CONTINUATION COMPILATION FOR CONCURRENT LOGIC PROGRAMMING

B. Kemp and P. Soper
Dept. of Electronics and Computer Science,
University of Southampton,
Southampton, S09 5NH, UK.

A. King
Computing Laboratory,
University of Kent at Canterbury,
Canterbury, CT2 7NF, UK.

Abstract. A new and powerful approach to threading is proposed, that is designed to improve the responsiveness of concurrent logic programs for distributed, real-time AI applications. The technique builds on previously proposed scheduling techniques to improve responsiveness by synchronously passing control and data directly from a producer to a consumer. Furthermore, synchronous transfer of data requires less buffering and so less garbage is produced. Arguments are also passed in registers, further reducing overheads.

Keywords. Concurrent logic programming, responsiveness, garbage collection, program optimisation and abstract interpretation.

1 Introduction

Real-time telecommunications switching systems can be expressed elegantly as concurrent logic programs [Elshiewy 90] and, more generally, symbolic knowledge-based control applications fit well with the concurrent logic programming model [Elshiewy 90]. Declaratively, concurrent logic programs inherit a logical interpretation. Operationally, concurrent logic programs define asynchronous message-passing between reactive processing agents. The paradigm is therefore potentially a distributed AI programming language par excellence. However, although the concurrent logic programming paradigm has been distilled into real-time languages like Sandra (which supports persistence, secure communication, recovery, clocks, *etc.* [Elshiewy 90]) and RGDC (which has been compared to a symbolic or logical version of occam [Cohen$^+$ 91]), the expressiveness of concurrent logic programming does not come without cost. The efficiency of a traditional (sequential) AI language like Prolog comes, in part, from its control strategy. The control strategy of Prolog brings with it an efficient stack based implementation. Concurrent logic languages, on the other hand, substitute control-flow with a data-flow model based on asynchronous message-passing which, in turn, incurs overheads from scheduling, argument copying and an increased memory turnover. High memory turnover can, in particular, impede the performance of a distributed system due to increased garbage collection. These overheads can compromise the responsiveness of concurrent logic programs and therefore it is vital that these overheads are reduced if concurrent logic languages like Sandra and RGDC are to be widely used in real-time distributed AI applications.

This paper deals with the thorny issue of improving the responsiveness of concurrent logic programs. The approach involves refining thread-based compilation schemes to increase the responsiveness of the scheduler to incoming messages, reduce argument copying, and also avoid the generation of unnecessary garbage. Basically, thread-based compilation [King$^+$ 92, Massey$^+$ 93b, Massey$^+$ 93a] automatically introduces control-flow back into data-flow logic programs. Specifically, threading boils down to deducing at compile-time a partial schedule of processes, or equivalently the body atoms of a clause, which is consistent with the program behaviour. To avoid compromising program termination, an ordering of the atoms is determined which does not contradict any data-dependence of the program. In general the processes cannot be totally ordered and thus the analysis leads to a division into threads of totally ordered processes. In this way the work required of the run-time scheduler is reduced from ordering processes to ordering threads and, additionally, memory management is improved.

In the thread approach, pairs of coroutining atoms give rise to cyclic data-dependencies that can only be resolved at run-time with a scheduler. Coroutining atoms are therefore allocated to different threads and thus incur the usual performance penalty associated with frequently suspending processes. Threading can only increase throughput by reducing the number of suspensions if the program gives rise to processes that suspend relatively infrequently. Thus threading favours computationally-intensive programs [Ueda$^+$ 90, Ueda$^+$ 92] but is of limited use for programs which make extensive use of back-communication and coroutining [King$^+$ 92], programs which [Ueda$^+$ 90, Ueda$^+$ 92] dub storage-

```
qs([], Hd, Tl)        :- Hd = Tl.
qs([N | Ns], Hd, Tl):- pt(Ns, N, Lo, Hi), qs(Lo, Hd, [N | Mid]), qs(Hi, Mid, Tl).

pt([N | Ns], Mid, Lo, Hi) :- Mid < N | Hi = [N | His], pt(Ns, Mid, Lo, His).
pt([N | Ns], Mid, Lo, Hi) :- Mid >= N | Lo = [N | Los], pt(Ns, Mid, Los, Hi).
pt([] , _, Lo, Hi)            :- Lo = [], Hi = [].
```

Figure 1: The quicksort program qs/3.

intensive. Storage-intensive programs might typically implement reconfigurable data structures or capture state as networks of almost always suspending processes [Ueda+ 90, Ueda+ 92]. Unfortunately, storage-intensive programs often arise in concurrent logic programming [Taylor+ 89]. This paper presents continuation compilation as a way of addressing this threading deficiency. Continuation compilation refines threading by introducing control-flow in a way that is applicable to the whole spectrum of computationally-intensive and storage-intensive programs. The net effect is improved responsiveness for a wider class of program.

Continuation compilation adopts a message-oriented approach to threading. The idea behind message-orientated scheduling [Ueda+ 90, Ueda+ 92] is to compile stream communication into procedure calls to the consumer of a stream to improve the responsiveness of a system to message-sends and hence optimise storage-intensive programs. A similar control mechanism is used in continuation compilation. Although communication is asynchronous and therefore streams can act as unbounded buffers, continuation compilation endeavours to reduce the overheads of synchronisation and buffering by transferring control to the consumer process when the producer process writes to a shared variable. The producer and consumer processes are not restricted to being siblings in the proof tree (i.e. need not arise from the same clause) as in the previously proposed threading techniques [King+ 92, Massey+ 93b, Massey+ 93a], but instead are permitted to be arbitrary producer and consumer processes. Thus continuation compilation in some ways generalises conventional threading. Continuation compilation also refines message-orientated scheduling. Message-orientated scheduling tunes storage-intensive programs for improved responsiveness to incoming messages at the expense of increasing the cost of broadcasting. Basically, message-orientated scheduling optimises process switching for goals which suspend and resume frequently. Continuation compilation, on the other hand, applies threading in a partial way to avoid performance degradation. As a consequence, continuation compilation can be implemented with a conventional abstract machine [Kimura+ 87] embellished with only some basic message-oriented features. Continuation compilation also augments the message-oriented scheduling model with a form of compile-time garbage collection. From an analysis of the producers and consumers of data, data-structures constructed in a producer that are immediately destructed in the consumer can be passed in registers. This can avoid structure manipulation and save on garbage collection.

The paper is structured as follows. Section 2 illustrates the basic ideas behind continuation compilation with a motivating example. In section 3, the focus is on the analyses which underpin continuation compilation whereas section 4 presents an abstract machine which supports the continuation compilation view of threading. Finally, section 5 presents the related work and concluding discussion. For reasons of brevity, the paper assumes a basic knowledge of advanced AI programming and the optimisation technique of abstract interpretation [Cousot+ 92].

2 Intuition behind Continuation Compilation

To illustrate the basic ideas behind continuation compilation, consider the execution of the goal qs([2, 3, 1], L, []) and the program qs/3 of figure 1. The program qs/3 encodes the recursive quicksort algorithm using difference-lists to amortise the overhead of list concatenation. Figure 2 depicts the control-flow for continuation compilation (marked in thick arrows) superimposed on the proof tree for the goal qs([2, 3, 1], L, []) (illustrated in thin lines).

Figure 2 illustrates that under qs/3, the goal qs([2, 3, 1], L, []) reduces to the sub-goals pt([3, 1], 2, Lo_1, Hi_1), qs(Lo_1, L, [2|Mid_1]) and qs(Hi_1, Mid_1, []). The two consumer processes qs(Lo_1, L, [2|Mid_1]) and qs(Hi_1, Mid_1, []) suspend on the shared variables Lo_1 and Hi_1 until the producer process pt([3, 1], 2, Lo_1, Hi_1) binds either Lo_1 or Hi_1. Once Lo_1 or Hi_1 are bound the consumers can be resumed. In the case of Lo_1, for instance, control can be passed from pt([3, 1], 2, Lo_1, Hi_1) to qs(Lo_1, L, [2|Mid_1]). In fact this is always possible since Lo_1 has a unique producer and therefore the qs/3 process cannot be awakened by another process. (Single-threaded analysis [Sundararajan+ 92] can be used to detect this single producer property.)

One characteristic feature of continuation compilation is that transfer of control is (partially) effected by over-loading the standard suspension mechanism with special control machinery. Conventionally a suspension is implemented by hooking the suspended processes off the variable which induced the suspension [Kimura+ 87]. Specifically, the unbound variable is adjusted to point to a list of suspended process descriptors so that on binding the variable, the process descriptors can be inserted back into run queue [Kimura+ 87]. Thus, the processes will eventually be re-executed. In continuation compilation, the vari-

ables Lo_1 and Hi_1 are created with special references to the process descriptors for qs(Lo_1, L, [2|Mid_1]) and qs(Hi_1, Mid_1, []), and also, the two qs/3 processes are omitted from the run queue. Therefore, in one sense, the qs/3 processes are suspended prematurely – a technique which has been dubbed pre-suspension. Thus, on binding either of Lo_1 or Hi_1, the consumer can be identified immediately and therefore the control passed to it.

As a consequence of effecting a fast form of process switch, continuation compilation can also refine memory management. Specifically, in the conventional approach, binding either Lo_1 or Hi_1 creates a list cell in the heap [Kimura+ 87]. This is because streams can act as buffers between asynchronous processes. Put another way, streams are required to live as long as the processes which reference them and thus a run-time garbage collector is needed to free up the heap. Continuation compilation changes this by making process interactions more synchronous. For instance, once Lo_1 is bound in pt([3, 1], 2, Lo_1, Hi_1), control is immediately passed to qs(Lo_1, L, [2|Mid_1]) so that because Lo_1 is not referenced elsewhere and its life-time is short, the list cell for Lo_1 can be allocated to registers. In fact, continuation compilation can refine the handling of streams still further. Since pt([3, 1], 2, Lo_1, Hi_1) can only bind Lo_1 to a list cell and qs(Lo_1, L, [2|Mid_1]) can only match Lo_1 against a list cell, the cell construction is actually redundant. Thus superfluous structure manipulation is avoided by allocating the head of the list to one register and the tail of the list to another. This improves register usage and therefore reduces the overhead of argument copying in conventional abstract machines [Kimura+ 87].

In figure 2, a thick-arrow connects the reduction for qs([2, 3, 1], L, []) to the reduction for pt([3, 1], 2, Lo_1, Hi_1). The reduction of pt([3, 1], 2, Lo_1, Hi_1), however, introduces two further sub-goals, Hi_1 = [3|His_1] and pt([1], 2, Lo_1, His_1). Applying the binding Hi_1 = [3|His_1] passes control to qs(Hi_1, Mid_1, []) which, in turn, reduces to the sub-goals pt(His_1, 3, Lo_2, Hi_2), qs(Lo_2, Mid_1, [3|Mid_2]) and qs(Hi_2, Mid_2, []). The variables Lo_2 and Hi_2 are created, like before, with special references to the process descriptors for qs(Lo_2, Mid_1, [3|Mid_2]) and qs(Hi_2, Mid_2, []). To deal with the variable His_1, however, requires slightly more subtlety. Note that the process pt([1], 2, Lo_1, His_1) is a unique producer for His_1 and that because control is always passed immediately from the unification Hi_1 = [3|His_1] to qs(Hi_1, Mid_1, []), the variable His_1 is guaranteed to be unbound when the goal pt(His_1, 3, Lo_2, Hi_2) is created. Thus pt(His_1, 3, Lo_2, Hi_2) can be created in the suspended form with the variable His_1 updated with a special reference to process descriptor for pt(His_1, 3, Lo_2, Hi_2). Hence, with a more general notion of pre-suspension, the process pt(His_1, 3, Lo_2, Hi_2) can immediately be located and awakened once His_1 is bound.

As a final note, observe that since control is passed from Hi_1 = [3|His_1] to qs(Hi_1, Mid_1, []), the process pt([1], 2, Lo_1, His_1) cannot be reduced immediately. Therefore it is inserted into the run-queue. In a sequential implementation, pt([1], 2, Lo_1, His_1) is removed from the run queue after the reduction of qs(Hi_1, Mid_1, []). This run queue access is reflected in figure 2 with a circle. More generally, circled control-flow links in figure 2 indicate enqueuing and dequeuing operations for the run queue. Alternatively, in a parallel implementation, the thread of control for qs(Hi_1, Mid_1, []) could be executed by another processor. Thus, in a multi-processor implementation, circled control-flow links indicate opportunities for parallelism.

3 Analysis for Continuation Compilation

Although continuation compilation significantly revises and extends the basic threading model [King+ 92], continuation compilation has been designed to be underpinned by existing and well-understood forms of program analysis. Specifically continuation compilation can be automated with the abstract interpretation techniques of single-threadedness analysis [Sundararajan+ 92] and local suspension analysis [Codish+ 93]. Single-threadedness analysis [Sundararajan+ 92] identifies variables which have a single producer (writer) process and a single consumer (reader) process. Single-threaded variables have nice garbage collection properties [Sundararajan+ 92] but also the producer and consumer processes implicitly define a data-dependence. Tracing these data-dependences is, in essence, continuation compilation.

Continuation compilation translates implicit data-dependencies of a program into explicit scheduling operations, and therefore, like other forms of threading [King+ 92], can potentially compromise the termination characteristics of a program. It is vital, therefore, that continuation compilation is applied in a principled way which ensures safety. Specifically, the optimisation must not introduce indefinite suspensions into any part of the computation. To avoid the introduction of indefinite suspensions, local suspension analysis [Codish+ 93] can be used.

Local suspension analysis is formulated in terms of the confluence semantics [Codish+ 93] for tractability. The basic problem is that the complexity of analysing concurrent programs can quickly become unmanageable. The naïve approach of enumerating every possible interleaving (or scheduling) of the processes of the program is only practical for trivial examples. In contrast, with the confluent semantics, any fair scheduling rule will produce an equivalent set of derivations (modulo the order of clause application). Consequently, only one scheduling policy needs to be considered. A tractable form of local suspension analysis follows because if there are no local suspensions arising from execution with the confluent semantics, none are possible in any scheduling with the standard semantics. Hence, by approximating the confluent semantics, the program can be proved to be local suspension free [Codish+ 93].

Local suspension analysis aims to prove that the program is locally suspension-free. In effect, it is a binary analysis. Either the analysis successfully infers local suspension-freeness, or it fails in the sense that

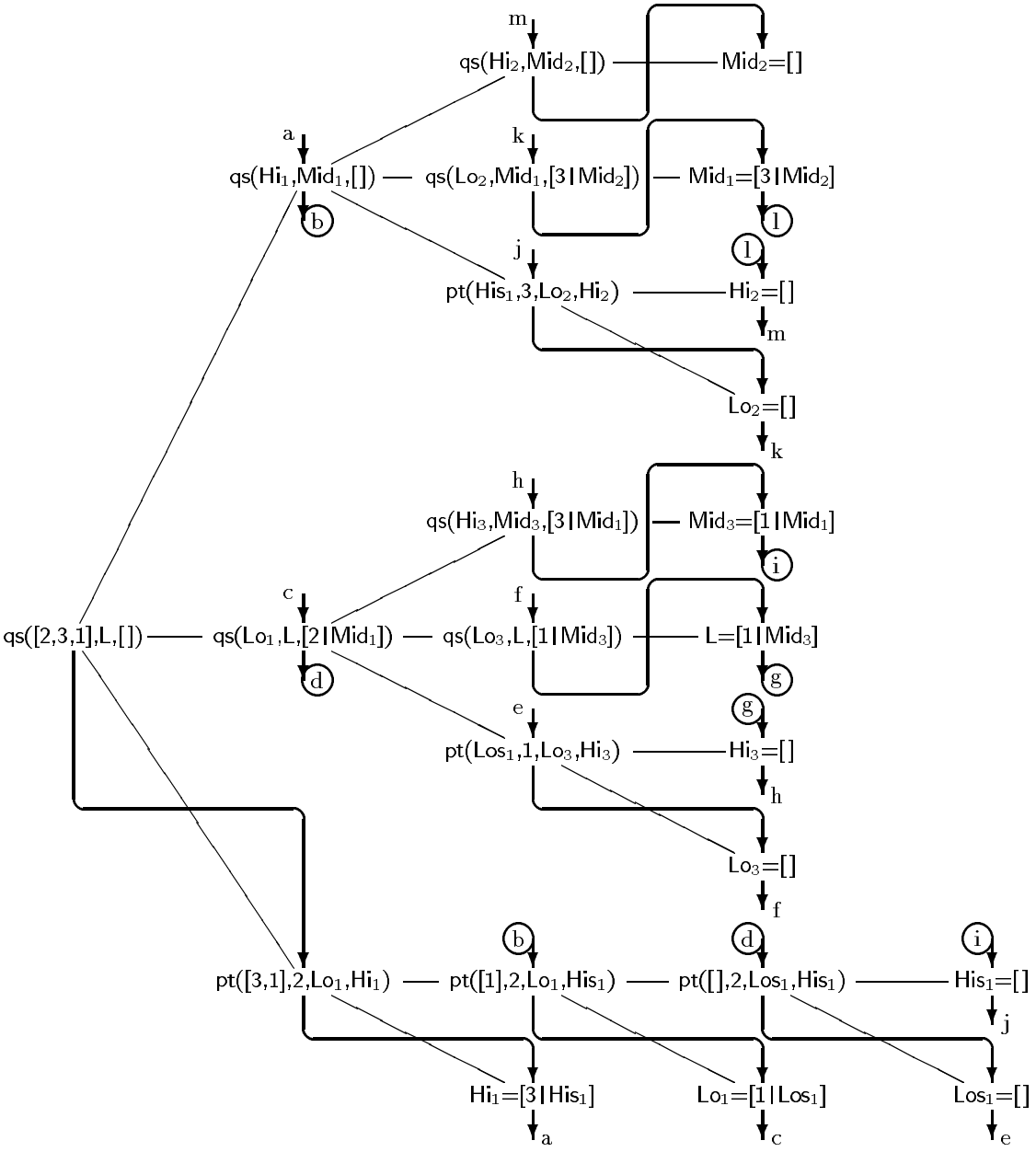

Figure 2: Execution of the goal qs([2,3,1], L, []) under continuation compilation

it can only safely conclude that indefinitely suspended processes may arise. The binary nature of local suspension analysis, however, can impede the efficient introduction of continuations. The optimisation can potentially be applied at many points in the program, not all of which are necessarily safe. A naïve strategy for finding a safely optimised version of the program would be to enumerate and analyse each possible version. This is potentially inefficient. A more efficient approach would be to perform the analyses incrementally as follows. First, the abstract program is analysed, using continuation compilation wherever possible. Specifically, a finite graph of abstract states is constructed with edges representing transitions between states. If local suspension-freeness does not follow from the graph [Codish$^+$ 93], instances of continuation compilation that may have induced the suspensions are identified and removed. The resulting program is then re-analysed. Re-analysing the program is not expensive since large portions of the graph can be reused in the second phase of analysis. This technique is applied iteratively until either all continuations are removed or no local suspensions remain in the graph. In both cases, the resulting program is safe.

To illustrate the basic approach, Figure 2 depicts the execution of the goal qs([2, 3, 1], L, []) for the optimised version of quicksort (introduced in Section 2). Conveniently, here the confluent coincides with the concrete semantics (because quicksort is deterministic [Codish$^+$ 93]). Since no indefinite suspensions occur in the transitions, it follows that the optimisations are safe for the goal qs([2, 3, 1], L, []). In practice, to cover all possible goals, an abstraction of the con-

fluent semantics would be used to finitely prove local suspension-freeness.

4 Abstract Machine beneath Continuation Compilation

To make concrete the actual mechanisms for the optimisations, they are presented as changes to the KL1 abstract machine [Kimura+ 87]. In the KL1 abstract machine, argument and temporary registers (A and X) are actually synonymous, and refer to a fixed vector in the processor's memory. When a process is scheduled the first n registers are restored from the goal record. Thus a process may always assume that Aj contains the j'th argument. Because the optimisations proposed here require a large number of context switches it would not be practical to save and restore these registers. In view of this, for the modified abstract machine, Aj now refers to the j'th argument in the goal record. Thus the A and X register sets are no longer the same and therefore extra instructions have implicitly been defined. For instance the instructions set_value Xj, Gi and set_value Aj, Gi are no longer equivalent. For compatibility, the X registers are initialised from the A registers when a process is scheduled normally, that is, not via a continuation.

There are six new instructions needed to directly support the addition of continuation compilation to the KL1 abstract machine [Kimura+ 87]. The suspend_XXX instructions are always followed by a restore_redo instruction unless the consumer is guaranteed to already be suspended on the continuation structure. The restore_redo would be unnecessary if (say) the producer and consumer were sibling goals, as in the second clause of qs/3.

Continuations are implemented as shared structures. The producer and consumer processes each suspend on the first argument of the continuation structure while waiting for the other process. Since data is passed in abstract machine registers that are not normally saved on suspension, the remainder of the structure is used to save the necessary context for a producer process.

The suspend_goal and suspend_execute_via instructions between them implement the pre-suspension of processes.

proceed_via Ai The register Ai dereferences to a functor of arity $n+1$ that possibly has a (consumer) process suspended on its first argument. Atomically, the pointer to this process is saved and the variable cleared.

If there was no consumer waiting then the current process' code pointer is set to the address of the next instruction and the first n temporary registers X1 ... Xn are saved in the 2nd to $n+1$'th arguments of the functor. Finally, the current process is suspended on the functor's first argument.

Otherwise, the current process is terminated as if by a proceed instruction except that control is immediately passed to the consumer process that was suspended on the first functor argument.

suspend_or_execute_via p, suspArgNo, Ai The register Ai dereferences to a functor of arity $n+1$ that possibly has a (consumer) process suspended on its first argument. Atomically, the pointer to this process is saved and the variable cleared.

If there was no consumer waiting then the current process' code pointer is set to the address of the next instruction and the first n temporary registers X1 ... Xn are saved in the 2nd to $n+1$'th arguments of the functor. Finally, the current process is suspended on the functor's first argument.

Otherwise, the current process' code pointer is set to the code for procedure p. If argument suspArgNo dereferences to a bound variable then the current process is added to the ready queue. If argument suspArgNo dereferences to an unbound term then the current process is suspended on that variable.

Control is then immediately passed to the saved (consumer) process.

suspend_execute_via p, suspArgNo, Ai The register Ai dereferences to a functor of arity $n+1$ that possibly has a (consumer) process suspended on its first argument. Atomically, the pointer to this process is saved and the variable cleared.

If there was no consumer waiting then the current process' code pointer is set to the address of the next instruction and the first n temporary registers are saved in the 2nd to $n+1$'th arguments of the functor. Finally, the current process is suspended on the functor's first argument.

Otherwise, the current process' code pointer is set to the code for procedure p. The argument suspArgNo dereferences to a functor of arity $m+1$. If there is a process suspended on the first argument of that functor, it is added to the ready queue. The current process is then suspended on the first argument of this functor. Control is then immediately passed to the saved (consumer) process.

suspend_goal p, suspArgNo The argument suspArgNo dereferences to a functor of arity $n+1$. If there is a process suspended on the first argument of the functor, that process is added to the ready queue. The newly created process is then suspended on the functor's first argument.

save_suspend Ai The register Ai dereferences to a functor of arity $n+1$. The first n temporary registers, X1 to Xn, are saved into the 2nd to $n+1$'th arguments of the functor. If there is a process suspended on the first argument of the functor, it is added to the ready queue. The current process is then itself suspended on the functor's first argument.

restore_redo Ai, lab The register Ai dereferences to a functor of arity $n+1$. Load the 2nd to

```
pt/4:       wait_functor        c/3,X2,pt/4/3
            integer             X1, psusp
            integer             A2, psusp
            lesser              A2, X1, pt/4/2
L2:         suspend_execute_via pt/4,1,A4
            restore_redo        A4, L2
pt/4/2:     greaterequal        A2, X1, psusp
L3:         suspend_execute_via pt/4, 1, A3
            restore_redo        A3, L3
pt/4/3:     wait_constant       [], X2, psusp
            create_goal         send_nil, 1
            set_value           A4, G1
            enqueue_goal        send_nil/1
L4:         proceed_via         A3
            restore_redo        A3, L4
psusp:      save_suspend        A1
            restore_redo        A1, pt/4
%
% send_nil(Z) :- Z=[].   % Compiler generated
%
send_nil/1: put_constant        [], X2
L5:         proceed_via         A1
            restore_redo        A1, L5
```

Figure 3: Abstract machine instructions for pt/4

> $n+1$'th arguments of the functor into the first n temporary registers, X1 to Xn. Jump to the instruction with label lab.

Figures 3 and 4 show the abstract machine code for the example quicksort program. The automatically generated predicate send_nil/1 is included to allow more than one continuation to be followed from within a single clause body. send_nil/1 is passed a continuation and run as a normal process. When executed, it gives control to the consumer process waiting on the continuation. For example, the third clause of pt/2 has two assignments but a single process cannot jump to two different consumers. The send_nil/1 process deals with the second continuation.

Interfacing issues can be simplified by augmenting the program with auxiliary predicates. The auxiliary predicates reduce code size by providing a uniform mechanism for interfacing between the predicates using continuations and the rest of the program. Without an interface predicate, for example pt/4 would take data from different sources, only some of which necessarily use continuations to pass control. Detailed discussion is omitted here for brevity.

5 Related work

Korsloot and Tick [Korsloot+ 91] have presented some initial ideas on how to introduce sequentiality into concurrent logic programs. Data-dependencies are derived by the mode algorithm of [Ueda+ 90] and [Korsloot+ 91] give several examples of how the data-dependencies can be used to order the atoms of a clause. A more complete description of the mode algorithm is given in [Massey+ 93b]. The procedure for

```
qs/3:       wait_constant       [], X2, qs/3/2
            get_value           A3, A2
            proceed
qs/3/2:     wait_functor        c/3, X2, qsusp
            create_goal         qs, 3
            set_functor         c/3, X4, G1
            write_variable      X5
            write_variable      X5
            write_variable      X5
            set_value           A2, G2
            set_list            G3
            write_value         X1
            write_variable      X5
            suspend_goal        qs, 3, 1
            create_goal         qs, 3
            set_functor         c/3, X6, G1
            write_variable      X7
            write_variable      X7
            write_variable      X7
            set_value           X5, G2
            set_value           A3, G3
            suspend_goal        qs, 3, 1
            create_goal         pt, 4
            set_value           X2, G1
            set_value           X1, G2
            set_value           X4, G3
            set_value           X6, G4
            suspend_goal        pt, 4, 1
            proceed
qsusp:      save_suspend        A1
            restore_redo        A1, qs/3
```

Figure 4: Abstract machine instructions for qs/3

sequentialisation is *ad hoc*, has no supporting theory, and consequently there is no guarantee that deadlock is avoided. Consequently, additional abstract machine machinery has been proposed in [Massey+ 93a] which breaks prematurely deadlocked threads. This contrasts with the approach of the present paper which soundly and systematically introduces threading. Furthermore, continuation compilation generalises the threading framework of [King+ 92] by relaxing the conditions on the producer and consumer processes which can be scheduled at compile-time. Threading is no longer confined to sibling processes but is extended to arbitrary producer and consumer pairs.

Continuation compilation also refines message-orientated scheduling [Ueda+ 90, Ueda+ 92]. Message-orientation optimises storage-intensive programs at the expense of increasing the cost of one-to-many communication. Continuation compilation, on the other hand, applies threading in a more conservative way to avoid any possible performance degradation. Also, to apply message-orientated scheduling, the program has to conform to a moded language subset [Ueda+ 90, Ueda+ 92]. Continuation compilation,

```
% toCont([X | Xs], Ys) :-  % Compiler generated
%    Ys := [X | Zs],
%    toCont(Xs, Zs).
%
toCont/2: wait_constant            [], A1, tc/2/2
          put_value                A1, X2
L0:       proceed_via              A2
          restore_redo             A2, L0

tc/2/2:   wait_list                A1, tcsusp
          read_variable            X1
          read_variable            A1
          put_value                A2, X2
L1:       suspend_or_execute_via   toCont/2,1,A2
          restore_redo             A2, L1

tcsusp:   suspend                  toCont/2

% go(In,Out) :- toCont(In,Cont),qs(Cont,Out,[])
go/2:     put_functor              c/3, X3
          write_variable           X4
          write_variable           X4
          write_variable           X4

          create_goal              qs, 3
          set_value                X3, G1
          set_value                A2, G2
          set_constant             [], G3
          suspend_goal             qs, 3, 1

          create_goal              toCont, 2
          set_value                A1, G1
          set_value                X3, G2
          enqueue_goal             toCont/2
          proceed
```

Figure 5: Abstract machine instructions for the top-level predicate go/2

on the other hand, is applicable to any concurrent logic program. Furthermore, continuation compilation is underpinned by analyses which can be proven correct. In contrast, message-orientated scheduling relies on an *ad hoc* constraint-based mode analysis algorithm.

Continuation compilation also has a lot in common with compile-time garbage collection techniques [Foster+ 91, Sundararajan+ 92], for instance, the analysis machinery which is used to identify producer and consumer pairs. Continuation compilation, however, develops the compile-time garbage collection idea by in addition to recycling the memory, also optimises the control aspects of synchronisation.

A novel approach to threading has been proposed, designed to improve the responsiveness of a wide class of concurrent logic programs. The technique targets the overheads of scheduling, argument copying and increased memory turnover which have traditionally impeded the usefulness of concurrent logic languages like Sandra and RGDC for real-time distributed AI applications. Continuation compilation refines scheduling by optimising process switching for goals which suspend and resume frequently. Argument copying is reduced by passing values in registers. Memory turnover is minimised because communication is made (partially) synchronous and therefore buffering is reduced. Finally, and in contrast to other threading schemes, continuation compilation is underpinned by analyses which can be proven correct. Future work will focus primarily on implementation and benchmarking.

6 References

[Codish+ 93] M. Codish, M. Falaschi, K. Marriott, and W. Winsborough, Efficient Analysis of Concurrent Constraint Logic Programs. In *20th International Colloquium on Automata, Languages, and Programming*, Springer-Verlag, 1993.

[Cohen+ 91] D. Cohen, M. M. Huntbach, and G. A. Ringwood, Logical occam. Technical report, Dept. of Computer Science, Queen Mary and Westfield College, 1991.

[Cousot+ 92] P. Cousot and R. Cousot, Abstract interpretation and application to logic programs. *J. of Logic Programming*, 13(1, 2, 3 and 4):103–179, 1992.

[Elshiewy 90] N. A. Elshiewy, *Sandra: Robust Coordinated Reactive Computing.* PhD thesis, Dept. of Telecommunications and Computer Systems, 1990.

[Foster+ 91] I. Foster and W. Winsborough, Copy avoidance through compile-time analysis and local reuse. In *SLP'91*, pages 455–469, San Diego, USA, 1991. MIT Press.

[Kimura+ 87] Y. Kimura and T. Chikayama, An abstract KL1 machine and its instruction set. In *SLP'87*, pages 468–479, 1987. IEEE Press.

[King+ 92] A. King and P. Soper, Serialisation analysis of concurrent logic programs. In *ALP'92*, Pisa, 1992. Springer-Verlag.

[Korsloot+ 91] M. Korsloot and E. Tick, Sequentializing parallel programs. In *Phoenix Seminar and Workshop on Declarative Programming*, Sasbachwalden, Germany, November (1991). Springer-Verlag.

[Massey+ 93a] B. C. Massey and E. Tick, The diadorra principle: Efficient execution of fine-grain concurrent languages. Technical report, Dept. of Computer Science, University of Oregon, 1993.

[Massey+ 93b] B. C. Massey and E. Tick, Sequentialisation of parallel logic programs with mode analysis. In *LPAR'93*, 1993.

[Sundararajan+ 92] R. Sundararajan, A. V. S. Sastry, and E. Tick, Variable threadedness analysis for concurrent logic programs. In *JICSLP'92*, pages 493–508, Washington, USA, 1992. MIT Press.

[Taylor+ 89] S. Taylor and I. Foster, *Strand: New Concepts in Parallel Programming.* Prentice-Hall, (1989).

[Ueda+ 90] K. Ueda and M. Morita, A new implementation technique for Flat GHC. In *ICLP'90*, pages 3–17, Jerusalem, 1990. MIT Press.

[Ueda+ 92] K. Ueda and M. Morita, Message-orientated parallel implementation of model flat ghc. In *FGCS'92*, pages 799–808, Tokyo, Japan, 1992. Ohmsha Ltd.