Proceedings of the Workshop on Practical Implementations and Systems Experience in Logic Programming

E. Tick (Editor)

CIS-TR-93-12
June 1993

Abstract

The *Workshop on Practical Implementations and Systems Experience in Logic Programming* was held on June 25 1993 in conjunction with the International Conference on Logic Programming in Budapest, Hungary. The workshop was devoted to *implementing* logic programming languages, both sequential and parallel, and related languages such as functional and object-oriented languages. The emphasis was on real-life experience with implementations: compilation, optimization, runtime kernels, performance and systems building tools, empirical performance analysis, and benchmarking.

DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE
UNIVERSITY OF OREGON

Table of Contents

Carpe Diem (Some Implementation Aspects of Turbo Erlang)
B. Hausman
Ellemtel Telecommunications Systems Laboratories

From Modules to Structured Theories and Logic Objects: a Common "Program-based" Run-time Support
E. Denti, A. Natali and A. Omicini
DEIS — Università di Bologna

SelfLog: Language and Implementation
M. Bugliesi and G. Nardiello
Università di Padova

To Trim or not to Trim?
J. Noyé
European Computer-Industry Research Center (ECRC)

MProlog: an Implementation Overview
Z. Farkas, P. Szeredi, and P. Köves
IQSOFT (SZKI Intelligent Software Ltd.), Budapest

Issues in Implementing ACE: A Stack Copying Based And-Or Parallel System
E. Pontelli, G. Gupta and M. V. Hermenegildo
New Mexico State University and Universidad Politecnica de Madrid

Extension of Strand with Linda-Like Operations: Implementation and Performance Study
G. Czajkowski and K. Zieliński
University of Mining and Metallurgy, Kraków

Call Forwarding: A Simple Low-Level Code Optimization Technique
S. K. Debray, K. De Bosschere and D. Gudeman
The University of Arizona and Universiteit Gent

A Flexible Scheduling and Memory Management Scheme for Non-Deterministic And-parallel Execution of Logic Programs
K. Shen and M. Hermenegildo
Bristol University and Universidad Politecnica de Madrid

Memory Management in Quintus Prolog
J. Crammond and T. Lindholm
Quintus Corporation, Palo Alto

Implementing Logic Programming Systems: The Quiche-Eating Approach
S. K. Debray
The University of Arizona

Concurrent Logic Programs *a la Mode*
E. Tick, B. C. Massey, F. Rakoczi, and P. Tulayathun
University of Oregon

Better Late Than Never
M. Meier
European Computer-Industry Research Center (ECRC)

Carpe Diem[1]
(Some Implementation Aspects of Turbo Erlang)

Bogumił Hausman
Computer Science Laboratory
Ellemtel Telecommunications Systems Laboratories[2]
Box 1505, S-125 25 Älvsjö, Sweden
email: bogdan@erix.ericsson.se

Abstract

Erlang is a concurrent programming language designed for prototyping and implementing reliable real-time systems. Erlang is used in a number of experimental telephony applications both within and outside Ericsson [1]. Our sequential implementation of Erlang where Erlang programs are compiled into the C language (the implementation is called Turbo Erlang) was described in [6]. In this paper we treat some implementation aspects of the Turbo Erlang system which were not treated earlier in [6].

1 Introduction

Erlang is a concurrent programming language designed for prototyping and implementing reliable real-time systems. Erlang was developed at the Computer Science Laboratory, Ellemtel Telecommunications Systems Laboratories [3, 4]. Erlang provides support for programming concurrent applications, a special syntax for referring to time (time-outs), and explicit error detection capabilities. Some basic elements of Erlang programming are presented in the following sections. Erlang in its design inherits some ideas from concurrent logic programming languages. It does not allow destructive assignment of variables and uses pattern matching for variable binding and function selection.

In our recent paper [6] we have presented an efficient and portable sequential implementation of Erlang (called Turbo Erlang), where Erlang programs are compiled into the C language. Portable means that the implementation can be ported to any processor having a GCC compiler. We have chosen GCC to allow global register allocation and usage of labels as first-class objects, which in turn simplified return address administration, and generation of unique labels.

The Turbo Erlang system performs very well in comparison with other high-level language implementations like SICStus Prolog (compiling to native code, [7]), jc (Janus compiled to C, [5]), and the JAM Erlang implementation (Erlang compiled into instructions for a virtual machine, which are interpreted by an emulator, [2]).

Comparing further to Janus, in our system each Erlang module is compiled into a C function residing in a separate file and the module functions can be accessed from other modules with no overhead. In the Janus system all programs are compiled into one C procedure which does not allow for large-scale applications.

[1] "seize the day" Horatius (1:11)
[2] a company jointly owned by Ericsson and the Swedish PTT

The comparison with C shows that the performance of Turbo Erlang is not far from the performance of highly optimized C code [6]. We expect that after applying some compile-time optimizations we will be able to run large applications in Erlang as quickly as the same applications written from the beginning in C or C++.

For further details on the performance evaluation and comparison with other high-level language implementations see [6].

In this paper we describe some implementation aspects of Erlang which were not treated in [6], for example error recovery, catch and throw and compiling Erlang control structures. To make the paper self-contained we repeat some parts already described in [6].

The paper is organized as follows. Erlang is introduced in Section 2. In Section 3 we present the turbo Erlang abstract machine (TEAM). Section 4 describes generation of the C code. The final conclusions are presented in Section 5.

2 The Erlang Language

2.1 Data Objects

An Erlang term is either a constant (integer, float, atom, process identifier), compound term, or a variable. A compound term is either a tuple or a list. Tuples are used for storing a fixed number of data objects: `{T1,T2,...,Tn}` is a tuple of size `n`. A list is either the empty list `[]` or a pair `[H|L]`.

Erlang does not allow destructive assignment of variables, and the first occurrence of a variable is its binding instance (i.e. all Erlang terms are ground).

2.2 Function Evaluation

Erlang programs are composed of functions. The definition of a function consists of a group of clauses each having a head, an optional guard and a body.

The Erlang evaluation mechanism is based on pattern-matching directed invocation. After a call of a function, the call is matched sequentially against heads of the clauses defining the function (in their textual order). After a successful match the clause guard if defined (a guard consists of a simple test or a sequence of simple tests) is evaluated and if the evaluation succeeds the clause body is chosen for execution. If no rule matches the call an error is generated. Expressions in the selected clause body are evaluated sequentially.

2.3 Modules

The module system in Erlang is based on function modularity, i.e. it works by limiting visibility of the functions contained within a given module. Functions which are to be evaluated from outside the module must be explicitly exported. A function can be called from other modules using a name qualified by a module name containing the function.

2.4 Case Expression

The case expression allows choice between alternatives within the body of a clause:

```
case Expr of
    Pat1 [when Guard1] -> Seq1;
    Pat2 [when Guard2] -> Seq2;
    ...
    PatN [when GuardN] -> SeqN;
end
```

First the expression `Expr` is evaluated and then the result is sequentially matched against `Pat1,...,PatN`. After the successful match the corresponding action is taken (respectively `Seq1,...,SeqN`). If no pattern matches the evaluation result an error is generated.

2.5 Multi-Process Programming

Support for multi-process programming in Erlang consists of the following primitives: `spawn` for starting a concurrent process, `send` for sending a message to a process, and `receive` for receiving a message from a process.

The primitive `spawn(Module,Func,[Arg1,...,ArgN])` starts a concurrent process which evaluates the function `Module:Func(Arg1,...,ArgN)`, and returns a process identifier which can be used to communicate with the process.

To send a message `Msg` to an Erlang process `Pid` the following syntax is used:

```
Pid ! Msg
```

where `Pid` must be a valid process identifier, and `Msg` can be any Erlang term.

To receive messages from different concurrent processes Erlang provides the syntax:

```
receive
    Pat1 [when Guard1] ->
        Action1;
    Pat2 [when Guard2] ->
        Action2;
    ...
[after
    Time ->
        ActionTimeOut]
end
```

which causes the current process to suspend until a message is received which matches one of the patterns `Pat1,Pat2,...` or until an optional time-out occurs.

The Erlang interprocess communication mechanism allows messages to be received in a different order to which they were sent (i.e. receive has the selective and buffering characteristics).

2.6 Error Recovery

Since telephony applications programming requires error recovery in the event of an unplanned failure, Erlang provides explicit error detection capabilities.

Erlang processes can be linked (links are bidirectional) to other communicating or cooperating processes. Upon abnormal termination (a run-time error) a process sends a special exit signal to all the currently linked processes. A default action is that upon receiving an exit signal a process terminates and propagates the exit signal to other linked processes.

Another possibility is that a process can receive exit signals within a receive statement and perform any required action before termination:

```
receive
    Pat1  [when  Guard1]  ->
        Action1;
    ...
    {'EXIT',PidN,MsgN}   ->
        ActionN;
[after
    Time  ->
        ActionTimeOut]
end
```

where the message `MsgN` from process `PidN` is an explicitly received exit signal.

2.7 Catch and Throw

The Erlang catch and throw mechanism can be used for a non-local return from a function, and for protecting bad code. To illustrate its usage let us analyse the execution of the following Erlang function:

```
foo(1) -> hello;
foo(2) -> throw({myerror,abc});
foo(3) -> 1 div 0;
```

Calling a function `foo(1)` results in the atom `hello`. When we call `foo(2)`, since there is no catch, the current process exits and sends exit signal to all linked processes. When we call `foo(3)`, an error is detected and the process exits sending the exit signal `{'EXIT',Pid,bad_arith}` to the linked processes.

If the `foo/1` function is called within a catch:

```
demo(X)  ->
    case  catch  foo(X)  of
        {myerror,Args}   ->
                    {user_error,Args};
        {'EXIT',Pid,What}   ->
                    {caught_error,What};
        Other  ->
                    Other
    end.
```

calling `demo(1)` results in the atom `hello`, calling `demo(2)` results in a tuple `{user_error,abc}` since `throw` jumps to the nearest catch, and `demo(3)` results in `{caught_error,bad_arith}` since the exit signal `{'EXIT',Pid,bad_arith}` is caught by the catch.

2.8 Code Management

Erlang supports dynamic loading of code (dynamic means that the code is loaded and linked at run time), and the use of multiple versions (currently two) of the same module. Loading a new module version means that all new calls to the module are done to the new version while all processes already executing the old version continue to do so. It is possible to remove the old version when there are no processes executing it, and Erlang provides primitives for checking if there are any processes executing a given version of code.

The above features (dynamic loading, multiple versions of code) are required in telephony applications used to control telephony hardware which cannot be stopped for software updates.

3 The Turbo Erlang Abstract Machine (TEAM)

The construction of the turbo Erlang abstract machine (TEAM) has been influenced by ideas behind the conventional WAM [8, 9] and the Janus Virtual Machine (Janus compiled to C) [5]. Differences come from the fact that Erlang has a functional nature, and has fully ground terms, and has explicit notions of concurrency and time.

3.1 Data Objects

An Erlang term is represented by a 32-bit unsigned word containing a value and a tag. The tag (4-bit) resides in the least significant bits and distinguishes the type of the term.

The value part of an atom is an index into a global atom table where the atom is represented. The value part of an integer is the integer itself. The value part of a list is a pointer to two consecutive heap locations with two tagged objects (the head and tail of the list). The value part of a tuple is a pointer to a heap object containing tuple size followed by the tuple elements. The value part of a float is a pointer to a heap object containing a 2-word float value. The value part of a process identifier is the process identifier itself.

3.2 Data Areas

The data areas are the code area, containing loaded compiled C code, and a stack and heap. The stack contains call frames with local variables and return addresses. The heap contains terms created by the Erlang execution.

The stack and the heap are allocated as one memory area and grow towards each other. Having the heap and the stack allocated together makes testing for memory overflow very efficient as we compare two pointers (top-of-heap pointer, and top-of-stack pointer) residing in hardware registers.

Each call frame on the stack starts with return address followed by local variables which are accessed by integer offset from a pointer to the top of the stack. Frames are allocated only after a clause guard is evaluated and if the clause body contains local variables and function calls. Frames are allocated and discarded by the macros `Allocate(N)` and `Deallocate(N)` with explicitly given frame size `N`. As we are using GCC a return address is a stored address to a label in the generated C code and return from an Erlang function is simply a goto instruction to the given label. For garbage collection purposes the stack structure can be obtained by looking for stored return addresses represented as unsigned integers and thus having the two last bits set to 0. To distinguish the return addresses from other data objects the data objects do not use tags having the two last bits set to 0.

Our garbage collection algorithm is a simple stop-and-copy one. After each garbage collection the total size of the heap and stack area is dynamically adjusted to follow the execution requirements, i.e. it grows or shrinks.

3.3 Registers

The TEAM uses a set of registers declared as C global variables. Some of the variables are declared as global register variables (this feature is provided by GCC in contrast to cc). We have the following registers: top-of-heap pointer, top-of-stack pointer, return-address pointer (pointer where to go when a function is ready executed), and argument registers (to pass function parameters).

3.4 Arithmetic

When arguments have mixed type or their type is unknown at compile time we call a general purpose C procedure to do the arithmetic. When arguments are integers the basic arithmetic operations are directly compiled into C code.

Type tests and term comparisons can reside in a guard or a body part of an Erlang clause. The corresponding conditional instructions contain a label stating where to go when the condition is not met. The labels in guard parts point to the next clause to be tried or to an error action if there are no clauses left. The labels in body parts point to an error action because a called Erlang function should not fail.

Term types are provided at compile time by guard type tests (e.g. `integer(Term)`, `atom(Term)`), since only terms with the required type can pass the guards at run time.

3.5 Functions

To call an Erlang function we pass parameters in a WAM-like style loading argument registers `x(N)`, and update the return-address register. In order not to destroy the argument registers all guard operations and tests are performed in temporary registers. On function return the return value is stored in `x(0)`.

3.6 Concurrency

Erlang processes are dynamically spawned and killed during the execution. Each process has its own heap and stack area. For concurrency purposes the TEAM provides suspension and scheduling mechanisms. A suspending process stores its current state in its suspension record and is added to a scheduler queue. To guarantee a fair scheduling a process is suspended after a fixed number of reductions and then the first process from the queue is resumed.

To receive messages each process has its own local message queue. Sending a message results in copying the message into the receiver heap and storing the message reference in the receiver message queue. While waiting for messages to be received a process is swapped out, and is added to the scheduler queue only when a new message is received (i.e. the addition is done by a process sending the message), or a time-out occurs.

3.7 Error Recovery

Each Erlang process has its local list containing all processes the process is linked to. As mentioned earlier upon abnormal termination (a run-time error) a process sends a special exit signal to all the currently linked processes. Sending the signal means that all the linked processes (residing in the scheduling queue and waiting to be swapped in) have their resumption addresses updated to execute the code responsible for exiting. A process upon receiving an exit signal propagates the signal to other still alive linked processes.

If the linked processes are to receive the exit signal within a receive statement, the signal is sent as an ordinary message.

3.8 Catch and Throw

When a process executes a *catch*, the catch resumption address is saved in a local frame on the stack. Upon exiting or executing *throw* the stack is searched for a saved resumption address and the execution continues there. To know if a process executes within a catch there is a process specific counter of saved catch resumption addresses. An example of the C code corresponding to a catch is shown later in Section 4.5.

4 Generation of C Code

Using GCC and thus using labels as first-class objects allows us to structure the generated C code in a very flexible way. In GCC labels can be saved in global data structures so the TEAM execution can jump into different C procedures without calling the procedures themselves and thus avoiding the C procedure call overhead. The only requirement is that the TEAM uses only global or static declared data objects during the execution.

4.1 Global Data Objects

The generated C code accesses a set of global variables (some of them residing in hardware registers) corresponding to the TEAM registers. There is a global atom table (containing representation of atoms), global module table (containing some Erlang module specific information), and a global function table (containing addresses of exported Erlang functions, the addresses at this stage are labels in the generated C code).

4.2 Modules

Each Erlang module is compiled into a C function residing in a separate file. The C function consists of two parts: an initialization part, and a code part.

The initialization part is responsible for updating the global atom table, exporting some local Erlang functions, fetching addresses of functions from other modules, and updating the global module table. The initialization part ends with the C return instruction and is executed when the module is loaded.

The code part consists of the C code corresponding to compiled Erlang functions. The functions are accessed by goto instructions where the required label is taken from the global function table. Calls (jumps) to functions in the same module are done directly. Labels to functions in other modules are fetched from the global function table at load time. As the fetching is done at load time all functions residing in other modules can be accessed with no overhead when the module code is executed.

4.3 Functions

To illustrate the C code corresponding to an Erlang function let us analyze `nrev/1`:

```
nrev([H|T])  ->  append(nrev(T),[H]);
nrev([])  ->  [].
```

the corresponding C code (`x(0)`, `x(1)` are argument registers, `y(0)` is a local variable; the code consists of C macros; labels written in italic are the macros C code) is:

```
nrev_1:                                 label acting as function address
    Clause;                             {__label__ next; locally declared label
    TestNonEmptyList(x(0),next);        if x(0) is not a non-empty list goto next
    Allocate(1);                        check memory overflow, allocate stack frame,
                                                save return address register
    GetList2(x(0),y(0),x(0));           get head y(0) and tail x(0) of list x(0)
    Call(nrev_1,1);                     check for context switch, set new return
                                                address, goto nrev_1,
                                                on return x(0) contains return value
    TestHeap(2);                        check memory overflow
    PutList2(x(1),y(0),NIL);            create a list x(1) with head y(0) and tail NIL
    CallLast(append_2,2,1);             get return address, deallocate stack frame,
                                                check for context switch, call append_2
    ClauseEnd;                          next:}

    Clause;                             {__label__ next;
    TestNil(x(0),next);                 if x(0) is not NIL goto next
    Return;                             goto return address, return value in x(0)
    ClauseEnd;                          next:}

    ErrorAction(FunctionClause);        goto error handler, indicate kind of failure
```

The above code uses some GNU extensions to the C language, i.e. there are locally declared labels which simplifies generation of unique labels. A locally declared label is used as well inside the `Call(append_2,2)` macro to generate a new return address:

```
#define  Call(LBL,Arity)
    ({__label__  ret;                   locally declared label
    RetAddr  =  &&ret;                  set new return address
    Dispatch(LBL,Arity);                check for context switch
    goto  LBL;                          goto to called Erlang function
    ret:  })
```

where `&ret` is the address of label ret and is one of the GNU extensions to the C language, `RetAddr` is the return-address register pointer.

4.4 Case Expression

If the case expression previously introduced in Section 2.4 gets the form:

```
call_case(N)   ->
    case  N  of
        1  -> 10;
        2  -> 20;
        N  -> N
    end.
```

the corresponding C code (`x(0)` is an argument register; the code consists of C macros; parts written in italic are the macros C code) is:

```
call_case_1:                                          label acting as function address
     Case;                                            {__label__ ret;
     TestPattern(                                     {__label__ next;
         Equal(x(0),make_integer(1),next);            if x(0) is not 1 goto next;
         PutInt(x(0),10);                             set x(0) to 10; goto ret;
                                     );               next:}
     TestPattern(                                     {__label__ next;
         Equal(x(0),make_integer(2),next);            if x(0) is not 2 goto next;
         PutInt(x(0),20);                             set x(0) to 20; goto ret;
                                     );               next:}
     TestPattern(;);                                  {__label__ next; goto ret; next:}
     CaseEnd;                                         ErrorAction(CaseClause);
                                                      ret:}
     Return;                                          goto to return address
```

where:

```
#define  TestPattern(Test_Action)
    {__label__   next;
        Test_Action;
        goto  ret;
    next:}

#define  Case  {__label__  ret

#define  CaseEnd
    ErrorAction(CaseClause);
    ret:}
```

Notice that the empty `TestPattern(;)` is expanded into `goto  ret`, since the return value `N` resides already in `x(0)`.

The above C code illustrates an extensive use of locally declared labels to generate the required flow of control.

4.5 Catch and Throw

Another example of the generated C code is the following Erlang program containing a catch expression previously discussed in Section 2.7:

```
protect_bad_code(X)   ->
    catch  foo(X).
```

the corresponding C code is:

```
protect_bad_code_1:                 label acting as function address
    Allocate(1);                    check memory overflow, allocate stack frame,
                                           save return address
    Catch(y(0));                    {__label__ ret;
                                           increment catch counter, save resumption
                                           address ret in stack frame as y(0)
    Call(foo_1,1);                  call foo/1
    CatchEnd(y(0));                 decrement catch counter, clear resumption address
                                    ret:}
    Deallocate(1);                  get return address, deallocate stack frame
    Return;                         goto return address
```

where:

```
#define  Catch(Y)
    {__label__  ret;
    c_p->catches++;                 increment catch counter
    Y  =  make_catch(&&ret)         save catch resumption address ret in stack frame

#define  CatchEnd(Y)
    c_p->catches--;                 decrement catch counter
    make_blank(Y);                  clear catch resumption address
    ret:}
```

The catch resumption address is stored in the stack call frame, and is searched up when a thrown is executed, or an exit signal is received. Since it is possible to have many nested catch constructions, when a catch is executed the corresponding catch resumption address is cleared. Each process has its own counter of pending catches, `c_p->catches`, to know if there are catch resumption addresses to be found on the stack.

4.6 Code Management

A newly compiled Erlang module (residing in a separate file) can be dynamically loaded at run time. First the compiled C code is linked to the executing C code and then a C procedure corresponding to the new module is called. Calling the procedure activates its initialization part which in turn updates entries in the global function table corresponding to the module Erlang functions.

If there is already an old loaded version of the module, loading the new version means that all new calls to the module are done to its new version while all processes already executing the old code continue to do so (the old code is still loaded but cannot be accessed through the global function table).

As mentioned earlier it is possible to remove the old code when there are no processes executing it (we free the memory block containing the code). The information about address area a code occupies is saved in global module table at load-time. To know if there are any processes executing the code, all processes have their stacks checked for stored resumption addresses pointing into the code address area.

5 Conclusions

We have presented an efficient and portable sequential implementation of Erlang (called Turbo Erlang), where Erlang programs are compiled into the C language.

The presentation includes some implementation aspects not treated in our previous paper [6]: error recovery, catch and throw, compiling Erlang control structures, checking for code removal. The proposed solutions can be of interest to people who work on practical implementation of other high-level languages.

The presented solutions illustrate that the GNU extensions to the C language (using labels as first-class objects) make it easy to generate a required flow of control, and simplify structure and access to the final object code.

Acknowledgements

We would like to thank Robert Virding and Mike Williams for discussions considering implementation aspects of Erlang. The C level dynamic link editor was written at our laboratory by Torbjörn Törnkvist. We are also grateful to Hans Nilsson for his comments on earlier drafts of this report.

References

[1] Ahlberg, I., Danne, A., and Bauner, J-O., *Prototyping Cordless Using Declarative Programming*, XIV International Switching Symposium, Yokohama (1992).

[2] Armstrong, J. L., Däcker, B. O., Virding, S. R., and Williams, M. C., *Implementing a Functional Language for Highly Parallel Real Time Applications*, in Proceedings of SETSS 92, Florence (1992).

[3] Armstrong, J. L. and Virding, S. R., *Programming Telephony*, in *Strand - New Concepts in Parallel Programming* by Foster, I. and Taylor, S., pp. 289-304, Prentice Hall (1990).

[4] Armstrong, J. L., Virding, S. R., and Williams, M. C., *Concurrent Programming in Erlang*, Prentice Hall (1993), to be published.

[5] Gudeman, D., De Bosschere, K., and Debray, S., *jc: An Efficient and Portable Sequential Implementation of Janus*, in Proceedings of the Joint International Conference and Symposium on Logic Programming 1992, pp. 399-413, MIT Press (1992).

[6] Hausman, B., *Turbo Erlang: An Efficient Implementation of a Concurrent Programming Language*, submitted to International Logic Programming Symposium 1993.

[7] *SICStus Prolog User´s Manual, Version 2.1 #6*, Swedish Institute of Computer Science (August 1992).

[8] Warren, D.H.D., *An Abstract PROLOG Instruction Set*, SRI Technical Note 309 (October 1983).

[9] Warren, D.H.D., *PROLOG Implementation and Architecture*, Tutorial notes from the 3rd International Logic Programming Conference, London (1986).

From Modules to Structured Theories and Logic Objects: a Common "Program-based" Run-time Support

Enrico Denti, Antonio Natali and Andrea Omicini
DEIS – Università di Bologna
viale Risorgimento, 2
40136 Bologna - Italy
{enrico, natali, andrea}@deis33.cineca.it

Abstract

This work presents the main features of CSM, a contextual logic programming system (fully compatible with the SICStus Prolog programming environment) which aims at supporting evolutive software development methodologies in logic programming. Rather than fixing a set of policies, CSM provides mechanisms for the (dynamic) configuration of structured theories, handled as first class objects. CSM mechanisms allow code reusability and sharing, and provide binding rules fully supporting object-oriented programming techniques, without undermining logic programming declarativeness. The implementation is a peculiar CSM aspect, since contexts are represented by descriptors (similar to SICStus module descriptors), configured at context creation time, with no modification at abstract machine level. While this choice implies relatively high costs for context creation, it prevents any semantics distortion or computational overhead with respect to standard (SICStus) Prolog programs, and allows users to exploit open, evolving software components at a cost comparable with that of more static approaches to program composition.

1 Introduction

The modern approach to software design, development and maintenance, demands new models of application building where the traditional notion of phase-refinement is replaced by that of *software evolution* [4]. A well-known example of evolutionary system development is *prototyping*, whose effectiveness crucially depends on the possibility of changing prototype behavior with less effort than would be required to modify the production software.

When traditional (imperative) languages are used to build the final product, higher-level languages and special-purpose tools are required in order to deal with prototypes. In such a case, a prototype is usually intended to be only a partial representation of a system that can be used as an aid to analysis and design, rather than as a production tool.

Such a reductive view of prototyping seems to be intrinsically overcome when applications fit the area of logic programming, since logic languages can greatly reduce the distance between specification and coding as well as between prototype description and system implementation. However, in order to fully support software development techniques based on software evolution, logic languages have to be enhanced with suitable constructs for programming-in-the-large.

Among the proposals to extend logic programming towards modularity and object-oriented software modeling, *contextual programming* [8] aims at introducing a set of general purpose abstractions and mechanisms rather than prefixed knowledge structuring models and policies.

A context is defined as the *composition* of a set of theories (also called modules or *units*), each identified by a name and constituted by a collection of Horn clauses. Since contexts can be specified in a static way or dynamically created through *extension goals* such as u>>g, the contextual programming model provides a uniform approach to both static and dynamic knowledge composition and object-oriented programming. In fact, the extension goal can be interpreted as a request to execute a method (goal g) in the object obtained by pushing unit u on top of the object constituted by the current context. Moreover, when they are interpreted as structured *binding environments*, contexts can support a number of different *binding policies*, each leading to a different component architecture. Using the terminology reported in [7], *local* or *eager binding* mechanisms lead to conservative, closed worlds (where each structured component is self-contained, i.e. its behavior does not depend on external knowledge) while *lazy binding* supports evolving, open worlds (where the component behavior can change when specialized by new units).

While these mechanisms have already been proved to be powerful and general enough to provide a unifying support to the notions of object, inheritance, viewpoints, etc. (see [7] or [1]), how to give them an effective implementation is still an open problem. In order to give contextual logic programming the power of a fully developed programming environment (including an optimizing compiler, an efficient run-time support and all the classic programming tools like debuggers, graphical interface, etc.), the simplest strategy is to integrate contextual mechanisms into a widely-used and well-supported logic programming system. However, even in this case different choices are possible, some of which have been compared in [3]:

- contexts can be implemented through a meta-interpreter
- contextual programs can be translated into conventional Prolog programs
- contexts can be supported by extending the underlying (Warren) abstract machine

In this work we will discuss in detail another implementation approach, called *program-representation-based*, inspired by the SICStus Prolog programming environment [11], and also been advocated for SEPIA [9]. This version of contextual programming has been called CSM, i.e. Contexts like SICStus Modules, since the representation of a context is quite similar to that of a SICStus Prolog module. As a consequence, each context is a first-class object, identified by a unique name, which can work as a structured logic theory as well as a conventional object that can be part of different prototypes. A context database is maintained by CSM, and can be exploited by higher-level tools in order to assist the programmer in building and executing applications.

While other proposals (e.g. that reported in [6], [2]) aim at finding a good balance between expressive power and efficiency, CSM aims at fully supporting the highly dynamic nature of contextual mechanisms. The notion of theory is exploited both at abstract and implementation level to support conventional object-oriented concepts such as inheritance, classes, fixed or mutable instances with private or shared state. Besides, more advanced concepts like hypothetical reasoning and dynamic object (re)classification can be expressed.

The work is organized as follows. In section 2 we briefly recall the main concepts and mechanisms related to contextual programming and their impact on software design and development. Section 3 is devoted to the presentation of the main characteristics of the CSM implementation. Conclusions and further remarks are reported in section 4.

2 Contexts as Structured Software Components

In CSM, a context is a first-class structured object identified by an atomic name. A context `c` is recursively defined as a pair (`u`,`c1`), where `u` is the name of the *top unit* of `c`, and `c1` identifies the *first subcontext* of `c`. The atom `emptyCtx` is used to denote the empty context, given as a constant. Since a unit is a named collection of (Prolog) Horn clauses, a context can be represented by the ordered list of its composing unit names, e.g. $[u_n, \ldots, u_1]$, where u_n is the first (most recent) unit of the context. [1]

The constructor `createCtx(U,C1,C)` builds a new context by extending the context referred to by the logic variable `C1` with the unit referred to by `U`, and links the unbound variable `C` to the new context identifier. Two selectors `topUnit/2` and `sub1Ctx/2` are introduced so that the goal `createCtx(U,C1,C), topUnit(C,U),sub1Ctx(C,C1)` is true when `U` and `C1` are properly instantiated.

The built-in predicate `C<-G` , where `C` is a context and `G` a goal, performs a *switch* of the current computation (line of reasoning) to context `C`, solving goal `G` within the structured theory `C`. In an object-oriented perspective, the context-switch operator sends message `G` to object `C` whose methods are the clauses made visible by `C`.

Context extension operators `>:>/2` and `>>>/2` (for *cactus* and *linear* extension, respectively, as mentioned in [1]), allow users to implicitly create and use objects in a dynamic way.

CSM supports two kinds of non-local calls (as defined in [7]) and two corresponding binding policies: *eager* calls and *lazy* calls. Let us consider a predicate call `p/n` within a unit u_i in the *current context* $[u_n, \ldots, u_i, \ldots, u_1]$. If a predicate `p/n` is eager-called within u_i, a definition for `p/n` is sought through the subcontext $[u_i, \ldots, u_1]$, called the *current bindcontext*. On the other hand, when a predicate call `q/n` is lazy-called within a unit u_i, a definition for `q/n` is sought through the whole current context.

In order to support non-local binding policies, CSM keeps track of both the current context and the current bindcontext, and provides two primitive operators: `curCtx(C)`, which binds `C` to the current working context, and `curBindctx(C)`, which binds `C` to the current bindcontext.

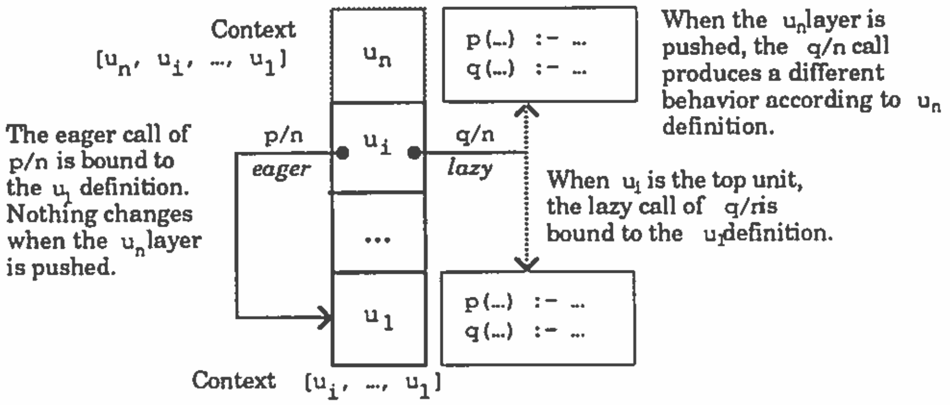
Context
$[u_n, u_i, \ldots, u_1]$
u_n
p(...) :- ...
q(...) :- ...
When the u_n layer is pushed, the q/n call produces a different behavior according to u_n definition.
The eager call of p/n is bound to the u_i definition. Nothing changes when the u_n layer is pushed.
p/n
eager
u_i
q/n
lazy
When u_i is the top unit, the lazy call of q/n is bound to the u_i definition.
...
u_1
p(...) :- ...
q(...) :- ...
Context $[u_i, \ldots, u_1]$

Figure 2.1. Eager and lazy binding policies.

CSM binding policies allow users to structure components in different ways. In particular, eager binding leads to conservative, closed architectures, while lazy binding supports evolving, open architectures. As shown in figure 2.1, the definition of `p/n` in u_i depends only on the

1 For instance, using list notation, context `[u2,u1]` is defined as the pair `(u2,[u1])`, where `u2` is its top unit and `[u1]` its first subcontext. Since the empty list `[]` is used for the empty context, `(u1,[])` denotes context `[u1]`.

previous layers $u_{i-1},…,u_1$. If every other call in u_i is local or eager, the behavior of the component represented by the context $[u_{i-1},…,u_1]$ can be completely defined when the context is created. Even if a more specialized component is built, by pushing new layers upon u_i, the behavior of the sub-component $[u_{i-1},…,u_1]$ remains unchanged. Thus, eager binding cannot support the incremental programming methodology of the OO paradigm, where the behavior of class methods depends on specific instance knowledge. On the other hand, lazy binding allows a unit to work as an open, evolving component, with no need to rewrite it in order to enhance or extend its features. In fact, lazy binding is the mechanism required to support the notion of *self* in OOP. In figure 2.1, since `q/n` is lazy-called, the behavior of the sub-component $[u_i,…,u_1]$ changes when the upper layer u_n is added, since a new definition for `q/n` is found there.

To express local, eager and lazy calls, CSM adopts the following syntactic conventions. A goal `g` called in a unit `u` is treated as a *local* call if there is a clause for it in `u` (i.e., when $\texttt{functor(g)} \in \upsilon_d$, where u_d is the set of *local* predicates explicitly defined within unit `u`). Otherwise, (i.e. when $\texttt{functor(g)} \notin \upsilon_d$) the call is considered to be an implicit request for an *eager* call, and eager binding policy is adopted. *Lazy* calls have to be explicitly expressed by prefixing the `#` (or the equivalent `self`) operator to a goal call. Moreover, an eager call can be explicitly forced through the `super` prefix operator. Let us consider the following example.

Example 2.1.

```
:- unit(carOwner).                :- unit(pcOwner).
owns(antonio,alfa33).             owns(antonio,lemon).
owns(evelina,volvoPolar).         owns(enrico,ibicomp).
owns(paola,fiat500).              owns(paola,macFX).
hasAcar(X)  :- owns(X,_).         hasApc(X)  :- owns(X,_).

:- unit(hasA).                    :- unit(hasMore).
hasA(X)  :- #hasAcar(X).          hasBoth(X)  :- hasAcar(X),
hasA(X)  :- #hasApc(X).                          hasApc(X).
```

A possible world for car owners is the context `[carOwner,hasA]`, which is built as a consequence of the following query:

```
?- createCtx(hasA,emptyCtx,HasCtx),
   createCtx(carOwner,HasCtx,CarOwnerCtx),
   CarOwnerCtx <- hasA(X).
```

or of the simpler and equivalent query

```
?- [carOwner,hasA] <- hasA(X).
```

The above queries would return:

```
   X = antonio; X = evelina; X = paola;
   no
```

In fact, a binding for `hasA/1` in `[carOwner,hasA]` is found in unit `hasA`, where only the lazy call of `hasAcar/1` succeeds in finding the `carOwner` clause; then, the `owns/2` call is treated as a local call and returns the above computed answers. While unit `carOwner` works as a closed theory, unit `hasA` is an open theory whose behavior is determined by the current context: in fact, all its calls are non-local. Furthermore, as they are lazy-calls, evolutive computations may occur. For instance, if a goal

```
pcOwner >>> hasA(X)
```

is called when `[carOwner,hasA]` is the current context, and `X` is unbound, we get:

```
X = antonio; X = evelina; X = paola;
X = antonio; X = enrico; X = paola;
no
```

Due to its "evolutive" implementation, the same goal `hasA(X)` now gives different answers, since it is now solved in the extended context `[pcOwner,carOwner,hasA]`.

On the contrary, unit `hasMore` is still an open theory (since it uses two undefined predicates, therefore interpreted as two eager calls), but is conservative, since its behavior is completely determined by the previous layers (subcontexts) in the context. For instance, if we try to solve the following goal in the context `[pcOwner,carOwner,hasA]`, with `X` unbound:

```
hasMore >>> hasBoth(X)
```

we should obtain:

```
X = antonio; X = paola;
no
```

Now, any other try to further extend the obtained context `[hasMore,pcOwner,carOwner,hasA]` with any other unit will not modify the situation: `hasBoth/1` in `hasMore` will always return the same answers as long as the sublayer (its subcontexts) remains unchanged.

❑

Expressive power of contexts

Contextual mechanisms provide a high modularity degree for knowledge representation and control, together with the capability of dynamically altering software system behavior and architecture with no permanent side-effects on already working components. Modularity is directly captured by the concept of unit. Units composing a context can be interpreted as an explicit representation of an inheritance tree branch: thus, contexts can be interpreted as object hierarchies, where both static (the concept of *super*) and dynamic (the concept of *self*) inheritance can be exploited. As a result, contexts provide a modeling power similar to that of object-oriented languages. Besides integrating both static and dynamic configuration of structured objects in a unique framework, they promote a declarative style of programming, with uniformity of properties (no distinction between methods and state) and referential transparency in method calling. Dynamic context extension can be used to perform run-time specialization of statically-built software components.

The following example aims at illustrating the above considerations.

Example 2.2.

```
:- unit(animal).                    :- unit(elephant).
move(walk)    :- #legs(_).          legs(4).
move(run)     :- #legs(2).          ears(2).
move(gallop):- #legs(4).            trunk(1).
move(fly)     :- #wings(2).

:- unit(fred).                      :- unit(bird).
legs(3).                            legs(2).
                                    wings(2).
```

An elephant object can be represented by context `[elephant, animal]`, which expresses the fact that an elephant can be seen as a specialization of the concept of animal.[2] A goal like

```
?- [elephant,animal] <- move(X)
```

would produce the following computed answers:

```
    X = walk; X = gallop;
    no
```

We could say that the `elephant` class statically inherits `move/1` methods from its parent class `animal` which, for its part, dynamically inherits the number of `legs` from its subclass.

If `fred` is a three-legged elephant, the goal

```
?- [fred,elephant,animal] <- move(X)
```

yields, as expected, the answer

```
    X = walk;
    no
```

since the "self"-binding of `animal:legs/1` now involves unit `fred`.

The same result can be achieved by performing a linear context extension operation:

```
    fred >>> move(X)
```

while working in the "elephant context" `[elephant, animal]`. Such an operation may be seen as a dynamic specialization of the elephant class, in the sense of `fred`'s three-legged being. ❑

CSM mechanisms allow users to extend the scope to the field of hypothetical reasoning and dynamic object classification. For instance, if we do not know whether `fred` is an elephant lacking a leg or a bird with an extra leg, we can try to extend both classes (`[elephant, animal]` and `[bird, animal]`) with `fred`, and then reason on the consequences of the two cases. .

2 Proper CSM declarations, such as `unit elephant inherits animal`, also allow static taxonomies to be created.

However, while contexts can suitably model the notion of class and instance, they lack the ability to capture the concept of a uniquely identifiable object with an internal modifiable state. In fact, in the form discussed here, contexts can support functional objects only. An extension of CSM to express objects with evolving state is presented in [10].

3 The Implementation

To be effective, contextual mechanisms have to be implemented efficiently. But efficiency is not enough. In order to put contexts in an effective software engineering perspective, contextual mechanisms should be integrated in an efficient, widely-used and well-supported system. To achieve this goal, we selected the SICStus Prolog programming system. Moreover, we decided that the following requirements had to be met:

- the implementation of contexts must introduce neither semantics distortion nor computational overhead with respect to standard (SICStus) Prolog programs;
- contexts should be first-class objects which can be referred to by logic variables;
- contexts should be memo-ized in an internal system knowledge base and reused when needed.

Given these requirements, the challenging issue was to extend an already existing system so as to obtain a good degree of efficiency with minimal intrusion. In order to achieve this we explored the space of the implementation techniques depicted in the tree of Figure 3.1:

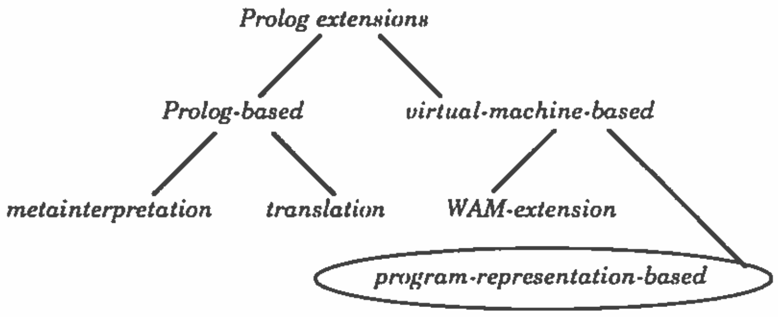

Figure 3.1. A taxonomy of Prolog extension techniques.

Translation and metainterpretation are absolutely non-intrusive techniques, since any *Prolog-based* approach adds a Prolog layer between the user and the underlying Prolog machine, leaving the machine itself untouched. In particular, *translation* transforms a contextual program P into a pure Prolog program P', semantically equivalent to P. With *metainterpretation* the Prolog interpretation cycle, extended with the new linguistic features, is reproduced by a metainterpreter, handling contextual programs as its data.

The main drawback of any Prolog-based approach is that it is affected by quite a low degree of efficiency. As illustrated in [3] (where some comparison figures are reported), a translator always works better than a metainterpreter, but it is affected by an undesirable performance decrease when program complexity grows.

The *virtual-machine-based approach* classically addresses the efficiency issue by modifying the language run-time support. In particular, the *WAM-extension* technique consists of enhancing the abstract Prolog machine in order to directly support new language features by introducing a set of new instructions and structures (a possible solution for contexts is reported in [6]). This technique has two main drawbacks (for a more complete discussion see [9]). The first is that the WAM emulator code is highly optimized, i.e. highly intricate and tricky. The second is that the WAM extension suffers from the overhead it imposes on all computations, including standard Prolog ones.

However, the particular structure of the SICStus machine emulator suggests a specialized version of the virtual-machine-based approach, directly based on the representation of Prolog programs. In figure 3.1 such an implementation approach has been called *program-representation-based.* In fact, the SICS-WAM relies on a program representation structured as a tree, rather than on a plain sequence of WAM instructions. In Figure 3.2 we depict the tree-based program representation as a separate block handled by a specific subsystem (the program representation manager).

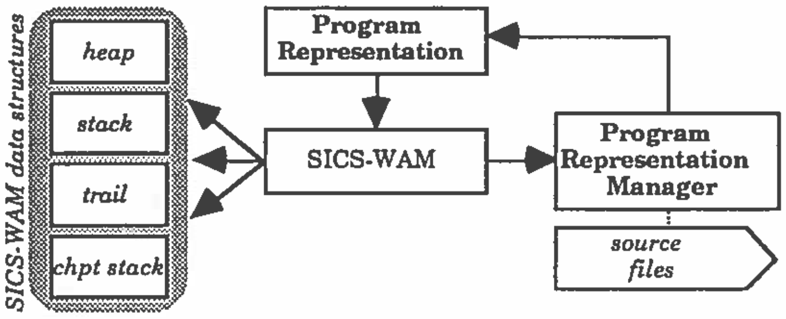

Figure 3.2. A SICStus Prolog architecture view

The root of the program tree is a hash table which leads to module descriptors. Each module has a name and a separate predicate space, represented again by a hash table pointing to predicate descriptors. These are the tree leaves, each one pointing to its code; for its part, the predicate code can be structured into several interconnected chains of clause codes (as in the case of compiled code). This structure is motivated essentially by the need to support some non-standard features, such as:

- a flat module system with no change to the basic WAM structure;
- a more efficient clause indexing scheme (avoiding `try`, `trust` and other indexing WAM instructions);
- incremental loading of clauses and dynamic predicates (for clause assertion and retraction);
- coexistence of compiled, interpreted and C-written Prolog predicates.

CSM architecture

As the SICS-WAM can access clause codes at several different levels of the tree, an *indirect addressing mechanism* is provided, even if compiler smartness forces almost-direct access in most frequent (and useful) cases. This configuration allowed us to meet two fundamental requirements for context implementation:

1) a *global naming* mechanism, associated to a partition of the clause space in order to implement the unit abstraction;
2) an *indirect addressing* mechanism, since contexts need dynamic binding.

The first requirement was satisfied by simply mapping CSM units onto SICStus modules, so as to reuse SICStus module names and separate predicate spaces. In particular, binding of local calls may consequently be delegated to SICStus mechanisms so that CSM intra unit computations are handled by SICStus as normal Prolog computations.

As far as the second point is concerned, our first choice was also to represent *C*ontexts as *S*ICStus *M*odules (whence CSM) so as to use the SICStus indirect addressing scheme for context binding, and to have contexts (for free) represented as physical objects in the system memory. On the other hand, this seemed to contrast with our "minimal intrusion" requirement: in fact, a frequent operation like context creation easily led to an undesirable crowding of SICStus module space; moreover, the need to record and handle specific CSM information forced us to extend SICStus native structures. Then, we finally switched to a configuration of the *CSM Knowledge Base* which uses SICStus unit representation in terms of modules as its underlying layer, and superimposes all CSM-specific information.

A context is then implemented like a SICStus module in the CSM KB space. That is, similarly to SICStus modules, a context is represented by a *context descriptor*, with its own identifier and predicate space. The descriptor of a context c constitutes the concrete representation of an abstract binding set c_d associated with c; c_d can be formally defined as follows:

$$\texttt{emptyCtx}_d ::= \emptyset \qquad \textit{(apart from Prolog built-in predicates)}$$

$$c_d ::= \{(u,p/n) \mid p/n \in u_v\} \cup \{(m,p/n) \mid (m,p/n) \in c1_d, p/n \notin u_v\}$$
$$\textit{(if } \texttt{topUnit(c,u), sub1Ctx(c,c1)} \textit{ holds)}$$

where u_v is the set of predicates (implicitly or explicitly) declared as *visible* in u. In other words, c_d consists of those predicates declared as *visible* in the top unit u of c, and of those predicates made visible by the first subcontext of c, unless redefined by u.

The context predicate set is represented via a hash table, working as a binding environment descriptor. Thus, context creation (which constitutes the CSM main computational cost when performed at run time[3]) consists essentially of a suitable (hash) table configuration. First, a new copy of the first-subcontext table is allocated, then the top-unit visible predicates are introduced. CSM default predicate-overriding semantics is achieved by overwriting the first-subcontext predicate entries with the top unit ones when collisions occur.

Contexts are permanently recorded in the CSM KB; they are deallocated only when a component unit is redefined. Thus, context creation needs to be performed only once: when a context has to be used again during a computation (for instance, after backtracking), it is simply restored from the CSM KB through the *context table*.

Context predicate tables are used to perform non-local binding efficiently. When a non-local call (eager or lazy) takes place in a context, a hash access to the context predicate table is performed in order to find (if a suitable predicate exists) the proper predicate information, similar to the SICStus module call mechanism. All information needed (the owner unit and the corresponding bindcontext, the code address, etc.) can then be accessed at once, so that the code can be immediately executed after the current bindcontext has been properly set.

Since contexts are completely configured at creation time, only one context table access is needed at run time for both eager and lazy binding. Thus, also evolutive software components

[3] Actually, CSM also allows static is-a hierarchies to be declared. Thus, when a unit hierarchy is defined statically, the corresponding context is built at compile time, so that the only run time cost is related to context retrieving from CSM KB.

can be implemented as efficiently as conservative ones (see [5] for a comparison). This overcomes one of the main problems of previous contextual programming implementations, without excluding further optimizations for eager calls, as discussed in [6]. On the other hand, this technique is flexible enough to be uniformly applied to both static and dynamic unit composition. [2] describes a highly optimized approach to inheritance between logic theories, tailored to the static case: however, it implies a rather strong modification of the host system.

On the contrary, with respect to the "minimal-intrusion" implementation requirement, CSM computation evolves as a normal Prolog computation until a particular CSM operator or an *undefined predicate* exception is caught. In this case, a non-local call is performed using the current (bind)context as binding environment; then, when the call has been bound and the correspondingcontext switch been done, computation proceeds as a normal SICStus Prolog one, till the next CSM event.

Thus, CSM "overrides" SICStus Prolog only in the case of non-local calls, and no overhead is introduced by local CSM computations with respect to SICStus native mechanisms. Although a reasonable degree of efficiency is achieved, the cost of a non-local call is not negligible. Global efficiency in the execution of a CSM program can be appreciably affected by the number of non-local calls performed by the program. At any rate, since this number is strongly related to program design and architectural issues (such as granularity of unit definitions), giving quantitative figures is not straightforward. Moreover, the extra-cost of non-local binding seems unavoidable, at least when dynamic inheritance is concerned.

4 Conclusions

Thanks to efficient, reliable, compiled implementations of Prolog, logic languages can be effectively exploited to develop applications according to a declarative style, so as to reduce the gap between software specification and implementation. CSM contextual extensions aim at enhancing Prolog with concepts and mechanisms for programming-in-the-large, giving effective support to evolutive criteria of software development. In fact, CSM contexts can work as structured objects, to be first used in a prototyping phase, and later refined, in order to match requirement adjustment during application development. Software evolution can then take place incrementally with no gap (and no need for extra-language tools) between the definition of prototypes and final implementation.

As far as CSM implementation is concerned, traditional techniques for extending Prolog turn out to be either quite inefficient or intrusive. Moreover, requiring a full integration with the host system of choice involves proper reusing and reengineering of existing components. Our choice to explore the so-called "program-representation based" implementation approach within the SICStus Prolog system, allowed us to meet important requirements, such as efficiency, non-intrusiveness of the extension, and full integration with SICStus programming tools. The resulting CSM run-time support delays, until context creation time, the allocation and configuration of those descriptors usually built up at compile-time by more (class-based) static languages. In this way most of the object-oriented notions (such as object creation, message passing, inheritance) have a close counterpart in the CSM contextual model, which extends its scope to advanced features such as hypothetical reasoning and dynamic (re)classification.
In the version discussed in this paper, CSM does not touch on the problem of objects with mutable state, which is the main issue of our current research. First results on this topic have been illustrated in [10].

Acknowledgements

This work has been partially supported by the "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of Italian National Research Council (CNR) under grant nº 890004269. We would like to thank Evelina Lamma, Paola Mello and Michele Bugliesi for many useful discussions we had with them, and DS Logics who sponsored part of the CSM development.

References

[1] A. Brogi, E. Lamma, P. Mello. *A General Framework for Structuring Logic Programs.* C.N.R. Technical Report "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo", No 4/1, May 1990.

[2] M. Bugliesi, G. Nardiello. *SelfLog: Language and Implementation.* In these proceedings.

[3] E. Denti, E. Lamma, P. Mello, A. Natali, A. Omicini. *Techniques for Implementing Contexts in Logic Programming.* In E. Lamma and P. Mello, editors, Proceedings of the Third Workshop on Extensions of Logic Programming WELP92, Bologna, Italy, February 1992. LNAI, volume 660, pages 339-359 , Springer-Verlag.

[4] *Rapid Prototyping.* Computer, volume 22, No. 5, IEEE Computer Society, May 1989.

[5] B. Jayaraman, G. Nadathur. *Implementation Techniques for Scoping Constructs in Logic Programming.* In K. Furukawa, editor, Proceedings of the Eighth International Conference on Logic Programming, Paris (F), The MIT Press, Cambridge, Massachusetts, 1991.

[6] E. Lamma, P. Mello A. Natali. *An Extended Warren Abstract Machine for the Execution of Structured Logic Programs.* Journal of Logic Programming 1992, No. 14, pages 187-222, Elsevier, New York.

[7] P. Mello, A. Natali, C. Ruggieri. *Logic Programming in a Software Engineering Perspective.* In E.L. Lusk and R.A. Overbeek, editors, Proceedings of the North American Conference on Logic Programming NACLP89, The MIT Press, Cambridge (USA), 1989.

[8] L. Monteiro, A. Porto. *Contextual Logic Programming.* In G. Levi and M. Martelli, editors, Proceedings of the Sixth International Conference on Logic Programming, The MIT Press, Cambridge (USA), 1989.

[9] M. Meier, J. Schimpf. *An architecture for Prolog Extensions.* In E. Lamma and P. Mello, editors, Proceedings of the Third Workshop on Extensions of Logic Programming WELP92, Bologna, Italy, February 1992. LNAI, volume 660, pages 319-338, Springer-Verlag.

[10] A. Natali, A. Omicini. *Objects with State in Contextual Logic Programming.* Proceedings of the Fifth International Symposium on Programming Language Implementation, Tallin, Estonia, August 1993.

[11] Swedish Institute of Computer Science. *SICStus Prolog User's Manual.* Kista, Sweden, January 1993.

SelfLog: Language and Implementation

M. Bugliesi and G. Nardiello
Dip. di Matematica Pura ed Applicata
Università di Padova
Via Belzoni 7, I-35131 Padova, Italy
phone: +39-49-83.19.82
e-mail: {michele@goedel,giuseppe@hilbert}.unipd.it

Abstract

We discuss a modular extension to logic programming with built-in mechanisms for inheritance and message passing. The notions of inheritance we consider encompasses the two notions known in the literature as static and dynamic inheritance. The significance of this extension has been addressed in the recent literature. In this paper the emphasis is primarily on the implementation. We describe an extension of the WAM which supports these new mechanisms, and we outline the current implementation together with some optimizations. We finally discuss the advantages of our approach with respect to other implementation techniques for similar extensions of logic programming found in the literature.

1 Introduction

The need for a modular extension to logic programming has been advocated by a number of researchers during the last decade. Their work has been motivated by the assessed agreement that, in spite of its declarativity, logic programming does not scale very well when it comes to designing practical applications. Its unit of abstraction – relations – appears in fact to be too fine-grained to support the design and the development of large programs.

Originated by Miller's seminal paper on the subject [10], a number of different modular extensions of logic programming have been proposed in the literature.

One of the directions this research has evolved along has been inspired by the idea of extending logic programming with some of the distinguishing features of the Object-Oriented programming paradigm. The declarative nature of logic programming, coupled with the knowledge-structuring power inherent in the O-O notions of inheritance and message-passing seemed in fact to provide an adequate basis for a promising solution. From a logical point of view, an object – the O-O unit of abstraction – has a natural interpretation as a logic theory ([9]): an object is simply a collection of axioms which describe what is true about the object itself. Although somewhat reductive, since it deliberately excludes any notion of state associated with objects, this interpretation has nevertheless the significant advantage of leaving ground for elegant semantic characterizations of this approach to integration ([2]).

Inheritance systems or *method systems* are the terms sometimes used to qualify the class of languages which encompass only the aspects of Object-Orientation directly related to inheritance. Most of the O-O extensions of logic programming found in the current literature fall in this class and differ only in the definition of inheritance they consider (cf. [3], [4], [13], [12] for examples).

The approach we study in this paper conforms with this view. The language we consider, SelfLog, exhibits an embedded form of modularity whereby sets of clauses can be collected into named theories called *units*, and units can be composed to form larger programs. Unit composition occurs at two different levels: units can be linked into static *isa*-hierarchies and can communicate by requesting one another the evaluation of a goal. A program in the language is thus viewed as a set of unit-hierarchies where the computation proceeds by exchanging the control between hierarchies with the evaluation of message-goals. The semantics of the *isa* composition encompasses the notion of overriding between inherited definitions and the two mechanisms known in the literature as *early* (or *static*) and *late* binding. The significance of this extension of logic programming has been already addressed in the literature (see [2] for a fuller discussion about these issues). In this paper the emphasis is on the implementation aspects of the extension.

1.1 Goals and Outline

The architecture we propose is based on the WAM, and it provides a highly efficient support for the new mechanisms of inheritance (early and late binding) and message-passing. The implementation of late-binding is realized by adapting the indirect-call mechanism used in modern Object-Oriented languages to conform with the non-deterministic type of computation peculiar of the WAM. Two additional registers are used to integrate in the WAM the workings of late binding and message-passing. The resulting architecture gives evidence of how the new mechanisms find a natural and effective integration with the original design of the WAM. Most of the implementation burden is in fact left to the compiler while the extensions have a minimal impact on the run-time support. The significant consequence is that the new architecture leaves room for all the optimizations – *tail recursion elimination*, *environment trimming* and *indexing* – which are crucial to the efficiency of the WAM. This is indeed one of the distinguishing features of our approach with respect to similar modular extensions of logic programming proposed in the literature. The highly dynamic nature underlying other composition mechanisms, such as those described in [8] and [7], leads in fact to the definition of a run-time support with a much richer dynamic component for which these optimizations are no longer applicable (see section 5 for a detailed discussion on this and related issues). From this point of view, the approach we propose here seems to hit a good balance between gained expressive power and retained efficiency of the implementation.

Plan of the paper. In the next section we give a brief overview of the language we employ throughout. In section 3 we describe the extensions needed to support the new mechanisms. In section 4 we describe the current implementation together with some optimizations. We also address an independent extension of the language to allow different modes (*extension* and *overriding*) for inheritance to coexist, and we analyze the impact of this extension on the current architecture. In section 5 we compare our approach with some related proposals. We conclude in section 6 by summarizing the key points of the paper.

2 The Language

SelfLog is defined along the guidelines of [4] and [13]. Programs in this language are defined as sets of units where each unit is an *ordered sequence* of definite clauses of the form *Head :- Body*. The head of a clause is a positive atomic formula and the body is a conjunction $G_1, \ldots, G_n$. The G_is are either positive atoms (goals) or message-goals of the form $u : G$, where G is again assumed positive and atomic. The units of a program can be structured hierarchically. A unit declaration has the general form u *isa* su :: [⟨*clause list*⟩] and has two effects: it associates ⟨*clause list*⟩ with unit u and it identifies su as the immediate ancestor of u. For any unit u, $\mathcal{H}(u)$ denotes the *hierarchy associated with* u, that is the ordered sequence of units obtained by taking the *isa*-transitive closure starting from u. When $\mathcal{H}(u) = u\ isa\ u_1\ isa \cdots isa\ u_n$, we say that u and u_n are respectively the *top* and *bottom* units of $\mathcal{H}(u)$ and that u_n is a *base* unit (with no ancestors). The *isa*-composition provides a support for single inheritance only: the *isa*-hierarchies contained in a program form a tree-like structure where each unit has at most one immediate ancestor.

This composition of units encompasses a combination of the forms of inheritance addressed earlier in the introduction. Both defined with an overriding semantics, they embed the two mechanisms known as static inheritance – a là Simula67 – and dynamic inheritance – a là Smalltalk. This is done in the same spirit as in C++ where virtual and *standard* functions can be freely intermixed. In contrast to that case, here the combination occurs at the call level rather than at the definition level. Hence, the same predicate name may be evaluated according to either of the two mechanisms depending on the call. We assume the dynamic mechanism as the default evaluation rule and use explicit markers to enforce the evaluation of a goal according to the static mechanism. A pound sign (#) used as the prefix of a goal will serve this purpose. Marked calls will be henceforth referred to as *early calls* as opposed to the qualification *late* used to refer to unmarked calls.

Operational Semantics. The operational semantics of SelfLog can be formally characterized by describing how a non-deterministic interpreter should function. The behaviour of the interpreter is defined in terms of computation states of the form $\langle u, \mathcal{H}, \theta, G\rangle$ $\mathcal{H}$ is the current *isa*-hierarchy, θ is the current substitution, where $G\theta$ is the current resolvent and u is the unit of $\mathcal{H}$ where $G\theta$ is being evaluated. The course of action is driven by the reduction-rules defined below. In the following, $\mapsto$ denotes the reduction relation over states, $\mapsto^*$ stands for the reflexive and transitive closure of $\mapsto$, and, for any atomic goal A, $\widehat{A}$ denotes the predicate symbol of A.

(MSG) $\langle u, \mathcal{H}, \theta, \overline{u} : G\rangle \ \mapsto\ \langle \overline{u}, \mathcal{H}(\overline{u}), \theta, G\rangle$;

(EARLY) $\langle u, \mathcal{H}, \gamma, \#A\rangle \ \mapsto\ \langle u_k, \mathcal{H}, \sigma, \bar{B}\rangle$ where *(i)* u_k is the first unit containing a definition for $\widehat{A}$ found in a top-to-bottom scan of $\mathcal{H}(u)$ (the sub-hierarchy of $\mathcal{H}$ associated with u), *(ii)* $A' \text{:-} \tilde{B}$ is a clause of u_k, *(iii)* $\theta = mgu(A\gamma, A')$ and $\sigma = \gamma\theta$,

(LATE) $\langle u, \mathcal{H}, \gamma, A\rangle \ \mapsto\ \langle u_k, \mathcal{H}, \sigma, \bar{B}\rangle$ where *(i)* u_k is the first unit defining $\widehat{A}$ found in a top-to-bottom scan of $\mathcal{H}$ (*independently of* u), and as before, *(ii)* $A' \text{:-} \tilde{B}$ is a clause of u_k, *(iii)* $\theta = mgu(A\gamma, A')$ and $\sigma = \gamma\theta$,

$$\text{(AND)} \qquad \langle u, \mathcal{H}, \theta, (G_1, G_2)\rangle \;\longmapsto\; \langle u, \mathcal{H}, \sigma, G_2\rangle \;\text{ if }\; \langle u, \mathcal{H}, \theta, G_1\rangle \;\longmapsto^*\; \langle u', \mathcal{H}', \sigma, \Box\rangle$$

No reduction occurs over states of the form $\langle u, \mathcal{H}, \theta, \Box\rangle$ whose associated resolvent is empty. If $\langle u, \mathcal{H}, \epsilon, G\rangle \longmapsto^* \langle u', \mathcal{H}', \theta, \Box\rangle$ then $\theta_{|var(G)}$ is an answer substitution for G.

It follows from the reduction rule (MSG) that in any computation state $\langle u, \mathcal{H}, \theta, G\rangle$, $\mathcal{H}$ is the hierarchy associated with the unit which is the receiver of the last message-goal. In fact, the effect of a message-goal $\overline{u} : G$ is to enforce the evaluation of the goal G in the hierarchy of *isa*-ancestors associated with unit $\overline{u}$. As for late and early calls, the different behaviour is reflected in the different hierarchies where a matching definition for the current resolvent is searched: the (whole) current hierarchy, $\mathcal{H}$, for late calls, the sub-hierarchy $\mathcal{H}(u)$ whose top unit is the current unit for early calls. In both cases the overriding semantics is captured by the fact that only the *first* matching definition is considered for reducing the resolvent of the current state.

An example will help clarify the use and the semantics of the various features of SelfLog.

Example 2.1 The following units describe a naive graphical environment for displaying geometric figures. We assume that figures can be displayed only if framed into enclosing windows. Unit *picture* below defines the method *display/1* which takes care of creating the frame before displaying the requested figure. Unit *square* inherits the definition of *display/1* from *picture* and overrides *picture*'s definition of *new/2.*

$$square\ isa\ picture :: [new(Size, Obj) \text{ :- } create\text{-}sq(Size, Obj).]$$

$$picture :: \left[\begin{array}{lll} display(Size) & \text{:-} & \#new(Size, Frame), \\ & & frames : popup(Frame). \\ new(Size, Frame) & \text{:-} & create\text{-}frame(Size, Frame), \\ & & new(Size, Obj), \\ & & put(Obj, Frame). \end{array}\right]$$

The definitions of *put/2* and *create-frame/2* in unit *picture* and the definitions in unit *frames* have been omitted being irrelevant to the purpose of the example. Consider now the message-goal *square:display(4)* invoked from a third arbitrary unit and requesting that a square of side 4 be displayed. Evaluating *square:display(4)* amounts to evaluating *display(4)* in the hierarchy *square isa picture*. The sequence of steps is outlined below. *display/1* is not defined by *square* and thus its definition is inherited from *picture*. Then, the definition for the early call *#new(Size,Frame)* is selected from the hierarchy $\mathcal{H}$*(picture) = picture* of the unit where the call occurs. Thus, the evaluation proceeds by first creating the frame and then evaluating the goal *new(Size, Obj).* Now the definition for the late call *new(Size,Obj)* is selected in the hierarchy $\mathcal{H}(square)$ associated with the unit which was the receiver of the last message-goal. The overriding semantics of the *isa*-composition guarantees that *square*'s definition of *new/2* is the only available definition. Then the computation comes to an end by first creating the square, putting it into the frame and finally popping-up (and displaying) the composite object.

Notice that the above use of early and late calls is crucial to achieve the correct behaviour. A late call for *new/2* in the definition of *display/1* would fail to create the frame needed to display the square. An early call for *new/2* in *picture*'s definition of *new/2* would result into an infinite loop. $\Box$

3 Enhancing the WAM

The model for implementing SelfLog is designed as an enhanced WAM equipped with new instructions and data structures needed to support the new mechanisms. As already anticipated, the impact of the extension on the original structure of the WAM is minimal. In the following, we assume that the reader is familiar with the WAM design and terminology (cf. [1] for a comprehensive description).

The new architecture extends the WAM with a new static area and two additional machine registers. Associated with each unit, the static area holds an address table, the *unit's p-table*, which records the addresses (in the code area) of the predicates which are defined or called by the unit's clauses. The p-tables provide the run-time support for realizing the indirect-call mechanism used to evaluate a late call. The idea is to set up the p-tables so as to enable the late-call dispatcher to retrieve the address of the callee by a direct access (by offset) to the p-table of the top unit of the current hierarchy (which is the receiver of the last message goal). The reference to this p-table is held, at any instant of the computation, in a new register named SELF. The contents of SELF will obviously be overwritten by the invocation of a message-goal and consequently, since a message-goal $\overline{u} : G$ may in general occur in a conjunctive resolvent, a corresponding register, C_SELF, will be used to restore the correct value of SELF after returning from the call $\overline{u} : G$.

3.1 Initialization of the p-tables

The entries in the units' p-tables are filled in during compilation. Each hierarchy in the program can be compiled independently of the others. In contrast, units belonging to the same hierarchy are required to be compiled in an ordered sequence starting with the base of the hierarchy. Therefore, when compiling a unit we may assume that the p-table of the unit's immediate ancestor (if any) is available.

Let $u \rightarrow$*p_table* denote the p-table associated with unit u. We assume that the compiler keeps a symbol table associating with each predicate name p/n (symbol and arity) the offset at which the corresponding address, $@p/n$, occurs in the unit's p-table. The notation $u \rightarrow$*p_table(off)* will be used to refer to the address occurring at offset *off* in $u \rightarrow$*p_table*.

Consider now the compilation of a unit u whose immediate ancestor is unit su. The trick is to set up the p-tables so as to ensure the alignment of the p-table entries for the same predicate names in all the units belonging to the same *isa*-hierarchy. To this purpose, the p-table for u is initialized to the p-table of su. New entries will then be created, or existing entries overwritten, as the compilation of the unit's clauses proceeds. The source code is compiled according to the standard scheme of the WAM. There are, however, some relevant modifications.

A clause is compiled by first considering the predicate name p/n of its head. If $@p/n$ occurs at offset *off* in $u \rightarrow$*p_table*, then the address of the definition which is being compiled overwrites the address found in $u \rightarrow$*p_table(off)*. Otherwise, if $@p/n$ does not occur in $u \rightarrow$*p_table*, a new entry is created and initialized with the new address. Then, for each atom G in the body of the clause, we distinguish two cases:

(a) G is atomic and corresponds to a (late/early) call for predicate q/m. If $@q/m$ occurs in

u→*p_table*, then proceed without modifying the p-table for u. Otherwise, create a new entry and initialize the corresponding address to the address of the failing routine.

(b) G is a message-goal of the form $\overline{u} : q(\cdots)$ (equivalently $\overline{u} : \#q(\cdots)$). If the p-table for $\overline{u}$ already exists, then behave as in (a) using $\overline{u}$'s p-table. Otherwise proceed without any action on the p-tables.

Notice that in case (b) the compilation of u proceeds without interruption and the creation of the p-table for $\overline{u}$ is deferred until this unit gets compiled. This is needed to ensure the alignment of the p-table entries in the *isa*-hierarchy which $\overline{u}$ belongs to.

So far, we have identified only the actions to be taken for setting up the p-tables and we haven't yet specified the rules for generating the compiled code. Before doing so, however, it's worth trying to justify the use of the p-tables intuitively.

The idea is indeed rather simple. Let $\langle u, \mathcal{H}(\overline{u}), \theta, G\rangle$ be the current computation state and let G be an atomic goal (not a message-goal) whose predicate symbol is q/m. Then the address of the associated procedure – if any – will be held in the p-table of one of the units in the hierarchy $\mathcal{H}(\overline{u})$ associated with the receiver $\overline{u}$ of the last message-goal. We distinguish two cases:

1. If the call is an early call, then the address $@q/m$ is either already contained in u→*p_table* (since all of u's ancestors have already been compiled) or it will be after compiling the local definition. Hence, the call can be compiled (or backpatched) using the address found in u's p-table. Notice that, if neither u nor any of its ancestors contain a definition for q/m, then, in view of point (a) above, the address associated with the call causes the invocation of the failing procedure (which is just what we would expect).

2. Otherwise, if the call is a late call, then the address $@q/m$ will be held in $\overline{u}$→*p_table* no matter whether $\overline{u}$ defines or inherits the definition. Notice that $@q/m$ is also contained in u→*p_table*, just because the call to q/m occurs in u. Now, since the entries in the p-tables of all the units in $\mathcal{H}(\overline{u})$ are aligned, the offset *off* of the entry $@q/m$ is the same in $\overline{u}$→*p_table* and in u→*p_table*. Hence, the address of the callee can be simply retrieved from $\overline{u}$→*p_table(off)*. Furthermore, since *off* is known at the time u is compiled, the late call can be translated into an indirect-call to the procedure whose address is held in $\overline{u}$→*p_table(off)*.

   As already mentioned, this translation schema is made to work by having **SELF** point to the p-table of the last receiver of a message goal ($\overline{u}$ in this case).

With this picture in mind we can now look at the new steps needed for code generation.

3.2 Code Generation

The typical WAM instruction pattern for a clause is a sequence of interleaved *put* and *call* instructions for each of the goals of its body – the sequence being opened by the *allocate-get* and closed by the *deallocate-proceed* patterns. Environment allocation/deallocation is optimized by the *Last Call Optimization* (LCO): the last call in the body of a clause reduces to a simple jump to the address of the associated procedure after having restored the continuation for the clause by deallocating the associated environment.

Early calls. For early calls, the code-generation schema of the WAM is kept essentially untouched. The address of the callee is known at the completion of the compilation of each unit and the call can be compiled or backpatched as standard calls are compiled in the WAM (cf. [1]). There is only one additional step. In fact, since the evaluation of the call might involve a message-goal, the value of SELF needs saving in the continuation register C_SELF before making the standard call to the associated procedure.

Late calls. The evaluation of late calls relies on the indirect-call mechanism outlined above. Let $q(\cdots)$ be the late call and $\langle \mathit{off} \rangle$ be the offset of $@q/m$ in the p-table of the unit u where the call occurs. The effect of an indirect call is to first fetch the address at offset $\langle \mathit{off} \rangle$ in the p-table referenced by SELF and then to evaluate the call. We can thus introduce a new instruction for the extended architecture which realizes the effect of a late call. The definition of the new instruction is displayed below.

$$\begin{array}{lcl} \texttt{latecall}\ \langle \mathit{off} \rangle & ::= & \langle \mathit{addr} \rangle := \mathrm{SELF} \rightarrow \mathit{p\_table}(\langle \mathit{off} \rangle) \\ & & \mathrm{C\_SELF} := \mathrm{SELF} \\ & & \texttt{call}\ \langle \mathit{addr} \rangle \end{array}$$

As for early calls, the value of SELF is saved in C_SELF before evaluating the standard `call` instruction of the WAM.

Message-goals. The compilation of message-goals follows the same pattern as that of early calls. The only difference is that the p-table of the unit specified in the message-goal might not exist when the goal gets compiled. Hence, backpatching the call might entail waiting for the compilation of the unit specified in the message-goal. This is accomplished by a final *linking* phase when all the unresolved cross-references between the program's *isa*-hierarchies can be finally computed. Early and late calls can be treated uniformly within a message-goal because the unit specified in the message-goal is also the unit in which the goal gets evaluated.

At run time, the effect of a message-goal of the form $\overline{u} : q(\cdots)$ will be simply to first set SELF to point to the p-table of $\overline{u}$ (after having saved the value of SELF) and then to evaluate $q(\cdots)$. The corresponding instruction is displayed below.

$$\begin{array}{lcl} \texttt{msgcall}\ \overline{u}, @q/m & ::= & \mathrm{put}\ \langle \mathit{arguments\ of}\ q/m \rangle \\ & & \mathrm{C\_SELF}\ :=\ \mathrm{SELF} \\ & & \mathrm{SELF}\ :=\ \overline{u} \rightarrow \mathit{p\_table} \\ & & \texttt{call}\ @q/m \end{array}$$

where $@q/m$ is the address for q/m held in $\overline{u} \rightarrow \mathit{p\_table}$ and `call` is again the standard call instruction for the WAM.

3.3 Impact on the WAM instruction-set

A few additional extensions to the call-return mechanism of the WAM are needed to ensure a correct treatment of the two registers SELF and C_SELF. We have already said that the value of SELF must be saved in C_SELF before actually calling a procedure. A dual operation will be required upon returning from the call. Therefore proceed has to ensure that the value of SELF be restored by making the assignment SELF := C_SELF before proceeding to the continuation.

Environment allocation. allocate and deallocate will also be extended so as to ensure that the value of C_SELF be saved upon entering the clause and restored before exiting (along with the standard information contained in the environment frames). Notice that SELF needs *not* saving in the environment (the relation between SELF and C_SELF is essentially the same as that between the program counter and the continuation point).

Choice-point manipulation. Both SELF and C_SELF need instead saving in (and restoring from) the choice-point frames in order to ensure that all the alternative clauses start off with the same SELF and associated continuation. The WAM instructions for choice point manipulation (cf. [1]) will be therefore extended accordingly.

Last-call Optimization. An important property of the treatment of SELF and C_SELF is that it is fully compatible with the Last Call Optimization scheme of the WAM: assume we deallocate the environment of a clause before evaluating the last call. At this point there is no need to save the value of C_SELF before evaluating the last call, since the value of C_SELF for the last call and for the clause itself simply coincide. The execute instruction of the WAM is thus left untouched and the last call of a clause transfers the control to the callee without any further action. As an additional remark note that the assignment C_SELF := SELF can be safely dropped in several cases (other than LCO) upon early and late calls. The reason is easily justified by observing that the value of C_SELF is constant for all such calls in the body of the clause. Hence, its value can be set once and for all at the first call. This single assignment must be ensured: if we dropped it, upon returning from the first call, proceed would set SELF to the value of C_SELF for the parent goal (which is obviously incorrect). A refined solution could rely on the existence of two different instructions for returning respectively from message-goals and from ordinary calls. Under this assumption, the value of SELF is to be restored only upon returning from a message-goal whereas early and late calls need not – and *must not* – save the contents of SELF.

Indexing. The final point to notice is that the overriding semantics of the *isa*-composition allows us to leave the indexing schema of the WAM untouched. This simple solution is applicable since different definitions for the same predicate in different units are always mutually exclusive. This issue will be furtherly discussed in the next section where we study the extension of the language with a third form of inheritance with an *extension* (to be contrasted to *overriding*) semantics.

4 Current implementation

An experimental implementation of SelfLog has been realized by extending the WAM engine and the associated compiler provided by SICStus-Prolog. The module facility already supported by SICStus has greatly eased the implementation effort. Associated with every module, SICStus keeps a hash-table storing pairs $\langle\, name, addr \,\rangle$ where the first component denotes the name of the predicate and the second refers to the entry point of the corresponding procedure in the code area. The tables are used both at compile time for backpatching (SICStus is a one-pass compiler) and at run time, to allow calls for compiled and interpreted predicates to be freely intermixed. In the extended architecture they serve also a third purpose. The p-tables described in the previous section are in fact physically embedded in the hash-tables and these, in turn, are treated by the late-call dispatcher simply as address-tables and accessed by offset.

SICStus defines also a *system module* containing the definition of all the built-in and system predicated. In the enhanced architecture, this module serves as the root of all the *isa*-hierarchies of user-defined units. The compilation of the base-unit of any such hierarchy starts off by assuming the system module as the base's super-unit and then proceeds as described in section 3.

4.1 Optimizations

The architecture we have outlined so far leaves room for further optimizations. There are two sources of overhead with respect to the WAM we would like to remove. The first is determined by the increased number of memory accesses needed to save and restore the contents of the two registers SELF and C_SELF. The second comes from the increased size of the environment (one extra cell) and of the choice-point (two extra cells) associated with a call. The modified structure of these components raises a further problem: existing Prolog programs would need re-compiling to work correctly on the new architecture even though they make no use of the extended features of SelfLog.

There is, however, a relatively easy and elegant way to get around both these problems.

Separating two control components. The solution is similar to that outlined in [8] but more effective. The idea is to separate the two control components present in the the run-time support of the extended architecture, the one which is directly related to the modular extension, and the original control component of the WAM. In terms of the data structures we have considered, this amounts to splitting the contents of the choice point (and of the environment) for a call in two components, the first holding the contents of the registers SELF and C_SELF, the second saving the state according to the original workings of the WAM. In other words, we can devise a refined architecture which uses a new stack – the *unit-stack* – to hold the choice-point and environment information relative to SELF and C_SELF. The new stack grows and shrinks synchronously with the stack of activation frames and choice-points of the WAM (the *local stack*). To handle the new stack correctly, we also need two extra registers, corresponding to the B and E registers of the WAM, holding respectively the pointer to the last choice-point and to the last environment stacked on the unit-stack. The values of the two registers are saved in (and restored from) the unit-stack.

We can finally complete the design by accomodating two separate types of control instructions in the instruction set for the new architecture to manipulate respectively the local stack and the unit-stack.

Within this architecture, the execution of a Prolog program can safely be carried without any reference to the modular component of the run-time support (or portions thereof). The only assumption is that the compiler be instructed not to generate any instruction for manipulating the unit-stack when processing one such program. We will therefore allow two modes to guide the compilation of a program so as to ensure the consistency of the generated code with respect to the run-time data structures involved in the computation. A further pleasant consequence is portability: any existing piece of WAM code for Prolog programs, could be ported directly on the new abstract machine and would run as efficiently as on the WAM.

4.2 An extension mode for inheritance: indexing revisited

We now discuss an independent extension to the architecture we have described so far, to account for an *extension* mode for inheritance. The idea is to allow the definition of a predicate in a unit to extend (as opposed to override) the definition of the same predicate found in the unit's ancestors. Assuming that the ancestors have been compiled, we would like to generate the indexing code for the extended definition by "merging" the indexing trees of the new and the existing definitions.

The WAM indexing schema requires partitioning the sequence of clauses defining a predicate into a sequence of subsequences where each component subsequence is either a single clause (with a variable occurring as the first argument), or a maximal subsequence of contiguous clauses whose first arguments are not variables. The extension mode for inheritance can be accommodated in this schema by having the compiler link the last subsequence of a predicate's definition in a unit, with the first subsequence of the definition for the same predicate found in the unit's ancestor.

Let T be the last subsequence for a predicate p/n in unit u, su be the immediate ancestor of u and let $S_1, ..., S_m$ denote the sequence of subsequences defining the same predicate in su. We distinguish two cases depending on the fact that T and S_1 are, or are not, to be merged into single subsequence as the result of composing u with su.

The case when T and S_1 don't merge is actually the easier one. The expected alternative for the subsequence T is provided by S_1. Yet, jumping to the entry point of S_1, after failing on T, would cause the allocation of a further *useless* choice-point (S_1 being opened by a `try` instruction). Hence, after a failure on T, the control should skip the first and jump directly to next instruction of S_1. Of course, it should be also guaranteed that all the remaining alternatives $S_2, \cdots, S_n$ will be eventually taken into account. The desired flow of control is achieved by means of an additional instruction which provides the link between T and S_1. This instruction is appended at the end of the code for T and has the following format: `retry` $next(S_1)$ `else` S_2. Its effect is the same as that of `retry_me_else` S_2 except that the control is transferred conditionally to $next(S_1)$ and not to the instruction following `retry_me_else`. The translation schema is completed by replacing the trust instruction which would normally open the code for T by a `retry_me_else` instruction whose argument is the entry-point for the additional instruction described above.

The case when T and S_1 belong to the same *logical* subsequence is handled along the same pattern outlined above. The difference is that now the dispatching code for T (cf. [1]) must be instrumented to contain also the references the entry points corresponding to the clauses of S_1.

5 Related work

The implementation of inheritance systems in logic programming has already been considered in the literature. However, inheritance has often been seen as a special case of more general structuring mechanisms, and the implementation of inheritance systems simply as a byproduct of more general architectures.

The approach developed in [8] represents one such proposal. The language of *statically linked units* of [8] constitutes essentially the same inheritance system as that defined by SelfLog. As in our case, early calls are compiled away and evaluated by address. The implementation of late binding relies instead on the classical look-up semantics originally introduced for Smalltalk. As in our case, there is a p-table associated with each unit. The difference is that the p-tables in this case hold pairs of the form (predicate-name/address) and are accessed by name (and not by offset). Thus the binding name/address for a call is computed at run-time via a look-up performed on the p-tables associated with the units belonging to the current hierarchy. Furthermore, in order to allow the integration with more dynamic composition mechanisms (the *dynamically-linked-units* considered in [8]), the p-tables are embedded into the data structures held on the local stack. The performance of the resulting system is thus seriously affected by several factors: the intrinsic inefficiency of the look-up, a significantly increased growth of the local stack, and finally by the the fact that the modified structure of the control component disallows in several cases the application of LCO.

The extended WAM of [8] is however well-suited for implementing several composition mechanisms, other than inheritance, defined on top of Contextual Logic Programming ([11]). In particular, the run-time look-up used to compute the bindings names/addresses allows one to dynamically modify the definition associated with a predicate call. This is accomplished in [8] by compiling an extension goal $u \gg G$ as a sequence of instructions that first *extend* the current hierarchy with u and then evaluate G in the new hierarchy. A look-up model for the implementation of Contextual Logic Programming has also been proposed in [6], where a cache is used to speed-up the evaluation by avoiding the look-up whenever possible.

A different technique for implementing Contextual Logic Programming and its variations described in [8] is proposed in [5]. Here the idea is to keep a hash table (called *context*) defining the bindings for each predicate call at each stage of the computation. Upon evaluating an extension goal the context is updated so as to reflect the modifications of the bindings encompassed by the extension goal. This update is realized using essentially the same technique described for the construction of our p-tables. The evaluation of a goal uses an indirect call mechanism to retrieve from the context the address of the associated definition. In contrast to our case, the context is updated dynamically and is accessed by hashing on the name of the predicate (and not by offset).

Essentially the same technique is used in [7] for implementing the language of *embedded implications* described by Miller in [10]. The peculiar feature of the solution proposed in [5] is that contexts are not hidden in the implementation but, instead, they can be referred to explicitly within the language. The implementation relies, as in our case, on the module facility provided by SICStus Prolog. Contexts are implemented as SICStus modules, they can be given names and these names can be used to instantiate variables. The remarkable consequence is that the operations for manipulating the context can be implemented as SICStus built-in's and do not require any extension of the SICStus engine.

6 Conclusion

We have described an extended logic language, SelfLog, which offers built-in mechanisms for inheritance (early and late binding) and message-passing.

Our focus has been on the implementation. We have described an architecture for SelfLog based on the WAM which has been extended to support these new mechanisms, and we have outlined the current implementation built on top of the SICStus Prolog engine. The proposal is designed to ensure that the extensions have a minimal impact on the original design of the WAM, so that all the optimizations that are crucial to the efficiency of the resulting system can be applied. We have also discussed some optimizations of the architecture and we have addressed an independent extension to allow extension and overriding modes for inheritance to coexist in the language.

Acknowledgements. This work has been partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR under grant 91.00898.PF69. Thanks are due to Antonio Natali from DEIS in Bologna, for first pointing out the idea which inspired this work, and to Andrea Omicini, also from DEIS, for his help in the implementation. Special thanks are due to DS Logics s.r.l – Bologna, for making the experimental implementation possible on their version of the SICStus-Prolog system.

References

[1] H. Aït-Kaci. *Warren's Abstract Machine: a Tutorial Reconstruction*. MIT Press, 1991.

[2] A. Bossi, M. Bugliesi, G. Gabbrielli, G. Levi, and M. C. Meo. Differential Logic Programs. In *Proceedings of POPL'93*, pages 359-370. ACM, 1993.

[3] A. Brogi, E. Lamma, and P. Mello. Objects in a Logic Programming Framework. In *Proc. 1st and 2nd Russian Conf. on Logic Programming*, Lecture Notes in AI, number 52, Springer-Verlag, 1992.

[4] M. Bugliesi. A Declarative View of Inheritance in Logic Programming. In *Proceedings of JICSLP'92*, pages 113-127, MIT Press, 1992.

[5] E. Denti, A. Natali and A. Omicini. From Modules to Structured Theories and Logic Objects: a Common "program-based" run-time Support. In *these Proceedings*.

[6] A. M. Dias. An Implementation of a Contextual Logic Programming System. Ms Thesis - Comp. Sci. Dept. Universidade Nova de Lisboa (in portuguese), Lisbon, 1990.

[7] B. Jayaraman and G. Nadathur. Implementation Techniques for Scoping Constructs in Logic Programming. In *Proceedings of the 8th ALP Int. Conf. on Logic Programming*, pages 871-886, MIT Press, 1991.

[8] E. Lamma, P. Mello, and A. Natali. An Extended Warren Abstract Machine for the Execution of Structured Logic Programs. *Journal of Logic Programming*, (14):187–222, 1992.

[9] F. G. McCabe. *Logic and Objects. Language, Application and Implementation.* PhD thesis, Dept. of Computing, Imperial College of Science and Technology, Univ. of London, 1988.

[10] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming,* 6(2):79–108, 1989.

[11] L. Monteiro and A. Porto. Contextual Logic Programming. In *Proc. 6th Int. Conf. on Logic Programming,* pages 284-302, MIT Press, 1989.

[12] L. Monteiro and A. Porto. A Transformational View of Inheritance in Logic Programming. In *Proc. 7th Int. Conf. on Logic Programming,* pages 481-494 MIT Press, 1990.

[13] L. Monteiro and A. Porto. Syntactic and Semantic Inheritance in Logic Programming. In *Proc. of the PHOENIX Seminar on Declarative Programming,* (to appear).

To Trim or not to Trim?

Jacques Noyé
European Computer-Industry Research Centre
Arabellastr. 17
D-8000 Munich 81, Germany

Abstract

In an attempt to further reduce memory consumption and improve locality of reference, the Warren Abstract Machine (WAM) introduces trimming as a generalization of last call optimisation (LCO). As implemented in the WAM, trimming has however some drawbacks. As a result, many variants, including going back to pure LCO, have been suggested. This paper reviews the variants known to the author, pinpointing basic mechanisms and trade-offs. In particular, two basic trimming strategies are singled out: an eager strategy maintaining an up-to-date value of the top of the current environment frame and a lazy strategy delaying trimming until garbage collection. We also briefly discuss three ways of removing unsafe variables: delayed trimming, global analysis, and eager globalization of unbound variables. The paper includes data on the reduction of stack size enabled by trimming and on cost of a higher number of unsafe variables in trimming.

1 Introduction

In Prolog implementations based on the Warren Abstract Machine (WAM), *trimming*, a generalization of *tail recursion optimisation* (TRO), or more precisely *last call optimisation* (LCO), makes it possible to reuse permanent variable slots in the environment frames as soon as these slots are no longer needed, as well as detect garbage linked to these slots. The gain over pure last call optimisation is however not immediately clear. Firstly, the management of the environment stack is more costly. Secondly, avoiding dangling references via *unsafe* variables introduces an a priori significant overhead.

In the vein of [11], this paper reviews various ways of implementing LCO and trimming in WAM-like abstract machines, pinpointing basic mechanisms and trade-offs. It is organized as follows: Section 2 is a short reminder of how trimming is achieved in the WAM. Section 3 introduces basic variants including the introduction of an environment size pointer, eager trimming, and GC trimming. From the basic variants, Section 4 builds more sophisticated ones using instruction specialisation, an explicit top of the stack pointer, and delayed environment creation. Section 5 looks at the gain of environment size enabled by trimming. Finally, Section 6 focuses on the cost of unsafe variables. Section 7 concludes the paper.

We assume familiarity with the WAM [23, 1].

2 Trimming in the WAM

2.1 The Implementation

The basic idea of LCO, not specific to Prolog, is that, thanks to a continuation passing mechanism and argument passing via registers, procedure activation frames can be deallocated earlier

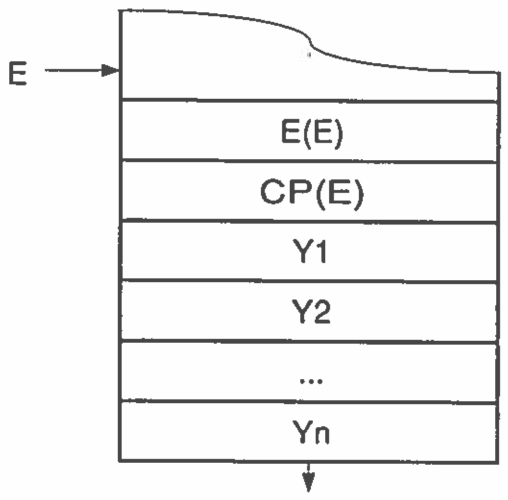

Figure 1: WAM - Environment Layout

than on procedure return: after argument passing to the last call has taken place. The main benefit of LCO is a drastic saving of stack and heap space. In particular, iterative (deterministic) procedures can be executed with constant stack space. The saving of heap space is due to the fact that compound terms initially referenced from deallocated frames can be recovered via garbage collection.

The seminal report on the WAM [23], as well as an older design including LCO [22], show that an efficient implementation of LCO was a fundamental concern when designing the WAM. As a result, the WAM introduces trimming, a generalization of LCO whereby activation frames are trimmed during execution by discarding variables no longer needed.

We shall now look at the details of trimming, assuming that the reader has a basic understanding of the basic WAM principles, in particular its dual continuation/backtracking stacking mechanism, and its implementation of the logical variable.

Let us quickly remind that deallocation of (part of) an environment (our previous activation frame) on reaching the last goal of a clause does not mean that memory can be reused. Indeed, if the execution is non-determinate, i.e. there are still alternatives left within the clause, the contents of the environment (both continuation and binding information) will be needed again on backtracking. As a matter of fact, the top of the stack `a` is always given by `a = max(e, b)`, where `e` and `b` are the top of the current environment and the top of the current choice point, respectively.

Also, an early recovery of permanent variables (the variables allocated in the environment) makes it necessary to set up a non-trivial binding scheme together with some run-time checking on argument passing to avoid dangling references. In particular, *unsafe* variables may have to be *globalized*, i.e. copied onto the heap, if it turns out that they have not been bound before executing the goal in which they occur last.

As for the details, the environments is structured as given in figure 1. The stack is assumed to grow towards higher addresses, towards the bottom of the page on the figures. Continuation information (the continuation program pointer `CP` and the environment pointer `E`) is pushed onto the stack first, and the permanent variables are ordered according to their last occurrence in the clause, with the ones to be deallocated first closer to the top of the environment.

```
allocate
        oldE := E
        E := max(B, oldE + env_size(CP))
        CP(E) := CP
        E(E) := oldE

deallocate
        CP := CP(E)
        E := E(E)

call proc_address, n
        CP := CP + 1
        P := proc_address
```

Figure 2: WAM - Control Instructions Involved in Trimming

The environment pointer E is made to point to the bottom of the environment[1]. As a result, it does not need to be updated each time trimming takes place, and the permanent variables can then be accessed via a fixed offset from E.

Abstract code for the control instructions affected by trimming, i.e. the instructions dealing with environment allocation and deallocation, and non-terminal calls, are given in Figure 2.

Following [23], each `call` instruction includes a parameter giving the number `n` of permanent variables occurring in the tail of the clause. Indeed, when executing the i^{th} call instruction `call proc_address, n(i)`, the permanent variables that occurred last in the goal `i` have just been passed via the argument registers. The corresponding cells on top of the stack, between `E + 2 + n(i - 1)` and `E + 2 + n(i)`, are not necessary any longer, and the new top of the environment is given by `E + 2 + n(i)`.

With the choice point frame pointer B pointing to the top of the current choice point[2], the top of the stack is then given by `max(E + env_size(E), B)`, i.e. `max(E + 2 + n, B)`, where `n` is the current number of permanent variables. The trick is then that, by definition of CP, `n` is available from the `call` instruction preceding the instruction pointed to by CP!

Note that, practically, `call` instructions should rather be passed a parameter `env_size` equal to `n + 2` in order to avoid computing `env_size(E)` at run-time. Also, in `allocate`, the new value of E should be computed as **if** `E < B` **then** `E := B` **else** `E := oldE + env_size`. We will keep the formulation as a `max` operation for the sake of conciseness.

2.2 Drawbacks

There are two basic reasons for reconsidering trimming as implemented in the WAM.

Firstly, the computation of the top of the stack via the continuation pointer breaks the clean

[1]The bottom of a frame is also the top of the previous frame. The top of a frame can be chosen to correspond either to the frame item pushed last onto the stack or to the first free stack slot. [23] chose the former. This choice has no influence on the discussion.

[2]In [1], B points to the bottom of the choice point for didactic reasons (similarity between B and E). This however makes computing the top of the stack more costly, increases the size of choice points by one slot, and requires the introduction of a new register. The latter point could actually be avoided.

```
allocate
        oldE := E
        E := max(B, oldE + N)
        CP(E) := CP
        E(E) := oldE
        E(N) := N

deallocate
        CP := CP(E)
        E := E(E)
        N := N(E)

call proc_address, env_size
        CP := CP + 1
        P := proc_address
        N := env_size
```

Figure 3: WAM + Environment Size Register - Instructions

conceptual separation between code and data, which is not without practical impact. Let us, for instance, assume native code running on a Harvard architecture (separate code and data cache). The data cache is clearly going to be perturbated by scattered accesses to code memory. A more extreme case is provided by dedicated machines such as the PLM [8, 7]. For the sake of simplicity, the PLM does not include any general caching mechanism; read access to code is restricted to instruction fetching in the prefetch unit and hash table access by some indexing instructions. Code manipulation, e.g. assert/retract, is done on the host (the PLM is a co-processor). Even if (Prolog) code and data are cached together (byte-code compilation, or native code and single instruction/data cache), one may wonder about the locality of access to continuation code.

Secondly, compared to LCO, trimming increases the number of unsafe variables, hence a possibly significant execution overhead due to `put_unsafe` instructions replacing simpler `put_value` instructions.

3 Basic Variants

There are basically three ways of removing code access without altering the positive effects of trimming: adding a new state register holding the environment size, *eager trimming*, and *GC trimming*.

Environment Size Register The size of the current environment can be stored in a new state register `N` as part of the execution of `call` instructions. This register has then to be saved in each environment and choice point frame to keep the state consistent when backtracking and deallocating environments; the size of the environment becomes part of the continuation

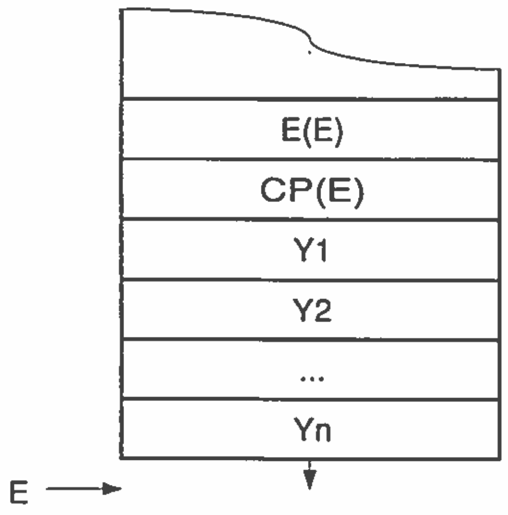

Figure 4: Eager Trimming - Environment Layout

```
allocate env_size
        a := max(E, B)
        [a+] := E
        [a+] := CP
        E := a + env_size

deallocate size_decrement
        E := E - size_decrement
        CP := CP(E)
        E := E(E)

call proc_address, size_decrement
        E := E - size_decrement
        CP := P + 1
        P := proc_address
```

Figure 5: Eager Trimming - Instructions

information together with `E` and `CP`. This is the approach followed in the PLM [8, 7] and its successor X-1 [6, 24]. The environment layout is similar to the WAM layout. The instructions `allocate`, `deallocate`, and `call` are modified as indicated in Figure 3.

Eager Trimming `E` can also be defined to point to the top of the environment frame, and modified eagerly, as parts of the environment are not needed any longer, by the call instructions (with the exception of the first call in the clause). Pure LCO, i.e. LCO without trimming, is performed as in the WAM. With `E` pointing to the top of the environment frame, computing the top of the stack is much simpler: `a = max(E, B)`. The price to be paid is that permanent variables cannot be accessed via fixed offsets from `E` any longer. The offsets change from one goal to another. However, this does not slow down the execution; the offsets can be computed at compile-time. In the following, this solution will be referred to as *eager trimming*. The corresponding environment layout and code are given in Figures 4 and 5. Note that the parameter `size_decrement` of `deallocate` is merely used to access `E` and `CP`; it has no trimming function. Also, the value of `size_decrement` is zero at the first call (the whole environment is needed).

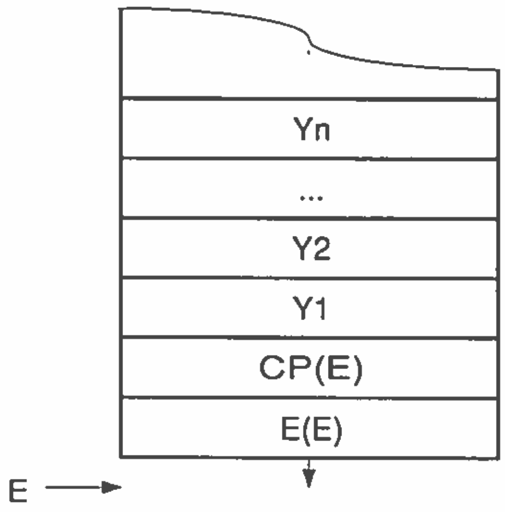

Figure 6: GC Trimming - Environment Layout

GC Trimming A more drastic approach, *GC trimming*, consists of delaying trimming until garbage collection. That is, there is no attempt to trim the environment during normal execution but the garbage collector is given the means to tell between dead and alive permanent variables. Garbage linked to dead permanent variables can then be reclaimed and the local stack possibly compacted. If the garbage collector is not activated, GC trimming boils down to pure LCO.

The basic implementation of GC trimming is very similar to eager trimming, see the environment layout and instructions in Figures 6 and 7. `E` can, as in eager trimming, point to the top of the environment, making the computation of the top of the stack easy. The environment layout is different in that continuation information can now be stored at the top of the environment; it can easily be accessed on environment deallocation. Also, the permanent variables are ordered in reverse order, compared to eager trimming (and the WAM), so that the bottom of the active part of the environment can be computed by the garbage collector (as `E - env_size(CP)`). Finally, there is no need to update `E` on each `call` instruction, which means that permanent variables are accessed via fixed offsets from `E`.

```
allocate env_size
        a := max(E, B) + env_size
        [a+] := E
        [a+] := CP
        E := a

deallocate
        CP := CP(E)
        E := E(E)

call proc_address, env_size
        CP := P + 1
        P := proc_address
```

Figure 7: GC Trimming - Instructions

4 More Sophisticated Variants

Previous sections covered three basic ways of handling environments and trimming in WAM-like abstract machines[3]. We shall now build more sophisticated variants by introducing specialisation, an explicit top of the stack, and delayed environment creation.

4.1 Specialisation

Specialisation is a well-known principle of instruction set design. In [1], it is referred to as *WAM Principle 3* and defined as follows:

> Particular situations that occur very often, even though correctly handled by general-case instructions, are to be accommodated by special ones if space and/or time may be saved thanks to their specificity.

A basic example is the introduction of a `call_first` instruction in eager trimming. Such an instruction avoids the computation `E := E - 0` on first calls.

Another example, which can be applied to all the previous variants, consists of merging `deallocate` and `execute`, the instruction implementing terminal calls, into a `call_last` instruction. In that case, a frequent sequence of instructions is specialised into a single instruction. The benefit is twofold: the code is more compact and, more importantly, it uncovers potential micro-parallelism, e.g. by filling pipeline breaks with different threads of computation. For instance, the WAM versions of `deallocate`, `execute`, and `call_last` would cost on KCM 4, 3, and 4 (instead of 7) cycles, respectively. Such effects can typically be obtained on current

[3]A missing combination consists of laying out the environment frame so that E points to the bottom of the frame, with the permanent variables at the bottom of the frame, and E and CP at the top.

```
allocate env_size
        [A+] := E
        [A+] := CP
        E := A + env_size

call_first proc_address
        A := E
        CP := P + 1
        P := proc_address

call proc_address, size_decrement
        E := E - size_decrement
        A := max(E, B)
        CP := P + 1
        P := proc_address

call_last proc_address, size_decrement
        E := E - size_decrement
        CP := CP(E)
        E := E(E)
        A := max(E, B)
        P := proc_address
```

Figure 8: Cheap Trimming - Instructions

general purpose machines via the use of delayed branches, and may apply to both native code and byte-code emulators[4].

4.2 Explicit Top of the Stack

An alternative to computing the top of the stack on demand, when pushing a new frame onto the stack, is to maintain an up-to-date value of the top of the stack in a new state register `A`.

Applying this idea together with specialisation to eager trimming directly leads to *cheap trimming*, the scheme presented in [11]. The corresponding instructions are given in Figure 8.

The overall benefit of cheap trimming over eager trimming is not clear. It depends on low-level considerations like the cost of the `max` operation[5], whether `A` can be stored in a hardware register, and so on. It also depends on the program. On the one hand, the top of the stack has not to be recomputed via an a priori costly `max` operation on pushing a new frame on the stack. Moreover, this operation is pushed towards the end of the clause, which may be beneficial in case of an early failure. On the other hand, additional `max` operations have to be performed on intermediate procedure calls, and computing the top of the stack eagerly may be wasted time in case failure occurs before any other frame has been pushed onto the stack.

[4]One argument sometimes raised against specialisation in byte-code emulators is however that it makes the emulator bigger, possibly degrading its code reference behaviour.

[5]On KCM, for instance, the `max` operation can be microcoded and take place in parallel with a jump; it is

```
allocate
        [A+] := E
        [A+] := CP
        E := A

call proc_address, env_size
        A := max(B, E + env_size)
        CP := P + 1
        P := proc_address

call_last proc_address
        CP := CP(E)
        E := E(E)
        A := max(B, E + env_size(CP))
        P := proc_address
```

Figure 9: WAM with Top of Stack Register - Instructions

```
allocate env_size
        [A+] := E
        [A+] := CP
        E, A := A + env_size

call proc_address, env_size
        CP := P + 1
        P := proc_address

call_last proc_address
        CP := CP(E)
        E := E(E)
        A := max(E, B)
        P := proc_address
```

Figure 10: GC Trimming with Top of Stack Register - Instructions

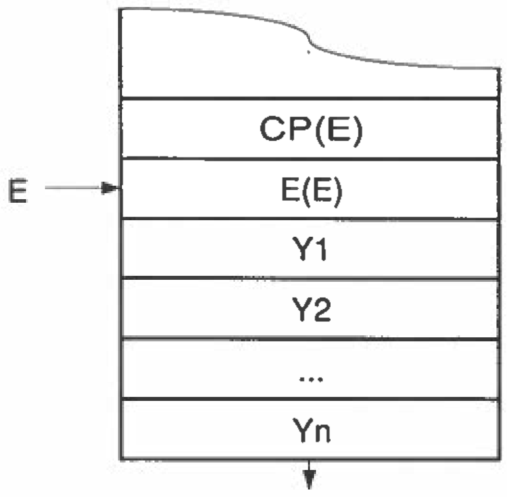

Figure 11: Standard Activation Frame - Environment Layout

```
allocate
        [A+] := E
        E := A

call proc_address, env_size
        A := max(B, E + env_size)
        [A+] := P + 1
        P := proc_address

call_last proc_address
        cp := CP(E)
        E := E(E)
        A := max(B, E + env_size(cp))
[A+] := cp
        P := proc_address
```

Figure 12: Standard Activation Frame - Instructions

Of course, the same modifications can be applied to the other variants. For comparison purposes, Figures 9 and 10 give the instructions corresponding to modified WAM and modified GC trimming.

Let us mention, for the sake of completeness, that a slight modification of the WAM environment layout and instructions leads to handle environment frames as standard procedural activation frames, see Figures 11 and 12, to be compared with Figures 1 and 9. The idea is to use the machine stack as the Prolog stack; A is mapped to the top of the machine stack[6],

relatively less costly in a call_last instruction than in an allocate instruction.

[6] A points then to the last item pushed onto the stack. To facilitate comparisons, we have ignored the fact that

```
preallocate
        E2 := max(B, E + env_size(CP))

postallocate
        CP(E2) := CP
        E(E2) := E
        E := E2

proceed
        E2 := E
        P := CP
```

Figure 13: WAM and Delayed Environment Creation - Instructions

the register CP disappears, and env_size corresponds to the number of permanent variables in the environment. This variant is mainly interesting on traditional CISC machines, with small hardware register files and support for manipulating the return address on the stack.

Sepia [13], whose initial target machine was the Motorola 68020, combines a standard activation frame layout with GC trimming.

4.3 Delayed Environment Creation

Delayed environment creation is inspired by a general principle of procrastination (and its dual principle of eager consideration [9]): focus on operations which may result on earlier failure and delay the operations which have to be done in any case.

We will assume that the environment may be partially filled with permanent variables during unification. This assumption is basically due to the shortage of hardware registers; in general, there are not enough hardware registers to cache all the permanent variables. The issue is therefore to delay the saving and updating of E and CP until the occurrence of the first permanent variable.

Mats Carlsson [3] suggests the following minimal modification of the WAM. The environment layout (Figure 1) does not change. The allocate instruction is split into a preallocate and postallocate instruction, see 13. The preallocate instruction computes the top of the stack which is stored in an intermediate register E2 used to access newly created permanent variables. The postallocate instruction, placed just before the first call instruction (i.e. one can apply specialisation here) fills in the environment with continuation information, and updates E. Note that E2 is also reset to E on returning from a halting clause (proceed instruction). This makes it possible to use a single base address to access permanent variables.

Note that, accessing permanent variables from two different base addresses, one until the first call (included) and one for the rest of the body, can be implemented without much difficulty; the offsets can be computed at compile-time, and the instructions accessing permanent variables updated to include an additional parameter indicating the base address (byte-code emulators may incur a small overhead though).

In the Or-parallel system Aurora/Foxtrot [4], the same scheme is used, but applied to the

the machine stack usually grows towards lower addresses.

```
call_first proc_address
        [A+] := E
        [A+] := CP
        E := A
        CP := P + 1
        P := proc_address

call proc_address, env_size
        CP := P + 1
        P := proc_address

call_last proc_address
        CP := CP(E)
        E := E(E)
        A := max(E, B)
        P := proc_address
```

Figure 14: GC Trimming with Delayed Environment Creation - Instructions

WAM with a top of stack register. The introduction of binding arrays to record bindings to variables shared by several parallel OR-branches complicates trimming in a number of ways. In particular, trimming forces tidying up the trail on pruning, an operation which in a sequential context is best left to the garbage collector.

The same kind of scheme can also easily be combined with eager and GC trimming. The environment layout of GC trimming favours however a slightly different approach, namely to first fill the binding part of the environment, accessing permanent variables already pushed onto the stack via the top of the stack. This means that offsets may change from one access to the other, but, again, all the offsets can be computed at compile-time. The `allocate` instruction can then be completely merged together with the first `call` instruction. This leads to the code of Figure 14. This is the solution which was chosen for KCM [2, 15].

4.4 Relationship with Or-control

Split-stack Architectures Both the abstract machines of KCM and SICStus/Aurora have a split-stack architecture, i.e. an environment stack and a choice point stack. This architecture was initially mentioned in [19] as a way of increasing the stack locality. It has also the good property of allowing the cut operator to recover buried choice points, as evoked in [18], and is a prerequisite for shallow backtracking support [3, 12].

With such an architecture, the top of the environment stack is given by `a = max(e, eb)` where `eb` marks the place where the current choice point would have been in a single-stack architecture, i.e. the top of the environment stack when the choice point was created [18]. `e` can also be described as the environment backtrack point, by analogy with the heap backtrack point of the WAM.

The issue of choosing between a single-stack and a split-stack architecture and the issues discussed above are largely decoupled. Some details of the split-stack implementation may however influence the relative efficiency of the different environment handling solutions. For

Benchmark	max sampled size (Prolog words)	gain of compaction (%)		
		on max size	average	std. deviation
boyer	487	19	18	3
browse	1441	0	3	1
chat_parser	667	4	4	1
crypt	43	0	0	0
deriv	55	0	0	0
fast_mu	109	3	0	1
flatten	29	0	0	0
meta_qsort	2048	0	0	0
mu	52	15	13	4
nand	1970	1	1	1
nreverse	142	0	0	0
poly_10	95	9	7	3
prover	42	17	8	8
qsort	63	8	4	3
queens_8	69	0	0	0
query	32	0	0	1
reducer	483	23	14	11
sdda	43	7	3	3
sendmore	29	0	0	0
serialise	31	10	8	4
simple_analyser	161	6	4	3
tak	186517	5	4	1
unify	144	1	1	0
zebra	31	0	0	0

Table 1: Environment Compaction Enabled by Trimming

instance, on creating a choice point, the top of the environment stack `eb` has to be saved in the choice point (this replaces the computation of the top of the stack in the single-stack model). This value can be cached in a register (as the heap backtrack point). If it is not, the computation of `max(e, eb)` requires an additional memory access.

Overlapping Choice Point and Environment The idea is to share continuation and cut information between a consecutive choice point and environment belonging to the same procedure call. This obviously requires a single-stack architecture and continuation information at the bottom of the environment. This organisation is studied in [11].

5 Stack Compaction Enabled by Trimming

Sepia and KCM, the two implementations of GC trimming known to the author, do not compact the stack, based on the intuition that the small loss of stack size is not worth the cost. This intuition is supported by Table 1.

Each Aquarius benchmark [21] was run on a KCM, in a loop, and sampled every second from KCM's host. That is, the host would stop the KCM, examine the stack, compute the amount of memory occupied by environments and, within these environments, by dead permanent variables, and finally resume the execution. 100 samples were taken for each program, with the exception of tak for which only 12 samples were taken. It turns out that the initial version of tak [17] was slightly modified in order to exercise determinism extraction. Standard compilers are not able to recognize the determinism of the program any longer, with the result that many choice points are created at run-time, preventing LCO from recovering many environments.

For each program, the table gives the maximum environment stack size sampled, the gain that would have been achieved by stack compaction on the corresponding sample, the average gain on all the samples, and its standard deviation. The programs were compiled with the option in-line generic arithmetic.

The table shows gains much lower than the 25% to 30% figures put forward by [11].

Benchmark	expanded			in-line			integer		
	delayed	standard	%	delayed	standard	%	delayed	standard	%
boyer	3.9744	4.1058	3	3.6376	3.7634	3	3.4599	3.5865	4
browse	2.5460	2.5461	0	2.4997	2.4997	0	2.4821	2.4822	0
chat_parser	0.3739	0.3761	1	0.3741	0.3759	0	0.3737	0.3759	1
crypt	0.0293	0.0298	2	0.0170	0.0170	0	0.0145	0.0145	-0
deriv	0.0003	0.0003	-0	0.0003	0.0003	-1	0.0003	0.0003	0
fast_mu	0.0055	0.0055	0	0.0034	0.0034	0	0.0027	0.0027	0
flatten	0.0107	0.0108	1	0.0098	0.0099	0	0.0095	0.0096	1
meta_qsort	0.0310	0.0260	-16	0.0254	0.0252	-1	0.0249	0.0248	-0
mu	0.0023	0.0023	-1	0.0022	0.0021	-5	0.0021	0.0021	0
nand	0.1200	0.1168	-3	0.0940	0.0944	0	0.0893	0.0898	1
nreverse	0.0008	0.0008	-0	0.0008	0.0008	-0	0.0008	0.0008	0
poly_10	0.2496	0.2501	0	0.2253	0.2252	-0	0.2072	0.2072	-0
prover	0.0029	0.0030	3	0.0030	0.0029	-3	0.0028	0.0028	0
qsort	0.0031	0.0032	1	0.0017	0.0017	1	0.0014	0.0014	1
queens_8	0.0321	0.0328	2	0.0138	0.0139	0	0.0059	0.0059	-0
query	0.0276	0.0277	0	0.0169	0.0169	-0	0.0130	0.0130	0
reducer	0.2729	0.2742	0	0.2598	0.2602	0	0.2497	0.2577	3
sdda	0.0218	0.0218	0	0.0212	0.0213	1	0.0213	0.0219	3
sendmore	0.9498	0.9575	1	0.3593	0.3626	1	0.1397	0.1430	2
serialise	0.0019	0.0019	0	0.0016	0.0016	2	0.0014	0.0014	1
simple_analyser	0.1461	0.1368	-6	0.1134	0.1174	4	0.1105	0.1140	3
tak	2.0302	2.0386	0	1.2820	1.2820	0	1.0716	1.0716	0
unify	0.0139	0.0142	2	0.0100	0.0102	2	0.0084	0.0087	4
zebra	0.2277	0.2841	25	0.3712	0.2277	-39	0.2274	0.2275	0
average			1			-1			1
std. deviation			6			8			1

Table 2: Overhead of put_unsafe Instructions

6 The Cost of Unsafe Variables

Both KCM and Sepia rely on *delayed trimming* [1][7] to keep the same set of unsafe variables as with LCO. Delayed trimming should not be confused with GC trimming. It consists of delaying trimming a variable until just *after* the goal in which the variable occurs last. The variable is then safe; the WAM binding scheme guarantees that no reference to the variable can outlive the goal execution. As far as memory consumption is concerned, delayed trimming is conservative. Though the variable becomes garbage during the execution of the goal, this garbage and the garbage connected to it cannot be reclaimed until the goal has been solved (or has failed).

In order to get a feel for the overhead due to an increased number of unsafe variables, the KCM compiler was modified to generate put_unsafe instructions as in standard trimming. Table 2 gives in % the overhead of the corresponding code compared to delayed trimming on the Aquarius benchmarks. Execution times are in seconds. They have been obtained by

[7]Delayed trimming is introduced in [1] as a way to fix a slight inconsistency between the WAM variable classification and trimming.

incrementally compiling and running each program one after another in a repeat/fail loop such that each program runs about 20 seconds. Different compilation options were used: expanded, generic in-line, and generic integer arithmetic. As can be seen, the execution overhead is actually not significant.

We conjecture that similar results would be obtained when looking at the impact of increasing the number of unsafe variables, hence, possibly, the number of globalizations, on the size of the heap.

Let us note here, that the KCM implements a variant of [1] which consists of including a preliminary determinism test in `put_unsafe`. In non-determinate states, `put_unsafe` behaves as `put_value`; unnecessary globalizations are avoided. In terms of speed, the instruction is speeded up in non-determinate cases, but slowed down in determinate cases.

There are two other possible answers to the question of the cost of unsafe variables. The first one is that global analysis should make it possible to remove most unsafe variables. This possibility was already evoked in [22]. The second answer is that one can simply remove unsafe variables by globalizing unbound permanent variables by default. Such a scheme has been implemented in the VAM [10] and Aquarius [20]. Execution is speeded up (trailing is simplified, `put_unsafe` and `unify_local_value` instructions disappear) at the cost of an increase in heap size. The magnitude of the increase is an open issue, see [14].

7 Conclusion

Two potential drawbacks of trimming as implemented in the WAM have been described, justifying the investigation of alternative implementations.

The first drawback, systematic code access when pushing a new frame onto the stack, can be solved efficiently either by making E points to the top of the environment frame and updating it eagerly, or by delaying trimming until GC. Both solutions can be used in their basic form, with delayed environment creation, or an explicit top of stack, if there are other benefits in introducing such a register. The trade-off is then between speed (GC trimming without stack compaction) and reclaimed memory (eager trimming). We expect both solutions to have similar performance. In particular, memory leaks due to not compacting the stack are typically small.

The second drawback, namely a higher number of unsafe variable is of limited significance. Switching from LCO to trimming (as far as unsafe variables are concerned) does not seem to impact the performance of KCM in a noticeable way. We conjecture that this would apply to most standard Prolog systems, which are based on an equivalent compiler technology. Moreover, unsafe variables can, at least partly, be eliminated by global analysis. Finally, it is possible to trade memory for speed, either via delayed trimming, or an eager globalization of unbound variables.

As a result, the potential overhead incurred by the implementation of trimming does not justify switching back to LCO, taking the risk of important memory leaks due to inaccessible garbage on the heap.

Let us finally mention that "to trim or not to trim may" not be the question as the future arguably belongs to heap-based systems [5, 16].

Acknowledgements

The author would like to thank Mireille Ducassé and Evan Tick for their comments on drafts of this paper as well as Joachim Schimpf for fruitful discussions.

References

[1] H. Ait-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction.* The MIT Press, 1991.

[2] H. Benker, J.-M. Beacco, S. Bescos, M. Dorochevsky, Th. Jeffré, A. Pöhlmann, J. Noyé, B. Poterie, A. Sexton, J.-C. Syre, O. Thibault, and G. Watzlawik. KCM: A Knowledge Crunching Machine. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 186–194. IEEE, June 1989.

[3] M. Carlsson. On the efficiency of optimising shallow backtracking in compiled Prolog. In G. Levi and M. Martelli, editors, *Proc. of the 6th Int. Conf. on Logic Programming*, pages 3–16. MIT Press, 1989.

[4] M. Carlsson and P. Szeredi. The Aurora abstract machine and its emulator. Research Report R90005, SICS, 1990.

[5] B. Demoen and A. Mariën. Implementation of prolog as binary definite programs. In A. Voronkov, editor, *Logic Programming, Proceedings of the 1st and 2nd Russian Conference on Logic Programming*, number 592 in Lecture Notes in Artificial Intelligence, pages 165–176. Springer-Verlag, 1992.

[6] Tep Dobry. A coprocessor for AI; LISP, Prolog and data bases. In IEEE, editor, *Compcon 87*, pages 396–402, February 1987.

[7] T.P. Dobry. *A High Performance Architecture for Prolog.* Parallel Processing and Fifth Generation Computing. Kluwer Academic Publishers, 1990.

[8] T.P. Dobry, A.M. Despain, and Y.N. Patt. Performance studies of a Prolog machine architecture. In *The 12th Annual International Symposium on Computer Architecture*, pages 180 – 190. IEEE/ACM, IEEE Computer Society Press, June 1985.

[9] R. Kowalski. *Logic for Problem Solving.* Artifical Intelligence Series. North-Holland, 1979.

[10] A. Krall and T. Berger. Fast prolog with a VAM1p based prolog compiler. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, number 631 in Lecture Notes in Computer Science, pages 245–259, Leuven, 1992. Springer-Verlag.

[11] A. Mariën and B. Demoen. On the Management of Choicepoint and Environment Frames in the WAM. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 1030–1050. MIT Press, 1989.

[12] M. Meier. Shallow bactracking in Prolog programs. Technical report, ECRC, April 1989.

[13] M. Meier, A. Aggoun, D. Chan, D. 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 11th World Computer Congress IFIP'89.* IFIP, August 1989.

[14] Net Talk, term comparisons with variables. ALP Newsletter, Volume 5/4, November 1992.

[15] 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, Forthcoming.

[16] P. Tarau. Ecological memory management in a continuation passing Prolog engine. In Y. Bekkers and J. Cohen, editors, *Proceedings of the International Workshop on Memory Management (IWMM 92)*, number 637 in Lecture Notes in Computer Science, pages 344–356, September 1992.

[17] E. Tick. Lisp and Prolog memory performance. Technical Report 86-291, Stanford University, January 1986.

[18] E. Tick. *Memory Performance of Prolog Architectures.* Frontiers in Logic Programming Architecture and Machine Design. Kluwer Academic Publishers, 1988.

[19] E. Tick and D.H.D. Warren. Towards a pipelined Prolog processor. In *1984 International Symposium on Logic Programming*, pages 29–40. IEEE Computer Society Press, February 1984.

[20] P. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming.* PhD thesis, University of California, Berkeley, December 1990.

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

[22] D.H.D. Warren. An improved Prolog implementation which optimises tail recursion. In *Proceedings of the Logic Programming Workshop, Debrecen, Hungary*, July 1980.

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

[24] Xenologic Inc. *X-1 Assembly Programming Manual*, November 1987.

MProlog: an Implementation Overview

Zsuzsa Farkas & Péter Szeredi
IQSOFT
H-1142 Teleki Blanka u. 15-17
Budapest, Hungary

Péter Köves
H-1125 Béla király út 17
Budapest, Hungary

Abstract

The paper describes some interesting features of MProlog, a structure sharing Prolog implementation developed in Hungary in early eighties. Following a brief overview of system structure, we describe some special language elements of MProlog, with the emphasis placed on implementation aspects. Several extensions to Warren's structure sharing compilation model, employed in the MProlog compiler, are also discussed. Finally, a brief description of the implementation technology of MProlog is given.

1 Introduction

MProlog is one of the earliest full-scale commercial implementations of Prolog [2, 7, 8]. It was designed and implemented in Hungary in the late seventies and early eighties, as a successor to the early Hungarian Prolog interpreter of 1975 [9]. It is based on the first Warren compiler model of 1977 [10, 11]. MProlog has been ported to a wide range of different computer architectures including personal computers, workstations and mainframes. Although its age has made it obsolete in some people's eyes, the MProlog system has a number of features and uses several implementation techniques which scarcely appear in later systems. This paper is an attempt to collect some of those ideas in MProlog that may still have current interest, with the emphasis placed on the implementation issues.

2 System Structure

The MProlog system offers two paths for program execution. During development, the Program Development SubSystem (PDSS) provides an interactive environment for writing and testing programs. For application delivery the Production System is offered, which uses the traditional translate-link-execute model.

2.1 The Program Development SubSystem (PDSS)

PDSS[1] is a comprehensive environment for the development of MProlog programs providing support for program entry, editing, execution, tracing, saving, etc.

[1]MProlog PDSS predated the identically named PDSS of the FGCS project [4], with which it has no relation whatsoever.

PDSS is implemented in MProlog itself. The majority of its functionality is accessible from user programs, and its behavior can be influenced via a range of hooks. It is also available as a module that can be linked into a user program.

For the programs entered, the system retains the complete source, including variable names. PDSS represents the program as a syntax-tree, which is always presented to the user in a standard, pretty-printed format. A central concept of PDSS is that of the current *focus*, a position in the syntax-tree serving as an implicit context for subsequent commands. Another central notion of PDSS is that of the *selector*, a uniform mechanism for selecting from the syntax-tree. For example, the PDSS command

```
list Module comp* Predicate */2
```

lists all two-argument predicates in all modules whose name begins with comp. Selectors are used in most commands, such as listing, typing, editing, positioning, moving, etc.

PDSS provides an elaborate four-port tracing facility. The debugger is implemented in Prolog using exception handling and a few special external predicates. The heart of the implementation is a special predicate which causes the system to raise an exception after a given number of procedure calls. Other debugger features make use of predicates for accessing the Prolog stack, e.g. for displaying the ancestor list of a goal. Given that the MProlog interpreter is implemented in a traditional algorithmic language, it is interesting to note that such a small set of tools suffice for the implementation of a full-featured debugger.

PDSS endeavors to emulate the entire execution environment of a user program. This presents its particular problems as PDSS itself is an MProlog program run by the same Prolog machine. In a sense this makes PDSS similar to an operating system. For this reason there are two switchable global execution environments in MProlog, one for the PDSS and a separate one for the user. These execution environments contain various flags, current input-output channels, etc., so that, for example, PDSS does not have to worry about saving the current output channel, before displaying a message on the terminal.

2.2 The Production System

In the Production System (see Fig. 1), MProlog source modules are first converted into a compact internal format using the pretranslator. The translation process includes syntactic and semantic analysis (such as variable classification), and produces a binary module containing Prolog code in a format suitable for interpretation or further compilation. Pretranslated modules are linked—in MProlog terminology, consolidated—into an executable program by the consolidator program. The Consolidator also has the capability of consolidating several binary modules into a single new binary module. Since the user may give interface specifications for the resulting module (based, of course, on the interfaces of the component modules) this effectively provides a way of constructing subsystems to any level. The standard module, which is usually a required part of any binary program, contains the description and/or code for the MProlog built-in predicates. This module is constructed (from several MProlog source modules) in a way no different from user modules, as the MProlog language contains facilities for specifying the linkage to the built-in procedures. In fact, it is possible to build custom programs which use a different "standard" module specifying perhaps just a subset of the complete range of standard procedures.

The PDSS is also provided as a binary module as part of the production system, so it is possible to build user programs which make use of PDSS functionality. PDSS provides several procedures to enable utilization of some of its advanced capabilities.

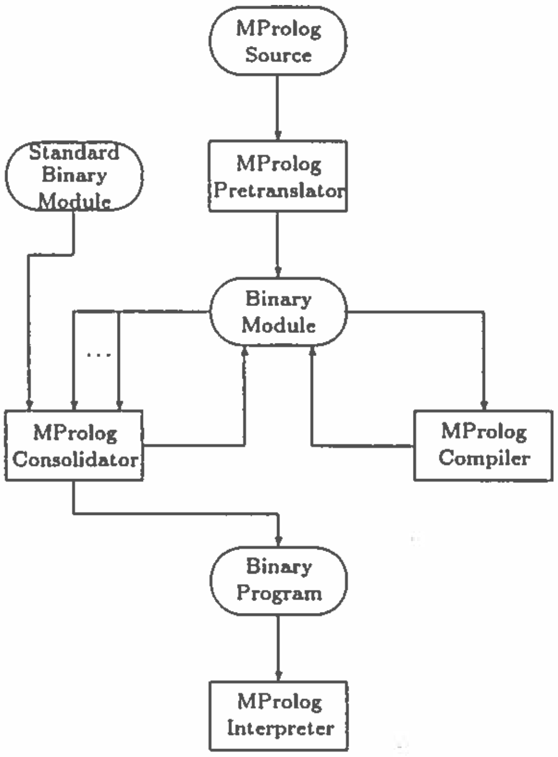

Figure 1: THE MPROLOG SYSTEM

MProlog also has a native code compiler for certain architectures (IBM 370, M68000, VAX, Intel 386), as well as a byte-code compiler. The compiler transforms a pretranslated module to a compiled module, with the resulting module also amenable to consolidation. This then, provides a means of freely intermixing compiled and interpreted code.

The compiler encompasses a number of important improvements over the original structure sharing compiler model [10, 11], including techniques similar to those used in the WAM [12] (see Sec. 4).

The binary program resulting from consolidation is executed by the interpreter. Unlike some other Prolog implementations, in MProlog the interpreter is the core of the system. It is written in CDL2 (see Sec. 5) and in addition to interpretation proper, it serves as the run-time system (i.e. memory management, etc.) for the compiled code. It is also the repository for the built-in procedures described by the standard module.

Notice that it is possible to deliver applications even in interpreted form without supplying source-code to users. This may be important in a non-academic environment.

3 Special features of MProlog language

In this section we present some of those aspects of the MProlog language which differentiate it from other Prolog implementations.

3.1 Modularity

MProlog is a modular language—hence the M in the name. Programs may be constructed from several source modules with language facilities for the specification of the interface between the modules. The MProlog module concept is name-based, i.e. all occurrences of a given atom in a module are either *hidden* from or *visible* to the outside world. A further independent property of atoms is whether they can be *coded* (i.e. replaced by internal codes during the pretranslation-consolidation process), or their *symbolic* form (i.e. the list of characters they are composed of) has to be preserved till run-time. The visibility and codedness attributes of atoms can be specified in the module interface using the above emphasized names. It may be interesting to note that a double quoted string in MProlog denotes an atom identical to its single quoted variant, but with an implicit *symbolic* declaration, thus forcing the atom to be preserved till run-time.

MProlog has a hierarchical module concept: several modules can be linked to form a new module with a possibly restricted visibility interface (see Sec. 2.2). Another interesting feature is that complete modules can be deactivated and reactivated at low cost during run time. This feature can be used to implement a simple form of program overlaying, or to help in the debugging of a program by deactivating a compiled module and replacing it by its consulted version.

3.2 Multi-level Indexing

MProlog offers a user-selectable multi-level indexing facility. By supplying a `match_order` declaration for a predicate, the user can specify for which arguments should the indexing be applied, and in what order. The pretranslator builds a complete tree of choices based on the principal functor of the argument positions specified in the match-order declaration.

```
                match_order flight(-,-,2,1).
                mode flight(-,?,?,+).

(1)             flight(ba001, daily,  london,  paris).
(2)             flight(af002, monday, london,  lyon).
(3)             flight(ba002, monday, glasgow, X) :-
                     marseille_or_paris(X).
(4)             flight(ba071, sunday, glasgow, paris).
```

Figure 2: SIMPLE MATCH_ORDERING

Let us illustrate this feature on a simple example in Fig. 2. The `match_order` declaration in the figure specifies that only the fourth and third arguments should be indexed and indexing should prefer the fourth argument over the third. The control tree for the example is given in Fig. 3.

The control tree has as many branching levels as there are argument positions to be catered for. On each level a branch of the tree corresponds to each constant (or main functor) which occurs throughout all clause heads in the given argument position. Unless the argument has mode +, an extra branch is created to cover the case when the incoming argument is a variable.

A clause which has a variable in an indexed head-position will be carried over to all branches of the level corresponding to the given position. Furthermore an "otherwise" branch is formed

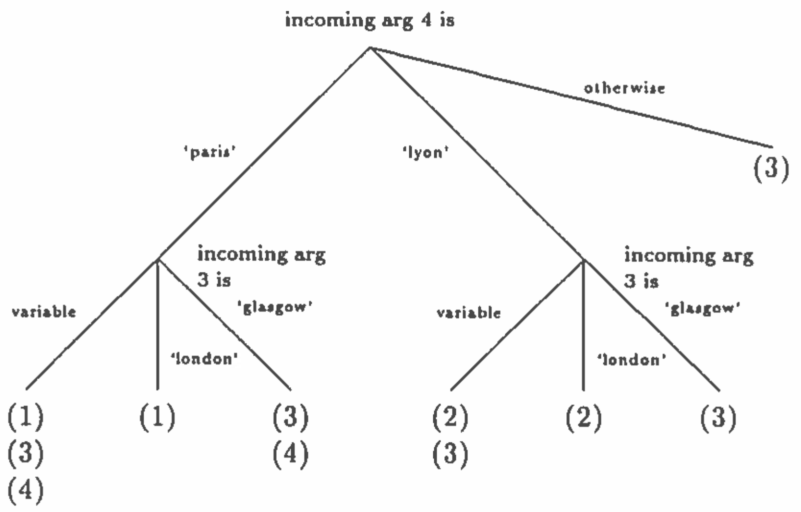

Figure 3: Control tree for a match_ordered predicate

to contain all such clauses. This branch will be entered if the incoming argument has a functor different from all explicitly mentioned ones.

3.3 Exception Handling

MProlog has an elaborate exception handling mechanism. It enables the construction of safe subsystems, which can completely protect themselves from any errors arising in other components (a typical example being the PDSS itself). A predicate may be executed in a protected environment, specifying a handler to invoke in case of an exceptional condition. Barring the "stack full" type exceptions, most of the errors can be handled locally, the call of the handler replacing the call causing the error. The handler may then correct the error and continue execution following the call in error, or wind back the stack to the surrounding protection level and repeat the exception handling procedure there.

Fig. 4 shows an example of an exception handler in MProlog. This is a four argument predicate specifying what to do in an exceptional situation of a certain kind. When an exception occurs, the system invokes the current exception handler with the first three arguments instantiated to terms describing the exception: its name, the call causing the exception, and the number of the argument responsible for the exception. If the exception handler succeeds, it returns, in its fourth argument, a term to replace the call causing the exception.

```
plus_handler(non numeric, Result is X+Y, 2, Replacement_Goal) :-
     Result is_a number,
        ( X is_a var, Y is_a number, Replacement_Goal = X is Result-Y ;
          X is_a number, Y is_a var, Replacement_Goal = Y is Result-X
        ).
```

Figure 4: An exception handler for inversion of addition

The exception handler in Fig. 4 specifies that in case of an exception called `non numeric` (note that non is a prefix operator), raised for the second argument of an additive is call, a replacement goal involving a subtraction should be executed, when the addition can be inverted. Such an exception handler can be activated for the duration of a specific subcomputation using the built-in predicate call `protected(`*Subgoal*`,plus_handler)`. Executing e.g. `12 is X+10` within *Subgoal* will thus result in `X` being instantiated to 2.

3.4 Undoable Built-in Predicates

MProlog offers a range of built-in predicates whose side-effects are undone on backtracking. There are undoable versions of assert and retract as well as of predicates implementing global (destructively assignable) variables. These groups of predicates were introduced in the early Hungarian Prolog system at the specific request of application developers, e.g. [5]. The implementation of the undo operations is safe with respect to cut, as undoing is triggered by special records saved on the trail stack.

A further interesting group of undoable built-in predicates is that of token and term input. A circular buffer is used by the MProlog tokenizer to store the tokens read in. Token and term input predicates trail the value of the current token pointer of this circular input buffer before the actual input is made. If such an input predicate is backtracked over, its side effect is undone by moving the current token pointer back to its original position (of course, this backtracking capability is limited by the size of the buffer). Subsequent input predicates will read the tokens from the buffer, and switch back to "real" input only when the sequence of unread tokens is exhausted. Undoable input is again a feature first introduced in the early Hungarian Prolog implementation and praised by numerous users as a tool more convenient than DCGs for writing parsers in Prolog.

Fig. 5 shows a simple grammar fragment together with its parser written in MProlog. Here `read_token` is the undoable built-in predicate for reading in a single token, while `read_args` and `read_expr` are user-defined predicates for parsing the appropriate grammar part. All in all, the main advantage of these predicates is that one need not bother with DCGs and their paraphernalia for simple parsing applications.

```
⟨statement⟩ ::=                          read_statement(S):-
   ⟨procedure call⟩ |                       read_procedure_call(S);
   ⟨assignation⟩                            read_assignation(S).

⟨procedure call⟩ ::= ⟨id⟩ ( ⟨args⟩ )     read_procedure_call(call(Id,Args)) :-
                                            read_token(Id), read_token('('),
                                              read_args(Args), read_token(')').

⟨assignation⟩ ::= ⟨id⟩ = ⟨expr⟩          read_assignation(assign(Id,Expr)) :-
                                            read_token(Id), read_token(=),
                                              read_expr(Expr).
```

Figure 5: PARSING IN MPROLOG

4 The MProlog Compiler

The MProlog compiler is based on Warren's second structure sharing model [11]. This model has been extended with a number of features. Some of these were adapted from elsewhere (e.g. WAM-like representation of empty variables), others were developed by ourselves independently of work proceeding in the field, e.g. the handling of disjunctions, compilation of built-in predicates. Here we shall concentrate on some features which we feel are of interest to the community of implementors. For a more detailed account, see [6].

4.1 Molecules

A basic construction of the structure sharing model is the molecule which is used to represent a dynamic instance of a structure. It consists of two components: a pointer to the skeleton (or static part) of the structure together with a pointer to the global stack frame which contains the variables of the structure. One of the reasons that made DEC-10 Prolog such an efficient implementation is that it could represent these two components in a single machine word; this is not possible on 32 bit machines with large address spaces.

Our solution to this problem was to represent molecules in two machine words; however we do not allocate two words to each variable catering to the contingency that it might contain a structure. Instead when a molecule is needed it is created on the global stack and a pointer to it (a so called *molecule reference* or *molref*) is placed into the variable.

In general it is impossible to know when, and how many molecules are created—this considerably complicates the management of the global stack. We have noticed however, that in many cases (which occur frequently in Prolog programs) it is in fact possible to know about the molecules that will be created during execution and therefore the space for molecules can be allocated during compile time. This is the case for structures appearing in head arguments in an output position and for all structures in call arguments.

4.2 Executable List Structures

One of the main drawbacks of structure sharing is the overhead of accessing arguments of structures and lists. Constants are accessed directly while variables require (an indexed) indirect reference. This may be implemented efficiently only on architectures supporting data driven indirection (such as the DEC-10). On architectures lacking this feature implementation amounts to runtime interpretation of structure arguments.

The MProlog compiler contains a solution which avoids the interpretation overhead for the case of some list structures (those with constant and variable arguments only). This optimization is carried out in two steps.

First we introduce two new registers, *head* and *tail* (`H` and `T`) and a variant of the `uskel` instruction specialized for the case of lists, called `ulist`.

The `ulist` instruction checks whether the corresponding call argument is a list and loads the head and tail of it into the `H` and `T` registers. The contents of these registers are in completely processed form (molecules are constructed if necessary) ready for assignment to a variable or unification with a constant. The latter is performed by `uvar_list` or `ucons_list` respectively. The first advantage of this transformation is that the bulk of the work is done in the `ulist` instruction allowing inline code to be generated for the deep arguments.

The next step follows from the realization that `ulist` must interpret the arguments stored in the list skeleton in order to load the list argument registers. In order to eliminate this interpretation we decided to generate a *substructure access subroutine* as part of the list skeleton following the stored structure. Note that this is significant for output arguments only, where the molecule constructed from such a structure will eventually be "executed" by a later `ulist` instruction.

```
SKEL:     data(functor(./2))
          data(global(1))        ; X
          data(global(2))        : L1
          list_args_start        ; The start of the subroutine
          var_list_arg(H,global(1))
          var_list_arg(T,global(2))
          list_args_end          ; The end of the subroutine
```

Figure 6: THE CODE FOR A LIST STRUCTURE

Fig. 6 shows an example of code involving the head-tail access subroutine, the data structure generated from the third argument of `append(...,[ X | L1 ])`. When the append predicate is executed, a molecule is created which has a pointer to the address SKEL as the skeleton component. When a subsequent `ulist` instruction attempts to decompose such a molecule, it identifies the presence of the head-tail access subroutine and simply invokes it to perform the loading of the list argument registers. In spite of this solution being complicated by the requirement of remaining compatible with the existing interpreter, the resulting code for list processing predicates is comparable in speed with WAM-based copying implementations.

4.3 Elimination of Trailing

The optimization described in this section were motivated by realization that trailing of variable assignments is often superfluous. Similar techniques used today are based on global analysis of the program. We restricted our efforts to a simple local analysis.

Let us consider the case of an output argument of a clause head which is a constant or a structure. The general code for such an argument would perform the following activities:

1. Dereference the actual argument.
2. Check if the dereferenced variable is empty.
3. Check if the empty variable needs trailing (and perform the trailing if necessary).
4. Perform the actual assignment of the constant or the structure to the empty variable.

It is frequently the case that some or all of the activities 1. through 3. can be eliminated. For example if the predicate is called with a very first occurrence of a variable in the given output argument position then steps 1. and 2. are not needed. If in addition it is known at compile time that there are no choice points both within the calling predicate and within the called predicate then step 3. can also be omitted.

We concentrated our efforts on methods to eliminate the need for the trailing check since we found that in most of cases it is rather difficult to deduce at compile time that a non-first occurrence of a variable is still empty. It was thus decided that specialized code will be generated for an output argument of a predicate if:

- no choice point is created upon entry to the predicate,
- all clauses of the predicate contain a constant or a structure in the given argument position
- for all clauses of the predicate the unification code for the arguments preceding the given output argument can not result in any variable outside the predicate being assigned a value.

If the above conditions hold, the code for activities 1. through 3. for the given output argument is factored out of the clauses leaving only the actual assignment (activity 4.) in the clause code. The other activities are performed at the beginning of the predicate and so they can be skipped if the analysis of the context of a call shows that they are superfluous.

In fact this is realized by providing two additional entry points for the predicate. The first entry point will be used if the call context is known to be determinate and the actual argument is a variable. These conditions ensure that if the actual variable is empty then no trailing is needed. Since in most practical cases the variable will be empty, this means that the tasks 1. through 3. will be replaced by a single test for emptiness. The second entry point is used if in addition to these conditions the actual argument is known to be empty (first occurrence of a variable). In this case control is passed directly to the base code of the predicate.

4.4 Compiling Arithmetic and Built-in Procedures

Users typically rely heavily on the standard procedures provided with a Prolog system, therefore it is highly advantageous to generate in-line code for the most frequently used built-ins.

MProlog does provide in-line compilation for arithmetic and other common routines. Of interest is what happens when problems arise during the execution of built-in procedures (e.g. wrong data-types, uninstantiated variables etc.). Since MProlog has a very powerful and general exception-handling system we definitely do *not* want to get into exception handling in the compiled code, that is best safely left up to the interpreter. Therefore the generated code contains both the in-line code for the error-free case as well as the interpretative call to the original built-in procedure.

```
          integer_arithmetic_start(L)
          integer_load(2,I1)
          integer_var_load(Y,I2)
          integer_dyadic(*,I1)
          integer_var_load(Z,I2)
          integer_dyadic(+,I1)
          integer_unify(X,I1)
          goto(D)
    L:    var_arg(A0,"X")
          skel_arg(A1,"2*Y+Z")
          call("is/2")
    D:
```

Figure 7: CODE GENERATED FOR ARITHMETIC

Fig. 7 shows the code produced from the MProlog call `X is 2*Y+Z`. The way this works is the following: if the in-line code (instructions through the `goto`) can be executed without error then we are done. If any problems arise (overflow, a floating-point number, etc.) control is transferred to label L and the call is executed as a normal (interpreted) call with the interpreter handling the subsequently arising exception[2].

4.5 Compilation of match_order

For each node of the control tree the compiler needs to generate code that performs branching to the code of the appropriate subtree. In principle this is done in three steps. First an instruction

switch_on_arg(PAR,L_{var},L_{cons}),

similar to switch_on_term of the WAM, performs the basic separation of cases (jumps to L_{var} if PAR is an empty variable, jumps to L_{cons} if it is a constant, continues with the next instruction if it is a structure). The second step—which is absent in the present implementation—performs a branching based on some hash coding. Finally there are two sequences of compare-and-branch instructions one for constants and one for skeletons (if_cons and if_skel).

For the leaves of the control tree the compiler generates the usual "predicate" code which is either a choice $\Longrightarrow$ try $\Longrightarrow \cdots \Longrightarrow$ no_choice $\Longrightarrow$ trust sequence (if more than one clause remains) or just a trust instruction. The latter being a simple jump instruction can be left out and instead the branching code of the previous node is made to jump directly to the code of the clause.

The code responsible for branching performs some functions of the unification code. For example, the code for unifying an input mode argument with a constant is superfluous, as the check for the identity of the argument has already been done in the branching code (see e.g. the fourth argument of the predicate in Fig. 2). In general, it is the testing part of the unification which is performed by the branching code while the generating part (e.g. variable assignment) is not. This also implies that depending on which of the actual call arguments are instantiated different parts of the unification code for a clause can become superfluous.

One way to avoid doubling of the unification code would be to generate a specialized code for the clause head depending on at which branch of the control tree it occurs. Since this would considerably increase the size of the code we decided to use another approach: to transfer some parts of the generative unification code from the code for clause heads to the branching code, so that the code for clause heads can be the same for all branches.

Constants are the most frequent type of arguments in match_order-ed predicates and also the simplest to handle. We decided to generate no unification code for constants occurring in a match_order-ed position. This means that we have to handle the case of the appropriate call argument being a variable in the branching code. This is achieved by introducing a special assignment instruction assign_call_cons, which is generated before the try or trust instruction on those branches of the tree which include a "variable" branch. Since there may be several such branches included more than one assign_call_cons may be needed.

Structures are much more difficult to handle than constants. Factoring out the code for a molecule assignment would have complex consequences on the unification code. For example, through such an assignment, all variables occurring in the skeleton of the molecule would become accessible to the "outer world", which means they need to be initialized, and also they cannot be considered to have a first occurrence (uvar instruction) in the subsequent unification code.

Because of such problems we decided to consider factoring out structures only for input mode positions, i.e. when the call argument cannot be a variable. Even in this simpler case a

[2]This approach was independently arrived at by the implementors of ALS Prolog, cf. [1].

few points should be noted. The branching code covers only the testing of the main functor, the unification instructions for the arguments of the structure are still needed. Recall that in the unification code a `uskel_in` (`uskeld` in [10]) instruction is responsible for testing the main functor but it sets the structure argument access registers (`Y` and `B` of [10]), too. This means that only the first of such instructions can be factored out in full and for the remaining ones a simplified version can be generated whose only task is to load the structure argument access registers.

We also note that if a variable occurs in a match_ordered position of a clause head it is not factored out for reasons similar to those quoted for structures in non-input arguments.

As an example, Fig. 8 shows the code generated from the slightly simplified version of the program in Fig. 2.

```
match_order flight(-,-,2,1).
mode flight(-,?,?,+).

flight(ba001, daily,  london,  paris).
flight(af002, monday, london,  lyon).
flight(ba071, sunday, glasgow, paris).

   flight:   arrive(4)
             switch_on_arg(local(4),error,Lab1)
             fail
   Lab1:     if_cons('paris',Lab2)
             if_cons('lyon',Lab3)
             fail
   Lab2:     switch_on_arg(local(3),Lab4,Lab5)
             fail
   Lab4:     choice
             assign_call_cons(local(3),'london')
             try(Clause1)
             nochoice
             assign_call_cons(local(3),'glasgow')
             trust(Clause3)
   Lab5:     if_cons('london',Clause1)
             if_cons('glasgow',Clause3)
             fail
   Lab3:     switch_on_arg(local(3),Lab6,Lab7)
             fail
   Lab6:     assign_call_cons(local(3),'london')
             trust(Clause2)
   Lab7:     if_cons('london',Clause2)
             fail
```

Figure 8: Code generated for simple match_ordering

Note also that the code generated for each clause only contains code for the first two arguments:

```
  Clause<i>: ucons_out(local(1), ... )
             ucons(local(2), ... )
             return
```

5 Implementation Technology

The basic components of MProlog are implemented in the systems programming language CDL2 [3], while PDSS and the compiler are implemented in MProlog itself.

CDL2 is an open ended language, i.e. it lacks any primitive operations. These have to be supplied in the target language (most of the time the assembly language of the target computer, or sometimes a higher level language, such as C). CDL2 is thus more like a recursive macro language, with the control structure very similar to that of Prolog without backtracking. The CDL2 Lab, a sophisticated program development environment, has been used as the development tool for MProlog. The Lab, with its powerful inter-module code-generation scheme, helped a lot in efficient porting of MProlog to a wide range of architectures. Several features of MProlog PDSS were also modeled on the CDL2 Lab.

The compiler is written in Prolog. Fig. 9 shows the main phases of compilation.

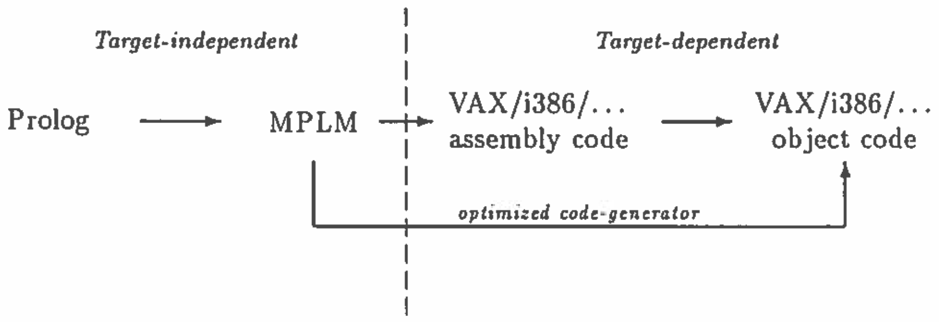

Figure 9: THE STRUCTURE OF THE COMPILER

The machine independent part of the compilation (to the left of the vertical line in the figure) produces a modified variant of Warren's PLM abstract machine code [10] called MPLM. This phase is followed by a machine dependent code generator and an assembler specific for each target environment. The code generator is essentially a collection of macros, giving the symbolic assembler code for each MPLM instruction (see e.g. the left hand side of Fig. 10). Thus the output of the code generation phase for a clause is an assembly program (represented as a Prolog structure). In the final phase this program is assembled into absolute machine code using an assembler also written in Prolog.

The earlier version of the compiler worked exactly as described above. Experience showed that the compiler was spending an inordinately large fraction of its time in the assembly phase. We therefore decided to get rid of the assembly phase. The final solution for this was to keep the assembly description for readability, and use partial evaluation techniques for automatically transforming the description for each MPLM instruction to a version containing direct machine code. The input for partial evaluation process is the abstract code description module plus the assembler; it essentially executes the assembler for each MPLM instruction leaving only those portions for run time which depend on the values of MPLM instruction arguments. In addition to this fairly general procedure, the partial evaluator contains a domain specific routine for collapsing sequences of code generating instructions into a single call containing a longer run of code. The last is supported by a specific external procedure for code emission.

The right hand side of Fig. 10 shows an example of the result of the partial evaluation process.

```
mplm_uvar(PAR,VAR) :-

  generate [
        i(mov,  eax,opr(PAR)),
        i(mov,  opr(VAR),eax),
        i(cmp,  eax,const(max_variable)),
        b(ja,   LAB),
        i(lea,  plmreg(preg1),opr(VAR)),
        i(call, rts(uvar)),
  LAB::   nil
        ].
```

```
mplm_uvar(PAR,VAR):-
  modify_rm_byte(VAR,6,X),
  gen_code([
      "8B",PAR,
      "89",VAR,
      Z: "3DFFFFFF0077",
             byte(Y-Z-1),
      "8D",X,
      Y: "FF1530260100"]).
```

Figure 10: I386 ASSEMBLY AND OBJECT CODE OF AN ABSTRACT INSTRUCTION

6 Conclusion

The MProlog system is one of the most elaborate, longest used, widely ported Prolog implementations, which encapsulates the experience of a range of applications. We tried to summarize those features of MProlog that do not appear in the mainstream Prolog implementations and which we think to be of potential interest to the community of implementors of logic programming languages.

We have shown MProlog programming examples demonstrating the outstanding capabilities of the system in several areas, such as exception handling, parsing based on undoable built-ins and multi-level indexing. We have also described some principal features of the MProlog compiler, illustrating that a structure-sharing model can be improved to support efficient execution of most common simple programming constructs, such as lists.

Acknowledgements

The authors are grateful to all colleagues participating in the development of MProlog during the last fifteen years.

References

[1] *ALS Prolog Beta-Test Documentation.* Applied Logic Systems Inc., 1986.

[2] J. Bendl, P. Köves, P. Szeredi. *The MProlog system.* In *Preprints of the Logic Programming Workshop*, Debrecen, Hungary, July 14-16, 1980.

[3] *CDL2—Language Reference Manual.* Epsilon, Berlin, 1984.

[4] T. Chikayama, H. Sato, and T. Miyazaki. Overview of the Parallel Inference Machine Operating System PIMOS. In *FGCS*, pages 230–251, Tokyo, November 1988. ICOT.

[5] Iván Futó, Ferenc Darvas, and Péter Szeredi. The application of PROLOG to the development of QA and DBM systems. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 347–376. Plenum Press, New York and London, 1978.

[6] Péter Köves and Péter Szeredi. Getting the most out of structure sharing. In *Collection of Papers on Logic Programming*, pages 69–84, SZKI, Budapest, 1988.

[7] *MProlog Language Reference.* SZKI-Logicware, Budapest-Toronto, 1985.

[8] *MProlog Logic Lab Reference.* SZKI-Logicware, Budapest-Toronto, 1985.

[9] Péter Szeredi. PROLOG – a very high level language based on predicate logic. In *Proceedings of the Second Hungarian Computer Science Conference*, pages 853–866, John von Neumann Computer Society, Budapest, 1977.

[10] D. H. D. Warren. *Implementing Prolog—Compiling Predicate Logic Programs.* DAI Research Report 39–40, University of Edinburgh, 1977.

[11] D. H. D. Warren. *An Improved Prolog Implementation which Optimizes Tail Recursion.* In *Preprints of the Logic Programming Workshop*, Debrecen, Hungary, July 14-16, 1980.

[12] D. H. D. Warren. *An Abstract Prolog Instruction Set.* SRI Technical Note 309, SRI International, 1983.

Issues in Implementing ACE: A Stack Copying Based And-Or Parallel System

Enrico Pontelli and Gopal Gupta
{epontell,gupta}@nmsu.edu
Department of Computer Science
New Mexico State University
Las Cruces, NM 88003

Manuel V. Hermenegildo
herme@fi.upm.es
Facultad de Informática
Universidad Politecnica de Madrid
28660-Boadilla del Monte,
Madrid, SPAIN

1 ACE

The ACE (And-Or/Parallel Copying-based Execution) model [GHPS93] uses stack-copying [AK90] and recomputation [GH92] to efficiently support Or- and Independent And-parallel execution of logic programs. ACE can be viewed as a combination of MUSE [AK91] and &-Prolog [HG90]. This paper gives a brief overview of ACE, and then discusses the several implementation issues that arise in combining or-parallelism and independent and-parallelism.

2 Independent And Parallelism

ACE inherits the model of independent and-parallelism adopted in &-Prolog which includes several ways of expressing parallelism. Such parallelism can be automatically uncovered by the &-Prolog parallelizing compiler or annotated by the user. For simplicity, in this discussion we will assume that Conditional Graph Expressions (CGEs) are being used to express and-parallel execution of a set of literals. CGEs are annotations of the form:

$$..., (\text{<conditions>} \Rightarrow \text{literal}_1 \,\&\, ... \,\&\, \text{literal}_n), ...$$

where <conditions> is used to check the independence of the literals in the CGE. Whenever this condition is satisfied the literals belonging to the CGE can be executed in and-parallel.

In implementing ACE, as has been shown earlier [GHPS93], one can adopt the techniques used for implementing &-Prolog without much change. In fact, ACE inherits &-Prolog's execution model, as implemented in the P-WAM abstract machine [HG90,SH93], itself an evolution of the original RAP-WAM [H86]. One of the characteristics of this execution model is that backtracking into a CGE is forced to strictly follow the Prolog semantics. The literals are scanned from right to left in search of a new solution and once this has been found all the literals on the right are recomputed in parallel.

However, we have made some modifications to the P-WAM model that facilitate incorporation of or-parallelism. While in P-WAM the continuation of a CGE is assigned to the worker who finishes the last goal in the conjunction, in ACE we assign such continuation to the worker executing the rightmost subgoal of the CGE.

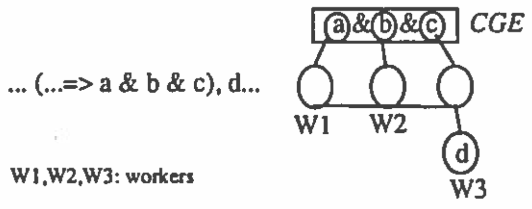

FIGURE 1: Execution of a CGE

This approach leads to the following advantages:

- It always avoids the use of wait-markers on the choice point stack - necessary sometimes in the P-WAM to separate the CGE subgoals from the continuation subgoals of the CGE for a proper backtracking order;
- make backtracking from outside into the CGE always immediate, without requiring any further synchronization activity.

It should be noted that this approach has the disadvantage of sometimes leaving the stack set associated with the rightmost goal blocked waiting for other goals to complete (the agent or process itself can migrate to a different stack set, however).

As in the P-WAM the backtracking process is not controlled by a single worker (the same which started the CGE), but is distributed among the workers executing the CGE. The same worker which fails takes care of killing the rest of the CGE (inside failure) or of propagating the backtracking to the subgoal on its left (outside backtracking). Also inherited from the P-WAM is that the And-scheduler adopted does not put any restriction on the subgoals that can be taken from another worker; this is different from the original proposal of RAP-WAM [H86], in which only subgoals satisfying a specified ordering relation are considered, in order to avoid backtracking on trapped subgoals. Thus in P-WAM and ACE the trapped subgoal problem has been tackled by keeping the physical and logical structure of the choice point stack distinct (i.e. a logical contiguous part of the stack may be actually spread in different parts of the physical stack). This approach may cause holes to appear in the stack. A hole will be reclaimed when everything above it has been reclaimed. Thus, though this approach is more expensive in space, it is quite simple to implement. It has been adopted in other systems (e.g. Aurora [LWH90], where the holes are called ghost nodes).

3 Or Parallelism

The Or-Parallel model that ACE employs is based on the idea of stack copying, developed for the Muse system [AK90]. The different workers involved in the computation operate using an identical logical address space (i.e. each worker allocates its stacks at the same logical addresses); this allows ACE to reduce the job sharing operations (stealing of or-parallel work) to a simple copying of contiguous chunks of memory from one worker to the other (without any form of pointer relocation). In this way multiple environments that are needed for or-parallelism are maintained by keeping separate copies of the execution stacks.

In the stack copying approach, whenever a worker runs out of work, it tries to detect another worker which has unexplored alternatives and copies its complete choice point stack, reactivating the desired alternative by backtracking.

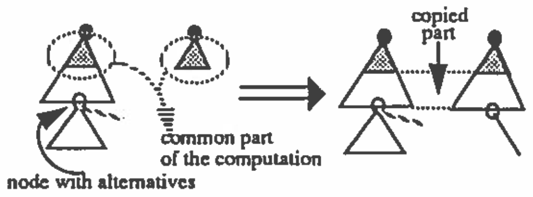

FIGURE 2: Sharing between two teams

A choice point from which different workers may take alternatives is said to be *shared*. For each shared choice point a data structure, called shared frame, is built. This is actually the only real data structure shared among different workers. For the rest each worker has its own private execution environment and, except for the sharing operations, it acts like a standard sequential engine.

4 ACE

4.1 Organization of the Workers

Conflicting requirements are imposed by the two models adopted respectively for or- and and-parallelism, in terms of the visibility of structures: Or-parallelism requires a strict separation among the execution environments of different workers, while and-parallelism requires complete visibility of the execution environment of each worker.

This situation is solved by organizing the workers (individual WAM-like engines) into teams. Each team, which is created by an initial worker (called *master* of the team) contains a certain number of workers which execute a certain branch of the computation tree in and-parallel. The different teams are assigned to execute different branches of the computation tree in or-parallel.

This organization of workers into teams requires an appropriate configuration of the memory, as illustrated in figure 3.

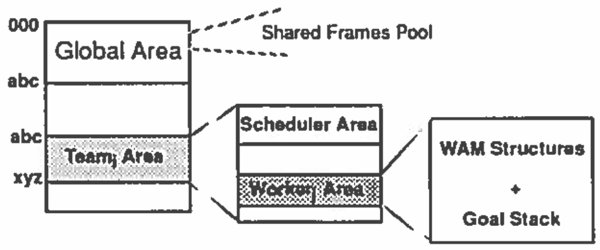

FIGURE 3: Memory organization

It is important to notice that each team sees its own logical address space starting from address 000 and going to address xyz; this approach allows the possibility of performing copying among teams without any address relocation.

4.2 Sharing Assumptions

ACE tries to stay as close as possible to Prolog semantics. Thus, it recomputes independent goals in a CGE, rather than reusing their solutions. Recomputation of independent goals not only simplifies the implementation, it also allows the inclusion of side-effects and extra logical predicates in a natural way [GS92].

This choice of recomputing goals has an influence in the Or-scheduling phase, both for

1. detecting when an alternative is to be made visible for sharing;
2. detecting which parts of the computation tree should be copied during the sharing.

The basic sharing assumption that we adopted can be described by the following rule: an untried alternative may be stolen from a choice point C iff

a. C is not in the scope of any CGE;
   or
b. if C is below the nested CGEs E_1, ..., E_n and s_j indicates the subgoal of the CGE E_j through which the branch leading to C is passing, then
   $\forall i: 1 \le i \le n, \forall j: j \le s_i$, branch($E_i$, j) is completed

The main idea behind this concept is to disallow sharing of an alternative with another team as long as the computation on the left in the tree is not terminated.

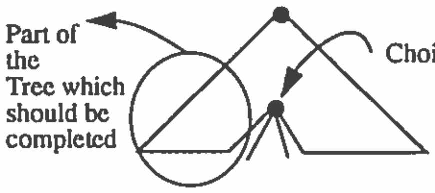

FIGURE 4: Sharing assumptions on a computation tree

Together with simplifying the implementation of side-effects and extra-logical predicates, the sharing assumption

- simplifies identification of stack parts to be copied;
- simplifies the reactivation of execution in the team which is picking up work;
- furthermore, allowing a copy when the subgoals on the left have not terminated yet may lead to situations in which the cost of the sharing is not balanced by a satisfactory level of parallel execution. The following example is perhaps an extreme case but illustrates the idea:

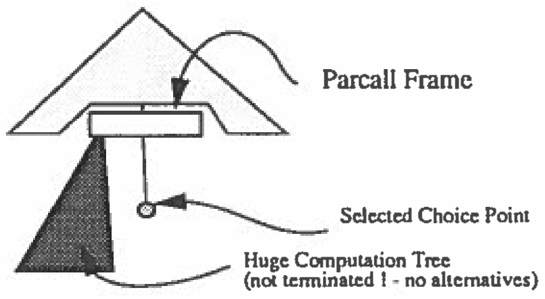

FIGURE 5: Computation tree with wide failing subtree

Sharing of the choice point shown would lead to an expensive copying of the data corresponding to the dark shaded part. If this huge computation fails, the whole computation will fail and the stack-copy operation will turn out to be a useless overhead. Essentially, one should avoid picking work from goals to the right since the more to the right a goal is in a CGE the more speculative it is going to be (unless all goals to its left in the CGE have finished).

4.3 Work Load Detection

Inside a team each worker keeps track of the work that it can make available to others for and-parallel execution. This information can be automatically obtained from the current value of the pointer to the goal stack, in which and-parallel work (pointers to untried subgoals of the CGEs) is recorded.

More complex is keeping track of the global amount of (or-parallel) work available for sharing that is present inside a team.

The available or-parallel work of a team is given by the number of unexplored alternatives located in the private part (i.e. part of the tree which has not been made public or made available for sharing) of the tree computed by the team, counting only the choice points satisfying the sharing assumptions. This is similar to the idea of keeping track of *private richness* in MUSE, where richness indicates the amount of or-parallel work available (in terms of number of alternatives currently untried) at any given moment in the private part of the tree.

In order to support this approach for detecting or-parallel work, some fields have been added to the CGE's descriptor (the PARCALL frame of [H86]). Each subgoal of a CGE has an associated field in which the amount of work currently existing in the subtree rooted at such subgoal is accumulated, and a mark which specifies if the computation on the left has already been completed. Whenever a new choice point is created and it falls in the scope of a CGE, the new alternatives are recorded in the descriptor of the subgoal and, if the mark is set, the richness of the subtree is added to the current richness of the CGE. Whenever a subgoal of a CGE completes execution, we check if any goals to the right have finished, and if so, their richness is added to the richness of the CGE, and so on.

4.4 Memory Copying

In the usual Or-parallel systems based on stack copying, detecting the part of stack to be copied is quite straightforward, since the private part of the computation tree is completely stored between the current top of stack and the top of stack at the time of the previous sharing operation.

When and-parallelism is introduced, the situation becomes more complex, since the private part of the tree is spread over different processors. Furthermore, the fact that no restrictions are imposed on the selection of subgoals at the and-scheduler level may lead to arbitrary intermixing of private and shared parts of the tree on the worker's stacks.

A consequence of this is that since irrelevant parts of the tree may be intermixed with relevant parts (w.r.t. the sharing operation) on a worker's stack, during the copying operation some irrelevant parts may be copied. The irrelevant parts are either holes in the stack, or parts that need to be backtracked over by the copying processor before it can continue execution with an untried alternative.

The sharing operation involves three successive steps:

1. the stealing team (called *sharer* team) detects the best[1] team with available work (called *sharee* team) and figures out the parts of sharee's stacks to be copied. This is done by first detecting the choice point at which the last sharing between the two teams took place.
2. the actual copy operation is performed. The relevant parts detected in step1 are transferred and the copying is performed in parallel between different workers of the two teams. Conditional bindings are removed by backtracking if the first policy was used, or by resetting variables in the copied trail if the second policy was used.

1. the criteria for choosing the best team are a straightforward extension of those used in MUSE.

3. the stealing team reactivates the computation. This is realized by a mechanism that is already in place in P-WAM / RAP-WAM. If the choice point from which we are taking a new alternative is below the scope of a CGE, then as soon as a solution for the current subgoal is determined all the subgoals on the right in the CGE are reactivated in And-parallel.

The main problems occurs in the first stage. Clearly stack-frames that existed in the sharee at the time this last sharing was done also exist in the sharer and need not be copied. The sharer first backtracks to this choicepoint where sharing took place last, and then it find out how much to copy from the sharee's stacks. Here, there are two possibilities: either the sharer can copy everything from the sharee's stacks above the last shared choice-point (including those parts that are after the choicepoint from which an untried alternative will be subsequently picked up by the sharer) or copy only those parts of the stacks that are between the last shared choicepoint and the choicepoint from where the sharer will finally pick work.

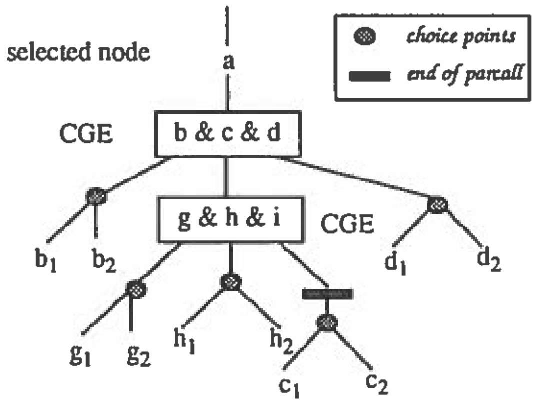

FIGURE 6: Computation tree with CGEs and Or-nodes

Consider the tree generated by ACE shown above. If, during a sharing operation, the node g is assumed as starting point for copying, i.e. it is the last shared choicepoint, then in the first approach stack frames corresponding to h, i, c and d will be copied (even though the solution for these goals may not have been reached yet). The advantages/disadvantages of this approach are the following:

-) it is very simple and does not require a previous traversal of the computation tree (in sharee's stacks);

-) the time for which the two teams need to synchronize will probably be minimum (recall that when a team is sharing work with another, the sharee cannot do any useful execution);

-) a successive traversal of the computation tree should be made by the sharee team in order to detect the choice point (satisfying the sharing assumption) from which an alternative can be taken;

-) the copying is blind and may lead to the transfer of large amounts of information which will be immediately marked as rejected (i.e. will be immediately backtracked over and reclaimed).

In the second approach, the branches on the right of g (h, i, c and d) should be traversed and the position of such computations in the workers' stacks determined. Only those parts of the stacks of the sharee will be copied that correspond to state of the sharee's stacks when the choicepoint up to which we want to do the new sharing operation came into existence. (Thus, in this approach the sharer needs to know in advance the choice point from where it will be picking the alternative to compute the limits of the stacks in the sharee that should be copied.) In this way only the part of the computation tree between the starting point and the chosen choice point is copied. Note that the trail stack of branches h, i, c and d will also have to be copied to remove conditional bindings in the copied parts (In the first approach this implicitly happens during backtracking). The advantages/disadvantages of this approach are perfectly symmetrical to the ones described above.

In the first approach, where the difference between the sharee and sharer's stacks is blindly copied, we have the following question: suppose the branch i is not complete but branch d is; the question is whether also the choice points in d's branch should be made public or not. Because the choice points that will be produced later by the sharee in i will be, of course, private, which will result in arbitrary mixing of public and private parts in the logical structure of the tree (they are already mixed in the physical stacks, as pointed out earlier). ACE does not allow the mixing of public and private parts in the logical tree. Indeed, this is what is enforced by the sharing assumption.

In order to obtain a reasonable trade-off between the two described policies, the following heuristic has been introduced: if the sharee is currently executing below the scope of any CGE, then the second policy is used, otherwise the first policy is adopted. The key observation is that whenever a team is outside the scope of the CGEs, all the branches of the parcalls are completed and the sharing assumption are obviously satisfied. This makes less probable the risk of copying large sections of the computation tree to be subsequently discarded because of non-terminated branches in the middle. This approach constitutes the most natural extension of the incremental copying adopted in the stack copying implementations of Or-parallelism. If however the team is inside a CGE, with unfinished goals in this CGE, then the blind copying of the difference may copy too much irrelevant stuff to be immediately backtracked over.

5 Conclusions

In this paper, we presented ACE, a model capable of exploiting both And-parallelism and Or-parallelism in Prolog in an efficient way. The Or-parallel model that ACE employs is based on the idea of stack-copying developed for MUSE, while the model of independent And-parallelism is based on &-Prolog. We discussed the problems that arise in ACE, along with our solutions to them, in combining the capabilities and memory management of &-Prolog with the stack-copying approach of MUSE.

ACE is currently being implemented on a 20 Processor Sequent Symmetry in collaboration between New Mexico State University and the Technical University of Madrid. Starting from the MUSE implementation, we are adding parts of the &-Prolog machinery and the mechanisms described in this paper, to obtain an implementation for ACE. Since the automatic parallelization task is almost identical in ACE and in &-Prolog, the ACE compiler incorporates large parts of the &-Prolog compiler, which is itself also evolving.

Acknowledgments

We would like to thank Vitor Santos Costa and Kish Shen for many discussions. The ideas presented in this paper are based on earlier papers that introduced the ACE Model [GHPS93]. Research reported here is supported by NSF (Grant CCR9211732), Sandia National Labs (Grant AE-1680), Oak Ridge Associated Universities (Grant CK004448), ESPRIT Project 6707 PARFORCE, and NATO (Grant CRG921318).

References

[AK90] K. Ali and R. Karlsson, "The Muse Or-parallel Prolog Model and its Performance", In *Proceedings of the 1990 North American Conference on Logic Programming*, MIT Press.

[AK91] K. Ali and R. Karlsson, "Full Prolog and Scheduling Or-parallelism in Muse." In *International Journal of Parallel Programming*, 1991, Vol. 19, No. 6, pp. 445-475.

[GH92] G. Gupta and M.V. Hermenegildo, "Recomputation Based And-Or Parallel Execution of Logic Programs." in *FGCS '92*, ICOT.

[GHPS93] G. Gupta, M.V. Hermenegildo, E. Pontelli, and V. Santos Costa, "The ACE Model: And/Or-Parallel Copying-based Execution of full Prolog." Technical Report NMSU-TR-92-CS-13, New Mexico State University.

[GS92] G. Gupta and V. Santos Costa, "And-Or Parallel Execution of full Prolog based on Paged Binding Arrays." In *Proceedings of the 1992 Conference on Parallel Languages and Architectures Europe (PARLE '92)*, Springer Verlag, LNCS 605, June 1992.

[H86] M.V. Hermenegildo, "An Abstract Machine for Restricted And Parallel Execution of Logic Programs." In *1986 International Conference on Logic Programming*, London, LNCS, Springer-Verlag.

[HG90] M.V. Hermenegildo and K.J. Greene, "&-Prolog and its performance: Exploiting Independent And-Parallelism." In *Proceedings of the 1990 International Conference on Logic Programming*, MIT Press, pp. 253-268.

[LWH90] E.Lusk, D.H.D.Warren, S.Haridi et. al, "The Aurora Or-Prolog System", In *New Generation Computing*, Vol. 7, No. 2,3, 1990, pp. 243-273.

[PG93] Pontelli E. and G. Gupta "ACE: Or + Independent And Parallelism." Internal Report, New Mexico State University.

[SH93] Shen K. and Hermenegildo M "Scheduling and Memory Management in And-Parallel Execution of Logic Programs Revisited" Internal Report, Technical University of Madrid.

Extention of Strand with Linda-like operations. Implementation and performance study[1]

G. Czajkowski and K. Zieliński
Institute of Computer Science
University of Mining and Metallurgy
al. Mickiewicza 30, 30 - 059 Kraków, Poland
e-mail: { grzes@tulip, kz@daisy }.ics.agh.edu.pl

Abstract

The aim of this paper is to analyze implementation issues and performance of the STRAND88 system extended with Linda-like communication mechanisms. The extended model consists of the standard STRAND system and an abstraction of the tuple space, that is, virtual distributed associative memory, together with appropriate interface operations, similar in semantics to those offered by the Linda computational model. Linda plays a twofold role in our project — first, it serves as a model of our extension and second, one of Linda implementations provides underlying communication mechanisms. The paper presents an overview of the extended model, then selected implementation issues are shown. Finally, performance results together with a detailed discussion follow.

1 Introduction

Concurrent logic programming languages offer a high-level, elegant, natural and often efficient approach to programming distributed systems [8]. The growing number of such languages, ranging from experimental university research projects to fully fledged commercial products proves their attractiveness. Many applications written in parallel logic languages show that even most intricate problems of distributed computing may be effectively solved using only pure logic programming techniques. However, there is a price to be paid for retaining a clear logical meaning of a program. Managing distributed (possibly ordered) data with (preferably fast) multi-process access and complex interprocess communication protocols are not easy tasks to accomplish in the pure LP. Programs that have to implement features mentioned above are often clumsy, difficult to read and inefficient.

The project we describe adds an abstraction of *tuple space* (modeled after Linda computational paradigm) together with its interface operations to the STRAND88 system. Linda has a twofold role in our project. First, its operations serve as a model for the ones we added to STRAND. Second, a public domain C-Linda version, POSYBL-1.102[2] was used as a basis of our implementation. In our previous paper [2] we have presented a numerous examples of programming in the extended model. The purpose of this work is to discuss some implementation details, together with performance data. The most important factors influencing efficiency of the new system are singled out. Finally perspectives for future implementations, based on more efficient tools, are examined.

[1]This work was sponsored by Komitet Badań Naukowych under grant no.: 8.0077.91.01.

[2]POSYBL-1.102 was written by G. Sxoinas from the University of Crete.

2 Extending existing model

Before moving to main topics of this paper, the STRAND88 language and the idea of Linda are shortly described. The extension of STRAND with Linda—like operations is presented then.

2.1 STRAND abstract computational model

A basic abstraction in a STRAND computational model is a **process pool**, that is, a dynamically changing set of concurrent processes. They communicate via shared variables and data structures. If there are more than one clause matching a process to be reduced — only one of them is chosen, in a nondeterministic way. Variables in processes from a process pool can be assigned values only by an explicit assignment — not by matching and subsequent binding. STRAND is a committed choice language.

2.2 Linda

Linda is not an independent language on its own. It is rather a small set of operations added to the existing language and facilitating the distributed computation execution [1]. As our implementation is based on C-Linda, in this subsection we use function and constant denotational conventions of C. Linda introduces a notion of a **tuple space** (called hereinafter TS), that is, an abstract model of virtual distributed associative memory. The tuple space may be used by all processes in a uniform fashion, regardless of the fact if data to be accessed are stored locally or remotely. The only entities to be found in the tuple space are *tuples*. They may come in two flavors. *Passive* tuples are sequences of typed fields with values. An example can be *("str1", 7)*. *Active* tuples are computations that turn into passive tuples after a completion of their tasks. Because a spawning of new processes is very easy in concurrent logic committed choice languages in general and in STRAND in particular, active tuples and their evaluation mechanism will not be considered. Passive tuples can be manipulated using the following operations, each of them taking a tuple as its argument:

- **void out()** — enters its argument tuple into TS, i.e.: *out("str1", 7)*
- **void in()** — blocks until the tuple matching its argument tuple is found in TS. When a matching tuple is found, it is removed from TS and required argument tuple fields may be assigned a value. For example, *in("str1", ? IntVal)* will take (possibly after an arbitrary long period of waiting) a two field tuple out of TS. The first field of this tuple must be a string *"str1"*, the second — integer, whose value is assigned to *IntVal*.
- **int inp()** — a predicate version of *in()*. If a matching tuple is found a constant *1* is returned. If there is no matching tuple in TS — *inp()* returns immediately with *0*.
- **void rd()** — similar to *in()*, with one important difference: the matching tuple is left without any change in TS.
- **int rdp()** — a predicate version of *rd()*

2.3 The STRAND extended model

The STRAND extended model consists of the standard STRAND computational model and a STRAND tuple space (referred to as STS) communication semantics (Fig. 1). Syntax and

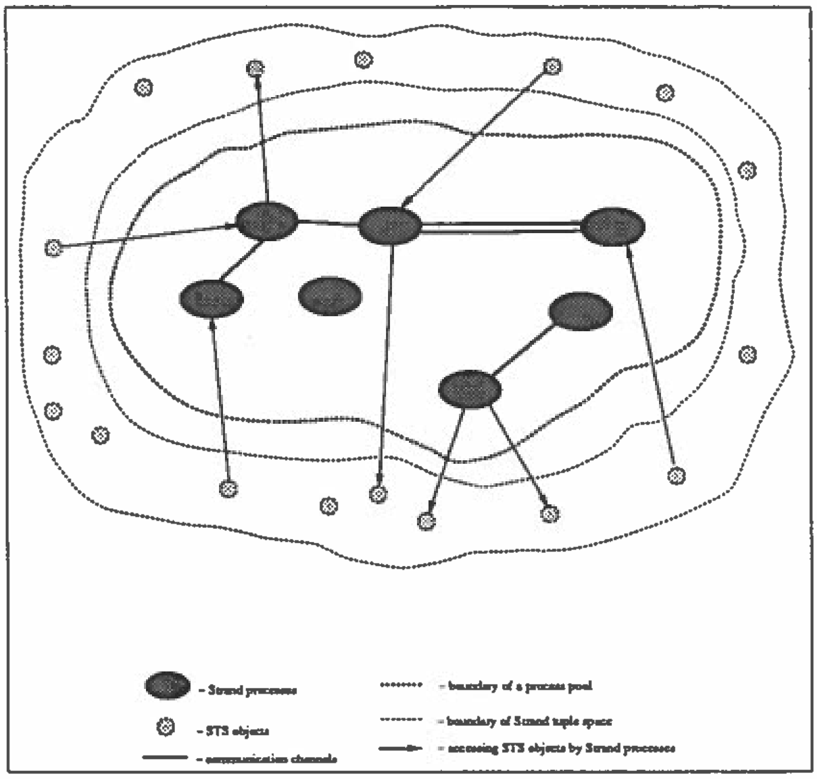
- Strand processes
- STS objects
- communication channels
- boundary of a process pool
- boundary of Strand tuple space
- accessing STS objects by Strand processes

Figure 1: **STRAND processes using STRAND tuple space.**

description of these operations are given in the next subsection. This subsection presents a general view of the extended model.

Semantics of STS operations is very similar to that defined by Linda. To a large extent STS has been designed like a Linda tuple space, described earlier. However, in STRAND it has a different meaning than a tuple space in, for example, *C*. In sequential languages Linda allows computations to be parallelized. As an addition to a parallel logic programming language, Linda operations do not provide anything basically new because processes may transfer data using LP protocols. However, there is a large class of problems that could benefit from using Linda operations.

First, communication protocols may be considerably simplified. Processes using Linda operations become strongly decoupled. Instead of setting up a connection between two processes using a shared variable, these processes may use STS as a shared memory and access it whenever there is such a need. Interprocess communication becomes symmetric, whereas in the LP model sharing a variable forces processes to communicate over it using a *producer—consumer* protocol. In STRAND a connection between processes is *permanent* while our model offers *temporary* connections, that is, a process interfaces with STS for the duration of the call' only. To use an analogy from computer networks, STRAND communication resembles circuits while our model offers more a flexible connectionless protocol.

Second, processes (so called *blackboards*, described later on) managing data that may be accessed by other processes are almost unnecessary. STS may be treated as a blackboard and its interface operations as queries to a blackboard. Besides simplifying a source code (there is *no* definition of a blackboard) a programmer does not have to set up any interprocess communication.

Third, an additional degree of nondeterminism is introduced because order of searching STS by added operations is not determined. Writing a process that has to pass two integers to two other processes (each process gets one integer) is very simple in STRAND. However, if passing integers must be nondeterministc, such a program becomes more complex. The extended model provides equally simple techniques for both deterministic and nondeterministic behaviour of programs.

2.4 Operations of the extended model

In the previous subsection a term STS (STRAND tuple space) has been introduced. Such a name was chosen for the sake of uniformity with Linda terminology. However, in LP languages tuples are important but not only data structures. For that reason data found in STS will be called *objects.*

In order to assess usefulness of different operations, we have decided to implement the following five:

- *st_out(Data, Status)* — enters *Data* into STS. *Data* may be any STRAND data — list, tuple (possibly nested) or basic type data. If *Data* contains variables, *st_out/2* blocks until they are assigned values. After the operation has been completed, Status is assigned an empty list. For users convenience *st_out/1*, without the synchronizing output argument, *Status*, was also implemented.
- *st_inp(Data, Status)* — operation similar to *inp()* in Linda. *Status* indicates if the required tuple was found, that is, it is assigned a constant *1* if *STSData*, matching *Data*, was found in STS and removed from it and *0* otherwise. If *Data* contains variables they will be assigned appropriate values. For example, after *st_out({ 1, { 2 }, 3}, _)* has been completed, *st_inp({ 1, A, B }, Status)* will succeed, that is *A* will become { *2* }, *B* will be assigned a value *3* and *Status* equal to *1* will indicate success.
- *st_rdp(Data, Status)* — same as *st_inp/2*, but the matching object (if there is such) is not removed from STS.
- *st_in(Data)* — this process will not be reduced until an appropriate object is found in STS. Similarity to *in()*, a matching object will be removed from STS.
- *st_rd(Data)* — this process will block until an appropriate object is found and read (but not removed) from STS.

It should be emphasized that after having put an object to STS, in order to be read it needs no longer to be an argument (or part of an argument) to any process. Such an object may be accessed by any STRAND process using only extended model operations. If an object has been *out*ed on a given processor, it is not moved from it by internal STS managing processes. Any object from STS is accessible from all STRAND machines, and all calls are *transparent* — there is no difference in syntax and semantics between accessing local and remote STS objects. STS managing processes ensure that STS object matching an argument to *st_in/1* or *st_inp/2* is atomically locked before being removed from STS. A consequence is that after *st_out(1, _)* only one of the two simultaneous calls to *st_inp(1, Done)* will succeed.

3 Implementation issues

There were two main problems to be solved in implementing our project:

- Providing distributed STRAND tuple space maintenance.
- Representation of STRAND data structures in C.

This section gives some insight into how we have coped with the above mentioned.

3.1 Maintaining thge STRAND tuple space

The most important decision in our project was how to implement STS. As our major objective was to assess usefulness of extending STRAND model and we wanted to concentrate on programming techniques in the extended model we decided to reuse existing code and base our STS implementation on the POSYBL-1.102 compiler.

This system is not as fast as commercial Linda implementations (like SCA Linda, for example) but it has three very important advantages:

- POSYBL uses the standard C compiler (*cc*), which allows the POSYBL code to be incorporated into the STRAND Abstract Machine.
- POSYBL daemons run in background even if no Linda program is being executed. Due to that fact, POSYBL may be started before STRAND. STS operations will have a direct access to Linda tuple space (TS) during the whole time of computations. In contrast, the SCA Linda system runs only during the computation of some Linda program. The fact that POSYBL runs all the time may be utilized in one more way. Instead of reading the data from a disk they can be kept in STS from one STRAND session to another. Our implementation was also successfully used to provide communication between two users, each of them using STRAND.
- POSYBL system is supplied with its C sources. We have taken advantage of this fact and changed slightly definitions of some operations to make them better suit our requirements.

The last point requires some explanations. There are only three out of five Linda operations acting on passive tuples implemented in POSYBL, namely *out()*, *in()* and *rd()*. We have changed the code of the last two and turned them into their predicate counterparts, *inp()* and *rdp()*. In that way for example *st_inp/2* may be implemented directly, using *inp()*.

We would like to stress that there is no connection between the STRAND process pool and the STRAND tuple space. Processes from the pool may access STS but STS itself is managed by processes from outside the pool.

Using the existing Linda system code was only the first step towards a successful implementation. A second, more interesting for us, was how to represent the STRAND data in the Linda tuple space. The next subsection covers this topic.

3.2 Data representation

Before describing a translation between STS objects and Linda tuples we must describe what kinds of data may be stored in POSYBL TS.

POSYBL tuples can be composed of the basic type data (e.g. *int*, *char*) and vectors of these types. A value of every tuple field must be contained inside a type indicating function, e.g. *(lint(7), lstring(char_ptr))*, *(lint(int_val), lnint(int_ptr, vect_len))*. If the type indicating function starts with a letter '*q*', it denotes data which value will be taken from TS (we will call such fields *q-fields*), i.e. *in(lint(7), qlint(&int_val), qlstring(&str_ptr))* will remove the first matching tuple from TS and assign appropriate values to *int_val* and *str_ptr*. No tuple may have a q-field as its first field, that is, the first field of every tuple must have a value. POSYBL uses the type and value of this field for hashing data in TS. As will be shown later, this information was very important for our implementation.

A foreign procedure may access the STRAND data via STRAND system calls. These calls provide information about a type and value (if any) of the STRAND data. A user can also build the STRAND data inside foreign procedure and then pass them to STRAND. We will discuss detailed implementation issues on two examples - *st_out/2* and *st_inp/2*.

3.2.1 Implementing *st_out/2*

Objects of basic types (e.g. integers, strings) may be transformed into POSYBL tuples directly. They get an additional tag field, indicating their type and the fact that they are the first level objects (that means, they actually were arguments to *st_out/2* and not components of some argument). Thus *st_out(1, _)* will call POSYBL in *out(lint(INT), lint(1))*, where *INT* is an integer constant. Such policy causes the first level STS objects of the same basic type to be stored in TS with the same hash key. That does not degrade a performance because almost all STRAND programs pass tuples between processes and not integers or strings. When strings are passed (i.e. '*done*' messages indicating a work completion in farmer-worker applications) they are usually identical strings and searching of a hash table for them is very fast. Similarly, it is rather unusual to pass big numbers of different integers or reals between processes. Objects of these types are usually encapsulated within a tuple.

A tuple must be linearized, that is, its tree-like structure must be transformed into a set of Linda tuples. This transformation must allow for a subsequent tuple restoring. The first level STS tuple is remembered in TS as a five field POSYBL tuple: *(lint(TUPLE), lnint(id_table, arity * 2), lint(arity), lstring(field1), lstring(field2))*. A vector of integers, *id_table*, contains a sequence of pairs: *(FieldId, FieldType)*. The meaning of *FieldType* is obvious. *FieldId* is network-wide unique number, associated with a particular field. This facilitates fast tuple fields searching, for there is only one unique hash value for that field, namely *FieldId* (if *FieldId* is equal to zero, then the value of this particular STRAND data is stored directly in the first-level Linda tuple — this is used for optimization, as shown below). A process providing unique numbers is written in POSYBL.

The last three fields are used for optimization (we show only one possible approach to optimization here — one of the later subsections shows other optimization methods). If the first argument is a string, the fourth field gets its value. Similarly, if the second field is an integer its value is stored in the fifth field. A description of *st_inp/2* will show how they are used for efficient TS searching.

The first-level STS list is represented as a four-field tuple: *(lint(LIST), lint(head_id), lint(head_type), int(tail_id))*. The second and fourth fields contain unique identifiers of the head and tail of the list, respectively. The third field is used for optimization. First level empty list, which is represented by STRAND as a tuple of arity *0*, in POSYBL becomes *(lint(ELIST), lint(0), lint(0), lint(0))*.

So far we have discussed the first level objects only. Lower level objects (fields of tuples or list components) will be scanned like the first level objects, but with two differences. Instead of

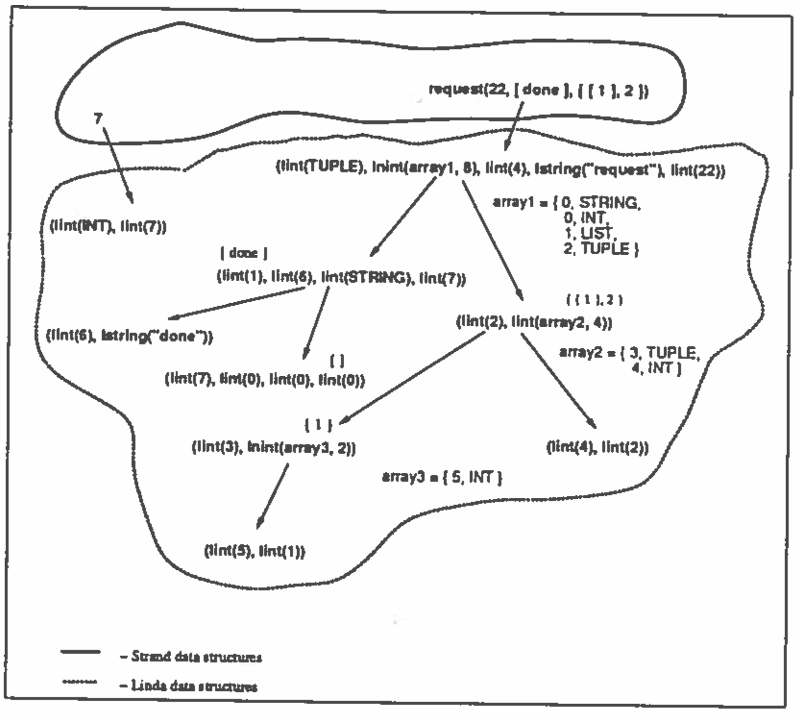
request(22, [done], { { 1 }, 2 })
7
(lint(TUPLE), lnint(array1, 8), lint(4), lstring("request"), lint(22))
array1 = { 0, STRING, 0, INT, 1, LIST, 2, TUPLE }
(lint(INT), lint(7))
[done]
(lint(1), lint(6), lint(STRING), lint(7))
{ { 1 }, 2 }
(lint(2), lint(array2, 4))
(lint(6), lstring("done"))
array2 = { 3, TUPLE, 4, INT }
[ ]
(lint(7), lint(0), lint(0), lint(0))
{ 1 }
(lint(3), lnint(array3, 2))
(lint(4), lint(2))
array3 = { 5, INT }
(lint(5), lint(1))
– Strand data structures
– Linda data structures

Figure 2: **Destructuring Strand data.**

having a predefined integer constant describing a data type as the first field, lower level objects will have unique identifiers there. It will prevent an unauthorized access, binding the field with the structured object. The second difference is that lower level tuples do not carry arity and values of their first two fields with them. It is a reasonable approach, since including this information in a Linda tuple describing a nested STRAND object would not speed up searching. A unique value of the first field of Linda tuple suffices. Fig. 2 shows how STS objects are represented in TS. Besides, if an object is a string composing the first field of the first level tuple or an integer composing its second field, it is not stored in Linda tuple space. Information about its value is present in Linda tuple describing a first level STRAND object.

The definition of *st_out/2* is rather simple and will not be given here. A structured object is scanned down recursively and if data is a variable, the procedure uses STRAND supplied *WaitB()* in order to wait for its binding. Only after all structure components are stored in STS, a Linda tuple giving the information about the first level of the object is *out*ed.

3.2.2 Implementing st_inp/2

An object in STS may be accessed by processes from a STRAND process pool. If some process calls *st_inp(Y, Status)*, STS must be searched. This search terminates when either some STS object, *X*, matches *Y* or there is no such an object *X*. The searching of STS should be as efficient as possible.

The searching of STS for the first level objects of basic types is rather simple. A user may be interested in finding out whether a given object is in the tuple space, i.e. *inp(7, Status)*. The

st_inp() - this procedure is visible by the STRAND user as a foreign kernel

```
void st_inp(Object, Status)
    switch (get_type(Object))
        case VAR: get any object from STS ; break;
        case INT: * Status = inp(lint(INT), lint(Integer(Object))); break;
        case STRING: * Status = inp(lint(STRING), lstring(String(Object))); break;
        case TUPLE: * Status = find_tuple(Object); break;
        case ELIST: * Status = inp(lint(ELIST), lint(0), lint(0), lint(0)); break;
        case LIST: * Status = find_list(Object); break;
```

find_tuple() - searches TS for the tuple matching its argument

```
int find_tuple(Tuple)
    ok1 = ok2 = 0;
    if ((get_type(arg = GetArg(Tuple, 1))) == STRING)
        ok1 = 1;
        str_val = String(arg);
    if (arity > 1)
        if ((get_type(arg = GetArg(Tuple, 2))) == INT)
            ok2 = 1; int_val = Integer(arg);
    if (ok1)
        if (ok2)
            if (!inp(lint(TUPLE), qlnint(&id_tab, &id_tab_len), lint(arity), lstring(str_val), lint(int_val)))
                return 0;
        else
            if (!inp(lint(TUPLE), qlnint(&id_tab, &id_tab_len), lint(arity), lstring(str_val), qlint(&int_val)))
                return 0;
    else
        if (ok2)
            if (!inp(lint(TUPLE), qlnint(&id_tab, &id_tab_len), lint(arity), qlstring(str_val), lint(int_val)))
                return 0;
        else
            if (!inp(lint(TUPLE), qlnint(&id_tab, &id_tab), lint(arity), qlstring(&str_val), qlint(&int_val)))
                return 0;
    NewTuple = TupleRestore(id_tab, arity, ok1, str_val, ok2, int_val);
    if ((flag = match_objects(NewTuple, Tuple, &BindList)) == 1)
        bind_vars(BindList);
    else
        flag = find_tuple(Tuple);
        st_out(NewTuple, &int_val); returning NewTuple to TS
    return flag;
```

Figure 3: Definition of *st_in()* given in the pseudocode.

Linda code calls *inp(lint(INT), lint(7))* and searching TS is done by TS managing processes.

If the user calls *st_inp(Var, Status)* and *Var* is an unbound variable (every STS object matches variable), the Linda code searches STS in the following order: tuples, lists, integers, strings, reals. Checking if there are objects of one of these types in STS is done by TS managing processes but looping over types is the task of added code.

According to our statistics[3] the most frequently exchanged information between STRAND processes are tuples, usually not nested or with a small nesting level. STS first level tuples are represented with three additional fields - arity, the value of first field (if it is a string) and value of second field (if it is an integer). Using these three fields speeds the searching up. Adding arity to the Linda tuple representing the first level STRAND tuple limits searching to STS tuples of the same arity. Without this limitation, *inp()* would carry on the search in the space of all tuples, irrespective of their arity, and in the worst case return all tuples.

Two other fields may limit the set of STS matching tuples even further. If the first two fields have appropriate types, their values are stored in the Linda tuple. Fig. 3 shows an implementation of *st_inp()* with searching optimizations mentioned above. The function *find_tuple()* scans TS for the first tuple matching the given one, *T*. If a tuple *S* was extracted and *S* does not match *T*, *S* is not entered into TS until the exhaustive search is over. The rationale behind such policy is that *S* will not be (unnecessary) examined once again by one of the recursive calls to *find_tuple()* and the searching of TS will be faster. If *S* were to be returned to TS *before* the recursive call in *find_tuple()*, in the worst case the algorithm would never terminate (in practice, stack overflow would stop computations) . *S* would be taken from TS, examined, put back, taken from TS ... over and over again, because the TS searching is nondeterministic.

In Fig. 3 we have omitted, for readability's sake, declarations of variables and definitions of some functions. One point that requires explanation is that *TupleRestore()* and *match()* can actually be performed as one function. In such a case, if a tuple *S* extracted from TS does not match an argument to *find_tuple()*, all Linda tuples defining *S* may be left in TS with the exception of the one describing the layout of the top level of *S*. In the simplified example in Fig. 3 such optimization is not performed.

4 Performance study

As has already been shown [2], for a wide class of problems the extended model offers smaller, more compact and readable solutions. This alone makes the idea of using Linda-like operations in STRAND interesting and validates our effort. This section gives comparison of performance between certain problems written in pure and extended STRAND.

4.1 Choice of benchmark problems

In previous sections we have listed reasons why the POSYBL Linda implementation was chosen as a basis of our project. POSYBL has some features, not offered by other available Linda implementations, suited very well for our needs. Due to them we could implement all presented operations and develop and test new programming techniques in the extended model. However, being a public domain product, POSYBL is not very efficient. In particular, it does not offer

[3]A number of programs has been examined to obtain some statistical insight into the kind of data exchanged between STRAND processes. The programs varied from symbolic computations and neural networks simulations to distributed computations of fractals and ray-tracing images. We have also used some programs written by the students.

very high transmission speeds. It sends data between workstations slightly faster than STRAND but is still much slower than some of the commercial products[4]. On the average, transmitting data between workstations using POSYBL is about 7 per cent faster than using STRAND, as shown in Fig. 4. This number is the limit of any possible speed gains in programs sending large amounts of data between workstations. Thus, moving the responsibility for data transport between workstations from the STRAND kernel to Linda internal tuple space managing processes does not result in considerable efficiency gains.

packet size in bytes	*STRAND*	*POSYBL*	*SCA Linda*
16	30.04	27.56	2.71
64	30.01	27.96	3.46
256	30.06	27.90	4.17
1024	30.03	28.01	4.68

Figure 4: **Data transmission times (im milliseconds) between two SPARC-2 worksations in POSYBL, SCA Linda and STRAND systems.**

The fact that STRAND and POSYBL transmit data at roughly the same speed implies that the extended model will not be able to provide faster transmission between different machines. As a consequence, comparing execution times of programs that run on the same machine or are distributed over the network will give us the same information about efficiency of extended model operations versus standard STRAND techniques. To put it more concisely, we propose that the equation $t_c(S_n) \approx t_c(E_n)$ is true for all $n > 0$, where *S* and *E* denote programs written in pure and extended STRAND, respectively, *n* is the number of processors the computation runs on and t_c returns the time taken to transmit data between workstations during the program execution. Because of that fact we will analyze programs executed on one workstation. Performance of the extended model STS interface operations in programs executed on one workstation is influenced by two separate factors: time taken to decompose or rebuild STRAND data structure and time taken to execute appropriate Linda operations. This will be analysed in detail in one of the next sections.

There is no reason in comparison of the performance of *the same* STRAND program in two different models, since such comparison would give two identical results (because, when not used, the mere existence of the extended model operations does not influence th execution times of STRAND programs) and no information about added Linda-like operations. The conclusion is, that the performance of two completely different programs must be measured, because we are comparing different computational models, not different compilers and the quality of compiled code. The programs will be functionally equivalent but one of them will be written in standard STRAND while the other will use the extended model operations. It would be also desirable to use only added STS interface operations to communicate processes in programs written in the extended model. Programs using a mixture of interprocess communication techniques, some from pure and some from extended STRAND, would not give us accurate information about performance of added operations. In our view the most representative examples are managing large sets of data and sorting lists. Following subsections analyze performance of programs

[4]The authors have carried out extensive tests, comparing data transmission speed of numerous LAN distributed computing systems. Both STRAND and POSYBL are one order of magnitude slower than the fastest systems, PVM, P4 and SCA Linda v.2.4.6 [10].

dealing with problems mentioned above. Before giving performance results, we must describe shortly how the three systems used in the project measure time.

4.2 Measuring time

Three systems, STRAND88, POSYBL and SCA Linda were used in performance measurement experiments. Each of these systems has its own timer utilities.

The total execution time of a process in STRAND may be measured using *time_call({ Module:ProcessName , Time }, Status)* system call. The accuracy is one millisecond. This call returns time taken to execute *Module:Call* .

POSYBL and SCA Linda use similar built-in timer utilities. The fragment of the code which execution time is measured is bracketed by *start_timer()* and *timer_split()* system calls. In both systems the accuracy is one microsecond.

The values of all measurements throughout this program are given in milliseconds.

4.3 Managing large sets of data

Benchmark programs will have to extract and add items of data to the set of data. In the extended model, the set will be represented by STS. The user cannot influence the order in which STS is being searched. Providing data with additional tags (i.e { *input, 0* } and { *output, 7* } instead of *0* and *7*, respectively) may constrain the set of possibly matching STS objects and, in consequence, speed up a searching process, but the user cannot modify the basic STS searching algorithm.

In standard STRAND a user must program the searching procedure himself (it is usually called a *blackboard* [3]) , choosing the most appropriate data structures and algorithms. We have measured execution times of programs that add and remove data from an ordered list, an ordered but unbalanced binary tree and an AVL tree. During the whole experiment each of those structures contained about 5^6 (15625) entries. These entries (tuples) were also entered into STS. The second field of each entry consisted of an unique integer key. The first field was an empty list or a string. With the optimization mentioned in the former section, { *[], IntVal* } would be represented in TS as two Linda tuples, while { *StringVal, IntVal* } would be represented as one Linda tuple. Thus, locating the matching tuple (if there exists one) would take the same time in both cases, while extracting the tuple from STS would require two *inp()* calls in the first case and one such call in the second case.

Computational model	*Extended model*		*STRAND*		
Data structure	STS first field is an empty list	STS first field is a string	ordered list	unbalanced binary trees	AVL trees
Time of extracting an item	12	9	174.4	14.9	8.3
time of adding an item	81	77	153.5	78.0	30.4

Figure 5: **Average times of extracting and adding data to STS and managing different data structures in STRAND.**

Fig. 5 gives averaged times of adding and removing a data item from STS and from an appropriate STRAND data structure maintained by a blackboard process. 10000 elements to be entered and extracted were chosen at random. The program written in the extended model is a viable solution for managing distributed sets of data. AVL trees are faster but the cost is a considerable complexity of the program [5].

4.4 Sorting lists

In this section we would like to compare performance of two programs that sort lists. In STRAND we use quicksort. In the extended model we use a program that hardly has a counterpart in logic programming (Fig. 6) [2]. It takes advantage of the most basic idea behind the Linda paradigm — memory (in our case — STS) is not addressed by some index value, but by its contents. Thus, if we know the value of some data, it may be *read*, instead of *searched*, from such virtual associative memory. These numbers to be sorted are initially in STS and each of them is not smaller than some lower bound and not greater than some upper bound. The sorting program takes this numbers, in the ascending order, out of the STS. If the difference between bounds is m and the number of integers in STS is n, than the number of calls to *st_inp* grows linearly with n and m, and is equal exactly to $m + n$, giving linear complexity.

LowerLimit and _UpperLimit_ must be integer constants

```
sort(Sorted) :- sort(LowerLimit, UpperLimit, [], Sorted).
sort(N, Limit, List, Sorted) :- N > Limit | Sorted := List.
sort(N, Limit, List, Sorted) :- otherwise —
    st_inp(N, Found), decide(N, Found, N1, List, List1), sort(N1, Limit, List1, Sorted).
decide(N, 1, N1, L, L1) :- N1 := N, L1 := [ N | L ].
decide(N, 0, N1, L, L1) :- N1 is N + 1, L1 := L.
```

Figure 6: **Sorting objects in STS - problem's complexity is proportional to the number of elements in STS.**

Of course, the algorithm is not perfect (cannot sort real numbers, for example, and it is not always possible to estimate lower and upper bounds) but gives the general idea of how STS and its interface operations may be used.

Fig. 7 presents results of sorting randomly chosen integers smaller than 1001 and greater than 0. Performance of the extended model program is worse for lists shorter than about 25000 . However, the time of computation grows linearly with the number of sorted integers so for longer lists this algorithm sorts faster than quicksort.

4.5 Factors influencing extended model performance

There are two important parameters which must be taken into account while evaluating extended model operations. First of them is the nesting level of STS object. Fig. 8 contains averaged times of executing *st_out(Tuple, _)* and *st_inp(X, _)*. *Tuple* had one field, { *tuple*, { .. { *1* } .. } } , where the experiment parameter was the nesting level of this field, that is, the number

[5]The STRAND code for managing the mentioned data structures, used in measurements, is available from the authors by e-mail.

Length of a sorted list	*STRAND — quicksort*	*Extended model — program from Fig. 6*	*Ratio of the times*
1000	1244	5840	0.21
2000	2069	8760	0.24
5000	9687	17403	0.56
10000	17461	31593	0.55
15000	33903	45441	0.74
20000	49857	58973	0.84
25000	79218	72807	1.08
30000	107280	85764	1.25
35000	134654	90150	1.49
40000	157431	114826	1.36
45000	232596	127734	1.82
50000	298451	139539	2.13

Figure 7: **Average time of list sorting in standard and extended STRAND.**

of parentheses pairs bracketing integer constant *1*. *X*, the argument to *st_inp/2*, is an unbound variable. There are no other objects in STS at the time *st_inp/2* is called, so no time is spent on searching STS.

The dependency is almost linear. The time of executing *st_out/2* grows faster than that of *st_inp*. The reason is that entering the structured object into STS requires calling the process providing the networkwide unique identifiers.

Two separate factors contribute to the overall time of entering and extracting a tuple out of STS — calling POSYBL system and destructuring or rebuilding STRAND data. The column labeled *analyzing - 1* in Fig. 8 gives the time of preparing the tuple for being entered into STS (instead of *out()*, the dummy C procedure was called, which returned immediately). The column *analyzing - 2* shows the time C code needs to scan a STRAND tuple, that is to descend recursively through all nesting levels to a constant *1*. After comparing *analyzing - 1* and *analyzing - 2*, we can see that scanning STRAND data is very fast. A considerable amount of time is consumed by process providing unique identifiers, and that contributes mainly to much bigger numbers in *analyzing - 1* than in *analyzing - 2*. Measured times of building STRAND tuples are presented in the column labeled *creating*.

The second parameter influencing the speed of extended model operations is the size of the argument. As the measurements have shown, a change in the length of a string (from 16 bytes up to 1 Kbyte) that is taken and entered into STS does not practically influence the performance.

5 Optimization methods

In the previous section we have mentioned that STS interface operations are optimized for tuples that have a string as the first field and an integer value as the second field (provided that the arity is at least *2*). These fields are stored in the Linda tuple describing the top-level layout of the STRAND tuple *T*, instead of forming Linda tuples associated with the particular field of *T*. The choice of such fields and types was based on statistics — most tuples have a string as their

nesting level	*st_out*	*st_in*	analysing — 1	*analysing — 2*	*creating*
0	12	18	2	0.14	0.27
1	15	20	2	0.24	0.28
2	18	21	2	0.24	0.29
3	22	23	2	0.25	0.31
4	25	25	2	0.25	0.32
5	27	27	2	0.25	0.32
6	30	29	2	0.25	0.34
7	33	30	2	0.25	0.36
8	37	32	2	0.26	0.38
9	40	35	2	0.26	0.40
10	45	37	3	0.26	0.41
15	63	48	3	0.28	0.45
20	97	63	3	0.29	0.53
25	105	68	3	0.31	0.59
50	195	128	4	0.37	0.84
75	301	196	6	0.41	1.13
100	489	270	8	0.47	1.26
200	993	679	15	0.69	2.42
500	2493	1571	34	1.43	5.88

Figure 8: **Time of executing *st_out* and *st_inp* as a function of nesting level of STS object.**

first field and an integer as the second one. However, for some programs different optimizations might be more suitable. For example, programs which pass lists of integers between processes would run much faster if the list-oriented optimization policy were employed. Instead of breaking a list containing N integers into $2N + 1$ Linda tuples, it might be stored as a single Linda tuple, containing a vector of appropriate integers. Fig. 9 gives the comparison of performance between usual and optimized methods of representing lists in the Linda tuple space. Optimized STS interface operations are much faster. Because of the fact that *out()* and *inp()* are called only once in the system optimized for lists regardless of the list length, the execution time of an operation grows very slowly with the length of a list.

Unfortunately, POSYBL implementation makes it impossible to transform effectively *any* STRAND data structure into a vector of integers, containing coded field types and their values. Such approach is feasible, but completely inefficient. Such *one STRAND object — one Linda tuple* mapping would cause STRAND objects to be stored as one big set in TS, without grouping them according to their types. This would introduce great overhead. For example, in order to check if a five-field STRAND tuple exists in STS the searching algorithm would have to take every Linda tuple in turn out of STS and examine it, until all have been checked or until the matching one has been found. In our approach, only the set (possibly empty) of five-field STS tuples is scanned. If there is no five-field STRAND tuple in STS and N other objects, the list-optimized algorithm must take all of them out of STS, decode them and put back into STS, which results in $2N$ calls to underlying Linda tuple space. In contrast, our approach would require only one call to Linda tuple space to make sure no matching tuple is available. In short,

length	*st_out*		*st_in*	
	standard	*optimized*	*standard*	*optimized*
0	4	4	4	4
1	12	4	13	4
2	19	4	19	4
3	25	4	25	4
4	32	4	31	4
5	39	4	37	4
6	45	4	43	4
7	52	4	49	4
8	58	4	56	4
9	65	4	62	4
10	72	4	68	4
15	105	4	99	5
20	138	4	129	5
25	171	4	160	5
50	336	5	313	6
75	502	5	467	7
100	667	6	620	8
200	1329	6	1234	9
500	3313	8	3076	10

Figure 9: **Time of executing *st_out* and *st_inp* as a function of list's length.**

the best results would be obtained with optimizing only for a single data type. More general optimizations could lead to a degraded performance for some classes of problems.

Our system can be started with one of the three options (they exclude each other) — optimizing for lists of integers, lists of strings and tuples. If tuple-oriented optimization is chosen, a list of first-level fields with types must be specified in a command line of the system, e.g. *stl -tuple 1i 2s 3r* would optimize for tuples with a first field being integer and/or a second one — string and/or a third one — real.

6 Perspectives

Our project was based on a rather inefficient Linda implementation. Availability of commercial Linda systems producing code that could be incorporated into the STRAND abstract machine could result in much better performance of the extended model. The cost of replacing POSYBL with some other compiler would then be very low because all the work has already been done.

The tables below present some performance parameters of SCA Linda.

- Transmission of data between workstations is much faster, as shown in Fig. 4.
- Time of execution of *out(1)* and *in(1)* (tuple space was empty before putting an integer into it and contained only one number, *1*, before extracting it), as compared in Fig. 10, is much better in SCA Linda than in POSYBL.

	POSYBL	*SCA Linda*
out	2.04	0.20
in	3.12	0.19

Figure 10: **Averaged time of executing *out(1)* and *inp(1)* in two Linda implementations.**

In previous sections we have listed reasons why SCA Linda could not be used as a basis of our implementation and why POSYBL was chosen. However, there are no reasons why fast, commercially available Linda systems cannot be implemented with the speed similar to that of SCA Linda and open run-time system similar to those of POSYBL. With such a system, provided that its C code could be incorporated into STRAND Abstract Machine, our implementation would be much faster (about one order of magnitude) than it is now. What is also important, all the work has already been done. Destructuring tuples, rebuilding them and mapping between STS and TS objects has already been implemented. With new Linda all that would have to be done to have new, extended STRAND up and running would be to examine all the idiosyncrasies of the new Linda and change the form of calls to *in(), inp(), rd(), rdp() and out().*

7 Conclusions

The STRAND execution model may be extended with a set of Linda-like operations and an abstraction of tuple space. This STRAND tuple space is orthogonal to the STRAND process pool. Processes may access STS objects without setting up a shared data structure connection first. Connection is established only for the time of the call duration. The presented extension of STRAND has been implemented and the performance of presented techniques has been evaluated. This study proved that the proposed approach may be implemented and effectively used. Important factors influencing the efficiency of the extended model have been examined. The extended model may in future have much better performance than it has now if the C code of commercial Linda implementations can be incorporated into the STRAND Abstract Machine.

References

[1] N. Carriero and D. Gelernter. *How to write parallel programs: A guide to the perplexed.* ACM Computer Surveys, Sept 1989.

[2] G. Czajkowski and K. Zieliński K. *Extending the Parallel Logic Programming Paradigm with Linda-like Operations.* Internal Report, Institute of Computer Science, UMM.

[3] I. Foster and S. Taylor S. *Strand - New Concepts in Parallel Programming.* Prentice-Hall, Inc, 1990.

[4] D. Gelernter. *Parallelism to the People, or, Experience with Programminig in Linda.*

[5] *POSYBL-1.102 - system documentation.*

[6] L. Sterling and E. Shapiro. *The Art of Prolog.* The MIT Press Cambridge, 1986.

[7] *STRAND'88 USER MANUAL, BUCKINGHAM RELEASE, JUNE 1990.* Strand Software Technologies Limited.

[8] E. Tick. *Parallel Logic Programming.* MIT Press, Cambridge MA, 1991.

[9] E. Tick and C. Banerjee. *Performance Evaluation of the Monaco Compiler and Runtime Kernel.* International Conference on Logic Programming, Budapest, June 1993.

[10] K. Zieliński, M. Gajęcki and G. Czajkowski. *Software Environments for LAN Distributed Computing — a Comparison Study.* Internal Report, Institute of Computer Science, UMM.

Call Forwarding: A Simple Low-Level Code Optimization Technique

Saumya K. Debray[†] Koen De Bosschere[‡] David Gudeman[†]
† Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721, USA

‡ Vakgroep Elektronica en Informatiesystemen
Universiteit Gent
B-9000 Gent, Belgium

Abstract

This paper discusses *call forwarding*, a simple interprocedural optimization technique for dynamically typed languages. The basic idea behind the optimization is very simple: generate multiple entry points for procedures such that a call to a procedure can be directed to the appropriate entry point, bypassing unnecessary code wherever possible. As shown by our experimental results, this simple optimization can be surprisingly effective, and lead to significant performance improvements.

1 Introduction

The code generated for a function or procedure in a dynamically typed language typically has to carry out various type and range checks on its arguments before it can operate on them. These runtime tests can incur a significant performance overhead. As a very simple example, consider the following procedure to compute the average of a list of numbers:

```
average([], Sum, Count, Ave) :- Ave is Sum/Count.
average([H|L], Sum, Count, Ave) :-
     Sum1 is Sum+H, Count1 is Count+1, average(L, Sum1, Count1, Ave).
```

In a straightforward implementation of this procedure, the code generated checks the type of each of its arguments each time around the loop: the first argument must be a (empty or non-empty) list, while the second and third arguments must be numbers.[1] Notice, however, that some of this type checking is unnecessary: the expression `Sum+H` evaluates correctly only if `Sum` is a number, in which case its value is also a number; similarly, `Count+1` evaluates correctly only if `Count` is a number, and in that case it also evaluates to a number. Thus, once the types of `Sum` and `Count` have been checked at the entry to the loop, further type checks on the second and third arguments are not necessary.

The procedure in this example is tail recursive, making it easy to recognize the iterative nature of its computation and use some form of code motion to move the type check out of the

[1] In reality, the generated code would distinguish between the numeric types `int` and `float`, e.g., using "message splitting" techniques as in [3, 4]—the distinction is not important here, and we assume a single numeric type for simplicity of exposition.

loop. In general, however, such redundant type checks may be encountered where the definitions are not tail recursive and where the loop structure is not as easy to recognize. An alternative approach, which works in general, is to generate multiple entry points for the procedure **average/4**, so that a particular call to this procedure can be directed to the "appropriate" entry point, bypassing any code it does not need to execute. In the example above, this would give exactly the desired result: last goal optimization would compile the recursive call to **average/4** into a jump instruction, and noticing that the recursive call does not need to test the types of its second and third arguments, the target of this jump would be chosen to bypass these tests. We refer to this optimization, where multiple entry points are generated for a procedure and the branch instructions at different call sites for that procedure are "forwarded" so as to bypass unnecessary code, as "call forwarding."

Notice that in the example above, even if we generate multiple entry points for **average/4**, the optimization works *only if the tests are generated in the right order*: since it is necessary to test the type of the first argument each time around the loop, the tests on the second and third arguments cannot be bypassed if the type test on the first argument precedes those on the other two arguments. One solution to this problem is to try and choose an order for the tests carefully so as to allow as many tests to be skipped as possible over all call sites for a procedure (taking relative execution frequencies into account, so that frequently executed call sites get to skip more tests wherever possible): it turns out, however, that choosing an optimal order is NP-complete [5]. Another solution is to copy the type tests etc. that are necessary for a call site from the callee to the call site, so that at runtime these tests are executed before jumping to the callee rather than after the jump has been executed. This approach is optimal in the sense that each call site executes exactly those instructions that it has to, and can be done efficiently in polynomial time. However, it has the problem that such unrestricted copying can lead to code bloat, since there may be many call sites for a procedure, each of them requiring the copying of many instructions. A reasonable solution to this problem is to impose a bound on the number of instructions that can be copied to any particular call site. In the current `jc` implementation, this bound is 1.

The issues and optimizations discussed in this paper are primarily at the intermediate code, or virtual machine instruction, level: for this reason, we do not make many assumptions about the source language, except that it is dynamically typed. This covers a wide variety of languages, e.g., functional programming languages such as Lisp and Scheme (e.g., see [12]), logic programming languages such as Prolog [2], Strand [6], GHC [13] and Janus [9, 10], imperative languages such as Icon [8] and SETL [11], and object-oriented languages such as Smalltalk [7] and SELF [4]. The optimization we discuss is likely to be most beneficial for languages and programs where procedure calls are common, and which are therefore liable to benefit significantly from reducing the cost of procedure calls. The assumption of dynamic typing implies that the code generated for a function or procedure will contain type tests, initialization actions (especially for variadic procedures), etc. Such tests on the parameters to a procedure are typically carried out at the entry to the procedure. Moreover, they can typically be carried out in any of a number of different "legal" orders (in general, not all orderings of entry actions may be legal, since some actions may depend on the outcomes of others—for example, in the recursive clause for the procedure **average/4** above, the type of the variable `H` cannot be checked until the first argument to the procedure has been verified to be a cons cell). The code generated for a procedure therefore consists of a set of entry actions in some order, followed by code for its body. For simplicity in the discussion that follows, we assume that each entry action corresponds to a single virtual machine instruction.

2 Implementation Details: Call Forwarding in jc

The jc system [9] is an implementation of a logic programming language called Janus [10], and is currently available by anonymous FTP from cs.arizona.edu. It translates Janus source programs into a sequence of virtual machine instructions, carries out a variety of transformations and optimizations at the virtual machine level, then macro-expands the virtual machine instructions into C code that is compiled using a C compiler. Call forwarding is implemented at the virtual machine level, after all the procedures in the program being compiled have been translated to virtual machine instructions. At this point, procedure calls have been translated to jump instructions, and all instructions are annotated with information about the contents of the general-purpose registers (this includes information about types, constant values, and whether or not the register has been dereferenced). Thus, there is no reason to distinguish procedure calls from other sorts of jumps, and call forwarding, as implemented, is actually a generalized jump optimization.

Before proceeding further, we describe *jump target duplication*, a very simple code transformation that plays an important role in our implementation of call forwarding. The idea is very simple: consider a code sequence involving an unconditional jump:

```
    ...
    goto L
    ...
 L: target_instr_1
    target_instr_2
    ...
```

Then, the instruction target_instr_1 can be duplicated before the jump instruction,[2] and the jump instruction retargeted, to produce the following:

```
    ...
    target_instr_1
    goto L
    ...
 L: target_instr_2
    ...
```

Call forwarding is carried out in three logical passes (but implemented as two real passes):

Pass 1: Local Forwarding : Redirect each jump instruction (including procedure calls) to bypass any instruction at the jump target that can be inferred to be redundant given the information about register contents available at the jump instruction.

Intuitively, this can be thought of as "sliding" the target of the jump instruction over redundant instructions until an instruction is encountered that cannot be inferred to be redundant.

Pass 2: Jump Target Duplication : Examine up to k instructions at the jump target (where k is fixed beforehand) to see if duplicating some of these instructions and retargeting the

[2]In general, it is necessary to ensure that this does not affect use-definition relationships between the jump instruction and the instruction target_instr_1 being duplicated. This may happen if the jump instruction has side effects, e.g., in a "jump to subroutine" instruction that affects the stack pointer. For the simple cases that we consider in the jc implementation of call forwarding, however, such problems do not arise.

jump instruction will uncover further redundant instructions that can be skipped. If so, carry out Jump Target Duplication for the appropriate instructions.

In the benchmarks we examined, it was seen that in most cases, it was enough to consider just one instruction in this pass. For this reason, and because the $k = 1$ case is so simple to implement, the current implementation uses $k = 1$.

Pass 3: Local Forwarding : As for Pass 1.

Passes 2 and 3 are implemented as a single pass over the code, because all of the information needed for pass 3 is gathered in pass 2, and would have to be regenerated if the passes were separated.

3 An Example

Here we illustrate the effects of call forwarding on the code produced by the jc compiler. A point to note is that in Janus, a clause can assign a value to a variable only if that variable is marked as a producer, or "teller": this is indicated via an annotation '^' in the source program and the occurrence of a special tag (the *teller tag*) at runtime. Conversely, a value can be read from an occurrence of a variable only if it is an "asker" occurrence. In jc, arguments are passed in registers as in the WAM, with argument 1 in register r1, argument 2 in register r2, and so on.

Consider the clause

```
fact(N, ^F) :- int(N), N > 0 | fact(N-1, ^F1), F = N*F1.
```

The code generated for this clause before optimization has the form:

```
L0:  if ¬IsInteger(r1) goto G2      % jump to next clause if r1 not an integer
     if ¬IsTeller(r2) goto G2       % jump to next clause if r2 not a teller
     if r1 ≤ 0 goto G2
     ap := 0                        % set activation pointer to 0
     MemCheck(4)                    % enough space for the stack frame?
     AllocFrame(4)
     sp[-1] := r1                   % save N in stack frame
     sp[-3] := r2                   % save F in stack frame
     r1 := r1-1
     r2 := MakeAsker(sp[-2])        % sp[-2] is the asker end for F1
     r2 := MakeTeller(r2)           % r2 is the teller end of F1
     SetReturnAddr(Ret1)            % set return address
     goto L0                        % recursive call
Ret1: ...
```

Note that immediately before the instruction "goto L0" corresponding to the recursive call, we know that r1 contains an integer value and r2 contains a teller. Because of this, the tests on these registers implemented by the first two instructions in the sequence above are unnecessary for the recursive call, and can be bypassed. The first pass of call forwarding does this, producing the code

```
L0:  if ¬IsInteger(r1) goto G2
     if ¬IsTeller(r2) goto G2
L1:  if r1 ≤ 0 goto G2
     ap := 0
     MemCheck(4)
     AllocFrame(4)
     sp[-1] := r1
     sp[-3] := r2
     r1 := r1-1
     r2 := MakeAsker(sp[-2])
     r2 := MakeTeller(r2)
     SetReturnAddr(Ret1)
     goto L1
Ret1: ...
```

Now, the register ap is guaranteed to be 0 at the point of the recursive call, so there is no need to reexecute the instruction 'ap := 0' in the recursive call.[3] Unfortunately, the recursive call enters at L1 and cannot skip the instruction 'if r1 ≤ 0 goto G2'. Pass 2 of our algorithm notices that if we used jump target duplication to duplicate this instruction and retarget the jump instruction to the next instruction, it would uncover the instruction 'ap := 0' that could be skipped by the recursive call. Pass 2 therefore carries out jump target duplication (reversing the sense of the jump along the way as an optimization), resulting in the following (useless labels have been deleted for clarity):

```
L0:  if ¬IsInteger(r1) goto G2
     if ¬IsTeller(r2) goto G2
     if r1 ≤ 0 goto G2
L2:  ap := 0
     MemCheck(4)
     AllocFrame(4)
     sp[-1] := r1
     sp[-3] := r2
     r1 := r1-1
     r2 := MakeAsker(sp[-2])
     r2 := MakeTeller(r2)
     SetReturnAddr(Ret1)
     if r1 > 0 goto L2
     goto G2
Ret1: ...
```

The sense of the jump is reversed during jump target duplication because we assume that early clauses are more likely to commit: this heuristic is useful for recursive procedures encoded with the recursive clauses first. Recall that Pass 2 is not actually implemented as a separate pass, so this code is not ever actually produced. Now Pass 3 can skip the instruction 'ap := 0', producing the following code as its output:

[3]This instruction is necessary for reasons having to do with the Janus runtime system that are beyond the scope of this paper.

Program	w/o forwarding (ms)	with forwarding (ms)	% improvement
binomial	5.95	5.14	13.6
hanoi	186.0	163.0	12.4
tak	299.0	207.0	30.8
nrev	1.17	0.716	38.8
qsort	2.31	1.87	19.0
merge	0.745	0.613	17.7
dnf	0.356	0.191	46.3

Table 1: Performance Improvement due to Call Forwarding

```
L0:  if ¬IsInteger(r1) goto G2
     if ¬IsTeller(r2) goto G2
     if r1 ≤ 0 goto G2
     ap := 0
L3:  MemCheck(4)
     AllocFrame(4)
     sp[-1] := r1
     sp[-3] := r2
     r1 := r1-1
     r2 := MakeAsker(sp[-2])
     r2 := MakeTeller(r2)
     SetReturnAddr(Ret1)
     if r1 > 0 goto L3
     goto G2
Ret1: ...
```

4 Experimental Results

We ran experiments on a number of small benchmarks to evaluate the efficacy of the optimization, i.e., the performance improvements resulting from it. The numbers presented reflect the performance of jc on a Sparcstation-1.

Table 1 gives the improvements in speed resulting from our optimizations, and serves to evaluate the efficacy of call forwarding . The time reported for each benchmark, in milliseconds, is the time taken to execute the program once. This time was obtained by iterating the program long enough to eliminate most effects due to multiprogramming and clock granularity, then dividing the total time taken by the number of iterations. The experiments were repeated 20 times for each benchmark, and the average time taken in each case. Call forwarding accounts for improvements ranging from about 12% to over 45%. Most of this improvement comes from code motion out of inner loops: the vast majority of type tests etc. in a procedure appear as entry actions that are bypassed in recursive calls due to call forwarding, effectively "hoisting" such tests out of inner loops. As a result, much of the runtime overhead from dynamic type checking is optimized away. When combined with other optimizations, such as inlining of arithmetic operations, environment allocation optimization, etc., the combined optimizations give rise to

Program	jc (J) (ms)	Sicstus (S) (ms)	S/J	Quintus (Q) (ms)	Q/J
`hanoi`	163.0	300.0	1.8	690.0	4.2
`tak`	207.0	730.0	3.5	2200.0	10.6
`nrev`	0.716	1.8	2.5	7.9	11.0
`qsort`	1.87	5.1	2.7	9.4	5.0
`factorial`	0.049	0.44	8.9	0.27	5.5

Table 2: The Performance of `jc`, compared with Sicstus and Quintus Prolog

Program	jc (J) (ms)	C (unopt) (ms)	C (opt: -O4)	J/C-unopt	J/C-opt
`nrev`	0.716	0.89	0.52	0.80	1.38
`binomial`	5.14	4.76	3.17	1.08	1.62
`dnf`	0.191	0.191	0.061	1.00	3.13
`qsort`	1.33	1.25	0.34	1.06	3.91
`tak`	207.0	208.0	72.0	1.00	2.88
`factorial`	0.049	0.049	0.036	1.00	1.36

Table 3: The performance of `jc` compared to C

speed improvements typically ranging from 27% to almost 70%.

Table 2 puts these numbers in perspective by comparing the performance of `jc` to Quintus and Sicstus Prolog, two widely used commercial Prolog systems. On comparing the performance numbers from Table 1 for `jc` before and after optimization, it can be seen that the performance of `jc` is competitive with these systems even before the application of the optimizations discussed in this paper. It is easy to take a poorly engineered system with a lot of inefficiencies and get huge performance improvements by eliminating some of these inefficiencies. The point of this table is that when evaluating the efficacy of our optimizations, we were careful to begin with a system with good performance, so as to avoid drawing overly optimistic conclusions. Finally, Table 3 compares the performance of our Janus system with C code for some small benchmarks.[4] Again, these were run on a Sparcstation 1, with `cc` as the C compiler. The programs were written in the style one would expect of a competent C programmer: no recursion (except in `tak` and `nrev`, where it is hard to avoid), destructive updates, and the use of arrays rather than linked lists (except in `nrev`). It can be seen that even without any global dataflow analysis, `jc` is not very far from the performance of the C code, attaining approximately the same performance as unoptimized C code, and being a factor of between 3 and 4 slower than the code produced by optimizing at level `-O4`, on most benchmarks. On some benchmarks, such as `nrev`, `jc` outperforms unoptimized C and is not much slower than optimized C, even though the C program uses destructive assignment and does not allocate new cons cells, while Janus is a single assignment language where the program allocates new cons cells at each iteration—its

[4]The Janus version of `qsort` used in this table is slightly different from that of Table 2: in this case there are explicit integer type tests in the program source, to be consistent with `int` declarations in the C program and allow a fair comparison between the two programs. The presence of these tests provides additional information to the `jc` compiler and allows some additional optimizations.

performance can be attributed at least in part to the benefits of call forwarding.

5 Related Work

The optimizations described here can be seen as generalizing some optimizations for traditional imperative languages [1]. In the special case of a (conditional or unconditional) jump whose target is a (conditional or unconditional) jump instruction, call forwarding generalizes the flow-of-control optimization that collapses chains of jump instructions. Call forwarding is able to deal with conditional jumps to conditional jumps (this turns out to be an important source of performance improvement in practice), while traditional compilers for imperative languages such as C and Fortran typically deal only with jump chains where there is at most one conditional jump (see, for example, [1], p. 556).

When we consider call forwarding for the last call in a recursive clause, what we get is essentially a generalization of code motion out of loops. The reason it is a generalization is that the code that is bypassed due to call forwarding at a particular call site need not be invariant with respect to the entire loop, as is required in traditional algorithms for invariant code motion out of loops. The point is best illustrated by an example: consider a function

```
f(x) = if x = 0 then 1
       else if p(x) then f( g(x-1) )      /* Call Site 1 */
       else f( h(x-1) )                   /* Call Site 2 */
```

Assume that the entry actions for this function include a test that its argument is an integer, and suppose that we know, from dataflow analysis, that `g()` returns an integer, but do not know anything about the return type of `h()`. From the conventional definition of a "loop" in a flow graph (see, for example, [1]), there is one loop in the flow graph of this function that includes both the tail recursive call sites for `f()`. Because of our lack of knowledge about the return type of `h()`, we cannot claim that "the argument to `f()` is an integer" is an invariant for the entire loop. However, using call forwarding we can bypass the integer test in the portion of the loop arising from call site 1. Effectively, this moves some code out of "part of" a loop. Moreover, our algorithm implements interprocedural optimization and can deal with both direct and mutual recursion, as well as non-tail-recursive code, without having to do anything special, while traditional code motion algorithms handle only the intra-procedural case.

Chambers and Ungar consider compile-time optimization techniques to reduce runtime type checking in dynamically typed object-oriented languages [3, 4]. Their approach uses type analysis to generate multiple copies of program fragments, in particular loop bodies, where each copy is specialized to a particular type and therefore can omit some type tests. Some of the effects of the optimization we discuss, e.g., "hoisting" type tests out of loops are similar to effects achieved by the optimization of Chambers and Ungar. In general, however, it is essentially orthogonal to the work described here, in that it is concerned primarily with type inference and code specialization rather than with code ordering. Because of this, the two optimizations are complementary: even if the body of a procedure has been optimized using the techniques of Chambers and Ungar, it may contain type tests etc. at the entry, which are candidates for the optimization we discuss; conversely, the "message splitting" optimization of Chambers and Ungar can enhance the effects of call forwarding considerably.

6 Conclusions

This paper discusses *call forwarding*, a simple interprocedural optimization technique for dynamically typed languages. The basic idea behind the optimization is extremely simple: generate multiple entry points for procedures such that a call to a procedure can be directed to the appropriate entry point, bypassing unnecessary entry actions wherever possible. As shown by our experimental results, this simple optimization can be surprisingly effective, and lead to significant performance improvements.

Acknowledgements: The work of S. K. Debray and D. Gudeman was supported in part by the National Science Foundation under grant number CCR-9123520. The work of K. De Bosschere was supported by the National Fund for Scientific Research of Belgium and by the Belgian National incentive program for fundamental research in Artificial Intelligence, initiated by the Belgian State—Prime Minister's office—Science Policy Programming.

References

[1] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers – Principles, Techniques and Tools*, Addison-Wesley, 1986.

[2] M. Carlsson and J. Widen, *SICStus Prolog User's Manual*, Swedish Institute of Computer Science, Oct. 1988.

[3] C. Chambers and D. Ungar, "Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically Typed Object-Oriented Programs", *Proc. SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990, pp. 150–164. *SIGPLAN Notices* vol. 25 no. 6.

[4] C. Chambers, D. Ungar and E. Lee, "An Efficient Implementation of SELF, A Dynamically Typed Object-Oriented Language Based on Prototypes", *Proc. OOPSLA '89*, New Orleans, LA, 1989, pp. 49–70.

[5] K. De Bosschere, S. K. Debray, D. Gudeman, and S. Kannan, "Call Forwarding: A Simple Interprocedural Optimization Technique for Dynamically Typed Languages", manuscript, Dept. of Computer Science, University of Arizona, Nov. 1992.

[6] I. Foster and S. Taylor, "Strand: A Practical Parallel Programming Tool", *Proc. 1989 North American Conference on Logic Programming*, Cleveland, Ohio, Oct. 1989, pp. 497-512. MIT Press.

[7] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[8] R. E. Griswold and M. T. Griswold, *The Implementation of the Icon Programming Language*, Princeton University Press, Princeton, 1986.

[9] D. Gudeman, K. De Bosschere, and S. K. Debray, "jc : An Efficient and Portable Implementation of Janus", *Proc. Joint International Conference and Symposium on Logic Programming*, Washington DC, Nov. 1992. MIT Press.

[10] V. Saraswat, K. Kahn, and J. Levy, "Janus: A step towards distributed constraint programming", in *Proc. 1990 North American Conference on Logic Programming*, Austin, TX, Oct. 1990, pp. 431-446. MIT Press.

[11] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg, *Programming with Sets: An Introduction to SETL*, Springer-Verlag, 1986.

[12] G. L. Steele Jr., *Common Lisp: The Language*, Digital Press, 1984.

[13] K. Ueda, "Guarded Horn Clauses", in *Concurrent Prolog: Collected Papers*, vol. 1, ed. E. Shapiro, pp. 140-156, 1987. MIT Press.

A Flexible Scheduling and Memory Management Scheme for Non-Deterministic, And-parallel Execution of Logic Programs

Kish Shen[§]
Computer Science Department
Bristol University, United Kingdom
kish@acrc.bristol.ac.uk

Manuel Hermenegildo
Facultad de Informática
U. of Madrid (UPM), Spain
herme@fi.upm.es

Abstract

In this paper, we examine the issue of memory management in the parallel execution of logic programs. We concentrate on non-deterministic and-parallel schemes which we believe present a relatively general set of problems to be solved, including most of those encountered in the memory management of or-parallel systems. We present a distributed stack memory management model which allows flexible scheduling of goals. Previously proposed models (based on the "Marker model") are lacking in that they impose restrictions on the selection of goals to be executed or they may require consume a large amount of virtual memory. This paper first presents results which imply that the above mentioned shortcomings can have significant performance impacts. An extension of the Marker Model is then proposed which allows flexible scheduling of goals while keeping (virtual) memory consumption down. Measurements are presented which show the advantage of this solution. Methods for handling forward and backward execution, cut and roll back are discussed in the context of the proposed scheme. In addition, the paper shows how the same mechanism for flexible scheduling can be applied to allow the efficient handling of the very general form of suspension that can occur in systems which combine several types of and-parallelism and more sophisticated methods of executing logic programs. We believe that the results are applicable to many and- and or-parallel systems.

Keywords: Parallelism, Logic Programming, Memory Management, Performance Analysis, And-parallelism, Or-parallelism, Implementation

1 Introduction

Systems which exploit implicit parallelism in logic programs are attractive because they allow the achievement of faster execution speeds without an increase in programming complexity. Desirable objectives of such systems are to preserve the programming model, offer actual speedups with respect to state of the art sequential systems, and achieve these results with resource efficiency comparable to that of such sequential systems. Understandably most of the research so far has concentrated on the first two objectives. The state of the art is such that several models and some actual implementations have shown a good measure of success at offering a standard programming model and good speedups (*e.g.* [2, 13, 11, 1, 29], *etc.*). With respect to the third issue, resource efficiency, attention has concentrated mostly on processor efficiency, since it is so related to speedup when the number of processors is limited. On the other hand comparatively little effort has been devoted to other issues which can dramatically affect the practicality of an implementation. Perhaps the most important of these issues is *memory efficiency*.

In fact, effective memory management is one of the main reasons for the efficient performance of sequential Prolog systems for realistic, large application programs. It is expected that this issue will become of increasing importance

[§]Some of the research reported in this paper was carried out while this author was at the Computer Laboratory, University of Cambridge, Cambridge, U.K.

in parallel systems: the need to allocate storage for several execution threads can multiply storage requirements. The evaluation of many parallel implementations to date, focussed as mentioned above mainly on the execution speed and the speed up achieved, has been done generally for comparatively small programs that did not consume large amounts of memory. While memory management inefficiencies will not have a large effect on the execution of these programs if enough raw memory is available, they may be a serious limiting factor when real applications are tackled. Without efficient memory management the favourable speedup results obtained to date with parallel logic programming systems will simply not easily carry over to larger programs which consume large amounts of memory.

In this paper, we examine the basic issues and problems of memory management of certain classes of parallel logic programming systems. We consider mainly "distributed-stack" [4, 9] systems, *i.e.* those which mimic in many ways the sequential execution model (based generally on the WAM [26] or a derivative of it) with the objective of preserving speed and resource efficiency at least during sequential threads of execution. These models are generally based on having multiple sets of stacks ("multiple WAMs") on which the processors can work concurrently. In a previous paper [10] we showed how memory management and scheduling are intimately related. Thus we will also necessarily consider scheduling issues. Our aim is to achieve efficient memory usage while allowing flexibility in scheduling. We propose several more flexible solutions to some of the problems raised in [10], and we provide data which we hope can aid an implementor in choosing a scheme. For conciseness our discussion will concentrate on the particular issues that arise during the exploitation of non-deterministic and-parallelism, given their generality: because of the tree topology of or-parallel execution, and as shown in [10], from the memory management point of view, the problems which appear during exploitation of this type of parallelism are a subset of those which appear during arbitrary non-deterministic and-parallelism, which is topologically more general than a tree.

2 General Approach

Sequential logic programming systems obtain much of their performance from doing their own stack-based memory management and through compilation. Storage space is recovered automatically on backtracking, reducing the need for an explicit garbage collector. In addition, a compiled system is more memory efficient than an interpreted system because the compilation process reduces the amount of information that needs to be replicated from one procedure call to another. Moreover, in many systems (such as the DEC-10 Prolog machine [25] and the WAM [26]), further storage optimisation is obtained by the use of a *two stack model*, where the storage of variables is divided between two areas — the local and global stacks. This allows storage in the local stack to be recovered as soon as a clause has been completed without an alternative. Furthermore, through last call optimisation [26], local stack frames (the WAM "environments") can be often reused, effectively turning recursion into iteration.

Ideally, we would like memory management on parallel systems to achieve the same results as those achieved in sequential systems: recovery of storage space during backtracking, minimisation of the replication of state information, and early recovery of some additional storage space. A compiled parallel system is the first step to more efficient memory management, and we shall describe our system in that context, although the techniques should be applicable to interpreted systems as well. Before we introduce our specific approach, we first discuss some general properties of the parallel systems we are considering.

We adopt the subtree-based approach to executing Prolog programs in parallel, which is common to many models. In this approach parallelism is achieved by allowing several entities –which are often called workers– to simultaneously explore the and/or search tree of a program. Each such worker explores the search tree in much the same way as sequential Prolog: depth-first, left-to-right. Generally, each worker will be assigned to a different part of the tree. Thus, the search tree can be thought of as being divided into subtrees, each of which is executed sequentially and referred to as a task. In the case of or-parallelism these subtrees are generally branches of the tree, while in the case of and-parallelism they are contiguous parts of one or more branches. It is often the case that the subtrees are not determined a priori but rather as the tree is being dynamically constructed: as a worker works on a task, opportunities for parallelism are identified and thus marked. When a worker finishes exploring a subtree, it may start exploring another sub-tree which has been identified for parallel execution – this process is referred to as **stealing a task**. It should be noted that if there are no free workers the tasks or subtrees identified by a worker will (eventually) be explored by this worker.

In our attempt to devise efficient memory management schemes for the subtree-based approach our starting point is the "distributed stack" scheme (and its restricted version, the "cactus stack" scheme) [4, 28, 9, 27] which has been used repeatedly in implementations because it offers the potential to achieve the above mentioned goal of approaching sequential memory efficiency. This approach is based on the following observation: as each worker executes a task much as a sequential Prolog system would, many of the implementation techniques used in sequential a Prolog system can be adopted for use in parallel execution, hopefully retaining many of the advantages of the sequential implementation. In particular, the stack-based storage model of sequential Prolog systems is extended for parallel execution in the form of a distributed stack model.

Following on the distributed stack idea we assume the program to be compiled into instructions which are quite similar to those of a Prolog engine, with perhaps some additional instructions related to parallelism. We view each of the workers introduced above as composed of a processing element, capable of executing such code in much the same way as sequential Prolog, and its associated storage, *i.e.* a set of stacks, consisting of the normal sequential Prolog stacks plus perhaps some other areas needed for parallel execution, and a number of registers. As we shall show, it is often useful to distinguish between these two components of a worker. Following [10, 11] we refer to all the storage areas mentioned above as a **stack set** and to the processing element as an **agent**. A worker can be then seen as a given agent attached to a given stack set. Efficient use of the agents, which are really representing the physical processors of the underlying parallel machine, is necessary to achieve good speedups. Efficient use of the stack sets is necessary to keep memory usage reasonable.

Given a task to work on, an agent can execute the related instructions and use the stack set it is attached to in much the same way as in standard Prolog execution. On the other hand a significant deviation can appear for example when the task is finished (also when it suspends): If there are more tasks available, and in order to use the agents and stack sets efficiently the simplest thing is to use the same agent and stack set to perform the new task by using the space beyond that already used by the older task. Thus, the contents of a stack set can be seen as divided into areas, each corresponding to a task. Each such area is referred to as a **stack section**. The ordering of the stack sections on the worker's stack set is the chronological order in which the worker executed the tasks associated with the stack sections. In order to distinguish and manage such sections, they are separated from each other by **markers** [9]. In addition, depending on the nature of the section above or below them, some markers may serve some additional special functions. The marker scheme can be used for both or- and and-parallel systems. In an or-parallel system, specially marked ("public") choice-points can serve as markers (as is done in *e.g.* Aurora [6]), each one corresponding to a "fork" in the parallel task tree. However, in an and-parallel system, not only forks but also "joins" have to be performed on the tasks representing sibling and-goals and more coordination is needed. Thus additional data structures have to be provided to serve as several types of markers.[1] Therefore, an and-parallel marker scheme can be regarded as a generalisation of an or-parallel marker scheme. In this paper we will concentrate on the more general and-parallel scheme, with the understanding that the solutions proposed and results obtained can be applied to an or-parallel scheme, perhaps with simplifications.

The general structure of a marker is shown in Figure 1. A marker is divided into three general parts, a *management* part for managing the marker and sections, a *state* part which stores the pointers to the various stacks, and the values of some other state registers when the marker was allocated, and a *special* part, which serves the special requirements of the various types of markers (and is empty for the basic markers).

Markers are linked to the next and previous markers in the same stack set. The marker can contain a third pointer, the continuation pointer, the use of which will become apparent in the following sections. Figure 1 shows a task (shaded gray) started in the top stack fragment, and continuing in the bottom stack section.

For concreteness, we assume the and-parallel execution of body goals inside a clause is managed by a P_Call Frame, which can be viewed as being part of the environment (and is thus implemented in &-Prolog [11]), or as a type of marker (and is implemented as such in DASWAM [18]). Each and-goal is represented by a "slot" in the frame, which holds enough information to allow the forward and backward execution of the goal. In addition, the P_Call Frame holds a pointer to the task following the completion of the and-parallel execution. This allows the last agent to finish an and-goal in the CGE

[1] It is possible to use only choice-points as markers if and-parallelism is restricted to "deterministic" goals, as in systems such as PNU-Prolog [16] and Andorra-I [29].

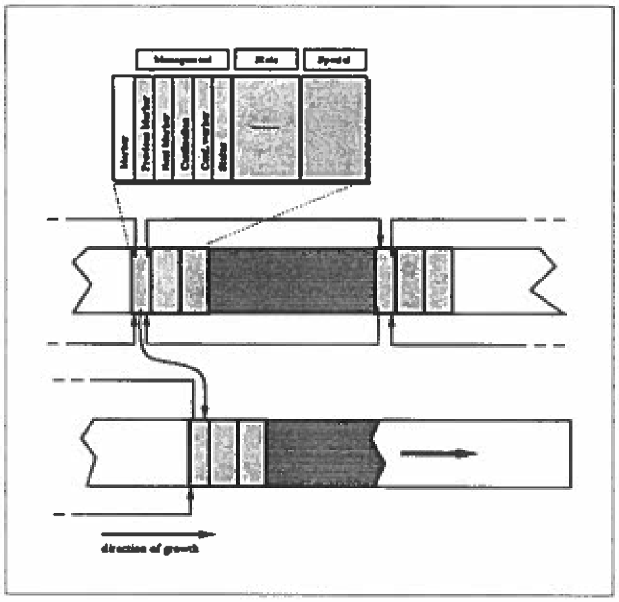
Figure 1: Structure of a marker

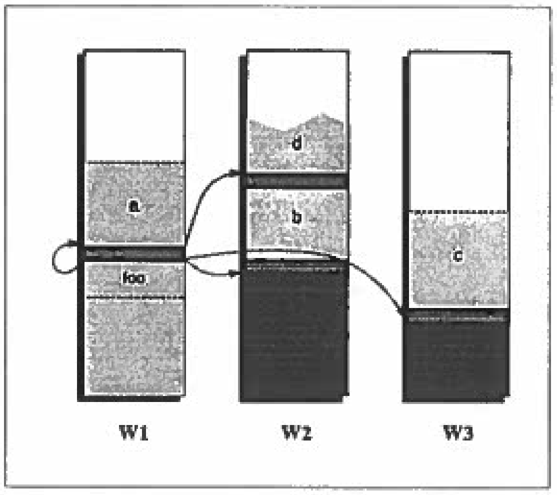

Figure 2: Stack states for a distributed stack scheme

to continue the execution after the and-parallel goals without changing stack set. Figure 2 illustrates the use of the marker scheme (with the P_Call Frame considered as a marker) to represent the following &-Prolog program fragment:[2]

```
foo :- (a & b & c), d.
```

A possible parallel execution of this clause is shown in Figure 2. The and-task of concern is shaded in light grey in the different stacks. W1 is the worker that executes `foo`, which is pushed onto the top of W1's stack. At the CGE, `a` is executed locally, while `b` and `c` are executed remotely (in parallel with the execution of `a`) on W2 and W3 respectively. When `a`, `b` and `c` have all finished execution, W2 picks up the goal after the CGE, `d`, and continues the execution, leaving W1 and W2 idling. Before executing `b` or `c`, both W2 and W3 were idling and were therefore able to pick up `b` and `c`. Both have performed some work, and used their stack. This "old" work is separated from the current work by a *marker*. Markers are also used to mark the start of an CGE (*e.g.* the one separating `foo` from `a`). These specialised *parcall markers* (which correspond to P_Call Frames) contain pointers to link the stack sections of the sibling and-goals of the CGE, and a pointer to the stack section following the CGE.

One major difference between a distributed stack scheme and a sequential stack scheme is that backtracking can occur in any of the stack sections in a stack set, so each stack set can have multiple points of backtracking (and potentially multiple points of growth) at the same time. The pattern of contraction and growth is thus affected by what each stack section represents, and this results in a close relationship between memory management and goal scheduling. This leads to the problems of "trapped goals" and "garbage slots" [9] (also referred to as "holes" in or-parallel systems). In a previous paper [10], solutions were proposed to solve these problems by placing some constraints on how goals should be scheduled. In this paper, we extend this previous work by developing mechanisms that allow for more flexible schedulers.

In addition, we discuss the implementation of *suspension* and *resumption* of a task. The scheme proposed allows a worker to suspend a task and work on another task, with the suspended task resumed later on by any worker. Suspension is very important for at least the following reasons:

- it allows the implementation of Prolog side-effects by allowing a task to suspend when it tries to perform a side-effect. This approach can give rise to more parallelism than in a synchronisation block approach (*e.g.* [8, 14]).

[2]The scheme described here is an extension of the original scheme, where the stack set that started the and-parallel execution had to be the one used for the task following the completion of the and-parallel execution.

- it allows the implementation of more flexible scheduling strategies, *e.g.* strategies that give lower priority to potentially wasted work (*e.g.* [20, 3]).
- it allows the implementation of dependent and-parallelism, by allowing a task to suspend when it does not meet the condition of and-parallel execution (*e.g.* [18]).
- it allows the delaying of execution of some goals, which can be used to implement more sophisticated control than in Prolog, *e.g.* delayed execution ([15, 29]) and constraint satisfaction [23].
- By generalising the suspension mechanism further, then when the space used by a stack is exhausted, additional memory for the stack can be allocated somewhere else in the virtual memory. This allows stacks to be expanded dynamically during execution. Unlike conventional stack shifting in sequential Prolog, no copying of the old stack to a new area is needed, as disjointed stacks are already allowed. Stack shifting is important for a practical sequential Prolog system, but is of even more importance in a parallel system where different stack sets in a system may have very different stack usage requirements.

3 Scheduling Issues

As already mentioned, two important problems which may arise in the distributed stack scheme are the "garbage slot" problem and the "trapped goal" problem. Under the full generality of the distributed stack scheme, each stack set can have many stack sections, and backtracking can occur in these stack sections at any time. Essentially, the "garbage slot" problem comes from the fact that if backtracking is not performed at the top of the stack set, it will leave "holes" in the stack, essentially areas of wasted memory. The "trapped goal" problem occurs when forward execution is to be resumed on a non-topmost stack section that has been backtracked. As the section is not topmost, the task cannot simply proceed blindly using the space available in the stack section as its growth will be blocked by later sections.

As shown in [10] these problems simply will not occur if "age" constraint is imposed on the sections that can be allocated on a stack set. The simple rule is that a task can only be executed on a stack set if it is guaranteed that it will always be backtracked over before all the tasks allocated before it in that stack set. This amounts in practice to a notion of a task being "appropriate" for execution on a given stack set and to restricting scheduling in such a way that the age constraints are respected – this is referred to as the "goal restriction" method. In addition to avoiding the problems mentioned above this restriction has the additional advantages of minimising the number of markers, and the communication needed during backtracking.

The greatest concern with the "goal restriction" approach is that, assuming a simple arrangement of n workers (agent/stack set pairs), parallelism can become limited since these workers may not find appropriate tasks even though other tasks were available. In order to evaluate the impact of goal restriction on speedup we have implemented both a goal restriction scheduler and a flexible scheduler (using the method to be described later) in one of our systems (the pseudo-parallel DASWAM) and compared the speedups obtained from both schedulers for the benchmark programs "boyer", "orsim", "tak", and "bt_cluster". The first three were used in our previous high-level simulation study [19], but, since the limitations of the high-level simulator are not present larger problems are tackled. Boyer is *boyer_nsi* from that study, but here the theorem used in Tick's original query [22] is used. Orsim is the same as the *orsim* program from the simulation study (*i.e.* one of the simulators themselves), and it simulates the program *atlas* with its full database.[3] Tak is the same program as in the simulation study. The last program, bt_cluster, is Tony Beaumont's modified version of British Telecom's original clustering algorithm. As can be observed in Figures 3–4, the results for the first three programs show that the flexible scheduling is substantially better (bt_cluster showed the same results for both scheduling strategies).[4] In most cases, the graph for the restricted scheduling shows regions where the speedup curve becomes nearly flat, *i.e.* where adding workers does not seem to improve the execution time of the program, until suddenly the performance jumps.

[3] orsim(sp2) in [19] was simulating the atlas program with a reduced database.

[4] Note that the speedups are simulated speedups. The cost for locking is not taken into account. In addition, the cost for determining an appropriate goal in the restricted scheduling, is also not taken into account.

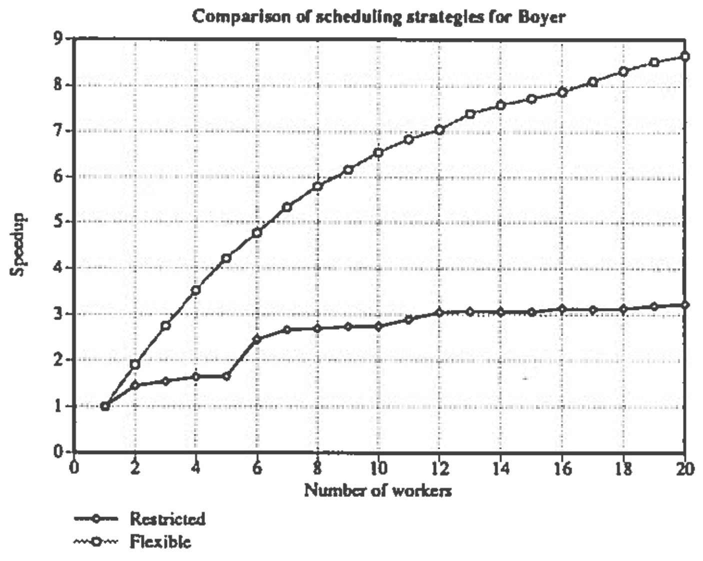

Figure 3: Speed-up for Boyer under the two scheduling strategies

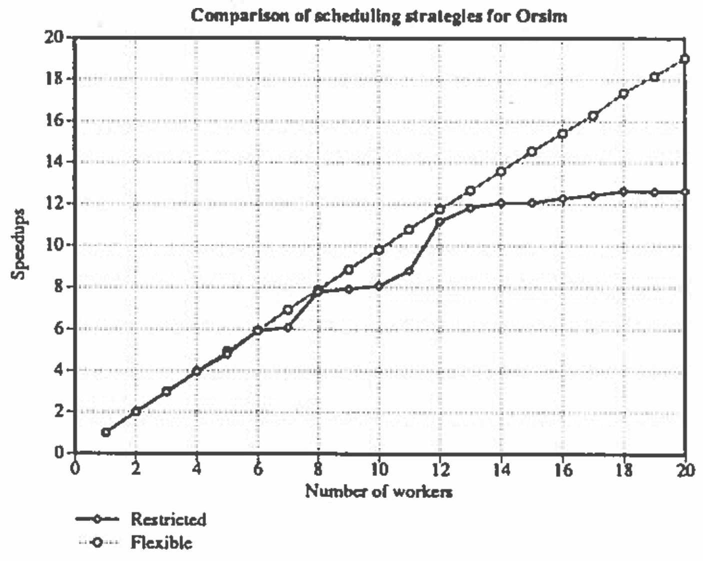

Figure 4: Speed-up for Orsim under the two scheduling strategies

The reason why the restricted strategy can sometimes be much worse than the flexible strategy, as in the flat speedup areas encountered above, is probably because it can limit the ability for a worker to seek new work very early on. Consider the a simple clause "`foo :- (a & b)`" where a generates more parallelism. A worker which executed `b` (or its subgoals) would not be able to exploit the parallelism created by a since all the computation in a and its descendants is younger than those in `b` and therefore it is not appropriate for the worker's stack set. From the results presented it appears that such situations can be quite common. The reason that the speedup curve for the restricted scheduling flattens out frequently is

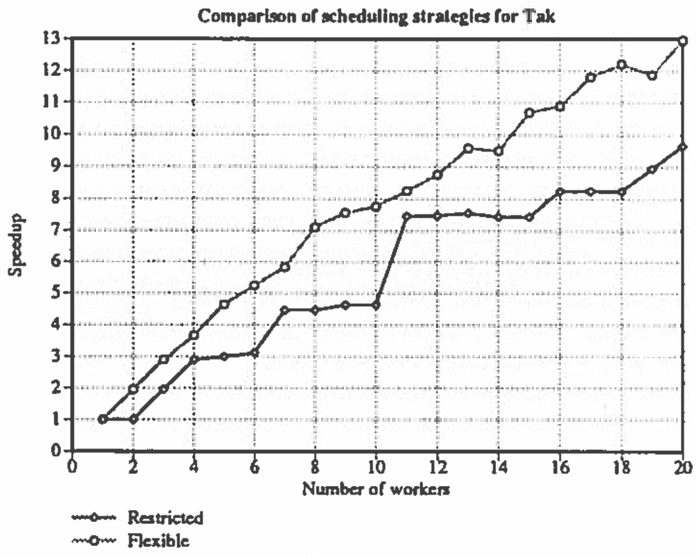

Figure 5: Comparison of speed-ups for Tak under the two scheduling strategies

because b in the above case may initially contain some small amounts of parallelism early in the execution, whereas the more significant parallelism occurs deeper in a. Thus new workers will initially exploit the parallelism in b, giving only small improvements, and then being restricted from exploiting the more profitable parallelism in a. Finally, a newly added worker (with empty and thus unrestricted stack set) can exploit the parallelism in a giving a performance jump. It should be noted that, as workers are added, both scheduling strategies will eventually achieve the same maximum speedup, but a much larger number of workers is needed.

While speedup alone can be a determinant in choosing a flexible scheduling strategy, another problem is the overhead involved in checking tasks for appropriateness, which was not included in the previous measurements. Methods for doing this have been suggested in [10], by Lin [12], and (under a slightly different context) by Tebra [21]. However, in the first scheme the cost can become unbounded both in terms of space and time, and in the other two cases the amount of exploitable parallelism is artificially limited. This is therefore an additional argument in favour of flexible scheduling.

Although the exact impact of goal restriction was not known at the time, a scheme for allowing a certain degree of flexible scheduling was outlined in [10]. This scheme is based on the above mentioned observation that the speedup of both schemes will be identical if enough workers are present. However, of the parts that comprise a worker (agent/stack set) it is really only the stack set that has an age: the agent has no state and could work on another stack set. Stack sets correspond to memory and agents to processors. Of the two, processors are in principle the scarcest resource in a typical parallel machine. The scheme improves the utilisation of agents at the expense of memory consumption: it allows the creation of more stack sets than agents, and the movement of agents from one stack set which is too old to run the currently available goals to another one which is free (no agent is working on it) and is young enough. However, there are still problems with this approach:

- As there are more stack sets than workers, not all stack sets will be used at any point in time. Thus, the storage available in the unused stack sets can actually be also seen as a "hole". As a significant amount of memory is needed for each stack set, and the requirements of the scheduling strategy can require the creation of a large number of stack sets, the multiple stack set approach can lead to a large consumption of (virtual) memory. This may be less of a problem perhaps in future systems, but is certainly a problem in current ones. In particular, consider the case for Boyer: the speedup here for the flexible scheduler increases much quicker than for the restricted scheduler, so many more stack sets would be needed in the restricted strategy before the same speedup as the flexible strategy is achieved.

This could be ameliorated by allocating small stack sets and allowing them to be continued in other stack sets, but, as we will see, this amounts to a similar level of complexity than the scheme we propose, which would however leave less holes.

- The problem of the overhead involved in checking for appropriate goal remains.
- Goal selection can still be restricted in the end in practice. In a real system, the number of stack sets will necessarily be limited, possibly to at most a few times the number of agents available.
- The same inefficiencies will appear when dealing with suspended tasks. In order for backtracking to occur only at the top of each stack set, only "younger" tasks can be allocated on top of a suspended task. If suspension is frequent (as is the case in many dependent and-parallel programs), then the stack sets will rapidly all become suspended, preventing further parallel work.

Because of all the reasons above, and the already discussed usefulness of suspension, clearly a scheme that allows flexible scheduling is desirable. We describe such a scheme in the following section.

4 Dealing with Garbage Slots and Trapped Goals

As an alternative to the appropriate goal schemes, we propose a flexible scheduling scheme which places no restrictions on what task can be started on a stack set after the current task completes or suspends. As we have shown in the previous section this lack of restrictions on scheduling will allow better speedups than with previous approaches. However, the "trapped goal" and "garbage slot" problems, which were effectively avoided in the appropriate goal scheme, will now have to be dealt with in some way.

It is first important to note that these two problems are not of the same importance. Garbage slots in themselves are not a great problem, as their space can eventually be recovered through backtracking and/or by garbage collection. In fact, in or-parallel systems (where only the "garbage slot" problem can occur) it is often tolerated: Aurora [13] allows garbage slots inside its worker's stack. A study of actual performance of the or-parallel PEPSys system suggests that selecting only "appropriate" goals also gives poorer performance in such systems than a more flexible goal selection scheme which can select any goal, even after accounting for the overhead involved in the more complex stack management and the memory loss due to the garbage slots in the stack [7].

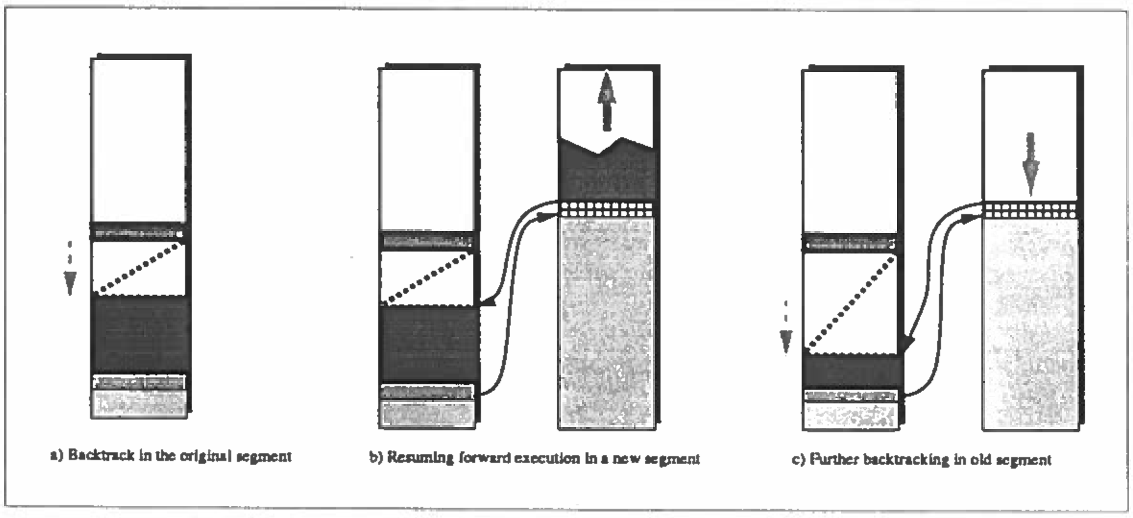
a) Backtrack in the original segment
b) Resuming forward execution in a new segment
c) Further backtracking in old segment

Figure 6: Solving the trapped goal problem

On the other hand the trapped goal problem is more serious: it must be dealt with in some way if a flexible goal scheduling scheme is to be used because trapped goals may simply result in overwriting of valid memory areas and thus incorrect execution. We propose to deal with the problem by introducing a special type of marker: the *continuation marker*.

Essentially, a continuation marker allows the execution of a task to continue in another part of the distributed stack. Thus, the state of a task can spread across one or more disjointed sections. These sections are linked into one logically contiguous section by the continuation markers. The use of the continuation marker for solving the trapped goal problem is shown in Figure 6. The darkly shaded areas correspond to the stack space currently used by a goal, and "garbage" space is indicated by a diagonal dotted line. At first, backtracking takes place in the original stack section, creating some "garbage" space as indicated in Figure 6a. When forward execution is resumed, a continuation marker is allocated on the top of the stack of a free worker, and execution and growth of the stack is resumed at the new worker, instead of at the original stack section. The continuation marker contains a pointer pointing to the top of the still valid work in the old stack section, as shown in Figure 6b.

The system has to be able to handle the backtracking of a goal across the disjointed sections. If backtracking occurs in the new stack section, and it continues to the continuation marker, further backtracking is facilitated by following the pointer in the continuation marker to the old section, the old section is shrunk further, and forward execution is resumed with the pointer in the continuation marker updated, as shown in Figure 6c. Space used by a goal can of course be spread across many stack sections, so the backtracking on a "trapped" section may recursively lead to a continuation marker that points to an even older section for the same task. This is easily handled by the continuation markers.

A question remains however: who should be responsible for performing backtracking on the old section? One possibility is to allow only one agent to work on any stack set at a time — the "owner" of the stack set. This agent can be interrupted from whatever it is doing to do the backtracking. Another alternative is to allow the originally backtracking agent to continue the backtracking. This backtracking is called **remote backtracking**, as it is performed by an agent "remotely", on a stack set that it does not currently own. The advantage of this approach is that the owner of the stack set is free to continue its own work, and multiple backtrackings can occur at the same time in any stack set, giving more scope for parallelism during backward execution. This second approach was taken in both &-Prolog / PWAM [11] and DASWAM [18], in which the scheme proposed has been implemented. In this approach, the general organisation of a stack set is such that there is only one point at any time where the stack can grow — the top of the stack in the topmost section (the **stack tip**); but there can be multiple remote backtrackings (up to one per each non-topmost section) going on in the non-topmost sections at the same time.

One disadvantage of remote backtracking is that it is more complicated and requires more locking and synchronisation than ordinary backtracking. A marker stores part of the state of a worker — the value of various state registers (including all the top of stack pointers) at the time the marker is allocated. This state facilitates the start of remote backtracking. The marker following a section stores enough state information to allow backtracking to start on that section before (below) it. When remote backtracking is first initiated, as in Figure 6a, the new worker reads the values of the various state registers in the marker into its own corresponding registers (having saved the original values beforehand), and remote backtracking is started. When an alternative is found, the original values of the registers for the remotely backtracking worker are restored, and forward execution is started in that worker's top of stack.

With the flexible strategy, a limited form of selecting 'appropriate' goals can still be performed at very little cost: when a task is successfully completed, the worker may then try and select an appropriate sibling and-goal of the just completed task as its next task. That is, it can select a goal in the same CGE that is to the right of the just completed task. This can be done very cheaply because all sibling and-goals of a CGE are recorded in the P_Call Frame. If no such goal is available, any available goals can then be selected.

5 Dealing with suspension

As already discussed, suspension is very useful for many purposes. The same mechanism used for the flexible goal scheduling can be extended to allow new work to be started on top of a suspended stack section so that the suspended task can later be continued elsewhere in the distributed stack when the task is allowed to resume. The continuation marker can be used to allow this continuation. An additional marker is needed to record the state of the worker at suspension so that the state can be restored when the task is resumed. The suspension marker saves enough of the state of a worker to allow the goal to be resumed elsewhere. The amount of information that needs to be recorded depends on how general the suspension is (*i.e.* where it is allowed to take place). For example in &-Prolog and in many other systems, suspension takes place before a goal is called, and in this case, the argument registers do not need to be recorded. In the case of DDAS [18], suspension can occur at any point from the call of a goal to when head unification succeeds, and thus the argument registers need to be preserved. This new marker is called a *suspension marker*. An alternative is to push the state on to the global stack, using mechanisms similar to those used to implement delay primitives in standard Prolog implementations [6].

6 Dealing with Variable Age

Using the flexible scheduling scheme outlined above, there is in principle no need to keep track of task age for scheduling purposes. However, there is still another reason why some scheme for detecting certain ages of objects may be needed: the treatment of unbound variables. In the WAM, when two unbound variables are unified, the older variable is set to point to the younger variable where, since the memory areas are contiguous (and the local and global stack arranged in the appropriate way) is done by a simple address comparison. This is needed to prevent dangling pointers when the environment is deallocated because of last call optimisation or during backtracking. In our parallel approach, we maintain all sequential optimisations inside the execution of each task, thus we also need to deal with the variable age problem. However, it is complicated by the possibility of binding two unbound variables in different stack sections. The mechanism for recognising precedence for selecting "appropriate" goals can also be used to decide how two unbound variables should be bound together [10]. However, once the need for this mechanism is obviated, in the flexible scheduling scheme it may be an unnecessary overhead to preserve this mechanism only for variable age comparison purposes.

We propose three alternative ways to solve these problems. One way is to allow two variables in different stack sections to bind in an arbitrary order (address comparisons can still be used for variables in the same stack section). This means that such variables would have to be globalised and/or trailed in order to avoid dangling pointer problems. As it cannot in general always be known at compile time which variables would be bound to other variables outside their own section, this probably means that an undetermined number of variables will have to be globalised. This scheme has the advantage that it does not require modification to the variable representation. The main disadvantage is that the benefit of splitting the variables into local and global variables is lost and that some additional trailing may be done. Furthermore, it is not clear if the separation into local and global variables is such a great advantage, and recent high-performance sequential Prolog systems such as Van Roy's BAM [24] does not distinguish local and global variables. Thus, this may be a reasonable approach.

Another way to deal with this problem is based on the observation that in order to ensure the correct binding of the local variables, a much less general scheme is needed than that required for determining when a task is "appropriate". In the WAM, a sequentially older local variable can only be directly unified with a younger local variable if the younger variable occurs in a subgoal of the goal the older variable occurs in. Thus, to maintain the correct binding order in local variables, only the ordering of stack sections within the same and-goal is needed. This can be done by associating an age level with each stack section. The age is incremented each time a parallel conjunction is encountered (with each sibling and-goal given the same age to start with), and also each time a new section is started. The age level must also be associated with each variable, indicating when the variable was created. This is implemented by introducing an "unbound" tag to represent the unbound variable, with the "value" part representing the age level of the variable. The mechanism for doing this is similar to the unbound tag needed for maintaining binding arrays in the or-parallel SRI model [27].

The third solution is a combination of the two solutions proposed above: rather than keeping the age of all unbound variables, tags are kept only for the stack sections (storing them in the corresponding marker). Addresses can then be directly compared if they are in the same stack section. Otherwise the addresses of the markers are compared. The success of this approach (as that of the first one) depends on how easy it is to determine to which segment an address belongs. This is rather easy if stack section allocation is segmented, but that can result in inefficient use of storage.

7 Dealing with Cuts

In a sequential WAM such as SICStus' WAM, which has separate local and control stacks, the execution of a cut will be able to remove arbitrarily many choice points (up to the choice point representing the parent goal) from the top of the control stack. This is done by simply setting the top of control stack register to point at the last choice point that is outside the scope of the cut. However, such a simple scheme is not sufficient for a distributed stack, as the choice point to cut to may be in a different stack section. In fact, there can be arbitrarily many stack sections between the current stack section and the stack section that choice point is located on.

Three general situations can be recognised when a cut is encountered:

- The cut cuts to a choice point within the current section. The normal SICStus WAM cut mechanism is used to deal with this.
- The cut cuts to a choice point outside the current section, but still within the same task. First, the top of control stack is reset to that of the current marker, removing any choice points allocated since this stack section was started. Next, choice point has to be removed from the previous stack sections, until the choice point to cut to is reached. This is done by following the pointers in the marker in reverse chronological order, starting from the current stack section, and performing the cut operations on these previous stack sections.

  Each of these previous stack section are bounded by a markers both before and after the stack section. To facilitate the cut operation, each marker contains a pointer field which points to the last valid choice point (if any) on the stack section before it. Initially, when the marker was allocated, this field is set to point at the top of control stack. When a cut operation is performed, this last valid choice point pointer is set to point either at the choice point to cut to, if it is in this stack section, or to the marker before the stack section if the choice point is outside this stack section. In the latter case, the marker before the stack section is used to locate the logically previous stack section, and the cut operation performed recursively on that section.
- The cut cuts across sibling and-goals to its left. An example of this is:

  ```
  foo :- (true => a & b & (c, !) & d).
  ```

  This cut cuts away the choices of `a`, `b`, `c`, as well as `foo`. The main problem is that `a` and `b` are executing in parallel, and may still be executing when the cut is encountered. The effect of the cut is performed in two stages: the choices of `c` is pruned when the cut is encountered, using the methods just described. The slot associated with `a` and `b` are then marked with a 'cut' flag. The pruning of choices on `a` and `b` then takes place when all sibling and-goals between them and the cut has returned a solution, *i.e.* `b` is pruned when `b` returns a solution, `a` is pruned when both `a` and `b` have returned a solution (the finishing of the task that finishes later initiates the pruning). However, if an and-goal to the left of the cut fails, then the 'cut' flag is reset.

In our current systems, the space represented by the discarded choice points on the non-current stack sections cannot be immediately recovered, leaving 'garbage slots' in the control stack. As the markers have to be retained to allow detrailing of variables during the actual backtracking. Note that this is independent of what goal selection scheduling strategy is being used.

The space can be recovered by a garbage collector, or alternatively, if the control stack is separated into a choice point stack for choice points only, and a marker stack for markers only. Some redesign of the existing scheme would be needed, but in principle this would make the recovery of the space occupied by the choice points easier.

Note that no parallelism is lost (except for whatever overhead is needed to perform the cut) in dealing with cuts. This is in contrast to dealing with other side-effects, where the task performing the side-effect must in general suspend until it is leftmost.

	Time	Stack Set	Local	Control	Global	Trail	Goal	Total
Atlas, seq.	17200	-	15	9	8	7	-	39
Atlas, flex	17800	0	17	54	8	10	2	91
Atlas, flex2	17800	0	17	54	8	12	0	91
		1	2	12	0	0	0	12
		T	19	66	8	12	0	103
Atlas, res2	17800	0	17	54	8	12	0	91
		1	2	12	0	0	0	12
		T	19	66	8	12	0	103

Table 1: Absolute Memory Usage - Atlas

	Time	Stack Set	Local	Control	Global	Trail	Goal	Total
orsim seq.	9050000	-	3683	1608	600832	148365	-	754488
orsim flex	9069226	0	45965	89366	601548	230624	0	967503
orsim flex10	926152	0	4666	9172	61135	22562	0	97335
		1	4465	8792	59563	22938	0	95758
		2	4841	8669	60726	22610	0	96846
		3	4465	8464	58930	23230	0	95089
		4	4880	8568	61487	22798	0	97733
		5	4277	8824	58133	23230	0	94464
		6	4503	9202	59113	23382	0	96200
		7	4541	8545	59655	22510	0	95251
		8	4390	8915	59379	23318	0	96002
		9	4955	9400	62551	22996	0	99902
		T	45956	88551	541017	229574	0	905098
orsim res10	1124508	0	4517	8462	57861	21628	0	92468
		1	4466	8468	57913	21814	0	92661
		2	5590	11768	72294	28284	0	117936
		3	4203	8084	55921	21428	0	89636
		4	5591	10862	74033	28202	0	118688
		5	3677	8001	49708	19358	0	80744
		6	3790	7039	49873	19142	0	79844
		7	4542	7629	57593	21516	0	91280
		8	5329	9991	69849	26636	0	111805
		9	4278	9037	56362	22532	0	92209
		T	45983	89341	601407	230548	0	967271

Table 2: Absolute Memory Usage - Orsim

8 Memory Performance

The results corresponding to flexible scheduling in Figures 3–5 were obtained from direct measurements of our implementation of the memory management and flexible scheduling schemes presented in the previous sections, using remote backtracking and variable age tags. These results show that the approach proposed can effectively achieve better speedups than the existing restricted scheduling based approaches. However, and as mentioned before, we are interested not only in speedups, but also in memory efficiency. In this section we study the efficiency of the memory management and flexible scheduling proposed and compare it to that of sequential systems. We also present results using the proposed memory management scheme but with restricted scheduling, and compare them to the results for unrestricted scheduling. The results are presented in Tables 1 to 3 and Figures 7 to 9 for the boyer, orsim, and atlas programs. The atlas program is a standard Prolog benchmark program, and has been used as a benchmark also in our high-level simulation studies [19]. It is chosen because it is a program with a little and-parallelism and much backtracking, which is unlike the other two programs.

Tables 1 to 3 show memory usage figures, all of them taken just before the end of the execution of the program (at the end of the execution all storage is recovered through backtracking). The first column indicates the program. The second column is the time, expressed in terms of abstract machine instructions executed, at which the stack sizes were measured, the third column is the id. of the stack set for which the measurement is taken (we are assuming one agent per stack set). This is followed by the actual memory utilisation for the for the four main stack areas used by WAM/PWAM/DASWAM,

	Time	Stack Set	Local	Control	Global	Trail	Goal	Total
Boyer, seq.	138000	-	14	0	7351	2542	-	9907
Boyer, flex.	138410	0	22465	131712	7243	13776	0	175196
Boyer, flex10	21134	0	2350	13811	813	1430	0	18404
		1	2303	12999	733	1344	0	17379
		2	2085	13078	658	1346	0	17167
		3	2279	13264	734	1402	0	17679
		4	2021	12892	642	1328	0	16883
		5	2018	12729	607	1298	0	16652
		6	1975	12133	602	1334	0	16044
		7	2601	13698	893	1452	0	18644
		8	2517	13432	783	1322	0	18054
		9	2320	13208	758	1424	0	17710
		T	22469	131244	7223	13680	0	174616
Boyer, res10	50292	0	5872	34341	1921	3620	0	45754
		1	565	2946	169	286	0	3966
		2	2	48	0	0	0	50
		3	1500	8531	517	880	0	11428
		4	1298	7480	441	800	0	10019
		5	6023	35415	1919	3694	0	47051
		6	1069	6401	358	658	0	8486
		7	318	1855	110	210	0	2493
		8	5782	34359	1796	3608	0	45545
		9	54	336	12	18	0	420
		T	22483	131712	7243	13674	0	175112

Table 3: Absolute Memory Usage - Boyer

in words (32 bits in size) and for the PWAM/DASWAM goal stack. The last column is the total memory usage in that stack set. Individual stack utilisation is for each stack set, in the cases where there is more than one. In those cases the last row represents the global utilisation for the corresponding type of stack. The last column of this row indicates the total overall amount of storage used. In addition to the precise figures for one instance of time, we also present the evolution of memory usage over the execution of the whole program in Figures 7–9. In these cases, only the total memory usages of all workers (in the cases where there are more than one) for the particular stack is shown. The data for the goal stack is not shown as its usage was relatively quite small.

Results are given for a sequential WAM model based on the SICStus abstract machine [5], which contains several optimisations over the original WAM [26], running the unannotated program (*i.e.* with no parallel constructs); for PWAM/DASWAM running the annotated program on one worker, using flexible scheduling (the restricted scheduling results are the same for this case), and for the annotated program on ten workers both using flexible and restricted scheduling. In the case of atlas, the parallel cases are for two workers, since that is the maximum amount of parallelism. Note that the behaviour of PWAM/DASWAM when running an unannotated program by a single worker is virtually identical to that of the WAM. In any case the worst case scenario of the larger markers and the use of variable age tags, which requires the trailing of two words per variable instead of one, is measured. In addition, the P_Call Frames are allocated on the control stack.

The figures show that the memory usages for the flexible and restricted strategies are very similar, showing that "garbage slots" are not a real problem, at least for these examples. One additional advantage of the flexible strategy is shown by the stack usages of individual workers. In orsim and boyer, which show good speedups, work is divided reasonably evenly between the workers, so the memory usage is divided also reasonably evenly: each worker uses about $\frac{1}{N}$ the amount of memory of the 1 worker case, where N is the number of workers. In the case of the restricted scheduler, the memory usage is divided less evenly as some workers are prevented from working because no appropriate work is available. This difference in utilisation can make the system run out of memory in one stack set when there is still plenty of space in others and force potentially expensive stack set shifting.

Turning now to a comparison between the memory usages of the parallel case for different numbers of workers (the tables show this for one and ten or two workers) we observe that the total amount of memory used is in fact very similar.

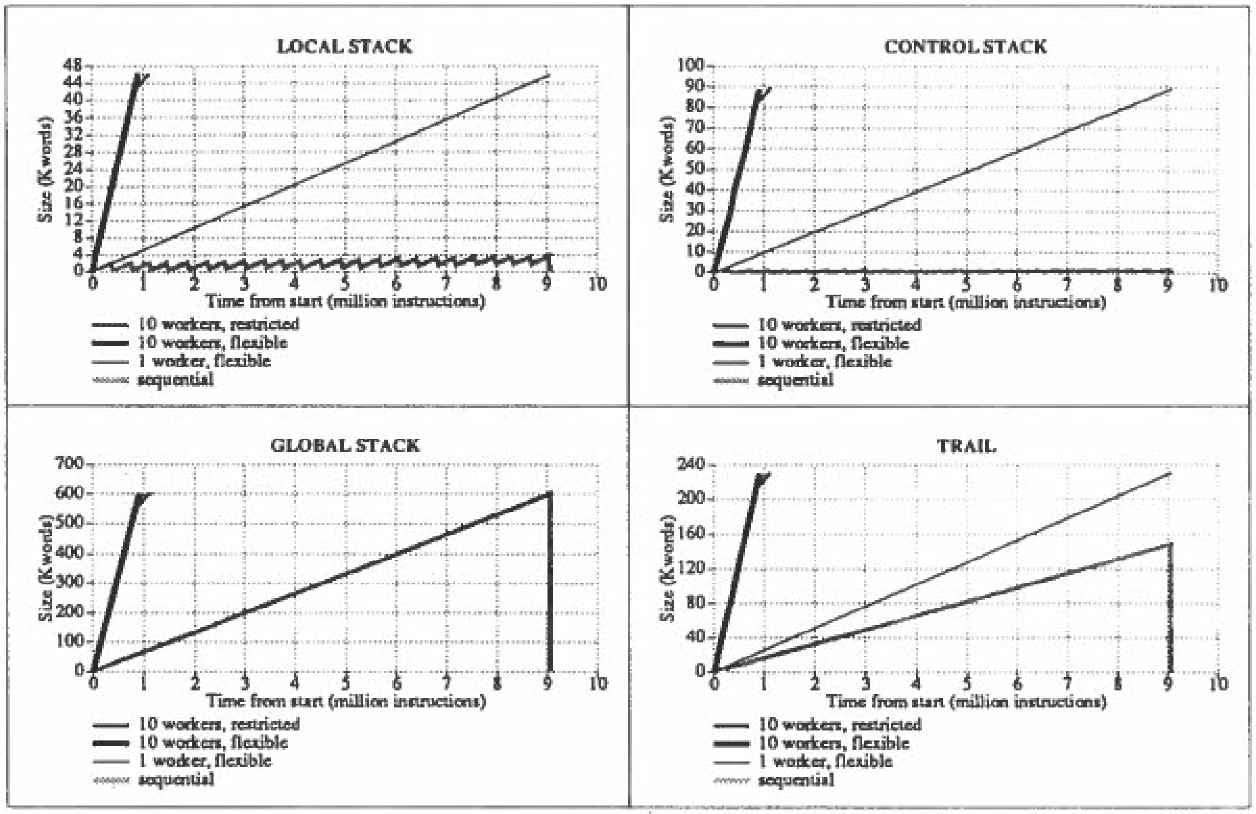

Figure 7: Memory usage for orsim

That is, using more workers does not seem to increase the total memory usage. In fact, the detailed figures would suggest that utilisation decreases slightly, though this might be just a side effect of sampling at one particular point. This preservation of the amount of memory usage is unlike many other parallel schemes, where such usage would increase with an increasing number of workers.

It is also apparent that our approach is able to effectively recover memory on backtracking, as in a sequential system, as shown by Figure 9. This result is further supported by data for the system running programs with dependent and-parallelism, presented in [18].

We now compare the memory utilisation in the parallel case with that of the sequential case. We can observe that the total memory utilisation of the control and local stacks is about the same in the parallel case as in the sequential case for atlas, but much greater for orsim and boyer. The usage of the trail is also generally greater, but this is due mainly to the additional cost of variable age tags. The usage for the global stacks, however, is very similar in all cases. In fact, in the graphs the lines representing the various configurations in the global stack often merge into each other, because their usage is so close. The fact that the global stack usages are similar for the sequential and parallel cases suggests that the variable age tag method is able to preserve the optimisations of the two stack model, without globalising many more variables than in the sequential case, although at the cost of larger trail utilisation. Turning to the combined memory usages of all the stack sets, the parallel system uses between from 30% more memory for orsim to about 18 times more memory for boyer than the sequential system. Both boyer and orsim are close to worst cases from the point of view of total memory utilisation for our approach for a number of reasons: they are highly deterministic and recursive, with much fine grain parallelism which includes the last goal in tail recursive clauses. This, as we will see later, is what creates the large control and local stack utilisations. Furthermore, in boyer the global stack is not very heavily used, which makes the control and local stack figures more directly affect the total.

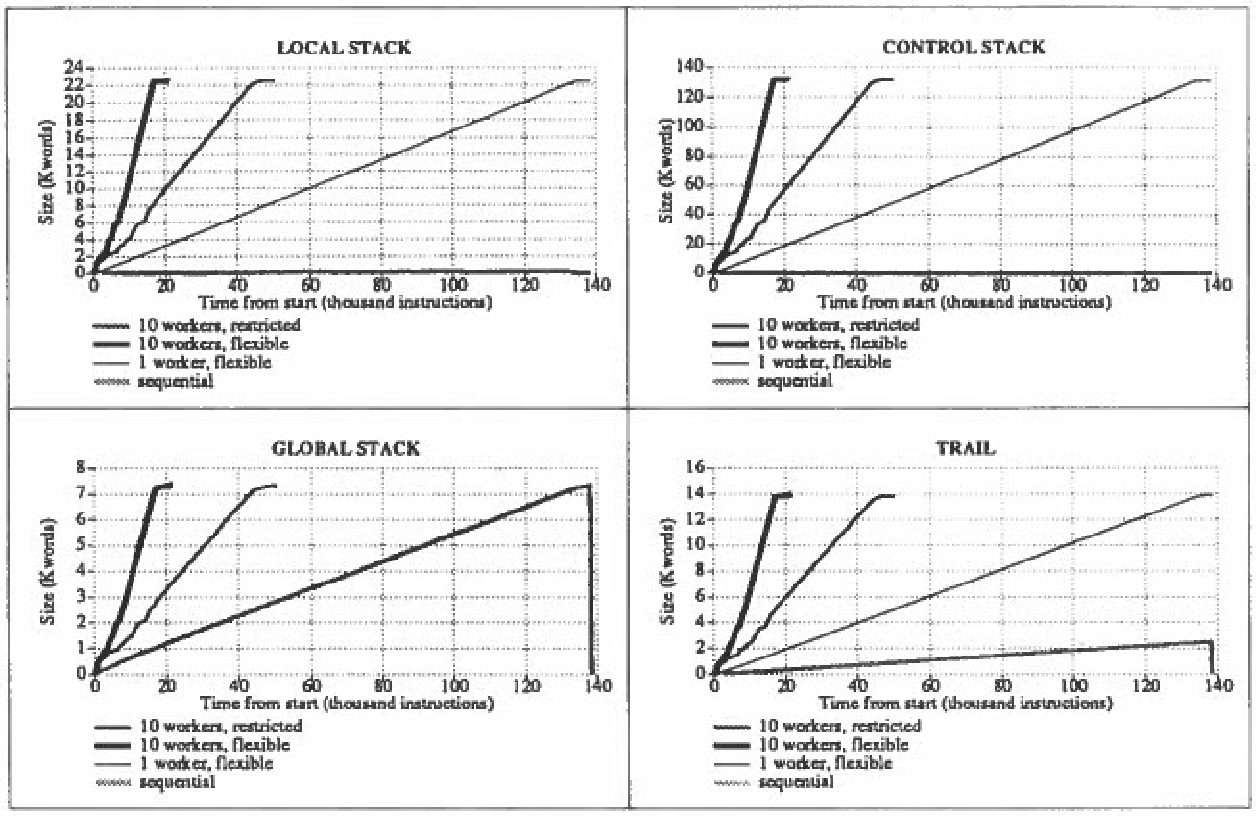

Figure 8: Memory usage for boyer

While the memory consumption is high in absolute value, specially in the case of boyer, it is not that high from the point of view of standard measures often used for parallel systems.[5] These measures are based on integrating the memory demand in time. Thus, the fact that in our approach memory utilisation is spread evenly among workers means that *individually* they do not use excessively more memory than the sequential case and, although the overall memory consumption is higher, that memory is used for a shorter period. In the case of orsim, the memory usage of each worker using the above mentioned measure is actually much smaller than the sequential usage. For boyer, the demand for memory is actually not much worse in the parallel case than in the sequential case: although boyer uses about 18 times more memory in parallel at the end of the execution (figure 8 suggests that this ratio grows steadily from the start of execution), it also runs about 6.5 times faster, so the actual demand is generally less than twice as much as in the sequential system. Moreover, as the total amount of memory used does not seem to increase with increasing number of workers, whereas the speedup does increase, the integrated memory demand would decrease with increasing number of workers, and eventually (in this case probably at around 20 workers) be less than that of the sequential case. Of course, this measure of memory usage assumes that a parallel machine has more real memory than the sequential machine. In fact, ideally, a machine with N processors should have N times as much real memory.

However, independently of the fact that the memory demand achieved can be considered reasonable, we feel there are ways the situation can be improved. For this purpose we will discuss the reasons behind the additional consumption in the different stacks, reason about how much it is ideally possible to achieve, and propose some solutions.

In sequential execution, last call optimisation allows the local stack's memory to be reused. When this last goal is executed in and-parallel, we cannot expect it not to use more space, as what was executed sequentially and which would

[5] Thanks to David H. D. Warren for pointing us to these measures.

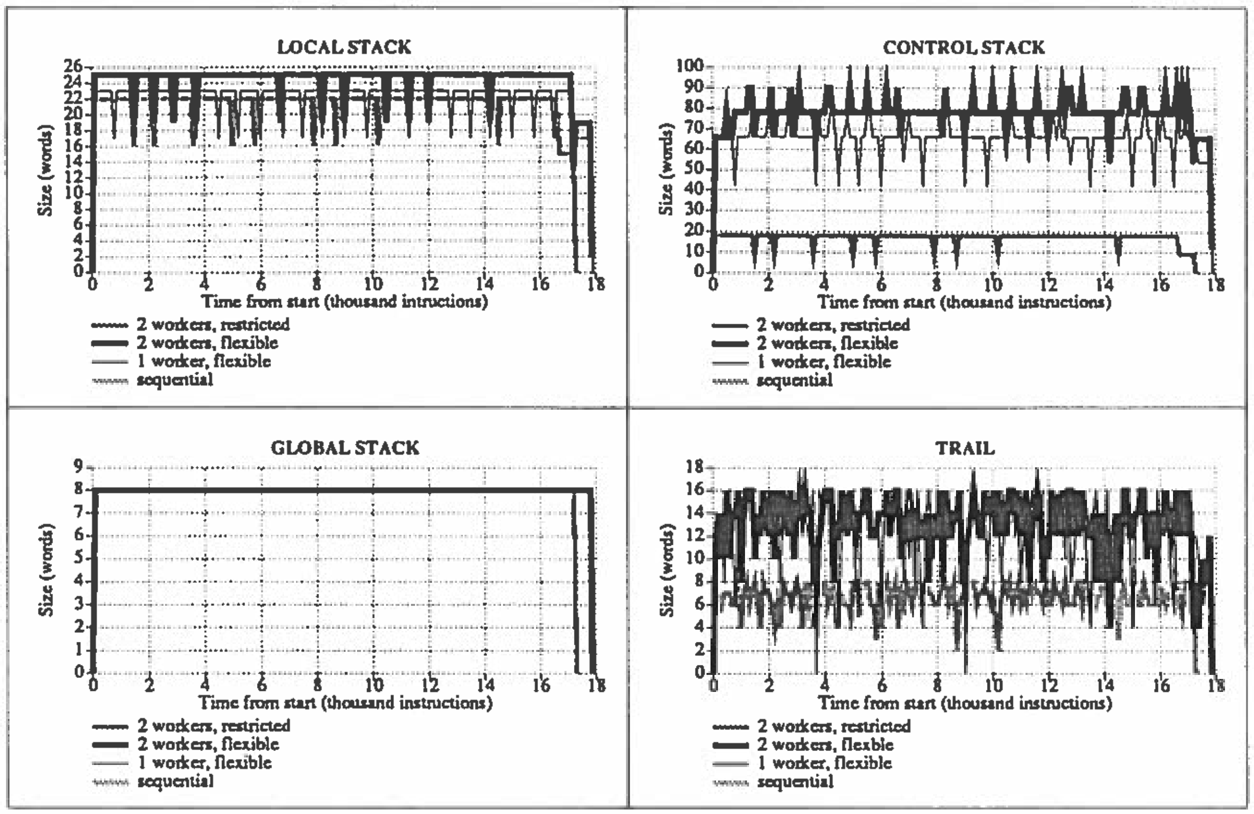

Figure 9: Memory usage for atlas

thus allow the re-usage of the same space, is now executed in parallel and thus must use separate spaces. This is a similar situation to or-parallelism, except that in that case the separate usage occurs all the time (instead of being just a special case), and for all stacks. However, it is reasonable to expect the ideal usage for *each* worker's local stack to be as low as the sequential case. This is not currently achieved by our systems. Thus improvement certainly seems possible, possibly by applying some forms of last call optimisation even in parallel (currently nothing special is done). This could be done for example by having the last goal to be picked up in the parallel call deallocate the environment when after it has constructed its arguments and by preventing parallelisation when it is detected that local execution will occur (*i.e.* parallel goals will end up executed on the current stack set by the current agent). Garbage collection can also be used to recover this memory.

Regarding the control stack, and for deterministic, fine grain cases such as boyer, this area is lightly used in the sequential execution because few if any choice points are allocated. In contrast, for the parallel case, it is to be expected that a marker scheme would use more memory: many markers are allocated on the control stack, because many parallel goals are generated. On the other hand our figures represent an upper bound on consumption since they assume that the markers contain all the information to support dependent and-parallelism. For a purely independent and-parallel system, the markers are actually smaller, and thus the usage of the control stack would also be smaller. Furthermore, our measurements do not include any of the many optimisations which are possible in which the sizes of markers allocated are reduced. These optimisations include allocating smaller markers when it is known that the parallelism exploited is independent; allocating only minimal markers when the goal executed is deterministic (and does not fail), when the same worker is executing successive sibling and-tasks; reduction of the sizes of markers during garbage collection, *etc.* We are actively researching ways to detect and apply some of these optimisations.

Finally, the trail usage figures are quite interesting. The test for trailing is different in the parallel and sequential case. In the sequential case, a binding is trailed if one or more choice points were allocated between the creation of the variable being bound and the binding. The trail test can be implemented by keeping track of the last choice point and comparing its age to that of the variable. A similar scheme can be used in the parallel case, except that the last choice point does not always correspond to the one in the sequential case. The last choice point at the start of execution of each and-goal in a CGE is the last choice point before the entry into the CGE. This can mean that less bindings will be trailed in the parallel case, as is reflected in the trail usage for orsim: recall that variable tags are being used, and so the parallel trail usage should be twice that of the sequential one if the same number of bindings are trailed. The actual parallel usage is less than twice, showing that less bindings are being trailed. On the other hand, the storage corresponding to a goal can spread across many stack sections, and the last choice point may be far removed from the section in which the variable is bound. In the implementation measured we simply trail a binding if the variable was created outside the current section, in order to avoid more complex tests. Thus, the number of bindings trailed can also be more than in the sequential case, as is shown by the trail usage of boyer. It should be reasonably easy to design a more sophisticated trailing test so that less bindings would be trailed in such cases. Note that for atlas, the number of trailings appears to be more or less the same in the sequential and parallel case, as the trail is used more or less twice as much in the parallel case.

9 Dealing with Signals

In and-parallel execution, events that takes place on one task can affect the behaviour of other tasks. For example, under the "restricted" intelligent backtracking scheme of &-Prolog [9], when a goal in a CGE fails, all sibling and-goals are "killed". In DDAS [18, 17], there is even more interaction between and-goals because of the dependent and-parallelism.

Such communications between tasks can be implemented by allowing tasks to send signals to each other. For example, when a task is told to undo its computation (referred to as **roll back**), a 'kill' or 'redo' signal is sent to the task. A 'kill' signal informs the task that receive the signal that it is to be killed. A 'redo' signal, which is needed in DDAS, means that after undoing the computation, the task starts forward execution again. A 'kill' signal does not restart execution of the task. The decision of which signal to send is determined by the exact backward execution scheme used, and will not be discussed further here. Here our interest in how memory can be recovered.

As already discussed, a task is represented in the distributed stack by one or more sections that are logically linked by the continuation markers. The task receiving the signal may not be active, *i.e.* it is not actively being worked on as some worker's top-most stack-section. Indeed, a task may have started its own and-parallel execution, and thus is composed of many descendant and-tasks. Thus, there is no simple representation for a task. However, the start of a task is well defined: a task begins when it picks up an and-goal, and starts execution on it. The start of a task is thus represented by the first stack section of the task. Each slot in the P_Call Frame represents one and-goal in the CGE, contains a pointer to the start of their respective first sections, and the ID of the worker that executed the first section. When a signal is sent to a task, it is sent to the worker that executed the first section, along with the address of where the section is. The worker is interrupted by the signal, and process the signal before returning to doing its original work.

Once a worker receives a signal for a task that it started, the signal must be propagated to the other stack sections of the task. This is done by following the pointers in the various markers to the other stack sections. For both 'kill' and 'redo', the work done by the task receiving the signal is rolled back in much the same way as the undoing of work during backtracking, except that alternatives represented by choice points are not tried. The process of undoing a piece of work may lead to more 'kill' signals, *e.g.* if there are nested CGEs inside one of the tasks being killed. However, in practice, many of these signals (especially in DASWAM) apply to the same tasks, and the system filters out signals that are sent to a task that has already received the same signal. The task is rolled back in semi-chronological order in that stack sections representing later work of a task is undone before those representing the earlier work. The exception is that work done by sibling and-goals can be rolled back in parallel. Note that if a stack section is being remotely backtracked upon when a signal is received, the roll back does not occur until the remote backtracking has finished — this prevents multiple backward execution on the same segment.

The actual mechanism used can best be illustrated by an example. Figure 10 shows an example stack state for a still executing CGE. The lightly shaded stack sections are all executing the same and-task: At W1, during the execution of stack segment "a", a CGE is encountered, and two descendant and-tasks, executing segment "b" (on W1) and "c" (on W2), are

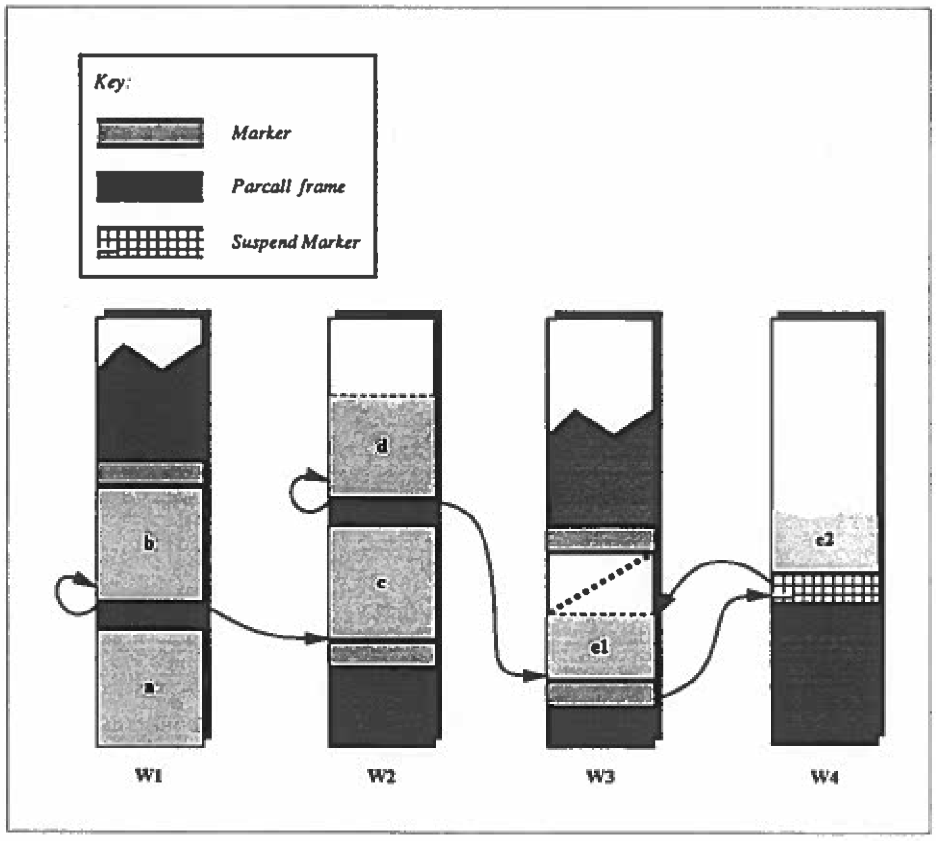

Figure 10: Example stack state before rollback

started. At some point, segment "b" is completed, and a new segment started on top of it. Segment "c" encounters another CGE, spawning segments "d" (executed locally on W2) and "e" (executed on W3). Segment "d" is completed, but no new work is available, so W2 goes into the idle state. Task "e" is for some reason (*e.g.* a suspension that has been resumed) split into two segments: "e1" on W3, and "e2" on W4. e1 has been partially remotely backtracked, and segment "e2" is in the process of forward execution. At this point, the task associated with segment "a" receives a 'kill' signal.

The rollback has to undo the states of segments "a" to "e2". A child section is undone before its parent — *i.e.* starting from "b", "d" and "e2", and working up the hierarchy to "a". The reason for this is that the propagation of the kill signal to descendant and-tasks is asynchronous and takes a finite amount of time, so it is dangerous to undo an ancestral stack state when its descendant may still be running (because they have not yet received the kill signal). For example, if segment "e2" is still running, it might access its ancestral stack segments "e1", "c" and "a". Thus the kill signal is propagated to the youngest child segments before the killing starts. In this example, "b", "e2" and "d" are rolled back, when "e2" has been rolled back, "e1" is rolled back. Segment "c" is rolled back when both its descendant segments ("c" and "e1") are undone. Again, "a" is not rolled back until both its children — "b" and "c" — are rolled back.

Each worker is responsible for performing the roll back in its stack set. One reason for this is to keep the roll back algorithm relatively simple. Another reason is that unlike remote backtracking — where the backtracking worker can perform backtracking on another worker's stack, here there are opportunities for parallelism: *e.g.* segments "b", "d" and "e2" can be rolled-back in parallel with each other.

The case is simple for sections "d" and "e2", as they are the topmost sections. The same applies to section "c", as by the time it is allowed to be roll back, section "d" would be undone already, and "c" would have become the topmost section. In the cases of "b" and "e1", they are not the topmost sections of their worker's stack set during the roll back. In these cases, the worker has to freeze the current work it is doing, perform the roll back, and then go back to its current work.

9.1 Multiple kill/redo signals

During a roll back, a worker may receive other 'kill' or 'redo' signals. Some of these will be to other parts of the stack set, and are independent of the current roll back. These are accumulated and dealt with one after the other. However, some kill/redo signals would interact with the current roll back, because they affect the and-task being rolled back. For example, in figure 10, consider the case of section "a" receiving a kill signal and section "c1" receiving a redo signal when the roll back of "a" is being performed. Another possible interaction is section "a" first receiving a redo, and later a kill signal.

When a signal is sent to a task, the marker representing the start of that task is marked with a flag (saying that the task is 'to be killed' or 'to be redone'). If a subsequent signal is sent to the task (either propagated from another signal to an ancestral task, or a direct signal to this task), then a 'kill' signal would override any 'redo' signal. This simply means setting the flag to 'to be killed'. Otherwise the new signal is filtered out, as the correct action is already taking place.

This scheme is able to handle multiple roll backs, and is thus more flexible than the globally synchronised scheme of APEX [12].

10 Conclusions

We have studied aspects of memory management in the context of non-deterministic and-parallel systems. We have presented a memory management scheme that allows more flexible scheduling of tasks for than previous proposals, and shown that it offers several advantages over such proposals. We also discussed how cuts and roll backs can be handled in our scheme. Although we have concentrated on WAM-based environment stacking models, we believe most of our findings should also apply to other stack-based approaches such as goal stacking models. We have shown that the mechanisms presented are also useful for a number of other purposes, such as the efficient support of suspension in the context of non-deterministic and-parallel goals. We are actively researching many of these possibilities that the scheme has opened up for us, which include the support of dependent and-parallelism (for example as in the DDAS scheme), the efficient parallelisation of constraint logic programming systems (where suspension is often heavily used), more efficient support of side effects, *etc.* Although our results show that by some measures the demand on memory of our current implementations is still close to, and sometimes even better than, that of a sequential system, we can still sometimes use significantly larger raw amounts of total memory than such systems. We are actively researching ways in further reducing the total amount of memory used, basically through determinacy analysis and compilation of marker creation. In addition, a garbage collector that can deal with the special features of our scheme will also be very useful.

11 Acknowledgements

The authors would like to thank Francisco Ballesteros, Mats Carlsson, Manuel Carro, Vitor Santos Costa, and David H. D. Warren for their help and discussions on the work presented in the paper.

Kish Shen would like to thank British Telecom for supporting his fellowship at Bristol University, under which some of this work was performed. He also gratefully acknowledges the partial support of his studies by the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom, under which the early part of this work was performed.

The work reported in this paper was supported in part by ESPRIT projects "PARFORCE" and "ACCLAIM".

References

[1] K. A. M. Ali and R. Karlsson. The Muse Approach to Or-Parallel Prolog. Technical Report SICS/R-90/R9009, Swedish Institute of Computer Science, 1990.

[2] U. C. Baron, J. Chassin de Kergommeaux, M. Hailperin, M. Ratcliffe, P. Robert, J.-C. Syre, and H. Westphal. The Parallel ECRC Prolog System PEPSys: An Overview and Evaluation Results. In *Proc. FGCS'88*, 1988.

[3] A. J. Beaumont and D. H. D. Warren. Scheduling Speculative Work on Or-parallel Prolog Systems. Technical report, Computer Science Department, University of Bristol, 1993. to appear in ICLP'93.

[4] P. Borgwardt. Parallel prolog using stack segments on shared memory multiprocessors. In *International Symposium on Logic Programming*, pages 2–12, Silver Spring, MD, February 1984. Atlantic City, IEEE Computer Society.

[5] M. Carlsson. *SICStus Prolog Internals Manual*. SICS, 1989.

[6] M. Carlsson. *Design and Implementation of an OR-Parallel Prolog Engine*. PhD thesis, The Royal Institute of Technology, Stockholm, Mar. 1990.

[7] J. Chassin de Kergommeaux. Measures of the PEPSys Implementation on the MX500. Technical Report CA-44, European Computer-Industry Research Centre, Arabellaastr. 17, D-8000 München 81, Germany, 1989.

[8] D. DeGroot. Restricted And-Parallelism and Side Effects. In *Proc. 1987 SLP*, 1987.

[9] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, The University of Texas At Austin, 1986.

[10] M. V. Hermenegildo. Relating Goal Scheduling, Precedence, and Memory Management in AND-parallel Execution of Logic Programs. In *Proc. Fourth ICLP*, 1987.

[11] M. V. Hermenegildo and K. J. Green. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *Proc. Seventh ICLP*, 1990.

[12] Y. J. Lin and V. Kumar. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141. University of Washington, MIT Press, August 1988.

[13] E. L. Lusk, R. Butler, T. Disz, R. Olson, R. A. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepielewski, and B. Hausman. The Aurora Or-Parallel Prolog System. In *Proc. FGCS'88*, 1988.

[14] K. Muthukumar and M. V. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. In *Proc. Sixth ILCP*, June 1989.

[15] L. Naish. *Negation and Control in Prolog*, volume 238 of *LNCS*. Springer-Verlag, 1986.

[16] L. Naish. Parallelizing NU-Prolog. Technical report 87/17, The Department of Computer Science, University of Melbourne, 1987.

[17] K. Shen. Exploiting And-parallelism in Prolog: the Dynamic Dependent And-parallel Scheme (DDAS). In *Logic Programming: Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 717–731, 1992.

[18] K. Shen. *Studies of And/Or Parallelism in Prolog*. PhD thesis, Computer Laboratory, University of Cambridge, 1992.

[19] K. Shen and M. V. Hermenegildo. A Simulation Study of Or- and Independent And-parallelism. In *Proc. ILPS*, 1991.

[20] R. Sindaha. The Dharma scheduler – Definitive scheduling in Aurora on Multiprocessors Architecture. In *Proc. Fourth IEEE Symposium on Parallel and Distributed Processing*, 1992.

[21] H. Tebra. *Optimistic And-Parallelism in Prolog*. PhD thesis, Vrije Universiteit te Amsterdam, 1989.

[22] E. Tick. Memory Performance of Lisp and Prolog Programs. In *Proc. Third ICLP*, 1986.

[23] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. Logic Programming. The MIT Press, 1989.

[24] P. L. Van Roy. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, Nov. 1990.

[25] D. H. D. Warren. Implementing prolog - compiling predicate logic programs. Technical Report 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[26] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical report, SRI International, 1983.

[27] D. H. D. Warren. The SRI Model for Or-Parallel Execution of Prolog – Abstract Design and Implementation Issues. In *Proc. Fourth SLP*, 1987.

[28] D. S. Warren. Efficient Prolog Management for Flexible Control Strategies. In *Proc. Second SLP*, 1984.

[29] R. Yang, A. J. Beaumont, I. Dutra, V. Santos Costa, and D. H. D. Warren. Performance of the Compiler-based Andorra-I System. Technical report, Department of Computer Science, University of Bristol, 1992. to appear in ICLP 93.

Memory Management in Quintus Prolog

Jim Crammond and Tim Lindholm
Quintus Corporation
2100 Geng Road
Palo Alto, CA 94303, USA

Abstract

In this paper we describe some of the memory management features of Quintus Prolog Release 3. The main feature of this release is its embeddability, which enables Prolog to be used as a component of an application written in other languages. As a result, the memory system of Quintus Prolog had to be redesigned as it could no longer assume full control of the process address space.

We outline the main components of the memory management system and include some discussion of the types of programs that have given the memory system problems, in particular with memory fragmentation, and how we solved these.

1 Introduction

Much attention is given to the speed of Prolog systems when discussing implementations and yet for many programs where speed could be of some importance, it is often the case that memory usage is of greater concern. For example, if a program consumes sufficient virtual memory to cause swapping to occur then execution speed becomes almost meaningless as the process may spend more time swapping memory to disk than actually executing Prolog code. Worse still, if a Prolog system cannot dynamically expand all Prolog memory areas then the program could fail to complete execution if some area becomes full.

The traditional WAM model [1, 2] defines various stacks that reside in a contiguous area of memory. The problems of expanding stacks if they become full are not usually considered in discussions on the WAM architecture.

In this paper we begin by describing how the traditional WAM model of memory was adapted in Quintus Prolog to support embeddability. The major change here was the implementation of WAM stacks in discontiguous memory. Then we look at four components of the Quintus Prolog memory system: the stack shifter, garbage collector, atom garbage collector and finally the memory allocator for dynamic code and other items.

2 Embedding

A major theme in the development of Release 3 of Quintus Prolog was the ability to embed Prolog within applications written in other languages, particularly C. Embeddability constitutes a suite of otherwise dissimilar features that both extend Prolog's expressiveness and make it sufficiently well-behaved to be used as a component of a multi-paradigm application. These features include a bidirectional foreign interface that allows all data structures from Prolog and the foreign language to be interchanged, for the application to be able to define `main()`, and for the application writer to be able to redefine Prolog's interface with the operating system (e.g. its memory management and I/O).

Well-behavedness means that Prolog could no longer make certain historical assumptions regarding other application components' use of resources. Previous versions of Quintus Prolog assumed control of the address space, requiring that Prolog's run-time data areas (its stacks and code space) be in contiguous memory. It also means that Prolog must not intercept signals that it doesn't really need to know about. Finally, an embedded component needs to be very careful about using resources, especially memory, as it is competing for resources with other software components of the application.

In Quintus Prolog releases prior to Release 3, run-time memory was allocated by the UNIX system call `brk(2)`. Prolog's run-time memory was extended using `sbrk(2)`. In order to keep the memory returned by successive `sbrk(2)` calls contiguous, Prolog imposed some severe restrictions on non-Prolog use of the UNIX heap. Primarily, it required that no foreign component of a Prolog application be able to call `brk()` or `sbrk()`. It defined its own versions of these functions that simply reported an error. Furthermore, because the UNIX heap allocator `malloc(3)` gets its own raw memory from `brk()` or `sbrk()`, Prolog had to ensure that the UNIX `malloc(3)` not be used. Rather, Prolog defined its own versions of `malloc()` and related routines that allocated space out of memory that Prolog controlled.

These restrictions (and others) implied that applications had to be designed so that Prolog controlled the application. There were certain benefits to this: all memory used by an application came from the same memory pool, meaning that intelligent memory management was less likely to be frustrated by memory that was not in its domain [1]. However, in an embedded context, where Prolog is only a component of a complex heterogeneous application, these restrictions could be fatal.

With Release 3 of Quintus Prolog the memory used by Prolog can be discontiguous and interspersed with memory used by other parts of an application in which Prolog is embedded. This is achieved by building Prolog's memory management on top of a primitive layer comprising three C functions: `QU_alloc_mem()`, `QU_alloc_init_mem()` and `QU_free_mem()`.

`QU_alloc_mem(u_long size, u_long alignment, u_long *actualsize)` requests a piece of memory at least `size` bytes long aligned at alignment (a power of 2). The memory returned need not be aligned at `alignment` – Prolog will make use of any extra space given by `actualsize`. `QU_alloc_init_mem()` is like `QU_alloc_mem()` but is called the first time Prolog needs memory, so that any initializations the primitive layer wants to do can be done.

`QU_free_mem(char *mem, u_long size)` returns Prolog memory back to the primitive layer (as a result of a call to the built-in `trimcore/0`).

Quintus supplies a library containing default definitions of these functions based on `sbrk(2)`. The functions are automatically linked into Prolog executables if the application builder doesn't specify alternative definitions. But if Prolog is to be embedded into a system that does its own memory management, these routines can be redefined to use the native support. Doing so can be quite simple: the low-level memory allocation functions of Quintus Prolog can be redefined on top of `malloc(3)` in about a page of C.

3 Changes to the WAM Memory Architecture

When the "classic" WAM memory architecture is discussed, the WAM's various stacks (global stack, local stack, trail and PDL) are typically shown as occupying a single contiguous chunk

[1] It is still possible to achieve this same effect in Quintus Prolog Release 3 by using the provided `QP_malloc()` and `QP_free()` functions, variants of the `malloc(3)` family that allocate space from memory under Prolog's management.

of memory. The WAM makes use of this model to implement certain operations very cheaply, such as determining the ages of variables when unifying two unbound variables and combining two overflow checks by checking for pointer overlap when two stacks grow towards each other.

To support embeddability, Quintus reengineered its WAM to operate with discontiguous stacks. Prolog's stacks are now linked lists of fixed-sized blocks of memory. There are three basic types of memory blocks: those used for global stack, those for local stack and those used for code, atoms and other memory requests (generically referred to as "code space").

There were significant challenges in making this work without grossly impacting the space or time performance of Quintus Prolog. One was the need to consider alternatives to the WAM's equivalence of variable address to variable age. Another was the complexity of shifting or garbage collecting discontiguous stacks. Finally, given the large amount of runtime normally spent doing stack overflow checking, that primitive operation needed to be fast.

The WAM needs to know the relative ages of any two variables that it is about to bind, in order to avoid dangling references later on. By keeping its variables on stacks, it ensures that relative variable age (for variables within each stack) is logically equivalent to the variable's address on the stack. The WAM also enforces ordering between the global and local stacks themselves. As a result, it is easy and fast to determine which variable to bind to the other by simply comparing their addresses.

It was eventually decided that it was not feasible to relax this variable ordering regimen. The alternatives tended to add significant overheads in both memory and runtime. Instead the stack shifter had to be rewritten to allow any new blocks received from `QU_alloc_mem()` to be inserted into Prolog memory such that the ordering of stack segments remained the same (although discontiguous). This involves moving currently in-use blocks around as appropriate to make use of new ones.

The global stack blocks each contain portions of global stack growing up from the bottom of the block and of trail growing down from the top of the block. This means that these stacks can continue to use the standard implementation tricks that make overflow checking fast. However, this does add complexity to the garbage collector as explained later.

As a result of these choices, Quintus Prolog Release 3 did not lose any measurable speed, but gained flexibility of memory management both for the current requirements of embeddability and for possible future work in multithreading and parallelism.

4 Stack Shifter

Shifting the stacks involves incorporating the new memory and moving in-use memory so as to free space where space is needed. When doing this, all pointers to the memory being moved have to be relocated.

Prior to Quintus Prolog Release 3, when the top of Prolog memory was extended the stack shifter simply had to move some portion of in-use memory up by some offset in order to create space where it was needed. Now that Prolog no longer controls the allocation of memory, it has to be able to use whatever block of memory is received from `QU_alloc_mem()` – whether below, above or in the midst of the existing stacks. This involves more complex planning for shifting.

A *local shift* involves only shifting blocks in the local stack. In this case only the blocks being shifted and the ones above those being shifted can contain pointers that need relocating. A *global shift* is more expensive as it involves shifting blocks both in the global stack and local stack and requires scanning all stack blocks for pointers to relocate. One benefit from discontiguous stacks is that stack shifting no longer has to be performed in order to expand code space; this significantly reduces the number of global shifts that are typically executed.

If stack shifting occurs after a garbage collection then Prolog may have free memory blocks under its control that have not yet been returned to the system (via `QU_free_mem()`). In this case, when a request for expansion is made, then some planning is performed to choose the best block of memory from its own pool to perform the minimal amount of work in the stack shifter. This may, for example, result in a local shift instead of a global one.

Planning, as well as choices of block size and layout have ensured that the stack shifting continues to be very fast.

5 Garbage Collector

The component of the system that suffered most because of embeddability changes was the garbage collector [2]. Maintaining fast overflow checking by pointer comparison requires that both pointers be in the same block. That is not a problem on Prolog's forward execution, where if one of the two stacks in a block overflows, both stacks are advanced into the next block. Similarly, this is not normally a problem on backtracking as choice points record pointer pairs that, when restored, will point into the same block. But difficulties arise as a result of garbage collection, which (hopefully!) shrinks the global stack.

It would be undesirable to merely shrink each block's segment of the global stack, as that would only leave usable recovered space in the topmost block. Rather, garbage collection needs to be able to move non-garbage terms into lower blocks. If the global stack is moved disregarding the rest of the WAM state, however, it is very possible that the WAM's pointer overlap overflow check will have been invalidated by moving the global stack pointer to another block. What is worse, pointer relationships that could become current after backtracking also need to be maintained.

Thus, it is crucial that the garbage collector not only be cognizant of what is and is not garbage, but to intelligently compact terms to be retained with respect to severe constraints. These constraints are effectively carried in the choice points, each of which defines a local stack, global stack and trail segment that could at some point become current again. The garbage collector uses this information to pack in-use memory blocks such that valid relationships will exist between each of these stack segments after garbage collection.

Another aspect of the garbage collector is that it has to be able to recognize whether variables in environments contain valid values. Quintus found that initializing environment variables when the environment is allocated resulted in slowdowns in execution of almost all programs (with some particularly bad cases) and thus adopted the policy of leaving variables uninitialized and letting the garbage collector determine whether such variables point to valid terms on the global stack when marking.

The Quintus garbage collector has also been designed such that it can be called from almost any point in execution, such as global stack overflows in the middle of unification, or during the assertion of a clause or in copy_term, etc. This means that the argument registers can be uninitialized as well as environment variables and also that some other parts of the WAM state, such as an incomplete term on the global stack.

The benefit of this approach over one where garbage collection can happen only at specific points in clause execution is that Quintus Prolog does not need to ensure that each overflow

[2] For the Quintus garbage collector, we salute Anil Nair.

condition cannot happen in advance of the various operations like `assert`, `copy_term`, etc. It also makes it possible to construct Prolog terms whilst in C code without imposing any burden on the programmer to worry about space allocation.

6 Atom Garbage Collection

In addition to the features collectively known as embeddability, Quintus customers have requested functionality that move Prolog toward support for programming perpetual processes (or servers). This section discusses an element of recent work in that direction.

Quintus' internal data structures representing atoms limit the number of atoms that can exist concurrently to 2 million. Furthermore, unused atoms have historically never been reclaimed in Quintus Prolog. In a typical server application that is expected to run continually, the Prolog process will slowly create more and more atoms, most of which are only needed for a relatively short time period to service some request. The process could consume unnecessarily large amounts of virtual memory over time and eventually would need to be restarted to clear out garbage atoms.

A new feature in Release 3.1.4 is the addition of atom garbage collection where memory for the strings of atoms no longer referenced can be freed. The atom garbage collector scans compiled and interpreted code and the Prolog stacks to mark all atoms that are in use. It then scans the atom symbol table to remove all unreferenced atoms. The strings for removed atoms are freed, and their symbol table slots chained into a free list. New atoms are allocated numbers from the free list if possible.

A certain amount of complexity was introduced by the fact that Prolog atoms can be passed to C (via `foreign/[2,3]` facts with `+atom` arguments). In addition, the strings for atoms passed to C (via `+string` arguments) are in fact the strings in the atom symbol table, which might potentially be reclaimed. In both these cases, special care needs to be taken by the user if atoms or strings passed into C space are to be guaranteed to survive atom garbage collection.

Quintus chose to solve these problems by never invoking atom garbage collection automatically, but rather by forcing the user to invoke it explicitly via a built-in predicate. In addition to ensuring that the user knows when atom garbage collections are going to happen, this means that most operations to be done on atoms (or their strings) in C will be completed before an atom garbage collection could happen. Only atoms or strings that are to be stored in C space across multiple calls from Prolog to C need to be concerned with protection from atom garbage collection.

Under this scheme, atoms passed to C are reclaimable by default. The user is provided with functions that can register an atom with the atom garbage collector, protecting it from collection over a time it is expected to survive. (Other functions are provided to unregister an atom if it becomes reclaimable again.) It is the user's responsibility to identify and protect the appropriate atoms.

The alternative to this scheme would be to make all atoms passed to C automatically safe from atom garbage collection, and give users the option of declaring appropriate atoms as reclaimable. However, such a scheme imposes overhead on all atoms passed through the foreign interface, tends to protect atoms that you don't mean to protect (e.g. when system code passes them), and forces you to do work to make the majority of atoms reclaimable rather than to protect the minority.

Within the implementation of atom garbage collection, a reference counting system is used so that code protecting or unprotecting atoms passed to C does not overrule the wishes of other code (e.g. libraries) that may be concerned with the same atoms. When using Quintus' qpc compiler, compiled Prolog code (and its atoms) may also be placed in the UNIX executable's text segment. The text segment cannot be modified, of course, so these atoms are effectively made permanent.

With the addition of atom garbage collection, the limit on atom length has been raised from 1023 bytes to 32767 bytes. On earlier systems this limit had been artificially retained to direct users away from relying heavily on longer atoms that would never be reclaimed.

7 Code Space Management

In addition to the management of the Prolog stacks, Quintus Prolog takes great care when managing its code space (a classical heap, although that term is normally used to refer to the WAM global stack). The code space stores compiled and interpreted code, atoms, procedure records, internal tables and space allocated by `QP_malloc()`. Prolog's interface to the code space management is via a group of routines roughly equivalent to the `malloc(3)` family.

The majority of the intelligence going into code space management is in the maintenance of the code space free lists. Memory is initially allocated from the underlying system using functions described earlier, and put on the free lists. Quintus uses a hybrid allocation strategy that is first-fit for certain sorts of requests and essentially best-fit for others. An automatically-invoked utility called the "congealer" is used to periodically traverse the free lists to reduce the fragmentation that inevitably occurs as memory is turned over.

A restriction on code space management is that code space structures are not currently position independent. Although they could in principle be moved via brute-force relocation, this is currently not done. Because the code space is the home for structures of dramatically different lifetimes (everything from transitory assertions to permanent atoms), it is possible for long-lived structures to cause memory management problems.

On `trimcore/0` or `trimcore()`, Quintus Prolog frees to the underlying system all the memory it can. With the default `QU_free_mem()` this is done using `sbrk(2)` to bring down the UNIX "break". The only memory that can be freed is that between the highest in-use chunk and the top of the UNIX "break". Thus, even when relatively little memory is actually in use after a trimcore, there are several ways for the Prolog process to still hang on to large amounts of memory. It is always possible for the application to call `brk()/sbrk()/malloc()`, which will tend to allocate new space on top of that currently used by Prolog. Although there is nothing wrong with this, as long as that foreign space remains in use Prolog cannot bring down the UNIX "break". Similarly, if Prolog allocates a long-lived piece of code space (an atom or a clause), that code space can force Prolog to retain a relatively large footprint even after `trimcore`.

An especially annoying (and painfully common) case of this behavior can occur when an application repeatedly allocates, frees and fragments a chunk of memory. If the fragmentation was for a long-lived (but probably small) structure, and the remaining (probably large) fragment cannot satisfy the allocation in the next iteration, new memory must be allocated. Although this behavior can be shown in many contexts, the most common one is encountered when opening and closing files and asserting a clause in a loop, where the memory structure being allocated, freed and fragmented is an 8192 byte I/O buffer.

The result of this repeated fragmentation and new allocation is a program that requires absurd amounts of virtual memory to run, and whose memory statistics show that it is allocating new memory despite having large amounts "free". It is indeed true that the memory is free (and it is not lost), but it is simply not available in chunks that satisfy the requests of the program!

Quintus Prolog 3.1.4 adds certain heuristics to try to reduce the effects of this cyclic fragmentation. In addition to the previously mentioned free list regimen, repeated fragmentation is avoided by ensuring that once a large chunk of memory is fragmented, that chunk will continue to be fragmented to satisfy smaller memory requests. Although such changes reduce the problem in many cases, they do not remove it altogether. Until all of code space can be relocated, pathological cases of free list management will remain possible.

8 Conclusion

We have discussed various aspects of the memory management system in Quintus Prolog. In order to achieve embeddability Quintus Prolog moved to a model of using discontiguous stacks for the WAM data areas. This required some design changes and in general Quintus tried to maintain efficiency by arranging that certain important WAM optimizations could still be used during "normal" execution with this model at the expense of more complex actions in the relatively infrequent calls to the stack shifter and garbage collector.

We have implemented a garbage collector for atom space to enable Quintus Prolog to be used for perpetual processes and have improved the code space manager to reduce the chances of virtual memory growing due to memory fragmentation.

Acknowledgement

The memory management work in Quintus Release 3 benefited from design discussions with a number of people at Quintus.

References

[1] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, Artificial Intelligence Center, SRI International, 333 Ravenswood Ave, Menlo Park CA, 1983.

[2] H. Aït Kaci. *Warren's Abstract Machine.* MIT Press, Cambridge MA, 1991.

Implementing Logic Programming Systems: The Quiche-Eating Approach

Saumya K. Debray
Dept. of Computer Science
The University of Arizona
Tucson, AZ 85721, USA

Real Men don't eat quiche.
— Bruce Feirstein

In recent years, it seems to have become somewhat unfashionable to build implementations of logic programming languages by translating them to Prolog. Instead, implementors appear, more and more, to be designing their own abstract instruction sets and indulging in a great deal of slow and painful low-level hacking. This paper argues that in many cases, it is preferable to build systems by translating programs to Prolog, using a good underlying Prolog system, and using dataflow analysis and high-level optimizations to reduce performance overheads. In support of our arguments, we compare two sequential implementations of committed choice languages: QD-Janus, which translates to Prolog, and FCP(:), which compiles to a low-level byte-coded instruction set. Even though QD-Janus took significantly less time to implement, its performance is significantly better than that of FCP(:).

1 Introduction

Implementations of logic programming languages can be broadly divided into two classes: *Real implementations*, where programs are compiled into low-level instruction sets designed for the purpose, or into C or assembly language, and which involve much pain and suffering for the implementors; and *Quiche-eating implementations*, where programs are compiled into Prolog or other comparable high-level language, and which require considerably less effort on the part of the implementor.

An examination of implementation-oriented papers published within logic programming circles over the last decade or so reveals an intriguing trend. A decade ago, when available Prolog systems were, by and large, (relatively) slow and unsophisticated, it was not uncommon to find implementors of other (i.e., non-Prolog) logic programming languages building their systems by translating their programs to Prolog. Since then, Prolog implementations have seen huge improvements: they have have become cleverer, faster, and more flexible, with robust and efficient sequential and parallel implementations, optimizing and parallelizing compilers, extensions to unification involving constraint solving over various domains, and flexible control strategies. At the same time, implementors of other logic programming languages appear to have become increasingly reluctant to consider translation to Prolog as a viable implementation alternative (or, if they are not, are too embarrassed to talk about it much); instead, they rely more and more on designing and implementing their own low-level abstract instruction sets. In this paper, we argue that in many cases, translation to Prolog offers numerous advantages and few (often surmountable) disadvantages compared to building an implementation from the ground up, and should be more seriously and more frequently considered as an implementation vehicle for other logic programming languages.

There are a number of obvious advantages to implementing a logic programming language by translation to Prolog, stemming largely from the smaller "semantic distance" between two logic programming languages compared to that between a logic programming language and a low-level abstract instruction set: a translator to Prolog is likely to be simpler to build than one to a lower-level instruction set, more reliable, and easier to maintain, debug, and port. As far as we can see, the main argument against compilation to Prolog is that this incurs an unacceptable performance penalty.[1] We claim, however, that this argument is, in most cases, specious. It seems to us that implementors opting to design and build their own low-level implementations often fail to give adequate consideration to the fact that writing a compiler and run-time system for a low-level instruction set is only part of the task of building a good implementation: it is necessary also to construct a garbage collector, stack shifter, etc., which are well-understood yet nontrivial and important components of a system. When translating to Prolog, these components are generally available in the underlying Prolog system, and do not have to be built from scratch. Because of this, and the speed with which a translator to Prolog can be implemented, we can very quickly build a "base implementation" consisting of a simple translator to Prolog (together with the garbage collector, stack shifter, etc., that come with the underlying Prolog system). Then, the considerable amounts of time that would otherwise have been spent building a compiler to a lower-level instruction set, reimplementing a garbage collector, etc., can instead be devoted to implementing program analyses, transformations, and optimizations aimed at overcoming the performance penalty of a naive translation to Prolog. This is, in our opinion, likely to be considerably more interesting and intellectually rewarding, and may very well make it possible to attain, with much less effort, performance comparable to that of a customized lower-level implementation of similar functionality.

In this paper, we examine these arguments in the concrete context of two committed-choice language implementations: QD-Janus [2], a sequential implementation of Janus [7] built on top of Sicstus Prolog v2.1; and a sequential implementation of a dialect of Flat Concurrent Prolog by Kliger [6]. The languages are essentially the same, modulo minor differences in the concrete syntax, so the comparison of these systems is meaningful. We demonstrate that even though the QD-Janus implementation was completed within a fairly short period of time, the optimizations implemented make it possible to eliminate most of the overheads compared to the underlying Prolog system, and produce a system that is considerably faster than the customized low-level implementation of Kliger.

2 Language Overviews

Janus is a flat committed choice language that is similar, in many respects, to Flat Concurrent Prolog [6] and Flat GHC [10]. A clause is of the form '*Head* :− *Guard* | *Body*' where *Head* is an atom (the head of the clause), *Guard* is a set of *ask actions*, i.e., primitive tests (the guard), and *Body* is a set of procedure calls and *tell actions.* There are a few minor differences, compared to other committed-choice languages, in terms of the primitive data types (and hence, the set of ask and tell actions) supported: unlike FCP and FGHC, Janus supports arrays and bags. The

[1] It is rumored that implementations involving a great deal of incomprehensible low-level grunge are sometimes perceived as being "sexier" than those effecting an easy-to-follow translation to Prolog, making papers about the former easier to publish (especially if the title contains phrases like "Extension to the WAM") and detracting from the appeal of the latter. Such rumors are, of course, scurrilous, baseless, irresponsible, and probably untrue, and should be condemned by all right-thinking people.

most important difference, however, is the *two-occurrence restriction* of Janus. This restriction states, essentially, that in any clause, a variable whose value cannot be inferred to be atomic from the guard operations is allowed to have at most two occurrences: one of these occurrences is annotated to be the "writable" occurrence, and the other is the readable occurrence. Only the writable occurrence of a variable may be assigned to. Thus, variables in effect serve as point-to-point communication channels; other language constructs allow many-to-one and one-to-many communication.

A detailed discussion of the rationale for the two-occurrence restriction is beyond the scope of this paper: the interested reader is referred to [7]. For our purposes, it suffices to note that it effectively forces each variable to have exactly one reader and one writer. It has been our experience that the restriction can pose problems for both the implementor (determining whether an arbitrary Janus program satisfies the restriction is undecidable) and the programmer (the transcription of even simple algorithms, such as matrix transposition, can become quite awkward). Moreover, it is not obvious to us that these problems are offset by significant programming or implementation advantages elsewhere. For these reasons, QD-Janus does not enforce the two-occurrence restriction, and no aspect of the QD-Janus implementation depends on it. In fact, because QD-Janus uses the underlying Prolog system's unification routines, it can be thought of as simply a sequential implementation of FCP or FGHC, but with a quirky concrete syntax. (One could argue that since Janus requires the writable occurrence of a variable to be annotated as such, and that this annotation has to be checked at runtime, it is burdened with additional runtime overheads that are not incurred in FCP or FGHC. As we will see, however, the performance of QD-Janus is nevertheless not too bad.)

The dialect of FCP of interest here is FCP(:), considered by Kliger [6]. Apart from the set of primitive data types supported (and the two-occurrence restriction of Janus, which we will ignore), the main differences between FCP(:) and Janus are the following: (*i*) in FCP(:), tell actions are permitted in guards: a clause is of the form '*Head* :− *AskGuard* : *TellGuard* | *Body*' where *AskGuard* is a set of ask actions and *TellGuard* is a set of tell actions; and (*ii*) unlike Janus, explicit annotations are not necessary for the writable occurrence of a variable.

3 Implementation Overviews

3.1 QD-Janus

The entire QD-Janus system is currently written in Prolog and implemented on top of Sicstus Prolog v2.1 [1]. The entire system is currently about 4500 lines of Prolog code, and is available by anonymous FTP from `cs.arizona.edu`. A detailed discussion of the system is beyond the scope of this paper, but the interested reader is referred to [2].

The analysis most fundamental to the QD-Janus compiler is *demand analysis*. This is a purely local analysis (i.e., not requiring fixpoint computations over the entire program) of the head and guards of the clauses for a procedure to determine how the input arguments must be instantiated if any of the clauses are to commit. Based on this analysis, procedures are classified as either "unimodal," where a single "instantiation pattern" describes the instantiation necessary for the head arguments in order that any clause for that procedure commits; and "multimodal," where different clauses have different instantiation patterns. As an example, the procedure

```
fact(0, ^F) :- F = 1.
fact(N, ^F) :- N > 0 | fact(N-1, ^F1), F = N*F1.
```

is unimodal, because each of its clauses requires that the first argument be instantiated and the second argument be a teller. On the other hand, the procedure

```
merge(L, [], ^O) :- O = L.
merge([], L, ^O) :- O = L.
merge(L, [H|L1], ^O) :- O = [H|O1], merge(L, L1, ^O1).
merge([H|L1], L, ^O) :- O = [H|O1], merge(L, L1, ^O1).
```

is multimodal because different clauses demand different instantiation patterns in order to commit, e.g., the first and third clauses do not require that the first argument be instantiated, while the second and fourth clauses do not require that the second argument be instantiated. The instantiation patterns obtained from demand analysis govern the tests generated to determine whether a procedure should suspend. Not unexpectedly, it turns out that the code that can be generated for unimodal procedures is considerably simpler than for multimodal procedures. The tests generated for suspension represent overhead compared to the underlying Prolog implementation, and dataflow analysis is used to reduce these tests where possible.

Another important analysis—again, one that is local to a procedure—is that for common functors/annotations in clause heads and calls. The idea is that in the clauses for `merge/3` above, it is apparent from an examination of the clause heads that all of the clauses require the third argument to be a teller, while each of the recursive calls in the body has the third argument explicitly annotated as a teller, so these calls will always satisfy the teller requirement on the third argument. The runtime checks on the third argument can therefore be omitted for recursive calls. This is akin to code motion out of loops. At this time, this optimization is done only for procedures that are direct recursive (as an example of the importance of this optimization, the `mastermind` benchmark considered in Section 4 pays a performance penalty of over 13% because this optimization is not currently applied to indirectly recursive procedures).

Apart from these optimizations, the compiler uses simple dataflow analyses to obtain information about "calling patterns" for procedures, about their suspension behavior, and about the instantiation of their outputs (this analysis is fairly crude at this time, and can potentially be improved significantly). This information is used for various optimizations when generating Prolog code, such as reducing unnecessary runtime suspension tests, performing arithmetic in-line where possible, etc.

Even though the QD-Janus compiler is written entirely in Prolog and relies heavily on various analyses and optimizations to improve performance, compilation speed is quite fast, and is dominated by the I/O time for reading in the Janus programs and writing out the translated Prolog code. This indicates that we could improve the sophistication of the analyses and optimizations considerably without noticeably affecting translation speed.

3.2 FCP(:)

Kliger's implementation compiles FCP(:) programs into emulated byte code for a virtual machine instruction set. The compiler uses procedure-level and program-level analyses to effect a number of optimizations. The most important of these optimizations is decision graph compilation, where the guards of the clauses defining a procedure are compiled into a decision graph (which can be thought of as a compact representation of a decision tree). The idea here is

Program	QD-Janus (QD) (ms)	Sicstus Prolog (S) (ms)	QD/S
`nrev(30)`	1.16	0.84	1.38
`hanoi(17)`	2913.2	2642.0	1.10
`e`	688.0	680.0	1.01
`qsort(50)`	18.0	13.5	1.33
`pi`	345.6	345.5	1.00
`dnf(50)`	126.5	123.7	1.02
`pascal(200)`	872.0	730.5	1.19
`queen(8)`	7350.0	4860.0	1.51
`prime(10000)`	14659.3	10612.0	1.38
Geometric Mean of QD/S :			1.20

Table 1: Relative Speeds of QD-Janus and Sicstus Prolog

to organize tests common to different clauses in a procedure in order to avoid executing them repeatedly. Apart from this, global dataflow analysis is used to optimize general unifications to simple assignments where possible, to eliminate tests in guards where their results can be inferred, to eliminate dead code in sub-graphs of decision graphs, to execute simple body goals in-line, and to reduce memory use by unboxing integers.

4 Performance

This section compares the performance of QD-Janus with (*i*) the underlying Sicstus Prolog implementation, and (*ii*) the FCP implementation of Kliger that compiles to a low-level abstract machine instruction set. The underlying hardware platform in each case is a Sparcstation-2 with a 40 MHz clock. The benchmarks, taken mostly from Kliger's thesis [6], are listed in the Appendix.

4.1 QD-Janus compared to Sicstus Prolog v2.1

The performance of QD-Janus compared to the underlying Sicstus Prolog v2.1 system (compiling to native code) is given in Table 1. This indicates the overheads introduced by the execution model of Janus, such as checking whether a procedure should suspend if its inputs are inadequately instantiated, which cannot be removed by the QD-Janus compiler. The figures indicate that this overhead is not very large, typically below 50%: indeed, on some programs where the cost of floating-point computations dominates (e.g., the benchmarks `e` and `pi`), the overheads incurred by QD-Janus are seen to be negligible. Even in a program such as `nrev`, which does very little "interesting" computation, it can be seen that QD-Janus is only 38% slower than the Prolog program.

These results validate our claim that by careful generation of Prolog code, using dataflow analysis and optimization to avoid unnecessary work where possible, it is possible to implement a system whose performance is not significantly worse than that of a good Prolog system.

4.2 QD-Janus compared to FCP(:)

Program	QD-Janus (QD) (ms)	FCP (ms)	FCP/QD
`nrev(30)`	1.16	4.5	3.89
`pascal(200)`	872.0	4490	5.14
`queen_1(8)`	49.8	110	2.21
`prime(10000)`	14659.3	16560	1.13
`tak`	367.2	1720	4.68
`combination`	257.4	1892	7.35
`deriv`	144.2	690	4.78
`mastermind`	24866.3	30200	1.21
`nand`	1197.2	4480	3.74
Geometric Mean of FCP/QD :			3.22

Table 2: Relative Speeds of QD-Janus and FCP

The speed of QD-Janus compared to Kliger's FCP(:) implementation [6] is given in Table 2. It can be seen that the performance of QD-Janus is better than that of the FCP(:) implementation on most benchmarks, being more than thrice as fast on the average, even though the FCP(:) system compiles down to a lower level and carries out various low-level optimizations, such as generating decision trees, that are not done by the QD-Janus system. These numbers validate our claim that careful compilation to Prolog, with a good underlying Prolog implementation, can produce performance comparable or superior to that of a customized low-level implementation.

Table 3 gives the relative heap utilization of QD-Janus compared to FCP(:).[2] It can be seen that the heap requirements of QD-Janus are typically two to three orders of magnitude better than those of the FCP(:) implementation, with an average heap consumption that is more than 100 times lower. We conjecture that FCP(:) has such relatively high heap requirements because it allocates activation frames on the heap; by contrast, QD-Janus uses a stack-oriented scheme that requires less space, is easier to reclaim, and can be expected to be more efficient.

5 Discussion

QD-Janus began as a project to gain experience with the operational behavior of Janus programs and to provide a "sanity check" for a lower-level implementation of Janus we were working on [3]. As such, we did not expect very high performance from it (indeed, the "QD" in the name derives from "Quick and Dirty"). Many of the analyses and optimizations we implemented, for example, were intended to determine, quickly and without too much effort, whether enough information could be obtained on typical programs to make useful optimizations possible. The entire system took one person, working on and off over a couple of years, a total of some 8 to 10 weeks to implement. Of this, about 2 weeks, i.e., 20%-25% of the time, was spent in understanding the lexical and syntax conventions of Janus and modifying O'Keefe's public-domain tokenizer and parser to handle them. Another 2 to 3 weeks were spent implementing the rest of the compiler and the runtime system. All of the various dataflow analyses and optimizations were completed

[2]The heap usage figures reported in Kliger's thesis are in thousands of heap words [6], and have been multiplied by 4 in this table to get the memory usage in kilobytes.

Program	QD-Janus (QD) (Kbytes)	FCP (Kbytes)	FCP/QD
nrev(30)	0.42	516	1228.6
pascal(200)	664.0	1792	2.7
queen_1(8)	0.08	20	250.0
prime(10000)	9.8	9468	966.1
combination	821.0	2136	2.6
deriv	0.57	768	1347.4
mastermind	29.1	8676	298.1
Geometric Mean of FCP/QD :			135.4

Table 3: Heap Usage of QD-Janus and FCP

in about 5 or 6 weeks. It can be seen from this that with a compiler written in Prolog and using Prolog as the target language, an entire system can be implemented in a surprisingly small amount of time: in our case, the "base implementation" without any optimizations took about a month to build, and the various frills added to it later took another month or so. Notice, moreover, that after these two months we had a sequential Janus system with a garbage collector, stack shifter, (somewhat limited) debugger, etc. By contrast, the FCP(:) compiler implemented by Kliger [6] was intended for research into compilation issues for committed-choice languages, which implies that performance of the generated code was a significant concern from the beginning.

The relative performance of the two systems, nevertheless, is quite surprising: the QD-Janus implementation is significantly faster than the FCP(:) implementation on most of the benchmarks tested. Obviously, this is due to the excellent performance of the Sicstus Prolog system underlying QD-Janus: we would not expect its performance to be as good if we were using a slower implementation of Prolog. But one of the points of this paper is precisely this: since there are a number of excellent high-performance implementations of Prolog around, we should take full advantage of these systems, and the huge amounts of work that have been invested in them, rather than reinvent (parts of) it ourselves at great labor and cost.

An important difference between these systems is that the FCP(:) implementation relies on time-slicing to ensure fairness, while QD-Janus does not guarantee fairness. While time-slicing incurs some runtime overhead, it does not account for all of the speed difference between QD-Janus and FCP(:). Indeed, even if we were to implement time-slicing in QD-Janus, and this were to double the execution time of every program (an estimate that is absurdly generous for all but the most trivial of programs), the resulting system would still be somewhat faster than the FCP(:) implementation described above.

Another point to note is that while QD-Janus does not implement decision trees, which appear to be the single most important optimization in the FCP(:) implementation, it would not be difficult to do so in Prolog. This is because a node in a decision tree—say, one labelled with the test 'X > Y'—can be compiled into Prolog code of the form

```
dec_tree_p(X, Y) :-
     X > Y -> dec_tree_1_p(X, Y) ; dec_tree_2_p(X, Y).
```

Because of last-call optimization, on most Prolog systems this would be compiled into a single arithmetic comparison instruction, followed by a conditional jump to dec_tree_1_p/2 and an unconditional jump to dec_tree_2_p/2. This is exactly what would be realized in a low-level implementation of decision trees (depending on the details, a low-level implementation might be able to eliminate the unconditional jump by "falling through" from the conditional jump before it, but this seems unlikely to make a huge performance difference in practice). The point here is that with careful attention to the Prolog code generated, it is possible to realize the effects of many low-level optimizations at a high level.

Finally, this discussion has been restricted to sequential implementations. We believe that our general argument—namely, that it is often better to build implementations by compiling to a high-quality implementation of another logic programming language rather than build an implementation from scratch—continues to hold for parallel systems as well, though at this point this is only conjecture, unsupported by empirical data. It would be an interesting experiment to build a parallel system using QD-Janus in conjunction with a parallelizing Prolog compiler, such as &-Prolog [4], and compare its performance with that of customized parallel implementations of committed-choice languages, such as Monaco [9]. We have not had the time to carry out experiments along these lines at this point.

6 Conclusions

While the quality of generally available Prolog implementations has improved greatly over the last decade, it appears to be unfashionable to implement logic programming languages by compiling them to Prolog: instead, researchers spend a great deal of time building low-level implementations from scratch. We argue in this paper that their time would be better spent in compiling to Prolog, using a good underlying Prolog system, and using the time thus saved to implement good program analyses and optimizations to overcome the overheads associated with a straightforward compilation to Prolog. In support of our arguments, we compare two sequential implementations of committed choice languages: QD-Janus, which translates to Prolog, and FCP(:), which compiles to a low-level byte-coded instruction set. Even though QD-Janus took significantly less time to implement, its performance can be seen to be significantly better than that of FCP(:).

Acknowledgements: This work was supported in part by the National Science Foundation under grant number CCR-9123520.

References

[1] M. Carlsson and J. Widen, *SICStus Prolog User's Manual*, Swedish Institute of Computer Science, Oct. 1988.

[2] S. K. Debray, "QD-Janus: A Sequential Implementation of Janus in Prolog", *Software—Practice and Experience* (to appear).

[3] D. Gudeman, K. De Bosschere, and S. K. Debray, "jc : An Efficient and Portable Implementation of Janus", *Proc. Joint International Conference and Symposium on Logic Programming*, Washington DC, Nov. 1992. MIT Press.

[4] M. Hermenegildo and K. Greene, "The &-Prolog System: Exploiting Independent And-Parallelism", *New Generation Computing*, to appear.

[5] A. Houri and E. Shapiro, "A Sequential Abstract Machine for Flat Concurrent Prolog", in *Concurrent Prolog: Collected Papers*, vol. 2, ed. E. Shapiro, pp. 513-574. MIT Press, 1987.

[6] S. Kliger, *Compiling Concurrent Logic Programming Languages*, Ph.D. Thesis, The Weizmann Institute of Science, Rehovot, Israel, Oct. 1992.

[7] V. Saraswat, K. Kahn, and J. Levy, "Janus: A step towards distributed constraint programming", in *Proc. 1990 North American Conference on Logic Programming*, Austin, TX, Oct. 1990, pp. 431-446. MIT Press.

[8] E. Tick, *Parallel Logic Programming*, MIT Press, Cambridge, 1992.

[9] E. Tick and C. Bannerjee, "Performance Evaluation of Monaco Compiler and Runtime Kernel", *Proc. 1993 International Conference on Logic Programming*, Budapest, June 1993.

[10] K. Ueda, "Guarded Horn Clauses", in *Concurrent Prolog: Collected Papers*, vol. 1, ed. E. Shapiro, pp. 140-156, 1987. MIT Press.

A List of Benchmarks

`nrev(30)` – Naive reverse of a list of length 30.

`hanoi(17)` – The Towers of Hanoi program, translated from [5].

`e` – 10,000 iterations of a program to compute the value of the constant $e = 2.71828\ldots$ to a tolerance of 10^{-6} by summing the series $\sum_{n \geq 1} \frac{1}{n!}$.

`qsort`– quicksort of a list of 50 integers.

`pi` – a program to compute the value of π to a tolerance of 10^{-6} using the identity $\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots$.

`dnf(50)` – A program for the "Dutch national Flags" problem, by V. Saraswat.

`pascal(`n`)` – A program to compute the n^{th} row of Pascal's Triangle [8]. The numbers given are for $n = 200$.

`queen(`n`)` – A program to find all solutions to the n-queens problem [8]. The numbers given are for $n = 8$.

`queen_1(`n`)` – A program to find one solution to the n-queens problem, by S. Kliger. The numbers given are for $n = 8$.

`prime(`n`)` – A program to compute the primes upto n using the Sieve of Eratosthenes [8]. The numbers given are for $n = 10,000$.

`tak` – The Takeuchi benchmark. The numbers given are for `tak(18,12,6,_)`.

`combination` – A combinations program by E. Tick [8]. The numbers given are for `combo(6,[1,2,3,4,5,6],_)`.

`deriv` – A symbolic differentiation program by D. H. D. Warren.

`mastermind` – A mastermind program using naive generate and test [8]. The numbers given are for `go(3,3,_)`.

`nand` – A NAND-gate circuit designer using pipeline filter [8].

Concurrent Logic Programs *a la Mode*

E. Tick, B. Massey, F. Rakoczi, and P. Tulayathun
Dept. of Computer Science
University of Oregon
Eugene, OR 97403, USA
tick@cs.uoregon.edu

Abstract

This paper describes and compares two compile-time analysis algorithms for deriving the *path modes* of a moded concurrent logic program. A path describes a subterm of a procedure argument. Deriving all path modes is a way to perform certain valuable optimizations, such as sequentialization of too-fine concurrent tasks, and scheduling to reduce suspension. We describe our own variation of Ueda and Morita's original constraint propagation scheme, which includes our novel circular unification algorithm. We also describe an alternative method of finite domain analysis that we developed. The two methods are critiqued and we show the relationship between them.

1 Introduction

Mode information has been shown to be quite useful in the efficient compilation of logic programming languages. In general, possible variable modes are "in" (meaning that a variable will not be bound by the current goal) and "out" (meaning that a variable will not be bound outside of the current goal). Traditionally, mode information facilitates the strength reduction of unification operators into matches and assignments. There are numerous methods for automatic derivation of mode information from logic programs, e.g., [1, 2, 3, 7]. In concurrent logic programs, the logic variable is overloaded to perform synchronization. Mode information can thus be used to optimize code generated for argument matching.

We are interested in concurrent logic programs in the FCP(:, |) language family [8] that include ask (passive unification for input matching) and tell (active unification for exporting bindings) guards. This represents a broad class of flat committed-choice programs. Figure 1 shows a sample program used throughout the paper to illustrate the analysis techniques discussed herein. A sample query would be: ?- $q([2,1,3], Y, [])$, returning $Y = [3,2,1]$.

Ueda and Morita [11] proposed a mode analysis scheme for such programs, based on the representation of procedure paths and their relationships as rooted graphs ("rational trees"). Unification over rational trees combines the mode information obtainable from the various procedures. For example, in a procedure that manipulates a list data stream, we might know that the mode of the car of the list (that is the current message) is the same mode as the cadr (second message), caddr (third message), etc. This potentially infinite set of "paths" is represented as a concise graph. Furthermore, a caller of this procedure may constrain the car to be input mode. By unifying the caller and callee path graphs, modes can be propagated. The analysis is restricted to "moded" flat committed-choice logic programs. These are programs in which the mode of each path in a program is constant, rather than a function of the occurrences of the path. This is not regarded as a major drawback, since most non-moded flat committed-choice logic programs may be transformed to moded form in a straightforward fashion.

$$
\begin{array}{l}
q(T_0, Y_0, Z_0) \text{ :- } T_0 \; = \; [] \; : \; Y_0 \; =_0 \; Z_0. \\
q(T_1, Y_1, Z_1) \text{ :- } T_1 \; = \; [X_1 \mid Xs_1] \; : \; U_1 \; = \; [X_1 \mid Vs_1] \; \mid \\
\qquad s(Xs_1, X_1, L_1, G_1), \\
\qquad q(L_1, Y_1, U_1), \\
\qquad q(G_1, Vs_1, Z_1). \\
\\
s(T_2, \_ , L_2, G_2) \text{ :- } T_2 \; = \; [] \; : \; L_2 \; =_1 \; [], G_2 \; =_2 \; []. \\
s(T_3, Y_3, L_3, G_3) \text{ :- } T_3 \; = \; [\; X_3 \mid Xs_3 \;], X_3 < Y_3 \; : \\
\qquad G_3 \; =_3 \; [\; X_3 \mid Ws_3 \;] \; \mid \\
\qquad s(Xs_3, Y_3, L_3, Ws_3). \\
s(T_4, Y_4, L_4, G_4) \text{ :- } T_4 \; = \; [\; X_4 \mid Xs_4 \;], X_4 \geq Y_4 \; : \\
\qquad L_4 \; =_4 \; [\; X_4 \mid Ws_4 \;] \; \mid \\
\qquad s(Xs_4, Y_4, Ws_4, G_4).
\end{array}
$$

Figure 1: Quicksort FCP(:, |) Program: Normalized Form (Clauses 1–5)

As an alternative algorithm, we developed a projection of the previous work to a *finite domain of paths*. This has the nice property of obviating the need for unification over circular terms, thereby gaining some efficiency. We represent the relationships of a finite set of paths in such a way that all mode information directly available about this set of paths in a program may be efficiently derived. Again, moded programs are required.

The purpose of this paper is to describe and compare both algorithms in detail. As far as we know, this is the first document describing practical implementations for these mode analysis schemes. Section 2 describes our implementation of the constraint propagation algorithm. Section 3 describes our finite domain algorithm. In both sections we present examples to illustrate the concepts. The algorithms are compared in Section 4 and conclusions are summarized in Section 5.

2 Constraint Propagation Algorithm

Ueda and Morita's notion of "path" is adopted as follows: A path p "derives" a subterm s within a term t (written $p(t) \vdash s$) iff for some predicate f and some functors $a, b, \ldots$ the subterm denoted by descending into t along the sequence $\{< f, i >, < a, j >, < b, k >, \ldots\}$ (where $< f, i >$ is the i^{th} argument of the functor f) is s. A path thus corresponds to a descent through the structure of some object being passed as an argument to a function call. f is referred to as the "principal functor" of p. A program is "moded" if the modes of all possible paths in the program are consistent, where each path may have one of two modes: *in* or *out* (for a precise definition, see Ueda and Morita [11]).

> *Example*: Considering the quicksort example, the cadr of the first argument of procedure $q/3$ has an input mode specified as: $m(\{< q/3, 1 >, < `.\text{'}/2, 2 >, < `.\text{'}/2, 1 >\})$ $= in$. □

Both analyses presented in this paper exploit the rules outlined by Ueda and Morita. Their axioms are clarified and reformulated in Figure 2 ($m(p)$ means the mode of path p). In the constraint propagation algorithm, a graph is constructed representing the entire program. Hierarchically, we compute and combine graphs at three levels: modules, procedures, and clauses.

§1. For some path p in a clause, $m(p) = \text{in}$, if either

1. p leads to a non-variable in the head or body, or
2. p leads to a variable which occurs more than once in the head, or
3. p leads to a variable which also occurs in the guard at path p_h and $m(p_h) = \text{in}$

§2. Two arguments of a unification body goal have opposite modes, for all possible p, or more formally: $\{\forall p\ m(<=, 1 > p) \neq m(<=, 2 > p)\}$.

§3. If there are exactly two "occurrences," we have two possibilities:

1. If both occurrences are in the body, the modes of their paths are inverted.
2. If there is one (or more) occurrence in the head and one in the body, the modes of their paths are the same.

§4. If there are more than two "occurrences" of a shared variable (i.e., at least two occurrences in the body), the situation is even more complex:

1. If the body contains more than two occurrences of the shared variable and the head has no occurrences, then one of the modes is 'out,' and the others are 'in.' This means that one of the occurrences is designated as the producer of this variable.
2. If the head contains one (or more) occurrences of the shared variable (so the body has two or more occurrences), then the modes are as follows:
   (a) If the mode of the head occurrence is 'in,' the modes of all body occurrences are 'in' as well.
   (b) If the mode of the head occurrence is 'out,' then *one* of the body occurrences is 'out,' and the other body occurrences are 'in.'

Figure 2: Ueda and Morita's Mode Derivation Axioms (for Moded FGHC)

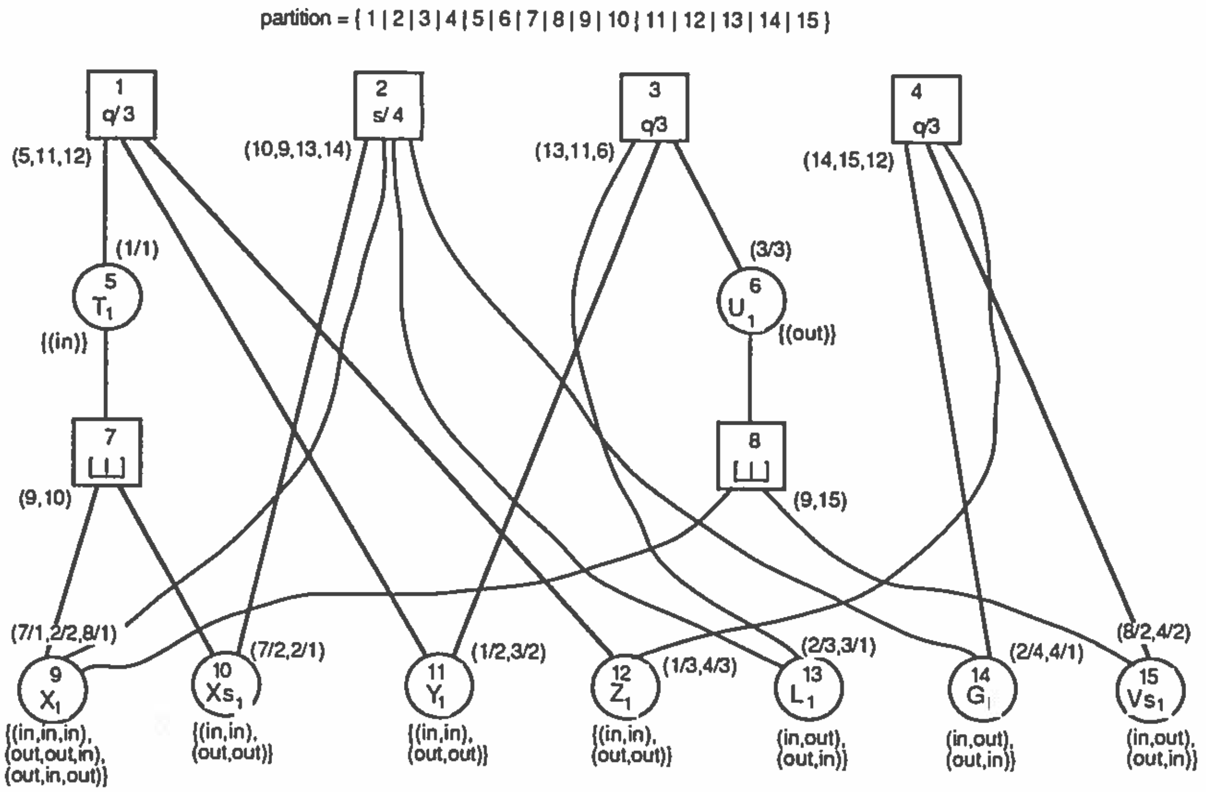

Figure 3: Initial Graph of Procedure $q/3$ (Clause 2), After Phases I-II

Top-down, we envision first constructing such a graph for each module of the program, and then connecting the graphs via imported/exported procedures. Within a module, we utilize this same strategy of first constructing procedure graphs ("local" analysis) and then combining graphs via inter-procedure call sites. Within a procedure, we first construct clause graphs, and then combine them via the heads.

Graph combination is formally unification, as described in Section 2.3. The methodology is guaranteed to terminate because graph unification can only reduce the structural complexity. Termination occurs when no further reduction is possible. However, the scheme is not guaranteed to terminate in the *minimum* number of graph reductions. In the future we intend to experiment with heuristics to guide graph reduction to quicker termination (see Section 2.3).

2.1 Data Structures

A program graph is a directed, multi-rooted, (possibly) cyclic graph composed of two types of nodes. To clearly illustrate the following definitions, Figure 3 presents a portion of the quicksort program graph. This portion corresponds to the initial graph for procedure $q/3$ (clause 2).

> *Definition:* A *structure node* (drawn as a square) represents a functor with zero or more exit-ports corresponding to the functor's arity. If the node corresponds to a procedure name (for clause heads and body goals), there are no associated entry-ports (i.e., it is a root). If the node corresponds to a data structure, there is a single entry-port linked to a *variable node* unified with that term. A structure node contains the following information: a unique identifier, functor, and arity. □

Example: Consider node 8 in Figure 3 holding the list functor of arity two. The entry-port connects to variable node 6 (U_1) and the two exit-ports connect to variable nodes 9 and 15 (X_1 and Vs_1 respectively). □

Definition: A *variable node* (drawn as a circle) represents a subset S of (unified) variables in a clause. Intuitively we think of these variables as aliases, and upon initial construction of the graph, S is a singleton (i.e., each unique variable in the clause has its own variable node initially). A node contains $k \geq 1$ entry-ports and $j \geq 0$ exit-ports, upon which directed edges are incident. A unique entry-port corresponds to each clause instance of each variable in S. An exit-port corresponds to a possible unification of the variable(s) to a term (exit-ports connect to *structure nodes*).[1]

A variable node contains the following information: a unique identifier and a *mode set* M. An element of M is a vector of length k containing self-consistent modes for the variable *instances* of S. To facilitate the implementation, each entry-port has a *name*: the identifier and exit-port number of its source node. Elements of M are *alternative* mode interpretations of the program. Initially M is computed by Ueda and Morita's rules.[2] Intuitively, graph reduction results in removing elements from M as more constraints are applied by local and global unifications. A fully-reduced graph, for a fully-moded program, has a singleton M in each variable node. □

Example: Consider node 15 in Figure 3 with entry-ports named (8/2,4/2). Node 15 holds variable Vs_1 with mode set {(in,out),(out,in)}. This set derives from rule §3.1 in Figure 2. □

In general, initial graphs, like that in Figure 3, will be multi-rooted directed acyclic graphs. The initial roots correspond to clause head functors, body goal functors, and body unification operators. In addition to the program graph, a *partitioned node set* is kept. Initially, each node is a singleton member of its own partition (disjoint set).

The mode analysis consists of three phases: I) creating a normalized form and initial graph; II) removing unification operators from the graph, and III) reducing the graph to a minimal form. These are described in the following sections.

2.2 Graph Creation

Phase I converts a flat committed-choice program into normalized form, an example of which is shown in Figure 1. Normalized form ensures that all variables are renamed apart among clauses within the same procedure and that each clause is flattened, i.e., all head structures and body goal structures are moved into ask and tell guards, respectively. An initial program graph is created from the normalized form, including mode sets for each variable node. The normalization implies a graph invariant that structure nodes cannot point to structure nodes (and variable nodes cannot point to variable nodes): the graph is a "layered network" (with cycles).

There is a minor trick required to deal with certain unification operators in phase I. Naively, a goal $X = f(Y)$ would result in a structure node ($=/2$) pointing to another structure node ($f/1$).

[1]An invariant preserved by unification is that no two child (structure) nodes of the same parent (variable) node can have the same functor/arity (see discussion in Section 2.3).

[2]The size of M increases with the complexity of the rules, e.g., rule §4 (Figure 2) can produce several vectors. By explicitly enumerating all possible modes initially, we simplify the analysis immeasurably.

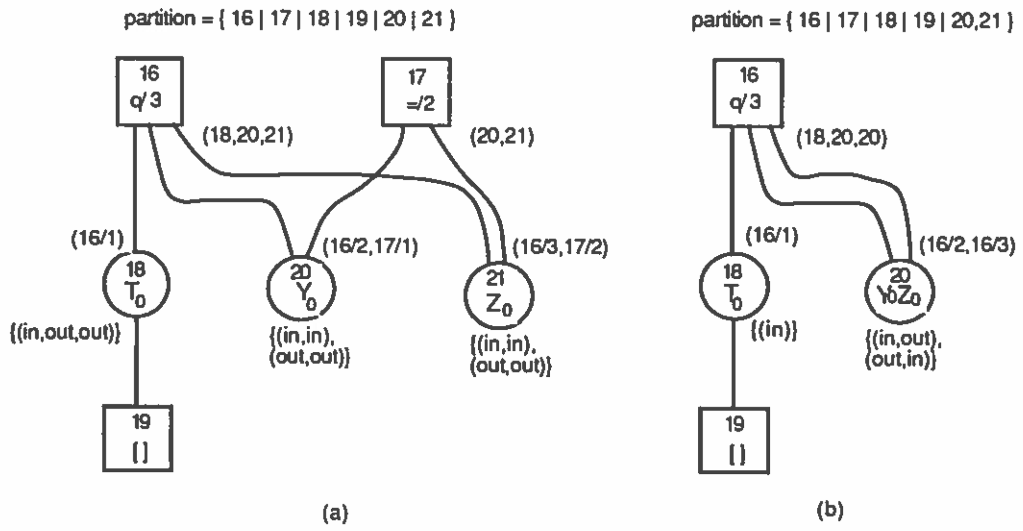

Figure 4: Builtin Tell Unification of Quicksort ($q/3$, clause 1)

To avoid this exception thereby facilitating the next phase, we introduce a *dummy* variable node as an intermediary between these two structure nodes. This is effectively a nameless placeholder that will be removed in phase II.

In phase II we remove all root nodes corresponding to builtin predicates. Intuitively, these predicates have fixed modes and thus their reduction acts as the boundary conditions anchoring subsequent constraint propagation by unification. Here we describe only reduction of (both active and passive) unification operators because they are the most frequent.

First consider a (passive) unify operator in the ask guard (all head unifications have been normalized into the guard). The structure node corresponding to the operator has two exit-ports indicating the operands of the unification. The two variable nodes attached to these exit-ports can be *merged* as follows. A cross-product of two mode sets is taken, resulting in a set of vector pairs. We retain those pairs that have *in* mode for both entry-ports arriving from the unification node, and discard all other pairs. The two vectors in each remaining pair are concatenated, forming a new mode set. The two variable nodes are fused into one node containing the new mode set just computed. The entry-ports and modes corresponding to the unify operator are removed.

Now consider a tell (active) unification goal. A similar merging operation is performed, keeping only those vector pairs that have *opposite* modes at the positions corresponding to the entry-ports arriving from the unification node (rule §2 in Figure 2). After phase II, builtin predicate nodes are no longer present in the graph. The resulting graph contains roots named only by clause heads and user-defined body goals.

Example: Phase II analysis of $q/3$ clause 1 is illustrated in Figure 4. The modes of the tell unification operator must be opposite. Thus we remove the unification operator node, combining its operands, resulting in the graph shown in Figure 4b. That concludes the analysis for clause 1 because there are no body goals.

Another example is shown in Figure 3, where the topology of node groups {5,7,9,10} and {6,8,9,15} result from the introduction of dummy nodes in phase I, followed by phase II analysis. □

2.3 Abstract Unification

Phase III of the analysis is to reduce the graph to a *minimal form* by successive node unifications. We perform "local" reduction first by collapsing recursive call sites (by abstract unification) with associated clause heads. Next we perform "global" reduction by unifying root nodes from different procedures. The abstract unification algorithm is the same, however, for any two (node) arguments, local or global.

Figure 5 gives the graph unification algorithm. We use the notation that a variable node v has the fields: *v.in* (vector of entry-ports, each of the form *id/index*, where *id* is the parent's node identifier and *index* is the parent's exit-port index), *v.out* (set of exit-ports), and *v.modes* (set of mode vectors). A structure node s has the fields: *s.out* (vector of exit-ports), and *s.fun* (functor/arity).

Unification is invoked as *unify(a,b)* of two nodes a and b (necessarily root structure nodes). The result is either failure, or success and a new graph (including the node partitioning) that represents the most general unification (mgu) of the two operands. Implied data structures used by the algorithm include the graph, the disjoint sets (i.e., node partitioning), and a *mark table* associated with pairs of nodes.[3]

Procedures *sunify* (structure node unification) and *vunify* (variable node unification) follow recursive descents. Initially all marks are cleared (1). Circular structures that represent infinite paths are handled properly by *marking* node pairs at first visit (2). If a given node pair has been previously marked, revisiting them immediately succeeds. Note that we mark *pairs* instead of individual nodes to handle the case of unifying cyclic terms of unequal periodicity.

> *Example*: Consider two graphs representing $m(\{<\text{`.'}/2, 2>, p\}) = m(p)$ and $m(\{<\text{`.'}/2, 2>, <\text{`.'}/2, 2>, p\}) = m(p)$. The former has a 1-period cycle and the latter a 2-period cycle. Suppose that node 1 corresponds to the structure node in the shorter cycle, and nodes 2 and 3 are in the longer cycle. Unifying the roots, we first mark pair {1,2}, then mark {1,3}, then terminate when pair {1,2} is found to already be marked. □

Two important operations for the disjoint sets data structure are *union(x,y)* and *find_set(x)*. Function *union(x,y)* unites two disjoint sets, where x belongs to the first disjoint set and y belongs to the second disjoint set. Procedure *union* returns the *canonical name* of the partition (3), i.e., the least identifier of the nodes. This facilitates reusing graph nodes while rebuilding the graph.[4] Function *find_set(x)* returns the canonical name of the disjoint set containing x.

The major complexity in the algorithm is in procedure *vunify*, where the abstract unification must merge the modes of the two argument nodes. First, mode vectors that are contradictory are discarded (4). If all mode vectors are contradictory then a mode error has occurred and unification fails. Otherwise redundant modes are removed and the two mode vectors are concatenated (5). Next we create the entry-port identifiers associated with the new mode vector (6). Lastly, children of the argument nodes that share equal functor/arity must be recursively unified (8). The exit-port identifiers consist of a single exit-port for each pair of children unified,

[3]Note that the new graph returned by *unify* has the same number of nodes as the original graph. However, for practical purposes *either* of the input roots can be discarded, turning unneeded nodes into garbage. The key point is that the graph must always be traversed according to the node partitions, so as the partitions grow larger, effectively the graph shrinks. To ensure that the graph shrinks in practice, unneeded nodes must be explicitly removed from the graph, an operation that we do not detail here.

[4]Optionally, the canonical name can be defined to be a new identifier, avoiding node reuse. This might facilitate searching for the modes of top-level variables, e.g., for data-dependency analysis.

```
unify($a$, $b$) {
        $\forall a, b$ clear mark($\{a,b\}$)                                    (1)
        sunify($a$, $b$)
}

sunify($a$, $b$) {
        if $a.fun \neq b.fun$
                return(failure)
        if mark($\{a,b\}$) clear then {
                set mark($\{a,b\}$)                                             (2)
                $\forall k \in [1, arity(a)]$ {
                        if vunify($a/k$, $a$.out[$k$], $b/k$, $b$.out[$k$]) failure then
                                return(failure)
                }
        return(success)
  }   }

vunify($i$, $a_v$, $j$, $b_v$) {
        $a$ = find_set($a_v$)
        $b$ = find_set($b_v$)
        if mark($\{a,b\}$) clear then {
                set mark($\{a,b\}$)
                $c$ = union($a$,$b$)                                            (3)
                — compute the compatible mode set
                define $u$ s.t. $a.in[u] = i$
                          $v$ s.t. $b.in[v] = j$
                $P = a.modes \times b.modes$
                $P' = \{(s,t) \in P \mid s[u] = t[v]\}$                         (4)
                if ($P'$ empty) then
                        return(failure)
                $c.modes = \{ s \,||\, (m_1, m_2, ..., m_{v-1}, m_{v+1}, ..., m_k) \mid$
                                $(s,t) \in P',\ t = (m_1, m_2, ..., m_v, ...m_k) \}$      (5)
                — compute the entry-port identifiers
                let $(p_1, p_2, ..., p_v, ..., p_k) = b.in$
                $c.in = a.in \,||\, (p_1, p_2, ..., p_{v-1}, p_{v+1}, ..., m_k)$        (6)
                — compute the exit-ports identifiers
                $U = a.out \cup b.out$
                $F_a = \{s.fun \mid s \in a.out\}$
                $F_b = \{s.fun \mid s \in b.out\}$
                $I = \{s \mid fun \in F_a \cap F_b,\ s.fun \in U\}$
                $c.out = a.out \cup I$                                          (7)
                — unify children with the same functor/arity
                $\forall (x,y) \mid x \in a.out, y \in b.out$ {
                        if $x.fun = y.fun$
                                if sunify($x$, $y$) failure then                (8)
                                        return(failure)
                }
        return(success)
  }   }
```

Figure 5: One-Pass Graph Unification Algorithm

included with exit-ports for all children for which unification does not take place (7). Intuitively, a variable node forms OR-branches with its children, whereas a structure node forms AND-branches with its children. In other words, the least-upper-bound (lub) of the abstract unification semantics at a variable node is a union of the structures that potentially concretely unify with the variable node.

Local analysis continues with unification of roots among clauses composing a single procedure definition. Local analysis terminates when no two roots have the same functor and arity within a procedure graph. This is perhaps not a time-optimal strategy, but was selected, in our prototype, for its simplicity. Analyzing non-recursive clauses first, and then unifying these clause-head roots with recursive call sites in other clauses, is expected to terminate faster. The rationale is similar to quickly reaching a fix point in abstract interpretation by approximating recursive calls with their corresponding non-recursive clause input/output relationships.

After local analysis we perform global analysis which unifies roots among different procedure graphs. Global analysis terminates when each root in the entire program is unique.

Example: Reconsider the quicksort program (Figure 1) to illustrate the unification algorithm. First, we consider the second clause of $q/3$ and construct the graph in Figure 3. Each node of the graph is assigned a unique identifier. Then entry-ports, exit-ports, and modes are shown.

We start the local analysis for this graph by unifying the node 1 with node 3. The result of this unification is shown in Figure 6.[5] That is we unify all three corresponding argument positions of both nodes. The first pair of arguments unified is node 5 (entry-port 1) with node 13 (entry-port 2). Initially these nodes belong to their own partitions, so *find_set* returns 5 and 13. The *union* returns the canonical name 5, the minimum of the two. Thus node 5 is overwritten with the new (fused) variable node.

For example, all possible modes of node 5 entry-ports are {(in)} and the mode set of node 13 is {(in,out),(out,in)}. The cross-product set is {(in)} $\times$ {(in,out),(out,in)} = {(in,in,out), (in,out,in)}. Element {(in,in,out)} is discarded because the first and the third positions (instances, or entry-ports, of the unified argument) do not match. Thus, the legal cross-product set is {(in,out,in)}. This is reduced to the mode set {(in,out)} and entry-port vector (1/1,2/3), removing redundancies. Note that by fusing the entry-port vectors, the edge from node 2 to node 13 has effectively been rerouted to fused node 5.

Exit-ports need to be traversed in a recursive descent. In this case, the exit-port of node 5 is simply inherited by the new node since node 13 has no children. Then the second and third arguments of nodes 1 and 3 are unified, resulting in Figure 6. Node 1 and node 4 are then unified, resulting in the graph shown in Figure 7. This is the final local unification possible for clause 2 since all roots now have unique functors.

We now do inter-clause analysis of $q/3$, unifying the first (root) nodes from the graphs represented in Figures 7 and 4b, resulting in the graph of Figure 8. This represents the most information that can be derived from $q/3$ alone. Note that the modes of X_1 and Xs_1 are still ambiguous. In normal operation, we would locally derive the modes for $s/4$, and then do global analysis, unifying the $s/4$ graph with node 2 of

[5]By convention we remove nodes 3, 12, and 13 from the illustration to clarify that they represent redundant information. In the implementation, these nodes might be kept in the graph if space was not a critical resource.

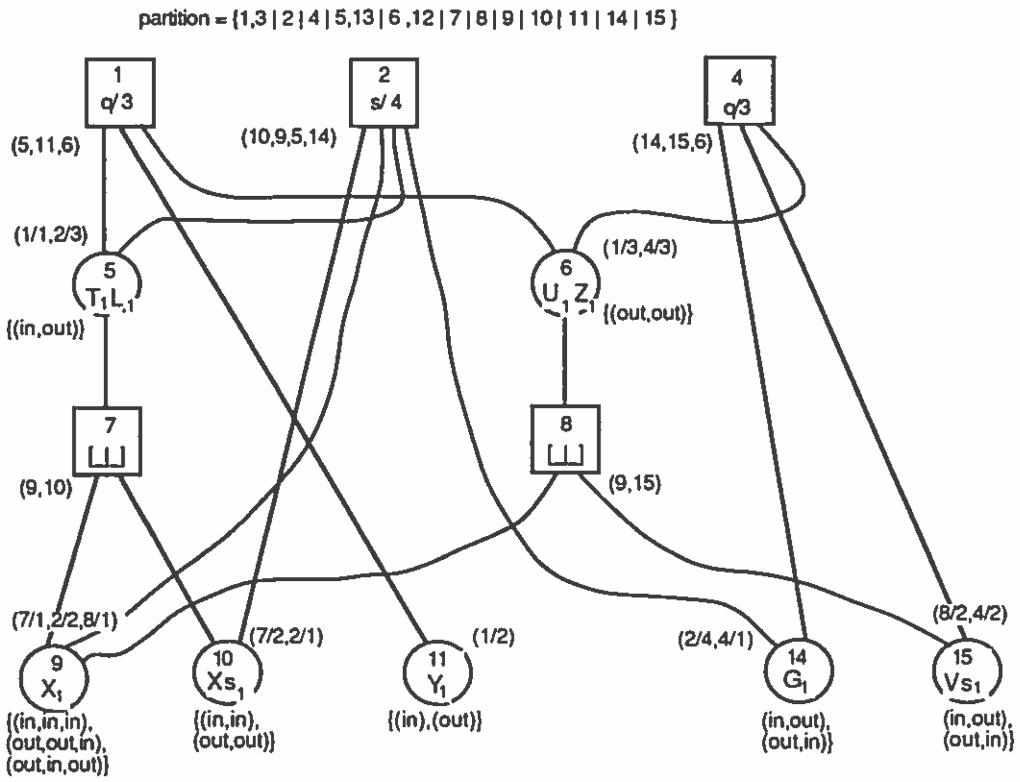

Figure 6: First Local Unification of $q/3$ (clause 2)

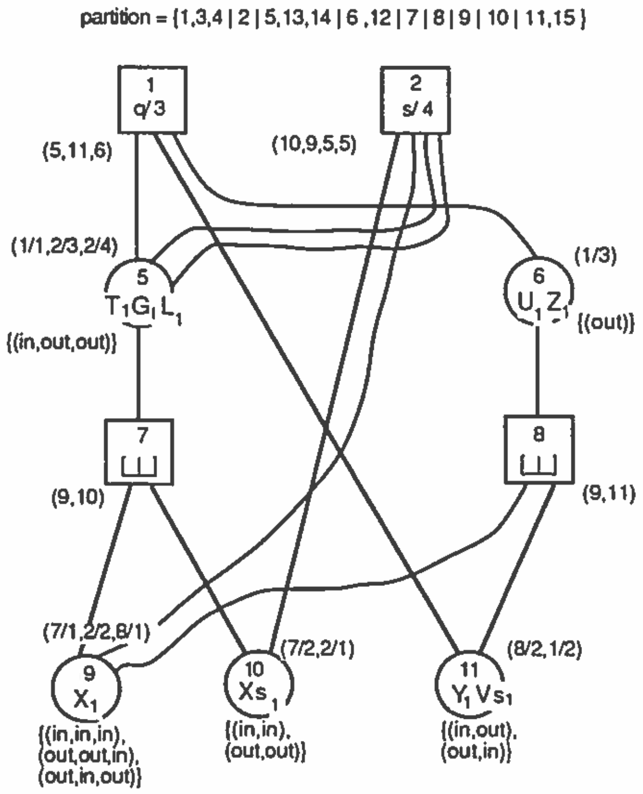

Figure 7: Second Local Unification of $q/3$ (clause 2)

partition = {1,3,4,16 | 2 | 5,13,14,18 | 6 ,11,12,15,20,21 | 7 | 8 | 9 | 10 | 19 }

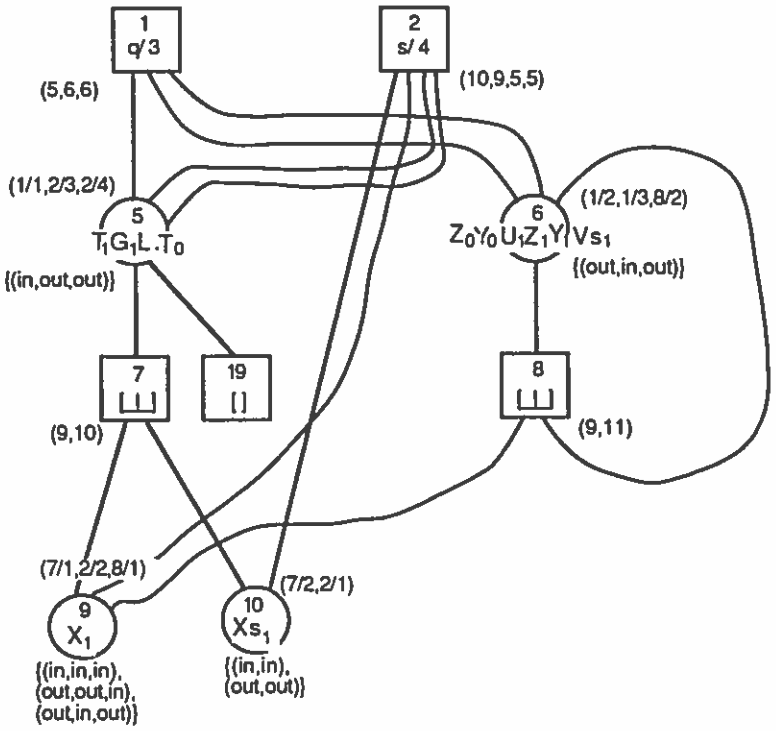

Figure 8: Global Unification of Quicksort ($q/3$)

partition = {1,3,4,16 | 2,24,27,38,48,51 | 5,10,13,14,18,30,31,37,
43,44,55,56,61 | 6 ,11,12,15,20,21 | 7 | 8 | 9 | 19 }

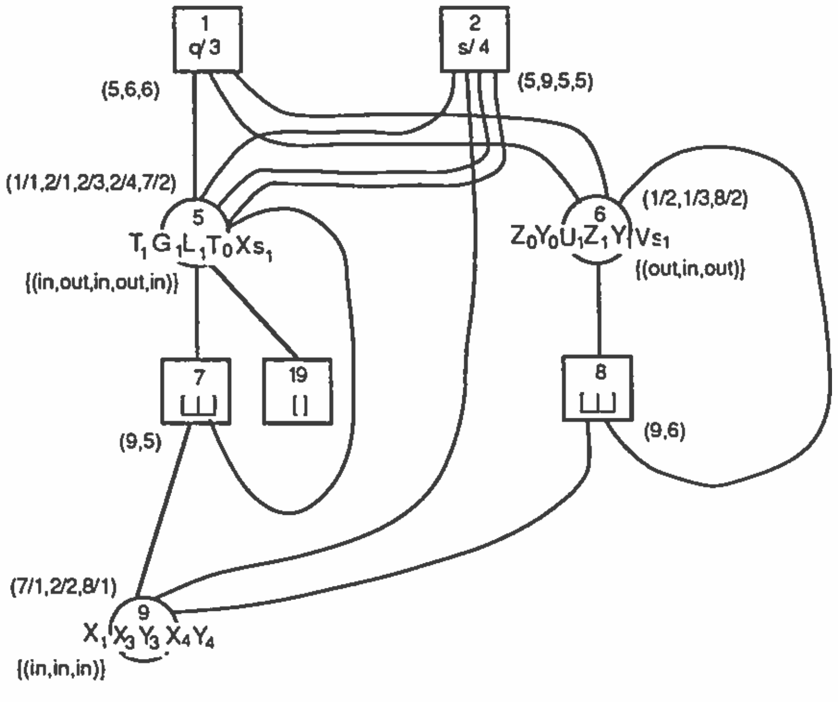

Figure 9: Minimized Graph of Quicksort

the $q/3$ graph. We show only the final resultant graph with all mode ambiguities removed in Figure 9. □

3 Finite Domain Analysis

The first stage of the alternative algorithm generates a finite set of paths whose modes are to be considered. Only "interesting" paths are generated in the first stage of our algorithm: effectively those paths locally derived from the syntactic structure of the procedures. There are three classes of interesting paths. The first class consists of paths that directly derive a named variable in the head, guard, or body of some clause. All such paths can be generated by a simple sequential scan of all heads, guards, and body goals of the program.

The second class consists of paths which derive a variable v in some clause, where a proper path through the opposite side of a unification with v derives a variable v'. More formally, consider a unification operator $v = t$ where v is a variable and t is some term other than a variable or ground term. Let v' be a variable appearing in t at path q, i.e., $q(t) \vdash v'$. Then if p is a path deriving v (by which condition p is also interesting), then the concatenated path $p \cdot q$ is also an interesting path. All paths in this second class may be generated by repeated sequential scanning of all unification goals until no new interesting paths are discovered. The necessity for repeated scans is illustrated by such clauses as

$$a(X, Z) :\!- \; Y = c(X), \;\; Z = b(Y).$$

where the interesting path $\{< a, 2 >, < b, 1 >, < c, 1 >\}$ given by the first unification body goal will not be generated until the interesting path $\{< a, 2 >, < b, 1 >\}$ in the second unification body goal is generated. Such repeated scans should occur infrequently in practice. In any case not more than a few scans are necessary — no greater number than the syntactic nesting depth of expressions containing unification operators.

The third class of interesting paths is generated by noting that if a path starting on the right-hand side of a unification body goal (i.e., a path of the form $\{<=, 2 >\}\cdot s$) is interesting, then so is the corresponding path starting on the left-hand side of that unification (i.e., $\{<=, 1 >\}\cdot s$).

In general, all interesting paths of a program are generated in a few sequential passes. The 39 interesting paths of quicksort, shown in Table 1, are generated in two passes. Note the correlation between these paths and the infinite paths represented in Figure 9. The interesting paths could be generated from a depth-one traversal of the graph, except for $\{< q/3, 2 >, < ., 1 >\}$ and $\{< q/3, 2 >, < ., 2 >\}$. These two paths are "hidden" because they cannot be derived from clause 2 of $q/3$ alone. However, the set of interesting paths produced is sufficient to mode the program in the sense of assigning an unambiguous mode to all syntactic variables. An important question is whether a finite set of paths represents a *minimal and complete* set of paths for the mode analysis in general. Unfortunately, as discussed below, there is good reason to believe that some fundamentally important paths may not be generated.

Once we have generated a set of interesting paths, our algorithm proceeds by simply noting the modes of paths, first directly, and then by examining relationships between paths. There are essentially four different stages in the algorithm:

1. Assert absolute modes for some paths.

2. Assert that all paths on opposite sides of a "tell" unification have opposite modes.

input		output	
user	builtin	user	builtin
$\{< s/4,1 >\}$	$\{<`<'/2,2 >\}$	$\{< q/3,2 >\}$	$\{<=_0,1 >\}$
$\{< s/4,2 >\}$	$\{<`<'/2,1 >\}$	$\{< s/4,3 >\}$	$\{<=_1,1 >\}$
$\{< q/3,1 >\}$	$\{<`\geq'/2,2 >\}$	$\{< s/4,4 >\}$	$\{<=_2,1 >\}$
$\{< q/3,3 >\}$	$\{<`\geq'/2,1 >\}$	$\{< s/4,3 >,< .,1 >\}$	$\{<=_3,1 >\}$
$\{< q/3,1 >,< .,1 >\}$	$\{<=_0,2 >\}$	$\{< s/4,3 >,< .,2 >\}$	$\{<=_4,1 >\}$
$\{< q/3,1 >,< .,2 >\}$	$\{<=_1,2 >\}$	$\{< s/4,4 >,< .,1 >\}$	$\{<=_3,1 >,< .,1 >\}$
$\{< q/3,3 >,< .,1 >\}$	$\{<=_2,2 >\}$	$\{< s/4,4 >,< .,2 >\}$	$\{<=_3,1 >,< .,2 >\}$
$\{< q/3,3 >,< .,2 >\}$	$\{<=_3,2 >\}$		$\{<=_4,1 >,< .,1 >\}$
$\{< s/4,1 >,< .,1 >\}$	$\{<=_4,2 >\}$		$\{<=_4,1 >,< .,2 >\}$
$\{< s/4,1 >,< .,2 >\}$	$\{<=_3,2 >,< .,1 >\}$		
	$\{<=_3,2 >,< .,2 >\}$		
	$\{<=_4,2 >,< .,1 >\}$		
	$\{<=_4,2 >,< .,2 >\}$		

Table 1: Interesting Paths of Quicksort (23 input, 16 output)

3. Proceed sequentially through the variables derivable from interesting paths, asserting all binary relations between paths.

4. Repeatedly consider multiway relations (rule §4 Figure 2) asserted by the clauses.

The first three stages have linear complexity. The multiway analysis is exponential in the number of variables, but by the time it is actually performed, most alternatives contradict the known modes, and thus are not explored. We found multiway analysis contributed only 2–7% of total analysis execution time in simple programs, and 11–20% in complex programs [4].

Some important practical and theoretical issues are raised by this algorithm. Some of these issues include the consistency, completeness, and safety of the mode analysis. It is not difficult to prove that the mode analysis algorithm is consistent in the sense that if, at some point in the analysis, path p is shown to have mode m, and if some subset of the interesting paths implies that p does not have mode m, then the algorithm will derive and report this contradiction.

The major barrier to the consistency of this algorithm is somewhat subtle: the non-modedness of a program may not be detectable if the analysis uses the wrong set of paths! This leads directly to a reasonable definition of a *complete* set of paths. A set of paths generated for a program is *complete* iff the existence of a consistent moding for the set of paths implies that the program is fully-moded. (We say that a program is "fully-moded" if the modes of *all* paths are known, and "moded" if the modes of *some* paths are known).

Thus, the infinite set of all possible paths is a complete set; however, we are interested in *finite* complete sets and in particular in a *minimal* complete set of paths for the program. Our path generation algorithm is incomplete; because of this incompleteness in path generation, the mode analysis algorithm we constructed is *unsafe*. It is a consequence of the incomplete set of generated paths that even if the program contains information about the mode of a path, that information may not be derived by the mode analysis algorithm. Thus, the analysis is unsafe in the sense the compiler may not detect mode contradictions in erroneous (not fully-moded) programs, and thereby produce erroneous mode information for programs that should be rejected altogether. Nonetheless, most generated paths in typical programs are moded by our analysis, and if the program being analyzed is known to be moded, all modes derived are correct. Thus, our mode analysis algorithm is a practical tool for many compiler optimizations.

4 Critique

We believe that our algorithmic specification and implementation of the *constraint propagation method* has contributed in the following areas:

- The algorithm to be used is specified precisely. In particular, the data structures used by the algorithm and the method of unification of subtrees are specified in sufficient detail that the algorithms for encoding the program and determining its modes are always well-defined.

- Our implementation of the algorithm is undergoing benchmarking and appears to behave correctly.

- The algorithm seems amenable to a proof of completeness and correctness via structural induction, as it is specified in a relatively denotational style.

Nonetheless, there is still work to be done. Most importantly, a formal proof of the correctness of the algorithm should be given, and the implementation should be tested with a variety of real-sized programs.

The *finite domain method* avoids circular unification by sacrificing completeness. Our initial belief was that real programs have very short paths, so that the construction of "interesting" paths would cover most programs. From analysis of small benchmarks, we note that programs with deep paths are not altogether rare. In our system, the programmer must declare the modes of these deep paths that the analysis cannot derive on its own, if completeness is desired. In addition, the finite domain method may fail to note a moding contradiction, and thus silently produce unsound output for erroneous inputs.

To compare the utility of the two methods for a set of benchmark programs, we plan to generate paths from depth-one traversals of the complete graph and feed these sets to the finite path analyzer. For this we can compute the percentage of all syntactic variable modes determined unambiguously. For fully-moded programs this indicates how well the finite domain method approximates the constraint propagation method.

We believe that the graph algorithm is consistent, complete, and safe, although as noted above this remains to be proved. Thus, in applications requiring these properties, we recommend the use of the graph algorithm, as we currently know of no way to repair the finite domain algorithm, although it may be possible.

Note that if either algorithm reports a contradiction, there is no obvious way to automatically correct it, or even to determine the minimal subset of paths involved in the contradiction. Thus, we restrict our attention to the language of fully-moded (and thus non-contradictory) programs, and check this constraint with the mode analysis algorithm. The current implementation will report any contradiction, ignore the contradictory assertion, and proceed with the derivation. This allows the user to examine the final modes produced by the analysis and determine which might be incorrect. In our experience, this is usually sufficient to correct the problem. In practice, in the absence of user intervention, this also allows the modes of most of the remaining paths to be determined.

We have recently developed an parallel execution model [6] which requires only a fairly precise approximation to the modes of a program to optimize the performance of the program — in particular, the technique requires neither complete nor correct mode information. We believe that our finite domain analysis may be an adequate choice for this case because it is simple to implement, efficient, and reasonably precise.

5 Conclusions

This paper informally describes two alternative algorithms for deriving, at compile time, the path modes of flat committed-choice languages. One contribution of the paper is to refine Ueda and Morita's proposed method of constraint propagation into an implementable and efficient algorithm. This includes our novel abstraction of unification over variable and structure nodes in the graph. In addition, we describe an alternative method based on a finite path domain, with the motivation of decreasing analysis time by avoiding cyclic unification. This comes however at the expense of completeness, because we cannot guarantee that all paths are constructed.

Future work in this area includes integrating these algorithms into the front end of our experimental FGHC compiler, Monaco [9, 10]. We have already experimented with code sequentialization via mode analysis [4, 5] and continue to explore this application. Another important problem to be solved is making the mode analysis algorithm *constructive* in the sense that it can point out probable causes (in the source code) of illegal moding.

Acknowledgements

E. Tick was supported by an NSF Presidential Young Investigator award, with matching funds generously provided by Sequent Computer Systems Inc.

References

[1] M. Bruynooghe and G. Janssens. An Instance of Abstract Interpretation Integrating Type and Mode Inference. In *International Conference and Symposium on Logic Programming*, pages 669–683. University of Washington, MIT Press, August 1988.

[2] S. K. Debray. Static Inference of Modes and Data Dependencies in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11(3):418–450, July 1989.

[3] S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, 5(3):207–229, September 1988.

[4] B. C. Massey. Sequentialization of Parallel Logic Programs with Mode Analysis. Master's thesis, University of Oregon, September 1992. Also available as Technical report CIS-TR-92-18.

[5] B. C. Massey and E. Tick. Sequentialization of Parallel Logic Programs with Mode Analysis. In 4^{th} *International Conference on Logic Programming and Automated Reasoning*, St. Petersburg, July 1993.

[6] B. C. Massey and E. Tick. The Diadorra Principle: Efficient Execution of Fine-Grain, Concurrent Languages. Dept. of Computer Science, University of Oregon, June 1993. *In progress.*

[7] C. S. Mellish. Some Global Optimizations for a Prolog Compiler. *Journal of Logic Programming*, 2(1):43–66, April 1985.

[8] E. Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):413–510, 1989.

[9] E. Tick. Monaco: A High-Performance Flat Concurrent Logic Programming System. In *PARLE: Conference on Parallel Architectures and Languages Europe.* Springer Verlag, June 1993.

[10] E. Tick and C. Banerjee. Performance Evaluation of Monaco Compiler and Runtime Kernel. In *International Conference on Logic Programming.* Budapest, MIT Press, June 1993. Also available as University of Oregon CIS-TR-92-21.

[11] K. Ueda and M. Morita. Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, 1993. In press.

Better Late Than Never

Micha Meier
European Computer-Industry Research Centre (ECRC),
Arabellastr.17, 8000 Munich 81, West Germany
micha@ecrc.de

Abstract

Some Prolog systems are able to delay the execution of Prolog goals and resume it later. We were involved in the design, implementation and evaluation of several such systems and we summarise here our experiences with them. First we describe a general structure of such 'coroutining' systems and then we concentrate on the particular features of and choices made in three ECRC's logic programming systems: ECRC-Prolog, SEPIA and ECL^{i}PS^e .

1 Introduction

There are quite a few Prolog systems that were built in ECRC since its creation in 1984. Most of these systems had the ability to extend the default Prolog control, at least by suspending some goals and waking them later. With the appearance of CLP-like languages this type of control became very important, because it allows an easy implementation of constraint solvers in Prolog. We were involved in the design and implementation of some of these systems and we would like to share some of our good and bad experiences with these systems. We first describe the general structure of any coroutining system and mention various ways to design its particular features. In the following sections we concentrate on the particular systems: ECRC-Prolog, which was built in 1984–1986, SEPIA, its successor from 1987–1992, and ECL^{i}PS^e , created in 1992, which is intended to unify all LP systems currently being used in ECRC.

2 General Structure of a Coroutining System

Although there are relatively many systems able to delay and resume Prolog goals, most of the design and implementation details are part of Prolog folklore and they have been described only in few papers. Woken goals sometimes behave like coroutines and this is why delaying and waking goals is often referred to as *coroutining*, although real coroutines require a much more complex support. The first Prolog system with coroutining was Prolog-II with its **geler/2** primitive [3], which in other systems has the name **freeze/2**. **freeze(Var, Goal)** calls **Goal** if **Var** is instantiated, otherwise **Goal** is suspended and woken only after **Var** is instantiated. Similar mechanisms were available in other systems: MU-Prolog [9], IC-Prolog [4], ESP [2], SICStus Prolog [1], NU-Prolog [10] or in committed-choice languages [12, 13, 5, 14].

During our designs we have identified the main features which determine particular approaches and answer the important questions. Below we list the main questions that have to be answered by every design.

2.1 When to Delay a Call

Which condition has to be satisfied or violated for a call to be suspended? The condition can be expressed as a property of the caller (as e.g. in **freeze/2**), or as a property of the whole procedure (various declarations - **wait, block, when, . . .**). The condition can specify when the call has to be suspended (e.g. *delay clauses* in SEPIA) or the other way round, when it is allowed to continue (e.g. *when declarations* of NU-Prolog); it can even pretend to do the former and do the latter, like the *wait declarations* of MU-Prolog.

Next question is, what flexibility is allowed for the condition itself, is it a simple variable test, a boolean condition, or even a Prolog goal? For many applications, e.g. to catch infinite loops, a simple variable test is sufficient, but for more elaborate control more flexibility is required.

2.2 How to Represent Delayed Calls and Variables

When a call is delayed, the system creates a *suspension* which contains enough information to wake the goal later. The suspension must contain at least the procedure identification and the arguments, for realistic applications it must contain a little more than that.

The suspensions have to be stored in a memory area that is more permanent than the environment (local) stack. This is necessary because the environment of a clause can be popped even if one of its subgoals was delayed. Woken suspensions which are no longer needed should be popped or garbage-collected.

Variables that may cause the goal to wake (we call them *suspending variables*) must be marked somehow and it must be possible to reach the suspension from these variables. This is usually achieved by binding these variables to a structure that contains a suspension pointer. As one variable may have several suspensions attached to it, and also one suspension may have several variables which can wake it, an appropriate data structure has to be used.

The suspending variables must be carefully chosen to avoid waking the goal when it is known that the condition for continuation is still not satisfied. For instance, if a goal delays until a set of variables becomes ground, it suffices to mark only one variable from the set.

Some built-in predicates are usually written in the implementation language. For them the suspending mechanism may be different, in particular the delaying condition is hardcoded.

2.3 When to Wake a Delayed Call

A suspended goal is usually woken when one of its suspending variables is instantiated. Some systems also allow waking when a suspending variable is bound to another one or updated in a different way, e.g. by adding a new suspended goal to it. There may be other events that might trigger the waking of a suspended goal, however in our systems we restricted ourselves to events caused by such variable updates.

Further question is, whether a woken goal is triggered immediately after the binding of the suspending variable (and thus possibly interrupting the unification), or in every clause's neck, or only at certain specified places.

2.4 How to Schedule Woken Goals

Programs that heavily use coroutining create a chaos of goals woken at different levels intermixed with normal goals and backtracks. Each coroutining system must decide how the woken goals will be scheduled: oldest suspended goals first or last, built-in suspended goals first, etc. It must be also specified if the execution of a woken goal can be interrupted by another woken goal or if new woken goals are put into a queue which is executed only after the first woken goal is completely solved.

In a WAM-based machine it is also necessary to include the woken goals somehow in the execution process although there is no **call** instruction that calls it; the continuation handling must be also specified.

2.5 How to Integrate Woken Goals in the Normal Execution

A woken goal behaves as if it were textually written in the body of the clause whose head unification has touched the suspending variable. This means that we have to interrupt somehow the execution of compiled goals, save enough data to be able to resume it later, start the execution of the woken goal(s), and finally resume the interrupted execution. This process has some important impacts on the WAM, on register optimisations, shallow backtracking, etc.

2.6 How to Re-delay a Woken Call

When the delaying condition is not a trivial one, it can happen that the goal cannot be allowed to continue even after one or more of its suspending variables have been instantiated. The goal is then said to **re-delay**. The re-delaying can be handled as a usual delay, which may cause some actions to be unnecessarily repeated (e.g. creating a suspension), or the system may use the fact that the goal was already delayed and re-use some of the already existing data. This, however, may again make the scheduling less transparent. An important detail is that when a call re-delays, new suspending variables might have to be taken, i.e. it is not generally possible to rely on the fact that the suspension is already connected with the right suspending variables. For example, if $p(X)$ delayed until X becomes ground, X is the suspending variable. Later, when X is bound to e.g. $f(Y)$, the goal is woken but it re-delays and a completely new suspending variable, namely Y has to be taken.

2.7 Memory Management and the Value Trail

Although it is possible to introduce coroutining into a Prolog system with no or little special-purpose data structures and mechanisms (an example is SICStus Prolog [1]), we have always based our design on a special architecture. It included special structures for the suspensions, for the variables themselves, special-purpose WAM-instructions and the value trail, i.e. a trail stack which records both the trailed address and the value previously stored in it.

The issue of garbage collection of the unused data has also to be addressed. For instance it is possible that some coroutining structures become garbage even if they are still accessible from some places.

All ECRC Prolog architectures used tagged words with at least 8 bits in the tag. This means that we never had problems defining new data types and indeed we did so in an abundant way.

2.8 What Can Be Done with the Delayed Goals

During the Prolog execution the user might want to ask some meta-queries about the suspended goals. He may ask what are the goals suspended by a particular variable, what are the goals suspended since the program start or another point in the execution. He may also want to debug the program and ask these questions to the Prolog debugger and expect further debugger support concerning the suspended goals. The least that must be provided is to collect floundering goals [1] at the end of the query execution and print them together with the usual answer substitution.

3 ECRC-Prolog

ECRC-Prolog was the first real Prolog system built at ECRC in 1984–1986. It was in fact a WAM-based compiler for an enhanced MU-Prolog [9], which generated a C program and this program was then normally compiled with the C compiler to yield a stand-alone binary program. This was a somewhat strange combination especially because the Prolog compiler was rather slow and the generated C program was usually very big. This resulted in such long compilation times of both the Prolog and C compilers, that the system (which was not even incremental) had only little chance to survive (and it did not).

The choice of MU-Prolog with its *wait declarations* was deliberate, at that time it seemed to be the Prolog system with the most advanced and flexible control features. We did not want to restrict ourselves to a pure **freeze/2** implementation; it seemed more logical to view the control as a property of the whole predicate and not only of a call to it. We are by the way still convinced that the declaration-based suspensions are more appropriate than annotations in the caller, because most [2] of the time the suspension is due to the arguments of the call and not to the place it is called from.

3.1 Delaying

Wait declarations were quite unique among similar concepts; first because few if any managed to use them correctly, we were all the time mixing and's and or's and 0's and 1's. Second, because the question whether a call delays or not could be answered only after the head unification. On one hand this gives the wait declarations more power than can be found in the static annotations or declarations, on the other hand the implementation was awkward.

A wait declaration specifies which of the goal arguments may be 'constructed' (this means something like 'instantiated', but not quite) and which not. There could be several wait declarations for one predicate and if a head unification violated all of them, the call was delayed. For example, the **append/3** predicate could be declared as

```
    :- wait append(1, 1, 0), append(0, 1, 1).
```

which means that a call to **append/3** delays if both the first and the third argument need to be constructed. The fact that first the unification is finished and only then the suspension is tested has several implications:

- The delaying depends on the clause head. Clauses without nonvariables in the head could actually not delay and it was indeed sometimes necessary to use dummy clauses at the beginning of the procedure which would instantiate the necessary arguments.
- It is possible that some clauses of the same predicate delay while others do not. Apart from being an interesting idea, this has complicated the implementation.

[1] i.e. goals that are still suspended

[2] Later (see 4.1) we have in fact found out that sometimes it actually *is* the caller who specifies if the call should delay or not.

- If the predicate has several matching clauses, a choice point must be pushed before the unification. However, if the call delays, this choice point has to be removed, otherwise we might backtrack through all clauses without actually executing them; this might be logically correct, but it is rather inefficient.

- If a call delays, its unification with the clause head must be undone (otherwise the call could not be woken, because there would be no variables whose instantiation would trigger the waking). It is in fact an interesting idea whether the instantiations of variables that do not influence waking could be kept or not.

If the delay condition is tested statically, at the beginning of the predicate code, the execution must always start at this point, and it is difficult or impossible to make optimisations that skip some code parts. The fact that the delaying in ECRC-Prolog was tested only after the head unification has the advantage that the predicate does not have to be entered at one precise point and this was used by the compiler for indexing: if it was known that after waking a certain argument must be instantiated, the resuming address was in the code that indexed (also) on this argument.

The unification of a call with a predicate with wait declarations was executed by special unification instructions which have created a bit mask of arguments that were constructed. When the unification failed, nothing happened. If it succeeded, there was a **neck** WAM instruction in the clause neck which has compared this mask with masks derived from the wait declarations. If there was a mask that allowed to continue, the execution continued normally (that is why the name *wait* is not quite appropriate). Otherwise, the system had to undo the unification, collect the suspending variables, make the suspension and link them together. Undoing the unification was a problem for calls without a choice point, because the binding of some variables might not have been trailed. One can of course change the rule and trail everything, but this seemed to be too high a price because the majority of goals do not delay. We have therefore introduced the *auxiliary trail* that recorded changes in the deterministic state. If the call delayed, it was used to undo the unification and to pick up the variables that were bound in the unification, which then became the suspending variables. If the call did not delay, the auxiliary trail was simply cleared. The auxiliary trail was used only in predicates with delay declarations and thus only in the special unification instructions that also had to construct the bit mask.

3.2 Data Structures

The suspension was represented by a *delayed environment*, which contained the following items:

- the call arguments and arity,
- the resuming entry point address,
- flag specifying if a choice point has to be created when the goal is woken,
- the woken goal continuation.

The suspending variables were bound to a word with a special **suspend** tag, whose value was a list of suspensions.

3.3 Waking

Whenever a suspending variable was instantiated or bound to another one, the **neck** instruction took care of waking the suspensions in its associated list. As the suspending variables were always trailed on the value trail, it sufficed to check whether the unification has modified the value trail. If a suspending variable was bound to another one the two lists were merged together and nothing was woken, otherwise the goals on the list were woken.

The clause whose head unification had woken some delayed goals was then itself suspended, a *resuming environment* was created which contained the WAM argument registers with meaningful values, and the **CP** and **PP** registers.

The handling of continuation was quite interesting: the WAM **CP** register pointed either normally to the code of the next goal to execute, but it could also point to a delayed environment or to a resuming environment. When some goals were woken, their suspensions were linked together using their continuation field, the last one pointed to the resuming environment of the clause that had woken them. The **proceed** instruction tested where **CP** pointed to and performed the appropriate actions. The **neck** instructions did not actually call the woken goals, it only linked them together and invoked the first one. In case one of the woken goals failed, we had thus done unnecessary work with linking the following ones. There were suggestions to call woken goals directly in each **neck** instructions and then to continue with the clause body. They were rejected because the goals had to be scanned anyway because of sorting (see below) and then it was simpler to use the direct continuations.

3.4 Scheduling

Since one goal can be suspended by more than one variable, it is necessary to mark the suspensions that were already woken and executed to prevent waking them again when another suspending variable is instantiated. In ECRC-Prolog this was done using the continuation field in the suspension; if it was set it meant that the goal was already woken and thus it was ignored for all the subsequent instantiations of other suspending variables.

Since we wanted to wake the goals strictly according to their age, the lists of all instantiated suspending variables were merged together and sorted so that oldest goals were guaranteed to be woken first. The woken goals were then inserted at the beginning of the current continuation chain. This means that the execution of a woken goal was itself interrupted when a new suspending variable was bound, and new woken goals were triggered immediately. Since the lists of woken goals were explicitly accessible, it would have been possible to insert the newly woken goals according to their age in this list so that waking would completely correspond to the age of the delayed goals, but this was rejected as exaggeration.

3.5 Re-delaying

Re-delaying a call was simpler than delaying - since the suspension already existed, it was simply added to the delay lists of all suspending variables in the call and its continuation was reset back to zero to mark that it was not woken. One could not be sure that the suspending variables for the re-delay are the same as those for the first delay. The consequence was that the same suspension might have occurred several times in the same list, but we did not find any efficient and general method to identify which variables already have this suspension in their list and which do not.

3.6 Memory Management

ECRC Prolog had three trails: the usual trail, the auxiliary trail needed to undo the unification when a deterministic call delayed, and the value trail. The main purpose of the value trail was for inserting new goals into the delay lists. This is in fact not necessary – the delay lists may be ended up by a variable and new goals can be simply appended to the end and this link is trailed as usual. This approach, however, has the disadvantage that the time needed to build the whole list becomes quadratic in the list length. Although it is possible to reduce this overhead by variable shunting [11], the quadratic complexity remains. We have therefore inserted new suspensions at the beginning of the delay lists and the link from the variable to the list was value-trailed.

One fact deserves mentioning: when a plain variable delays a goal and becomes a suspending variable, it seemed that it could be trailed with the normal trail because it had no value before. Later, when new goals delay on it, the value trail would be used. However, the fact that we had two *separate* trail stacks was the cause of the most obscure bug I've seen so far and searched for almost a week: there were cases when the variable was not untrailed properly, no matter if we first untrailed the normal or the value trail.

3.7 Support for the Delayed Goals

There was not much that could be done with suspended goals apart from waking them. Floundering goals were discovered by scanning the value trail at the end of the query. Value-trailed items that were lists of suspensions were searched for a suspension which was not woken and if so, the system printed the message that some goals are still delayed, but it did not say which ones. There was no debugger support nor any built-in predicates to access the delayed goals.

3.8 Conclusion

Coroutining in ECRC-Prolog was actually very efficient, even if there was much extra work to do, e.g. sorting the delay lists. It was an order of magnitude faster than MU-Prolog and, especially with disjunctive delays, than SICStus 0.3. The idea to unify first and then check if the call delays is quite interesting and close to concurrent Prologs and it has several advantages, however the form of the wait declaration was not quite appropriate.

One of the objectives was to wake the delayed goals exactly in the order they were delayed. However, it turned out that after several delays and wakings no-one really knows what is happening. No matter what waking order was taken, it was almost impossible to tell which woken goal comes from where and who is who's parent. Preserving the order was thus not really helpful.

Several users have had problems with cuts in their programs. We have tried to find a scheme that would be safe with respect to cuts and still not too expensive, but didn't find any. Since then we were convinced that one should not mix cuts and coroutining, and if possible get rid of the cut completely (keep only **once/1** and **->/2**). This does not solve the problem, but it makes a safer ground to build on.

4 SEPIA

In the SEPIA [7] design we have tried to learn from the problems and to do everything better (the "second system syndrome"). Since we felt that the main problems of the previous implementation were slow compilation, non-incrementality and bad performance, the main changes were made there. SEPIA is based on a WAM-emulator, its compiler is written in C to make the compilation as fast as possible, and there is a special compilation and execution mode for the execution with delayed goals. The aim is that in the non-coroutining mode the machine is as fast as possible, without any overhead caused by the special features of the system. There is, however, one major difference between SEPIA and other Prolog systems, namely the word size. In SEPIA the tag is 32 bits long and thus the size of every Prolog item is 2 words. There were several reasons for this, the main one was that we wanted both more space for the tag than 2 or 3 bits, and at the same time 32 bits for the value part so that pointers could be stored directly.

In SEPIA there are also two types of predicates: simple and regular, the former are an extension of in-line expanded predicates, any deterministic predicate written in C is simple, all others are regular. This distinction helps to identify sequences of predicates whose execution does not change any important WAM register.

4.1 Delaying

The delaying in SEPIA is controlled by *delay clauses*. A delay clause looks like

```
delay p(X, Y) if var(X), var(Y).
```

and it specifies explicitly under which condition the call to this procedure should delay. We have thus moved from dynamic delaying conditions in ECRC-Prolog to static ones. The use of delay clauses follows quite naturally from the requirements that the delaying must be flexible enough to allow specification of complex conditions, which are necessary to implement various constraints propagation schemes. The use of a delay condition is more natural than a 'continue' condition, because a predicate without any condition should never delay and thus an implicit continuation condition must be always assumed. A delay condition, on the other hand, expresses directly what should, or should not happen.

A delay clause in this form can be very easily compiled, by transforming it to

```
p(X, Y) :- var(X), var(Y), delay(p(X, Y)).
```

and this is more or less what the SEPIA compiler does. The head of a delay clause uses one-way pattern matching rather than the full unification, and also the body of the delay clause is not allowed to bind any variable in the call. This is important because delay clauses are in fact meta-clauses and they must not bind the object variables, they can only test the call arguments. The predicates allowed in the body of a delay clause are **var/1**, **nonground/1**, **\ ==/2** and user-defined external predicates. We planned initially to allow any subgoal in a delay clause, but it turned out that the combination of the above predicate was sufficient for almost all coroutining programs. For instance, the **and(In1, In2, Out)** predicate that implements the logical conjunction would have a delay clause

```
delay and(X, Y, Z) if var(X), var(Y), X \== Y, Z \== 1.
```

to delay exactly when it is not possible to solve it deterministically.

One special condition for the delay clauses was used very frequently, in particular in programs that implemented various sorts of constraint propagation. If we impose a constraint on a set of variables and we want this constraint to propagate as soon as possible, we have to put the suspended constraint on the delay list of every variable from this set and as soon as any of these variable is changed, we want to wake the constraint, do the propagation, and suspend again, unless the constraint is already solved. This cannot be done with the above mentioned predicates, unless we make specialised and awkward versions of the constraint that work with 2, 3, ..., n free variables in it. To allow this kind of processing, Joachim Schimpf devised the built-in condition **initial/1** that does the following: if the predicate is called directly, it succeeds and marks all variables in its argument as suspending variables. When the call is woken, this condition fails and the predicate will thus be executed.

This is in fact one of the rare occasions where the predicate should delay no matter what are its arguments, and it is thus the caller who decides about delaying. As a matter of fact, the solution with **initial/1** is just a trick to achieve caller-defined suspension, and it is often necessary to define auxiliary predicates to make it work. For example, when we define a constraint **</2** which takes as arguments two arithmetic terms with variables ranging over finite integer domains, in SEPIA it has to be implemented as follows:

```
A < B :-
    normal_form(A, B, A1, B1),
    propagate_lt(A1, B1).

propagate_lt(A, B) :-
    <update the domains of all variables to make
     them consistent with the constraint>
    ground((A, B)) ->
        true
    ;
        delay_lt(A, B).

delay delay_lt(A, B) if initial((A, B)).
delay_lt(A, B) :-
    propagate_lt(A, B).
```

In this way, after updating the domains the constraint is called recursively. This call delays and it waits for any variable occurring it to be updated, e.g. by modifying its domain.

4.2 Data Structures

The basic structure is similar to that of ECRC Prolog. A suspended call creates a suspension which contains its arguments, code address and the woken flag which specifies if this suspension has been already resumed or not. A suspending variable is represented by a sequence of at least three words, the first one is a variable with tag **suspending**, the second is a list of suspended goals that have to be woken when the variable is instantiated and the third one is a list of suspended goals to be woken even if the variable is bound to another suspending variable (this is necessary to implement **\==/2** in delay clauses).

There are also two additional stacks, one for the variables that are responsible for suspending the current goal. They are pushed on it by the body of delay clauses together with a flag which says which of the two delay lists should be used to hold this goal. The other stack is for suspending variables bound in the head unification; at the end of the unification they are collected from it and their suspensions are resumed.

4.3 Waking

The question of the interference of waking and the cut was analyzed thoroughly. The core of the problems is that the cut is a sequential operator whereas coroutining destroys the sequential execution. One problem concerns cutting over suspended goals:

```
max(X, Y, X) :- X >= Y, !.
max(_, Y, Y).
```

If X or Y is not instantiated, the test delays and the cut is executed, even if later Y is bound to a number greater than X. It would be too costly to implement this properly and therefore we decided to check this situation only in the debugger and let the debugger print a warning.

The other problem concerns the waking. If the head unification instantiates some suspending variables and there is a cut after the clause neck, should we wake the suspended goals before or after the cut? If we wake before the cut, we might cut away a choice point of a woken goal:

```
delay d(X, _) if var(X).
d(a, 1).
d(a, 2).
d(c, 2).

a(a) :- ..., !.
a(c).

p(X) :- d(X, Y), a(X), Y = 2.
```

When we call *p(X)*, the call to **d/2** will be woken inside **a/1**, its choice point will be cut and the unification $Y = 2$ fails. On the other hand, calling *p(a)* or *p(c)* succeeds.

If we wake after the cut, we might again commit to the wrong clause:

```
b(1) :- !.
b(2).

?- X > 1, b(X).
```

The call to $X > 1$ initially delays and if we first execute the the cut and only then wake the suspended goal, the query fails.

Since none of the alternatives is superior to the other and a sophisticated implementation would be too costly, we decided to take the pragmatic approach and wake when it suits best to the abstract machine. Therefore, SEPIA wakes only immediately before a regular goal or at clause end; sequences of simple (e.g. in-line expanded) predicates do not cause any waking. In this way, the user can still force waking before the cut, namely by inserting a regular goal (e.g. **true/0**) before the cut.

Waking is done as follows: in coroutining mode the compiler inserts **resume** instructions at places where a suspending variable might have been instantiated. The instruction checks the waking stack and if there are some suspending variables pushed on it, it calls the routine to wake these goals. To have more efficiency, the waking routine was hardcoded using special WAM instructions and it was a source of numerous bugs. It might even be that writing it straight in Prolog would have made it almost as fast and much simpler, especially if we take into account that we needed two copies of it, one for the optimised case and one for the debugger (SEPIA has no interpreter, the debugger uses the compiled code enhanced by some debug instructions). This scheme is quite efficient because very often the compiler can recognise that no variables will be bound (mode declarations help, too). On the other hand, it is not possible to mix code compiled in coroutining and non-coroutining mode and so it is allowed to switch on the coroutining mode only before any user predicate is compiled. Sometimes this is quite inconvenient.

There is an interesting situation when an iterative clause (a non-unit clause without an environment) has woken some goals. We have to call the waking routine, but the clause has no environment, and so we have no place to store the continuation. This problem was solved by allocating the environment in the **resume** instruction and deallocating it by a **deallocate** instruction that follows it. If there are no goals to resume, no environment is allocated and the **deallocate** instruction is skipped.

Before a suspension is invoked, its **woken** bit is set and this change is value-trailed if necessary.

4.4 Scheduling

There is no particular waking order in SEPIA, the new suspensions were inserted at the beginning of the delay list and after the unification all delay lists of bound suspending variables were linked together and woken in this order. To make the linking in constant time, a pointer to the last element in the list was stored in the suspending variable. Later this became a circular list so that it was possible to insert new suspensions both at the beginning and at the end of the list, but this feature was never really used, because the users wanted mostly predicates of some type to be woken first or last and this could not be guaranteed if the unification instantiates more than one suspending variable.

4.5 Re-delaying

After a goal was woken, but one of the delay clauses succeeded again, it is re-delayed. At this time the pointer to the suspension is still available and so only its **woken** bit is reset to zero (unlike in ECRC-Prolog this change does not have to be trailed) and the suspension is placed in delay lists of all suspending variables of the call. This is still slightly inefficient because the suspension might already be inside some or all of them. It could have been optimised - goals suspended by delay clauses with only **var/1** conditions are guaranteed to be in all appropriate delay lists on re-delay.

4.6 Memory Management

SEPIA has only one trail which can store entries of various types, each entry is tagged to recognise its type. All coroutining data structures are located on the global stack.

4.7 Support for the Delayed Goals

SEPIA, whose goal was to be a system that could be easily extended, needs various facilities to process suspended goals. We provided a predicate that, given a variable, converted its lists of suspensions to a list of goals and returned this to the user, so that it could be processed in Prolog. This turned out to be not quite sufficient, because the list does not contain the definition modules of the delayed procedures, so the goals cannot be called. In order to obtain a list of all currently suspended goals (needed also in the top level loop to print floundered goals), all suspensions are linked together, whenever a new goal is suspended, its suspension is prepended to this list. This is not too constly and it allows a very fast access to suspended goals; on the other hand, it complicates the garbage collection because all suspensions are accessible, even if they are garbage.

The debugger was enhanced to take into account the coroutining and the above mentioned cut warnings. It has delay and resume ports and commands to display suspended goals, to skip to the place where a goal is woken, etc.

4.8 Conclusion

The SEPIA coroutining turned out to be very useful and very efficient, we have not found any system with faster coroutining execution. The reason for this was of course that most of the primitives were hardcoded and impossible to change.

After some time, we implemented metaterms (attributed variables) [8] on top of the coroutining primitives, because it was very easy, but it was conceptually strange because normally one would expect coroutining to be built on top of attributed variables and not vice versa.

We have had many user requests to change the way coroutining works and it turned out that flexibility is more needed than performance. The users mostly wanted to use more than two delay lists in a suspending variable, or to change the waking order. With various constraint propagation systems being built on top of SEPIA it became clear that everybody would benefit from lifting the coroutining implementation to a level higher where it could be more easily changed.

5 ECL^{i}PS^e

With metaterms already available in the language it was clear that implementing coroutining on top of them would be easy and flexible enough. In ECL^{i}PS^e , which was created by merging SEPIA with another ECRC LP system, MegaLog, we have, together with Joachim Schimpf, made metaterms into first class Prolog objects. They have their own syntax, compilation, etc., and we have built coroutining strictly on top of the metaterms. The code changes were in fact not very big, we have mostly replaced large portions of C code by several Prolog lines. This means more or less that there is no particular support for coroutining any longer, suspending variables are just metaterms whose attributes store the suspended lists and whatever else is needed. We have of course provided macro transformations for backward compatibility so that delay clauses still work normally, however the user has the possibility to inspect and modify any part of the coroutining scheme.

We have imposed only some restrictions on the data structures used: the suspended lists are difference lists so that one can add new suspensions at the beginning or at the end (using value-trailed destructive assignments), the suspension is a special opaque data type and we provided a predicate to convert a goal to a suspension and vice versa, another one to insert a suspension to a specified delay list in the metaterm's attribute and finally one that invokes the suspension and sets its `woken` flag. A goal can be delayed explicitly by creating a suspension and inserting it into a delay list. Since unification of metaterms raises an event, the waking is completely taken care of by the metaterm event handler. The handler usually takes the attributes of suspending variables, finds the suspensions and calls them.

The default scheduling strategy is like the one of SEPIA, however the users have now the possibility to change the way suspensions are handled; they can define new suspended lists and the order in which they are woken, e.g. to wake simple deterministic goals first, etc. They can even define one global delay list where all suspensions are stored and woken strictly in the order of their suspension, or divide predicates into several classes and state that while a predicate of a certain class is executing, it must not be interrupted by waking a suspended goal of a lower class, etc.

The performance of the ECL^{i}PS^e coroutining is of course below that of SEPIA, the primitive actions (suspending and waking) are in average 50% to 100% slower, however in large programs the proportion of these primitive actions seems to be not significant. The main point is nevertheless the increased flexibility of the whole design which will allow us to experiment with new systems and schemes and which could in fact bring much higher gains in efficiency.

The experiences we have had so far with ECL^{i}PS^e are mostly positive. For instance, the library that implements arithmetic constraints over finite integer domains was adopted to the new scheme. It was previously written in SEPIA using only suspended goals. With metaterms it could be simplified and cleaned. While keeping the whole control and scheduling in Prolog, we have eliminated some of its bottlenecks due to slow Prolog processing of arithmetic expressions by rewriting them in C, and the performance of the resulting system is not too far away from CHIP, with the difference that in CHIP all the constraint processing is hardcoded in C while ECL^{i}PS^e can very easily define new constraints or change the control strategy for some or all of them.

Currently we are evaluating the new design and trying to identify features that may not fit together well. One of them is the question how to organise the use of metaterms. If one program assumes a certain structure of the metaterm attribute, it is not compatible with other programs that assume a different structure. It seems that making the metaterms module-dependent could be a simple way to solve it. Another problem is how to handle suspending variables in built-in predicates. When e.g. a suspending variable occurs in an asserted term, should all its suspensions be compiled with it or ignored? Similarly, in the instance test, we sometimes want to take the attributes into account and sometimes don't.

Since the delay clauses are no longer treated as special primitives, the compiler cannot take them into account and so it does not know whether a predicate might delay or not. It does not recognise re-delays, either. On the other hand, the user has the full control and can optimise re-delays explicitly, or even make iterative re-use of suspensions as suggested in [6]. This is a good example of the rule that the more a user can do, the less can be done by the compiler and vice versa.

Acknowledgements

We thank to Joachim Schimpf for valuable comments on a previous version of this paper and for his contribution in SEPIA and ECL^{i}PS^e development.

References

[1] Mats Carlsson. Freeze, indexing and other implementation issues in the WAM. In *Proceedings of the 4th ICLP*, pages 40–58, Melbourne, May 1987.

[2] Takashi Chikayama. Esp reference manual. Technical Report TR-044, ICOT, February 1984.

[3] Alain Colmerauer. Prolog II manuel de reference et modele theorique. Technical Report ERA CNRS 363, Groupe Intelligence Artificielle, Faculte des Sciences de Luminy, March 1982.

[4] S. Gregory K. L. Clark, F. G. McCabe. Ic-Prolog language features. In *Logic Programming, ed. Clark and Tarnlund*, pages 253–266. Academic Press, London, Departmemt of Computing, Imperial College, London, 1982.

[5] Yasunori Kimura and Takashi Chikayama. An abstract kl1 machine and its instruction set. In *Proceedings 1987 Symposium on Logic Programming*, pages 468–477, San Francisco, September 1987.

[6] Micha Meier. Recursion vs. iteration in Prolog. In *Proceedings of the ICLP'91*, pages 157–169, Paris, June 1991.

[7] Micha Meier, Abderrahmane Aggoun, David Chan, Pierre Dufresne, Reinhard Enders, Dominique Henry de Villeneuve, Alexander Herold, Philip Kay, Bruno Perez, Emmanuel van Rossum, and Joachim Schimpf. SEPIA - an extendible Prolog system. In *Proceedings of the 11th World Computer Congress IFIP'89*, pages 1127–1132, San Francisco, August 1989.

[8] Micha Meier and Joachim Schimpf. An architecture for prolog extensions. In *Proceedings of the 3rd International Workshop on Extensions of Logic Programming*, pages 319–338, Bologna, 1992.

[9] Lee Naish. An introduction to MU-PROLOG. Technical Report 82/2, University of Melbourne, 1982.

[10] Lee Naish. Negation and quantifiers in NU-Prolog. In *Third International Conference on Logic Programming*, pages 624–634, London, July 1986.

[11] Dan Sahlin and Mats Carlsson. Variable shunting for the WAM. In *Proceedings of the NACLP'90 Workshop on Prolog Architectures and Sequential Implementation Techniques*, Austin, October 1990.

[12] Ehud Shapiro. A subset of concurrent Prolog and its interpreter. Technical Report TR-003, ICOT, Tokyo, Japan, January 1983.

[13] K. Ueda. Guarded horn clauses. Technical Report TR 103, ICOT, 1985.

[14] Kazunori Ueda and Masao Morita. A new implementation technique for flat GHC. In *Proceedings of the 7th ICLP*, pages 3–17, Jerusalem, June 1990.