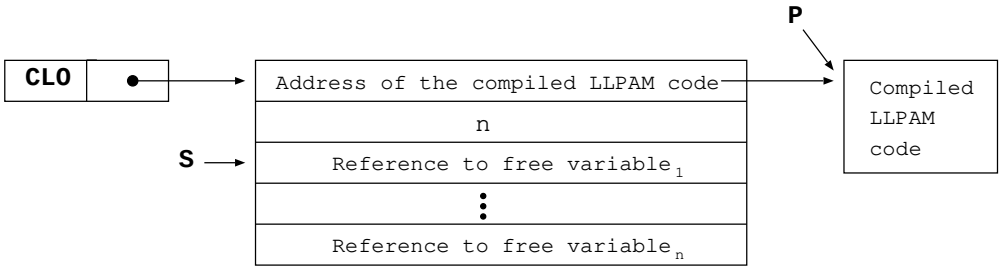

Figure 1: Closure Structure

```
    filter(Xs, Zs).                          % then call filter/2

% filter(Xs, Zs)
% Zs is a list of elements satisfying test/1 in Xs.
filter([], []).
filter([X|Xs], [X|Zs]) :- test(X), !, filter(Xs, Zs).
filter([_|Xs], Zs) :- filter(Xs, Zs).
```

Some useful applications of Lolli, such as a propositional theorem prover, a database query, and a natural language parser, are described in Hodas and Miller's paper [6]. In addition, some programs of BinProlog (including a path finding program) with using linear implication are described in BinProlog user guide [12] and the paper [13].

4 Compiling Resource Formulas

Since any resource formula can be translated into primitive resources, we will only discuss the compilation of primitive resources.

The compilation of resources without free variables is straightforward. They can be compiled just as usual clauses are because they don't require a variable binding environment. For example, the resource formula $\forall X.\forall Y.(q(X,Y) \multimap p(f(X),Y))$ can be compiled just like the clause $p(f(X),Y)$:-$q(X,Y)$.

```
get_structure f/1, A1
unify_variable A1
execute q/2
```

We now discuss compiling resources which contain free variables. For example, X is a free variable in the resource $\forall Y.(q(X,Y) \multimap p(f(X),Y))$. This resource can't be compiled like a usual clause because we should know the value of X at run-time for the consumption of the resources.

To solve this problem, we introduce a new data structure called *closure*. The closure structure consists of an address of compiled code followed by a variable binding environment, that is, a number of free variables and their references (Figure 1). Pointers to closure structures are tagged by a `CLO` tag value.

When the closure is called, the WAM register `S` is set to point to the third cell of the closure structure, and the instruction pointer `P` is set to the address of the compiled code. Therefore, the

compiled coded will begin with `unify_variable` instructions to retrieve the values of free variables. The following is an example code for the resource $\forall Y.(q(X,Y) \multimap p(f(X),Y))$:

```
unify_variable A3          % set the value of X to A3
get_structure f/1, A1
unify_value A3
put_value A3, A1
execute q/2
```

5 LLP Abstract Machine

The LLP Abstract Machine (LLPAM) is an extension of the Warren Abstract Machine (WAM) [14, 1]. The extension is mainly for efficient resource management. In this section, we will focus on the compilation of resources and only describe the differences with the previous LLPAM design [10, 7].

5.1 The Resource Table

The resource table `RES` is an array of records with the following structure.

```
record
  level: integer;          {consumption level}
  deadline: integer;       {deadline level}
  out_of_scope: Boolean;   {out_of_scope flag}
  pred: symbol;            {predicate symbol of the head part}
  code: closure;           {code of the resource}
  rellist: term            {related resources}
end;
```

`RES` grows when resources are added by $\multimap$ or $\Rightarrow$, and shrinks on backtracking. Each entry in `RES` corresponds to a single primitive resource.

The `level`, `dead_line`, and `out_of_scope` fields are for the management of resource consumption and their usage is explained in [7], so we omit the explanation in this paper. The field `rellist` is used to find the positions of $\&$-producted resources, and it is explained in [10], so we also omit the explanation here.

The `pred` and `code` fields are used to store the resource information. The field `pred` contains the predicate symbol of the resource, and the field `code` contains a pointer to the closure structure for the resource. In the previous design, a single field `head` is used to contain a term structure of the resource instead of these two fields.

5.2 Code for Resource Addition

A resource R is added by a goal $R \multimap G$ or $R \Rightarrow G$.

The following instructions are used to add primitive resources.

- put_closure L, n, Ai :
  This creates a closure structure with n free variables on HEAP, and sets Ai to point to it. The mode is set to write mode, and the unify_value instructions following this instruction are used to store the references to free variables.

```
Ai := <CLO, H>;
HEAP[H] := L;
H := H + 1;
HEAP[H] := <INT, n>;
H := H + 1;
mode := write;
```

- add_prim_res Ai, Aj :
  This adds a primitive linear resource to RES. Ai is the head part term of the resource, and Aj is a closure. The predicate symbol of Ai is stored in the pred field, and Aj is stored in the code field. The index register R pointing to the bottom of RES is incremented.

```
RES[R].pred :=  predicate symbol of Ai;
RES[R].code := Aj;
 Set level, dead_line, and out_of_scope fields;
 Record this entry in the hash table;
R := R + 1;
```

- add_prim_exp_res Ai, Aj :
  This behaves the same as add_prim_res, except the setting of the level and dead_line fields are set for primitive exponential resources.

In add_prim_res and add_prim_exp_res, the whole structure of the head part of the resource is used instead of only the predicate symbol of the head part. This is because we want the argument values of the head part to calculate the hash value of the added resource. In the current implementation, we only use the predicate symbol name (with its arity) and the first argument value for hash, but to improve the implementation in the future, we decided to pass all argument values.

The following code is for adding the primitive linear resource $q(X,Y) \multimap p(X)$ where A1 and A2 store the variable X and Y respectively.

```
put_structure p/1, A3
unify_value A1
put_closure L, 2, A4
unify_value A1
unify_value A2
add_prim_res A3, A4
```

```
L : unify_variable A3    % retreive X
    unify_variable A2    % retreive Y
    get_value A3, A1     % unify the 1st argument with X
    execute q/2
```

5.3 Code for Calling Resources

An atomic goal means resource consumption or an ordinary program invocation in LLP. In the previous LLPAM design, the execution of a `call` to an atomic goal A proceeded as follows:

(1) Extract a list of pointers to the possibly consumable resources in the resource table, `RES`, by referring to the hash table.

(2) For each `RES` entry R in the extracted list, attempt the following:

  (a) If R is out of scope, or is linear and has been consumed, fail.

  (b) Attempt to unify A with the `head` field of R.

  (c) Mark the entry R as consumed.

(3) After the failure of all trials, `call` the ordinary code for predicate A.

The above procedure was executed by a special built-in predicate, and it was invoked from the `call` instruction as described in [10]. This approach was taken only to make the implementation easy, but it makes the execution of LLPAM difficult to understand.

In this section, we describe a new code generation method for atomic goals. The outline of the execution of an atomic goal A with predicate symbol p/n is as follows:

(1) Extract the list of indices of the possibly consumable primitive resources in the resource table, `RES`, by referring to the hash table. In the current implementation, predicate symbol p/n and the first argument of A are used for the hash key. The two registers `R1` and `R2` are used to store the extracted lists of indices.

(2) For each `RES` entry R with predicate symbol p/n in the extracted lists `R1` and `R2`, attempt the following:

  (a) If R is out of scope, or is linear and has been consumed, fail.

  (b) Mark the entry R as consumed.

  (c) Execute the compiled code closure of R.

(3) After the failure of all trials, `call` the ordinary code for predicate A.

Step (1) is done by the `call` (or `execute`) instruction. The register `R1` is set to include the indices of the resources which have the same predicate symbol and the same first argument in its head part. The register `R2` is set to include the indices of the resources which have the same predicate symbol, but the first argument was an unbound variable when the resource was added. We need the resources in `R2` because their heads are also possibly unifiable with the goal A.

The following new instructions are used for steps (2) and (3).

- `try_resource` L :
  This behaves the same as "`try` L", but `R1` and `R2` are also saved in the created choice point frame.

- `restore_resource` :
  This restores the register values from the current choice point frame (indicated by the current value of `B` register).

- `retry_resource_else` L :
  This replaces the `R1` and `R2` values in the current choice point frame by their current values. The next clause field `BP` in the choice point frame is also replaced by L.

- `trust_resource` L :
  This sets the current value of the `B` register to its predecessor and then jumps to an address L.

- `pickup_resource` p/n, `Ai`, L :
  This finds the index value of the consumable resource with predicate symbol p/n from `R1` (or `R2` if `R1` is `nil`) and sets that index value to `Ai`. `R1` and `R2` are updated to have the remaining resources. If there are no consumable resources, it jumps to L.

```
found := false;
while R1 ≠ nil and not found do begin
    i := car(R1);
    R1 := cdr(R1);
    if RES[i].pred = p/n then
        found := true
end;
while R2 ≠ nil and not found do begin
    i := car(R2);
    R2 := cdr(R2);
    if RES[i].pred = p/n then
        found := true
end;
if not found then
    P := L;
```

- `consume Ai, Aj` :
  This marks the resource pointed by the index value `Ai` as consumed and sets `code` field value to `Aj`.

```
 Mark RES[Ai]  as consumed;
Aj := RES[Ai].code;
```

- `execute_closure Ai` :
  This executes the closure `Ai` (suppose it is $\langle$`CLO`$, c\rangle$) and then sets register `S` to $c + 2$ and set the mode to `read`. Then, it jumps to the address pointed to by `HEAP[`c`]`.

```
<CLO, c> := Ai;
S := c + 2;
mode := read;
P := HEAP[c];
```

```
p/2   :  pickup_resource p/2, A3, L'      % added
         try_resource L1
L0    :  restore_resource
         pickup_resource p/2, A3, L2
         retry_resource_else L0
L1    :  consume A3, A4
         execute_closure A4
L2    :  trust_resource L'
```

Figure 2: Code for $p/2$

For example, the following is the simplest code for an atomic goal with $p/2$ (where L' is the address of ordinary program code):

```
p/2   :  try_resource L1
L0    :  restore_resource
L1    :  pickup_resource p/2, A3, L2
         retry_resource_else L0
         consume A3, A4
         execute_closure A4
L2    :  trust_resource L'
```

The above code contains an obvious inefficiency. Even if there is no consumable resource, it allocates a new choice point frame by the `try_resource` instruction. The improved code (Figure 2) will begin with the `pickup_resource` instruction to check whether there are any consumable resources or not. If there are no consumable resources, it executes the ordinary program code immediately.

We now discuss the optimization of the code for atomic goals. For every execution of an atomic goal, the resource consumption must be examined, regardless of whether there exists an ordinary program invocation or not.

On the other hand, when there is no ordinary program invocation, an atomic goal means only resource consumption. Our design of optimization is limited to this case, and the essence of it is as follows:

- If there is only one consumable resource, all we have to do is consume it immediately.
- It is safe to discard the current choice point frame before consuming the last consumable resource.

The following new instruction is used to generate the optimized code for atomic goals.

- `if_no_resource` L :
  This scans whether there are consumable resources in `R1` and `R2`. If there are no consumable resources, jump to L.

```
p/2  :  pickup_resource p/2, A3, fail
        if_no_resource L1                 %  added
        try_resource L1
L0   :  restore_resource
        pickup_resource p/2, A3, L2
        if_no_resource L3                 %  added
        retry_resource_else L0
L1   :  consume A3, A4
        execute_closure A4
L2   :  trust_resource fail
L3   :  trust_resource L1                 %  added
```

Figure 3: Optimized code for $p/2$ when there are no programs

The above idea can be achieved quite easily by inserting the `if_no_resource` instruction just after the `pickup_resource` instructions. Figure 3 shows an optimized code corresponding to the Figure 2.

6 Performance Evaluation

In this section, we would like to show an improvement in performance by compiling resources instead of representing resources as heap terms. Currently we are developing the LLP compiler system based on the LLPAM presented in this paper, and a prototype system has been developed.

We use a simple puzzle "tiling board with Dominoes" (all solution) as a benchmark. Dominoes are puzzle pieces. Each piece consists of two equal squares. There are exactly two possible shapes. The goal of the puzzle is to place the dominoes so that they fit into the board of given dimension.

A LLP program (Figure 4) is used for the comparison. In the program, let the board size be (m, n), the following conditions can be expressed elegantly:

- All $m \times n$ units of the board can be used exactly once.
  This can be represented easily by mapping each unit to a resource `b(_,_,_)`.
- All $\frac{m \times n}{2}$ dominoes can be used and placed anywhere on the board.
  This condition can be expressed by mapping each domino to a &-product of rule-type resources which have `domino(_)` as their head parts. Placing a domino at (i, j) is done by consuming `b(`i`,`j`,_)` and `b(`i`,`$j+1$`,_)` (or `b(`i`,`j`,_)` and `b(`$i+1$`,`j`,_)`) in body parts.

Attack check is done automatically by consuming `domino(_)`.

The LLPAM code for Domino puzzle program is about 28% faster than code of previous version of the LLPAM. This means that compiling resources is about 28% faster than representing resources as terms in a heap memory. This speedup is due to the compilation of rule-type resources which have compound goal formulas as their body parts. Sadly, the speedup for other benchmarks which don't contain compound resources is smaller than that for Domino puzzle. Table 1 shows CPU times on our prototype system of Domino puzzle program finding all solutions for various board sizes. All times were collected on a SUN SPARCstation-20 running SunOS 4.1.4.

Board size (row, column)	# Runs Averaged	Previous version of the LLPAM	The LLPAM described here	% Δ
(2, 5)	10	533 ms	370 ms	30%
(2, 6)	10	5,863 ms	4,216 ms	28%
(3, 4)	10	6,016 ms	4,283 ms	29%
(2, 7)	10	71,750 ms	53,800 ms	25%
(4, 4)	5	1,247,700 ms	905,354 ms	27%

Table 1: Execution time of Domino puzzle (all solution)

The LLP compiler system [11] based on the previous LLPAM design has been developed since 1996. It consists of the LLP to LLPAM compiler (written in Prolog) and the LLPAM emulator (written in C). Prolog programs can be executed without any modification. The speed of the compiled code for Prolog programs is about 2 or 3 times slower than SICStus Prolog 2.1 WAM code. The latest package (version 0.43) including all source code can be obtained from the following site:

`http://bach.seg.kobe-u.ac.jp/llp/`

7 Conclusion and Future Work

In this paper we have described an extension of LLPAM for compiling resource formulas. To compile resources with free variables, we introduce a new data structure which consists of some compiled code and a set of bindings for free variables. Such a structure corresponds to the idea of a *closure* that is widely used in implementations of functional programming languages.

With regard to performance, the LLPAM code for a benchmark is about 28% faster than code of previous version of the LLPAM. This means that compiling resources is about 28% faster than representing resources as terms in a heap memory.

We note the similarity between our scheme and that used for the higher-order logic programming language λProlog. In [9], G. Nadathur et al. propose implementation method for allowing $D \supset G$ as a goal in λProlog. The implication goal $D \supset G$ corresponds to the goal $D \Rightarrow G$ of LLP. Indeed, the idea of closure is similar to that discussed in Section 5 (in [9]). Our scheme differs in details from that in [9] because of the existence of linear implication.

The following points are still remaining:

- Compiling resource formulas dynamically at execution time.
  Our scheme doesn't cover use of metavariables as resources. Dynamic resource compilation will give an answer to this problem.

- Allowing $\forall x.G$ as a goal formula.
  The paper [9] describe implementation method for allowing $\forall x.G$ as a goal in λProlog.

Currently, we are improving existing LLP compiler system to incorporate resource compilation mechanism presented in this paper.

```
%%%  Solver
solve_domino(M, N) :-
        D is M*N/2,
        row(M) =>
        column(N) =>
        num_of_dominoes(D) =>
        %((place_domino(D) & write_board(1,1)) -<> cont)
        (place_domino(D) -<> cont)
        -<>
        gen_res(M, N).

place_domino(0).
place_domino(N) :-
        N > 0,
        domino(N),
        N1 is N-1,
        place_domino(N1).

%%%  Create Resources
gen_res(0) :- cont.
gen_res(N) :-
        N > 0,
        ((b(I, J, N), J1 is J+1, b(I, J1, N)) -<> domino(N)
        &
        (b(I, J, N), I1 is I+1, b(I1, J, N) ) -<> domino(N))
        -<>
        (N1 is N-1, gen_res(N1)).

gen_res(I, J) :-
        I < 1,
        !,
        num_of_dominoes(D),
        gen_res(D).
gen_res(I, J) :-
        J < 1,
        !,
        I1 is I-1,
        column(N),
        gen_res(I1, N).
gen_res(I, J) :-
        J1 is J-1,
        b(I, J, _) -<> gen_res(I, J1).
```

Figure 4: Domino puzzle program used for the comparison

Acknowledgements

I would like to thank referees for giving us useful comments and suggestions. I'm grateful to Joel David Hamkins for his useful advice.

References

[1] H. Aït-Kaci. *Warren's Abstract Machine*. The MIT Press, 1991.

[2] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[3] J.-M. Andreoli and R. Pareschi. Linear objects: Logical processes with built-in inheritance. *New Generation Computing*, 9:445–473, 1991.

[4] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[5] J. Harland and D. Pym. The uniform proof-theoretic foundation of linear logic programming. In V. Saraswat and K. Ueda, editors, *Proceedings of the International Logic Programming Symposium*, pages 304–318, San Diego, California, Oct. 1991.

[6] J. S. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994. Extended abstract in the Proceedings of the Sixth Annual Symposium on Logic in Computer Science, Amsterdam, July 15–18, 1991.

[7] J. S. Hodas, K. Watkins, N. Tamura, and K.-S. Kang. Efficient implementation of a linear logic programming language. In *Proceedings of the 1998 Joint International Conference and Symposium on Logic Programming*, June 1998. (to appear).

[8] D. Miller. A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[9] G. Nadathur, B. Jayaraman, and K. Kwon. Scoping constructs in logic programming: Implementation problems and their solution. *Journal of Logic Programming*, 25(2):119–161, Nov. 1995.

[10] N. Tamura and Y. Kaneda. Extension of WAM for a linear logic programming language. In T. Ida, A. Ohori, and M. Takeichi, editors, *Second Fuji International Workshop on Functional and Logic Programming*, pages 33–50. World Scientific, Nov. 1996.

[11] N. Tamura and Y. Kaneda. A compiler system of a linear logic programming language. In *Proceedings of the IASTED International Conference Artificial Intelligence and Soft Computing*, pages 180–183, July 1997.

[12] P. Tarau. BinProlog 5.40 User Guide. Technical Report 97-1, Département d'Informatique, Université de Moncton, Apr. 1997. Available from `http://clement.info.umoncton.ca/BinProlog`.

[13] P. Tarau, V. Dahl, and A. Fall. Backtrackable State with Linear Assumptions, Continuations and Hidden Accumulator Grammars. In *ILPS'95 Workshop on Visions for the Future of Logic Programming*, Nov. 1995.

[14] D. H. D. Warren. *An abstract Prolog instruction set.* Technical Note 309, SRI International, October 1983.

Tracing Prolog Programs by Source Instrumentation is Efficient Enough

Mireille Ducassé

IRISA/INSA, Campus Universitaire de Beaulieu
F - 35042 Rennes Cedex, France, email: ducasse@irisa.fr

Jacques Noyé

École des Mines de Nantes, 4, rue Alfred Kastler, La Chantrerie
F- 44307 Nantes Cedex 03, France, email: noye@emn.fr

Abstract

Tracing by automatic program source instrumentation has major advantages over compiled code instrumentation: it is more portable, it benefits from many compiler optimizations, it produces traces in terms of the original program, and it can be tailored to specific debugging needs.

The usual argument in favor of compiled code instrumentation is its supposed efficiency. We have compared the performance of two operational low-level Prolog tracers with source instrumentation. We have executed classical Prolog benchmark programs, collecting trace information without displaying it. On average, collecting trace information by program instrumentation is about as fast as using a low-level tracer in one case, and only twice slower in the other. This is a minor penalty to pay, compared to the advantages of the approach.

To our knowledge, this is the first time that a quantitative comparison of both approaches is made for any programming language.

1 Introduction

There are four basic ways to trace executions [4]: manual program source instrumentation, automatic program source instrumentation, instrumentation of meta-interpreters, and compiled code instrumentation. Manual program source instrumentation is tedious and error-prone, and instrumentation of meta-interpreters is notoriously too inefficient. No comparison of automatic program source instrumentation and compiled code instrumentation exists.

Source instrumentation has major advantages over compiled code instrumentation. Firstly, it is more portable than compiled code instrumentation as it is independent from the low-level implementation details of any particular compiler. Secondly, the instrumented programs are compiled and therefore they can benefit from many of the available compiler optimizations.

Thirdly, the instrumented program is traced at the level of the original program. Specht [11] showed that this feature is particularly adapted to tracing deductive databases programs as they are heavily transformed during compilation. This is also the case for plain Prolog. Indeed, many optimizations are made at compile time and the resulting code is quite different from the original one. A low-level tracer cannot always reverse the optimizations to give the information in terms of the original program.

Fourthly, source instrumentation can be tailored to specific debugging needs, and therefore be more efficient. Indeed, in general not the whole trace information is needed for a given application or a given debugging request. A good example is an abstract tracer of a programming language whose compiler is implemented in Prolog. Producing the Prolog trace and then filtering it to get the abstract trace is not the most efficient way of processing. The instrumentation can be adapted to directly trace at the abstract level.

While source instrumentation has advantages over compiled code instrumentation, the usual argument in favor of compiled code instrumentation is its efficiency.

Beside Specht, Calejo [2] uses source instrumentation by program transformation to trace logic programs. No performance measurements are available for any of the systems.

Tolmach and Appel [14] designed and implemented a tracer for Standard ML based on automatic program source instrumentation. The resulting code runs only 3 times slower than optimized code. They conjectured that a low-level tracer would run at about the same speed. However they had no reasonable low-level tracer at their disposal with which they could compare their results.

We have performed such a comparison in the context of Prolog, using two quite different systems: ECLiPSe [8, 5], and SICStus [12]. As far as source instrumentation is concerned, it is based on an extension of O'Keefe's `advice` utility [1]. The instrumentation takes built-in predicates into account and provides the user with a more complete trace model.

We have executed classical Prolog benchmark programs [10] collecting trace information without displaying it. On average, collecting trace information by program instrumentation is only about twice slower than using a low-level tracer when using ECLiPSe and has similar performance when using SICStus. Even a slowdown of two is a minor penalty to pay, compared to the advantages of the approach. Actually, such a slowdown is, in many cases, negligible, either because only part of the execution is traced, or because the execution is dominated by input/output (be it as part of the application or as part of displaying trace information).

To our knowledge, this is the first time that a quantitative comparison of both approaches is made for any programming language.

In the following we first introduce the trace model used by both tracers. We then informally present the instrumentation by program transformation. The transformation can deal with predicates which are *traced* or *skipped*. The difference is that the execution of the subgoals of skipped predicates is, unlike the execution of the subgoals of traced predicates, not traced. Some counters which help to analyze the trace are then introduced. Finally, we present our experiments. The methodology is described followed by some of the results and a discussion.

2 Modeling Prolog executions

A Prolog trace is a sequence of events which gives a picture of a program execution. We use an execution model close to the classical box model of Byrd [1] in which execution events are bound to goals. There are different types of events, called *ports*. A *call* event tells that g is invoked and gives the instantiation of its arguments at the moment of the invocation. A $fail$ event tells that g fails. An $exit$ event tells that g succeeds and gives the resulting instantiation of the arguments. A *redo* event tells that the execution is backtracking either to g or to one of its subgoals. We have added the $unify$ event [6] which tells when the execution finds a clause that unifies with g and gives the resulting instantiation of the arguments, it also gives the unified clause.

[1] The `advice` utility is part of the DEC10 Prolog library, available by anonymous ftp from the AIAI of the University of Edinburg (aiai.edinburgh.ac.uk).

mem(X, List, Y, Z) ←
Y and Z are consecutive elements of List such that their sum is equal to X.

```
mem(X,[Y,Z|Zs],Y,Z) :-
   X is Y + Z.
mem(X,[_|Zs],Y,Z) :-
   mem(X,Zs,Y,Z).
```

The predicate `mem/4` *is transformed to produce trace information.*

```
mem(X,Xs,Y,Z) :-
   ( trace(call,mem(X,Xs,Y,Z))
   ; trace(fail,mem(X,Xs,Y,Z)),
     fail),
   mem$proc(X,Xs,Y,Z),
   ( trace(exit,mem(X,Xs,Y,Z))
   ; trace(redo,mem(X,Xs,Y,Z)),
     fail).

mem$proc(X,[Y,Z|Zs],Y,Z) :-
   trace(unify,mem(X,[Y,Z|Zs],Y,Z),1),
   X is Y + Z.
mem$proc(X,[_|Zs],Y,Z) :-
   trace(unify,mem(X,[_|Zs],Y,Z),2),
   mem(X,Zs,Y,Z).
```

Figure 1: A Prolog program and an instrumented version of it.

In an exhaustive trace, the events related to a given goal are not necessarily consecutive. They are intertwined with events related to subgoals and siblings. A more detailed description of the trace model can be found in [3].

3 Informal presentation of the instrumentation

The first subsection presents a transformation which is basically the one of the `advice` package of the DEC10 Prolog library, extended in order to deal with the *unify* port. The next subsection extends the transformation to deal with *skipped* predicates. Specific transformations for the cut and meta-calls are presented. Useful counters are then introduced.

3.1 Instrumentation of a single predicate

Figure 1 shows a Prolog program and a transformed version of it, where the predicate `mem/4` is transformed to produce trace information. The body of `mem/4` is replaced by trace instructions and a new predicate `mem$proc/4` is created. This predicate calls the body of the original predicate `mem/4`.

The predicates `trace/2/3` can, for example, display their arguments on the standard output or they can send them to a trace analyser such as Opium [3]. The three important properties of the predicates `trace/2/3` are that they (1) succeed exactly once, (2) bind no variable of the original program and, (3) have no side effect on the original program.

With this scheme, the displayed values of the variables depend on the stage of the execution which is traced. The `call` trace gives the value of variables at invocation time; the `unify` trace gives the value of variables resulting from a unification; the `exit` trace gives the value of variables resulting from the execution of the goal; the `redo` trace gives the same values as the previous `exit` trace related to the same goal; the `fail` trace gives the same values as the `call` trace related to the same goal. Indeed, the bindings that could have been made during the execution are undone.

Except for the trace information sent to the standard output, the behavior of the transformed program is the same as the initial one; goals are evaluated in the same order and the same substi-

```
1      :- mem(5, [2,3,2,4],Y,Z).
2   call    mem(5,[2,3,2,4],Y,Z)
3   unify(1) mem(5,[2,3,2,4],2,3)
4   exit    mem(5,[2,3,2,4],2,3)
5      Y = 2
6      Z = 3     More? (;)
7   redo    mem(5,[2,3,2,4],2,3)
8   unify(2) mem(5,[_,3,2,4],Y,Z)
9   call    mem(5,[3,2,4],Y,Z)
10  unify(1) mem(5,[3,2,4],3,2)
11  exit    mem(5,[3,2,4],3,2)
12  exit    mem(5,[2,3,2,4],3,2)
13     Y = 3
14     Z = 2     More? (;)
15  redo    mem(5,[2,3,2,4],3,2)
16  redo    mem(5,[3,2,4],3,2)
17  unify(2) mem(5,[_,2,4],Y,Z)
18  call    mem(5,[2,4],Y,Z)
19  unify(1) mem(5,[2,4],2,4)
20  unify(2) mem(5,[_,4],Y,Z)
21  call    mem(5,[4],Y,Z)
22  unify(2) mem(5,[_],Y,Z)
23  call    mem(5,[],Y,Z)
24  fail    mem(5,[],Y,Z)
25  fail    mem(5,[4],Y,Z)
26  fail    mem(5,[2,4],Y,Z)
27  fail    mem(5,[3,2,4],Y,Z)
28  fail    mem(5,[2,3,2,4],Y,Z)
29     no (more) solution.
```

Figure 2: Trace obtained by the resolution of goal `mem(5,[2,3,2,4],Y,Z)` with the instrumented program of Figure 1

tutions are performed in the same order. The operational semantics of Prolog is preserved.

The resolution by Prolog of the goal `mem(5,[2,3,2,4],Y,Z)` of the instrumented program generates the trace of Figure 2. Line numbers have here been added by hand.

3.2 *Skipped* predicates

Figure 2 shows a trace related to one predicate, `mem/4`. In order to see information related to other predicates one can transform them, if their definition is available. This is unfortunately not the case for built-in predicates. Indeed, their source code is usually not available and, moreover, they are often not even implemented in Prolog.

Yet, a detailed trace of built-in predicates is not desirable. They are normally not under scrutiny while debugging user programs. What is interesting is to trace the fact that they are called and which result they return. The same applies to user-defined predicates which have been tuned and are assumed correct. We call these predicates *skipped*. Built-in predicates are handled in the same way as skipped predicates.

In the following, the predicates whose definitions are transformed are called *traced* to distinguish them from skipped (and built-in) predicates.

We therefore introduce a special transformation for skipped predicates. The principle is to use the same wrapper as for traced predicates, but instead of wrapping the definition, the transformation wraps the invocation.

Figure 3 shows the modifications of the instrumentation for the skipped predicate `is/2`, and the resulting differences on the trace. The invocation of `X is Y+Z` is replaced by the invocation of the wrapped version namely `is$trace(X, Y+Z)`[2]. The wrapped invocation is constructed only once.

Figure 3 also shows some resulting modifications of the trace. For example, one can see that the first clause of `mem/4` does not produce any solution because `5 is 4+2` fails.

[2]In this case the redo is useless, as `is/2` is known not to be resatisfiable.

A clause calling a skipped predicate

```
mem$proc(X,[Y,Z|Zs],Y,Z) :-
   trace(unify,mem(X,[Y,Z|Zs],
                   Y,Z),1),
   X is Y + Z.
```

is transformed into

```
mem$proc(X, [Y, Z|Zs],Y,Z) :-
   trace(unify, mem(X,[Y,Z|Zs],
                    Y,Z),1),
   is$trace(X,Y + Z).
```

```
is$trace(X, Expr) :-
   ( trace(call,is(X, Expr))
   ; trace(fail,is(X, Expr)),
     fail),
   is(X, Expr),
   trace(exit,is(X, Expr)).
```

The portion of trace

```
19  unify(1) mem(5,[2,4],2,4)
20  unify(2) mem(5,[_,4],Y,Z)
```

becomes

```
19  unify(1) mem(5,[2,4],2,4)
    call    5 is 4+2
    fail    5 is 4+2
20  unify(2) mem(5,[_,4],Y,Z)
```

Figure 3: Modifications of the instrumentation for the skipped predicate `is/2`, and the resulting differences on the trace.

3.3 Cut, conditionals and meta-calls

The wrapping of an invocation of the built-in predicate `!/0` cannot be folded into a predicate `!$trace`, otherwise the scope of the cut operation would be incorrect. The wrapping is therefore done in place as illustrated by Figure 4. Note that, in the transformed program, the cut operation correctly cuts the choice points attached to the goals `q$proc` and `s`. It does not cut the choice points attached to the wrapping of `q`, nor the ones attached to the goal `t`. When the execution backtracks to `t`, the redo information of line 12 correctly informs that there is some backtracking inside the box of `q`. The same in-place wrapping is done for conditional constructs.

The goals which are arguments of meta-call predicates have also to be wrapped, especially if they are skipped. Moreover, most of the time their value is not known at transformation time, in that case the wrapping is postponed until execution time. This is illustrated by Figure 5. The predicate `wrap_goal/2` is called at execution time. Note that in the general case, with the counters introduced in the following, the predicate `wrap_goal/2` requires additional arguments. It is therefore slightly more complicated and costly.

3.4 The *call number* and *depth* counters

Considering lines 18-26 of Figure 2, one can see that they only deal with predicate `mem/4`. However, all the lines do not relate to the same goal. Indeed, line 18 relates to goal `mem(5,[2,4],Y,Z)` while line 23 relates to `mem(5,[],Y,Z)`. As it is necessary to be able to easily distinguish between different goals, some information about the invocation number must be introduced in a trace line/event, namely the *call number*. The *depth* of execution is also very useful when reconstructing search and proof trees. We therefore add another counter which represents the size of the invocation stack.

Figure 6 shows the transformation of `mem/4` in order to get the *call number* and *depth* counters. The predicates `mem$proc` and `is$trace` are also modified. A new predicate, `mem$trace/5`, is created. It initializes the `depth` to `1` while keeping the same external interface for predicate

Initial program

```
q :- s, !, t.
q .

s.

t.
t.
```

q is basically transformed into

```
q :-
    trace_call_fail(q),
    q$proc,
    trace_exit_redo(q).

q$proc :-
    trace_unify(q, 1),
    s,
    trace_call(!),
    !,
    trace_exit(!),
    t.
q$proc :-
    trace_unify(q, 2).
```

The corresponding trace

```
      :- q, fail.
1   call   q
2   unify  q
3     call   s
4     unify  s
5     exit   s
6     call   !
7     exit   !
8     call   t
9     unify  t
10    exit   t
11  exit   q
12  redo   q
13    redo   t
14    unify  t
15    exit   t
16  exit   q
17  redo   q
18    redo   t
19    fail   t
20  fail   q

no (more) solution.
```

Figure 4: Transformation of a predicate with a cut, and its corresponding trace.

Initial program

```
q(X) :- call(X).
```

q is basically transformed into

```
q(X) :-
    trace_call_fail(q(X)),
    q$proc(X),
    trace_exit_redo(q(X)).

q$proc(X) :-
    trace_unify(q(X), 1),
    wrap_goal(X, WrappedX),
    trace_call_fail(call(X)),
    call(WrappedX),
    trace_exit_redo(call(X)).
```

```
wrap_goal(Goal, WrappedGoal) :-
    (  traceable(Goal)
    -> WrappedGoal = Goal
    ;  WrappedGoal =
         (trace_call_fail(Goal),
          Goal,
          trace_exit_redo(Goal))
    ).
```

The corresponding trace

```
:-  q(true).

1  call  q(true)
2  unify  q(true)
3    call  call(true)
4      call  true
5      exit  true
6    exit  call(true)
7  exit  q(true)
```

Figure 5: Transformation of a predicate with a meta-call, and its corresponding trace.

`mem/4`. The predicate `incr_call/1` dynamically increments the global counter which handles the *call number*. *Depth* is incremented into `NewDepth`, which is passed to the subgoals by `mem$proc/7`. We introduce the `trace_call_fail/3` and `trace_exit_redo/3` predicates to ease the presentation.

Figure 6 also shows a portion of trace taking skipped predicates and counters into account. The first number in a line is the invocation number (*call number*), the second one, in brackets, is the *depth*. Note that the indentation has been calculated according to the *depth* information, starting with an indentation of 0 for depth 3.

4 Experimental Results

4.1 Methodology

In order to get a feel for the relative performance of program source instrumentation versus compiled code instrumentation, we have implemented, using ECLiPSe, the program transformations described above. We have also implemented, both in ECLiPSe and SICStus, the kernel of an interactive tracer, hereafter referred to as Poppy.

A trace can therefore be generated in two ways, through compiled code instrumentation or through source code instrumentation.

In order to use compiled code instrumentation, the program is compiled and then executed in debug mode (the interpretation of the instructions by the virtual machine produces trace information). Note that, when using ECLiPSe, the program must be compiled in a special debug compilation mode (specific ECLiPSe virtual machine instructions are produced). This makes it possible to mix, in debug mode, fast optimized code, and slower debuggable code. As far as SICS-

Transformation of mem/4*, a new predicate* mem$trace/5 *is created*

```
mem(X,Xs,Y,Z) :-
   mem$trace(1,X,Xs,Y,Z).

mem$trace(Depth,X,Xs,Y,Z) :-
   incr_call(CallNb),
   trace_call_fail(mem(X, Xs, Y, Z),
                   Depth, CallNb),
   NewDepth is Depth + 1,
   mem$proc(Depth, CallNb, NewDepth,
            X,Xs,Y,Z),
   trace_exit_redo(mem(X, Xs, Y, Z),
                       Depth,CallNb).

mem$proc(Depth, CallNb, NewDepth,
         X, [Y,Z|Zs], Y, Z) :-
   trace(unify,
         mem(X, [Y,Z|Zs], Y, Z),
         Depth, CallNb, 1),
   is$trace(NewDepth,X,Y+Z).
mem$proc(Depth,CallNb, NewDepth,
          X, [_ |Zs], Y, Z) :-
   trace(unify,
         mem(X, [_ |Zs], Y, Z),
         Depth, CallNb, 2),
   mem$trace(NewDepth,X,Zs,Y,Z).
```

```
trace_call_fail(G,Depth,CallNb) :-
   trace(call,G,Depth,CallNb).
trace_call_fail(G,Depth,CallNb) :-
   trace(fail,G,Depth,CallNb),
   fail.

trace_exit_redo(G,Depth,CallNb) :-
   trace(exit,G,Depth,CallNb).
trace_exit_redo(G,Depth,CallNb) :-
   trace(redo,G,Depth,CallNb),
   fail.
```

The portion of trace equivalent to the lines 18-26 of Figure 2 with skipped predicates and counters is now

```
5[3] call    mem(5,[2,4],Y,Z)
5[3] unify(1) mem(5,[2,4],2,4)
6[4]   call 5 is 2 + 4
6[4]   fail 5 is 2 + 4
5[3] unify(2) mem(5,[_,4],Y,Z)
7[4]   call    mem(5,[4],Y,Z)
7[4]   unify(2) mem(5,[_],Y,Z)
8[5]      call    mem(5,[],Y,Z)
8[5]      fail    mem(5,[],Y,Z)
7[4]   fail    mem(5,[4],Y,Z)
5[3] fail    mem(5,[2,4],Y,Z)
```

Figure 6: The transformation of mem/4 in order to get the *call number* and *Depth* counters. A portion of the trace with counters and skipped predicates.

tus is concerned, we have used SICStus version 3.7 beta [12]. This version makes it possible to trace compiled code (previous versions were relying on interpreted code for debugging) but debugging support is hidden to the compiler.

In order to use source code instrumentation, the program is first instrumented through program transformation. The instrumented program is then compiled and executed in optimized (non debug) mode. The trace is produced through calls to Poppy, *i.e.*, basically, calls to the predicates trace_call/3, trace_call_fail/3, trace_exit/3, trace_exit_redo/3, and incr_call/1. The predicate trace_call/3 is a specialization of trace_call_fail/3, used for built-in predicates which are known not to fail (*e.g.* write/1) and trace_exit/3 a specialization of trace_exit_redo/3 used for built-in predicates which are known not to be resatisfiable (*e.g.* is/2).

In order to capture the cost associated to managing several tracing modes, as well as spied predicates, a subset of the usual tracing facilities has been implemented: step by step mode, *leap* mode (no trace is displayed until a spied predicate is encountered), *skip* mode (no trace is displayed until a trace line referring to the same call is encountered), and trace disabling.

The trace generated by ECLiPSe, SICStus, and Poppy have been made as close as possible. The

leashing modes[3] of ECLiPSe have been set so that *next* events, specific to ECLiPSe, are not traced but so that *unify* events, not traced by default, are traced. As both ECLiPSe and SICStus do not explicitly trace disjunctions and conditional constructs, producing specific tracing predicates for these constructs has been disabled from our transformations. Tracing usual sequences consisting of a type checking predicate followed by a cut, used for indexing purposes, has also been disabled. Indeed, such sequences are not shown by ECLiPSe nor by SICStus (tracing them would mean deteriorating indexing).

However, discrepancies between both traces remain. Sometimes ECLiPSe and SICStus do less work. SICStus does not deal with *unify* ports and ECLiPSe does not show the results of successful unification of a fact. Neither ECLiPSe nor SICStus show pseudo backtracking (as *redo* events) into deterministic subtrees of the execution tree. Sometimes, they do more. Having direct access to the run-time data structures, both ECLiPSe and SICStus have the possibility of displaying information such as the determinacy of an exit. Moreover, when tracing a cut, all the goals made deterministic are shown by ECLiPSe.

All our measures have been made in leap mode, without any spy point set. This means that the trace information is produced but not displayed. Indeed the time spent displaying the trace would dominate the time spent in the tracing machinery.

Our comparisons are based on the Aquarius benchmark suite. This is a set of benchmark programs put together, from a number of different sources, as part of the Aquarius project (University of California, at Berkeley). This suite is well known in the Prolog community; it has, among others, been used to assess the performance of the BAM processor [7], the KCM machine [9], the Aquarius Prolog Compiler [10], and Parma [13], a Prolog compiler for the MIPS RISC architecture. It is also interesting in that it includes programs of different sizes and programming styles and does not discard the use of important built-in predicates such as `arg/3`, `functor/3`, or `write/1`, which often represent an important share of the execution time. Note that new versions of three programs, **flatten**, **reducer** and **sdda**, without calls to `write/1`, have been added to the initial benchmarks to get a better feel for the weight of this predicate. They are distinguished by a `_nw` suffix.

The programs were executed on a lightly loaded SUN Ultra-1/SunOS 5.5.1 workstation, using ECLiPSe version 3.5.2 and SICStus 3.7 beta, with garbage collection on. When using SICStus, both the `compactcode` and the `fastcode` compilation mode have been measured. The first mode produces byte-coded abstract instructions whereas the second one produces native machine instructions.

The execution times (user cpu time including garbage collection time) have been obtained by incrementally compiling and running each program one after another in a repeat/fail loop such that each program runs about 20 seconds. This guarantees that the timing intervals are well above the clock accuracy and reduces cache effects.

4.2 Results

The main results of our experiments are shown in figure 7. This figure gives for each compilation environment, ECliPSe, SICStus in compact compilation mode, and SICStus in fast compilation mode, the ratios between the execution times for producing a program trace with the native tracer and with Poppy, using the methodology previously described. A ratio greater than one means that Poppy is slower than the corresponding native tracer.

The corresponding average ratios are 1.18 (SICStus fast), 1.26 (SICStus compact), and 2.18

[3]The *leashing mode* of a port tells whether this port should be included in the trace and lead to ask the user for a new command.

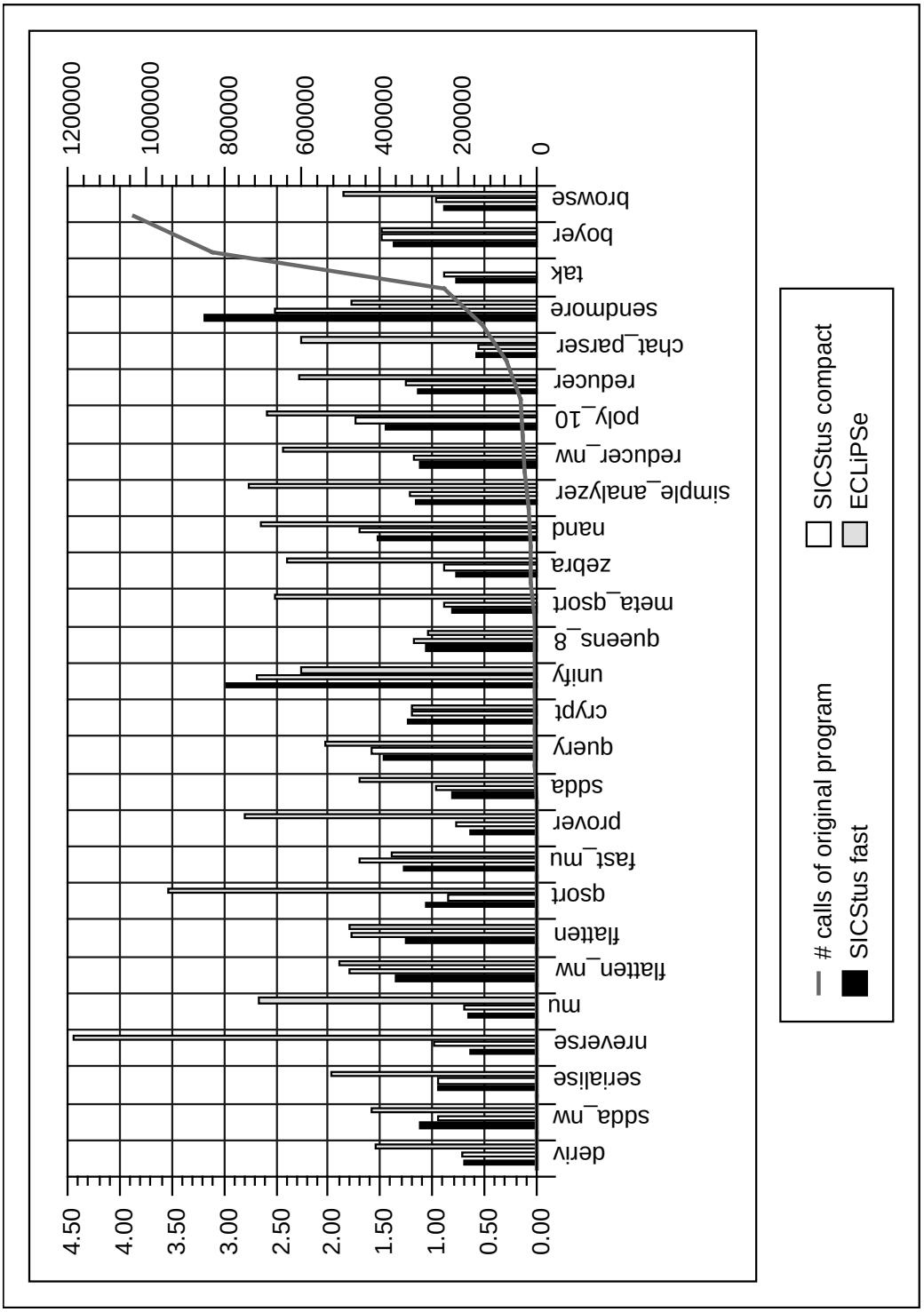

Figure 7: Source instrumentation vs code instrumentation ratios.

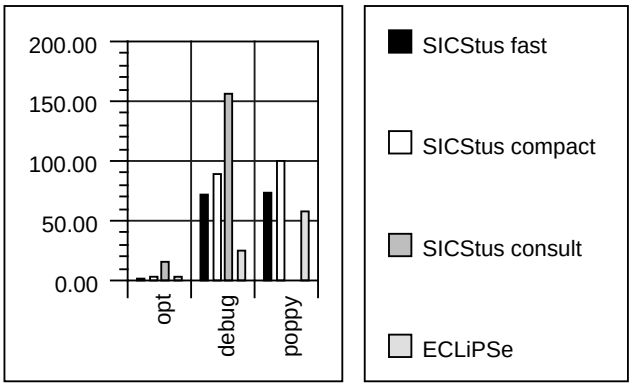

Figure 8: Relative speed depending on compilation mode and environment.

(ECLiPSe) with standard deviations of 0.62, 0.53, and 0.73, respectively. These results show that source instrumentation is a viable alternative to compiled code instrumentation.

Actually, it may be in some extreme cases, the only alternative. This is illustrated by the program `tak`. Using the standard configuration of ECLiPSe, running this program in debug mode, using the tracer of ECLiPSe and its ad-hoc data structures, results on a heap overflow, while Poppy handles the tracing without any problem. This is the reason why there is a bar missing for `tak`.

Looking at the results obtained with ECLiPSe, another bad result is obtained for `nreverse`. The result was somehow predictable; naive reverse is a totally deterministic program with no failure and no built-in predicate. The transformation introduces two choice points per invocation (one via `trace_call_fail/3` and one via `trace_exit_redo/3`). Assuming the transformation could tell that the program was deterministic, through program analysis, or user annotation, these choice points could be eliminated by replacing calls to `trace_call_fail/3` by calls to `trace_call/3`, and calls to `trace_exit_redo/3` by calls to `trace_exit/3`. Running the corresponding program almost halves the ratio. Looking at the results obtained with SICStus, however, `nreverse` behaves very well. Analyzing theses discrepancies requires further work.

Figure 8 gives relative average speedups, using SICStus in fast compilation mode as a reference. This figure shows the very significant slowdown resulting from setting debugging on[4]. As for source instrumentation, the slowdown is not surprising when looking at the static and dynamic effects of instrumentation. Indeed, the compiled instrumented programs are on average 3.06 (SICStus fast), 3.75 (SICStus compact), and 3.56 (ECLiPSe) times bigger than the compiled initial programs. As far as the execution is concerned, the ECLiPSe profiler shows that, on average, the number of goal invocations is multiplied by 17.6 and the number of choice points by 11.6. Nevertheless, debugging compiled code is faster than debugging interpreted code.

[4]This advocates for making it possible to mix code compiled in optimized mode and code compiled and debug mode.

The figure additionally shows that the debug mode of ECLiPSe is quite effective. The relative slowness of Poppy on SICStus comes, at least partly, from the slowness of the blackboard (see [12]), compared to the global variables of ECLiPSe. Indeed, the blackboard in SICStus and global variables in ECLiPSe are used to implement the call number, whose handling has, according to preliminary profiling, a significant weight.

When tracing in a standard way, *i.e.* mainly on a step by step basis, so much time is taken by the display of the data that the time taken by tracing through instrumentation, even if twice longer than tracing at the abstract machine level, should not be an issue. When looking at elapsed times vs cpu times, it is significant that, compared to their "no write" version, the elapsed time ratios of the three programs **flatten**, **reducer**, and **sdda** are quite good (on ECLiPSe, the ratios are around 1 or below).

When used together with a trace analyzer such as Opium [3], the context is different. The whole trace may have to be examined by the trace analyzer before the user sees any result. Thus the time taken by the tracing mechanism may be an issue. A solution may be, as previously mentioned, to resort to user annotation or better compiler technology to improve the transformation. Another interesting path could be to customize the instrumentation depending on the user request.

5 Conclusion

We have presented a program transformation making it possible to trace Prolog programs without resorting to a low-level tracer.

As far as efficiency is concerned, we have shown that a typical slowdown of 2 is observed, in the less favorable environment, when generating the trace of standard benchmark programs. Considering the advantages in terms of portability and versatility, this slowdown is quite acceptable. We do not expect it to be significant when dealing with "standard" tracing, dominated by input/output. This may be more of an issue when connecting tracing to a trace analyser such as Opium [3]. Indeed, large fractions of the execution may need to be analysed and therefore traced. We are currently extending our first prototype with a connection to Opium to check the issue.

If performance turns out to be an issue, it could be interesting to resort to program analysis to improve the transformation. Another possibility would be to customize the instrumentation according to the user requests.

We are currently working on the prototype, adding some more basic tracing facilities and making the implementation more portable. This would provide Prolog systems without proper tracing facilities with a standard tracer. This would also make it possible to assess the approach in different Prolog environments, comparing it with other trace extraction approaches, and looking at the influence of the compilation technology.

Acknowledgements Erwan Jahier and Sarah Mallet gave fruitful comments on earlier versions of this article.

References

[1] L. Byrd. Understanding the control flow of Prolog programs. In S.-A. Tärnlund, editor, *Logic Programming Workshop*, Debrecen, Hungary, 1980.

[2] M. Calejo. *A framework for declarative Prolog debugging.* PhD thesis, New University of Lisbon, Portugal, March 1992.

[3] M. Ducassé. Opium: An extendable trace analyser for prolog. *The Journal of Logic programming*, 1999. To appear in the special issue on Synthesis, Transformation and Analysis of Logic Programs, A. Bossi and Y. Deville (eds), Also Rapport Technique INRIA 3257.

[4] M. Ducassé and J. Noyé. Logic programming environments: Dynamic program analysis and debugging. *The Journal of Logic Programming*, 19/20:351–384, May/July 1994.

[5] ECRC, Munich, Germany. *ECLiPSe 3.5 - User Manual*, Dec. 1995.

[6] M. Eisenstadt. A powerful Prolog trace package. In *Proceedings of the 6th European Conference on Artificial Intelligence*. North-Holland, 1985, September 1984.

[7] B. Holmer, B. Sano, M. Carlton, P. Van Roy, R. Haygood, W. Bush, and A. Despain. Fast Prolog with an extended general purpose architecture. In *Proceedings of the* 17^{th} *Annual International Conference on Computer Architecture*. IEEE/ACM, IEEE Computer Society Press, May 1990.

[8] M. Meier, A. Aggoun, D. Chan, P. Dufresne, R. Enders, D. Henry de Villeneuve, A. Herold, P. Kay, B. Perez, E. van Rossum, and J. Schimpf. SEPIA - an extendible Prolog system. In *Proceedings of the IFIP '89*, 1989.

[9] J. Noyé. An overview of the Knowledge Crunching Machine. In M. Abdelguerfi and S. Lavington, editors, *Emerging Trends in Database and Knowledge-base Machines*. IEEE Computer Society Press, 1994.

[10] P. V. Roy and A. M. Despain. High-performance logic programming with the Aquarius Prolog compiler. *Computer*, 25(1):54–68, Jan. 1992.

[11] G. Specht. *Source-to-source Transformationen zur Erklärung des Programmverhaltens bei deduktiven Datenbanken*. PhD thesis, Technischen Universität München, Juni 1992. In German.

[12] Swedish Institute of Computer Science. *SICStus Prolog User's Manual*, march 1998. Release 3.7 beta.

[13] A. Taylor. High performance Prolog on a RISC. *New Generation Computing*, 9:221–232, 1991.

[14] A. Tolmach and A. Appel. A debugger for Standard ML. *Journal of Functional Programming*, 5(2):155–200, April 1995.

SICStus MT—A Multithreaded Execution Environment for SICStus Prolog

Jesper Eskilson Mats Carlsson
Intelligent Systems Laboratory
Swedish Institute of Computer Science
Box 1263, SE-164 29 Kista, Sweden
jojo@sics.se, matsc@sics.se

Abstract

The development of intelligent software agents and other complex applications which continuously interact with their environments has been one of the reasons why explicit concurrency has become a necessity in a modern Prolog system today. Such applications need to perform several tasks which may be very different with respect to how they are implemented in Prolog. Performing these tasks simultaneously is very tedious without language support.

This paper describes the design, implementation and evaluation of a prototype multithreaded execution environment for SICStus Prolog. The threads are dynamically managed using a small and compact set of Prolog primitives implemented in a portable way, requiring almost no support from the underlying operating system.

1 Introduction

Prolog has historically been centered around the concept of asking queries to a database and receiving answers and other output on a terminal or other output device. This has proven to be an excellent way of providing many software solutions.

But computer applications have become more and more complex, involving many independent and different subproblems. For example, a WWW-server normally consists of a part which continuously listens for connections on sockets. This is usually done by employing multiple *threads*. Doing it without threads forces the programmer to manually switch between executing the different subproblems, making the program inefficient and difficult to write and maintain.

Now, threads is not a new concept in Prolog or in other languages. Most modern languages support threads in some way. Most notably are C [16, 19], ERLANG [1], Java [11], and Oz 2.0 [23, 24]. Many logic programming systems implement threads (although they are mostly referred to as "processes"), such as CS-Prolog [12], KL1 [7], IC Prolog II [8], Multi-Prolog [4], but these are all relatively small and experimental systems. There are no (to the best of our knowledge) widely-used commercial Prolog systems supporting threads.

Summing up, multithreading is a necessary language requirement in order to facilitate the development and maintenance of large software systems, such as WWW-servers and intelligent software agents. This paper presents the design, implementation and evaluation of such an extension to SICStus Prolog [6]. Different options are discussed for the following design issues: whether to use native threads, scheduling algorithm, time quanta and thread switching, programming interface, communication and synchronization, and blocking system calls. A preliminary performance analysis is given.

The rest of the paper is organized as follows: The necessary concepts are defined in Section 1.1. We then state the requirements of our prototype in Section 1.2. We then describe the important design issues (Section 2, 3, 4) and their impact in the implementation. Section 5 describes the programming interface and Section 6 describes the solution to the problem of blocking system calls. Sections 7 and 8 evaluate the performance of our implementation and include references to related work. We end with some conclusions.

1.1 Preliminaries

Since SICStus Prolog (like most other Prologs) is executed on a virtual machine–also referred to as *emulator*–it is important to realize where the multithreading comes in. This paper discusses the implementation of *Prolog threads*, i.e. where two or more predicates appear to be running in parallel. This should not be confused with using *native threads* where parts (functions) of the emulator appear to run in parallel by using a native threads package such as *Pthreads* [19]. See Section 2 and Section 9.

1.2 Requirements

The purpose of this work has primarily been to prove that support for multiple threads is realistic in an industrial-strength Prolog system. To prove this, we must show that introducing support for multiple threads does not cause any significant inconveniences for the user such as reduced execution speed or excessive memory consumption. In other words, the prototype must provide the benefits of multiple threads without removing any (or as little as possible) of the benefits of *not* supporting threads.

2 Native Threads

The first design issue which needs to be settled is whether to use native threads. Native threads have two major benefits. The first, and perhaps most important, is that they make it possible to utilize multiple CPUs (and other machine- and operating-system specific features). The second is related to blocking system calls and will be discussed in Section 6.

We have chosen not to use native threads at all in this first version of SICStus MT, for a couple of reasons. First, native threads are not available on all platforms. If this prototype relies on native threads we restrict our results to those platforms with native thread support. Second, native threads are not crucial in order to prove the feasibility of threads in a industrial-strength Prolog. Third, native threads can be implemented later on if desirable. That is, avoiding native threads in the prototype does not rule them out completely. See also Section 9.

3 Necessary Modifications

The two basic characteristics of any language implementation are its storage and execution models. In order to support multithreading in SICStus, changes are needed to both of these.

3.1 Storage Model

The storage model need to be modified to that some data-areas are kept to each thread. There are five kinds of data-areas in a WAM-based [2] emulator:

The Static Area contains a variety of objects, such as interpreted and compiled clauses, atoms, indexing tables, and so on.

The Local Stack is also called environment stack and contains procedure frames, which mainly consist of permanent variables (variables that survive predicate calls).

The Global Stack contains Prolog terms. This is usually the largest area.

The Choicepoint Stack contains the choicepoint records. These record contain information about what to do during backtracking.

The Trail Stack contains conditional variable bindings, i.e. variables that should be reset to unbound upon backtracking.

In addition to this, we have the set of abstract machine registers organized as a data structure *WS* for *W*orker *S*tructure, which contains program counters, stack boundaries, choicepoint-information, etc.

The bulk of the static area is kept global in order for threads to be able to share code. The local, choicepoint, and trail stack must be kept private, since they are directly related to how the program is executed. The same goes for the abstract machine registers, the WS. The WS is combined with thread-related information (such as status-flags, thread ID, message-port, etc.) to form a data-structure representing a thread.

The global stack has to be kept private to each thread. Even if it would be attractive to employ a shared global heap to be able to share data between threads (such as in Oz 2.0), there are several complications. For example, it would be very difficult to support multiple threads backtracking at the same time.

3.2 Execution Model

Like the storage model, the execution model needs to be modified in order to support multiple threads. Since native threads are not utilized and due to our requirements on the scheduler (see Section 4), the execution model must support time-sharing the emulator between the different threads.

The main problem to solve is to determine where in the emulator loop threads should be switched in and out. The place where this is done is called *the synchronization point*. A natural candidate for the synchronization point is the *overflow-check*. This is a handler for certain kinds of asynchronous events, such as stack overflows, goals woken by variable bindings, and goals invocated by interrupts. The benefit of using the overflow-check as the location of the synchronization point is that the execution state is fully well-defined, so that a context switch between two threads is trivially implemented by exchanging references to the WS.

4 Scheduling

Scheduling [15] can be compared to motion picture soundtracks: if it is done well, it is not noticed—it just contributes to the overall impression of the performance.

The main requirements of the scheduler, apart from having a low scheduling overhead, is that it should be *preemptive*. Without preemption, it is impossible to write applications which use threads to perform independent work simultaneously (like serving two clients at the same time).

The algorithm used in SICStus MT is a variant of Round-Robin scheduling [22, 25], called Priority Round-Robin (PRR). PRR scheduling is conducted like Round-Robin scheduling with the difference that Round-Robin is only practiced among threads with equal priority. Among threads with different priority, the priority determines which thread is scheduled for execution. This algorithm is far from perfect. The main disadvantage is that it is not *fair*[1]. However, it is robust, easy to implement, and provides good performance in most cases.

4.1 Choice of Time Quantum

The size of the time quantum is crucial to (P)RR scheduling. The time-quantum is defined as the maximum period of time a thread is allowed to execute before it is interrupted by the scheduler. The size of this period has a large impact on the efficiency of the scheduling. If the period is too large, response times will become too long. If the period is small, the scheduling overhead will become too large.

1. One possible way of obtaining a time quantum is setting up a timer interrupt and thereby rescheduling at a fixed-time interval (for example, 50 ms), rounded up to the next arrival at the synchronization point (see Section 3.2).

2. It is also possible to reschedule based on the number of executed WAM instructions. This has the benefit of not requiring any timer interrupts. It can also be used if the same execution sequence must be repeated (for debugging purposes, for example). The major problem is efficiency; the overhead turned out to be almost 20%. It would also require substantial modification to the native code kernel, which would be tedious to implement and would also degrade the performance of native code execution.

3. By reducing the granularity of WAM instruction counting and instead reschedule every *n*th arrival at the overflow-check, we obtain a reasonable time-quantum (empirical tests showed that the overflow-check occured spontaneously roughly with the same frequencies as a timer-interrupt set of 50 ms), and at the same time we are not bound to using timer-interrupts.

5 Programming Interface

The set of predicates used to create, destroy, and in other ways manipulate threads has deliberately been kept to a minimum in order to keep the implementation simple and the ideas clear. The following predicates are defined by SICStus MT.

`spawn(:Goal, -ThreadID)` Spawns a new thread and schedules it for execution. The new thread will execute the goal `Goal`, similarly to the predicate `call/1`. `ThreadID` will be bound to the identifier of the new thread. Together with the new thread, a *message port* (also referred to as *input queue*) will be created. This port is intended to be used for synchronization and communication between different threads.

`send(+ThreadID, +Term)` Sends `Term` to the thread indicated by `ThreadID`. This predicate always succeeds (or throws a domain error exception). `Term` will be inserted last in the in the receiving thread's input queue. Unbound variables are consistently copied, i.e. `f(X,X)` becomes `f(Y,Y)` on the other side.

[1] A *fair* scheduling mechanism guarantees that threads will not starve, regardless of the number of threads waiting to run.

`receive(?Term)` Extract the first element in the thread's input queue that is unifiable with `Term`. If no such term exists, the thread is suspended.

`self(-ThreadID)` `ThreadID` is the thread identifier of the running thread.

`kill(+ThreadID)` Causes `ThreadID` to terminate. Always succeeds.

`wait(+Ms)` Suspends the currently running thread and then waits at least `Ms` milliseconds before resuming. The actual time elapsed before the thread is resumed is guaranteed to be *at least* `Ms` but could exceed this limit depending on two factors; the frequency of the timer-interrupts and the number of threads waiting to run.

See Figure 1 for examples of how to use these predicates.

```
reader :-
    receive(readlock(NextReader)),
    read(Term),
    self(Self),
    send(NextReader, readlock(Self)),
    reader.

run :-
    spawn(reader, ReaderA),
    spawn(reader, ReaderB),
    send(ReaderA, readlock(ReaderB)),
    wait(120000),
    kill(ReaderA),
    kill(ReaderB).
```

Figure 1: Example of using the "send" and "receive" primitives. This example spawns two threads which take turns reading one term each from standard input. After 2 minutes, both threads are killed and the program terminates.

5.1 Communication and Synchronization

The model of communication and synchronization is *message-based* as opposed to *blackboard-based* [3, 26], also known as *tuple space based* [13]. This means that the communication is based on sending explicit messages as opposed to using a shared store of some kind.

In our solution, each thread has a *unidirectional* message port where it can receive messages, consisting of standard Prolog terms. The port is invisible, i.e. it is treated as an integral part of the thread. A message can be any kind of Prolog term. Unbound variables are allowed, but these will be duplicated.

The communication mechanism is *asynchronous*. This means that the sender does not need to wait until the receiver is ready to receive the message. This means also that the mechanism is buffered, i.e. the communication media (the input queue) has a memory of its own where it can store messages until they are ready to be picked up by the receiving thread. The message port is basically a FIFO-structure, which means that it is completely ordered. However, as we will discuss later on, the programmer can specify the order in which terms are received.

Since shared heaps is not an option, sending a message must be done by copying it into the static area and inserting it into the receiving thread's input queue. If the receiving thread is suspended on a call to `receive/1` it is *lazily* switched in for execution, i.e. just moved to the ready-list. If the receiving thread is suspended for some other reason, nothing happens. The alternative would be *eager* thread switching, i.e. preempting the receiving thread, disregarding any priorities. See Section 7 for a discussion on the performance of these two approaches. When the receiving thread is eventually resumed, it must unpack the message on its own heap.

5.1.1 Receiving Messages "Out of Order"

The ERLANG implementation of the `send` and `receive` primitives features a very practical construction: the ability to specify which messages to accept for a particular call to `receive`. This is also referred to as *message non-determinism* [3].

In the Game-of-Life benchmark, described in Section 7, there is a particular construction which relies on out-of-order receives. Since the cells work asynchronously (without a global "conductor" telling them when to do a state transition), each cell needs to make sure that the incoming messages are grouped by generation. This means that if a cell in generation x receives a message from a cell which is in generation $x + 1$, it must be able to defer that message until itself is in generation $x + 1$. In our implementation, this is solved by letting each cell loop through all its neighbors and for each one, wait for a message from that particular neighbor. In this way, we are guaranteed that the messages are processed in the correct generation. This would be very tedious to code without language support, since we would need to keep a separate list of terms which were received "too early", a list which needs to be maintained, sent around to all predicates calling `receive/1`, and searched for each call to `receive/1`.

However, there are performance issues worth discussing here. Each time the receiving thread tries to execute a `receive(Term)` goal (either the first time around, or upon a thread switch in case it was suspended), it has to traverse the input queue, unpacking each message, delete if from the queue if it unifies with `Term`, otherwise keep searching. Unpacked messages that do not unify are reclaimed by Prolog's backtracking and so must be unpacked again the next time around. Even though the time to unpack a message is linear in its size, a given message may get re-examined many times. Also, out-of-order receives will result in a list of "currently unmatched" messages, i.e. messages that have arrived but are not unifiable with the argument to `receive/1`. This means that messages can be delayed in the input queue for a potentially indefinite period of time. See Section 7 for some performance figures.

6 Blocking System Calls

The problem of blocking system calls in user-level (as opposed to kernel-level) thread implementations is a well-known and well-investigated problem [25, 10]. The core of the problem is that the operating system kernel (by definition) is unaware of the existence of user-level threads. Therefore, when a blocking system call is performed, the kernel suspends the entire process for the duration of the system call, instead of scheduling another thread for execution, which is the desirable behavior.

6.1 Possible Solutions

There are not that many ways of solving the problem. We must in some way prevent a given blocking system call from blocking the entire process and find a way of scheduling another thread

instead. We have explored two approaches to the problem, the *cautious approach* and the *cavalier approach*.

The cautious approach uses a relatively complex mechanism in order to examine system resources in order to determine, without making the call, whether or not the system call would block the process. If the call could not be performed without blocking the process, the thread is suspended and another thread is switched in. Otherwise, the thread continues with the read and returns normally. The main problem of this approach is its complexity. Each system call which might block must be preceded by a piece of code (called *jacket*) in order to determine whether or not the system call would block or not. This turned out to be quite non-trivial—the documentation on when system calls block is often inadequate and examining system resources not a very portable procedure. Another drawback is the additional overhead of the jacket code.

The cavalier approach relies on the fact that most operating systems support some way of performing system calls without blocking the process at all, commonly known as *asynchronous I/O*. Instead of trying to determine on beforehand whether or not a system call is about to block, we simply perform the system call asynchronously. A check is made after the call to determine if the call was completed and if not, the thread is suspended and then resumed when the asynchronous system call can be retried (as a result of a `SIGIO` signal indicating I/O completion).

The cavalier approach wins the game on the fact that it is simpler to implement and more robust. We leave it to the individual system call to determine whether or not it is about to block. This relieves us from having to write specialized code for each system call which is not only tedious but also error prone.

6.2 Emulator Support

The solutions discussed above both need a mechanism for communicating with the emulator. More precisely, they need to be able to inform the emulator when a thread should be suspended as a result of a blocking system call. The idea is to introduce an extra return code for predicate-calls in addition to `TRUE`/`FALSE` (which represent success and failure, respectively). The new return code is called `SUSPEND` and is returned when the thread executing the predicate should be suspended.

Since the process of suspending a thread as a result of making a blocking system call varies significantly depending on the nature of the system call, the actual work of suspending a thread (setting bits, moving threads between lists, etc.) is done by the code performing the system call. The only action required by the emulator is to immediately jump to the synchronization point in order to perform a thread switch.

6.2.1 Suspending The Emulator

The emulator will sometimes be in the situation where there are no more threads to schedule. For example, this happens immediately at startup in the development system when the top-level thread waits for input from the user. This should cause the Prolog process to suspend itself, just as if it would have if it had performed a normal, blocking system call.

This is implemented by a small piece of code in the scheduler. When the scheduler has suspended the top-level thread and realizes that the ready-list is empty, it suspends the process [2] by calling `pause()`. The process is then resumed (i.e. `pause()` returns) when a signal is received. The signal is triggered by one of two reasons. Either the asynchronous I/O mechanism sends a signal to

[2]Suspending and resuming processes is—as one would expect—platform-dependent. We will only describe the procedure used under Solaris; we expect the procedure to be fairly similar on most modern operating systems.

indicate that the I/O call was completed, or the timer mechanism sends a signal to indicate that we have one or more threads suspended on a call to `wait/1` and that we need to examine the queue to schedule those threads for which the time-quantum has expired. These actions are taken in the signal-handler routines, so that when `pause()` returns, we examine the ready-list and if everything went right we should have a thread waiting to execute. However, for different reasons we might have received a false alarm, in which case we will simply be suspended again.

7 Performance Evaluation

We briefly evaluate the space complexity and execution speed of our implementation. We have used two benchmark programs: *Game-of-Life* [14], a simulation of a simple biological environment, and a matrix multiplication program. The purpose of the former benchmark is to measure message-passing overheads, whereas the latter benchmark gives an idea of the intrinsic scheduling overheads.

7.1 Memory Consumption

Naturally, the objective during the implementation process has been to keep the threads as light-weight as possible. It is quickly realized that the space occupied by the WS is negligible compared to the stack-consumption. The WS is outweighed 64:1 if the global, local, choicepoint, and trail stacks are 32k, 16k, 8k, and 8k, respectively. These figures are the recommended ones. Setting them lower causes excessive stack-shifting and memory fragmentation.

Related to this is the fact that the address space in SICStus is only $2^{28} = 256$ Mb (on a 32-bit architecture) since the 4 upper bits are used for tagging data-cells. This not only means that the data-areas cannot exceed 256 Mb, it also means that they have to be located at optimal places for this to be possible. In other words, if the address space becomes fragmented (by stack-shifting, for example), the limit of 256 Mb will drop even further. The impact of this limit will become clear in the next section.

7.2 Execution Speed

The figures in the following section have been obtained by running SICStus MT executing emulated code on a Sun UltraSPARC 248 MHz.

7.2.1 Game-of-Life

This is a variant of Conway's Game Of Life [14], a simulation of a simple biological environment. Basically, it consists of a matrix of cells, each of which may or may not contain an organism. There are rules for how the live cells reproduce for each generation and depending on the initial population, many interesting and fascinating patterns occur. We have implemented the game in a way which makes it an excellent benchmark for measuring the efficiency of employing several (hundreds or even thousands) threads. By spawning a separate thread for each cell in the matrix and using the inter-thread communication mechanism to propagate information, we can get good information about the scheduling and communications overhead. See Figure 2.

The Oz 2.0 benchmark is not as relevant as the other three since it does not use the message passing mechanism to propagate information. This eliminates the overhead of scheduling threads in order to receive messages (which turns out to be the major bottleneck in our implementation).

The Java-implementation (Sun's JDK 1.1.4) displays surprisingly bad performance, despite the fact that no message copying is done. We also found that there is also only a very small difference

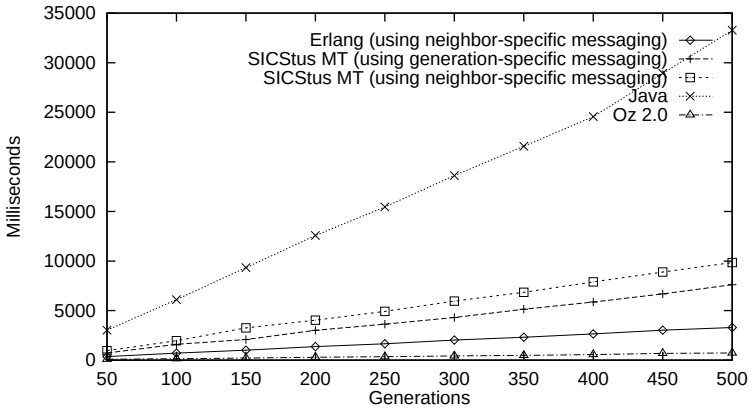

Figure 2: Execution times for the Game-of-Life benchmark. The board is 10 $\times$ 10 cells.

between JIT-compiled Java and byte-interpreted Java, which leads us to believe that it is the Java scheduler which is inefficient.

Comparing lazy switching and eager switching (see Section 5.1) using the Game-of-Life benchmark showed that lazy switching is more efficient (approximately 700 ms or 7%). This is caused by the fact that lazy switching decreases the number of messages that are unpacked in vain (i.e. messages that do not match the first argument of the call to `receive/1`).

There is a considerable overhead in the message passing mechanism. The overhead can be divided into two parts. The first part is caused by having to copy messages between heaps. This is not a very big overhead, especially when the messages are small. The dominating overhead is caused by the out-of-order receive mechanism (see Section 5.1.1). This mechanism allows the receiving threads to be more or less specific about which messages should be received. The problem is that the sending thread has no way of finding out if a message is "appropriate", i.e. if it matches the first argument to `receive/1`. Instead, it must always resume the receiving thread so it can make the decision itself. This is where *neighbor-specific* and *generation-specific* messaging come in. The former means that messages are only identified by their generation and the latter means that messages are identified by the sending cell (or thread). Neighbor-specific messaging causes more messages to arrive out-of-order, and hence induces a higher overhead which can clearly be seen in the figure.

This overhead can be reduced by being more selective about when to resume threads blocked on `receive/1`, so we minimize the number of threads which are resumed in vain. Lazy switching is one way of doing this. Another way is to allow the *sending* thread to inspect the first argument to `receive/1` (in the receiving thread, that is) and avoid resuming threads when the message does not match. This should enable us to eliminate all unnecessary scheduling of threads blocked on `receive/1`.

Yet another improvement is to utilize indexing [20] to search the input queue. The queue is currently searched linearly, but by using indexing, the search would become considerably more efficient. The actual improvement, however, depends heavily on the application. If the average length of the input queues is large (which is the case if there are many messages arriving out of order), this improvement will have larger impact than if the queues are mostly empty.

7.2.2 Matrix Arithmetic

Since our implementation of the Game-of-Life benchmark relies on the existence of threads, it is difficult to use that benchmark to get a grip on the overhead of SICStus MT compared with single-threaded SICStus.

To get some figures on this, we used a very simple matrix multiplication benchmark which can operate in two different modes, *threaded* and *meta-called*. The first spawns a thread for each cell in order to compute its value and the second simply performs a meta-call (using `call/1`), thereby not spawning any threads. These two modes corresponds to the *Parallel MT* and *Sequential MT* lines in the graph in Figure 3. The third line is obtained by running the matrix benchmark in sequential mode but using plain SICStus (version 3#5).

While the difference between lazy and eager switching was quite small in the Game-of-Life benchmark, it made a considerable difference here. The main thread receives a messages from each thread when it has completed, and eager switching caused the main thread to start executing for each such message while lazy switching only let the main thread execute once all multiplier threads had completed. The difference was large enough (roughly a factor 2 compared to Sequential MT) to exclude eager switching from an interesting comparison.

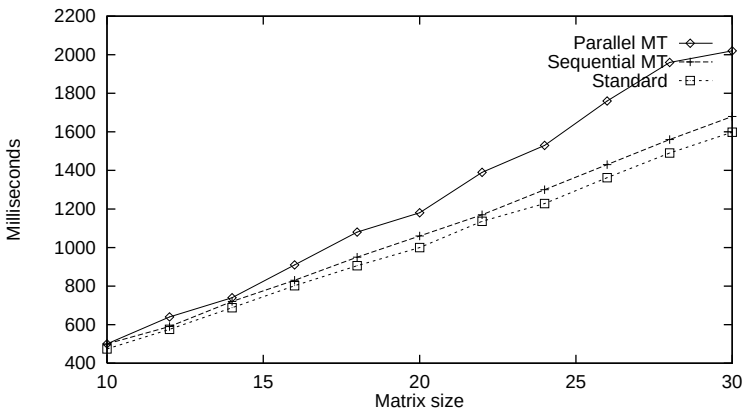

Figure 3: Execution times for multiplying a matrix using different version of SICStus.

The data in Figure 3 tell us that the overhead of supporting threads is very small (the benchmark had to be executed several times in order to obtain a reliable difference). This means that it is reasonable to include support for multiple threads in a released version of SICStus.

The benchmark also shows that the overhead of spawning a thread as opposed to doing a meta-call is reasonably small.

7.2.3 Performance Conclusion

Added up, it is reasonable to include support for multiple threads in a released version of SICStus, but the message passing overheads needs to be reduced in order to be competitive with implementations such as Erlang and Oz 2.0.

8 Related Work

A great deal of work on parallel execution of Prolog has been done during the past 15 years; see e.g. [9, 17]. However, this work has mostly been concerned with exploiting the parallelism which is implicit in logic programs.

Several concurrent logic programming system based on explicitly creating processes (or threads) have evolved during the last 15 years. A survey of the general issues can be found in [3].

CS-Prolog Professional [18] supports multiple processes and uses the message-passing paradigm for communication using explicitly created channels as media. Asynchronous message passing is not supported; instead a rendezvous (or hand-shake) model is used. However, forcing threads to synchronize whenever communication takes place tends to encourage deadlocks and is generally restrictive to the programmer.

KL1 [7] is a concurrent logic programming language based on *GHC* (Guarded Horn Clauses). The concurrency is similar to that of SICStus MT; threads are managed explicitly and communication is done using blocking variable bindings.

Multi-prolog [4] and *BlackLog* [21] use the blackboard paradigm for communicating between processes. Blackboard-based systems are inherently less scalable in applications where there is heavy communication between processes since all messages need to pass through one single point. On the other hand, blackboards tend to give the programmer more expressiveness since messages do not need to be sent to an explicit destination.

9 Future Work

9.1 Implementing Native Threads

The perhaps most interesting area for future work is the incorporation of native threads. As mentioned in section 2, native threads are not used at all in this prototype implementation. However, the benefits of using native threads (multiple CPU utilization, blocking systems calls, etc.) are important enough to justify native threads in a released version of SICStus.

Incorporating native threads raises a couple of design issues itself. For example, which native threads package should be chosen? Portability is an important factor in this choice. A second question is how to map Prolog threads on native threads. Should the presence of native threads be visible to the user, or should it be handled completely by SICStus itself? A third question is how do we handle the extra overhead connected with locking global datastructures, such as the Prolog database? These questions need to be answered before a large-scale implementation of native threads in SICStus is initiated.

9.2 Debugger

Debugging a multithreaded application is not an easy task. However, we are not really talking about debugging Prolog code (for which SICStus has very good support), but rather about debugging the concurrency introduced by multithreading. Extending the existing debugger with the basic support for debugging multithreaded code is not that difficult; the problem is the support for more advanced features such as deadlock detection, debugging race conditions, etc.

10 Conclusion

We have presented the design and implementation of SICStus MT, a multithreaded extension to SICStus Prolog. The following design issues were discussed in detail: whether to use native threads, the scheduling algorithm, time quanta and thread switching, programming interface, communication and synchronization, and blocking system calls. We gave a preliminary performance analysis, indicating minimal scheduling overheads, but significant message passing overheads in the implemented prototype. Work is under way to reduce these overheads.

References

[1] J. Armstrong, R. Virding, C. Wiström, and M. Williams. *Concurrent Programming In Erlang.* Prentice Hall, second edition, 1996.

[2] H. Aït-Kaci. *Warren's Abstract Machine—A Tutorial Reconstruction.* MIT Press, 1991.

[3] K. D. Bosschere. Process-based parallel logic programming: A survey of the basic issues. In Bosschere et al. [5].

[4] K. D. Bosschere and J.-M. Jacquet. Multi-Prolog: Definition, Operational Semantics and Implementation. In D. S. Warren, editor, *Proceedings of the ICLP'93 conference*, pages 299–313, Budapest, Hungary, June 1993. The MIT Press.

[5] K. D. Bosschere, J.-M. Jacquet, and A. Brogi, editors. *ICLP94 Post-Conference Workshop on Process-Based Parallel Logic Programming*, June 1994.

[6] M. Carlsson, J. Widén, J. Andersson, S. Andersson, K. Boortz, H. Nilsson, and T. Sjöland. SICStus Prolog User's Manual. SICS Technical Report T91:15, Swedish Institute of Computer Science, June 1995. Release 3 #0.

[7] T. Chikayama, T. Fujise, and H. Yashiro. A portable and reasonably efficient implementation of KL1. In Warren [27], page 833.

[8] D. Chu. I.C. Prolog II: a Multi-threaded Prolog System. In E. Tick and G. Succi, editors, *ICLP-Workshops on Implementations of Logic Programming Systems*, pages 17–34. Kluwer Academic Publishers, 1993.

[9] J. Conery. *The AND/OR Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, University of California at Irvine, 1983.

[10] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems—Concepts And Design.* Addison-Wesley, second edition, 1994.

[11] D. Flanagan. *Java in a Nutshell.* O'Reilly & Associates, second edition, 1997.

[12] I. Futó. Prolog with communicating processes: From T-Prolog to CSR-Prolog. In Warren [27], pages 3–17.

[13] D. Gelernter. Generative communication in linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, January 1989.

[14] J. Horton. Computer Recreations. *Scientific American*, March 1984.

[15] B. W. Lampson. A scheduling philosophy for multiprocessing systems. *Communications of the ACM*, 11(5):347–360, May 1968.

[16] B. Lewis and D. J. Berg. *Threads Primer—A Guide To Multithreaded Programming*. Prentice Hall, 1996.

[17] E. Lusk, D. H. Warren, S. Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[18] ML Consulting and Computing Ltd, Applied Logic Laboratory, Budapest, Hungary. *CS-Prolog Professional User's Manual, Version 1.1*, 1997.

[19] B. Nichols, D. Buttlar, and J. P. Farrell. *Pthreads Programming*. O'Reilly & Associates, 1996.

[20] R. Ramesh, I. Ramakrishnan, and D. Warren. Automata-Driven Indexing of Prolog Clauses. In *Proceedings of the Principles of Programming Languages*, 1990.

[21] D. G. Schwartz. *Cooperating Heterogenous Systems: A Blackboard-based Meta Approach*. PhD thesis, Department of Computer Engineering and Science, Case Western Reserve University, 1993.

[22] A. Silberschatz and P. B. Galvin. *Operating Systems Concepts*. Addison-Wesley, fourth edition, 1994.

[23] G. Smolka. The Oz programming model. In J. van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.

[24] G. Smolka. Problem solving with constraints and programming. *ACM Computing Surveys*, 28(4es), Dec. 1996. Electronic Section.

[25] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 1992.

[26] H. Taylor. Design of a resolution multiprocessor for the parallel virtual machine. In Bosschere et al. [5].

[27] D. S. Warren, editor. *Proceedings of the Tenth International Conference on Logic Programming*, Budapest, Hungary, 1993. The MIT Press.

Extension of the Revised Simplex Method for Detecting Implicit Equalities in Constraint Logic Programming

Philippe Refalo*
ILOG, S.A.
9 rue de Verdun - BP 85,
94253 Gentilly, France
refalo@ilog.fr

Abstract

This paper deals with the incremental detection of implicit equalities using the revised simplex method. This algorithm is more efficient and more suitable to practical problems than the tableau method that is usually applied in constraint logic programming. We describe and discuss the adaptation to the revised simplex of three approaches: the CLP($\mathcal{R}$) approach, the Prolog III approach and the quasi-dual one. All of them have been integrated into the constraint logic programming language Athena based on a revised simplex method over exact-precision rationals. This system is used to compare these methods on a set of typical CLP problems over linear constraints.

keywords: constraint logic programming, incremental linear constraint solving, implicit equalities.

1 Introduction

Detecting implicit equalities is closely related to solving linear constraints in constraint logic programming (CLP). An implicit equality is an inequation $a_1x_1 + \cdots + a_nx_n \leq a_0$ from a set of constraints S that can be converted into the equation $a_1x_1 + \cdots + a_nx_n = a_0$ without changing the set of solutions of S. For instance in the set $\{x_1 + x_2 \leq 2, 1 \leq x_1, 1 \leq x_2\}$ all inequations are implicit equalities. Indeed this set can be rewritten to $\{x_1 + x_2 = 2, x_1 = 1, x_2 = 1\}$ without changing its set of solutions.

Detecting implicit equalities is one of the fundamental operations for solving generalized linear constraints [13] that include disequations ($\neq$) and strict inequations ($<$). This method for solving constraints is integrated into languages like Prolog III [5] and Prolog IV [7].

This detection enables the discovering of all variables fixed to a single value. This is applied to the delayed solving of linear constraints [6]. In languages like CLP($\mathcal{R}$) [10] or Prolog III [5], solving the non linear constraint $2xy + 3z + t = 0$ is delayed until one of the variables x or y is fixed by the linear constraint solver. For instance, if x becomes fixed to 3, the constraint $6y + 3z + t = 0$ is then added to the linear constraint solver.

*A part of this work was done when the author was in Laboratoire d'informatique de Marseille, Faculte des sciences de Luminy, 13009 Marseille - France

Detecting implicit equalities is also usefull for cooperating constraint solvers that associate a constraint propagation based solver (on intervals of finite domains) with a linear constraint solver. The former gives bounds on variables to the latter while the latter maintains a rational solution of linear constraints and gives the former fixed variables [2, 7]. For instance in the set $\{x - y + z \leq 0,\ y - z \leq 0,\ 0 \leq x, x \leq 2, 0 \leq y, y \leq 2x \in \{0,1,2\}, y \in \{0,1,2\}\}$, the inequation $x \leq 0$ is an implicit equality. Consequently x is fixed to 0 and the new domain of y is $\{0,1\}$. This inference is not possible with standard finite domain constraint solvers.

Linear constraint solving mainly uses a Gaussian elimination method (for equations and disequations) and on the simplex method (for inequations[1]). Implicit equalities are incrementally detected by extensions of the simplex method. Among the first methods developed for this purpose, there are the `Prolog III` method [8, 5], the `CLP(`$\mathcal{R}$`)` method [16, 10] and the `CHIP` method [18], which is now also used in `Prolog IV` [1]. These methods are implemented with a tableau simplex algorithm over non-negative variables.

This work deals with the incremental detection of implicit equalities with the revised simplex method. This method is widely used for solving practical linear optimization problems [4, 3]. In comparison with the tableau simplex method, the revised simplex method has the advantages of being much more efficient (many optimizations are possible), more numerically stable (accuracy of computations can be controlled), and of using less memory (the rate of non-zero coefficients increases rapidly with the tableau simplex and not with the revised simplex).

The revised simplex method is applied to solving linear constraints in new generation CLP languages like `CLAIRE` [2] or `Athena` [15, 14]. Detection of implicit equalities is then based on the quasi-dual formulation of a system of inequations introduced by Lassez [11].

In this article we describe how to incrementally detect the presence of implicit equalities. We also detail adaptations to the revised simplex of the `Prolog III` method, the `CLP(`$\mathcal{R}$`)` method and the quasi-dual based method. The method using the lexicographic form [18] is not detailed here since it requires the simplex tableau to be applied. For each method, we provide the main theoretical results without providing proofs. They can be found in references which are given instead. We also provide the algorithm that identifies all implicit equalities.

These algorithms have been integrated to the CLP language `Athena`. This make quantitative comparison possible. The linear constraint solver of `Athena` is based on a revised simplex method over exact-precision rationals [15]. It includes advanced linear programming techniques such as LU factorization and smart pivoting techniques. This system is used to compare these approaches on a set of typical CLP problems over linear constraints.

The rest of this paper is organized as follows. Definitions and notations are given in Section 2. The solved form for linear constraints and the revised simplex method are presented in Section 3. Section 4 describes incremental linear constraint solving and detection of implicit equalities. Section 5 details the approach of `CLP(`$\mathcal{R}$`)`. The approach of `Prolog III` is described in Section 6, and the quasi-dual method is described in Section 7. Details about implementation and results from the practical comparison are given in Section 8.

2 Definitions and Notations

Let $\mathcal{R}$ be the set of real numbers and $\overline{\mathcal{R}} = \mathcal{R} \cup \{-\infty, +\infty\}$. Let V be an infinite and countable set of variables. A *linear term* is an expression α, αx or $t_1 + t_2$ where α is a real number, x is a variable from V and t_1 and t_2 are linear terms. A *linear constraint* is an expression $t_1 = t_2$ or $t_1 \leq t_2$ where

[1] A strict inequation is usually decomposed to an equation and a disequation

t_1 and t_2 are linear terms. These constraints are called respectively *equation* and *inequation*. A *constraint system* is a finite set of constraints. A variable has an *occurrence* in a term or constraint if it appears with a non zero coefficient in the term or constraint. Let S be a system of inequations. The system noted $S_=$ is composed of constraints of S where the symbol $\leq$ is replaced by $=$.

An *assignment* is a mapping from set V to set $\mathcal{R}$ that assigns a value to each variable of V. An assignment can be extended naturally to a mapping from the set of terms to $\mathcal{R}$. A *solution* of a constraint is an assignment σ such that σ is a solution of the constraint $t_1 \diamond t_2$ if $\sigma(t_1) \diamond \sigma(t_2)$ where $\diamond \in \{=, \leq\}$. An assignment is a solution of a constraint system if it is a solution of each constraint of the system. A system having at least one solution is *solvable*; otherwise it is *unsolvable*. A variable is *fixed* for a constraint system S if every solution of the system assigns the same value to the variable. An inequation $t_1 \leq t_2$ from a constraint system S is an *implicit equality* if every solution of S is a solution of $t_1 = t_2$.

In the following, matrix notations are used. Let A be an $m \times n$ matrix. The expression A_{i*} represents the i^{th} row of A, the expression A_{*j} represents the j^{th} column of A. Let $P \subset \{1, \ldots, m\}$, the expression A_P represents a submatrix of A whose columns are columns of A whose indexes are in P. The inverse of a square matrix M is denoted by M^{-1}.

If e is a row vector of n real numbers, x a column vector of n variables, b a column vector of m real numbers and β a real number, the expression $ex \leq \beta$ represents an inequality and the expression $Ax = b$ represents a system of equations. We implicitly assume compatibility of the sizes of matrices and vectors when they are multiplied. Matrices and sets are denoted by upper case letters, vectors by lower case letters and real numbers by Greek symbols.

3 Solved Forms and Revised Simplex

3.1 Tableau Solved Form

The tableau solved form is composed of a set of equations and a set of inequations that bound the value of variables. A system of linear constraints in tableau solved form is at first in *basic form*:

$$S = \begin{cases} x_B = Mx_L + M'x_U + b \\ l \leq x \leq u \end{cases} \tag{1}$$

where x is a vector of variables, b a column-vector of reals, M and M' are matrices of reals, l and u are vectors of elements of $\overline{\mathcal{R}}$ and where B, L and U are disjoint sets of indexes such that $l_i \neq -\infty$, for $i \in L$ and $u_i \neq +\infty$, for $i \in U$. The set of indexes B is called the *basis* of the system. Variables from vector x_B are called *basic variables*. Variables from vectors x_L and x_U are called *non-basic variables*. The values l_i and u_i are respectively the *lower bound* and the *upper bound* of the variable x_i. For each basic form there exists a unique assignment σ_S such that

$$\sigma_S(x_i) = \begin{cases} l_i & \text{if } i \in L \\ u_i & \text{if } i \in U \\ \sigma_S(t) & \text{if } i \in B \text{ and } x_i \text{ is basic in } x_i = t \end{cases}$$

When this assignment is also a solution of S the system is in *tableau solved form* and σ_S is called the *basic solution* of S. It is well known that any solvable system of linear equations and inequations can be mapped to tableau solved form [4].

3.2 Revised Solved Form

The revised solved form is the basis for the revised simplex method. The main difference from the tableau solved form is the basic variables that are not eliminated in the equations. However eliminating basic variables gives a system in tableau solved form. A system in *revised solved form* is a system

$$S' = \begin{cases} A_B x_B + A_L x_L + A_U x_U = d \\ l \leq x \leq u \end{cases} \tag{2}$$

where A_B is a non-singular square matrix and such that the system

$$\mathcal{T}(S') = \begin{cases} x_B = -A_B^{-1} A_L x_L - A_B^{-1} A_U x_U + A_B^{-1} d) \\ l \leq x \leq u \end{cases}$$

where basic variables are eliminated, is in solved form. The matrix A_B of basic variables coefficients is called the *basic matrix* of system S'.

A *regular* system is a system in revised solved form that does not contain implicit equalities. The purpose of this article is to solve and maintain incrementally regular systems.

3.3 The Revised Simplex Method

The revised simplex method considers a linear program of the form

$$\min\{ax_L + a'x_U \mid S\}$$

The system S is in revised solved form and the term $ax_L + a'x_U$ is an objective function where basic variables have been eliminated. It computes a *solved* linear program (i.e. such that $a_i > 0$ and $a'_i < 0$). The basic solution of S is then an optimal solution of the linear program (see [4]).

The revised simplex method first chooses an *entering variable* x_k such that $a_i < 0$ or $a'_i > 0$. A variation of the value $\sigma_S(x_k)$ decreases the value of the objective function.

The method next chooses a *leaving variable* from basic variables in order to make room for the variable x_k entering the basis and to change the value of x_k. To choose the leaving variable, the vector d of coefficients of x_k in the system $\mathcal{T}(S)$ is computed:

$$d = -A_B^{-1} A_{*k} \tag{3}$$

This vector is used to obtain the variation α_i of x_k (in absolute value) allowed by the bounds of the basic variable x_i and by the equation where x_i is basic. The greatest possible variation for x_k is $\alpha_r = \min \alpha_i$. When $\alpha_r = -\infty$ the algorithm stops. The variation of x_k is unbounded and thus the linear program is also unbounded.

In other cases, the system and the objective function are pivoted over the entering variable x_k and the leaving variable x_r. The variable x_r reaches one of its bounds and its index is added to one of the sets L or U. The index of variable x_k leaves the set where it appeared (L or U) and is introduced in the set B. Let B', L' and U' be the new set of indexes after pivoting. The new basis matrix is $A_{B'}$ and the new objective function $fx_{L'} + f'x_{U'}$ where the basic variables are eliminated, is computed as follows:

$$\begin{aligned} f &= e_{L'} - e_{B'} A_{B'}^{-1} A_{L'} \\ f' &= e_{U'} - e_B A_{B'}^{-1} A_{U'} \end{aligned} \tag{4}$$

It may happen after a pivot that the value of the objective function for the basic solution has not changed. This is the case if $\alpha_r = 0$. The pivot is then *degenerated*.

The efficiency of the revised simplex method hinges around the representation of the matrix A_B^{-1} and the update of this matrix during a pivot. Some of the numerous approaches for this are summarized in [4]. The representation used in `Athena` is an LU factorization of the basis matrix and the update is based on matrix accumulation [15, 14].

4 Linear Constraint Solving

This section briefly presents some fundamental results for incremental solving of equation and inequations and the detection of implicit equalities. References containing more details are provided.

Linear constraint solving in constraint programming must be incremental. ¿From a regular system S, and a constraint c, the basic operation is to compute a regular system S' equivalent to $S \cup \{c\}$.

4.1 Determining Solvability and the Presence of Implicit Equalities

Assume that the constraint c is an inequation $ex \leq \beta$. Determining the solvability of the system $S \cup \{c\}$ and the presence of implicit equalities in this system amounts to solving the linear program

$$\gamma = \min\{ex \mid S\}$$

with, for instance, the revised simplex method. The following three cases have to be considered:

- if $\gamma > \beta$ then $S \cup \{c\}$ is unsolvable ;
- if $\gamma < \beta$ then $S \cup \{c\}$ is solvable and does not contain implicit equalities;
- if $\gamma = \beta$ then $S \cup \{c\}$ is solvable and contains implicit equalities.

The absence of implicit equalities when $\gamma < \beta$ is proven in [13]. This fundamental result leads to efficient incremental methods for detecting implicit equalities. In other words, solving a system that does not contain any implicit equalities does not require anything more than searching for a solution of this system.

Observe that when $\gamma = \beta$, the inequation c is an implicit equality itself. Moreover, it is not the only one in the system $S \cup \{c\}$. Note also that iterations of the simplex method can terminate as soon as a solution is reached that gives the objective function a value strictly inferior to β.

If the constraint c is an equation $ex = \beta$, adding c is equivalent to adding the two inequations $ex \leq \beta$ and $\beta \leq ex$. The basic solution σ_S can often discard one of these two inequations. Indeed, if $\sigma_S(ex) < \beta$, it is sufficient to search for a solution of the inequation $\beta \leq ex$. Conversely, if $\sigma_S(ex) > \beta$, it is sufficient to search for a solution of the inequation $ex \leq \beta$. When $\sigma_S(ex) = \beta$, both inequations must be considered since each of them can be an implicit equality. However, since S does not contain any implicit equalities, if $ex \leq \beta$ is an implicit equality then the inequation $\beta \leq ex$ is not an implicit equality and vice versa.

The constraint is added to the system only if $\gamma < \beta$ (we will see later that it is not necessary to add it when $\gamma = \beta$). This problem have been studied for some time with the tableau solved form [10, 18, 9] and more recently with the revised solved form [2, 15, 14]. We will not go into further details herein since the purpose of this work is to describe the incremental computation of a regular system.

4.2 Determining All Implicit Equalities

Once the presence of implicit equalities is proven, the next step is to determine all implicit equalities of $S' = S \cup \{c\}$ and to replace them by equations to obtain the new system.

A naive approach is to solve a linear program for each inequation $\delta x_i \leq \alpha$ of the system S' to decide if it is an implicit equality or not. In [17] Telgen proposes some improvements to this method. He gives some syntactic criteria to identify some implicit equalities in a tableau solved form. For each system obtained by the application of the tableau simplex, the criteria identify additional implicit equalities. Moreover, the basic solution σ_S that is associated to each system is used to discard any inequation verifying $\sigma_S(\delta x_i) < \alpha$ (that is obviously not an implicit equality). It remains that in the worst case, this approach can solve n linear programs for n inequations in the system.

An ideal method would solve a single linear program for each implicit equality found. Most of the incremental approaches that are described herein are very close to the ideal method and sometimes better.

Each of the incremental method considers first the solved linear program proving the presence of implicit equalities in the system $S \cup \{ex \diamond \beta\}$ where $\diamond \in \{=, \leq\}$:

$$\min\{ax_L + a'x_U \mid S\} \tag{5}$$

This linear program contains informations for determining with certainty a first non empty set of implicit equalities. Let $l \leq x \leq u$ be the set of inequations of system S. The *restrictive inequations* of this linear program are the inequations of S that directly restrict the decrease of the objective function. They are characterized by the set

$$R = \{l_i \leq x_i \mid a_i \neq 0\} \cup \{x_i \leq u_i \mid a'_i \neq 0\}$$

As a fundamental result, each restrictive inequation of the solved linear program is an implicit equality in the system $S \cup \{ex \leq \beta\}$ (see [14]). This result is applied in most of the implementations of CLP languages over linear constraints.

It can be proved that the systems $S \cup \{ex \leq \beta\}$, $S \cup \{ex = \beta\}$ and $S \cup R_=$ are all equivalent (see [14]). Consequently, adding the constraint $ex \leq \beta$ or $ex = \beta$ amounts to fixing the variables appearing in $R_=$ to one of their bounds.

Each algorithm presented in the following considers a regular system S and a constraint $ax_L + a'x_U \leq \beta$ such that the linear program $\min\{ax_L + a'x_U \mid S\}$ is solved and $\sigma_S(ax_L + a'x_U) = \beta$. It returns a regular system S' equivalent to $S \cup \{ax_L + a'x_U \leq \beta\}$. It always detects implicit equalities as restrictive inequations of linear programs that are successively solved. The final system is thus built step by step by replacing each variable fixed to a value by this value.

5 Adaptation of the CLP($\mathcal{R}$) Method

The method described here is an adaptation of the incremental method developed in [16]. It has been integrated to the CLP($\mathcal{R}$) language [10] with the tableau solved form over non-negative variables.

5.1 Method Principles

The basic principle of this method is to consider each implicit equality found as an implicit equality in the solved linear program (5) and to add the corresponding equation in the system. If this

input : S a regular system and $ax_L + a'x_U \leq \beta$ an inequation
output : S' a regular system equivalent to $S \cup \{ax_L + a'x_U \leq \beta\}$
begin
1. let E be the set of restrictive inequations of $\min\{ax_L + a'x_U \mid S\}$
2. **repeat**
   a. remove a constraint $\delta x_i \leq \alpha$ from E and rewrite S to
   $$S \leftarrow \begin{cases} (S - \{l_i \leq x_i \leq u_i\}) \cup \{x_i \leq l_i\} & \text{if } \frac{\alpha}{\delta} = l_i \\ (S - \{l_i \leq x_i \leq u_i\}) \cup \{u_i \leq x_i\} & \text{if } \frac{\alpha}{\delta} = u_i \end{cases}$$
   b. solve the linear program $\gamma = \min\{\delta x_i \mid S\}$,
   c. let R be the new set of restrictive inequations
   d. **if** $\gamma = \alpha$ **then** $E \leftarrow E \cup R$
   e. $S \leftarrow S \cup \{x_i = \frac{\alpha}{\delta}\}$
   until $E = \emptyset$
3. **return** S
end

Figure 1: CLP($\mathcal{R}$) Method

addition involves the apparition of implicit equalities, the new solved linear program that proves this is used to identify a part of them as restrictive inequations. The corresponding equations are added the same way. When no more inequations need to be reintroduced, the final system is regular.

More precisely, for each restrictive inequation $l_i \leq x_i$ found, the equation $x_i = l_i$ is added to the system to obtain a system equivalent to

$$S \cup \{x_i = l_i\}$$

This system is obviously solvable since $\sigma_S(x_i) = l_i$. It is only necessary to decide if it contains implicit equalities. These are searched for in one of the systems $S \cup \{x_i \leq l_i\}$ or $S \cup \{l_i \leq x_i\}$. Since S is regular the constraint $l_i \leq x_i$ is not an implicit equality in S and we have to decide if $x_i \leq l_i$ is an implicit equality in the system

$$S' = (S - \{l_i \leq x_i \leq u_i\}) \cup \{x_i \leq l_i\}$$

For this purpose the linear program $\min\{x_i \mid S'\}$ is solved (the case where the restrictive inequation is $x_i \leq u_i$ is similar).

If one of the inequations $x_i \leq l_i$ or $u_i \leq x_i$ is an implicit equality, the minimization computes a linear program in revised solved form that gives new implicit equalities as restrictive inequations. The whole process is then repeated for each of them.

5.2 Algorithm

The algorithm described in figure 1 begins by computing the set E of restrictive inequations of the linear program given as input (step 1). An inequation $\delta x_i \leq \alpha$ from E is chosen and the bounds on x_i are replaced by the inequation $\alpha \leq \delta x_i$ (step 2a). The expression γx_i is then minimized (step 2b). If $\min\{\delta x_i \mid S\} = \alpha$, the inequation $\delta x_i \leq \alpha$ is an implicit equality and the restrictive inequations of the solved linear program are added to E (step 2d). The variable x_i is then explicitly fixed in the system S (step 2e).

This algorithm has two main advantages for incrementality. Stuckey emphasizes in [16] that the number of steps of the algorithm, and thus the number of linear programs solved, is equal to the number of implicit equalities found. So this method is equivalent to the ideal method. Moreover, the minimization at step 2b for deciding the presence of implicit equalities in the system $S \cup \{\delta x_i \leq \alpha\}$ is initialized with a system whose basic solution satisfies the constraint $\delta x_i = \alpha$. Consequently, few simplex iterations are necessary to prove that $\min\{\delta x_i \mid S\} = \alpha$ or $\min\{\delta x_i \mid S\} < \alpha$. Assuming that there is no degeneracy, a single pivot is sufficient to decide. This is often the case in practice.

6 Adaptation of the `Prolog III` Method

The method presented herein was developed for the linear constraint solver of `Prolog III` [5], with the tableau solved form over non-negative variables. The method is briefly described in [8] ; it has not been detailed nor proven.

When the first implicit equalities found as restrictive inequations are replaced by equations, the new system found may contain more implicit equalities. The `Prolog III` methods identifies some constraints that, once removed from this system, lead assuredly to a regular system. Those constraints are then reintroduced one by one in this system. If during the introduction the presence of new implicit equalities is detected, other constraints are removed and reintroduced the same way later on.

6.1 Definitions and Properties

In the `Prolog III` linear constraint solver, the constraints removed are the equations that contain an occurrence of a variable that also appears in the objective function of the solved linear program proving the presence of implicit equalities.

A more efficient approach is to remove some saturated inequations called bounding inequations. A *saturated inequation* of a system S in revised solved form is an inequation $l_i \leq x_i$ such that $\sigma_S(x_i) = l_i$ or an inequation $x_i \leq u_i$ such that $\sigma_S(x_i) = u_i$.

Definition 1 (Bounding Inequation) *The* bounding inequations *of a solved linear program* $\min\{ax_L + a'x_U \mid S\}$ *are the saturated inequations over variables that are basic in an equation of S and whose right member shares an occurrence of variables with the objective function.*

When restrictive inequations are replaced by equations, the resulting system is not necessarily regular. At this point, the removal of bounding inequations leads with certainty to a different but regular system [14].

Theorem 1 *Let P be a linear program in solved form whose system S is regular. Let R be set of restrictive inequations of P and S_B be its set of bounding inequations. The system $(S - S_B) \cup R_=$ is regular.*

Computing the system S_B requires computing in $\mathcal{T}(S)$ the coefficients of variables that have an occurrence in the objective function. These are obtained by solving the system (3) for each variable x_k that has a non zero coefficient in the objective function.

input : S a regular system and $ax_L + a'x_U \leq \beta$ an inequation
output : S' a regular system equivalent to $S \cup \{ax_L + a'x_U \leq \beta\}$
begin
1. let R be the set of restrictive inequations of $\min\{ax_L + a'x_U \mid S\}$
2. let E be the set of bounding inequations of $\min\{ax_L + a'x_U \mid S\}$
3. $S \leftarrow S \cup R_=$
4. **repeat**
   a. remove a constraint $\delta x_i \leq \alpha$ from E and solve $\gamma = \min\{\delta x_i \mid S\}$
   b. **if** $\gamma = \alpha$
      then $S \leftarrow S \cup \{x_i = \frac{\alpha}{\delta}\}$
      else $S \leftarrow S \cup \{\delta x_i \leq \alpha\}$
   c. let R be the set of restrictive inequations of $\min\{\delta x_i \mid S\}$ solved
   d. let S_B be the set of bounding inequations of $\min\{\delta x_i \mid S\}$ solved
   e. $S \leftarrow (S - S_B) \cup R_=$
   f. $E \leftarrow E \cup S_B$.
   until $E = \emptyset$
4. **return** S
end

Figure 2: `Prolog III` Method

6.2 Algorithm

The algorithm described in figure 2 begins by initializing to empty the set E of restrictive inequations to reintroduce (step 1) and the set S_B of bounding inequations to remove(step 2). It then computes the system $(S - S_B) \cup R_=$ which is regular (step 3). An inequation $\delta x_i \leq \alpha$ is chosen in the set E and is reintroduced in S by minimizing the term δx_i to decide if it is an implicit equality (step 4a). If so, the variable x_i is fixed to the value $\frac{\alpha}{\delta}$, otherwise the inequation is reintroduced in the system (step 4b). Then the algorithm continues until there are no more inequations to reintroduce.

Despite the improvement proposed, this algorithm can remove at each step as many inequations as there are equations. Reintroducing these constraints requires solving the same number of linear programs. This method is thus weakly incremental. However, the basic solution of the system S at step 2e is a solution of the equation $\delta x_j = \alpha$ and, as in the `CLP(`$\mathcal{R}$`)` method, the minimization requires few iterations in practice. Here again, a single non-degenerate pivot is sufficient to prove that the inequation is not an implicit equality.

In conclusion this method seems less efficient than the `CLP(`$\mathcal{R}$`)` method that solves only a linear program for each implicit equality found. Both methods are unable to prove in one step the absence of implicit equalities in the system $S \cup R_=$. This is possible with the quasi-dual method.

7 Adaptation of the Quasi-dual Formulation

The quasi-dual formulation of a system of linear constraints was introduced by Lassez [12]. The use of this formulation for detecting implicit equalities is suggested in [11] among other possible applications. This approach has been integrated to the `CLAIRE` language [2] and to the `Athena` language [15].

Let $S = \{Ax \leq b\}$ be a system of m inequations. The quasi-dual formulation of this system is

the linear program

$$\eta = \min\{yb \mid yA = 0, y \geq 0, \sum y_i = 1\}$$

where y is a row-vector of m variables. If $\eta = 0$ then the system S contains implicit equalities otherwise it does not. With this formulation, the absence of implicit equalities is decided by solving a single linear program. We consider here the dual of the quasi-dual, i.e. the linear program

$$\mu = \max\{x_0 \mid Ax + u.x_0 \leq b, x_0 \geq 0\}$$

where u is a vector of m coefficients, all of them set to 1. If these two linear programs are bounded, the well-known relation $\eta = \mu$ holds. As a consequence, the dual of the quasi-dual can also be solved to detect implicit equalities. This last formulation is more interesting since it involve fewer modifications of the original system S.

In the following, we introduce the (tableau or revised) auxiliary form that is an extension of the dual of the quasi-dual formulation to the (tableau or revised) solved form. The algorithm presented here computes a system in revised auxiliary form from a system in revised solved form by introducing a new variable in some equations. That variable is then minimized. If the optimal value is zero, then the system contains implicit equalities that are identified as restrictive inequations of the solved linear program. Those inequations are replaced by the corresponding equations and the whole process is repeated for the new system to find more implicit equalities or to prove that there are no more.

7.1 Definitions and Properties

Computing the auxiliary form of a tableau solved form S consists of introducing a new variable x_0 in each equation whose basic variable x_b is assigned to one of its bounds in the basic solution. The variable x_0 is not introduced in every constraint as in the dual of the quasi-dual formulation because the basic solution of S can discard a priori some inequations that are not implicit equalities (inequations $l_i \leq x_i$ or $x_i \leq u_i$ such that $l_i < \sigma_S(x_i)$ or $\sigma_S(x_i) < u_i$).

To clarify this presentation, the adaptation is first presented with the tableau solved form and then with the revised one.

Definition 2 (Tableau Auxiliary Form) *Let S be a system in tableau solved form (1). The* tableau auxiliary form *of S is the system*

$$S_A = \begin{cases} x_B = w.x_0 + Mx_L + M'x_U + b \\ l \leq x \leq u \\ 0 \leq x_0 \leq +\infty \end{cases}$$

where $\sigma_{S_A}(x_0) = 0$ and w is a vector such that

$$w_i = \begin{cases} 0 & \text{if } l_i < \sigma_S(x_i) < u_i \\ 1 & \text{if } \sigma_S(x_i) = l_i \\ -1 & \text{if } \sigma_S(x_i) = u_i \end{cases}$$

Note that S_A is in revised solved form since its basic solution corresponds to the basic solution of S on all variables except on x_0 which is assigned to zero.

input : S a regular system and $ax_L + a'x_U \leq \beta$ an inequation
output : S' a regular system equivalent to $S \cup \{ax_L + a'x_U \leq \beta\}$
begin
1. let E be the set of restrictive inequations of $\min\{ax_L + a'x_U \mid S\}$
2. **repeat**
   a. $S \leftarrow S \cup E_=$
   b. let S_A be the revised auxiliary system of S and x_0 be the var. introduced.
   c. **if** $\min\{-x_0 \mid S_A\} < 0$
      then return S.
      else let E be the set of restrictive inequations of $\min\{-x_0 \mid S_A\}$ solved.
   until false
end

Figure 3: Quasi Dual Method

Definition 3 (Revised Auxiliary Form) *Let S' be a system in revised solved form (2). The* revised auxiliary form *of S' is*

$$S'_A = \begin{cases} Ax_B + w'.x_0 + A'x_L + A''x_U = 0 \\ l \leq x \leq u \\ 0 \leq x_0 \leq +\infty \end{cases}$$

where x_0 is a variable assigned to zero in the basic solution of S'_A and w' is a vector such that $w' = -A_B w$ where w is the vector from definition 2.

Assume that $S = \mathcal{T}(S')$ then $S_A = T(S'_A)$. The vector w' is thus the appropriate vector to introduce in S'_A to verify this relation. The solving of a single linear program can determine the presence or the absence of implicit equalities in the system S'.

Theorem 2 *With notations of definition 2, the system S' contains implicit equalities if and only if* $\min\{-x_0 \mid S'_A\} = 0$.

This result leads to the incremental algorithm of figure 3.

7.2 Algorithm

This algorithm is rather simple. It identifies first the restrictive inequations of the solved linear program (step 1). It fixes the variables appearing in those inequations (step 2a). The auxiliary revised form is computed (step 2b) and the linear program of theorem 2 is solved with the simplex method (step 2c). If the optimal value is strictly negative, the system S is regular. Otherwise new restrictive inequalities are found and the algorithm continues. Note that the iterations of the simplex can be stopped as soon as the term $-x_0$ can be assigned to a strictly negative value for the basic solution.

In comparison with the previous methods, the quasi-dual method has many advantages. The computation of the auxiliary form is simple and proving the absence of implicit equalities requires solving a single linear program. In practice this proof needs few simplex iterations since the term $-x_0$ is already assigned to zero in the basic solution of the auxiliary forms.

		Quasi-dual meth.				Prolog III meth.				CLP($\mathcal{R}$) meth.			
Problems	Nb	Time	Piv	Step	IEPiv	Time	Piv	Step	IEPiv	Time	Piv	Step	IEPiv
Square 9	933	4360	2826	607	2826	4850	2756	174	2756	4680	2618	410	2618
Square 14	4293	42930	18485	2939	18485	47120	17631	861	17631	45610	16886	1435	16886
Transport	80	190	125	1	0	260	125	0	125	280	125	80	0
Donald	46	780	179	52	171	800	145	28	137	770	127	46	119
Periodic	39	110	96	46	96	130	72	17	72	120	67	23	67
Cut-stock	245	5000	909	207	887	5020	693	124	671	4650	587	245	565

Figure 4: Comparison of all Methods

8 Practical Results

The three approaches above have been integrated to the CLP language Athena. The linear constraint solver of Athena is based on a revised simplex method on exact-precision rationals. It includes advanced linear programming techniques and an efficient procedure for dynamic backtracking [15].

Various examples have been tested with the three approaches. Most of these examples are typical CLP problems over linear constraints that need to detect implicit equalities for handling disequations.

The Donald problem is the well-known cryptarithm problem [5]. Periodic solves the problem of proving that a mathematical sequence is periodic [5]. It contains a single disequation, however finding implicit equalities reduces the search space. The Square problem [5] is a two dimensional placement problem that is very combinatorial. It consists of filling a rectangle with squares of different sizes such that no squares overlaps. All pivoting operations done to solve this problem are for detecting implicit equalities. Two instances of this problem are solved: one with 9 squares, the other with 14 squares. Transport [19] is a deterministic problem of assigning providers to customers. At the optimum value all of the 80 variables are fixed showing that there is no other optimal solution. Cut-stock is a cutting stock problem solved to optimality with a branch and bound approach [19]. All these problems are non deterministic except for Transport.

Figure 4 shows the number of implicit equalities (**Nb**) to be found or each problem. For each method, it shows the computation time[2] to solve the problem (**Time**), the total number of pivotings required (**Piv**), the number of steps of the algorithm (**Step**) which equals the number of linear programs solved to detect implicit equalities, and the number of pivoting needed to solve those linear programs (**IEPiv**).

Results in this table show that no method is clearly superior to the others. The quasi-dual method seemed to be the best one and to be well suited to the revised simplex, but it is not far better that the others. On average it is 12% faster than the Prolog III method and 10% faster than the CLP($\mathcal{R}$) method. However, it can do many more iterations than the others, since fewer implicit equalities are discovered at each step. The exception is the Transport problem where the implicit equalities are all found at one time as restrictive inequations and the quasi-dual method proves that there are no more of them in only one step.

The Prolog III method performs much less iteration than the other methods. The advantage is negated by the need to compute columns of the tableau solved form to determine bounding inequalities. This result is surprising since it is the worst method in terms of complexity.

The CLP($\mathcal{R}$) method is slightly faster than the Prolog III methods. Note that it does not

[2] In milliseconds on a Sun Sparc 10

perform as many iterations as there are implicit equalities because transforming a restrictive inequation into an equation sometimes leads to a failure and consequently no linear program is solved for the remaining restrictive inequations. The performance of this method lies between the `Prolog III` and the quasi-dual methods.

9 Conclusion

This paper has described how to detect implicit equalities with the revised simplex algorithm. This algorithm is more efficient and more suitable to practical problems than the tableau method that is usually applied in CLP. We have described the adaptation to this simplex of three main approaches for the incremental detection of implicit equalities: the `CLP(`$\mathcal{R}$`)` approach, the `Prolog III` approach and the quasi-dual one.

A practical comparison with the CLP language `Athena` was done on some CLP problems. It shows that all these methods are almost equivalent from the efficiency standpoint. The quasi-dual method, which was considered the best method for the revised simplex, is not much faster that the others while the `Prolog III` method, which was considered computationally too costly, is the one that performs the least steps.

As a future study it would be interesting to combine the `Prolog III` and the quasi-dual methods in order to use the quick detection of the absence of implicit equalities of the quasi-dual method and the few number of iterations of the `Prolog III` method.

References

[1] F. Benhamou and Touraïvane. Prolog IV: langage et algorithmes. In *JFPL'95: IVèmes Journées Francophones de Programmation en Logique*, pages 51–65, Dijon, France, 1995. Teknea.

[2] H. Beringer and B. de Backer. Combinatorial problem solving in constraint logic programming with cooperating solvers. In *Logic Programming : Formal Methods and Practical Applications*. Elsevier Science Publishers, 1994.

[3] D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, Belmont, Massachusetts, 1997.

[4] V. Chvatal. *Linear Programming*. W.H. Freeman and Company, New York, 1983.

[5] A. Colmerauer. An introduction to Prolog III. *Communications of the ACM*, 33(7):69–91, July 1990.

[6] A. Colmerauer. Naive resolution of non-linear constraints. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 89–112. MIT Press, 1993.

[7] A. Colmerauer. Spécifications de Prolog IV. Technical report, Laboratoire d'Informatique de Marseille, 1996.

[8] M. Henrion. Les algorithmes numériques de Prolog III. Technical report, Prologia, 1989.

[9] J.-L. Imbert and P. van Hentenryck. On the handling of disequations in CLP over linear rational arithmetic. In F. Benhamou and A. Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 49–72. MIT Press, 1993.

[10] J. Jaffar, S. Michaylov, P. Stuckey, and R. Yap. The CLP($\mathcal{R}$) language and system. *Transactions on Programming Languages and Systems*, 14(3), July 1992.

[11] J.-L. Lassez. Parametric queries, linear constraints and variable elimination. In *DISCO 90, LNCS 429*, pages 164–173. Springer-Verlag Lecture Notes in Computer Science, 1990.

[12] J.-L. Lassez. Querying constraints. In *Proceedings of the ACM Conference on Principles of Database Systems*, Nashville, 1990.

[13] J.-L. Lassez and K. McAloon. A canonical form for generalised linear constraints. *Journal of Symbolic Computation*, (1):1–24, 1992.

[14] P. Refalo. *Resolution et implication de contraintes lineaires en programmation logique par constraintes.* Ph.D Thesis. Laboratoire d'informatique de Marseille, Marseille, 1997.

[15] P. Refalo and P. van Hentenryck. CLP($\mathcal{R}_{lin}$) revised. In *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 1–14, Bonn, Germany, 1996.

[16] P. J. Stuckey. Incremental linear constraint solving and implicit equalities. *ORSA Journal of Computing*, 3(4):269–274, 1991.

[17] J. Telgen. *Redundancy and Linear Programs.* Mathematical Centre Tracts, 1981. number 137.

[18] P. van Hentenryck and T. Graf. Standard forms for rational linear arithmetics in constraint logic programming. In *The International Symposium on Artificial Intelligence and Mathematics*, Fort Lauderdale, Florida, Jan. 1990.

[19] P. van Hentenryck and V. Ramachandran. Backtracking without trailling in CLP($\mathcal{R}_{lin}$). *ACM Transaction on Programming Langages and Systems*, 1(1):0–0, 1995.

Parallel Arc-Consistency for Functional Constraints

A. Ruiz-Andino †, L. Araujo †, F. Sáenz ‡, J. Ruz ‡
† Department of Computer Science
‡ Department of Computer Architecture
University Complutense of Madrid
arai@eucmax.sim.ucm.es, lurdes@eucmos.sim.ucm.es
fernan@eucmax.sim.ucm.es, jjruz@eucmax.sim.ucm.es
http://mozart.sip.ucm.es/ alvaro

Abstract

We present in this paper a parallel execution model of arc-consistency for Constraint Satisfaction Problems (CSP), implemented on a scaleable MIMD distributed memory machine. We have adopted the *indexical scheme*, an adequate approach to arc-consistency for functional constraints. The CSP is partitioned into N partitions, which are executed in parallel on N processors. Each processor applies sequential arc-consistency to its subset of constraints, updating remote domains of variables shared by any other processor. The parallel algorithm we propose is embedded in the constraint solver PCSOS (Parallel Constraint Satisfaction and Optimisation System). PCSOS invokes the parallel arc-consistency algorithm when performing labelling to obtain the solutions of a CSP. PCSOS is written in C, and it has been developed and tested on a CRAY T3E distributed memory multiprocessor with up to twenty-six processors. Results on speedup and behaviour of the system are reported for the search of the first solution of different CSPs.

1 Introduction

Constraint Programming over finite domains (CP(FD)) has been used for specifying and solving complex constraint satisfaction and optimisation problems, as resource allocation, scheduling and hardware design [6, 17]. Thanks to their combination of constraint solving and declarative style, Constraint Programming languages are able to find solutions to combinatorial problems comparable in efficiency to procedural languages, and yet requiring a much shorter development time. Finite domain Constraint Satisfaction Problems (CSP) usually describe NP-complete search problems. Much effort has been done to develop techniques in order to find solutions in polynomial time. It has been shown that by working locally on constraints and their related variables it is possible to dynamically prune the search space in an efficient way. Techniques following this approach are called arc-consistency algorithms. Waltz [18] proposed the first arc-consistency algorithm, and several improved versions are described in the literature [11, 16, 15]. Consistency-based constraint solvers eliminate inconsistent values from the solution space, reducing the size of the search space both before and while searching. Most CP(FD) languages are based on a constraint solver which integrates a search procedure with an arc-consistency technique.

A constraint satisfaction problem over finite domains may be stated as follows. Given a tuple $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$, where

- $\mathcal{V} \equiv \{v_1, \cdots, v_n\}$, is a set of domain variables,

- $\mathcal{D} \equiv \{d_1, \cdots, d_n\}$, is the set of an initial *finite domain* (finite set of values) for each variable,
- $\mathcal{C} \equiv \{c_1, \cdots, c_m\}$, is a set of constraints among the variables in $\mathcal{V}$. A constraint $c \equiv (V_c, R_c)$ is defined by a subset of variables $V_c \subseteq \mathcal{V}$, and a subset of allowed tuples of values $R_c \subseteq \bigotimes_{i \in \{j / v_j \in V_c\}} d_i$, where $\bigotimes$ denotes cartesian product.

The goal is to find an assignment for each variable $v_i \in \mathcal{V}$ of a value from each $d_i \in \mathcal{D}$ which satisfies every constraint $c_i \in \mathcal{C}$.

A constraint $c \equiv (V_c, R_c) \in \mathcal{C}$, $V_c \equiv \{v_1, \cdots, v_n\}$, is arc-consistent with respect to domains $\{d_1, \cdots, d_n\}$ iff for all $v_i \in V_c$, for all $a \in d_i$, there exists a tuple $(b_1, \cdots, b_{i-1}, a, b_{i+1}, \cdots, b_n) \in R_c$, where $b_j \in d_j$. A CSP is called arc-consistent iff all $c_i \in \mathcal{C}$ are arc-consistent with respect to $\mathcal{D}$.

In this paper, we will focus on the exploitation of parallelism available on arc-consistency propagation. The parallel algorithm we propose is the core of the system PCSOS [13] (Parallel Constraint Satisfaction and Optimisation System). Besides arc-consistency parallelism, PCSOS may also exploit parallelism in search and optimisation procedures.

Several parallel processing methods for solving CSPs have been proposed. In [7, 8] authors describe logic gate level arc-consistency parallel algorithms, extending similar approaches of [14] and [4]. In [5] two hardware-level parallel arc-consistency algorithms are proposed. The first one is expressed as a massively parallel digital circuit requiring a large number of very simple processing elements. The second one is more suited to be implemented on a highly parallel SIMD machine. In [20], a parallel constraint solving technique for a special class of CSP, acyclic constraint networks, is developed. It also presents some results on parallel complexity, generalising results in [9]. In [10], it is concluded that parallel complexity of constraint networks is critically dependent on subtle properties of the network that do not influence its sequential complexity. They propose massively parallel processing of arc-consistency with also very simple processing elements.

Nguyen, Deville and Baudot proposed in [2, 12] a similar approach to ours, but for binary CSPs, discussing distributed versions of AC-3, AC-4, and AC-6. Our work is focused on n-ary functional constraints (AC-5), and we report empirical data obtained running the parallel arc-consistency algorithm on a CRAY T3E distributed memory multiprocessor, performing a search for the first solution to CSPs. The parallel execution mode is based on the partition of the CSP into N sub-CSPs, which are executed in parallel on N processors. Each processor applies sequential arc-consistency to its subset of constraints, updating remote domains of variables shared by any other processor. We also discuss the main issues affecting the performance of the model, like the criteria to distribute constraints are among processors, and the frequency of updating shared variables.

The rest of the paper is organised as follows. Next section describes the sequential constraint arc-consistency algorithm, based on the *indexical* scheme. Section 3 presents the model developed to parallelise constraint arc-consistency, the parallel algorithm, and a number of issues affecting performance. Section 4 presents and discusses the experimental results. Finally, conclusions are drawn in section 5.

2 Constraint Propagation

Our starting point is a sequential constraint propagation algorithm which implements arc-consistency using the *indexical scheme* [3]. In this scheme, a constraint is translated into a set of reactive functional expressions, called *indexicals*, which maintain arc-consistency. An indexical has the form "v in $E(V)$", where $v \in \mathcal{V}$, $V \subseteq \mathcal{V}$, and $E(V)$ is a monotonic functional expression which returns a finite set of values. Given an indexical $I \equiv v$ in $E(V)$, we call V its *set of arguments*, and we say that, for all $v_i \in V$, I *depends on* v_i, and I *writes* the domain variable v.

A constraint $c \equiv (V_c, R_c)$ relating the set of domain variables $V_c \equiv \{v_1, \cdots, v_n\}$, is translated into a set of n indexicals $\{I_i \equiv v_i \text{ in } E_i(V_c - \{v_i\})\}_{1 \leq i \leq n}$. Each indexical I_i writes variable v_i and depends on the remaining $n-1$ variables. Functional expressions $E_i(V_c - \{v_i\})$ are properly defined for arc-consistency to be achieved (removal of inconsistent values) with respect to constraint c.

The set of finite domains which keeps the current domain of each variable in $\mathcal{V}$ is called the *store*. The initial value of the store is defined by $\mathcal{D}$. The execution of an indexical is triggered by changes in the domains of its set of arguments V in a data driven way. When an indexical is executed, the domain of v in the store is updated with $v \cap Eval(E(V))$, where $Eval(E(V))$ denotes the evaluation of $E(V)$ with the current domains of the set of variables V in the store.

For example, the arithmetic constraint $v_1 = v_2 + 4$ over finite integer domains is translated into two indexicals:

$I_1 \equiv v_1$ in $min(v_2) + 4$ to $max(v_2) + 4$;
$I_2 \equiv v_2$ in $min(v_1) - 4$ to $max(v_1) - 4$.

Whenever the lower or upper bound of v_2 ($min(v_2), max(v_2)$, resp.) changes, indexical I_1 removes inconsistent values from v_1's domain. Indexical I_2 behaves similarly. For instance, let $d_1 = d_2 = \{1, \cdots, 10\}$ be the initial domains for v_1 and v_2. I_1 prunes v_1's domain to $\{5, \cdots, 10\}$ and I_2 prunes v_2's domain to $\{1, \cdots, 6\}$. If some other indexical writing v_2 prunes d_2 to $\{4, 5\}$, then I_1 will prune d_1 to $\{8, 9\}$. Moreover, this modification will be *propagated* to other domains because of the execution of those indexicals which depend on v_1.

```
     function Arc-Consistent-CSP(⟨VarSet, DomSet, ConstrSet⟩):Store
     begin
 1     Store_Reset(Store, VarSet, DomSet);
 2     Queue_Reset(PropagationQueue);
 3     for each indexical I_i ∈ ConstrSet do
 4       if NOT Arc_Consistent(I_i,Store,PropagationQueue) then
 5         return FAILURE;
 6       end-if;
 7     end-for;
 8     while NOT Empty(PropagationQueue) do
 9       Queue_Pop(PropagationQueue, DomVar);
10       for each indexical I_i which depends on DomVar do
11         if NOT Arc_Consistent(I_i,Store,PropagationQueue) then
12           return FAILURE;
13         end-if;
14       end-for;
15     end-while;
16     return Store;
     end;
```

Figure 1: Arc consistency algorithm.

Figures 1 and 2 show the sequential arc-consistency propagation algorithm. Its input argument is the CSP $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ whose arc-consistency is to be achieved. The set of constraints $\mathcal{C}$ is expressed as a set of indexicals. The algorithm returns either a store where the domain for each variable has been pruned achieving arc-consistency, or FAILURE if inconsistency is detected (the domain of a

variable was pruned to an empty domain).

The arc-consistency algorithm executes indexicals until either the fixed point is reached, or inconsistency is detected. The fixed point is reached iff the store is arc-consistent. A propagation queue is used to schedule the execution of indexicals (`PropagationQueue`, Figure 1). As the result of the execution of an indexical (`Arc_Consistent()`), the domain of a variable may be pruned, and in such a case the variable is queued (`Update()`). Initially, all indexicals are executed (lines 3 to 7). The main loop (lines 8 to 15) iterates until either the propagation queue is empty, or inconsistency is detected. In each iteration, a variable is unqueued and those indexicals which depend on it are executed. Termination, correctness, complexity, and properties of the algorithm have been studied extensively in the literature [16]. Correctness is independent of the order of reexecution of indexicals, which constitutes the basis for the correctness of the parallel version of the algorithm.

```
    function Arc_Consistent( 'DomVar in E()',
                        Var Store, Var PropagationQueue ) :  BOOLEAN
    begin
1     NewDomain := Eval(E(), Store);
2     return (Update(NewDomain,DomVar,Store,PropagationQueue)<>EMPTY);
    end;

    function Update( NewDomain, DomVar,
                        Var Store, Var PropagationQueue):  RESULT
    begin
1     NewDomain := NewDomain ∩ Store[DomVar];
2     if Empty(NewDomain) then return EMPTY; end-if;
3     if (NewDomain ⊂ Store[DomVar]) then
4       Store[DomVar] := NewDomain;
5       Queue_Push(DomVar, PropagationQueue);
6       return PRUNED;
7     end-if;
8     return NOT_PRUNED;
    end;
```

Figure 2: Store and propagation queue updating.

3 Parallel Constraint Propagation

Arc-consistency algorithms in general, and the indexical scheme in particular, have an inherent parallelism. Each indexical behaves as a concurrent process which updates the store, triggered by changes in the store. There is an inherent sequentiality, as well, since an indexical I_i may be executed only as the consequence of a previous execution of another indexical I_j that writes a variable which I_i depends on. An indexical is *ready* if any of its arguments have changed after its last execution. At any time during the execution of the arc-consistency algorithm there will be a set of ready indexicals, called the *ready set*. In the sequential version of the algorithm, the ready set is formed by those indexicals which depend on the variables in the propagation queue. The basis

of our approach is to execute in parallel those indexicals in the ready set. Therefore, the parallel computational unit is the execution of an indexical.

The first decision to be made in order to define the parallel execution model is whether sharing or distributing the store and the set of indexicals. A straightforward implementation of the indexical execution model on a massively parallel processor would map a single indexical onto a single processor, sharing the store. Unfortunately, processor utilisation rate would be very low and communication volume would be very high. Since the current design is devoted to run on a distributed memory multiprocessor, our model is based on distributing the store and the set of indexicals, partitioning the CSP. Moreover, in order to reduce communication overhead, our model is based on a static partition of the CSP, instead of a dynamic one.

The set of constraints $\mathcal{C}$ is partitioned into n disjoint subsets of constraints, $\mathcal{C} = C_1 \cup \cdots \cup C_n$. This partitioning induces a distribution of the set of domain variables $\mathcal{V}$ in n not necessarily disjoint subsets $V_1, \cdots, V_n$ ($\mathcal{V} = V_1 \cup \cdots \cup V_n$). For all $I_j \in C_i$, the variable written by I_j, and those variables on which I_j depends on, constitute V_i ($\forall I_j \in C_i, I_j \equiv v_{I_j}$ in $E(V_{I_j}), V_i = \{v_{I_j}\} \cup V_{I_j}$.) Figure 3 sketches the partitioning process of the CSP.

Partitions $\langle V_i, D_i, C_i \rangle$ are mapped one-to-one to processing elements P_i. Each processing element P_i performs sequential arc-consistency, executing those indexicals in C_i, and consequently updating local copies of some variables in V_i. Since V_i are non-disjoint, some variables will be located at several processing elements. Therefore, each processing element P_i must broadcast any pruning on the domain of variable v to every processing element P_j which had been assigned any of those indexicals which depend on v. Upon receiving the notification, processing elements P_j intersect their local copies of the domain with the incoming domain, probably triggering further propagation. Communication among processors is also needed in order to detect termination of the algorithm, either because of reaching the global fixed point, or detecting inconsistency.

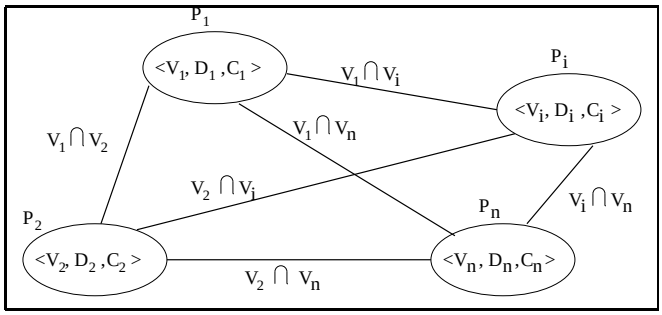

Figure 3: Partitioning the CSP. Sub-CSP $\langle V_i, D_i, C_i \rangle$ is assigned to processing element (PE) P_i. An edge between two PEs is labelled with the set of variables located at both PEs ($V_i \cap V_j$). Communication is needed to maintain the same domain for some of the variables in $V_i \cap V_j$.

3.1 Parallel Algorithm

Figures 4 and 5 show the parallel algorithm. Like the sequential algorithm, initially every indexical assigned to the processor is executed (lines 5 to 10). The main loop (lines 11 to 33) is executed until either global fixed point (`GlobalFixedPoint`) or inconsistency (`Failure`) is detected. The latter can be caused by one of the following:

- an empty domain resulting from the execution of a local indexical (`Local_Arc_Consistent()`).

- an empty domain resulting from the intersection of the local domain of a variable with the domain received from another processor (`Remote_Arc_Consistent()`).
- inconsistency, detected at (and broadcasted from) another processor (`RemoteFailure`).

Each processor maintains a private propagation queue (`LocalPropQueue`). The inner loop (lines 12 to 20) performs local propagation until either the queue is empty or inconsistency is detected, like the main loop of the sequential algorithm. Once a local fixed point is reached, the processor notifies to a *distinguished* processor of this status (`Notify_Local_Fixed_Point()`), and it waits (lines 23 to 28) until one of the following occurs:

- global fixed point is detected (`Check_Global_Fixed_Point()`).
- some other processor communicates inconsistency (`RemoteFailure`).
- the processor receives a message which updates its local propagation queue. In this case, the processor notifies it to the distinguished one, (`Notify_Active()`), and continues performing propagation.

When the local execution of an indexical (`Local_Arc_Consistent()`, Figure 5) results in the modification of the domain of a variable v (`Update()`, Figure 2), the processor broadcasts a message (`Broadcast_Update()`) to the set of processors that have been assigned any of those indexicals which depends on variable v. Upon receiving the message (`Remote_Arc_Consistent()`, Figure 5), these processors either detect inconsistency or properly update their local propagation queue and their local copy of variable v. Whenever a processor detects inconsistency, it broadcast the failure to the rest of processors (`Broadcast_Failure()`).

The algorithm terminates when every processor reaches a local fixed point and there are no messages pending on the communication network. The distinguished processor is the only responsible for the detection of termination. However, it performs local propagation as any other processor. In order to be able to detect the global fixed point, processors must notify to the distinguished one of reaching a local fixed point –along with the number of messages they have sent and received– (`Notify_Local_Fixed_Point()`), and of leaving it due to an incoming message (`Notify_Active()`). The distinguished processor keeps record of how many processors are at a local fixed point, and the number of messages sent and received by all processors. When termination is detected, the distinguished processor notifies it to the rest of processors (`GlobalFixedPoint`).

The parallel arc-consistency algorithm will usually be repeatedly invoked from a sequential labelling procedure, or any other search procedure which needs to achieve arc-consistency. Therefore, synchronisation among all processors is needed both at the beginning and at the end of the algorithm. For this purpose, after the initialisation of the communication status variables (`ParState_Reset()`, line 1, Figure 4) and before starting propagation, a `barrier()` primitive is used. Another barrier (line 35) is needed if the algorithm finishes with failure; otherwise, the global fixed point detection implies a synchronisation among processors.

3.2 Tuning the parallel solver

There are a number of issues to be specified in the parallel execution model affecting performance. We have tried different settings, analysing how they affect work load, communications volume, and overall relevance. Mainly, there is a trade-off between two main factors affecting performance:

- Run-time size and balanced distribution of the ready set.

```
     function Parallel-Arc-Consistent-CSP(
                    ⟨VarSubSet, DomSubSet, ConstrSubSet⟩ ) :  Store
     begin
 1     ParState_Reset();
 2     barrier();
 3     Store_Reset(Store, VarSubSet, DomSubSet);
 4     Queue_Reset(LocalPropQueue);
 5     for each indexical I_i ∈ ConstrSubSet do
 6       Failure := RemoteFailure OR
 7             NOT Local_Arc_Consistent(I_i,Store,LocalPropQueue) OR
 8             NOT Remote_Arc_Consistent(Store,LocalPropQueue);
 9       if Failure then break; end-if;
10     end-for;
11     while NOT Failure AND NOT GlobalFixedPoint do
12       while NOT Failure AND NOT Empty(LocalPropQueue)) do
13         Queue_Pop (LocalPropQueue, DomVar);
14         for each indexical I_i which depends on DomVar do
15           Failure := RemoteFailure OR
16                 NOT Local_Arc_Consistent(I_i,Store,LocalPropQueue) OR
17                 NOT Remote_Arc_Consistent(Store,LocalPropQueue);
18           if Failure then break; end-if;
19         end-for;
20       end-while;
21       if NOT Failure then
22         Notify_Local_Fixed_Point(···);
23         repeat
24           Failure := RemoteFailure OR
25                 NOT Arc_Consistency_Msg(Store,LocalPropQueue);
26           GlobalFixedPoint := Check_Global_Fixed_Point();
27         until Failure OR GlobalFixedPoint OR
28                   NOT Empty(LocalPropQueue);
29         if NOT Empty(LocalPropQueue) then
30           Notify_Active();
31         end-if;
32       end-if;
33     end-while;
34     if Failure then
35       barrier();
36       return FAILURE;
37     end-if;
38     return Store;
     end;
```

Figure 4: Parallel arc-consistency algorithm.

```
    function Local_Arc_Consistent( 'DomVar in E()',
                            Var Store, Var LocalPropQueue ):  Boolean
    begin
1     NewDomain := Eval(E(), Store);
2     switch (Update (NewDomain, DomVar, Store, LocalPropQueue))
3       case EMPTY :
4         Broadcast_Failure(RemoteFailure);
5         return FALSE;
6       case PRUNED:
7         Broadcast_Update(DomVar, Store[DomVar]);
8     end-switch;
9     return TRUE;
    end;

    function Remote_Arc_Consistent( Var Store,
                            Var LocalPropQueue ) :  Boolean
    begin
1     while NOT Empty(MsgQueue) do
2       Pop_Message(MsgQueue, DomVar, NewDomain);
3       if (Update(NewDomain,DomVar,Store, LocalPropQueue) = EMPTY) then
4         Broadcast_Failure(RemoteFailure);
5         return FALSE;
6       end-if;
7     end-while;
8     return TRUE;
    end;
```

Figure 5: Parallel arc-consistency functions.

- Volume of communications.

A higher availability of updated domains results in a larger ready set. In addition, the more spread among processing elements indexicals are, the better balanced the ready set is. However, in order to achieve higher updated domains availability, more frequent communications are required. Similarly, spreading indexicals requires broadcasting updated information to more processing elements. Therefore, the advantage of working with a large and well balanced ready set is weakened by the effect of a higher communications volume.

The introduced parallel execution model offers the possibility to control the trade-off between communications and ready set quality through the definition of, at least, two issues: partitioning of the CSP, and frequency of broadcasting/processing domain updates.

3.2.1 Partitioning the CSP.

A key factor for the efficiency of the parallel algorithm has shown to be the way the set of constraints is partitioned. A CSP $\langle \mathcal{V}, \mathcal{D}, \mathcal{C} \rangle$ can be represented as a hyper-graph where the set of nodes is

the set of domain variables $\mathcal{V}$ and the set of hyper-edges is the set of indexicals defined by $\mathcal{C}$. Therefore, partitioning the CSP among processors means partitioning the set of hyper-edges in disjoints subsets, inducing a not necessarily disjoint partitioning of the set of nodes. We have tested two different graph partition criteria:

- Strength of connection between partitions.
- Static estimation of run-time ready set distribution.

Strength of connection between partitions.

The graph connectivity may be considered in order to partition the graph in either *strongly connected subgraphs*, or *highly disconnected subgraphs*.

In the former case, communications are minimised, but the ready set will be, in general, badly balanced. A strongly connected partitioning induces an almost disjoint partitioning of the set of variables $\mathcal{V}$, thus avoiding communications. However, it is very likely that most of those indexicals which depend on a variable v are assigned to the same processing element P. Whenever variable v is pruned, the ready set is enlarged with those indexicals which depend on v, but almost all of them will be sequentially executed by P, thus loosing the potential parallelism exploitation.

In the latter case, the ready set is better balanced, but it is likely that almost every variable to be located at almost every processing element, increasing communications.

Experimental results show the benefit of a better balanced ready set versus a communications reduction. Moreover, partitioning the CSP in strongly connected subgraphs is a hard problem, whereas a highly disconnected CSP partitioning is easily achieved with a shuffle distribution of indexicals.

Static estimation of run-time ready set distribution

A constraint partitioning which balances the run-time ready set is expected to improve the performance, providing that communications do not increase. Since our approach is based on a static partitioning of the CSP among processors, balancing run-time ready set requires some kind of compile-time estimation.

Our approach to this estimation is to partition the constraint set in such a way that updating any variable amounts a similar number of indexicals to be executed by each processor [13]. We have defined an objective function, to be minimised, which considers the peak work load for each processor and variable. Experimental run-time work load measures have confirmed the accuracy of our static estimation. Since we are dealing with n-ary constraints, finding the optimal solution is a NP problem. Therefore, we recourse to an algorithm that assigns constraints one by one, in a decreasing arity order, greedily choosing the processor which minimises the objective function. Solutions found with this greedy algorithm have shown to be quite close to the optimal one when the CSP is constituted by a large number of low-arity constraints. Taking into account that this is just an estimation of the actual run-time ready set distribution, the greedy approach is fully justified.

3.2.2 Frequency of broadcasting/processing domain updates

There is a trade-off between quick transmission of updated information, which increases the size of the ready set, and communications overhead, which decreases performance. We have tested two

settings for the frequency of broadcasting domain updates:

- Immediate broadcast: domain updates are broadcasted as soon as they are locally produced.
- Fixed point broadcast: domain updates are locally queued and broadcasted whenever the local fix point is reached.

Similarly, the frequency of checking and processing incoming domains affects updated information availability. Checking for received updated domains each time an indexical is executed has shown to produce best results. In this way, processors are working with the best available updated information, while overhead due to checking for messages is negligible versus the execution time of an indexical.

4 Experimental Results

PCSOS is written in C, and it has been developed and tested on a CRAY T3E multiprocessor with up to twenty-six 400-MHz DEC Alpha processors, under UNICOS (UNIX) operating system. CRAY T3E is a scaleable parallel system which reduces latency providing sustained bandwidth. Processing elements in the T3E are connected by a bi-directional 3-D torus network achieving communication rates of 480 Mbytes/s. Parallel programming capabilities are extended through the Cray Shared Memory Library, which allows fast direct asynchronous access to remote memory.

Notification of local fixed point, activity, messages sent and received, and inconsistency detection has been implemented using the fast remote memory write feature of the CRAY T3E architecture. Queues of messages are used for receiving domain updates. Messages are broadcasted to queues also using the fast remote memory write feature. Variables used for global fixed point detection, located at the distinguished processor, must be accessed in mutual exclusion.

Reported results correspond to the search for the first solution, performing a first fail sequential labelling. Therefore, reported speedup is lower than speedup achieved in a single propagation cycle, since it comprises both phases, variable/value selection, executed sequentially, and constraint propagation, executed in parallel. This way to measure is more significant than measuring a single propagation cycle, since constraint propagation is usually embedded in some kind of a search procedure.

We have tested PCSOS on a set of benchmarks. Results obtained for two of them, *Equations* and *Suudoku* are reported. *Equations* is a synthetic benchmark. It is formed by sixteen blocks $\{B_1, \cdots, B_{16}\}$ of arithmetic equations. Each block contains fifteen equations among six variables. Blocks B_i, B_{i+1} are connected by an additional equation between a pair of variables, one from B_i and the other one from B_{i+1}. Coefficients were randomly generated. The total number of indexicals is 1,626. Sequential search for the first solution performs 15,969 calls to parallel Arc-Consistent-CSP, 12,438 of them detecting inconsistency. Execution time with just one processor on the CRAY T3E is 16.2 sec. Sequential execution time on a SPARC workstation is 30.40 sec., versus 26.48 sec. obtained with Sicstus Prolog 3.6. *Suudoku* is a crypto-arithmetic Japanese problem. Given a grid of 25x25 squares, where 317 of them are filled with a number between 1 and 25, fill the rest of squares such that each row and column is a permutation of numbers 1 to 25. Furthermore, each of the twenty-five 5x5 squares starting in columns (rows) 1, 6, 11, 16, 21 must also be a permutation of numbers 1 to 25. There are a total number of 28,193 indexicals. First solution is found after 72,196 calls to Arc-Consistent-CSP, 36,092 of them detecting inconsistency. Execution time with just one processor on the CRAY T3E is 152.1 sec. Sequential execution time on a SPARC workstation is 236.85 sec., versus 207.05 sec. obtained with Sicstus Prolog 3.6.

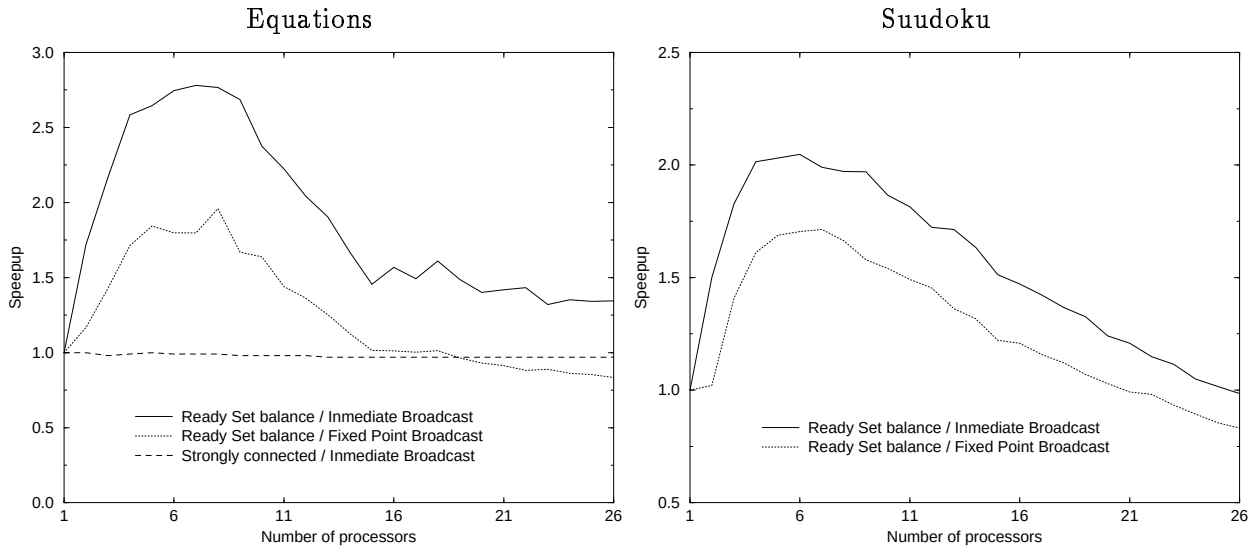

Figure 6: Speedup curves for selected benchmarks.

Charts in Figure 6 show, for each benchmark, the speedup vs. the number of processors. The ready set balance estimation was used, comparing broadcast frequency: immediate (solid line) vs. fixed point (dotted line). Chart for the *Equations* benchmark also shows the speedup obtained with a strongly connected graph partitioning (dashed line). This criteria has not been considered for the other benchmark, since it clearly provides worse results than ready set balance, and because it is too computationally expensive to apply.

It must be noticed that, for both benchmarks, speedup decreases as more processors are added beyond a problem dependent optimum number of processors. This behaviour has also been observed on the paralelisation of event-driven logic simulation [1] and other iterative algorithms. Figures 7, 8, 9 and 10 show the behaviour of two factors that give some hints for the decrease of the speedup.

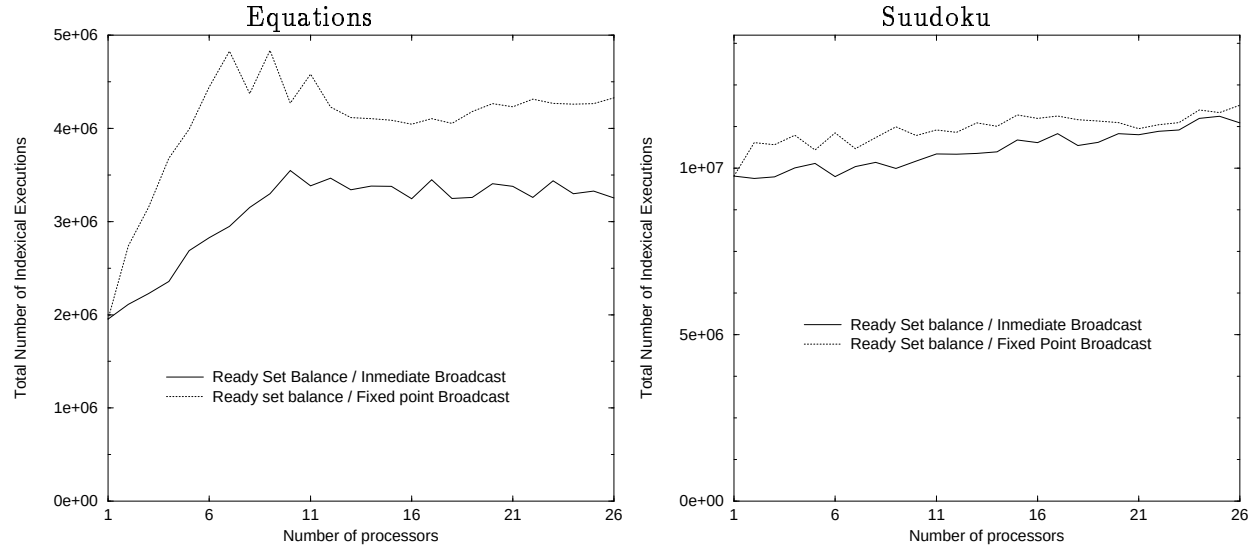

Figure 7: Total number of indexical executions

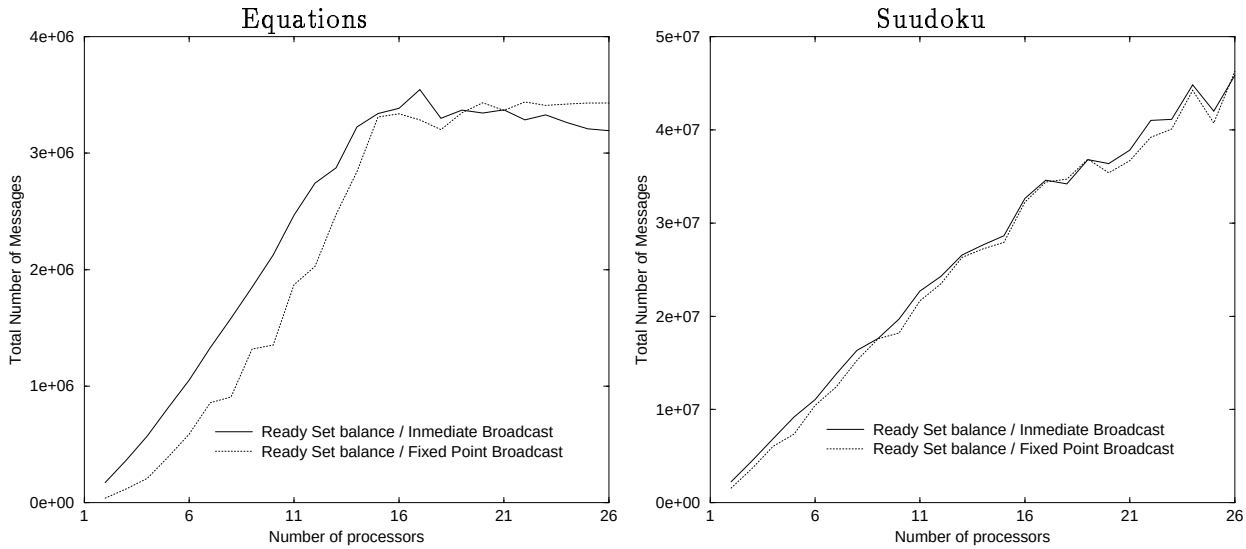

Figure 8: Total number of messages sent

Figure 7 shows the sum over all processors, over all propagation cycles, of the number of executed indexicals. It can be observed the increase of the total number of indexicals executions when running the problem in parallel, since some indexical executions work with less updated data. For both problems, broadcasting at the fixed point implies a higher number of indexical executions, but just a slightly lower number of messages (figure 8), thus explaining the lower speedup obtained.

Figure 8 shows the total number of sent messages. Results for the equations problem present a somewhat surprising behaviour: the total number of messages becomes stable or even decreases beyond a certain number of processors. This behaviour may be because variable domain changes are only sent to those processors which need them, and in this particular problem, as more processors are added, it is more likely that a change does not need to be sent to every processor.

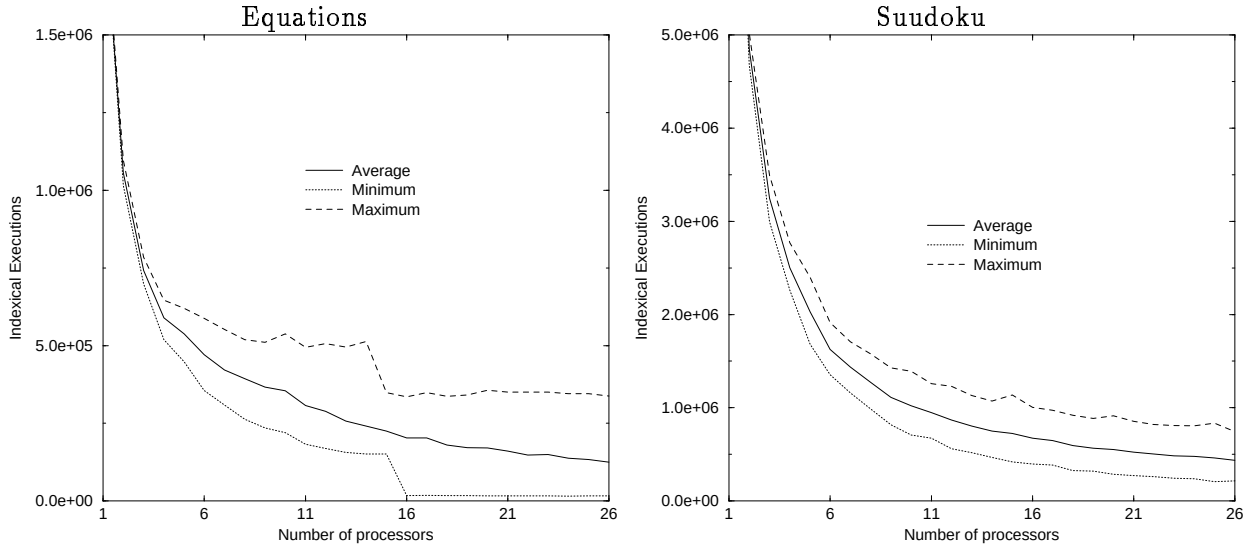

Figure 9: Indexical executions per processor (Ready Set balance / Immediate Broadcast)

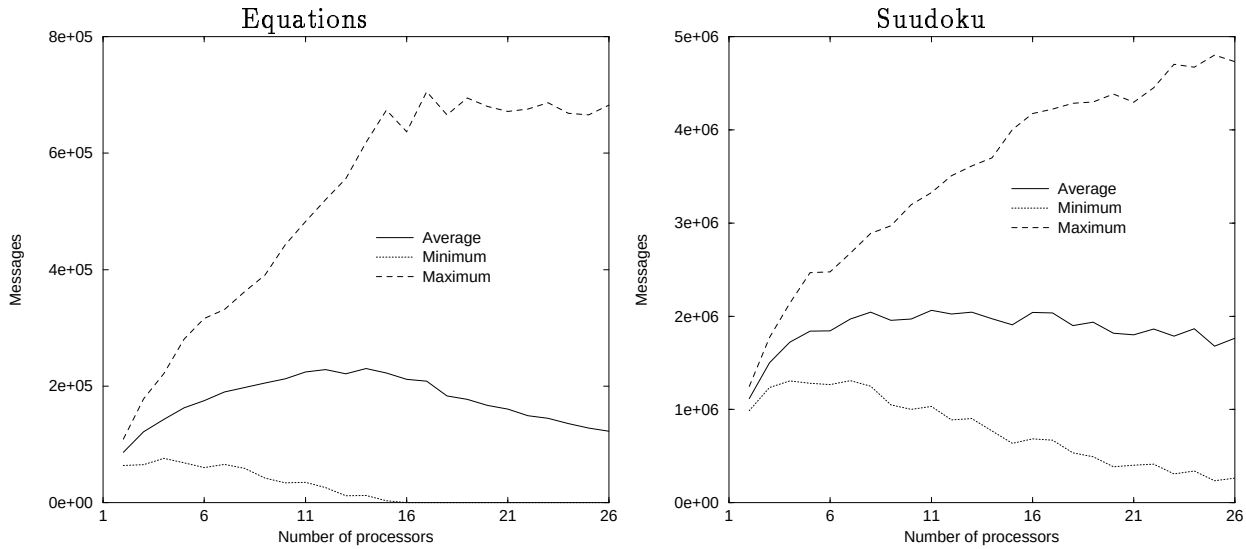

Figure 10: Messages per processor (Ready Set balance / Immediate Broadcast)

Figure 9 shows the average number of indexicals executed per processor (solid), and the sum, over all propagation cycles, of the minimum (long dashed) and of the maximum (dotted) number of indexicals executed per processor. Figure 10 compares the number of messages sent per processor. Both charts exhibit a large difference between minimum and maximum values, which increases as more processors are added. The difference between minimum and maximum indicates load work balance quality. The larger number of processors, the worse load work balance is. This behaviour limits the performance, since the execution time corresponds to the slower processor.

5 Conclusions

We have developed a parallel constraint solving system for functional constraints, based on the indexical scheme. This system has been implemented on a CRAY T3E, a distributed memory MIMD multiprocessor, and empirical data are reported for a set of benchmarks.

A number of issues affecting performance have been investigated in order to tune the model. The way constraints are distributed among processors, and the frequency of updating shared variables, are determining factors for the performance of the model. The study of the distribution of constraints among processors has shown that a strongly connected partitioning (high number of shared variables) is, in general, worse than a distribution which balances the ready set. Tests on broadcast frequency revealed the convenience of an immediate update. Benchmarks considered so far exhibit a speedup between 2.0 and 3.0, though better results may be expected for larger problems.

Acknowledgements

This work has been funded by PRONTIC TIC95-0433.

References

[1] Agrawal, V.D.: Performance analysis of synchronized iterative algorithms on multiprocessors systems. IEEE Transactions on Parallel and Distributed Systems, vol. 3, No. 6 (1992)

[2] Baudot, B., Deville, Y.: Analysis of Distributed Arc-Consistency Algorithms. Tech. Rep. 97-07. Uni. of Louvain, Belgium (1997).

[3] Codognet, P., Diaz, D.: A Simple and Efficient Boolean Constraint Solver for Constraint Logic Programming. Journal of Automated Reasonig. 17,1 (1996) 97-128.

[4] Collin, Z., Dechter, R., Katz, S.: On the feasibility of distributed constraint satisfaction. Proceedings of the International Joint Conference on AI (IJCAI) (1991) 318-324.

[5] Cooper, P.R., Swain, M.J.: Arc consistency: parallelism and domain dependence. Artificial Intelligence 58 (1992) 207-235.

[6] Dincbas, M., Simonis, H., Van Hentenryck, P.: Solving Large Combinatorial Problems in Logic Programming. Journal of Logic Programming 8 (1990).

[7] Guesguen, H.W.: Connectionist networks for constraint satisfaction. AAAI Spring Symposium on Constraint-based Reasoning (1991) 182-190.

[8] Guesguen, H.W., Hertzberg, J.: A perspective of constraint-based reasoning. Lecture Notes on Artificial Intelligence, Springer-Verlag (1992).

[9] Kasif, S.: On the parallel complexity of discrete relaxation in constraint satisfaction networks. Artificial Intelligence 45 (1990) 275-286.

[10] Kasif, S., Delcher, A.L.: Local Consistency in Parallel Constraint-Satisfaction Networks. Artificial Intelligence 69 (1994) 307-327.

[11] Mohr, R., Henderson, T.C.: Arc and path consistency revisited. Artificial Intelligence 28 (1996) 225-233.

[12] Nguyen, T., Deville, Y.: A Distributed Arc-Consistency Algorithm. Science of Computer Programming, 30 (1998) 227-250.

[13] Ruiz-Andino, A., Araujo, L., Ruz, J.: Parallel constraint satisfaction and optimisation. The PCSO system. Technical Report 71.98. Department of Computer Science. Universidad Complutense de Madrid (1998)

[14] Swain, M.J., Cooper, P.R.: Parallel hardware for constraint satisfaction. Proceedings of the National Conference on Artificial Intelligence (AAAI) (1988) 682-686.

[15] Tsang, E.: Foundations of constraint satisfaction. Academic Press (1993).

[16] Van Hentenryck P., Deville, Y., Teng C.M.: A generic Arc-consistency Algorithm and its Specialisations. Artificial Intelligence 57 (1992) 291-321.

[17] Wallace, M.: Constraints in Planing, Scheduling and Placement Problems. Constraint Programming, Springer-Verlag (1994).

[18] Waltz, D.: Generating semantic descriptions for drawings of scenes with shadows. Technical Report AI271, MIT, Cambridge, MA. (1972).

[19] Yokoo, M.: Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. Principles and Practice of Constraint Programming (1995) 88-102.

[20] Zhang, Y., Mackworth, A.K.: Parallel and Distributed Algorithms for Constraint Networks. Technical Report 91-6, Dept. of Computer Science, University of British Columbia, Vancouver, Canada (1991).

Natural Language Multiprocessing: A Case Study

Enrico Pontelli Gopal Gupta Janyce Wiebe David Farwell
Dept. Computer Science and Computing Research Laboratory
New Mexico State University
{epontell,gupta,wiebe}@cs.nmsu.edu, david@crl.nmsu.edu

Abstract

This paper presents two case studies of parallelization of large Natural Language Processing (NLP) applications using a parallel logic programming system (called "ACE") that automatically exploits implicit parallelism. The first system considered is *Artwork*, a system for semantic disambiguation, speech act resolution, and temporal reference resolution. The second system is *ULTRA*, a multilingual translation system. Both applications were originally developed in Prolog without any consideration for parallel processing. The results obtained confirm that NLP is a ripe area for exploitation of parallelism. Most previous work on parallelism in NLP focused primarily on parallelizing the parsing phase of language processing. The case studies presented here show that parallelism is also present in the semantic and discourse processing phases, which are often the most computationally intensive part of the application.

1 Introduction

Logic programming has been for a long time one of the programming paradigms of choice for the development of NLP systems. Logic programming languages (e.g., Prolog) offer features such as backtracking, unification, and symbolic data representation which greatly facilitate the development of NLP applications [13]. In recent years the use of declarative languages, such as Prolog and Lisp, for NLP applications has declined. This is mostly due to the increasing computational requirements of these applications, which were not satisfied by the older slow implementations of declarative languages. Even though implementations of these languages comparable in efficiency to imperative languages are available today (e.g., [3]), past experience has discouraged designers of NLP systems from using them.

The computational demands of today's NLP systems have reached the level where they are challenging for even the most efficiently implementable programming paradigms, including imperative languages. Exploitation of parallelism is an important technique, especially given that multiprocessor machines (e.g., dual and quad Pentium-based PCs) are readily and cheaply available in the market today. Logic programming languages have a distinct advantage over imperative languages with regards to parallelism: their mathematical semantics allow for *automatic* or *semi-automatic* exploitation of parallelism, thus relieving the programmer from most of the intricacies of parallel programming and debugging. One of the goals of this paper is to make the NLP community aware of the recent advances made in implementation and parallelization technology for logic programming, and their potential benefits to NLP systems. The current implementations of logic programming systems are comparable in execution efficiency to any other programming paradigm. We hope that all of these attractive features—the ability to automatically or semi-automatically exploit parallelism, realization of non-determinism, support of symbolic data representation, and

efficiency comparable to imperative paradigms—will convince NLP system designers to reconsider logic programming as an excellent implementation choice for many NLP applications. Given that parallelism can be exploited automatically, existing Prolog-based NLP applications can also be parallelized, by porting them with little or no effort to parallel systems. Fast parallel implementations of Prolog are currently available (e.g., [2]) or about to be released into the public domain, including the ACE system [7] used in this work.

In this work we present two case studies in which two large NLP applications, independently developed with no goal of parallelization in mind, have been studied and parallelized using the ACE parallel Prolog system. The parallelization effort was very limited in both cases and lead to excellent performance and speedups for both applications. In the rest of this paper we give a brief description of parallel logic programming and the ACE system followed by an overview of the two NLP applications considered (Artwork and ULTRA). Finally, we analyze the performance results obtained.

2 Parallelism in Logic Programming

Logic programming is a programming paradigm in which programs are expressed as logical implications. An important property of logic programming languages is that they are single assignment languages. Unlike conventional programming languages, they disallow destructive assignment and explicit control information. Not only does this allow cleaner (declarative) semantics for programs, and hence a better understanding of them by their users, it also makes it easier for a runtime evaluator of logic programs to employ different control strategies for evaluation. That is, different operations in a logic program can be executed in any order without affecting the (declarative) meaning of the program. In particular, these operations can be performed *implicitly in parallel*. Parallelization can be done directly by the runtime evaluator as suggested above, or, alternatively, it can also be done by a parallelizing compiler. The task of the parallelizing compiler is to unburden the evaluator from making run-time decisions regarding which parts of the program to run in parallel. Note that the program can also be parallelized by the user (through suitable annotations). In all cases, the advantage offered by logic programming is that the process is easier because of the more declarative nature of the language.

Three principal kinds of (implicitly exploitable) control parallelism can be identified in logic programs: *(i) Or-parallelism* arises when more than one clause defines some predicate and a literal unifies with more than one clause head—the corresponding bodies can then be executed in parallel with each other. Or-parallelism is thus a way of efficiently searching for solutions to the query, by exploring alternative solutions in parallel. *(ii) Independent and-parallelism (IAP)* arises when more than one goal is present in the query or in the body of a clause, and at runtime these goals to do not compete for any unbound variable. *(iii) Dependent and-parallelism (DAP)* arises when two or more goals of a clause access common variables and are executed in parallel.

2.1 The ACE System

The ACE model [7, 14] uses stack-copying [2] and recomputation [7] to efficiently support combined or- and independent and-parallel execution of logic programs. ACE represents an efficient combination of or- and independent and-parallelism in the sense that penalties for supporting either form of parallelism are paid only when that form of parallelism is actually exploited. This efficiency in execution is accomplished by introducing the concept of *teams of processors* and extending the stack-copying techniques to deal with this new processor organization.

Or-Parallelism in ACE: ACE exploits or–parallelism by using a stack copying approach [2]. In this approach, a set of processing *agents* (processors in the case of MUSE, teams of processors in the case of ACE—as explained later) working in or-parallel maintain a *separate* but *identical* address space (i.e. they allocate their data structures starting at the same logical addresses). Whenever an *or-agent* (agents working in or-parallel are termed or-agents) $\mathcal{A}$ is idle, it will start looking for unexplored alternatives generated by some other or-agent $\mathcal{B}$. Once a choice point p with unexplored alternatives is detected in the computation tree $\mathcal{T}_\mathcal{B}$ generated by $\mathcal{B}$, then $\mathcal{A}$ creates a local copy of $\mathcal{T}_\mathcal{B}$ and restarts computation by backtracking over p and executing one of the unexplored alternatives. The fact that all the or-agents maintain an identical logical address space reduces the creation of a local copy of $\mathcal{T}_\mathcal{B}$ to a simple block memory copying operation.

In order to reduce the number of copying operations performed (since each copying operation may involve a considerable amount of overhead), unexplored alternatives are always searched starting from the bottommost part of the tree; during the copying operation all the choice points in between are shared between the two agents (i.e., at each copying operation we try to maximize the amount of work shared between the two agents). Furthermore, in order to reduce the amount of information transferred, copying is done *incrementally*, i.e., only the difference between $\mathcal{T}_\mathcal{A}$ and $\mathcal{T}_\mathcal{B}$ is actually copied.

And-Parallelism in ACE: ACE exploits IAP using a recomputation based scheme—no sharing of solutions is performed (at the and-parallel level). This means that for a query like `?- a,b`, where `a` and `b` are nondeterministic, `b` is completely recomputed for every solution of `a` (as in Prolog). The computation tree created in the presence of and-parallel computation has a *parbegin-parend* structure, and the different branches are assigned to different agents. Since we are exploiting only *independent and-parallelism*, only independent subgoals are allowed to be executed concurrently by different *and-agents* (and-agents are processing agents working in and-parallel with each other). Dependencies are detected at run-time by executing some simple tests introduced by the *parallelizing compiler*. ACE adopts the technique originally designed by DeGroot [4] and refined by Hermenegildo [8] of annotating the program at compile time with *Conditional Graph Expressions (CGEs)*:

$$(\langle\, conditions \,\rangle \;\Rightarrow\; \mathtt{B}_1 \,\&\, \cdots \,\&\, \mathtt{B}_n)$$

where $\langle conditions \rangle$ is a conjunction of simple tests on variables appearing in the clause that verifies whether the arguments share any variables with arguments of other goals, and & denotes *parallel conjunction*. Intuitively, if the tests present in *conditions* succeed, then the subgoals $\mathtt{B}_1 \,\&\, \cdots \,\&\, \mathtt{B}_n$ can be executed in and-parallel, else they should be executed sequentially.

Since and-agents are computing just different parts of the same computation (i.e. they are cooperating in building one solution of the initial query) they need to have *different* but *mutually accessible* address spaces.

When a CGE is seen, a new descriptor for the parallel call (named *parcall frame*) is allocated, initialized, and all the subgoals but the leftmost one are loaded in a local work queue (*goal stack*) (the leftmost subgoal is directly executed by the same and-agent that created the parcall frame). The same processor performing the creation of the parallel call will eventually fetch and execute other unexecuted parallel subgoals from this parallel call, if necessary. An idle processor may pick work from the work queue of other processors. This will entail the identification of the subgoal to execute, the allocation of an initial data structure, the actual computation of the subgoal, and finally the allocation of further structures to identify the completion of the subgoal. *Backward execution* denotes the series of steps that are performed following a *failure*—due to unification or lack of matching clauses. In ACE, where both or- and and-parallelism are exploited, backtracking should also avoid taking alternatives already taken by other or-agents. In the presence of CGEs, standard backtracking should be upgraded in order to deal with computations which are spread

across processors.

And-Or Parallelism in ACE: In ACE a clear separation is made between exploitation of or-parallelism and exploitation of and-parallelism. Processors in the multiprocessor system are divided into *teams of processors*. At a higher level, these teams of processors *execute in or-parallel* with each other (i.e., a team will take up only or-parallel work). At a lower level, i.e., within each team, processors in the team *execute in and-parallel* with each other (i.e., along an or-branch taken by a team, processors in that team will execute goals arising in that branch in and-parallel). Thus, the the notion of *or-agent* is mapped to the notion of *team of processors* while the notion of *and-agent* is mapped to the notion of processors inside a team (i.e. each processor is an and-agent). Different approaches to incremental copying and heuristics have been developed [7].

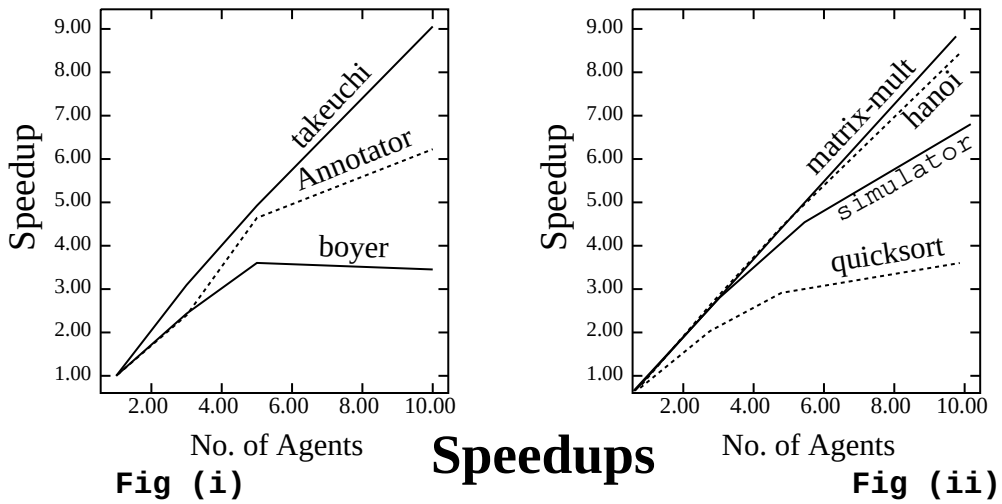

Figure 1: Speedups in ACE

ACE has shown remarkable performance results on a large selection of programs (Figure 1 presents the speedup curves obtained on some benchmarks). Furthermore, the parallelization overhead is extremely low, on average 5 to 10% w.r.t. sequential SICStus Prolog, on which the ACE engine is based.

3 NLP Applications

The field of NLP has changed dramatically over the last few years. The pressure from the users community has pushed NLP to increase the amount of effort towards the development of specific, practical applications and large-scale language processing systems. This has created an increasing need for efficient implementation schemes for many of the theoretical concepts developed in the past. NLP is probably one of the application fields with the highest computational requirements. Problems tend to generate high levels of nondeterminism and ambiguity, making exhaustive approaches entirely not feasible. This forces researchers to focus on smaller problems or operate under more conservative assumptions. Our thesis in this paper is that by resorting to exploitation of parallelism many of these conservative assumptions can be relaxed, as well as larger problems solved. We take advantage of parallelism implicit in NLP applications to achieve efficient execution as well as improve the precision of the solutions obtained (by exploring a larger portion of the search space). Our experience with parallelizing the Artwork and the ULTRA systems and the results obtained confirm our thesis. The elegance and the semantic clarity of logic programming not only

allowed simplification of the software development process, but also permitted automatic exploitation of parallelism; such exploitation of parallelism and execution speedups would not have been possible if the code were written in an imperative programming language, since for such languages automatic parallelization is extremely difficult.

3.0.1 Artwork:

When people write or converse, they leave much of what they are communicating implicit, leaving the listener or hearer to fill in the missing information by considering the surrounding context. Consider, for example, understanding the pronouns in the sentence "She was there that day." Who was where on what day depends on what was said previously. The tasks of the discourse processing component of a NLP system involve recovering information that is implicitly communicated in the text or dialog. Such a component performs these tasks by considering information across sentence boundaries, rather than processing each sentence in isolation.

Only recently have major efforts been invested in empirical investigations of computational theories of discourse processing (see, for example, the special issue of *Computational Linguistics* devoted to empirical studies in discourse [19]). The Artwork project is one such NLP project, funded by the Department of Defense. It targets *scheduling dialogs*, dialogs in which the participants schedule a meeting with one another. A fully automatic rule-based system was developed that performs many kinds of natural language disambiguation, including *semantic* disambiguation, and two types of discourse disambiguation: *speech-act* and *temporal reference* resolution.

The input to the system is the output of a semantic parser developed as part of the Enthusiast speech-to-speech machine translation project at CMU [10]; it determines, among other things, what type of event each utterance is about, and who the participants are. The output of the parser is ambiguous, so must be disambiguated by the Artwork system.

The first of the discourse ambiguities addressed involves a prominent view of language as goal-oriented behavior. Under this view, which is adopted in this project, utterances are produced by actions that are executed with the goal of having some particular effect on the hearer. These actions are called *speech acts*. The task of an understanding system is to recognize which speech acts the speaker is performing with his or her utterances [16, 17, 20]. Consider the utterance "2 to 4" (dos a cuatro), a common type of utterance in the scheduling dialogs. The speaker might be *suggesting* that they meet from 2 to 4; they might be *confirming* that 2 to 4 is the time currently being discussed; they might, with the right intonation, be *accepting* 2 to 4; and so on. The other kind of discourse ambiguity addressed is temporal [21]. The Artwork system tracks the times being talked about, determining implicit contextual information. For example, when a speaker refers to "2 to 4 am", Artwork looks back at the previous utterances, and decides which day, date, and month are being referred to. This involves a search for the best possible candidate which fits the constraints generated by the previous utterances. Temporal reference resolution is the primary focus of the current Artwork system. Wiebe et al. (1997) present the results of the system performing this task on unseen, held-out test data, taking as input the ambiguous output of the semantic parser (which itself takes as input the output of a speech recognition system [10]). The system performs well, and comparable results on similar tasks have not been published elsewhere.

The system parallelized in this work is an earlier version which performed speech-act and temporal reference resolution, but not semantic disambiguation. It includes the core architecture and speech-act and temporal-reference resolution rules of the current system. We expect the results obtained in this study to transfer easily to the latest versions of Artwork.

3.0.2 ULTRA:

ULTRA (Universal Language TRAnslator) is a multilingual, interlingual machine translation system. It can currently translate between five languages (Chinese, English, German, Spanish, and Japanese) with vocabularies in each language based on about 10,000 word senses. The multilingual system is based on a language-independent interlingual representation (IR) [6] for representing expressions as elements of linguistic acts of communication (e.g., asking questions, describing the world, promising that things will get done, etc.). Translation can be viewed as the use of the target language to express the same act as the one expressed in the source language. The IR is then used as the basis for analyzing or for generating expressions as elements of such acts in each of the languages [6].

Each individual language system is independent and has its own rules to associate the appropriate IRs to each expression of the language. The different language components interact exclusively via IRs—thus allowing, for example, a clear system design and making the addition of new languages very easy without unpredictable effects on the rest of the system. It also gives freedom to the implementor to choose the class of grammars and the parser (s)he prefers. Currently most of the language components are implemented as context-free grammars with complex categories.

The system uses relaxation techniques (grammatical relaxation, semantic relaxation, and structure relaxation) to provide robustness by giving preferred or "near miss" translations [6]. In addition, the adoption of language-independent semantic and pragmatic procedures allows, given a context, the selection of the best IR from the set of possible IRs for a given expression. The use of Prolog allowed the design of a highly declarative and perfectly bidirectional application.

The current prototype produces word, phrase, or sentence level translations and handles most basic declarative, interrogative, and imperative structures, including conjoined and subjoined constructions, while dealing with various types of sense disambiguation and structurally dependent anaphora and ellipsis.

4 Experimental Results

4.0.3 Parallelization of Artwork:

The parallelization of Artwork was performed semi-automatically. The ACE compiler was used to identify potential sources of parallelism. Additionally, the rich output of the ACE static analyzer [15] (e.g., sharing information) allowed us to identify features of the program that were limiting the parallelism exploitable. Very few hand-modifications of the original code were needed to considerably improve the speedups achieved, as discussed in the next subsection.

Ambiguity in NLP gives rise to a "combinatorial explosion" of possible interpretations. Consequently, Artwork may take up to a couple of hours to process a dialog [20]. Artwork offers a great deal of inherent parallelism to exploit, including or-parallelism and both DAP and IAP.

To process each utterance, the system applies all rules in its knowledge base. Each rule that matches the utterance fires, producing a partial representation. The rules are not all competitors; some of them target one ambiguity while other rules target others. All possible maximal merging of the results are formed, resulting in the set of interpretations the system chooses among to be the representation of the utterance. The application of the rules is realized through a subgoal of the form:

```
findall(R,int(PrevI,PrevUt,CurI,T_Ut,_,_,R),List)
```

where the `int` predicates is defined as:

```
int(PrevI,PrevUt,CurI,CurUt,CF,Rule,Res):-
```

```
    int01(PrevI,PrevUt,CurI,CurUt,CF,Rule,Res).
int(PrevI, PrevUt,CurI,CurUt,CF,Rule,Res):-
    int02(PrevI,PrevUt,CurI,CurUt,CF,Rule,Res).
....
```

The rules are independent of one another, which allows their application to the current utterance in parallel. The `findall` was unfolded giving rise to instances of IAP. The unfolding into and-parallelism was preferred to an or-parallel execution of `findall` to reduce the amount of parallel overhead.

The ACE compiler was capable of detecting a good source of DAP in the merging process, where partial results become known, by performing in parallel the separate merging of two incompatible partial representations with the others. The parallel annotation produced for this phase is:

```
buildComplete(ILTlist,PrevI,Ut,PrevU,Comp,CList):-
    getfirst(ILT,ILTlist,R), normalize(ILT,CurI),
    (dep([PList]) →
      buildPartials(PrevI,CurI,PrevU,Ut,PList) &
      merge(PList,CurI,NewComp)),
    append(Comp,NewComp,NList),
    buildComplete(R,PrevI,Ut,PrevU,NList,CList).
```

The two subgoals separated by '&' can be executed concurrently, and the variable annotated with dep represents the communication channel between the two parallel threads. Unfortunately, the presence of the append subgoal represents a barrier which does not allow the results to be immediately propagated to the continuation of the computation. This problem was tackled by hand-modifying the code to allow exploitation of parallelism between the processing of the different elements of the input list `ILTlist`. Smaller sources of parallelism are present in various parts of the program, e.g., IAP between different actions in the body of each rule.

From the point of view of or-parallelism, we have identified various components of the systems where a local search is performed, making them suitable for or-parallel execution. For example, table 1 illustrates the improvement in execution of the selection of prediction phase. Globally, or-parallelism did not produce excessive speedups, due to the shallow searches produced by the Artwork benchmarks available. Furthermore, or-parallelism in Artwork was negatively affected by frequent use of various side-effect predicates and cuts.

Query	**ACE Agents**			
	1	2	3	4
Sentence_1	4810	3620	1623	1503

Table 1: Parallel prediction (Sun Sparc, times in ms.)

Speedups of the Artwork system from exploiting and-parallelism was considerably more satisfactory. The initial speedups observed were modest, as illustrated in the first two lines of table 2. These modest speedups, as discussed above, are not due to a lack of parallelism inherent in the application, but to the occasional use of some partially non-declarative constructions. Very few hand-coded modifications (driven by the results obtained from the static analyzer) were performed to make the code more declarative. These modifications should not be regarded as "tinkering" with the program to elicit better speedups; rather, they were simply attempts to replace extra-logical built-ins and features of Prolog with declarative ones. The more declarative and semantically elegant the program the more parallelism there is to exploit. The excellent results obtained from this

modified version of the system can be seen in the last two lines of table 2 (speedups in fig. 2). The parallel overhead recorded is less than 10%.

Goals executed	ACE agents 1	2	3	5	10	15	18
Discourse 1	78300	62060 (1.26)	51679 (1.52)	50389 (1.55)	50192 (1.56)	50190 (1.56)	50190 (1.56)
Discourse 2	48570	31234 (1.56)	21539 (2.25)	21500 (2.26)	20935 (2.32)	20650 (2.35)	20622 (2.36)
Modified 1	73594	41345 (1.78)	27156 (2.71)	17950 (4.1)	11870 (6.2)	10365 (7.1)	10301 (7.14)
Modified 2	46529	23639 (1.97)	15919 (2.92)	9889 (4.71)	6200 (7.5)	4640 (10.03)	4409 (10.55)

Table 2: Execution Times for Artwork (Sequent Symmetry, times in ms.)

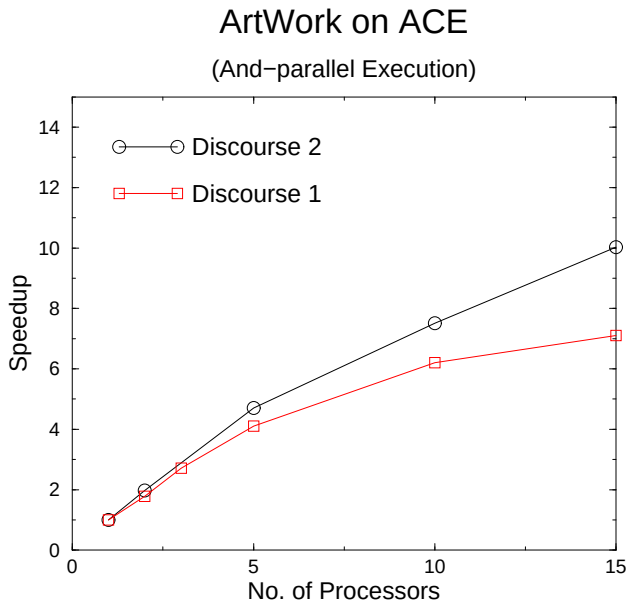

Figure 2: Speedups for and-Parallel Artwork

As part of future work, we plan to explore the use of or-parallelism to relax limitations in the number of input alternatives considered by the current system (recall that the input to the current system is ambiguous). The current limitations may lead to the exclusion of correct interpretations. We expect that processing alternative interpretations in parallel will allow the system to consider additional candidate interpretations, increasing the chance of a correct interpretations.

4.0.4 Parallelization of ULTRA:

The parallelization of the ULTRA system was performed completely automatically (note that the ULTRA system was developed long before the development of the ACE system was begun).

The analyzer was quite successful in detecting parallelism in the program. At the highest level, IAP was exploited by allowing concurrent translation of successive phrases belonging to the source text. The translation of each phrase was marked as a potential source of DAP by the dependent and-parallel analyzer [15], as illustrated below:

```
ctrans(Src_lg,Trg_lg,In,Out) :-
    (dep([I_rep])-> (analyse(Srg_lg,In,I_rep) &
  generate(Trg_lg,I_rep,Out));
```

```
   Out_String = "< unable to translate >").
```

where '&' denotes a parallel conjunction and the dep annotation identifies the source of dependency (*shared variable*). Similarly, the process of analyzing of each individual sentence to produce the corresponding IR contains considerable amounts of both DAP and IAP, as illustrated in the example below:

```
e_analyse(String,Struc) :-
  dep([List]) -> (prep_list(String,List) &
         e_prdctn(_A,_B,_C,Struc,List,[])).
```

All the lower level rules which identify the various sentence structures provide additional IAP:

```
e_prdctn(indpnt,conclusion,
     nil,[prdctn,[type,indpnt],
     [class,conclusion],[form,nil],
     [closing, Conclusion], Name],In,Out) :-
                split_input(In,In1,In2,Out),
                (esign_off(In1,Conclusion) &
                 ep_name(In2,human,Name).
```

IAP emerged consistently also in the the second phase of the translation, where IRs are mapped to sentences in the target language. The rule below is an example:

```
s_prdctn(indpnt,Pc,fin,
     [prdctn, [type, indpnt],
     [class, Pc], [form, fin],P1,P2]):-
 (s_prop1(dpnt,adv,fin,_M,_S_a1,_S_g1,P1) &
  s_prop1(indpnt,Pc,fin,_Pm,_S_a2,_S_g2,P2)),
 (Pc = dcl; Pc == imp; Pc = int).
```

No significant or-parallelism was detected in these examples (as indicated, for example, in table 3). In the next tables, *E-to-C* (*E-to-S*, *E-to-G*) indicates translation from English to Chinese (Spanish, German).

Query	**ACE Agents**		
	1	2	4
E-to-C	322669	290402 (1.11)	251682 (1.29)

Table 3: Or-parallelism in ULTRA (Sequent, ms.)

Table 4 presents the performance figures achieved running the parallelized version of ULTRA on three separate examples, translating the same text from English to three different target languages, Chinese, Spanish, and German. In most cases the improvement in execution time due to exploitation of parallelism is excellent, confirmed by the speedup curves (fig. 3). The parallel overhead is extremely low (around 5%).

Goals	ACE agents						
executed	1	2	3	5	10	15	18
E-to-G	562740	282339 (1.99)	190115 (2.96)	128509 (4.38)	65283 (8.62)	46507 (12.1)	37768 (14.9)
E-to-C	322669	160100 (2.0)	108642 (2.97)	70145 (4.6)	37088 (8.7)	24631 (13.1)	20817 (15.5)
E-to-S	91519	45756 (2.0)	30505 (3.0)	18370 (4.98)	10057 (9.1)	6779 (13.5)	5684 (16.1)

Table 4: Execution Times for ULTRA (Sequent Symmetry, times in ms.)

The automatic annotation performed by the ACE compiler was rather slow, due to the size and organization of the application—a single module of over 35,000 lines of Prolog code. A modular reorganization of the code and the use of incremental analysis techniques [8] will improve the speed of annotation. The annotator is written in Prolog, so the annotation process itself can be parallelized.

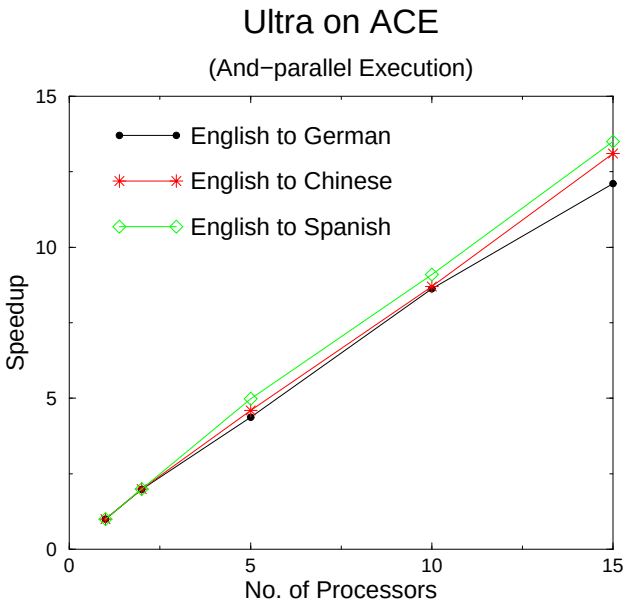

Figure 3: Speedups for ULTRA System

5 Comparison with Other Work

Experience shows that NLP applications are highly parallel in nature. Although considerable research has been proposed in using parallelism for NLP (see [1] for a review of approaches to parallel NLP), most of the approaches are based on ad-hoc and hand-made parallelization of NLP applications.

There has been research in the past in parallelizing the parsing phase of NLP [5], including some that use a logic programming approach [12, 18]. More recently, the Eu-PAGE systems [11] has been proposed, a parser generator capable of producing parallel parsers (based on PVM). However, not much parallelism has been extracted from other phases of NLP systems. The main exception to this is the PUNDIT system [9] which exploits (only) or-parallelism from an NLP system coded in Prolog running on a simulated parallel environment. The PUNDIT system is about 3,000 lines long. Thus, our proposed work can be seen as taking the work done on the PUNDIT system further and exploiting both or- and and-parallelism from NLP programs. It should be noted that almost all work on parallel NLP (including the PUNDIT system) relies on explicit parallelization of the NLP system by the programmer. Our research results in this respect are distinctive in that: (i) we exploit parallelism from NLP applications implicitly or with limited programmer intervention, and (ii) we have been successful in parallelizing a significantly larger NLP application (it should be noted that in the PUNDIT system, the developers of the original system were the ones who did the parallelization; the same is true of other parallel NLP systems), whereas in our case system development and parallelization was decoupled.

6 Conclusions and Future Work

In this paper we presented two case studies of parallel execution of large NLP applications—Artwork and ULTRA. In both cases we have taken advantage of the fact that the applications have been developed using logic programming, and logic programming is particularly suited to both automatic and semi-automatic exploitation of parallelism inherent in the application. In both cases we relied on the use of automatic compile-time analysis and run-time tools to extract parallelism (either directly or to supply information to the programmer to achieve this goal). Thus, the exploitation of parallelism required only a limited understanding of the behaviour and structure of the programs.

Our results show that the use of logic programming allows one to go beyond just the parallelization of the parsing phase of NLP systems, which has been the sole focus of most of the previous efforts. That is, it also allows the exploitation of parallelism from the semantic and discourse processing phases—which are typically the more computationally intensive.

In the case of ULTRA the goal of parallelization was quickly achieved with limited effort, thanks to the clean and declarative programming style adopted by the programmers who developed the system, necessitated by the requirement that computations be reversible [6]. In the case of Artwork the system had been developed following a more "imperative" approach in the code organization. We invested some effort in reorganizing some parts of the code and this proved effective in increasing the amount of and-parallelism. Our experience confirms the fact that adoption of an elegant, declarative style for program development not only makes the code more clear and readable, it also leads to an increase in the amount of parallelism inherent in the application exposed.

We are currently studying the possibility of using parallelism to improve the precision of the results produced by Artwork. For tractability, the rules for resolving temporal ambiguity are distinct from the rules for resolving speech act ambiguity. The temporal rules are applied in a first pass through the data, and the speech act rules are applied to the results of the first pass. This way, Artwork avoids considering all possible combinations of temporal and speech act features. However, the speech-act information cannot assist the system in resolving the temporal ambiguity. With a parallel implementation, more combinations can be processed, and this allows to perform a limited amount of integrated temporal and speech act processing.

References

[1] Adriaens, G., and Hahn, U. 1994. *Parallel Natural Language Processing.* Ablex Publishing.

[2] Ali, K., and Karlsson, R. 1990. The Muse Or-parallel Prolog.. In *NACLP.* MIT Press.

[3] M. Carlson and G. Gupta (eds.) 1996. *Journal of Logic Progr.* 29(1-3).

[4] DeGroot, D. 1984. Restricted AND-Parallelism. In *Conf. on 5th Generation Computer Systems.*

[5] Devos M. et al. 1988. The Parallel Expert Parser. In *Proceedings of COLING.*

[6] Farwell, D., and Wilks, Y. 1991. ULTRA: A multilingual machine translator. In *Mach. Transl. Summit.*

[7] Gupta, G., Pontelli, E. et al. 1994. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. ICLP'94*, 93–109. MIT Press.

[8] Hermenegildo, M. et al. 1995. Incremental analysis of logic programs. In *ICLP95.* MIT Press.

[9] Hirschman L. et al. 1988. Or-parallel Speedup in Natural Language Processing. In *ICLP88*. MIT Press.

[10] Levin, L. et al. 1995. Using Context in the Machine Translation of Spoken Language. In *Proc. Theoretical and Methodological Issues in Machine Translation.*

[11] Manousopoulou A. et al. 1997. Automatic Generation of Portable Parallel Natural Language Parsers. In *ICTAI*. IEEE Computer Society.

[12] Matsumoto, Y. 1986. A Parallel Parsing System for Natural Language Analysis. In *Int. Conf. on Logic Programming*. Springer Verlag.

[13] Pereira, F. and Shieber, S.M. 1987. *Prolog and Natural-Language Analysis.* Cambridge University Press.

[14] Pontelli, E. 1997. *High-Performance Parallel Execution of Prolog Programs.* Ph.D. Dissertation, NMSU.

[15] Pontelli, E., Gupta, G. et al. 1997. Automatic Compile-time Parallelization of Prolog Programs for DAP. In *ICLP97*. MIT Press.

[16] Reithinger, N. et al. 1995. Utilizing statistical dialogue act processing in verbmobil. In *ACL*, 116–122.

[17] Rosé, C. et al. 1995. Discourse processing of dialogues with multiple threads. In *Proceedings of ACL*, 31–38.

[18] Trehan, R. et al. 1988. A Parallel Chart Parser for the CCND Languages. In *ICLP88*. MIT Press.

[19] M. Walker and J. Moore (eds.) 1997. *Computational Linguistics* 23(1).

[20] Wiebe J. et al. 1996. ARTWORK: Discourse Processing in Machine Translation of Dialog. Technical Report MCCS96294, Computing Research Laboratory.

[21] Wiebe J. et al. 1997. An Empirical Approach to Temporal Reference Resolution. In *Proceedings 2nd Conference on Empirical Methods in NLP.*

Pseudo-Naive Evaluation: Mixing Top-down and Bottom-up Execution for Strongly Stratified Logic Programs

Donald A. Smith Mark Utting

Department of Computer Science
The University of Waikato
Private Bag 3105, Hamilton, New Zealand
Email: {dsmith,marku}@cs.waikato.ac.nz

Abstract

We introduce a new bottom-up method, pseudo-naive evaluation, for execution of mixed top-down/bottom-up logic programs. The method is intermediate in power between naive evaluation and semi-naive evaluation. Pseudo-naive evaluation adds a data-driven component to naive evaluation without explicitly collecting the 'delta' sets of new facts derivable at each iteration. Instead, pseudo-naive evaluation collects an abstraction of the delta sets, thereby simplifying the implementation. Pseudo-naive evaluation is most efficient on strongly-stratified programs: those programs for which all (positive and negative) bottom-up recursion is mediated by an increasing temporal parameter. We describe our compiler and interpreter and several interesting applications: an elevator simulator with GUI, a robot planner, and a graphical, individual-based ecology simulation. A desirable feature enjoyed by our system is that it runs piggyback on most logic programming implementations, as it uses the standard dynamic database (or equivalent library packages); but performance is good because most of the code is (compiled and) executed by the native Prolog system.

1 Introduction

Prolog, like most logic programming languages, uses a top-down execution mechanism. However, some algorithms are more naturally written in a bottom-up style. We are developing a pure temporal logic programming language, Starlog, that supports both bottom-up and top-down execution in a natural way. Developing execution mechanisms that support both efficiently is an interesting challenge.

Two common evaluation techniques for bottom-up systems are naive evaluation and semi-naive evaluation [12]. Naive evaluation evaluates all rules repeatedly, so is quite inefficient for many programs. Semi-naive evaluation improves efficiency by tracking newly-derived tuples (delta-sets) and only evaluating rules that use those tuples.

This paper describes an intermediate approach, which is more efficient than naive evaluation, but can be implemented more easily on top of standard Prolog systems than semi-naive evaluation. Essentially, our implementation compiles the bottom-up clauses into standard top-down Prolog clauses, then adds some simple control predicates that use a queue structure (an abstraction of the delta-sets of semi-naive evaluation) to execute the rules in a bottom-up fashion. Advantages of this approach are that it reuses the mature implementation and analysis techniques of standard Prolog, it naturally allows programmers to mix top-down and bottom-up code, and it supports declarative programming of I/O and change.

```
naive_evaluation:-
    rule(Head,Body),
    Body,                   % Try proving the body.
    \+ Head,                % Check that Head is new
    assert(Head),
    \+ some_change,         % Fail if some_change already.
    assert(some_change),
    fail.                   %Failure driven loop.
naive_evaluation:-
    retract(some_change),   % Fail if no new facts derived.
    naive_evaluation.       % Iterate.
naive_evaluation.           % End of evaluation.
```

Figure 1: Prolog Interpreter for Naive Evaluation

2 Naive and Semi-Naive Evaluation

Naive evaluation [12] is a direct implementation of the bottom-up, fixed point semantics of logic programs [9], wherein the interpreter computes a monotonically increasing set of atoms derivable from the program clauses. At each step i of the bottom-up computation, an atom H is added to the set S_i if it appears as the head of an instance of a program clause $H \leftarrow B$ whose body atoms B all appear in the set S_{i-1}.

Naive Evaluation

$$\begin{aligned} S_0 &= \emptyset \\ S_{i+1} &= S_i \cup \{H\mu \mid (H \leftarrow B) \in Rename(P) \\ &\quad \wedge (B\mu \subseteq S_i) \wedge (H\mu \notin S_i)\} \quad (i \geq 0) \\ S &= \cup_{i=0\ldots\infty} S_i \end{aligned}$$

(The optional condition $H\mu \notin S_i$ guarantees that only atoms not derived in previous steps are added at time $i+1$.) An interpreter for naive evaluation can be written easily in Prolog (Figure 1).

The inefficiency of naive evaluation results from two phenomena:

Naive-evaluation is not data-driven. Each rule $H \leftarrow B$ is tried, even if no (new) facts have been derived for atoms appearing in the body of the rule.

Naive evaluation duplicates work. Each fact derivable at iteration i of the bottom-up computation is also derivable at iteration j for all $j > i$. (Bottom-up computation of `ancestor/2` is an example of a computation that would result in duplication.)

Semi-naive evaluation [12] overcomes both inefficiencies by making sure that at least one newly derived fact is used to satisfy the body of a program clause.

Semi-naive Evaluation

$$\begin{aligned} S_0 &= \delta_0 = \{H \mid (H \leftarrow \mathsf{true}) \in P\} \ \text{(facts)} \\ \delta_{i+1} &= \{H\mu \mid (H \leftarrow B) \in Rename(P) \end{aligned}$$

$$\wedge(B\mu \cap \delta_i \neq \emptyset) \wedge (B\mu \subseteq S_i) \wedge (H\mu \notin S_i)\} \quad (i \geq 0)$$
$$S_{i+1} = S_i \cup \delta_{i+1} \quad (i \geq 0)$$
$$S = \cup_{i=0\ldots\infty} S_i$$

For each i, δ_i contains the facts newly derived at iteration i. The condition $B\mu \cap \delta_i \neq \emptyset$ is the test that avoids duplication of work. In addition, the same test makes computation data-driven, since only rules that have (newly derived) enabling data are tried.

We forgo presenting a Prolog interpreter for semi-naive evaluation. Indeed, the difficulty of writing such an interpreter – the management of the delta sets is the main stumbling block — is a major reason for our developing pseudo-naive evaluation.

Before describing pseudo-naive evaluation we must explain two preliminary notions: timestamps and strong-stratification.

3 Starlog, an Executable Temporal Logic

We have designed and implemented pseudo-naive evaluation for use as the execution mechanism of Starlog [8, 5], an executable temporal logic [18, 2, 4, 10, 1]. The primary motivation for using a temporal logic is that it enables declarative expression of I/O, destructive assignment, and other sorts of dynamic change. (Section 9 discusses several applications utilizing declarative I/O and change.) The present paper shows, in addition, that time can be used to structure, and possibly simplify the implementation of, traditional (bottom-up) logic programs.

A Starlog program consists of a set of timed and untimed program clauses. Untimed program clauses are identical in form and in execution to standard top-down Prolog clauses. Timed program clauses are distinguished from untimed clauses in that the head atom and zero or more of the body atoms have an additional distinguished argument, the timestamp, written as the second argument of the binary infix operator `@`. Timestamps are terms that must evaluate to natural numbers. A typical timed clause looks like

```
p(X)@T :- q(X,Y)@T, d(D), r(Y)@T-D.
```

Its declarative reading is roughly: "If `q(X,Y)` holds at time `T`, if `d(D)` holds, and if `r(Y)` holds at time `T-D`, then `p(X)` holds at time `T`." Timed program clauses are executed either top-down or bottom-up, depending on the presence or absence, respectively, of a declaration, such as `:- top_down p/1`, specifying that all clauses for a given head predicate symbol shall be handled top down. Thus, by default timed clauses are bottom-up.

The heads of bottom-up program rules represent facts to be made true, or (builtin) actions (e.g., output) to be performed, at a given time. Atoms in the body represent conditions to be checked. Conditions are facts derivable either by earlier bottom-up rules, by top-down calls, or by builtin environmental checks such as for the availability of data on an I/O channel. Crucially, clauses are causal: the bodies of bottom-up rules must refer to the past and may not refer to the future. (This will be made more precise in the next section.) Thus Starlog rules follow the "Imperative Future" paradigm [10].

The typical structure of a Starlog program is basically bottom-up: the outermost loop of execution, which interacts with the external world, is bottom-up and data-driven. (By analogy, in Haskell[6] or Mercury[16, 15], interaction with the real world is mediated by the outermost, singly threaded state of the world.) The reason for this is that in implementing the imperative future paradigm, Starlog uses the facts true at the current time, along with a record of some facts true in

the past, to compute the state of the world in the future. If one tried to execute Starlog top-down, say, by running a query `p(X)@100` or `p(X)@T`, then the system would have to backward chain on the rules and *guess*, recursively, which causes lead to the query's being true. This may result in a lot of search. Executed bottom-up, a Starlog program involves model construction rather than search. As a consequence, a Starlog program is basically data-driven, rather than goal driven. We envisage that in many applications, the data that will drive execution will be events, such as button presses, timer expirations, arrival of messages on a channel, and notification of completion of subtasks. The programmer will certainly need to be careful that the constructed model is not too large — that not too many irrelevant facts are derived. (A similar need for care is present for programmers using forward-chaining expert system languages.) So far, we have envisioned no need to implement magic sets [12], though we have not ruled out that option. Magic sets seem unwieldly to implement, analyze, and debug [1] and may be more appropriate for query-based deductive databases than for a (reactive) programming language like Starlog.

Nonetheless, we have found that it is most natural and efficient to specify that certain relations are to be evaluated top-down. Since we do not support magic sets, having top-down predicates simplifies control for those procedures that do need a goal-driven component. Furthermore, facts that are derived bottom-up are memoized (tabled), as in typical deductive databases, while facts derived top-down are not memoized. Since Starlog allows data structures as terms (unlike many temporal deductive databases), relations such as `append/3` and `sort/2` that create numerous intermediate data structures and that are likely used but once per input arguments, are probably best executed top-down. Lastly, rules that would need arithmetic constraints if executed bottom-up, due to unbound variables, can often be executed top-down without constraints. Consider, for example, a rule like the following from the elevator simulator of Section 9:

```
existsTimeElevatorHasJustStoppedIndicator(Floor,Tm10)@T:-
    elevatorHasJustStoppedIndicator(Floor)@U,
    Tm10< U, U =< T.
```

If executed bottom-up, the variable `T` would be unbound, and so the resulting tuple would involve an arithmetic constraint. But executed top-down, `T` gets bound. This implementation of Starlog[2] does not handle (nonground) arithmetic constraints.

Having both top-down and bottom-up control allows the programmer to take appropriate advantage of the complementary strengths of both Prolog and bottom-up deductive database technology. Sections 7 and 9 elaborate on this point.

4 Strongly-Stratified Programs

A *strongly stratified* program is one that does not contain any simultaneous recursion through bottom-up predicates: all recursion must occur in top-down relations, or be stratified through time. That is, if the call graph of the program contains a path from atom $p(A_1, \ldots, A_n)@T$ (or literal $\neg p(A_1', \ldots, A_n')@T'$) in the body of a rule to an atom $p(A_1', \ldots, A_n')@T'$ in the head of some rule, then either: p/n is executed top-down (by Prolog) or T must be strictly greater than T'.

In other words, within each time step, bottom-up computation is hierarchical (non-recursive). We emphasize that negation too must be strongly stratified.

[1] We base this opinion partly on our experience using the the Coral Deductive Database System [11, 17].

[2] A separate implementation does handle constraints.

A more formal definition of *strongly stratified* is that a program is strongly stratified if there exists a partial order $\prec$ over all the bottom-up predicate names (P/N) in the program such that, for every program clause

```
P@T <-- [not] P1@T1, [not] P2@T2, ...  [not] Pn@Tn.
```

(where the [not]'s indicate optional negation) it holds that

$$\forall\, i \in \{1, \ldots, n\} \;.\; T_i < T \vee T_i = T \wedge P_i \prec P \;.$$

Note that a strongly stratified programs can still contain recursion, but the recursion is restricted so that all recursive cycles of bottom-up predicate calls involve decreasing time.

Furthermore, the Starlog compiler must be able to statically derive the precedence relation. All predicates callable at a given time step must be ordered, statically, by the compiler, into strata such that if predicate q appears (positively or negatively) in the body of a bottom-up clause with head p, then q is in a strictly lower strata than p. This restiction on the structure of programs is by design. By analogy, just as Mercury places upon the programmer the obligation to write a well-moded program, so Starlog places upon the programmer the obligation to write a program for which the strong stratification is derivable by the compiler.

Is it reasonable to require all Starlog programs to be strongly stratified? We believe so, because a surprisingly large proportion of bottom-up programs are easy to write in a strongly stratified style. In addition, most programs that are not strongly stratified can be transformed into strongly stratified programs in a fairly straightforward way, or handled by a small extension to the pseudo-naive interpreter, as will be shown in Section 7. Furthermore, we believe that requiring programs to be strongly stratified is helpful from a correctness and software engineering point of view, as it forces the programmer to be explicit about termination and progress conditions of the code. Since bottom-up execution is hierarchical at a given time step, non-termination must result either from recursion through time, from top-down calls, or from infinite data.

A useful analogy is that just as Mercury supports the subset of Prolog programs that are well-moded (with statically identifiable modes and forms of nondeterminism), our pseudo-naive evaluation (efficiently) supports the subset of logic programs that are strongly stratified. Pseudo-naive evaluation itself is optimized to handle bottom-up hierarchical execution at each time step. We use the explicit timestamps or top-down calls to handle simultaneous recursion and to simulate the delta sets of semi-naive evaluation. We thereby obtain a more efficient, more data-driven implementation than naive evaluation. But the explicit timestamps are easier to implement efficiently than full delta-sets.

4.1 Compilation of Strong Stratification

It is tedious for programmers to have to specify the stratification partial order explicitly, so the first stage of our Starlog compiler infers it, by analyzing all the bottom-up clauses of the given program, then generating a suitable partial order. This is done by the following Prolog code in the compiler:

```
...
\+ (is_bottom_up_clause(Head@Time, Body),
    path(Body, Path, Constraints),
    member(Call@Ti, Path),
    \+ stratified(Call@Ti, Head@Time, Constraints)),
```

```
tsort,
...
```

Note that the Prolog code `\+(P, \+ Q)` is best read as *for all solutions to P, Q is true.* The `is_bottom_up_clause/2` predicate iterates through each of the bottom-up clauses in the Starlog program being analyzed. The `path/3` predicate converts Body to disjunctive normal form and for each disjunct, binds Path to the list of the timed calls within that disjunct and binds Constraints to a list of all the arithmetic constraints within that disjunct. For example, if Body is:

```
p(X)@T,
T1 is T-1,
(X = 0  ->  q(Y)@T1  ;  r(Y)@T2, T2 < T1)
```

then `path(Body,Path,Constraints)` will return two solutions:

```
Path = [p(X)@T, q(Y)@T1]
Constraints = [T1 = T-1, X = 0] ;
```

and

```
Path = [p(X)@T, r(Y)@T2]
Constraints = [T1 = T-1, T2 < T1] .
```

The `stratified(Call@Ti, Head@Time, Constraints)` predicate first tries to prove that `Ti<Time` follows from `Constraints` (it uses a simple theorem prover to do this). If it cannot prove this, then it proves that `Ti=<Time` follows from `Constraints`[3] and asserts an ordering tuple `is_before(P/N, H/M)`, where `P/N` and `H/M` are the outermost functors of `Call` and `Head`, respectively.

Finally, `tsort/0` performs a topological sort on the `is_before/2` tuples and generates an error if they contain any cycles, or flattens them into a total order over the predicate names otherwise. This total order is used by the pseudo-naive evaluator to determine the order of execution within each time step.

5 Pseudo-naive Evaluation

Pseudo-naive evaluation optimizes naive-evaluation by adding a data-driven component to computation — thereby overcoming the first source of inefficiency mentioned in Section 2 — without maintaining 'delta sets' δ_i. Section 7 discusses how we propose to overcome the second problem of naive evaluation, duplication of work, and how pseudo-naive evaluation can be used for programs involving simultaneous recursion.

For hierarchical or strongly stratified programs, delta sets provide no benefit, due to the lack of simultaneous recursion. However, naive evaluation would still be inefficient since naive evaluation is not data driven. Naive evaluation would try executing each program rule, whether or not new facts have been derived for atoms in the body of the rule.

Pseudo-naive evaluation adds a data-driven component to naive evaluation by maintaining, at each iteration i, a *token set* τ_i, rather than a delta set δ_i. A token set is an abstraction of a delta set. Token set τ_i consists of the set of relation names (predicate plus arity) which have received new facts at iteration i. A pseudo-naive interpreter uses the token set to skip over rules that cannot possibly fire, since no new facts have been derived for atoms in the bodies of those rules.

[3] It gives a warning if it fails to prove this, because it means that either the theorem prover is not strong enough, or the call is referencing future tuples which is a programming error.

```
:- mode pseudo_naive_evaluation(+).
pseudo_naive_evaluation([]).   % No more tokens.
pseudo_naive_evaluation([Token|RestTokens]):-
    setof(NewToken, enables(Token,NewToken), NewTokens),
    merge_tokens(NewTokens,RestTokens,Tokens),
    pseudo_naive_evaluation(Tokens).

:- mode enables(+,-).
enables(Token,NewToken):-
    rule(Token,Head,Body), % Index on Token
    Body,          % Try proving the body.
%In the actual implementation, Body is compiled as Prolog.
    \+Head,        % Make sure it's new.
    assert(Head),
    trigger(Head,NewToken).  % drive the data
```

Figure 2: Prolog Interpreter for Pseudo-naive Evaluation

Pseudo-naive Evaluation

$$
\begin{aligned}
S_0 &= \{H \mid (H \leftarrow \mathsf{true}) \in P\} \ \ (\text{facts}) \\
\tau_{i+1} &= \{p/n \mid \exists t_1, \ldots, t_n,\ p(t_1, \ldots, t_n) \in S_{i+1}\} \ \ (i \geq -1) \\
S_{i+1} &= S_i \cup \{H\mu \mid (H \leftarrow B) \in \mathit{Rename}(P) \\
&\qquad \wedge (\mathit{preds}(B) \cap \tau_i \neq \emptyset) \wedge (B\mu \subseteq S_i) \wedge \neg \exists j \leq i \ .\ H\mu \in S_j\} \ \ (i \geq 0) \\
S &= \cup_{i=0\ldots\infty} S_i
\end{aligned}
$$

In short, pseudo-naive evaluation is data-driven at the level of whole relations, but not at the level of each element of a relation.

An interpreter for pseudo-naive evaluation appears in Figure 2. According to the abstract definition of pseudo-naive evaluation in the previous paragraph, each τ_i (token set) contains a set of predicate names and arities p/n. As a refinement on that abstract definition, the interpreter implements the optimization in which τ_i contains a list of pairs, where each pair consists of a timestamp and the index of a rule to be tried. Section 8 describes several additional optimizations.

The procedure `merge_tokens(+,+,-)` sorts tokens according to the stratification order derived by the compiler. If an atom `p(T1,...,Tn)` appears as the trigger in the body of some program rule with indices I, for some arguments $T_1, \ldots, T_n$, then the fact `trigger(p(_,...,_),I)` must appear in the compiled program.

The next section elaborates on the compilation of triggers.

We note some similarity between our pseudo-naive interpreter and the techniques described in [19] for converting bottom-up rules into Prolog rules.

6 Compilation for Pseudo-Naive Evaluation

The most important aspect of compiling Starlog programs for the pseudo-naive evaluator is choosing one or more appropriate triggers for each clause. This is important, because clauses are only

evaluated when triggered, so the choice of trigger can affect both correctness (if a clause is not triggered at times when it might produce output, then outputs will be lost) and efficiency (tuples that occur infrequently make better triggers, because rules are evaluated less often).

Currently, our compiler automatically chooses a trigger for two simple cases, which cover the majority of clauses, and evaluates the remaining clauses at every time step. However, we also allow the programmer to annotate a clause to manually define one (or more) triggers for a clause, which overrides the compiler's choice. This is more flexible, but means that the programmer is responsible for completeness (choosing the wrong trigger may cause output tuples to be lost). When adding triggers by hand, we have occasionally found it useful to add several triggers to a single clause. It is not clear to us yet whether sophisticated triggering like this can be chosen automatically, or even checked for correctness by the compiler.

Given a clause, `H@T <-- Body`, (to simplify discussion, we assume in this section that all clauses are in Horn clause format) the two strategies used by the compiler are:

1. If `Body` contains some positive atom, $A@U$, such that $T - U$ is a constant natural number K (e.g. if T and U are identical, then $K = 0$), then the compiler uses A as the trigger, and evaluates the clause exactly K time units after any A tuple is created. The compiler does this by generating the following `rule/3` and `trigger/2` clauses (ID is a unique integer generated for each clause):

   ```
   rule(ID@T, Head@T, Body).
   trigger(A@U, ID@T) :- T is U+K.
   ```

   Note that there may be several atoms $A@U$ that satisfy this criterion. The choice between them does not affect soundness, but may affect efficiency. Ideally, the atom that is least frequent triggered should be chosen. Currently, we choose the left-most one that satisfies the above criterion, but it would be interesting to extend the compiler to use statistical information from previous runs as the basis for this choice.

2. If `Body` contains some positive atom, $A@U$, such that U is provably the latest time at which any other input tuples for this clause can arrive, then it is safe to trigger the clause at time U (it will produce outputs at time T, which may be in the future, but at least no outputs will be missed). The compiler generates :

   ```
   rule(ID@U, Head@T, Body).  % Note that Body contains A@U
   trigger(A@U, ID@U).
   ```

A disadvantage of this strategy is that some clauses may create many tuples in the future, which slows the system down.

7 Handling Simultaneous Recursion

There are at least three ways for a pseudo-naive interpreter to deal with programs that are not strongly stratified.

1. Make the culprit procedures top-down and use Prolog evaluation. For very many procedures — those efficiently executable by Prolog — this option will be entirely appropriate.

2. Make the procedure top-down and use tabling (OLDT resolution [14]), as in XSB [13]. This option will work for a larger class of programs (e.g., those with left recursion) and, unlike bottom-up computation without magic sets. has the advantage of being goal driven. Our implementation of Starlog runs also under XSB, and this option has been successfully used.

3. Add explicit timestamps to make the recursion strongly stratified, and use pseudo-naive bottom-up evaluation.

The last option, which we explain below, is effectively semi-naive evaluation, using the timestamped relations to hold the delta sets. We will illustrate these three methods via an example.

Consider the following program.

```
ancestor(A,D):- parent(A,D).
ancestor(A,D):- ancestor(A,Tmp), parent(Tmp,D).
```

Then the following query eventually loops using Prolog evaluation.

```
| ?-  ancestor(john,Who).
```

One could rewrite it to eliminate the left recursion. Alternatively, one used OLDT resolution (say, XSB) with `ancestor/2` declared to be tabled. Then both termination and efficiency would be achieved.

If we tried to use pseudo-naive evaluation with simultaneous recursion, the program would look like

```
ancestor(A,D)@T:- parent(A,D).
ancestor(A,D)@T:- ancestor(A,Tmp)@T, parent(Tmp,D).
```

The program would terminate correctly, but because `ancestor/2` tuples derived at time `T` immediately enable additional `ancestor/2` tuples to be derived at the same time `T`, pseudo-naive evaluation (like naive evaluation) would be inefficient. The lack of "delta" relations would result in each invocation of the body of the recursive clause at time `T` re-deriving all the tuples derived in earlier iterations at time `T`.

We are currently investigating another option, that preserves termination and efficiency. This is to strongly stratify the bottom-up recursion, by writing the program as follows:

```
ancestor_delta(A,D)@0:- parent(A,D).
ancestor_delta(A,D)@T:-
        ancestor_delta(A,Tmp)@T-1,   % trigger
        parent(Tmp,D),
        not(exists(U, (U<T,ancestor_delta(A,D)@U))).

ancestor(A,D)@T:- not(ancestor_delta(_,_)@T),ancestor_delta(A,D)@U,U<T.
```

At each time `T`> 0, the second clause is used to derive tuples that are newly enabled by tuples derived at time `T-1`. The negated call at the end of the recursive clause is needed to assure that a tuple `ancestor_delta(A,D)@U` derived at some time `U`$<$`T` in the past is not re-derived at time `T`. When a time `T` is reached when no more tuples can be derived, the union of the `ancestor_delta(A,D)@I` tuples for `I`$= 0, \ldots,$`T`-1 is the entire `ancestor/2` relation; hence, the last clause.

Now, for the above technique to be generally useful, a way is needed to *modularize* time, so that the code can be called at times other than `T=0`. For example, suppose the solutions to `ancestor/2`

are needed at some time T using the then current values of the parent/2 relation. We propose using the module system to isolate a local computation in its own 'timespace.' First, we place the code for ancestor/2 in its own module, with an interface definition that specifies parent/2 as input and ancestor/2 as output. One then invokes the module with a rule like

```
ancestor:parent(P,C)@T:- parent(P,C)@T,.....
```

This rule causes the parent/2 facts of time T to be bound as the parent/2 input tuples of the ancestor/2 procedure. The solutions are placed, at the calling module's time T, in a relation accessible in rule bodies by an atom like ancestor:ancestor(A,D)@T.

8 Other Features and Optimizations

Starlog implements a declarative form of state variable based on the from ... until operator of temporal logic. As in the event calculus [7], one specifies the initiating (terminating) conditions that cause a state variable to become true (false). For example, our elevator simulator contains the following rules

```
initiates(doorIsOpeningState)@T :-  doorHasStartedOpeningIndicator@T.
terminates(doorIsOpeningState)@T :- doorHasFinishedOpeningIndicator@T.
```

which say that when the elevator controller receives the specified indicator signal from the sensors, the doorIsOpeningState variable becomes true, and it remains true until the specified other indicator signal is received. The initiates/terminates idiom simplifies the coding of rules (e.g., frame axioms) involving persistent relations that would otherwise have to be written clumsily (and inefficiently) like

```
doorIsOpeningState@T+1:- doorHasStartedOpeningIndicator@T.
doorIsOpeningState@T+1:- doorIsOpeningState@T,not(doorHasFinishedOpeningIndicator@T).
```

Our system incorporates some optimizations that have not been described above. First, if every call to a nullary predicate $P/0$ is in a trigger position, it does not assert $P/0$ tuples at all, since the token that is inserted into the priority queue carries all the information that is needed.

In addition, the enables/2 predicate is implemented more efficiently than shown in Fig. 2, where rule bodies are treated as data. In reality, a rule that we have described as being compiled into:

```
rule(ID@U, Head@T, (P@U, Rest...).
```

is actually compiled into something like:[4]

```
rule(ID, U, Head@T) :- P@U, Rest... .
```

so that the body of the rule is compiled directly by the underlying Prolog compiler and so that first-argument indexing means that each token directly selects the correct triggered clause. These factors are some of the major reasons for achieving good speed and leverage from the underlying Prolog.

Our current implementation uses the dynamic database (assert/1) to store relations. This has the advantages of simplicity, transparency, and portability. The bodies of both bottom-up and

[4] In practice, all $P@T$ atoms are transformed so that the time argument T becomes an additional (first) argument of P. This improves indexing performance when atoms are looked up.

top-down clauses can be compiled and executed as normal Prolog rules, as discussed in the previous paragraph. Since our implementation is written in 'vanilla' Prolog, it is easy to port it to different Prolog implementations. Under XSB Prolog, we take advantage of user-specifiable indexing to specify additional indexing, where appropriate. In a language like Mercury that does not have `assert`, we could instead use data structures from a library package to store the relations.

Currently insertion of tokens into the priority queue (procedure `merge_tokens` of Fig. 2) is done in Prolog. Since tokens consist of pairs of integers (time plus rule index), management of the queue could easily be done instead by a foreign language procedure written in, say, C.

Garbage collection is important for Starlog since in the abstract bottom-up semantics, nothing is ever forgotten. The programmer can add declarations specifying when bottom-up tuples may be safely garbage collected. For example, the declaration

```
:- persists(elevatorHasJustStoppedIndicator/1,12).
```

says that tuples of the indicated bottom-up relation can be garbage collected after 12 time steps. We have yet to address automatic compile-time or runtime determination of when data can be garbaged collected. We suspect though, that some version of our current scheme will be necessary for complex relations.

9 Applications

We briefly describe three medium-size applications that run successfully in Starlog.

Elevator simulator This program implements a fully declarative simulator for a single elevator, with graphical animation and user input. For a specification of the simulator, see `<http://www.cs.waikato.ac.nz/~dsmith/CHALLENGES/details.html>`.

The animation shows, on the left, the position of the elevator. In the middle it shows unserviced call requests at each floor ("up" or "down"). On the right it shows (with a "+" sign) unserviced floor requests that result from passengers selecting a floor to go to.

There are, in effect, six processes that run concurrently:

1. The elevator controller itself (generates control signals, based on values of indicator signals, time delays, and global state variables).
2. The door motor (reacts to door control signals, generates door indicator signals)
3. The main motor (reacts to main motor control signals, generates main motor indicator signals)
4. Elevator-motion-detectors (generates indicator signals)
5. The animation process.
6. The passenger simulator.

There are two versions of the last process. One version randomly generates button presses, thereby simulating passengers. The second version allows asynchronous, external input of button presses by mouse. Currently, all animation is done with ASCII curses-style character graphics.

For reasonable parameters, our pseudo-naive interpreter executes the elevator simulation approximately four times faster than an earlier interpreter that used naive evaluation. Exactly how much faster it runs depends on the density of events: the longer the average interval between button presses, the more benefit accrues from data-driven execution.

graphplan [3] robot planner Space limitations preclude a full description of this application. The planning graph and its add, delete, and mutual exclusion edges are all represented relationally. Using relations instead of data structures has the advantage that data is more

accessible. Performance depends critically on the underlying Prolog's ability to do indexing. On some examples graphplan ran up to five times faster using XSB's flexible indexing declarations as compared to using XSB without extra indexing. The backward phase of the graphplan algorithm is executed top down, by virtue of a `top_down` declaration. It is easy to add heuristics; on some examples, we could thereby significantly outperform the original C-based implementation of graphplan.

Ecology simulation . This application is an individual-based ecology simulation with ASCII graphics animation. Each specie has its own subwindow showing population sizes at each location. For each specie one specifies its predators, preys, death rate (depending on hunger and age), initial population size, and moving speed. The code requires aggregates, randomness, and indeterministic choice. Using several hundred individuals, each 'generation' takes about a second running on a SPARCserver-1000.

Of these three applications, only the first benefits significantly from pseudo-naive evaluation: the last two would run about as efficiently (but not significantly more efficiently) using naive evaluation, since nearly every rule fires during every time step.[5] Indeed, naive evaluation is quite adequate for many procedures. This is symptomatic of a general phenomenon in high-level languages: the high-level features (e.g., unification, backtracking, delta sets, data-driven execution) are used in restricted contexts and in restricted ways. A well-designed language optimizes the common patterns of use. We think that in a typical application, the outermost 'event loop' will be bottom-up and data-driven (i.e., reactive), while much of the algorithmic logic of the code will be top-down, using Prolog data structures, backtracking, and unification, as needed.

10 Conclusion

We have presented a novel implementation technique, pseudo-naive evaluation, that combines the advantages of bottom-up and top-down logic programming. The essential idea is to prohibit simultaneous bottom-up recursion by requiring all recursion to be either top-down or mediated by a temporal stratification. Pseudo-naive evaluation optimizes execution of such programs by introducing a data-driven component to naive evaluation, without maintaining the explicit 'delta sets' of semi-naive evaluation. Pseudo-naive evaluation combines the simplicity of naive evaluation with much of the data-driven reactivity of semi-naive evaluation. By construction, pseudo-naive evaluation will be as efficient as semi-naive evaluation on strongly-stratified programs. A reactive, data-driven program will benefit significantly from pseudo-naive evaluation.

Areas of future work include

- automatic compilation of complex triggers
- garbage collection (automatic derivation of `:- persists` declarations)
- program analysis to support the above two
- modular time, using nested timestamps or local timespaces.

Acknowledgements

We thank John Cleary and Lunjin Lu for their considerable input into this research effort.

[5] In the language of Section 6, the rules have 'every_time' triggers.

References

[1] M. Abadi and Z. Manna. Temporal logic programming. *Journal of Symbolic Computation*, 8(3):277–295, September 1989.

[2] M. Baudinet, J. Chomicki, and P. Wolper. Temporal deductive databases. In Tansel et al. [18].

[3] A. Blum and M. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[4] J. Chomicki. Temporal query languages: A survey. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic: First International Conference*. Springer-Verlag, LNCS 827, 1994.

[5] J.G. Cleary. Colliding pucks solved in a temporal logic. In *Proc. Distributed Simulation Conference*, 1990.

[6] P. Hudak and P. Wadler et al. Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *SIGPLAN Notices*, March 1992.

[7] Robert Kowalski and Fariba Sadri. The situation calculus and event calculus compared. In Maurice Bruynooghe, editor, *Logic Programming - Proceedings of the 1994 International Symposium*, pages 539–553, Massachusetts Institute of Technology, 1994. The MIT Press.

[8] M. Liu and J.G. Cleary. Declarative updates in deductive databases. *Journal of Computing and Information*, 1:1435–1446, 1994.

[9] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.

[10] Mehmet A. Orgun and Wanli Ma. An overview of temporal and modal logic programming. In D. M. Gabbay and H. J. Ohlbach, editors, *Temporal Logic: First International Conference*. Springer-Verlag, LNCS 827, 1994.

[11] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of the International Conference on Very Large Databases*, 1992.

[12] Raghu Ramakrishnan, Divesh Srivastava, and S. Sudarshan. Efficient Bottom-up Evaluation of Logic Programs. In J. Vandewalle, editor, *The State of the Art in Computer Systems and Software Engineering*. Kluwer Academic Publishers, 1992.

[13] K. Sagonas, T. Swift, and D.S. Warren. XSB as an Efficient Deductive Database Engine. In *SIGMOD*, 1994.

[14] T. Sato and H. Tamaki. Old resolution with tabulation. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science, Berlin Heidelberg, 1986. Springer-Verlag.

[15] Z. Somogyi. *A Parallel Logic Programming System Based on Strong and Precise Modes*. PhD thesis, University of Melbourne, 1989.

[16] Z. Somogyi, F. J. Henderson, and T. C. Conway. Code generation for Mercury. In J. Lloyd, editor, *Logic Programming, Proceedings of the 1995 International Symposium*, Portland, Oregon, USA, 1995. MIT Press.

[17] Divesh Srivastava, Raghu Ramakrishnan, S. Sudarshan, and Praveen Seshadri. Coral++: Adding object-orientation to a logic database language. In *Proceedings of the International Conference on Very Large Databases*, 1993.

[18] A. Tansel, J Clifford, S. Gadia, S. Jajodia, A. Segev, and R. Snodgrass (editors), editors. *Temporal Deductive Databases: Theory, Design, and Implementation.* Benjamin/Cummings, 1993.

[19] J. Wunderwald. *Adding Bottom-up Evaluation to Prolog.* PhD thesis, Technical University of Munich, 1996.