No. 18/93

# The Compilation and Execution of Recursion-Parallel Prolog on Shared Memory Multiprocessors

*Thomas Lindgren*

Thesis for the Degree of
Licentiate of Philosophy

ISSN 0283-359X

UPMAIL
Computing Science Department
Uppsala University
Box 311
S-751 05 UPPSALA
Sweden

.

## Abstract

The parallel execution of logic programs, in particular written in Prolog, has great potential for efficient high-level parallel programming. Previous efforts have targeted the parallel execution of conjunctions and disjunctions of programs, so-called And-Or parallelism. A recent, promising approach is that of recursion-parallelism, where structurally recursive programs are exploited to yield large amounts of parallelism.

In this work, we describe techniques for compiling recursion-parallel Prolog programs. We define a restriction of general recursion-parallelism that is suitable for efficient parallel implementation. The compilation of this language consists of compile-time analysis by abstract interpretation followed by compilation.

The analysis is done to ensure that programs satisfy the required properties of efficient implementation and to locate opportunities of optimization. In particular, the analysis distinguishes a hierarchy of data sharing that the compiler attempts to exploit.

Analysis is followed by code generation for recursion parallel predicates *per se*, for predicates that work in a parallel setting (and must consider interactions with other processors when executing) and for standard sequential predicates. Modifications to the instruction set are described, along with the effects of the new instructions.

Experimental results on a range of benchmarks show that the compiler is frequently successful in removing parallelization overheads where unnecessary. This effect seems to stem from the use of locality information, detecting what shared terms are subject to time-dependent operations and the immutability of ground terms.

# Contents

# Introduction

## 1.1 PARALLEL COMPUTING

Increasingly large parallel computers are being constructed as the costs for building faster single-processor systems rise. For instance, in recent years, massively parallel super computers are being sold that consist of hundreds to thousands of general purpose micro processors. It is natural to expect this trend to continue, as the technology to build large-scale parallel machines matures; indeed, workstation vendors are announcing systems of four to thirty-two processors as part of their range of products.

While some principles for building a parallel machine are emerging, it is less clear how to efficiently program a parallel computer. There are several choices to make in order to arrive at a suitable model, and we will attempt to outline them below.

First, one may consider whether to program in a language that explicitly supports parallel processes that communicate through some medium (memory or message channels), whether one should instead use a sequential language and let the underlying programming language implicitly extract parallelism from this program, or whether one should use a concurrent language where operations are understood to be concurrent unless otherwise specified. There are advantages to each approach. In an explicitly parallel language, one has a fine control over how the parallel computation will proceed. The programmer can ensure that the system does 'the right thing' by specifying precisely what to do. On the other hand, writing a parallel program is more difficult than writing a sequential one, because of the potential interactions between processors. The problems are familiar from e.g., operating systems: deadlock, unsafe access to shared resources, starvation and so on. The parallel programmer may also have the responsibility for scheduling the generated tasks in some good or optimum ordering, or processors will spend time waiting for data rather than performing useful work. An implicitly concurrent language has the same problems as an explicitly parallel language, since the problems seem inherent to concurrency.

Programs that address all of these issues can often be highly efficient, but also hard to understand (writing and verifying parallel programs is a black art, though growing increasingly whiter).

A natural thought is then to offload some of these responsibilities onto the run-time system or programming language. There is a spectrum of trade-offs between control over the machine and clarity here: at one end is C with operating system primitives, where the programmer is responsible for everything. At the other end of the spectrum are languages where parallelism is more or less implicit. One of the most successful examples of the latter is Fortran for vector supercomputers, where a compiler recognizes programming idioms that are translated to sequences of parallel instructions, operating over vectors. Other examples include many declarative languages as well as dataflow languages that are based on implicit parallelism. For instance, several projects study how to execute Prolog programs in parallel; there is an active community working on parallel graph reduction that execute lazy functional languages in parallel; and finally, the dataflow languages, represented mainly by Id90 and Sisal, are having an impact on the research community.

In this work, we will attempt to strike a balance between the two extremes of completely explicit and completely implicit parallelism. We believe a programmer must have a clear programming model (or execution model) to write efficient parallel programs, but that the programmer is not interested in the 'boring' aspects of parallel computation. To this end, we somewhat informally partition aspects of concurrency into interesting and boring problems. These are all problems that must be solved for a parallel program to execute.

| Interesting | Boring |
|---|---|
| Scheduling tasks | Avoiding deadlock of computation |
| Studying high-level process interaction | Ensuring global termination |
| Partition problem into tasks | Consistent access to shared data |

We do not mean to imply that the tasks in the right column are intrinsically uninteresting, but rather that the programmer of a parallel machine quite often find them tricky to get right and hard to debug. We consider the left column interesting, since it has to do with the *performance* rather than correctness of the program.

We will study a programming language, Reform Prolog, which provides the programmer with a clear execution model that allows programmers to concentrate on the interesting tasks. Reform Prolog is based on the parallelization of a sequential language, and so avoids the problems of deadlock and global termination, at least to the same extent as sequential languages do. Reform Prolog also ensures consistent access to shared data by use of unification rather than monitors, semaphores and so on. Reform Prolog allows the user to specify processes simply as a list of data to be processed: thus, Reform Prolog is a data-parallel language, which has turned out to be a natural and powerful programming paradigm. Note that Reform Prolog does not use the SIMD (Single Instruction Multiple Data) execution model, which requires synchronization after every instruction, but rather the SPMD (Single Program Multiple Data) model, where all tasks execute the same program with different data, perhaps synchronizing at some program points. Reform Prolog is mainly intended for conventional shared-memory multiprocessors. Tasks interact by modification of shared (single assignment) variables, but follow the sequential semantics. The programmer can thus view the program both as a sequential program and a parallel one (albeit a very tractable one).

The main point of this thesis is not the design of the language, however, but rather its efficient implementation. We will consider compilation techniques that, due to the restrictions of using a high-level declarative language, are quite powerful when compared to similar imperative languages.

## 1.2 PARALLELISM IN PROLOG

We assume that the reader has some familiarity with Prolog and its implementation, and recommend Lloyd [46] for the theory behind logic programming, Sterling and Shapiro [66] for an introduction to Prolog programming and Aït-Kaci [1] for a tutorial on the implementation of Prolog.

From its inception, the potential parallelism of Prolog and logic programs has been recognized. Kowalski [40], Tärnlund [70] and others all proposed that the proof tree be explored in parallel.

The sources of parallelism are twofold. First, in the resolution process any goal in the resolvent may be selected for resolution, and second, resolution may be done against any clause. In principle, nothing stops us from performing several resolution steps simultaneously. Resolving several goals simultaneously is called *and-parallelism*, while simultaneously resolving a goal against several clauses is *or-parallelism* [2, 78, 34, 13, 55] In this thesis, we will concentrate on (dependent) and-parallelism.

Prolog adds impure primitives (i.e. operations that are not referentially transparent), the ability to prune proof trees, input/output and an implicit sequential operational semantics to the logic programming paradigm. A parallel implementation of Prolog must also consider these aspects, which may all act to limit the available parallelism.

## 1.3 PREVIOUS WORK

An and-parallel execution where parallel processes can affect the search space of other processes (e.g., by binding variables shared between processes) is called *dependent and-parallelism.* The main problem in implementing a dependent and-parallel system is to handle the interaction between processes. In particular, when two processes make conflicting bindings to a variable, (local) failure should occur. Backtracking from a failure is a quite complex operation in the general parallel case. Most proposed systems restrict the language to simplify or remove this problem.

A parallel logic programming system exploiting and-parallelism as well as or-parallelism was designed by Conery and Kibler [20], though with impractical overheads.

De Groot [27] and Hermenegildo [35] restricted parallel execution to the case when parallel processes do *not* share variables, by performing runtime tests to determine whether a given conjunction is to be run in parallel or sequentially. When goals do not share variables, they can be executed in parallel. Backtracking can then be managed locally, possibly followed by killing all sibling goals when one goal in a parallel conjunction failed. This approach was called *restricted* or *independent and-parallelism.* Systems that rely solely on independent and-parallelism have been investigated by Hermenegildo [35, 36, 37] and others. In general, the performance is quite good when independent and-parallelism can be exploited. The overhead for parallel execution is mainly related to process management, since processes never share data, and runtime tests for independence, when such tests cannot be eliminated by the compiler.

Subsequent work combined independent and-parallelism with or-parallel [31] and dependent and-parallel [30] execution to extend the scope for parallel execution. The execution engines for the combination proposals are far more complex than a sequential Prolog engine, which is unfortunate if the goal is high absolute speedups with respect to a sequential system.

Yang [81] defined syntactic conditions for programs for parallel execution and found that deterministic computations were well-suited for parallel execution, since conflicting bindings meant that part of the computation failed rather than backtracked. Warren [79] formulated the *Andorra principle* from this idea, where deterministic goals are executed in parallel and nondeterministic goals are suspended. Determinism is detected at runtime. Yang's static method was thus replaced with a dynamic method of detecting and exploiting parallelism. This approach is also called *determinism-driven parallel execution.* An implementation of the Andorra principle, Andorra-I, is described in several papers [56, 57, 58, 82].

Shen [62, 61] has recently developed a method that allows *general* dependent and-parallelism, by maintaining enough information to restart processes that have consumed bindings that are now invalid, and by restricting access to shared variables. The resulting engine executes at about 25% the speed of SICStus Prolog, a standard Prolog implementation, and provides almost transparent parallelization of Prolog programs.

A different approach to binding conflicts was proposed by Naish [53] with PNU-Prolog. By requiring programs to be *binding determinate*, i.e. not undo any bindings during parallel execution, the binding conflict problem was once again reduced to global failure. Binding determinism was a major influence on the Reform execution model, as discussed in Chapter 2.

It should be noted that the research described so far was based on Prolog, a sequential language. Another approach would be to introduce explicit concurrency into the logic paradigm, which was done by Clark and McCabe in IC-Prolog [14] and further developed by Clark and Gregory in their work on the Relational Language [15]. From this school of thought sprang three *concurrent logic languages*, Concurrent Prolog [60], Parlog [16] and Guarded Horn Clauses (GHC) [72]. A restriction of the latter was chosen as the base language of the Japanese Fifth Generation Project. They all had in common that nondeterminism was strictly curtailed by suspending until only one alternative was possible or committing arbitrarily to one alternative when several were available. A consequence was that binding conflicts caused global failure rather than backtracking. (Certain proposed metaprimitives allow programmers to encapsulate a computation to recover from failure.)

The fundamental concept underlying all these efforts is that *parallelism should be extracted automatically from the source program*, and that *the maximum parallelism available is sought.* A consequence is that processes are spawned conditionally due to local tests. Since the behaviour is then in general unpredictable, we call this approach *chaotic parallelism.*

In a chaotically parallel system, processes are created and destroyed dynamically. The grain size (amount of work) of processes varies unpredictably, or is uniformly small (on par with a procedure call). It is hard for the programmer to predict the amount of parallelism or the efficiency of the program; process scheduling is handled by the runtime system and often quite complex, since it is hard to tell the relevance of a given goal to the solution at runtime; in many languages, e.g., the concurrent logic languages, processes are extremely short-lived with lifetimes on the order of a procedure call. Furthermore, the typical case seems to be that most of these processes are suspended during execution, thus being essentially useless for the purpose of parallel speedup. Suspension is so common in concurrent logic langauges that recent implementations optimize this situation rather than computation [73]. Initially, it was thought that compilers could handle this complexity automatically. Some efforts in this direction have appeared recently [39, 47], though the absolute benefits are unclear.

We note that Shen has found that sequential Prolog programs seldom or never exhibit theoretical speedups of three or more orders of magnitude [62, p. 245]. Thus, even the sophisticated implementations described above will not provide unlimited scalability on arbitrary programs – a fairly obvious conclusion. Indeed, Tick [69] devotes much of a book to describing techniques to increase the available parallelism of a given logic program. Additionally, we must consider that the purpose of parallel execution is arguably to speed up a computation. An overly complex underlying execution model will hamper the efficiency of the system by limiting absolute speedups (while maintaining good or excellent speedup relative to the parallel system executing on one processor). The absolute speedup of computations relative to efficient sequential implementations is ultimately the gauge to measure a system's success.

A different approach is to provide the programmer with powerful linguistic constructs to specify large, regular parallel computations. The data-parallel paradigm, where the user specifies a single thread of control that operates simultaneously on many data objects, is suitable for this purpose. For instance, vector supercomputers as well as SIMD supercomputers are based on exploiting the data-parallel principle, while the SPMD model is arguably the most popular programming model for conventional multiprocessors.

Barklund and Millroth [6] constructed Nova Prolog, a data-parallel logic language, which later was generalized into *bounded quantifications* [7]. A bounded quantification expresses some action to be taken over a finite set of elements, which often allows data-parallel execution [4]. Voronkov [75] independently laid the theoretical foundations for bounded quantifications. Using bounded quantifications allowed a concise data parallel approach to logic programming. One of the advantages was the possibility of a simple process structure (e.g., for purposes of scheduling) while still exposing ample parallelism.

We find data-parallelism to be a simple and attractive approach to parallel programming, while still remaining reasonably general. The design of Reform Prolog is thus based on a data-parallel execution model.

## 1.4 RECURSION-PARALLEL LOGIC PROGRAMS

Tärnlund [70, pp. 63-65] noted that the potential parallelism in a clause increases with the number of goals in the clause. By preceding parallel execution with a phase of unfolding, the and-parallelism of a given program could be much larger than readily apparent.

Later, Tärnlund [71] defined an algorithm that applied this process systematically to recursive programs. Tärnlund's idea was essentially to create large clauses matching the query by resolving a clause against itself, creating a new clause which in turn could be resolved against itself, and so on. In this way, a large clause could be created in $\mathcal{O}(\log N)$ steps, for input size $N$. After this unfolding step, the clause was resolved against the query, matching it precisely. The result was a large conjunction amenable to parallel execution.

Millroth used a related method [48]. He noted that a linear structurally recursive clause $H \leftarrow \Phi, R, \Psi$ after $n$ unfolding steps looked like

$$(H \leftarrow \Phi_1, \Phi_2, \ldots, \Phi_n, R, \Psi_n, \Psi_{n-1}, \ldots, \Psi_1)\theta$$

The substitution $\theta$ is the accumulated substitution after the $n$ unfoldings. Millroth also considered the case of nonlinear recursion [50], but that subject is beyond the scope of this thesis.

One way of executing the clause body would be to execute the $\Phi$ conjunctions in a loop from 1 to $n$, then execute the recursive call (matching the base case or both base and recursive case), then execute the $\Psi$ conjunctions from $n$ to 1. The two resulting loops can then be parallelized by running each iteration of a loop as a process, subject to the appropriate synchronization constraints. Such constraints arise from data sharing between loop iterations and our desire to preserve sequential semantics.

Two problems remained: first, how to arrive at the form of the $\Phi$ and $\Psi$ instances without using unfolding? Millroth showed that programs using structural recursion on lists [49] and binary trees [50] could be written on a closed form, using loops and a single, large, compiled head unification that performed *all* the unifications of the unfolding steps, i.e., computed $\theta$ parametrized by $n$, prior to starting the loops. Second, how were the $n$ parallel processes to execute in dependent and-parallel mode? Answering this question lead to the design of Reform Prolog and the work described in this thesis.

## 1.5 LOOP PARALLELIZATION

Extracting parallelism from loops is the most popular method to parallelize imperative programs. There are essentially two methods: vectorization and concurrentization [80]. Vectorization translates program statements into vector instructions for supercomputers, and we will not consider it further. Concurrentization entails adding synchronization primitives to the loop body to ensure correct execution. Synchronization requirements are derived from *dependence analysis*, which determines what loads and stores can interfere. Recently, array dataflow analysis [28] has been developed to deal more precisely with this problem.

The concurrent loop is often called a **doacross** loop, since it conceptually spreads the loop iterations across the processors. There are two issues: scheduling and synchronization. The basic scheduling policies are either to statically assign processes to processors, or to make processors steal work from a shared queue of processes, which is called *self-scheduling* [67]. Since loop bodies often are small, consisting of tens or hundreds of instructions to be executed, various heuristics to reduce contention for the queue have been developed. This takes the form of stealing several consecutive iterations at once. We note that by increasing the process grain, the queue contention overhead can be reduced arbitrarily.

Synchronization is done to satisfy the dependences of later iterations on the values produced by earlier ones, or to ensure that the storage locations used by one iteration are not prematurely read or written by another. Wolfe mentions four main forms of synchronization: random synchronization (by explicit semaphore operations), critical sections, partitioning the loop into a pipeline of statements and using barriers so that all iterations complete one part of the loop before proceeding to the next. Random synchronization is the most general mechanism and can simulate the others. The synchronization method of Reform Prolog (explained in Section 4.8) is reminiscent of random synchronization, but has some differences: all wait-operations are conditional on the binding of the term examined, and no signal-operations are required, this being taken care of implicitly when a process becomes leftmost (or 'first') of all processes.

In the terminology of Pancake [54], the execution model of Reform Prolog employs a reusable thread pool, disallows nested parallelism and synchronizes using unstructured synchronization. Each thread in the thread pool executes the same program when active, and is inactive during sequential execution. We will call this an SPMD-model (Single Program, Multiple Data) with a slight abuse of notation. The execution model is further detailed in Chapter 2.

Automatic parallelization of imperative programs has met with difficulties. For instance, Banerjee et al [3] note: "Dependence analysis in the presence of pointers has been found to be a particularly difficult problem. ...Much work has been done on this problem, though in general it remains unsolved." Reform Prolog parallelizes programs with arbitrary pointer structures; the clean semantics and execution model of the language makes the required analyses simpler than in the imperative case. Singh and Hennessy [63] note that parallelization technology is too limited. In particular, loop parallelism by itself is insufficient to extract enough parallelism from the examined programs. We believe this is in part due to the inability to parallelize loops with procedure calls. Reform Prolog allows procedure calls in parallel procedures and checks statically that they conform to the execution model.

Harrison [33] develops a system that parallelizes recursive calls in Scheme. In particular, the implementation of recursion-parallelism is similar to ours, though presented in a more general context. Harrison performs side-effect and dependence analysis to restructure and parallelize Scheme programs. The main differences of our system are (i) that Prolog relies on side-effects in the form of single-assignment to a much higher degree than Scheme, which necessitates a different treatment from that described by Harrison, and (ii) that Harrison considers only **doall** loops and recurrences, which do not require the explicit insertion of synchronization instructions. The Reform compiler handles general **doacross** loops. Harrison performs a thorough job of restructuring the computation, which could be a useful future extension to our system.

Larus and Hilfinger describe an advanced parallelizing compiler for Lisp, Curare [43], that performs alias analysis prior to restructuring [42]. By computing the program dependence graph of the program, they can extract parallelism from the program. The alias analysis then serves as a data dependence analysis. Reform Prolog also computes an alias analysis, but extracts parallelism less freely. Furthermore, Reform Prolog reasons about locality of data, which Curare apparently does not. Locality information turns out to be very useful when processes are medium or coarse grained. Interestingly, Curare uses a *destination-passing* technique to improve parallelism, reminiscent of logic variables. The destination-passing style means a function takes an extra argument where the result of the function is stored.

It is too early to tell whether the simple form of loop parallelism used by Reform Prolog is sufficient to keep large machines busy when running realistic problems. We cannot say what further measures are required if this form of parallelism should prove to be too scanty. This thesis provides the means for exploring the limitations of simple loop-parallelism for a symbolic language.

## 1.6 THE DESIGN OF THE REFORM PROLOG SYSTEM

As we have seen, previous efforts in implementing parallel logic programming languages have concentrated on exploiting maximum available parallelism. In contrast, one of the goals in implementing Reform Prolog was to achieve a high *absolute* speedup of computations, when compared to a fast sequential system. These two ambitions are not the same. As long as there is enough parallelism to keep the multiprocessor computer executing the program busy, we will consider the amount of parallelism to be large enough. Using recursion-parallelism, the programmer can easily specify arbitrarily large parallel computations (e.g., by varying the size of the data structure operated upon). Rather than attempting to maximize the amount of parallelism, we wanted to execute each process as efficiently as possible. The basic execution engine of our parallel system should thus be as close to an efficient sequential execution engine as possible.

We decided to use an emulated implementation based on the Warren Abstract Machine [77] since the technology is well-proven. The obvious way to reach our goal was then to build upon the WAM instruction set by extending it with appropriate instructions, while avoiding altering the definitions of the original instructions. The compiler would then code 'difficult' constructs using several instructions and reduce these sequences to standard, sequential WAM code as far as possible. In particular, the sources of overhead were to be exposed to compiler optimization.

We note that when a parallel execution engine operates upon data that are not shared with other processes, it can in principle use the sequential operations instead of (slower) operations designed to work in a parallel context. If the compiler can identify local operations, it can emit code for the sequential case. In the best case where no data are actually shared, there is no penalty for parallelization apart from startup overheads of the computation. The main optimization effort of the compiler is to reduce parallel code to sequential code as far as possible. When the compiler can prove that suspension is unnecessary, suspension instructions can be removed. When the compiler can prove that a structure will not be created by a unification or other operation, or that the term unified is local to the process, locking unification instructions can be removed.

The interface between engine and compiler is the instruction set. The Reform Prolog instruction set includes the standard WAM instructions, modified to execute in a parallel context by using 'careful dereferencing', and also includes the necessary extensions to support parallel execution.

The extensions comprise two parts. First, the recursion-parallel execution model requires the ability to perform large head unifications and start many processes efficiently. To this end, we add vector-like instructions to perform $N$ (head) unifications in a single instruction, as well as instructions to set up and execute large numbers of processes concurrently. Second, the code of each process is extended to operate correctly in a parallel setting. In particular, we add explicit instructions to suspend a process in order to maintain sequential semantics and instructions to atomically create compound terms on the heap.

For the engine to remain simple and efficient, we restrict the execution model to encompass only *safe* programs that behave in a disciplined way with shared data. We consider maintaining information on the sharedness of terms at runtime to be a considerable source of expense, and so offload the responsibility of detecting safeness to the compiler.

We require the programmer to adhere to our parallelization conditions and check them at compile time. Thus, the ability to parallelize *all* programs was sacrificed in order to parallelize a useful subset of Prolog with high efficiency.

At this point, the design is separated into two useful components. The engine has the responsibility to perform process scheduling and management, as well as provide an efficient implementation of the instruction set. In the ideal case, each parallel process runs the same code as a sequential WAM and the overhead is minimal. Since many processes are created at once, and a large number of head unifications performed in a single instruction, the cost of creating a process and executing it is very low as compared to more flexible schemes. The result is a parallel engine with low or nonexistent overheads, where the price of parallelization is paid only where required.

The rest of this thesis describes the implementation of the compiler and assesses how close to the goals above we have managed to get. The other side of the coin, the Reform engine, is described and evaluated in Bevemyr's licentiate thesis [8].

## 1.7 WHY SHARED MEMORY?

Multiprocessors are divided into machines with a shared or partitioned address space. We chose to concentrate our efforts on multiprocessors with a shared address space and coherent caches for reasons of simplicity and flexibility. Prolog and other declarative languages frequently use complex data structures involving many pointers. An implementation for a partitioned address space system would thus have to interface two memories by translating all pointers into some address-independent format, send the data stream and convert the message back into pointers at the receiving machine. In a machine with coherent caches, this translation is handled implicitly by the hardware, which is likely to be more efficient than various software translations.

Furthermore, since today's partitioned address space machines also seem to require explicit procedure calls to send data between machines, there is additional overhead when transmitting a message. If messages are frequent, there will be a correspondingly larger overhead. Also, the execution engine must be prepared to handle references to remote data, which requires extra machinery and a possible slow-down of execution.

A more fundamental argument against the use of a shared address space is that previous shared-memory multiprocessors have had limited scalability, since the coherence protocols were based on broadcasts over buses (so-called snoopy caches). With the advent of directory-based cache coherence protocols as used in KSR-1 [12], DASH [45] and the Data Diffusion Machine [32], this is no longer an objection. The Scalable Coherent Interface, an IEEE standard for cache coherence, provides scalable directory-based cache coherence up to $2^{16}$ processors.

## 1.8 THESIS CONTRIBUTIONS

The contributions of this thesis are the following:

- We develop the first compiler for a recursion-parallel language.
- We define an efficient execution model for recursion-parallel Prolog, Reform Prolog.
- We define and use locality and safeness analysis to enable and optimize parallel execution. This includes identifying operations that cause dependences to retain sequential semantics.
- We describe compilation techniques for manipulating shared and time-dependent data, and several optimizations of this process.
- We show how to detect and compile parallel prefix operations such as reductions and scans in Reform Prolog.

Parts of this thesis, the execution model and instruction set, is joint work with Johan Bevemyr and Håkan Millroth. Parts of the thesis have been published previously [9, 10].

## 1.9 OVERVIEW OF COMPILER AND THESIS ORGANIZATION

When reading the rest of this thesis, it is useful to know the large-scale organization of the compiler. Compilation has five phases.

**Loading.** The program to be compiled is loaded into a database. Syntactic sugar is expanded away, leaving Horn clauses and reduction operations in parallel predicates are found.

**Analysis.** The program is analysed with respect to the defined export interfaces. This analysis is described in Chapter 3 and involves proving safeness of the computation, a fundamental concept in Reform Prolog, defined in Chapter 2, as well as inferring computations involving only data local to a process and types. If safeness cannot be proved, parallelization fails and sequential code is emitted.

**Annotation.** The results of the analysis are entered into the program. The annotator marks every variable occurence in every program clause with pertinent information. In this process, the annotator may also duplicate certain clauses due to safeness restrictions, and remove clauses found to be unreachable due to types inferred by the analysis phase. This phase is straightforward and is not described in the thesis.

**Code generation.** The annotated program is compiled into code for the Reform engine. Predicates are partitioned into three classes, sequential, recursion parallel and sequential-in-parallel-context. Sequential predicates are compiled as for a WAM. Recursion parallel predicates are compiled into code that performs large head unifications and starts many processes at once. Each such process is a conjunction of sequential calls, executed in a parallel context. The sequential predicates executed under a parallel call form the third category. Compilation of such predicates includes managing suspension and atomic data creation.

The thesis is organized as follows: In Chapter 2, the execution model is described and Reform Prolog defined. Chapter 3 describes the global analysis done as part of the compilation. Chapter 4 introduces the instruction set of the engine and describes the compilation techniques. Chapter 5 experimentally evaluates the compiler, and Chapter 6 concludes the thesis with a summary, future work and a comparison to parallelization of imperative languages.

# RECURSION-PARALLEL EXECUTION MODELS

BEFORE tackling the implementation issues of Reform Prolog, we shall discuss the language design per se. We define a restriction of Prolog as suitable for efficient parallel execution and discuss some alternatives.

The execution model was designed with Håkan Millroth and Johan Bevemyr.

## 2.1 BASIC CONCEPTS

A *recursion-parallel predicate* is a predicate defined by two clauses, the base case and the recursive case, by structural recursion. Reform Prolog recognizes in particular two cases as candidates for parallel execution: recursion over lists and integers. Millroth's Reform compilation also handles binary trees [50], and can be extended to handle other cases of structural recursion [51].

### 2.1.1 Interesting structural recursion

We take $\Upsilon, \Phi$ and $\Psi$ to be conjunctions of goals, excluding cut. For our purposes, the two interesting cases of structural recursion are then defined as follows. The first case is *list recursion.*

$$\begin{array}{rl} p([], \overline{Y_0}) & \leftarrow \Upsilon \\ p([X|Xs], \overline{Y}) & \leftarrow \Phi \wedge p(Xs, \overline{Z}) \wedge \Psi \end{array}$$

The second case is *integer recursion.*

$$\begin{array}{rl} p(0, \overline{Y_0}) & \leftarrow \Upsilon \\ p(N+1, \overline{Y}) & \leftarrow \Phi \wedge p(N, \overline{Z}) \wedge \Psi \end{array}$$

We say that $\Phi$ is the *left body* of the recursive clause (or predicate) while $\Psi$ is the *right body*. Each instance $\Phi$ or $\Psi$ is called a *recursion level*. We will, with an abuse of notation, also use recursion level as a synonym for 'process' subsequently.

We also note that any argument can be structurally recursed upon. For simplicity, we only consider the first argument of a clause as a candidate for detecting recursion-parallelism.

### 2.1.2 Relating parallel goals

In Prolog, goals are executed in a left-to-right order. When executing in parallel, we will employ this order to resolve conflicts and suspend goals. To this end, we say that in a conjunction $A \wedge G \wedge B$, where $A$ and $B$ are conjunctions of goals and $G$ is a single goal, that the goals in $A$ are *to the left of* $G$, or *precedessors of* $G$. Likewise, the goals in $B$ are *to the right of* $G$, or *successors of* $G$. If $A$ is empty, $G$ is said to be the *leftmost* goal; if $B$ is empty, $G$ is *rightmost*.

### 2.1.3 Variable classification

The structurally recursive form we have defined above leads to a simple classification of variables for purposes of analysis and compilation. The intuitive meaning is that a POSLIST argument destructures a call argument by successive recursive calls, a NEGLIST argument constructs an argument, INV variables are invariant while a NONE-NEG argument changes with each call without being one of the above.

Our interest in these classifications is that argument pairs that fit into one of the classes can be represented efficiently during parallel execution, as explained in the chapter on compilation. We use the variable classes during compilation as well as analysis. In the rest of this thesis, we will only fleetingly touch upon the POSLIST-K and NEGLIST-K classes, since they are handled similarly to their simpler counterparts.

Given a structurally recursive predicate on one of the two forms defined above, with the recursive clause:

$$p(X_1, \ldots, X_n) \leftarrow \Phi \wedge p(Y_1, \ldots, Y_n) \wedge \Psi$$

We say that a variable $A$ in argument $i$ occurs for the first time if there is no argument $j, j < i$ where $A$ occurs. For each pair of arguments in the head and recursive call of the clause, $X_i$ and $Y_i$, we define the pair to be:

POSLIST If $X_i = [X|Xs]$ and $Y_i = Xs$. and $X, Xs$ are variables, $Xs$ occurring for the first time. $X$ may have occured previously as the head of a POSLIST.

NEGLIST If $X_i = Xs$ and $Y_i = [X|Xs]$ and $X, Xs$ are variables, $Xs$ occurring for the first time. $X$ may have occurred previously as the head of a NEGLIST.

POSLIST-K If $X_i = [X_1, \ldots, X_k|Xs]$ and $Y_i = Xs$ and $X_1, \ldots, X_k, Xs$ all are variables occurring for the first time.

NEGLIST-K If $X_i = Xs$ and $Y_i = [X_1, \ldots, X_k|Xs]$ and $X_1, \ldots, X_k, Xs$ all are variables occurring for the first time.

INV If $X_i = Y_i$ and $X_i$ is a variable.

NONE-NEG If $X_i$ and $Y_i$ are distinct variables occurring for the first time.

The other cases are handled below. These classifications are further justified in Millroth's thesis [48]. Note the restrictions on previous occurrences of variables and the limited forms of terms occurring in the head or recursive call. Variables may occur arbitrarily in the left and right bodies, and need not occur in the head or recursive call. If not, the variable is local to the recursion level.

We say that in a POSLIST $[X|Xs]$ to $Xs$, the variable $X$ is the *head variable* of the argument, while $Xs$ is the *tail variable*. The same definitions are used for the NEGLIST classification.

Obviously, some arguments do not satisfy these classifications and are then rewritten, if possible, to fit into one of the above. For instance:

$$\begin{aligned} p([a|X]) \leftarrow p(X) &\Rightarrow p([A|X]) \leftarrow A = a, p(X) \\ p([Y,Y|X]) \leftarrow p(X) &\Rightarrow p([Y,Z|X]) \leftarrow Y = Z, p(X) \\ p(f(X)) \leftarrow p(g(Y)) &\Rightarrow p(A) \leftarrow A = f(X), B = g(X), p(B) \end{aligned}$$

Some clauses require further transformation, notably where the tails of lists are unified (e.g., $p([A|X],[B|X]) \leftarrow p(X,X)$). Currently, the compiler gives up and compiles such predicates for sequential execution. Other clauses are not obviously structurally recursive, and are not considered candidates for parallelization.

## 2.2 RECURSION-PARALLELISM

So where is the parallelism? We will exploit dependent and-parallelism, so the amount of parallelism available at a given moment roughly depends on the size of the resolvent of the program at that time.

If we assume a finite-sized machine, there is an upper limit on the number of processes where parallelism is sufficient to keep the machine busy – for instance, when the number of processes available exceed the number of processors by some factor. Beyond this point, the machine can at the moment not run the extra processes, but must resort to scheduling them between processors. Creating too many processes can thus reduce performance by consuming time and space resources.

In parallel Prolog, the amount of (and-parallel) work is roughly the number of goals in the resolvent. The size of the resolvent changes as processors perform resolution steps. Until the state of sufficient parallelism is found (and maintained), the machine will be underutilized as processors will be idle.

Our solution to these problems is the following:

1. Extract sufficient parallelism but avoid creating superfluous tasks (e.g. when the machine is busy).
2. Attempt to enter the state of full utilization as quickly as possible, and subsequently attempt to remain in this state until finished.

These goals are realized by recursion-parallelism and Reform compilation. We will first take an abstract view of creating processes and executing them. Consider a specific query:

$$p(T, Y_1, \ldots, Y_n)$$

If $p/n$ is structurally recursive on the form defined in Section 2.1, we create a new clause by unfolding the recursive clause of the recursive clause until the head of the new clause matches the query as long as unfolding At this point, we have a new clause:

$$(p(T', Y_1', \ldots, Y_n') \leftarrow \Phi_1, \Phi_2, \ldots, \Phi_m, p(T'', Y_1'', \ldots, Y_n''), \Psi_m, \Psi_{m-1}, \ldots, \Psi_1)\theta$$

The substitution $\theta$ is the accumulated result of the $m$ unfoldings As we will see later, the variable classes defined above can be used to compute $\theta$ while avoiding actually to perform the unfolding operations. This was originally shown by Millroth [48].

We resolve the query and the new clause, and arrive at a large conjunction of goals. A Prolog system can now exploit parallelism by executing the instances of $\Phi$ and/or $\Psi$ in parallel.

1. Perform a large head unification simulating the $m$ unfoldings constructing the goals to be executed.
2. Execute all the instances of $\Phi$ in a loop from 1 to $m$. This loop can be executed in parallel.
3. Execute the single remaining call to $p/n$, which matches the base clause and possibly also the recursive clause (e.g. if a list ends in a variable, both clauses may match that variable).
4. Finally, execute all instances of $\Psi$ in a loop from $m$ to 1. This loop can also be executed in parallel.

Thus, our system creates a bounded number of processes, which can be adjusted by the user by increasing input data appropriately. Note that the user could conceivably define too many processes, so that the system suffers from lack of space. This effect can be avoided by simple program transformations, either done automatically or by the programmer, e.g. by splitting a large data set into several smaller parts and repeatedly calling the parallel predicate. Currently, such a transformation must be done manually.

Since $m$ processes are created in a single step, the size of the resolvent increases quickly. This means that all processors can become active once the head unification has been performed, under the condition that $m$ is sufficiently large. One might argue that the large head unification could become a sequential bottleneck; we will address that issue in Chapter 4. Ideally, the number of iterations $m$ should be much larger than the number of processors, since the system then stays in the parallel loop longer than otherwise. If the number of iterations is less than the number of processors, some processors will be idle.

If recursion levels vary widely in the amount of work required to complete them, then the system can prevent load imbalances if the number of iterations is large, but not so well if it is small. The least 'unit of parallelism' of the system is the recursion level. If each individual recursion level is large, i.e. takes a long time to compute, then the system will be kept busy for longer.

Finally, note that the execution model does not provide any parallelism beyond recursion-parallelism. If the program does not contain predicates amenable to Reform compilation, it will not be parallelized. The fraction of time spent outside of recursion-parallel predicates ultimately limits the possible speedup, according to Amdahl's law.

Our experience so far is that many predicates can be parallelized as above, or be simply rewritten for this purpose. Furthermore, as we have argued in the introduction, parallel programs should be developed as such from the start. Hence, we can rely on the programmer attempting to follow our format as far as possible. It remains to be seen if the class of parallelizable programs is large enough, or if this simple class requires extensions to be practical.

**Example.** We give a predicate that can be executed in parallel. Note that the example is 'difficult' for conventional forms of parallelism, since it contains little or-parallelism and the and-parallelism increases slowly (one goal is added per resolution step). The predicate is list recursive.

```
nrev([],[]).
nrev([X|Xs],Zs) :- nrev(Xs,Ys), append(Ys,[X],Zs).
```

The large head unification of $nrev([1,2,3,4], RevList)$ creates the following goals:

$$\begin{array}{l} nrev([], Ys_4), \\ append(Ys_4, [4], Ys_3), \\ append(Ys_3, [3], Ys_2), \\ append(Ys_2, [2], Ys_1), \\ append(Ys_1, [1], RevList) \end{array}$$

Each of the append/3 goals is a separate process, all starting simultaneously after the sequential nrev/2 call finishes. Since the append/3 goals share variables, this is dependent and-parallel execution. □

## 2.3 SAFENESS

The execution model allows quite general dependent and-parallelism, and we will restrict this generality in order to arrive at an efficient and useful programming language.

Our approach is inspired by the PNU-Prolog language designed by Naish [53]. If we can ensure that bindings made to variables shared between processes never will be undone during parallel execution, then we are in a good position: if one process fails, then no other process can provide new bindings for a shared variable by backtracking (in parallel), so the parallel computation must fail. If this was not the case, the system would have to manage interprocess backtracking with all its attendant complexity. Naish describes the use of delay declarations and disciplined programming to adhere to this condition, which he calls *binding determinism.*

Naish's PNU-Prolog executes all subcomputations below a parallel call in parallel, extracting the maximum amount of parallelism from that call. We diverge from Naish at this point; we consider the amount of parallelism found by the recursion-parallel execution model to be quite adequate and execute all non-recursion-parallel goals sequentially. Because of this decision, we can lift some of the restrictions that Naish puts on PNU-Prolog programs.

We first define a variable binding to be *unconditional* w.r.t. a call if it is never undone by backtracking in that call. A variable is *shared* if it is accessible from two or more processes. A *safe parallel call* is then defined as:

**A parallel call is safe if all bindings to shared variables during parallel execution are unconditional with respect to the parallel call**

We parallelize recursive programs where all parallel calls are safe. Note that there may be unshared variables, namely, those created by the execution of each sequential goal. These may be bound conditionally, and backtracking may occur locally in a single goal *as far as no shared variable is reset.* Our approach thus permits some nondeterminism beyond that permitted by the proposal of Naish. On the other hand, we must somehow distinguish between shared and unshared variables. This can be done at runtime or at compile-time. We have chosen the latter and approximate the set of shared variables conservatively at compile-time. This estimation is described in Chapter 3.

What if one of the goals executed sequentially is a call to a recursion-parallel predicate? We may elect to execute it using the recursion-parallel principle, or not. In the former case, we say that the system supports *nested* recursion-parallelism; in the latter case, there is only *flat* recursion-parallelism.

Currently, our implementation supports only flat recursion parallelism. This decision implies that all processes are created prior to parallel execution and scheduling then becomes straightforward. In other parallel models of executing Prolog, scheduling can be a major problem, since it is not at all clear what process is best run next. In Reform Prolog, the next process to be executed is simply the next iteration of the parallel loop (i.e., recursion level).

**Example (cont).** Let us return to the nrev/2 example, armed with the concept of safeness. Since the first argument of the last append/3 three goals is a variable, executing the goals would create choicepoints and then bind argument 1 of each append/3 goal to [ ]. But doing so would violate safeness, and so these recursion levels suspend. So only the first append/3 goal can run, producing a binding for $Ys_4$. At this point, the second goal is activated. When the second goal binds its output variable, the third goal is activated. This leads to an overlapped, or pipelined, parallel computation. Safeness thus requires recursion levels to suspend on occasion. □

We note that, so far, processes only suspend when they are about to bind variables conditionally. If a preceding process has bound the variable already, no suspension is required. In a pure, safe program without time-sensitive operations, a recursion level only suspends on clause indexing.

## 2.4 HANDLING TIME-SENSITIVE OPERATIONS

We wish to run a practical subset of Prolog in parallel, not just logically pure programs. Prolog also includes impure constructs such as cuts, type tests, directed arithmetic and so on. These operations are desirable for a practical programming language, and so Reform Prolog must handle them. However, if impure operations are incorporated into the language, a parallel computation may violate sequential semantics even if the computation is safe.

**Example.**

```
rp([],[]).
rp([X|Xs],[Y|Ys]) :- p(X,Y), rp(Xs).

p(a,X) :- q(X).
p(b,c).

q(X) :- var(X),!,X = b.
q(c).
```

Given the call rp([a,b],[Y,Y]), the system spawns two recursion levels, p(a,Y) and p(b,Y). Both are deterministic and, hence, safe. (Bindings to Y are unconditional.) Now assume p(b,Y) is run before p(a,Y), and binds Y to c. Then p(a,c) will reduce to q(c) which succeeds.

Sequential Prolog would have a quite different behaviour: first, p(a,Y) reduces to q(Y) and in turn to Y = b. Then p(b,b) fails. Hence, the parallel execution model as described previously is unsound with respect to sequential Prolog execution.

The problem is that Y is sensitive to bindings out of the sequential order. Even if a recursion level is safe, binding Y would result in incorrect behaviour. The concept of safeness must be amended by special treatment of variables subject to time-dependent tests. □

When a variable subject to time-dependent operations may be further instantiated, the recursion level must suspend until the term has been instantiated by preceding recursion levels, or the recursion level becomes leftmost in the resolvent. At that point, it may go ahead and perform the operation, which now may instantiate the term if required (since the same operation would be done at the same point in a sequential computation).

We call terms containing only variables local to the recursion level *local terms*. Terms containing variables shared between recursion levels are called *shared terms*, and terms containing shared variables subject to time-dependent operations during parallel execution are called *fragile terms* (since they must be handled with care). Shared terms that do not contain fragile variables are called *robust terms*.

### 2.4.1 Resolving safeness at runtime

A recursion level might conditionally violate safeness during execution. If a recursion level is nondeterminate and unifies a shared term with a *possibly* nonvariable term, then safeness is violated if the term is an unbound variable (since the binding would be conditional), but not necessarily if the term is nonvariable (there may be no bindings made, only tests on the shape of the term).

One solution would be to reject parallelization of all predicates where such a situation occurs. However, it is easy to handle some of these cases at runtime instead. When safeness might be violated, the recursion level is suspended. If another recursion level further instantiates the term, the recursion level is safe and can proceed. When the recursion level becomes leftmost in the resolvent, it is supposed to bind the variable. This binding would be conditional, and the entire computation instead aborts with a runtime error message. Such suspension for safeness is done at all points of a unification where safeness may be violated. This is the solution chosen by Reform Prolog.

### 2.4.2 Constructs that disallow parallelization

Finally, we restrict parallelization to exclude predicates where there is I/O (since that would require global synchronization to retain Prolog's observable behaviour), assert/retract (which may require global synchronization as well) and metacalls such as call/1, findall/3 and others (since the system cannot generally prove them safe). Some of these restrictions can be lifted, e.g., completely disallowing predicates requiring global synchronization is unnecessarily severe at present, or disallowing higher-order constructs even when only local data are involved, and no nondeterminism is introduced. However, doing so would have made the design more complex without adding to our basic understanding of the issues. Future work may include removing some of the restrictions.

# Analysing recursion parallel programs

This chapter explains the use of global analysis in the compiler, provides an overview of the abstract domain used in the analysis and describes the analysis algorithm for both sequential and parallel execution.

## 3.1 STATIC PROPERTIES OF INTEREST

The execution model of Reform Prolog requires that parallel computations be safe and that sequential semantics is retained. The Reform compiler proves safeness through global analysis of the program. Furthermore, fragile terms must be identified to retain sequential semantics.

Even for a safe computation, there are two sources of overhead when executing a program in parallel rather than sequentially. First, the engine might suspend when manipulating fragile terms, which requires extra tests. Second, locking unification must be used to bind shared variables to compound terms.

Lacking information to the contrary, the compiler would have to assume the worst case: that all variables are fragile, and that all compound terms must be created using locking unification. The actual case is often different: most of the data is actually local to a computation and time-dependences comparatively few. The compiler eliminates suspensions and locking unifications based on determinacy, type and locality information.

### 3.1.1 Ensuring safe computations

The compiler can prove that a program is safe given information on types of variables, determinacy and locality. Locality information restricts the unifications to be checked to a subset; type information tells the compiler when terms may be instantiated and determinacy information shows where such instantiation may be unsafe.

### 3.1.2 Eliminating the overheads of parallelization

To preserve sequential semantics, fragile terms must not be instantiated out of the sequential order. At runtime, a Reform engine recognizes two states: the recursion level being executed is leftmost or non-leftmost. Fragile terms may only be bound when leftmost, while other terms can be bound at any time. (Thus, robust terms can be bound out of the sequential order; since by definition they are not subject to time-dependent operations, such as clause indexing, the optimization is safe.)

If the compiler would classify all shared terms as potentially fragile, programs would be unnecessarily sequential. No shared variable can then be bound out of order, which would hamper, e.g., programs that incrementally construct a data structure (such as difference list programs) over several recursion levels. This is completely unnecessary, since the sequential semantics need not change if a shared variable is bound out of the sequential order.

A second source of overhead is the requirement to use locking unification. The compiler can reduce such unifications to the sequential case when the unification can be proven to be a test (through type information), or the involved terms are local.

## 3.2 AN ABSTRACT DOMAIN FOR TYPES, LOCALITY AND SAFENESS

We briefly introduce global analysis by abstract interpretation and describe an abstract domain capturing the properties that interest us.

**Abstract interpretation.** Static analysis of a programming language can be expressed as an *abstract interpretation* of the formal semantics of the language, as proposed by Cousot and Cousot [21, 22]. The abstract interpretation of a program yields a conservative approximation of the runtime results. If properly defined, the abstract interpretation procedure is guaranteed to terminate. To define a static analysis by abstract interpretation, we roughly perform the following steps:

(i) Define a collecting fixpoint semantics for the language that collects the transitions and intermediate states from some set of initial states to some set of final states.

(ii) Define an abstract domain that expresses the properties we are interested in. The abstract domain should be a lattice. The abstract semantics must be defined to be terminating but can often be derived from the concrete semantics.

(iii) Define the concrete and abstract operations of the language. In logic programming, this is typically unification. Prolog adds other primitive operations.

(iv) Relate the concrete domain $D$ and abstract domain $D'$ by means of a Galois surjection, a pair of functions $\alpha : D \rightarrow D'$ and $\gamma : D' \rightarrow D$ such that $\alpha(\gamma(x')) = x'$ and $x \leq \gamma(\alpha(x))$, where $\leq$ is the ordering operation of $D$. The function $\alpha$ is called the abstraction function, while $\gamma$ is the concretization function. Prove the correctness of abstract operations using this connection.

(v) Prove termination by showing that the abstract operations are monotonic, and the abstract domain has no infinite ascending chains. (Infinite chains can be accomodated by using widening and narrowing operations [22], but we will not consider that approach further.)

Cousot and Cousot have shown that all relevant properties of the semantics are conservatively approximated if the domains are connected by a Galois surjection. Thus, for the analysis designer it remains to be constructed an appropriate semantics and an appropriate abstract domain.

For logic programming, there have been several proposals for abstract interpretation algorithms based on top-down and bottom-up semantics, or even their combination. We have chosen the framework proposed by Debray [25, 24], since it very clearly approximates a standard interpreter for Prolog. Since we analyse full Prolog, including cut, metaprimitives and higher-order operations, we felt the semantics should be fairly low-level and close to the way Prolog actually executes.

The abstract semantics yields a set of equations to be solved, either statically or iteratively. The framework of Debray performs a sequence of chaotic iterations [22] to arrive at a fixpoint.

We will be quite informal when defining our analysis. We will define an abstract domain that captures several properties that are interesting when executing Reform Prolog but refrain from formal proofs of correctness. We also describe Debray's analysis algorithm and our modifications to support analysis of parallel execution (Section 3.3).

**Basic concepts.** Debray's analysis algorithm operates on two objects: (call and success) patterns, and instantiation states (i-states). Patterns summarize information on calling or exiting a clause or predicate, while i-states represent the possible substitutions before or after a call. It is the task of the analysis designer to provide a set of operations on patterns and i-states, as defined below.

- An *abstraction function* $\alpha$ that takes a set of concrete calls and returns an abstract call summarizing the information, and a *concretization function* $\gamma$

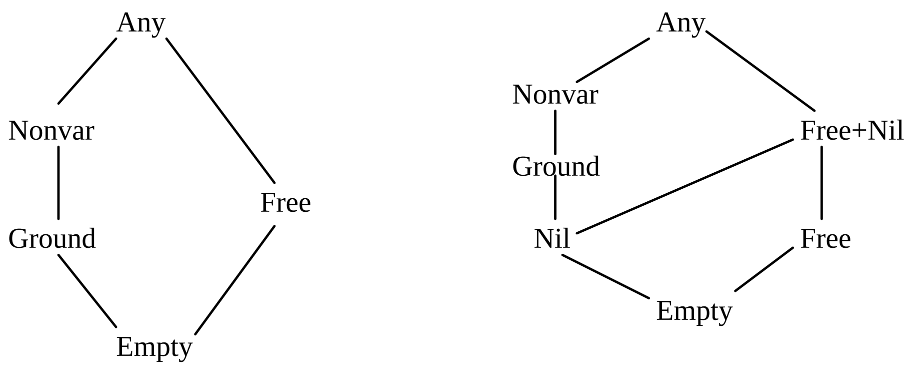

Figure 3.1: The Debray-Warren type domain (left) and a simplified version of the Reform compiler type domain

that takes an element of the abstract domain and returns the set of calls, substitutions or terms corresponding to it.

- A *least upper bound* (lub) operation $\sqcup$, that takes two patterns and constructs a new pattern that is the least upper bound of the two. Lub is used to summarize the information from several control paths at a merge point (e.g., after returning from a call.)
- A comparison operation $\sqsubseteq$ over patterns. This operation is used to find where abstract execution can be cut off, by using a previously tabulated answer. Furthermore, it is used to tell when a tabulated success pattern changes; if call and success patterns are different, this is in reality two operations.

The analysis framework also requires several operations that are described in Section 3.3. These are relatively straightforwardly defined from the descriptions below.

We will mainly be concerned with defining the patterns and i-states that are operated upon. Both these classes of objects are composed from several domains that represent types and constraints on variables.

### 3.2.1 Types

The type domain of the Reform compiler is an extension of the domain described by Debray and Warren [26, 24]. The Debray-Warren domain and a simplification of the domain used in the Reform compiler (without recursive list types) are shown in Figure 3.1. The full type domain of the Reform compiler, including difference list handling, is shown in Figure 3.2.

We take GROUND to be the set of ground terms, VAR to be the set of variables, FUN to be the set of function symbols, CONST the set of constants, and TERM

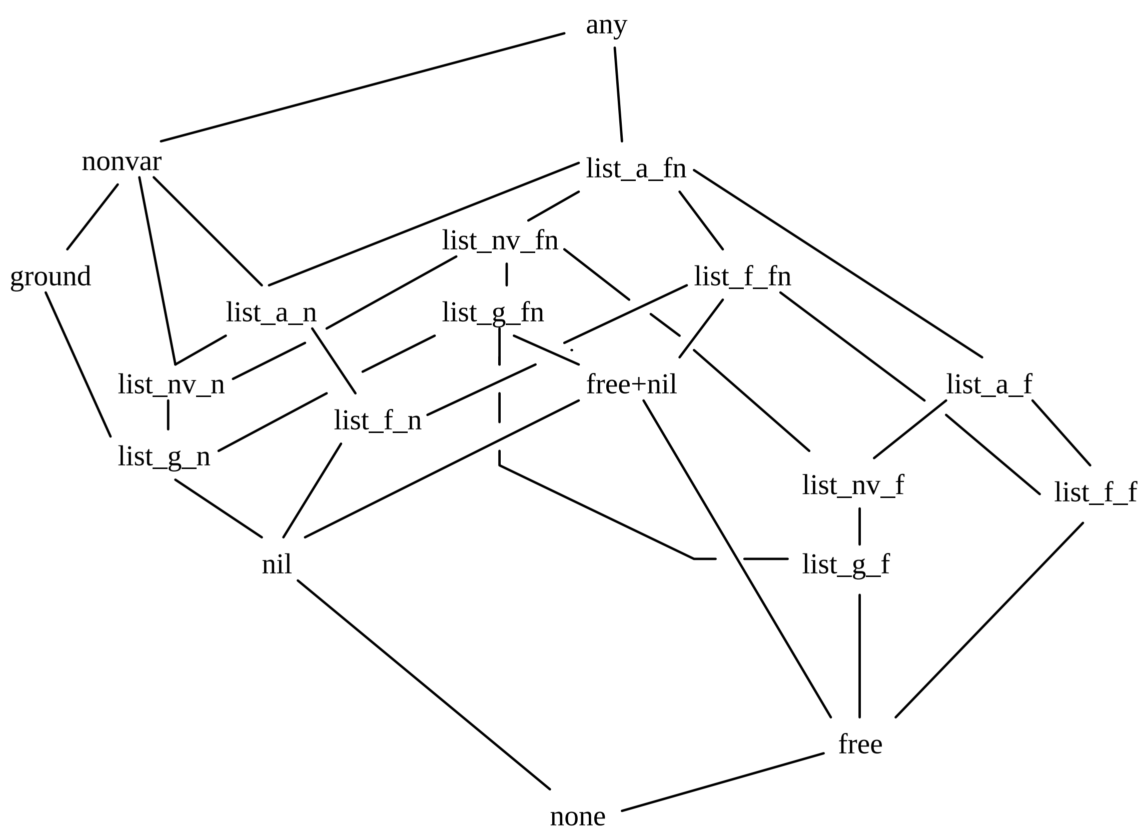

Figure 3.2: The full type inference domain TYPE

to be the set of terms. We restrict the list types to only represent lists, where list elements are one of **ground**, **nonvar**, **free** or **any** and the final tail of the list is **free**, **free+nil** or **nil**. We write list types as **list_{g,nv,f,a}_{f,n,fn}**.

We say that a list terminated with a free variable or the empty list is a *well-formed list*, while a 'list' terminated by some other term is not distinguished from other compound terms. A list terminated with the empty list is called a *closed list*, while other well-formed lists are *open lists*. In the same way, we speak of open and closed list types. The closed lists abstract to **list_$a$_n**, while open lists abstract to **list_$a$_{f,fn}**. Lists are mapped to types **nonvar** or **ground** if they are not terminated with **free**, **free+nil** or **nil** types. The restriction makes the type domain finite.

Concretization $\gamma : \mathrm{TYPE} \rightarrow \wp(\mathrm{TERM})$ of abstract values maps an abstract value to the set of concrete terms it represents.

$$\begin{array}{rcl}
\gamma(\mathbf{none}) & = & \emptyset \\
\gamma(\mathbf{nil}) & = & \{[]\} \\
\gamma(\mathbf{ground}) & = & \mathit{GROUND} \\
\gamma(\mathbf{nonvar}) & = & \{f(x_1,\ldots,x_n) \mid x_i \in \mathit{TERM}, f^n \in \mathit{FUN}\} \\
\gamma(\mathbf{any}) & = & \mathit{TERM} \\
\gamma(\mathbf{free}) & = & \mathit{VAR} \\
\gamma(\mathbf{free{+}nil}) & = & \gamma(\mathbf{free}) \cup \gamma(\mathbf{nil}) \\
\gamma(\mathbf{list\_}a\mathbf{\_}b) & = & \{[x_1,\ldots,x_k|xs] \mid k \geq 0, x_i \in \gamma(a), xs \in \gamma(b)\}
\end{array}$$

Abstraction $\alpha : \wp(\mathit{TERM}) \rightarrow \mathit{TYPE}$ can be defined from the concretization function.

**Aliasing.** We say that two terms are *aliased* when they share free variables. (Equivalently, they are said to *share*.) If instantiating one of the terms *must* bind the other to the same value, they are *certainly aliased*. If instantiating one of the term *may* instantiate the other, the terms are *possibly aliased*.

Assuming that the concretization of an abstract element is a set of terms, we say that an abstract domain is *substitution-closed* [25] when every element $T$ satisfies

$$\{x\theta \mid x \in \gamma(T)\} \subseteq \gamma(T), \text{for any substitution } \theta$$

Thus, all instantiations of a term belonging to the concretization of an abstract element also belong to the concretization of the same abstract element. This property allows us to ignore the effects of aliasing. Unfortunately, the locality (below), independence (below) and type domains are not substitution-closed.

Without recording which terms may share free variables (i.e. which terms are possibly aliased), analysis over a non-substitution closed domain may yield incorrect results [26].

**Example.** Consider the following program fragment:

```
..., X = Y, Y = 3, ...
```

Assume that X and Y are both **free** initially. When Y is unified with 3, it becomes **ground**. Without the knowledge that X and Y possibly are aliased, X will still be thought to be **free**, which is incorrect. If the analyser records that X and Y are possible aliases, then binding Y means X is *possibly* bound. The analyser approximates such a possible binding by setting the type of X to **any**. Note that if the analyser knows that X and Y are *certainly* aliased, X would become **ground** as soon as Y. □

We describe aliasing using descriptors of terms or parts of terms. A descriptor points out a part or the whole of a term. In our case, we are interested in the elements of a list, the final tail of a list or the entire term. Aliases are then expressed by grouping the descriptors that may or must share variables.

Assume that $X$ is a variable occurring in the clause currently being analyzed. A *variable descriptor* of $X$ is written as $\mathrm{all}(X)$, $\mathrm{elt}(X)$ or $\mathrm{tl}(X)$. The descriptor $\mathrm{all}(X)$ denotes the concretization of $X$, $\mathrm{elt}(X)$ denotes the list elements of the elements of the concretization of a list type $T$ and $\mathrm{tl}(T)$ denotes the last tail of elements of the concretization of a list type $T$.

We define a *parameter descriptor* of parameter $n$ of a call, as $\mathrm{all}(n)$, $\mathrm{elt}(n)$ or $\mathrm{tl}(n)$. The parameter descriptors denote parts of parameters to a call just as variable descriptors denote parts of the concretization of variables occurring in the clause.

A descriptor $d$ selects part of a term, which is defined by a function $sel(\cdot,\cdot)$ as follows: if the well-formed list $l = [t_1, \ldots, t_n|u]$ belongs to the concretization of $X$, then $sel(\mathrm{elt}(X), l) = \{t_1, \ldots, t_n\}$ and $sel(\mathrm{tl}(X), l) = \{u\}$. If a term $t$ belongs to the concretization of $X$, then $sel(\mathrm{all}(X), t) = \{t\}$. Parameter descriptors are handled similarly.

For example, if the variable $X$ has type **list_nv_f**, then for the element $[1, 2, 3|A]$ (which belongs to **list_nv_f**), the variable descriptor $\mathrm{elt}(X)$ refers to the elements 1,2,3 while $\mathrm{tl}(X)$ refers to the final tail $A$. The descriptor $\mathrm{all}(X)$ refers to the entire term $[1, 2, 3|A]$.

Parameter descriptors are used for call and success patterns to describe sharing between arguments. Source variable descriptors are used by i-states to describe dependences between variables in the clause being analyzed. For a given pattern or i-state, the types of the variables of the i-state and parameters, respectively, define a set of *available descriptors*, which may be used to describe aliases. We describe how the available descriptors are generated subsequently.

An *alias set* is a finite set of either variable or parameter descriptors, denoting that the described part of the elements in the concretization of the corresponding types may or must share variables.

Define a *partition* of a set $S$ as a finite set of sets $T$ such that (i) every element in $S$ is found in some set in $T$ and (ii) the sets in $T$ do not share elements:

$$S = \bigcup_{t \in T} t$$

$$\forall t_1, t_2 \in T : t_1 \cap t_2 = \emptyset$$

An *aliasing component* of an i-state or pattern is a partition of the available descriptors of the types assigned to variables or parameters of the i-state or pattern. The set of available descriptors depends upon the types involved and the kind of aliasing being described. The generation of these sets is described below.

Given an aliasing component $p$, two descriptors $d$ and $d'$ are aliased if there is a set $s \in p$ such that $d \in s$ and $d' \in s$. This is written as $d \overset{p}{\leftrightarrow} d'$.

We note that the set of available descriptors changes as types change: as list types appear, elt($\cdot$) and tl($\cdot$) descriptors may appear. As lists are grounded or collapse to other types, descriptors disappear. Thus, aliasing components must be updated to reflect changes in types. We will assume this to occur implicitly subsequently. Furthermore, aliasing is a transitive property by our definition: if two descriptors become aliased, then their aliases will also become aliased. This allows us an efficient implementation of aliasing in terms of equivalence classes and UNION-FIND algorithms [59].

**Possible aliases.** The analyser maintains possible aliases as an aliasing component. When some descriptor in a possible alias set may have been further instantiated, the other descriptors in the alias set may cause other variables to change type.

We say that a program variable $X$ is bound when elements of the concretization of $X$ would be bound due to some operation (such as unification). If some elements are bound, $X$ is possibly bound. If all elements are bound, $X$ is certainly bound.

When a possible alias of $X$ is possibly bound, the type of $X$ may be changed. If $X$ has type **free** or **free+nil**, it is changed to **any**. Since possible aliases express that concrete variables may or may not have been shared, the concretization of $X$ may or may not have become bound. The same holds for an open list **list_**$a$**_{f,fn}**: if an alias of the tail has been bound, the concrete lists may have become non-wellformed and the type of $X$ collapses to **any**.

The available descriptors for possible aliases are generated as follows. A variable $X$ with list type $T$ generates elt($X$) if $T$ = **list_{nv,a,f}_**$b$, and generates tl($X$)

if $T$ = **list**_$a$_{**f,fn**}. If $X$ has one of the **free**, **free+nil**, **nonvar** or **any** types, all($X$) is generated. Otherwise, $X$ is ground and generates no descriptor. The available descriptors of an i-state with variables $X_1, \ldots, X_n$ is then the union of the available descriptors of each $X_i$. Available descriptors for patterns are generated equivalently, from the numerical indices of parameters $1, \ldots, n$ for a predicate of arity $n$, rather than variable names.

Note that when taking the least upper bound of two aliasing components, we must ensure that the available descriptors are the same. Hence, the least upper bound operation for types with an aliasing component consists of (i) computing the new types, (ii) computing the available descriptors and adjusting the aliasing component, and (iii) computing the least upper bound of the new aliasing components. In the case of possible aliases, step (iii) merges aliasing classes by performing the coupled closure operation [24].

The *coupled closure* of two aliasing components $p'$ and $p''$ is the least aliasing component $p$ such that if $d \overset{p'}{\leftrightarrow} d'$ or $d \overset{p''}{\leftrightarrow} d'$, then $d \overset{p}{\leftrightarrow} d'$. The effect is to merge alias sets to reflect the possible aliases from any execution path.

**Example.** Consider the aliasing components

$$p' = \{\{\text{all}(1), \text{all}(2)\}, \{\text{all}(3), \text{all}(4)\}, \{\text{all}(5)\}\}$$

and

$$p'' = \{\{\text{all}(1)\}, \{\text{all}(2), \text{all}(3)\}, \{\text{all}(4)\}, \{\text{all}(5)\}\}$$

The coupled closure $p$ of $p'$ and $p''$ is then constructed as follows.

First, we know that descriptors all(1), all(2) are aliased, as are all(3), all(4) due to $p'$. From $p''$, we also get that all(2), all(3) are aliased. We see that elements all(1), all(4) and all(5) are 'aliased to themselves' (e.g., all(5) $\overset{p}{\leftrightarrow}$ all(5)). Hence, we get the following:

$$p = \{\{\text{all}(1), \text{all}(2), \text{all}(3), \text{all}(4)\}, \{\text{all}(5)\}\}$$

□

Comparison of possible aliasing components can be done by the standard construction: $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$.

**Certain aliases.** Two types are certainly aliased if their concretizations must share variables. We compute certain aliases for descriptors, expressing that parts of a term must be the same. Certain aliases are thus more limited than possible aliases.

The available certain aliasing descriptors are generated as follows. If $X$ has type **free** or **free+nil** or is a list type **list**_$a$_{**f,fn**}, generate the descriptor tl($X$). Otherwise, no descriptor is generated. Again, the available descriptors of an i-state are generated from the variables of the i-state, while patterns use numerical indices. We note that the certain aliasing descriptors select either a single variable or the empty list from a term, and that the set of available certain aliasing descriptors is a subset of the available possible aliasing descriptors.

When a variable $X$ is bound, and $X$ has a certain alias descriptor, the other descriptors in the certain alias set of $X$ are used to update the corresponding types. For a descriptor all($X$), the associated variable $X$ simply changes to the new type. For a descriptor tl($X$), the list type is updated appropriately.

We compute the least upper bound of two certain alias components as for a possible alias component, with the difference that step (iii) changes. The least upper bound of two certain aliasing sets with the same available descriptors is the *decoupled closure* of the components, such that $p$ is the decoupled closure of certain aliasing components $p'$ and $p''$ when $d \overset{p}{\leftrightarrow} d'$ iff $d \overset{p'}{\leftrightarrow} d'$ and $d \overset{p''}{\leftrightarrow} d'$. This is a conservative approximation since information is weakened. We amend this slightly, to make a descriptor for type **nil** available when needed, since the least upper bound of **nil** and **free** is **free+nil** and the certain aliases of the **free** type should be preserved. The result is still a conservative approximation, since the empty list is ground. Comparison of certain aliasing components can be done by the standard construction: $x \sqsubseteq y \Leftrightarrow x \sqcup y = y$.

**Example.** Take the following predicate:

```
p(X,X,Y,Z,Z).
p(X,Y,Y,Z,Z).
```

Assume the predicate is called with five unaliased **free** types. The available descriptors for both possible and certain aliases is the set:

$$\{\mathrm{all}(1), \mathrm{all}(2), \mathrm{all}(3), \mathrm{all}(4), \mathrm{all}(5)\}$$

Clause 1 then succeeds with certain aliasing component

$$\{\{\mathrm{all}(1), \mathrm{all}(2)\}, \{\mathrm{all}(3)\}, \{\mathrm{all}(4), \mathrm{all}(5)\}\}$$

while clause 2 succeeds with

$$\{\{\mathrm{all}(1)\}, \{\mathrm{all}(2), \mathrm{all}(3)\}, \{\mathrm{all}(4), \mathrm{all}(5)\}\}$$

The least upper bound is

$$\{\{\mathrm{all}(1)\}, \{\mathrm{all}(2)\}, \{\mathrm{all}(3)\}, \{\mathrm{all}(4), \mathrm{all}(5)\}\}$$

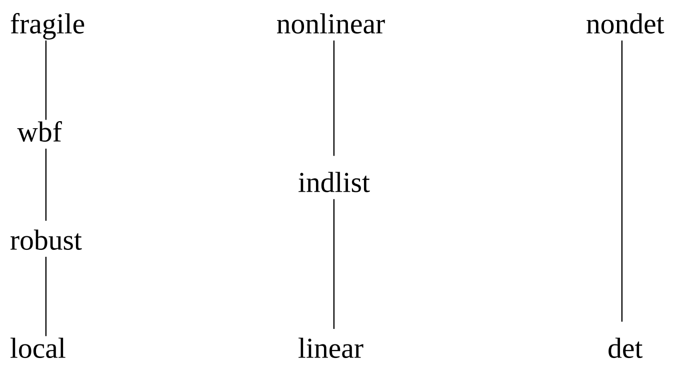

Figure 3.3: Locality, independence and determinacy domains

Since argument 4 and 5 are the same on both (all) execution paths, this is correct. The other certain aliases are lost, since they are no longer certain.

In the same way, the first clause yields the possible aliasing component

$$\{\{\text{all}(1), \text{all}(2)\}, \{\text{all}(3)\}, \{\text{all}(4), \text{all}(5)\}\}$$

while the second clause yields

$$\{\{\text{all}(1)\}, \{\text{all}(2), \text{all}(3)\}, \{\text{all}(4), \text{all}(5)\}\}$$

The coupled closure operation then joins $\{\text{all}(1), \text{all}(2)\}$ and $\{\text{all}(2), \text{all}(3)\}$ into one, which yields the least upper bound

$$\{\{\text{all}(1), \text{all}(2), \text{all}(3)\}, \{\text{all}(4), \text{all}(5)\}\}$$

since arguments 2 and 3 *may* refer to the same variable after the call. □

### 3.2.2 Locality

In our discussion on the execution model (Section 2.3-2.4.1), we found that robust and fragile variables are interesting, since they define where suspension and careful data handling is required. The Reform Prolog analyser, as a consequence, attempts to find locality information for terms and records what shared variables must be classified as fragile. During sequential execution all terms are local. When a parallel predicate is analysed, some terms are marked as shared as described in Section 3.4.1. When shared terms are subjected to time-dependent tests, they are marked as *wbf* (i.e., will be fragile). To subsequent recursion levels, *wbf* elements will appear as fragile elements, while to the current recursion level they are robust. Otherwise, shared types are assumed to be robust.

The full locality domain is then as follows.

$$\mathit{local} \sqsubseteq \mathit{robust} \sqsubseteq \mathit{wbf} \sqsubseteq \mathit{fragile}$$

The corresponding Hasse diagram is shown in Figure 3.3. Abstraction and concretization functions are defined with respect to sets of local, robust, will-be-fragile and fragile variables.

A set of terms is fragile if some of the free variables of the set belong to the set of fragile variables. Likewise, a set of terms is robust if it is not fragile (or wbf) and has free variables that belong to the set of shared variables. The other elements are defined similarly.

### 3.2.3 Independence

Our representation of aliasing can only express aliasing quite coarsely due to the transitivity property. For instance, if $X, X1$ and $X2$ are **any** types, and all are unaliased, the analyser approximates the unification $X = f(X1, X2)$ by the alias set $\{\mathrm{all}(X), \mathrm{all}(X1), \mathrm{all}(X2)\}$ which marks $X1$ and $X2$ as possibly sharing. While being a conservative approximation, it is an annoying source of imprecision.

In the case of non-ground lists, the analyser must also assume that list elements share variables even though this need not be the case. (Indeed, in our experience so far, it is quite uncommon.) Hence, aliasing information is too pessimistic here as well. The result is that the analyser must mark such terms as shared, which could degrade performance.

To address these problems, we extend the abstract domain described so far with an independence component that records the sharing between subterms in a type. A term is *linear* if all free variables of the term occur once in the term; the term is *nonlinear* if some variable may occur multiple times. If a list contains nonlinear elements, but there is no sharing between the list elements, the list is called an independent list, or simply an *indlist*. Compound terms apart from lists are classified only as linear or nonlinear.

We define the corresponding abstract elements as the collection of such terms, ordered as below.

$$\mathit{linear} \sqsubseteq \mathit{indlist} \sqsubseteq \mathit{nonlinear}$$

The domain is also shown in Figure 3.3. The abstraction and concretization functions are obvious. Concretely, independence information means that the analyser can ignore aliases between subterms when the corresponding type is linear, or an indlist for list types. In parallel execution, *linear* or *indlist* types can usually be marked as *local* rather than *shared* types. Note that the transitivity of aliasing prohibits the use of independence information when the subterms both are aliased to another term, as shown below.

**Example.** We will illustrate the use of independence information with an example.

Take the unit clause $p(f(X,Y),Z)$ and the partial call pattern $\langle linear, linear\rangle$ and no aliasing between argument 1 and 2. Then the analyser can conclude that X, Y and Z are not aliased: $\{\{\text{all}(X)\},\{\text{all}(Y)\},\{\text{all}(Z)\}\}$.

Now assume that the same clause is called with the same call pattern but that arguments 1 and 2 are possibly aliased. The analyser initially considers X and Y as unaliased, but then finds that X and Z are aliased, as well as Y and Z. The transitivity of aliasing then forces the analyser to conclude that $\{\{\text{all}(X),\text{all}(Y),\text{all}(Z)\}\}$. □

For some types, independence information is useless: ground types and non-structured types (**free, free+nil**) do not use independence information. The **any** and **nonvar** types can be linear or nonlinear. List types record independence information for list elements, which may have any independence value.

Obviously, the property of independence is not substitution-closed, since instantiation of a term may change its linearity. Furthermore, when two possibly aliased linear terms are abstractly unified, the result must be nonlinear. (This represents e.g. the case $f(X,Y) = f(Y,X)$, yielding $f(X,X)$.)

### 3.2.4 Safeness

The safeness domain is the following two-point domain:

$$det \sqsubseteq nondet$$

The domain is shown in Figure 3.3. The element *det* stands for certain determinacy, and *nondet* for possible nondeterminacy, measured either since the initial query, or since the enclosing parallel call. In practice, we are interested only in whether a parallel computation is deterministic since the process was started. Determinacy changes on a cut or as a result of abstract indexing (Section 3.4.2).

### 3.2.5 The full abstract domain

The full abstract domain is the composition of all the domains of the above and works with two kinds of objects, patterns and i-states. We first define these domains, then the domain used for predicates and that used for programs. Finally, we define the concretization of i-states and patterns.

The call and success patterns the analyser works with are elements of a family product domain defined below.

We say that the *reduction* of a domain [22] is the domain of equivalence classes of elements in the original domain, where elements are equivalent if they have the same concretization. For instance, there is no need to distinguish whether ground terms are linear, indlist or nonlinear since they contain no variables. Performing a reduction eliminates these superfluous elements from the domain.

We introduce the concept of an *adorned possible aliasing component*. Note that all descriptors in the same possible aliasing set will be assigned the same independence and locality values, since they describe that variables *may* be shared between processes, that terms *may* be nonlinear, etc., which are effects of possible occurrences of variables and hence of possible aliasing.

Hence, we subscript each aliasing set with the common independence and locality values of the descriptors. We have previously described how to take the least upper bound of a possible aliasing component. For an adorned possible aliasing component, the least upper bound of independence and locality of an alias set in the new component is simply the least upper bound of the independence and locality values of the joined alias sets, respectively.

**Example.** Consider the least upper bound of the adorned possible aliasing components:

$$\begin{array}{ll} & \{\{\mathrm{all}(1), \mathrm{all}(2)\}_{\mathit{linear,local}}, \{\mathrm{all}(3)\}_{\mathit{nonlinear,local}}\} \\ \sqcup & \{\{\mathrm{all}(1)\}_{\mathit{linear,local}}, \{\mathrm{all}(2), \mathrm{all}(3)\}_{\mathit{linear,robust}}\} \end{array}$$

As an intermediate step, we construct the coupled closure as

$$\{\{\mathrm{all}(1), \mathrm{all}(2), \mathrm{all}(3)\}\}$$

The coupled closure is then adorned with the least upper bound of independence and locality values of the constituent previous alias sets, *nonlinear* and *robust*:

$$\{\{\mathrm{all}(1), \mathrm{all}(2), \mathrm{all}(3)\}_{\mathit{nonlinear,robust}}\}$$

□

Take $T^n$ to be the product TYPE $\times \ldots \times$ TYPE of a dimension $n$ equal to the number of variables in the i-state to be described. Take $P$ to be the domain of adorned possible aliasing components over the variable descriptors of $T^n$ and $P'$ to be the domain of adorned possible aliasing components over the numerical descriptors of $T^n$. Take $C$ to be the domain of certain aliasing components over the variable descriptors of $T^n$ and $C'$ to be the domain of certain aliasing components over the numerical descriptors of $T_n$. Finally, $D$ is the determinacy domain $\mathit{det} \sqsubseteq \mathit{nondet}$.

We now define the abstract domains as:

- The domain of i-states describing $n$ variables is defined by the (reduced) product $T^n \times P \times C \times D \times D$. Note that $P$ and $C$ depend on $T^n$. The first of the two determinacy dimensions records the determinacy of the enclosing procedure (to be used for cut) while the second records the current determinacy.
- The domain of patterns $\mathrm{P}^n$ of procedures with $n$ parameters is defined as the (reduced) product $T^n \times P' \times C' \times D$.
- The domain of a predicate $p/n$ is then the product $CP \times SP$, where $CP = \mathrm{P}^n$ is the domain of call patterns and $SP = \mathrm{P}^n$ the domain of success patterns of the predicate.
- The domain of a program of $k$ predicates is the product of $k$ predicate domains.

The meaning of a pattern, i.e., its concretization, is defined as follows. For a predicate $P/n$, we assume there is a pattern with $n$ parameters.

The concretization of the pattern is then the set of terms $P(t_1, \ldots, t_n)$ which are generated by the types of the parameters, and satisfy the constraints on aliasing, independence and locality. The aliases and independence and locality information serve as constraints on the free variables of terms. In the case of possible aliases, we can only say that variables are definitely not the same if they are not possibly aliased. Certain aliasing descriptors denote a single variable or the empty list. If two terms described by certain aliasing descriptors are variables and certainly aliased, they must be the same variable.

The function $FV(t)$ returns the set of free variables of the term $t$. The function $ind(i,p)$ returns the independence value of $i$ in the adorned possible aliasing component $p$, while $loc(i,p)$ returns the locality value of $i$ in $p$. We define $d_i$ to be a descriptor of argument $i$, i.e. $\mathrm{elt}(i), \mathrm{tl}(i)$ or $i$, if $d_i$ is available in $p$. We assume this to be defined for both adorned and unadorned aliasing components.

To model the effects of locality, we assume there is a set of variables *FR* that represent the variables subject to time-dependent operations by preceding recursion levels in the current parallel call, if any. Furthermore, we define the set *WBF* to be the variables subject to time-dependent test on the current recursion level prior to the moment of concretization. The set *SHR* is the set of shared variables. These sets have the relation $FR \subseteq WBF \subseteq SHR \subseteq VAR$. Local variables must not be in *SHR*, robust variables not in *WBF* and variables that belong to the locality element *wbf* must not be in *FR*.

The concretization function is then defined as follows.

$\gamma_P(\langle\langle T_1, \ldots, T_n\rangle, p, c, d\rangle =$

$\{P(t_1, \ldots, t_n) \mid 1 \leq i \leq n \wedge t_i \in \gamma(T_i)\}$
where
$(i)$ if $\neg(\exists d_i, d_j : d_i \overset{p}{\leftrightarrow} d_j)$ then $FV(t_i) \cap FV(t_j) = \emptyset$;
$(ii)$ if $\exists d_i, d_j : d_i \overset{c}{\leftrightarrow} d_j$ then $sel(d_i, t_i) = sel(d_j, t_j)$ if $t_i, t_j \in VAR$;
$(iii)$ if $ind(i, p) = linear$ then
for $t \in \gamma(T_i)$, if $a, b$ are subterms of $t$ and
neither of $a, b$ a subterm of each other, then $FV(a) \cap FV(b) = \emptyset$
if $ind(i, p) = indlist$ and $[x_1, \ldots, x_k | xs] \in \gamma(T_i)$ then
for all $i, j$ s.t. $1 \leq i, j \leq k$, it holds that $i \neq j \rightarrow FV(x_i) \cap FV(x_j) = \emptyset$
otherwise (nonlinear case) no constraints;
$(iv)$ if $loc(i, p) = local$ then for $t_i \in \gamma(T_i)$,
$FV(t_i) \cap SHR = \emptyset$
if $loc(i, p) = robust$ then for $t_i \in \gamma(T_i)$,
$FV(t_i) \cap WBF = \emptyset$
if $loc(i, p) = wbf$ then for $t_i \in \gamma(T_i)$,
$FV(t_i) \cap FR = \emptyset$
otherwise (fragile case) no constraints.

An intuitive way of viewing the concretization is as a two-phase process. First the set of terms implied by the $t$ component are formed. Then the set is whittled down by considering the constraints on free variables.

**Example.** For instance, the element

$$\left\langle \langle \mathbf{free}, \mathbf{free{+}nil} \rangle, \{\{\mathrm{all}(1), \mathrm{all}(2)\}_{linear, local}\}, \{\{\mathrm{all}(1), \mathrm{all}(2)\}\}, det \right\rangle$$

has the concretization $\{p(x, x), p(x, [\,])\}$ and renamings thereof (modulo the *SHR*, *WBF* and *FR* components, which we have omitted). The state is known to be deterministic at this point. □

We define the concretization of i-states similarly. An i-state represents the set of substitutions possible at a given point in the program. For variables $\{X_1 \rightarrow T_1, \ldots, X_n \rightarrow T_n\}$ in the i-state, the concrete substitutions are formed as the set:

$$\{\{X_1/t_1, \ldots, X_n/t_n\} \mid 1 \leq i \leq n, t_i \in \gamma(T_i)\}$$

The independence, locality and aliasing components constrain free variables similarly to the pattern case.

## 3.3 ABSTRACT INTERPRETATION ALGORITHM

The basic global analysis algorithm is adapted from that of Debray [25, 24].

Debray describes a framework for global analysis. To actually arrive at a program analysis, the operations below must be provided. As mentioned previously, Debray's algorithm operates on call and success patterns and i-states that represent the states on entering and exiting a call, and between two calls, respectively.

- An operation initial_i_state that takes a clause and a call pattern and constructs an i-state representing the computation state after head unifications in the clause.
- An operation apply_i_state that takes an i-state and a concrete call and constructs a call pattern.
- An operation update_i_state that takes an old i-state, a concrete call and a success pattern and returns a new i-state representing the situation after the call.
- An operation final_i_state that takes the clause head and the i-state resulting from executing a clause body and constructs a success pattern.
- An operation analyse_primitive that takes an old i-state and a concrete call to a primitive operation (e.g. unification) and returns a new i-state.

The analysis algorithm uses several tables.

`MEMO(P)` A table mapping predicate names to $(C, S)$ pairs of a call pattern $C$ and a success pattern $S$. Initially, this table has entry $(\bot, \bot)$ for every predicate. On entering a call to $P$ with call pattern $C$, `MEMO(P)` is updated to $(C \sqcup C', S')$ if the previous entry was $(C', S')$. On exiting a call $P$ with success pattern $S$, likewise the update `MEMO(P)` $:= (C', S \sqcup S')$ is made.

`CLAUSES(P)` The program database; returns the clauses of the predicate `P`. The program database is fixed during execution. Dynamically modified predicates are treated as unknowns, giving a correct, though imprecise, result.

`CALLERS(P)` The call graph of the program is given as a mapping from a predicate to the predicates calling it.

`EXPORTS(P)` The entry points of the program to be analysed are given as *(P,CP)* entries, where *P* is a predicate and *CP* a call pattern of *P*. These are used to start analysis, and are ignored thereafter.

`PAR(P)` For parallel analysis, the analyser maintains a separate memo table, in all respects the same as `MEMO` except for where it is used.

The sequential analysis algorithm is shown in Figure 3.4 and Figure 3.5. The parallel analysis algorithm is described in Section 3.4. We briefly describe the structure of the sequential analyser below.

Analysis is started by putting the `EXPORTS` in a worklist. Predicates and call patterns are repeatedly removed from this list until it is empty. At this point, the algorithm has noted a set of predicates to reconsider, which becomes the new worklist. When this list is empty, the analysis has converged upon a fixpoint and terminates. The result is the `MEMO` table.

A call $C$ is computed by first looking up the predicate's entry $(C', P')$ in `MEMO`. If the current entry includes the call pattern (i.e., $C \sqsubseteq C'$), the recorded success pattern $P'$ is returned. Otherwise, the call pattern of the entry is updated to $C \sqcup C'$ and the predicate is marked for recomputation. The algorithm then analyses each clause, each returning a success pattern. When all clauses have been processed, `MEMO` is updated with the lub of all the success patterns. If the lub is not included in the old success pattern, all callers of the predicate are marked for recomputation.

A clause is computed by creating an initial environment, updating it to reflect the call pattern and analysing the body. When the body is finished, the resulting environment is applied to the head again, producing a success pattern. If the clause is certain to fail, $\perp$ is returned. (In fact, as soon as a call in the body fails, the analyser returns $\perp$ as the result. Further analysis is unnecessary and will worsen the entries in `MEMO` by adding useless information for situations that will never occur: the calls after the failing call.)

Clause bodies are analysed by analysing each call in succession, and updating the environment to reflect success patterns between calls.

## 3.4 ANALYSING REFORM PROLOG

In order to analyse recursion-parallel programs, we must extend the basic algorithm somewhat. We compute both parallel and sequential types for a predicate, and add operations that compute locality information for the types involved in the parallel call, and propagate bindings to types signifying shared between recursion-levels. Furthermore, to infer determinacy information with some precision, we introduce *abstract indexing*, which mimicks concrete indexing.

We begin by noting that the analyser now will act in two modes: sequential and parallel. We thus make use of the `PAR` table to store results of parallel analysis, add a 'mode' to the analyser to indicate which table to use when analysing a given predicate, and add a mode tag to entries in `WORKLIST` as well. From now on, we will assume that the analyser can operate either in sequential or parallel mode; the only difference is which memo table is used. The analyser only shifts from parallel to sequential mode and vice versa when a recursion-parallel predicate is analysed.

```
**analyse_program**() **returns** memotable
    $\forall P$: MEMO($P$) := ($\bot, \bot$);
    WORKLIST := EXPORTS;
    **while** WORKLIST $\neq \emptyset$ **do**
        WORKLIST := WORKLIST - ($P, CP$);
            *Let $C_1, \ldots, C_n$ be clauses of P*
            $SP := \sqcup_{i=1}^{n}$ **analyse_clause**($C_i, CP$)
            $SP'' := SP \sqcup SP'$;
            **if** $SP' \sqsubseteq SP''$ **then**
                Add CALLERS($P$) to WORKLIST;
            **fi**
        MEMO($P$) := ($CP \sqcup CP', SP \sqcup SP'$);
    **end while**
    **return** MEMO;
```

Figure 3.4: The analysis algorithm

### 3.4.1 Analysing parallel programs

The abstract interpretation algorithm we have used for analysing Prolog is insufficient for analysing parallel code. The reason for this is the parallel flow of control. In essence, when analysing a recursion level we cannot presume that previous recursion levels have terminated (which we do for the sequential case) nor that later recursion levels are inactive (which is again the case for sequential execution). We will see that the data-parallel style of recursion-parallel programs simplifies analysis as compared to more general efforts [25].

#### Locality annotation

Prior to parallel analysis of a recursion-parallel body, the types are annotated as robust or local; this phase simulates the large head unification phase of concrete execution. It is performed once per body to be executed, since locality information can change between calls. Types also change due to the head unification. We will not describe this process, but the reader may note that Chapter 4 describes the compilation of the large head unification. The types of arguments are straightforwardly derived from the workings of these instructions.

Locality annotation computes argument independence as previously explained and then performs a case analysis over each pair of arguments from the head of the call and the recursive call, by examining the classification of the argument (Section 2.1.3):

```
**analyse_pred**$(P, CP)$ **returns** succpat
    Lookup $(CP', SP') \in$ MEMO$(P)$;
    **if** $CP \sqsubseteq CP'$ **then return** $SP'$
    **else**
        Add $P$ to WORKLIST;
        MEMO$(P) := (CP \sqcup CP', SP')$;
        *let $C_1, \ldots, C_n$ be the clauses of $P$*
        $SP := \sqcup_{i=1}^{n}$ **analyse_clause**$(C_i, CP)$
        $SP'' := SP \sqcup SP'$;
        **if** $SP' \sqsubseteq SP''$ **then**
            Add CALLERS$(P)$ to WORKLIST;
        **fi**
        MEMO$(P) := (CP \sqcup CP', SP \sqcup SP')$;
        **return** $SP$;
    **fi**

**analyse_clause**$(H :\Leftrightarrow B, CP)$ **returns** succpat
    $A_0 :=$ initial_i_state$(CP, H, B)$;
    $(G_1, \ldots, G_n) := B$;
    **for** $1 \leq i \leq n$ **do**
        $A_i :=$ **analyse_goal**$(G_i, A_{i-1})$;
    **end for**
    **return** final_i_state$(A_n, H)$;


**analyse_goal**$(G, A)$ **returns** i_state
    **if** primitive$(G)$ **then return analyse_primitive**$(G, A)$
    **else**
        $CP :=$ apply_i_state$(A, G)$;
        $SP :=$ **analyse_pred**$(G, CP)$;
        **return** update_i_state$(A, G, SP)$;
    **fi**
```

Figure 3.5: The main procedures of the sequential analysis algorithm

POSLIST. Each element X of the list will exist in a different recursion level. If the list is linear, X is local (and linear). If the list is an indlist, X is local and nonlinear. If the list has only ground elements, X is ground. Otherwise, X is robust (and nonlinear). If there are references to the rest of the list in the left or right bodies, X is marked as robust unless ground, regardless of linearity.

NEGLIST. The elements of a NEGLIST are constructed during the computation. Thus, they are all **free**, unaliased and local.

INV. An invariant argument is shared by all recursion levels and thus robust. Likewise, the type is the same on all recursion levels.

NONE-NEG. For a NONE-NEG argument pair $(X, Y)$, the type of $Y$ is **free**, while $X$ has the type of the lub of the input argument and **free**. Both $X$ and $Y$ are robust, since they appear in adjacent recursion levels.

**Example.** The recursive clause of the naive reverse program:

```
nrev([X|Xs],Zs) :- nrev(Xs,Ys), append(Ys,[X],Zs).
```

The first argument is a POSLIST and the second is a NONE-NEG pair. Hence, for locality annotation, X will be local if Xs is linear, an indlist or ground. Ys and Zs will be robust. □

Note that no terms are annotated as fragile. Fragility is a consequence of the subsequent body execution. During execution of the body, the analyser notes all *robust* data that are subject to time-dependent tests; the term is then marked as *wbf*. When the recursion level exits, subsequent levels will view the *wbf* terms as *fragile*. This effect is taken care of by binding propagation.

## Binding propagation

Bindings to robust variables may propagate to the left as well as right during parallel execution; i.e., later recursion levels may bind variables of the current level. Likewise, preceding recursion levels may bind variables of the current level, before, after or during the execution of the current recursion level. This situation is simulated by weakening the information in the call pattern.

The variables affected by bindings from surrounding processes are precisely the shared variables, which also appear in the call pattern, and their aliases. The binding propagation algorithm computes the information that leaks through from the left (predecessor) recursion levels as well as the right (successor) recursion levels. The result is two new environments, which are combined with the original one using locality information and the least upper bound operation. The resulting environment is applied to the body call, producing a new call pattern. If this call pattern is strictly greater than the old one, the body is marked for recomputation, in both parallel and sequential mode. Binding propagation is computed as follows:

1. Compute the forward environment $A_F$ from the success pattern $SP$ by: apply the original environment $A$ to the recursive call $H'$, producing a call pattern. Then compute the initial i-state using this call pattern and the head $H$ of the recursive clause with initial_i_state to arrive at $A_F$.

2. Compute the backward environment $A_B$ from $SP$ and $A$ by: update $A$ to reflect $SP$, and finalize i-state to produce a success pattern (just as if exiting a clause). Then perform an update_i_state using the success pattern and the recursive call $H'$.

3. If this is the left body, the left environment $A_L = A_F$ and the right environment $A_R = A_B$, while if this is the right body, $A_L = A_B$ and $A_R = A_F$.

4. Construct the new environment $A'$ from $A_L, A$ and $A_R$. This is done by considering each variable in the environment. Note that all variables in a possible alias group have the same locality. First, for each variable $X$ in the environments:

   - If $A(X)$ is local, $A'(X) = X$.
   - If $A(X)$ is fragile or $A_L(X)$ is fragile, or $A(X)$ is wbf, then $A'(X) = A_L(X) \sqcup A(X)$
   - Otherwise, $A'(X) = A_L(X) \sqcup A(X) \sqcup A_R(X)$.

   The lub operation updates certain and possible aliases, independence and locality. Finally, we set the determinacy of $A'$ to *det*, since recursion levels always start as deterministic and never leave choice points. (The compiler would flag such code as unsafe.) Furthermore, all *wbf* elements in $A'$ are converted to *fragile*.

### Parallel analysis

We are now in a position to sketch the parallel analysis phase. There are three components: analysing a parallel predicate, a parallel clause and a parallel body.

Analysing a parallel predicate means analysing the base case and the recursive clause sequentially (e.g. if abstract indexing indicates the first argument can be a variable), and analysing the recursive clause with the parallel method. The result is the lub of the three success patterns.

When analysing a parallel clause, the reader may find it useful to refer to the execution model (Section 2.2). Analysing a parallel clause $H \leftarrow \Phi, H', \Psi$ means annotating the locality of variables, followed by analysing the parallel body $\Phi$. Such analysis returns an i-state summarizing the types after all processes in $\Phi$ have terminated; it is used to analyse the final call to the base case $H'$. When finished, analysis continues with locality annotation, followed by analysing the right body $\Psi$ in the same manner as $\Phi$ and eventually returns an i-state. This i-state is used to construct the success pattern of the parallel clause.

Analysis of a parallel body is done in two passes. Internally, the compiler rewrites the left and right bodies to calls to new predicates. This way, the compiler can treat parallel bodies just like normal predicates for the purpose of recomputation and storing call and success patterns.

First, the call to this new predicate, which represents the body, is analysed in parallel mode. The success pattern is used for binding propagation as described above, and may cause the body to be recomputed if the new call pattern is more general than the old one. Then, the body is analysed in sequential mode, again yielding a success pattern. This pattern is the result of analysing the call: it summarizes the situation when all processes have terminated (and not just the current one). Locality information is removed, since parallel execution is finished.

To give an intuition of the algorithm, the parallel phase finds the types that hold when any recursion level starts, while the sequential phase finds types that hold when all preceding recursion levels have terminated and no subsequent recursion levels have started. From this information, the annotation phase synthesizes the situation when the recursion level is leftmost but other recursion levels may be active; this is straighforward by using locality information to inhibit results from subsequent levels and otherwise computing the least upper bound. For fragile data, sequential types are used; robust data takes the least upper bound.

**Example.** Sequential types yield more precise results to subsequent execution. Consider the following program:

```
test_map :- map([1,2,3],Xs).

:- parallel map/2.
map([],[]).
map([X|Xs],[Y|Ys]) :- X=Y, map(Xs,Ys)
```

Clearly, after execution of map/2 the list Xs is ground. However, using only parallel analysis, we can only conclude that it is a list of arbitrary elements. The reason is that while binding propagation does not model being leftmost, neither does it model being rightmost. The success pattern holds for any process that terminates anytime during the execution of map/2. Binding propagation then concludes that *some* elements are ground while others are free; thus, the entire list must be approximated as having arbitrary elements (the least upper bound of ground and free elements in our domain).

Using sequential types, the result is the expected: when map/2 terminates, both arguments are lists of ground elements. Accurate success patterns do not really help the compilation of the current parallel predicate; however, subsequent predicates may benefit considerably by getting more precise call patterns. □

### 3.4.2 Abstract indexing

WAM-based Prolog implementations employ clause indexing [76, pp. 68-77][1] to narrow down the sequence of clauses to be tried when entering a call. In particular, indexing can find that a given call does not require a choicepoint, and furthermore that the first argument of a call must be of a certain type (e.g., a variable or a structure).

To properly simulate the actions of the actual implementation, our analysis algorithm also performs indexing but on the abstract type elements of the call at analysis time. We call this process *abstract indexing*.

Prior to analysis, the compiler computes indexing information based on type information of the first argument (*key*) of each clause. This process is similar to that of other WAM implementations; in particular, there is no effort to extend indexing to other arguments. The same indexing information is later used by the code generator to generate indexing code.

An indexing structure is a tuple $\langle V, C, L, N, S\rangle$ where $V$ and $L$ are sets of clauses and $C, N, S$ are *clause tables*. A clause table is either a set of clauses or a total mapping from constants or functors to clause sets. The subset $V$ is chosen if the key is an unbound variable, $C$ if the key is an atom, $L$ if a list, $N$ if a number and $S$ if the key is a structure. If the entry is a mapping, the argument is scrutinized further.

**Example.** Consider the naive reverse program.

```
nrev([],[]).
nrev([X|Xs],Zs) :- nrev(Xs,Ys), append(Ys,[X],Zs).
```

We compute the indexing tuple $\langle V, C, L, N, S\rangle$ for nrev/2, where $V$ is $\{1,2\}$, $N = S = \emptyset$, $L = \{2\}$ and finally $C = \{[] \rightarrow \{1\}\,;\mathbf{otherwise} \rightarrow \{\}\}$. □

The result at concrete execution is an ordered subset of the clauses of the predicate, to be tried in turn. In particular, if the set is a singleton, the call is deterministic; if the set is empty, the call fails immediately. During global analysis, we do not work with concrete terms, but rather with types representing sets of terms. To approximate the actions at concrete execution, we must follow *all possible paths* into the indexing structure, as given by the concretization of the type of the first argument.

A simple implementation could just take the union of all clause sets found by that method, and say that the call is deterministic if this union is a singleton. For instance, if nrev/2 above was called with a list argument, $L$ would tell us that only clause 2 would be tried. The disadvantage with this approach is that, e.g., when nrev/2 is called with a ground key, the analyser would approximate this with the clause set $\{1, 2\}$ and conclude that the calls are nondeterministic. We can do much better with a bit more work.

First, note that we enter the clauses after certain tests. If we enter the clause through selecting it from the $V$ clause set, we know the clauses must be called with the key being **free**; likewise, the argument must be **ground** when entering through $C$ or $N$, a **nonvar** type through $L$ and a **nonvar** when entering through $S$. (Since indexing does not determine that a list is a proper list, the $L$ category cannot improve results further.) Thus, the call pattern of a clause can be improved. Note that a clause can be entered from several points (e.g., both $C$ and $S$); in such a case, we must use an upper bound of the involved entry point types (e.g., **nonvar**). The abstract indexing algorithm of the Reform analyser uses this more sophisticated method.

**Example (cont).** To continue the example, assume that nrev/2 is called with a call pattern $\langle\mathbf{any}, \mathbf{any}\rangle$ argument. (We ignore determinacy for the moment.) For the $V$ category, the call pattern is $\langle\mathbf{free}, \mathbf{any}\rangle$; for the $C$ category, the call pattern is $\langle\mathbf{ground}, \mathbf{any}\rangle$; for the $L$ category, we have $\langle\mathbf{any}, \mathbf{any}\rangle$ while the $N$ and $S$ categories yield no call patterns. The upper bound of these call patterns is $\langle\mathbf{free{+}nil}, \mathbf{any}\rangle$ for clause 1 and $\langle\mathbf{any}, \mathbf{any}\rangle$ for clause 2. Both clauses must be analysed, as given by $V$ or $C$ and $L$ together. □

To compute whether a call is deterministic or not, we also use the jump table mappings of $C, L$ and $S$. If one of the entries lead to nondeterminacy, or the call pattern was labelled nondeterministic, the call is nondeterministic. Otherwise, it is deterministic. Thus, even if the analyser has to consider several clauses, the call pattern may still be deterministic.

**Example (cont).** In the nrev/2 example, $V$ yields that the first clause will be called with a *nondet* call pattern if the indexing argument is **free**, while $C$ and $L$ each yield *det* for the first clause. The second clause will be called with *det* both with a **free** or other argument, since it is the last in the sequence of clauses to be tried, and the implementation will have removed the choice point when the clause is entered, as an optimization. □

The result of the abstract indexing algorithm is a set of $(CP, C)$ pairs, with $CP$ a call pattern and $C$ a clause. Abstract indexing can be seen as a refinement of the ordinary abstract execution of a call, where all clauses use the same $CP$ (and all clauses must be analysed).

A pragmatic advantage of abstract indexing is that clauses are entered with intended types and determinacy. For instance, consider the program fragment

```
p(X) :- var(X),!,c1(X).
p(X) :- X = f(Y), c2(Y).
```

The programmer intends that p/1 is always entered deterministically and that X in the first clause is **free**, while X in the second clause has type **nonvar**. Abstract indexing takes this into consideration. Furthermore, the unification $X = f(Y)$ might be considered unsafe unless the analyser can tell that X must be **nonvar** at this point.

Abstract indexing also removes clauses from consideration. If p/1 above is called with a **free** argument, and all clauses are considered (as is done in other algorithms) then analysis of the second clause would imply that p/1 is unsafe, even though the situation never occurs at runtime.

We conclude that abstract indexing is highly useful in that it follows the expected behaviour of the system, meaning the programmer's intuition is still correct.

## 3.5 EXAMPLE

We illustrate our analysis algorithm and abstract domain by means of an extended example, the analysis of a parallel tree-insertion program.

The program inserts a list of key-value pairs in a tree with variables as leaves. During parallel execution, all processes simultaneously try to insert their values in the tree, and synchronize to maintain sequential semantics. The program assumes there are no repeated keys.

```
:- parallel tree_insert/2.

tree_insert([],_T).
tree_insert([(K,V)|Xs],T) :- insert(T,K,V), tree_insert(Xs,T).

insert(X,K,V) :- var(X),!,X = t(_,K,V,_).
insert(t(L,K0,V0,R),K,V) :-
        ( K @< K0 ->
          insert(L,K,V)
        ; K @> K0 ->
          insert(R,K,V)
        ).

?- tree_insert([(X,a),(Y,_),(Z,f(_))],T).
```

Operationally, each recursion level will descend through the tree as it becomes instantiated. When the recursion level is leftmost, it binds the leaf variable to insert the pair.

The analyser first infers that tree_insert/2 is called with a **list_nv_n** and a **free** argument. The list is linear and the predicate arguments are unaliased.

The first clause of tree_insert/2 yields **nil** and a **free** variable as a result. In this example, we need not analyse the second clause sequentially, so we refrain from this.

In the second clause of tree_insert/2, after locality annotation, variables K and V are **any**, unaliased, linear and local, while T is an INV parameter, **free**, unaliased and robust. We now consider the call to insert/3.

The first argument of insert/3 is noted as *wbf*, due to indexing. Both clauses are called deterministically, since they are mutually exclusive. The first clause returns that X has a nonvariable type, and that X,K and V are aliased and wbf. All of them are still linear. The second clause yields the same result.

The analyser now performs binding propagation. The left environment marks T as fragile, but since the list is linear, Xs are still considered local and not aliased to T. The right environment notes that Xs and T are aliased, fragile and linear. The original environment considers Xs and T to be unaliased and linear, Xs to be local and T to be robust.

The new environment is then constructed as: T is fragile and **any**, since it may be **nv** or **free**, and linear. Xs is a list of local, linear, nonvariable elements, and is unaliased. K and V have **any** type, are unaliased, linear and local. The influence of the right environment is blocked, since T has locality wbf due to the success pattern of insert/3 and Xs is a linear list.

The new call pattern of insert/3 is then: all three arguments are **any**, linear and unaliased. The first two arguments are local while the third argument is fragile.

This call pattern is greater than the previous (the third argument has a larger type and is fragile rather than robust). Thus, the call insert/3 is recomputed both in parallel and sequential mode.

## 3.6 RELATED WORK AND DISCUSSION

### 3.6.1 Abstract domain

Our domain does not in general track functors, which is done by more complex abstract domains (e.g. depth-k abstraction, Bruynooghe's and Janssens' type graphs [38]). However, we can handle difference lists, which type graphs are unable to capture. The reasoning is this: type graphs can express certain aliases only on paths through the graph that do not traverse an Or-node. Expressing recursive lists require Or-nodes. If a certain alias is not maintained, the tail of the list will collapse into a Max-node (our **any**), which collapses the entire type graph into a Max-node. One might thus consider extending the scope of certain aliases for type graphs somewhat.

Taylor [68] uses a domain quite similar to ours. His domain is more extensive in that it differentiates constants into numbers, atoms, and so on. Furthermore, it incorporates depth-k abstraction and maintains low-level information on free variables. Our treatment of lists is similar to Taylor's but more restricted. Taylor allows any type as the tail of a list, but has the same concretization. This leads in the case of a list ending in **any**, to the concretization:

$$\textbf{any}, [\alpha|\textbf{any}], \ldots$$

This of course yields:

$$\textbf{any} \sqsubseteq \textbf{list\_}\alpha\textbf{\_any}$$

At the same time, since **any** concretizes to all terms, we have:

$$\textbf{list\_}\alpha\textbf{\_any} \sqsubseteq \textbf{any}$$

Thus, Taylor's domain would seem to require a notion of equivalence classes of types. We think our restriction to proper list types manages to capture most of the uses of lists while avoiding such potential problems. Taylor also does not consider the use of **free+nil** in his domain, and might then suffer a loss of precision in the common case illustrated below:

```
q(Xs,Ys) :- p(Xs,Ys,[]).

p([],A,A).
p([X|Xs],A,C) :- r(X,A,B), p(Xs,B,C).

r(X,[X|A],A).
```

In this program, p/1 is called with the third argument being **nil** or **free**. The least upper bound of these two values is often defined to be **any**. By introducing **free+nil**, our analyser can continue the analysis with precise types.

Independence is also considered by Søndergaard [65] as well as Codish *et al.* [18]. These domains have a more precise notion of sharing, but do not trace types as extensively as ours. Furthermore, they do not incorporate the *indlist* type which seems to be pragmatically useful in the case of Reform Prolog.

The quality of abstract indexing is dependent on the programs analysed as well as the quality of the type analysis. The types used conform quite well to the WAM model of indexing but can be made better. In particular, distinguishing numbers, atoms and structures, as well as knowing when a list has at least one element could give far better precision in this regard.

The Andorra-I preprocessor [57] extends Taylor's domain with disjunctions of terms for better precision. For instance, the Andorra-I system can find that a variable is bound either to a structure $f(\ldots)$ or a structure $g(\ldots)$. The Andorra-I system uses determinism analysis and testing extensively to expose the maximum amount of parallelism under the Andorra principle (Section 1.3).

In practice, we have found that introducing an extra set of types, *lists with at least one element*, allows us to improve certain cases when abstract indexing or parallel analysis interact with list types and precision is lost as a result. Adding such types is an interesting possibility for future extensions.

### 3.6.2 Analysis algorithm

Taylor [68] considers several alternatives for fixpoint computation, also settling on maintaining a single call-success pattern pair. Debray [24] also considers this solution. Le Charlier and Van Hentenryck [44] use a more sophisticated method to avoid superfluous recomputation by keeping track of dependences between goals. Debray's algorithm statically approximates recomputation dependences through the call graph, while Le Charlier and Van Hentenryck does so dynamically.

Bruynooghe [11] has defined a framework based on abstract proof trees; in our work, we found Debray's more operational formulation more suitable for our purposes. We think our analysis could be recast into Bruynooghe's framework with a minor effort.

Debray [25] also considers a much more general framework for analysing concurrent logic programs. We have not compared our formulation with his, but note that more general parallel execution also may lead to a lessening of precision. This is inherent in a situation where a collection of processes can run in any order, since the analysis must then somehow accomodate all the possible execution orderings.

We could probably define an execution ordering for Debray's framework to simulate our algorithm.

Analyses of concurrent logic languages [19] suffer from the same disadvantages. There are properties that do not rely on execution order, however, and these can be inferred more efficiently [17, 25]. Likewise, Muthukumar and Hermenegildo's sharing analysis [52] detects sharing and groundness to reduce independence tests in independent and-parallel implementations, without considering parallel execution (which is not necessary).

Filé and Rossi [29] consider the general analysis of programs with cuts, and present a semantics where cuts are executed if execution is guaranteed to reach the cut, and ignored otherwise. The usefulness of this definition of course depends on whether execution frequently is *certain* to reach a cut, which is dependent on the abstract domain used as well as the program being analysed. Their treatment of cuts is more general than that of abstract indexing; it would be interesting to investigate the absolute differences in precision on real programs. We note that abstract indexing not only takes cuts into account (as part of the indexing algorithm) but also other type information.

The Andorra-I preprocessor analyses (at the time of writing) parallel programs simply by performing a sequential analysis (which may not reflect the runtime situation during parallel execution) and compensating this in the compiler. As far as we are aware, the Andorra-I system does not attempt to remove suspension and locking instructions as Reform Prolog does. To perform these optimizations, simulating the actual execution seems necessary.

Chapter 4

# Compiling recursion-parallel programs

This chapter describes the compilation of recursion-parallel programs. We discuss the concrete execution of Reform Prolog and show how recursion-parallel programs are compiled, how suspension and locking unifications can be eliminated and how arithmetic recurrences can be compiled.

## 4.1 OVERVIEW OF THE WARREN ABSTRACT MACHINE

The execution engine of Reform Prolog is based on the Warren Abstract Machine, or WAM [77]. In this section we provide a high-level overview of the machine. For a closer look, the reader is urged to consult Aït-Kaci [1].

The WAM consists of a set of registers and four global data areas: a code area, a local stack, a global stack and a trail stack. Registers hold procedure parameters and temporary values and point into the local and global stacks. To implement backtracking, the WAM stores a linked list of choice points representing unexplored branches of the proof tree (i.e., saved engine states). Forward execution is implemented by a linked list of activation frames, or environments. Environments contain a continuation pointer and the variables that survive the call, also called permanent variables. The WAM implements a pure caller-saves convention for registers: all registers that survive the call are stored in the environment. There are several optimizations to this basic scheme, but we will not linger. The local stack holds both choice points and environments.

Terms are stored on the global stack. Compound terms occupy several contiguous words, while variables and constants usually occupy one word each. When a variable is bound, the corresponding heap cell is assigned the new value. If the binding is conditional, meaning that the variable may assume another value in another branch of the proof tree, the address of the bound cell is pushed on the trail stack. When a variable is bound to a variable, a pointer chain is constructed. To find the value of a variable, the engine must follow such pointer chains. The engine always binds younger (more recently created) variables to older ones, which avoids circular pointer chains and dangling references on backtracking.

On backtracking, the most recent choice point is consulted and all data allocated since the creation of the choice point is discarded. All bindings to variables older than the choice point are undone by popping the trail stack and unbinding the addresses found there. The local and global stacks are reset to their state when the choice point was created. The choice point is then used to load the registers with the saved state and execution is resumed. If the current choice point has exhausted the available choices, it is popped and the next to last choice point becomes the next point to backtrack to. Otherwise, it is updated to resume execution at the next branch when invoked.

## 4.2 OVERVIEW OF RECURSION-PARALLEL EXECUTION

A Reform Prolog system consists of a number of *workers*, each being an independent process running a WAM modified for execution in a concurrent setting.

Only a single designated worker is active during sequential execution. This worker suspends during parallel execution, while all other workers execute the available recursion levels (Section 2.1). The sequential worker performs the large head unifications prior to starting parallel execution and in effect creates all recursion level invocations (i.e., processes) at once.

For compilation purposes, some of the source variables in the recursive clause are considered *vector variables*, while others are scalars, or *global variables*.

Special instructions create contiguously allocated lists where the elements can be accessed in constant time, so called *vector-lists*. A vector variable is a reference into a vector-list, moderated by an offset depending on the number of the current recursion level.

Global variables are shared by all recursion levels and reside in standard WAM registers of the sequential worker. Since this worker is inactive during parallel execution, other workers can fetch scalar variables directly from its registers. These registers are also called *global registers* during parallel execution. Global registers can be read by parallel workers but not written.

Workers may be assigned a number of recursion levels at the start of parallel execution (static scheduling) or dynamically fetch recursion levels from a queue (dynamic scheduling). In either case, knowing which process to run next is trivial. Since all recursion levels contribute to the final solution, there is no unnecessary work done as long as the call succeeds. (If a recursion level fails, all work on subsequent recursion levels is wasted, and can be regarded as speculative.)

If a recursion level suspends, the worker suspends with it. When there are more workers than processors, the operating system can then schedule another worker to run on the free processor. When the supply of recursion levels is exhausted, the worker terminates; when all workers have terminated, the parallel body has finished, and the sequential worker is reactivated. (Termination in this context is merely suspension to await more work, not actual termination of the worker process.)

Backtracking is done by the sequential worker, or locally by each parallel worker. If the sequential worker backtracks over a parallel call, resetting the state modified by many recursion levels is somewhat more complex than the standard untrailing operation, but not unduly so. Parallel workers backtrack locally as if sequentially.

We note that the total overhead for creating and executing a process can be summarized as:

- The cost for building the vectors, amortized over $N$ recursion levels.
- The scheduling cost of the loop initialization instruction, done once per worker, and that of the 'find next iteration' instruction (plus jump) executed once per recursion level.
- Finally, the instructions to move values in vector registers and global registers into temporary registers, paid once per recursion level.

The design and implementation of the Reform engine is described in greater detail by Bevemyr [8].

## 4.3 COMPILER OVERVIEW

The Reform compiler is responsible for the following tasks:

1. Generating code to perform head unifications and start all processes. This is described in Section 4.4, Section 4.5 and Section 4.6.
2. Ensuring safeness by compile-time suspension when shared variables may be conditionally bound. This is described in Section 4.8.
3. Ensuring sequential semantics by emitting correct suspension instructions in the presence of time-dependent operations. This is described in Section 4.8.
4. Ensuring that compound terms are created atomically on the heap when necessary. This is described in Section 4.7.

The schematic code generated for a (list recursive) recursion-parallel predicate P/N is the following:

```
P/N:
    switch_on_term
        Var: jump to sequential version of predicate
        Const: P/N/Base
        List: P/N/Rec
        Num: fail
        Struct: fail
P/N/Base:
    code for base case
P/N/Rec:
    allocate
    head unification instructions and recurrence code
    start_left_body L1
    save registers used in right body
    and set up arguments for recursive call
    call P/N
    restore registers used in right body and compute
    recurrences
    start_right_body L2
    deallocate
    proceed
L1: code for left body
L2: code for right body
```

If the left or right bodies are empty, some of the steps can be omitted. In particular, if there is no right body the allocation and deallocation of environments can be avoided along with the register save and restore sequences, while the **call** becomes an **execute** instruction.

Predicates called by a parallel predicate are compiled according to the standard WAM principles but also include suspension instructions and locking unifications. These instructions are described below.

Unless otherwise mentioned, the arguments to the described instructions are always temporary registers.

### 4.3.1 Unifications

In general, unification solves the equation $S = T$ over the Herbrand universe[1], where $S$ and $T$ are arbitrary terms. From here on, we will consider $X, Y, Z, \ldots$ to

[1]The concrete implementation allows circular unifications but does not guarantee termination when such terms are being unified.

be variables occurring in the program. Constants (atoms, numbers) are written as $a, b, c, d, \ldots$. Structures are written as $f(T_1, \ldots, T_n)$ for a structure with functor $f$ and $n$ arbitrary terms as arguments, or as $f(X_1, \ldots, X_n)$ for a structure with $n$ program variables as arguments. By convention, these variables all occur for the first time in the program. When lists are distinguished from structures, they are written in Prolog notation, $[T_1|T_2]$. Arbitrary terms are written as $T_1, T_2, \ldots$

If a variable occurs only once in a clause, that occurrence is a *void occurrence*. If a program variable occurs textually for the first time, we say that it is a *first occurrence*. All other occurences of the same variable are *subsequent occurrences*.

Prior to actual compilation, the compiler simplifies all unifications to one or more unifications of the following form:

- $X = a$
- $X = f(T_1, \ldots, T_n)$
- $X = [T_1|T_2]$
- $X = Y$

We say that a unification is *flat* if the terms $T_i$ on the right hand side are all atoms or variables occurring for the first time. A unification $X = Y$ is a *general* unification if both $X$ and $Y$ are subsequent occurrences. All unifications can be rewritten to a sequence of flat and general unifications.

Initially, our compiler will consider only the flat and general cases of unifications. Subsequently, this restriction is lifted to optimize the generated code.

### 4.3.2 Notation for types and locality of variables

Prior to compilation, all variable occurrences have been annotated with types and locality information. This annotation also makes the order of unifications explicit when ambiguous (e.g., in head unification, the arguments may be unified in any order).

Parallel and sequential types may differ when a predicate is called during parallel execution. We will write the parallel and sequential types of $X$ as $X$.par and $X$.seq, respectively, while their aggregation is simply $X$.type. The locality of $X$ is written as $X$.loc.

We define $X$.type = ($X$.par $\sqcup$ $X$.seq) unless $X$ has certainly been subject to a time-dependent test (e.g. indexing) in which case we modify $X$.par to reflect suspension, using $X$.seq.

## 4.4 HEAD UNIFICATION

| | |
|---|---|
| **build_rec_poslist** $A, N, V, T$ | **build_neglist** $A, N, V, H$ |
| **build_poslist** $A, N, V, T$ | **build_neglist_value** $A, N, V, W, H$ |
| **build_poslist_value** $A, N, V, T$ | **build_variables** $A, N, V, T$ |

The **build** instructions are vector-like operations that construct vector lists of length $N$ in a single abstract machine operation.

The head unification phase must determine the number of processes to be executed, and set up all recursion levels to be executed. Furthermore, recurrence operations are executed in the head unification phase.

The head unification instructions create vector-lists for arguments requiring them, and store a reference to the portions of these lists to be used in the single recursive call.

The code emitted for each argument depends on the argument classification (as described in Section 2.1.3) and whether this is the recursion argument.

POSLIST. A POSLIST has two subcases depending on whether the head argument has occurred previously or not. If the head argument has not occurred before and this is the recursion argument, emit:

**build_rec_poslist** $A, N, V, T$

If this is the first occurrence but not the recursion argument, emit:

**build_poslist** $A, N, V, T$

If this is a subsequent occurrence, emit:

**build_poslist_value** $A, N, V, T$

In all cases, $A$ is the corresponding argument register, $N$ stores the number of recursion levels, $V$ is the vector built or copied and $T$ is the final tail of the vector (to be used in the recursive call).

POSLIST-K. The POSLIST-K class is compiled using **build_poslist** or **build_rec_poslist** since none of the variables in the list have occurred previously by definition. If this is the recursion list, the step of the loop iteration is given by $k$.

NEGLIST. The NEGLIST class has two subcases depending on whether the head variable is a subsequent occurrence. If this is the case, emit:

**build_neglist_value** $A, N, V, W, H$

is used. If the variable is a subsequent occurrence, emit:

**build_neglist** $A, N, V, H$

The arguments are the same as for the POSLIST case, with the addition that the $W$ argument denotes the vector to be written and $H$ denotes the head of the list rather than tail, again to be passed to the recursive call.

INV. An invariant argument pair is accessed by standard WAM instructions or by **global** instructions during parallel execution.

NONE-NEG. In the NONE-NEG case, the compiler simply allocates a vector of free variables for subsequent use. This is done through

**build_variables**$A, N, V, T$

which builds a vector-list of free variables, except for the first element, which is $A$. The other elements are as above.

In some programs, variables are created in the left body and used in the right body. In a standard WAM, they would be allocated in an environment to survive the recursive call. This is not done in the recursion-parallel case, where such variables are allocated using a **build_variables** instruction.

**Example.** The following program illustrates the use of all head unification instructions:

```
p([X|Xs],[Y|Ys],[X|Zs],A,K,   Ws,    Ts) :-
   p(Xs ,   Ys,    Zs, A,L,[W|Ws],[W|Ts]).
```

The compiled code becomes:

```
p/7/rec:
    build_rec_poslist X0,Xn,Xv1,Xt1
    build_poslist X1,Xn,Xv2,Xt2
    build_poslist_value X2,Xn,Xv1,Xt3
    get_value X3,X3
    build_variables X4,Xn,Xv3,Xt5
    build_neglist X5,Xn,Xv4,Xh6
    build_neglist_value X6,Xn,Xv4,Xv5,Xh7
    execute p(Xt1,Xt2,Xt3,X3,Xt5,Xh6,Xh7)
```

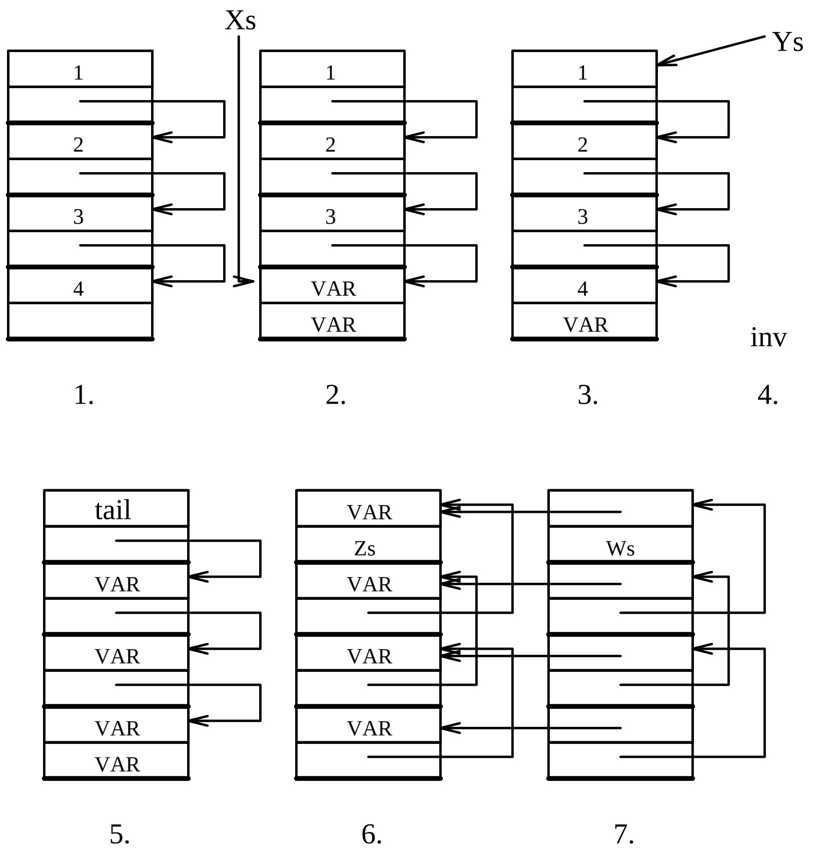

Figure 4.1: After executing build instructions

Register allocation then collapses moves from the **Xti** registers to the recursive call. Now, $X$ can be accessed relative to **Xv1**, $Y$ relative to **Xv2** and so on.

Let us consider the results of executing these instructions. Assume that p/7 is called as follows:

```
| ?- p([1,2,3,4],[1,2,3|Xs],Ys,inv,tail,Zs,Ws).
```

After the call, the heap looks like Figure 4.1. Now, recursion level 1 can access arguments at offset 1, level 2 at offset 2 and so on. Access to such arguments is further described in Section 4.5. □

It is straightforward, though tedious, to define the cases when the list is destructured or constructed in larger steps than one. In this case, the size and level registers must be manipulated to yield the correct number of recursion levels as

```
     start_left_body L1
  ...
L1: initialize_left S
L2: spawn_left S,L,N
     code for left body
     jump L2
```

Figure 4.2: Code for parallel execution of left body $\Phi$

```
     start_right_body L1
  ...
L1: initialize_right S,N
L2: spawn_right S,L
     code for right body
     jump L2
```

Figure 4.3: Code for parallel execution of right body $\Psi$

well as the correct offsets into the created vectors. Also, changing the step with the argument (e.g. p([X1,X2|Xs],Ys) :- p(Xs,[Y1,Y2,Y3|Ys])) requires adjusting the size register (in the example, divide by two then multiply by three to get the length of the Ys argument). If the step is non-unit, this also affects the process-starting instructions and the vector-list access instructions. The changes are relatively simple, and are omitted here.

## 4.5 SPAWNING RECURSION LEVELS

| | |
|---|---|
| **start_left_body** $L$ | **start_right_body** $L$ |
| **initialize_left** $S$ | **initialize_right** $S, N$ |
| **spawn_left** $S, L, N$ | **spawn_right** $S, L$ |
| **jump** $L$ | |

The instructions for starting parallel processes can be viewed dually: as if concurrently starting $N$ processes, or as specifying a loop from 1 to $N$, or $N$ down to 1, by some constant step $S$ (we assume $|S| = 1$ unless otherwise noted). The number $N$ is found from the recursion argument, e.g. by executing the **build_rec_poslist** instruction. Subsequently, we identify $N$ with the *recursion size register* holding it. The schematic code generated for a concurrent body is shown in Figures 4.2 and 4.3. We say that $L$ is the *recursion level register* and set by the **spawn** instructions. The recursion level register is used for accessing variables in vector registers, and is the current process number or dually the current loop counter value.

The same loop can be used for both statically and dynamically scheduled programs. In both cases, the **start** instructions act as barriers; all parallel workers must terminate before the **start** instruction finishes. The **initialize** instructions are used to compute the first iteration number to be executed. For the **left** version, this is the worker number. For the **right** version, this value is subtracted from the size $N$. (Both the left and right value must be adjusted to account for the **spawn** instruction to be executed immediately afterwards; this is straightforward.) Dynamically scheduled programs ignore these instruction, since the assignment of first recursion level to be executed is handled by conceptually stealing goals from a queue. The actual implementation consists of all workers atomically fetching and incrementing the 'loop counter'.

The **spawn** instructions implement the actual parallel loop. Statically scheduled programs have a private counter initialized by the **initialize** instructions, which is incremented or decremented by the worker number at each **spawn**. If this number is greater than the size $N$, or less than 1, the worker terminates. Dynamically scheduled programs find the same value as a shared variable, which is atomically incremented. When the counter is greater than $N$ or less than 1, the worker terminates.

Readers may convince themselves that these instructions specify a loop when interpreted sequentially. In this way, the Reform Prolog system could use the same compiler technology to optimize sequential execution; the difference is just that no suspension or locking instructions are emitted. The advantage of using the recursion-parallel instructions for sequential execution is that the overhead for recursion is lessened, while the drawback is that data that were previously on the stack are now heap allocated. This effect is caused by our use of **build** instructions rather than environment allocation/deallocation. This compilation method is currently not used in the Reform compiler.

## 4.6 ACCESSING RECURSION LEVEL ARGUMENTS

| | |
|---|---|
| **put_nth_head** $V, K, C, X$ | **unify_nth_head** $V, K, C$ |
| **put_nth_tail** $V, K, C, X$ | **unify_nth_tail** $V, K, C$ |
| **put_global** $G, X$ | **unify_global** $G$ |

In the execution of the left and right bodies, the recursion level must access the arguments constructed by the head unification phase. Arguments reside in vector registers or in global registers, and are to be moved into temporary registers (by **put** instructions) or written onto the heap as part of a structure (by **unify** instructions).

The head unification phase provides the compiler with information on where to find the values of source variables.

Vector variables have two possible locations. Recall that vectors are implemented as dense lists. For a vector $V$ with an index register $L$ and offset $K$, the particular list cell is accessed as $V[L+K]$. If the variable being retrieved occurred in the head of a list, it can be found as $V[L+K].hd$, while a variable occurring as the tail of a list is found as $V[L+K].tl$. For the described variable classes, $K=0$ or $K=1$. When the step $S$ is non-unit, some variables require a non-zero $K$ offset.

Global variables can be found in global registers, and are accessed by either **put_global** or **unify_global** instructions, depending on the destination of the value.

**Example.** Consider the program

```
p([],[],_).
p([X|Xs],[Y|Ys],N) :- q(X,Xs,Y,N), p(Xs,Ys,N).
```

Assume that the first argument is allocated to vector register **V** and the second argument to vector register **W**. Then the variables of recursion level $L$ can be found as:

`X`: $V[L+0].hd$;

`Xs`: $V[L+0].tl$;

`Y`: $W[L+0].hd$;

`N`: $G3$.

□

In the instruction set, **put_nth_head** $V, L, K, X$ or **put_nth_tail** $V, L, K, X$ are used for the assignments $X := V[L+K].hd/tl$. The **unify** instructions are used to move the vector value onto the heap.

## 4.7 LOCKING UNIFICATION FOR SHARED DATA

**lock_and_get_structure** $F, X, T$ **lock_and_unify_structure** $F, T$
**lock_and_get_list** $X, T$ **lock_and_unify_list** $T$
**unlock** $X, T$

Unifying an instantiated term with a variable means the variable is assigned the value of the term, which is usually written to the heap. In the case of a compound term, writing this value to the heap requires several WAM instructions. In a parallel system, this is a problem: if the value is not constructed atomically, other processes might encounter half-constructed terms on the heap. This would wreak havoc in the system, and must be avoided. The requirement that shared terms appear as if atomically constructed to all other recursion levels is called the *atomic write condition.*

We introduce new instructions for locking unification. If a locking unification instruction encounters a variable, the term to be built is constructed privately and atomically installed when the unification is completed (and the variable is unlocked). On the other hand, if the locking unification finds a nonvariable term, it behaves just like a standard unification instruction. This approach is similar to that of Naish [53].

A locking unification is done as follows:

```
lock_and_get_structure F/N Xi Xj
unify N arguments
unlock Xi Xj
```

The overhead of using locking instructions apart from the extra instruction, is that register allocation is worsened. In particular, the common optimization of moving a substructure of an argument register directly into the same argument register for the next call, is no longer possible due to the presence of **unlock**.

**Example.** The following code fragment from append/3 demonstrates worsened register allocation. Sequentially, the code is:

```
get_list X0
unify_variable X3
unify_variable X0
```

Using locking unification, the code becomes:

```
lock_and_get_list X0 X4
unify_variable X3
unify_variable X5
unlock X0 X4
put_value X5 X0
```

```
if   $X$.type is possibly uninstantiated
     and $X$.loc is shared
then use locking unification
```

Figure 4.4: Testing whether locking unification code is required

The reason is that the **unlock** instruction blocks the reuse of register X0. □

For these reasons and in an attempt to generate the same code as for WAM, the compiler avoids the use of locking unifications, compile-time analysis allowing. Unifications $X = a$, where $a$ is a constant, do not require locking, since they are assumed to be atomically written. General unifications handle potential locking implicitly. Only the (flat) case $X = f(T_1, \ldots, T_n)$ needs to be considered. The test is shown in Figure 4.4 and the justification is the following: since $X$ may be uninstantiated and $X$ is shared, the atomic write condition must be satisfied. The unification is flat, so the terms $T_i$ are atomic symbols or first occurrence variables. Such variables are known to be local. (Non-local variables cause general unifications.)

## 4.8 SUSPENDING RECURSION LEVELS

| | |
|---|---|
| **await_nonvar** $X$ | **await_nonvar** |
| **await_strictly_nonvar** $X$ | **await_strictly_nonvar** |
| **await_leftmost** | |

Suspension instructions may be introduced in the left and right bodies of the recursion-parallel predicate as well as in all predicates called by the recursion-parallel predicate.

A major source of suspension is unification within a recursion level, and the discussion will concentrate on this operation. Primitives and indexing also cause suspension, and the compiler removes suspension instructions from these operations by straightforward use of the techniques used below.

### 4.8.1 Compilation of suspending unifications

Before every flat unification against a register $X$ with fragile contents, the compiler emits:

**await_nonvar X**

The **await_nonvar X** instruction suspends until the heap location pointed to by $X$ becomes a nonvariable, or the recursion level becomes leftmost (or “first”) of all recursion levels.

If $X$ becomes nonvariable, the unification is a test and can proceed. If the recursion level becomes leftmost, the situation for the level is precisely that of sequential execution. At this point, the recursion level may bind variables indiscriminately, since it cannot affect preceding recursion levels. In particular, the unification following the **await_nonvar** instruction is done.

If the system possibly is in a nondeterministic state, and thus the unification possibly unsafe, the compiler prefixes each unification of a register $X$ holding a shared variable with an instruction

**await_strictly_nonvar X**

The instruction suspends if $X$ is a variable at the time of the test. If the recursion level becomes leftmost while $X$ is still a variable, it terminates execution with a runtime error. If $X$ is bound before the recursion level becomes leftmost, execution proceeds. Terminating with an error is required since binding the variable would mean conditionally binding a shared variable, which is prohibited by the execution model. So the **strict** instruction ensures that unifications are tests.

When a general unification $X = Y$ is to be done and either $X$ or $Y$ is fragile, the recursion level must suspend, unless being leftmost, since $X$ and $Y$ can be instantiated unpredictably by a general unification. This is done by emitting

**await_leftmost**

(If the state is nondeterministic, the analysis phase will have flagged the unification as unsafe already; in that case, an unknown number of variables could be bound conditionally by the unification.)

The compiler has previously extracted type, locality and determinacy information from the program. Using this information, it is possible to remove many suspension instructions.

Recall that suspension is necessary if the unification may bind a fragile variable (sequential semantics), or if a conditional binding to a shared variable may be done (safeness). If the compiler can ensure that these situations will not occur, the suspension instruction can be removed.

### 4.8.2 Optimizing flat unifications

The compiler keeps track of whether previous **await_leftmost** instructions have made the recursion level become leftmost. In that case, no more **await_leftmost**

```
if   recursion level is deterministic
     and $X$.loc fragile
     and $X$.type possibly uninstantiated
     and recursion level not leftmost
then emit await_nonvar Reg[X]
else
if   $X$.loc robust
     and recursion level is nondeterministic
     and $X$.type possibly uninstantiated
then emit await_strictly_nonvar Reg[X]
else skip (no suspension)
```

Figure 4.5: Generating suspension code for the flat unification $X = f(T_1, \ldots, T_n)$. *Reg[X]* denotes the register holding $X$.

or **await_nonvar** instructions need to be emitted. However, **await_strictly_nonvar** instructions cannot be simplified away since they also test safeness.

We use two tests to determine whether to suspend: one to suspend for sequential semantics, the other to suspend for safeness. In practice, both tests are combined into one. The tests are shown in Figure 4.5. The sequential semantics condition is motivated as follows: if $X$ is fragile and the recursion level is not leftmost, then some preceding recursion level may depend on the instantiation of $X$. In that case, binding $X$ would violate sequential semantics. However, if the recursion level is known to be leftmost, binding can be done immediately.

The second test ensures that a unification in a nondeterminate state is only a test: if the variable is shared and possibly uninstantiated, then the nondeterministic state may create a conditional binding. (This is true regardless of whether the computation is leftmost or not.)

**Example.** For instance, consider the second clause of the append/3 program.

```
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

We will compile the clause for use in a parallel setting, with varying knowledge of surrounding processes.

First assume that no information is known except that the state is determinate. In this case, the compiler must assume that all arguments have unknown parallel and sequential types and are fragile. The following code is generated:

```
% append( fragile any, fragile any, fragile any)
append/3/2:
    await_nonvar X0              %
    lock_and_get_list X0 X4      % *A1 = [
    unify_variable X3            %        X|
    unify_variable X5            %          Xs
    unlock X0 X4                 %            ]**
    put_value X5 X0              % A1 := Xs
    await_nonvar X2              %
    lock_and_get_list X2 X4      % *A3 = [
    await_leftmost               %
    unify_value X3               %       *X|
    unify_variable X5            %          Zs
    unlock X2 X4                 %            ]**
    put_value X5 X2              % A3 := Zs
    execute append/3             % execute append(A1,A2,A3)
```

We denote suspensions with a single ∗ while locking unifications are terminated with ∗∗ in the comments.

There are three suspension instructions, and the general unification forces sequentialization by the **await_leftmost** instruction. Furthermore, both of the unifications against compound terms must lock their arguments.

Let us try again with more compile-time knowledge. Let us assume that the first argument is known to have nonvariable parallel and sequential types and is shared. The second argument is uninteresting for our purposes while the third argument is a free variable that is shared. The computation is still assumed to be deterministic.

The code of this version of append is:

```
% append( shared nv , any, shared free )
append/3/2:
    get_list X0                  % A1 = [
    unify_variable X3            %        X|
    unify_variable X0            %          A1]
    lock_and_get_list X2 X4      % A3 = [
    unify_value X3               %        X|
    unify_variable X5            %          Zs
    unlock X2 X4                 %            ]**
    put_value X5 X2              % A3 := Zs
    execute append/3             % execute append(A1,A2,A3)
```

```
If   recursion level is not leftmost
     and $X$.loc is fragile
     and $Y$.type may instantiate $X$.type
then emit await_leftmost
```

Figure 4.6: Emitting code to suspend general unifications. This test is performed for both $X = Y$ and $Y = X$ in a general unification $X = Y$.

Even though the arguments are shared data, the compiler can simplify away all suspension instructions. Nine instructions are required, compared to the fifteen instructions required for the general (deterministic) case.

For the first suspension, the unification with the first argument was found to be a test because of the nonvariable type, and so both the suspension instruction and the locking unification could be simplified away. The third argument was known to be shared but not subject to time-dependent operations, so the suspension operations could be removed. However, the locking unification is still required since the argument was a free variable (Section 4.7). □

### 4.8.3 Optimizing general unifications

For a general unification $X = Y$, the compiler tests if it must suspend on account of $X$ being instantiated by $Y$ or vice versa. The test for possible suspension if $Y$ may instantiate $X$ is shown in in Figure 4.6.

A general unification instantiates terms unpredictably. If the term being instantiated is fragile, this could lead to a violation of sequential semantics so the compiler elects to suspend until this danger has passed (i.e. until leftmost).

We note that if the state is nondeterministic, a general unification could be unsafe, since we cannot predict what variables will be bound. The compiler then flags the unification as unsafe (at compile-time) and elects not to parallelize the computation.

An alternative to these tests would be to introduce specialized versions of general unification that take care not to bind variables in the first, second or both of the arguments. When leftmost, the appropriate action can be taken depending on whether the state is nondeterministic or not. Such instructions are currently not provided by the engine, but would not be hard to implement. If their absence becomes a liability, such instructions should be added and the test above be replaced with these unification instructions.

### 4.8.4 Suspending on the heap pointer

The final refinement to this scheme concerns the use of flat terms. The goal of the compiler is to generate almost the same code as the WAM, but using the definition of flat terms above, this is not possible for some terms. Consider the term $[a|Xs]$.

In the WAM, compiling the unification of register X0 with this term yields:

```
get_list X0
unify_constant a
unify_variable X1
```

However, we have previously insisted on compiling only flat unifications, since the suspension instructions require a temporary register to work with. The compiler would have to break this unification up into $X_0 = [X_2|Xs], X_2 = a$. This yields the code (sans locking unifications):

```
await_nonvar X0
get_list X0
unify_variable X2
unify_variable X1
await_nonvar X2
get_constant a,X2
```

It is easy to remedy the situation. We introduce two instructions that use the $S$ register of the WAM for suspension test instead: **await_nonvar** and **await_strictly_nonvar** of zero arguments. With these instructions, the code now becomes:

```
await_nonvar X0
get_list X0
await_nonvar
unify_constant a
unify_variable X1
```

### 4.8.5 Discussion

In assessing the suspension scheme, we note that different suspension instructions are required depending on whether the computation is deterministic or not. If the predicate can be entered deterministically as well as nondeterministically, the compiler must emit two versions of the predicate. Thus, code size could be almost doubled in the worst case.

If the size of the generated code is a problem, suspension instructions could be made conditional. Depending on whether the engine is in a nondeterministic state or not, strict or standard suspension would be enforced. The advantage is that only one version of the suspension operations is required, while the drawback is that an extra test is required on every suspension instruction. More seriously, the nonstrict case of suspension would execute more (vacuous) instructions, since more instructions can be eliminated in the non-strict case which must be retained in the strict case. A useful compromise would be to introduce these conditional suspension instructions and use them only where analysis information is insufficient, or only when nondeterminate entry of a clause is possible.

## 4.9 RECURRENCES

The compiler attempts to distinguish arithmetic recurrences in recursion-parallel predicates, since they are otherwise a source of unnecessary suspension. The compiler recognizes four kinds of recurrences:

- Reductions, on the form X is $E + Y$ or X is $E * Y$ where X and Y form an argument pair and neither X nor Y appear in the body. Reductions are compiled to expressions that compute:

$$(Y + \sum_{i=1}^{n} E_i) \qquad \text{or} \qquad (Y \cdot \prod_{i=1}^{n} E_i)$$

- Scans, that are reductions where X or Y appear elsewhere in the left or right body of the predicate. Intermediate results must then be retained and allocated to a vector.

- Inductions are defined as expressions $X$ is $Y+C$ or $X$ is $Y*C$, for constants $C$. They are compiled to expressions dependent on the recursion level counter. (The name 'induction' is used in the same sense as induction variables in the compiler literature, which are variables that always can be expressed in terms of other, basic, variables.)

- Integer recursions, that are special cases of inductions. Integer recursions are used to set the size register by computing an expression of the recursion argument.

### 4.9.1 Compiling recurrences

Integer recursion is compiled into instructions that set the recursion size register, followed by code as for an equivalent induction. Occurrences of induction variables in the left or right bodies are translated into arithmetic operations using the level register, while the value for the recursive call is computed using the size register.

Reductions can be compiled according to the following principle, which is a modification of that used by Barklund and Bevemyr in coding bounded quantifications [4]:

1. The sequential worker executes a primitive that creates a vector-list with a length equal to the number of workers, and initializes each entry of the vector to an appropriate value (for summations, 0; for products, 1). This is the temporary value table.

2. Each recursion level performs the appropriate arithmetic and then calls an accumulation primitive, which updates the worker's entry in the temporary value table.

3. When all parallel workers are finished, the sequential worker computes the product or sum of the temporary result table, arriving at the total value. If there was a non-unit initial value to the reduction (e.g., an input argument of 3 to a summation), this value is added or multiplied to the total. The result is placed in the appropriate register.

Thus, the reduction is compiled into $N$ workers producing partial results independently, followed by a fast phase of final summation. We could view this as the operations $x+x_1+x_2+\ldots+x_n$ being computed as $(x_1+\ldots+x_{n/w})+\ldots+(x_{n-n/w}+\ldots+x_n)$. (We assume $n \bmod w = 0$, the generalization is obvious.) Clearly, the reduction operation must be associative for this scheme to work. Since we limit reductions to sums and products, this is obviously the case.

Scan operations compute the same result as reductions, but differ in that the intermediate results are retained. Thus, on recursion level $i$, the partial result $x+x_1+\ldots+x_{i-1}$ is required. The compiler cannot take advantage of associativity to compute results 'ahead of time'. To avoid having each recursion level await its predecessors, a scan operation is coded as a primitive call, which is run before parallel execution begins and produces a vector-list of the required partial results. The generated list is then accessed just like any other vector-list when the partial result is required.

This scheme for coding recurrences is not incorporated in the compiler described in this thesis.

### 4.9.2 Some examples of recurrences

It is instructive to consider some programs containing inductions, integer recursion, reductions and scan operations.

**Example.** The second argument of the predicate length/2 is an induction:

```
length([],0).
length([X|Xs],N) :-
   length(Xs,M),
   N is M+1.
```

The code for the predicate is simple:

```
...
length/2/2:
    build_rec_poslist X0,Xsz,Xv,Xt
    $X2 := Xsz * 1 + 0$
    proceed
```

Note that this definition does not allow the generation of lists from a number, so directionality is not preserved. The compiler only generates this code if length/2 is declared parallel and the engine only executes the code if the first argument is a non-variable. If the first argument is a variable, the sequential code is called. □

**Example.** More complex examples are possible. For instance, the first argument of nrev∗/3 is an integer recursion.

```
nrev*(0,[],[]).
nrev*(N,[X|Xs],Zs) :-
   N > 0,
   M is N-1,
   nrev*(M,Xs,Ys),
   append*(N,Ys,[X],Zs).
```

In each recursion level, M and N can be computed as an expression of the first argument and the recursion level register. □

## 4.10 RELATED WORK AND DISCUSSION

### 4.10.1 Head unification

A previous study by Singhal and Patt concluded that the parallelism available in unification is quite small for normal programs since the terms involved are usually small [64]. Simulated speedups of a factor two were reported. However, Reform compilation collects a number of small unifications into a single, large one. By performing all the head unifications of the predicate at once, ample fine-grained parallelism is exposed.

In our current benchmark set, which emphasizes medium- and coarse-grained parallelism, the time spent in head unifications is small (on the order of 0.5% or less of the execution time [8]). On the multiprocessors used in our experiments, with 4 and 24 processors, this time was insignificant.

If the large unification instructions were to become a bottleneck, there are several things that can be done. Most of the **build** instructions contains large amounts of fine-grained parallelism. First, the tails of the vector list can be constructed in parallel, since the pointers are completely regular and the size of the vector is known in all cases but for the recursion argument. Furthermore, the heads of the vectors can often be constructed in parallel (e.g., for the non-**value**) instructions. In several cases, this is trivial since execution either creates $N$ variable cells, $N$ pointers, or the equivalent.

Furthermore, several **build** instructions can execute in parallel, as long as the instructions respect data dependences.

### 4.10.2 Creating processes

In other and-parallel systems, process spawning is gradual and dynamic. Predicates locally make decisions of whether to spawn new processes, depending on runtime information. The result is a tree of communicating processes that dynamically grows and shrinks. Management of this process tree can be quite complex.

Our model is different in that *all* processes to be executed are constructed at once, in the head unification phase. Finding new work is extremely simple, e.g., in a statically scheduled system by incrementing a counter and jumping conditionally to the beginning of the loop.

### 4.10.3 Suspension

Previously, most of the work on synchronization of dependent and-parallel languages has been set in the context of the concurrent logic languages (Parlog, Concurrent Prolog and GHC). Recently, parallel implementations of dependent and-parallel languages based on Prolog have appeared, e.g., Andorra-I.

In general and-parallel systems the objective is usually to maximize the amount of parallelism extracted. Thus, sequential execution is avoided as far as possible. Terms typically are unshared only when first initialized and then intermittently (e.g., if the system can somehow detect that all other sharing processes have terminated).

Kusalik [41] describes two method to serialize Concurrent Prolog programs. Concurrent Prolog lacks any serialization mechanism other than the commit operator, which requires all goals to the left of it to have terminated before solving goals to the right.

The first method relies on defining an operator $\triangleright$ by the (Concurrent Prolog) clause:

$$A \triangleright B \leftarrow A|B$$

The commit operator $|$ then ensures that $B$ will execute after $A$.

The disadvantage of this definition is that all of $B$ must await the completion of all of $A$. The $\triangleright$ operator is used to suspend $B$ until variables shared between $A$ and $B$ have been used by $A$, but for situations where $A$ can release the shared data before terminating, $B$ will suspend longer than necessary.

Kusalik's second method therefore is considerably more fine-grained, and relies on *control tokens* to communicate suspension information. Procedures are modified to take an extra argument for each shared variable. For two processes $A$ and $B$, $B$ suspends on the control token argument while $A$ uses the shared variable. When $A$ is finished with the shared variable, the corresponding control token argument is instantiated and $B$ can proceed.

Control tokens can be very useful to the programmer; for use in a parallelizing compiler there are several difficulties. The compiler must rewrite the program to use control tokens, which might be a difficult task. Relying on static means such as program rewriting to control suspension, by necessity requires approximation of when to suspend. Control tokens can model some situations better than the Reform Prolog synchronization scheme: when a variable is released from use, suspended computations may be resumed at once (rather than when becoming leftmost). On the other hand, control tokens require explicit release of the token which may be a considerable overhead if many tokens are used.

Crammond [23] describes the suspension machinery for an implementation of parallel Parlog, a concurrent logic language. Suspension is controlled by abstract machine instructions that are versions of the **get** and **unify** instructions of the WAM that suspend on unbound variables. Thus, these operations cannot be strength-reduced to their simpler sequential counterparts.

Much like Reform Prolog, the Andorra-I language as implemented by Costa, Warren and Yang [58], is an implementation of dependent and-parallel Prolog. Andorra-I extends Prolog with coroutining and constraint solving, which is not done in Reform Prolog. Furthermore, Andorra-I too bases suspension on compile-time analysis [57].

To handle the impure operations of Prolog, Andorra-I uses *sequential conjunction* to separate sensitive goals. The preprocessor uses the results of a preceding type inference phase to generate modes to find what calls are *sensitive*. Sensitive procedures include those where cuts may cause shared variables to be incorrectly restricted unless care is taken. Furthermore, primitives with side effects require all subsequent calls to delay[2], and the usual type tests are marked as sensitive. Finally, all the transitive callers of sensitive calls are marked as sensitive.

The Andorra-I preprocessor then introduces sequential conjunctions to the right of all sensitive calls. This ensures that time-dependent operations still executes correctly.

For instance, consider the following clause [57]:

```
p(X,Y) :- a(X,Y,W), b(X,W,Z), c(X,Z).
```

Assume that b/3 is sensitive. Then the preprocessor rewrites p/2 into the following clause, where the double-colon connective denotes sequential conjunction.

```
p(X,Y) :- a(X,Y,W), b(X,W,Z) :: c(X,Z).
```

This means c/2 will start only after both a/3 and b/3 have terminated. The use of sequential conjunction in Andorra-I is similar to Kusalik's first synchronization method and has similar disadvantages. Remember that the Andorra-I preprocessor marks all callers to a sensitive call as being sensitive transitively; the net result is to create a barrier synchronization of sequential conjunctions throughout the call tree. The solution in Reform Prolog allows for shorter suspension, in that time-dependent operations do not act as barriers.

Shen's DDAS [62] uses yet another scheme for synchronization between dependent goals. DDAS extends the conditional graph expressions [35] of independent and-parallelism with a new pseudo-test, dep/1. All variables in the argument of dep/1 are lazily marked as dependent, and only the leftmost process in the computation may bind dependent variables. All other processes suspend on encountering a dependent variable, until it is instantiated or they become leftmost. The execution model is thus viewed as an accelerator of standard Prolog computations: the leftmost process, corresponding to standard Prolog execution, executes normally while all other workers attempt to transparently solve non-leftmost goals.

The overheads of the DDAS scheme as described are the following:

- Suspension is handled inside the engine, so the compiler cannot reduce the cost of using dependent variables.

[2]Also, side-effects are only done when leftmost.

- The shared terms in the dep/1 call must be traversed and all variables marked as dependent (as well as unmarked when exiting the scope of the dep/1 call). Since there may be nested dep/1 declarations, this may be expensive.
- No distinction between fragile and robust terms is made; and it is unclear how time-dependent primitives are to be accomodated.

At present, dep/1 calls are introduced by the programmer. However, Shen notes that a compiler might do the job, similarly to compilers for independent and-parallel languages.

### 4.10.4 Locking unification

Most of the implementations seem to use the simple strategy of constructing the term and performing a general unification, when atomic writes are required.

For instance, Crammond notes [23]:

> When constructing a structure term in output unification, it is necessary to construct a complete term before unifying it with a goal call argument since some other process may start accessing it immediately. The usual WAM instruction sequence used in sequential Prolog ...is therefore inappropriate (p. 399)

In concurrent logic languages, this may be an acceptable solution since the status of a unification is known: it is either a test or an instantiation operation. However, in languages such as Prolog, unification may be both. One of the principles of the WAM is to compile away general unification as far as possible. The scheme used by Reform Prolog retains this property by using locking unification with special instructions. The benefits of this decision are that locking unifications can be replaced with standard unification instructions by the compiler, as described in Section 4.7, as well as retaining compiled unification even in the case where atomic writes are required.

For some methods to implement the **lock** and **unlock** instructions, consult Bevemyr's Licentiate thesis [8].

### 4.10.5 Recurrences

The advantage of encoding reductions as we do is that each worker can compute the partial result over its allocated recursion levels independently of the others, and compiled arithmetic can be retained. The sequential portions in creating an intermediate result vector and summarizing it should be quite small, though we have not measured them. We note that the vector building instructions typically create larger vectors and still require less than 1% of the execution time on the benchmarks.

The interpretive element of a scan, in that a term represents the arithmetic operations to be performed, could be eliminated by either compiling the scan into loop code similar to that used in sequential implementations of bounded quantifiers [5]. The sequential portion could be reduced by parallelizing the primitive scan operation at least partially. Neither of these approaches have currently been implemented.

# Evaluation

We have implemented the ideas presented in this thesis in a prototype compiler. While work still remains to be done, we present some preliminary results and point out some strengths and weaknesses we have found.

## 5.1 BENCHMARKS

We have compiled and executed a set of small and medium-sized benchmark programs to evaluate the compiler as a whole.

**Map.** This program applies a predicate to the elements of two ground lists. In the benchmark, the applied predicate simply counts to 100. The program illustrates the performance on simple independent and-parallel programs.

**Nrev.** The naive reverse program reverses a list in quadratic time. Each recursion level is an append/3 goal. The processes form a pipelined parallel computation.

**Nrev$*$.** This is a version of the naive reverse program originally due to Millroth. A call to the program supplies an extra size parameter, which drives the recursion. The effect is to give recursion levels more freedom to construct the list, since the data dependence between levels is broken. The program is given in Section 4.9.2.

**Tree.** This program inserts a list of key-value pairs in a tree with uninstantiated leaves, shared by all recursion levels. Each recursion level can (due to safeness) only insert its value in the tree when leftmost; it can, however, traverse the tree prior to that point, as long as no variables are bound. The code is a subpart of the Match program.

| Program | Sequential | One worker | 4 workers |
|---|---|---|---|
| Map | 7.32 | 7.37 | 1.88 |
| Nrev | 3.80 | 6.71 | 1.84 |
| Nrev∗ | 7.20 | 9.19 | 2.65 |
| Tree | 3.63 | 4.16 | 1.18 |
| Tsp | 37.39 | 40.12 | 11.94 |
| Match | 8.94 | 11.33 | 2.65 |

Figure 5.1: Execution times of benchmarks in seconds

**Tsp.** This program solves the travelling salesman problem using an approximation algorithm that guarantees a solution at most twice the cost of the optimal solution.

**Match.** The Match program attempts to find the best match of a string using dynamic programming. The application is in DNA matching. The results are collected in a tree shared by all recursion levels.

## 5.2 EXECUTION TIMES

We measured execution times by running the program compiled for sequential execution using a sequential implementation, running the parallelized program on one worker and running the parallelized program on 4 workers. The benchmark machine was a Sun Galaxy with 4 40-MHz Sparc processors and coherent caches. Each benchmark was run five times. Since benchmark timings varied considerably, we chose the best result in each category.

We compute the normalized speedup $N(P)$ of a program $P$ as the sequential time divided by the execution time on four workers. The speedup $R(P)$ relative to the parallel implementation is computed by substituting execution time on one worker for the sequential execution time. The normalized efficiency $NE(P)$ is the normalized speedup divided by the number of workers, while the relative efficiency $RE(P)$ is the relative speedup divided by the number of workers $w$. In our case, $w = 4$.

$$N(P) = T_s(P)/T_w(P)$$

$$R(P) = T_1(P)/T_w(P)$$

$$NE(P) = N(P)/w$$

$$RE(P) = R(P)/w$$

Speedups on four processors ranged from approximately 2.0 to 3.9 normalized to

| Program | Normalized | Relative |
|---|---|---|
| Map | 3.89 | 3.92 |
| Nrev | 2.07 | 3.68 |
| Nrev∗ | 2.72 | 3.47 |
| Tree | 3.08 | 3.53 |
| Tsp | 3.13 | 3.36 |
| Match | 3.37 | 4.28(!) |

Figure 5.2: Speedups normalized to sequential engine and relative to parallel engine

| Program | Normalized | Relative |
|---|---|---|
| Map | 97 | 98 |
| Nrev | 52 | 92 |
| Nrev∗ | 68 | 87 |
| Tree | 77 | 88 |
| Tsp | 78 | 84 |
| Match | 84 | 107 |

Figure 5.3: Efficiency of parallel engine compared to sequential engine in percent

a sequential engine, with most speedups greater than 3.0, while relative speedups were always greater than 3.4 . The anomalous superlinear relative speedup of the match benchmark seems to be due to the Sun machine giving a poor one-worker result. The precise reason for this behavior is unclear.

Relative efficiency is quite good at 87% or better. Normalized efficiency was generally around 80%, with a notable low point in Nrev, with an efficiency of 52% and Nrev∗ with an efficiency of 68%. We discuss the reasons for this below.

### 5.2.1 Comparison to previous implementations

Bevemyr, Millroth and the author have previously described an implementation of Reform Prolog for the Sequent Symmetry [9], where some of the above benchmarks were executed. There are some differences in the code executed as well as benchmarks, which we further describe below. However, there is some interest in comparing the benchmark results for the two implementations.

The Sequent Symmetry used for the previous study had 24 Intel 386 processors in a bus based cache coherent system. Some low-level implementation details differ in the implementations, since the Sun and Sequent machines use different methods to implement atomic operations. Our measurements were made with execution engines compiled with low-level debugging support. Without debugging support, execution engines become approximately 50% faster.

The results we reported for the Sequent machine were considerably better than those for the Sun machine in most respects except absolute runtimes, where the Sun machine achieved approximately the same runtimes on 4 processors as the Symmetry on 24 processors.

We executed the original, hand coded **Nrev** benchmark on the Sun machine and using the same implementations. We measured that $T_s = 4.37$, $T_1 = 5.13$ and $T_4 = 1.41$. This yields $R = 3.64$, $N = 3.10$, $RE = 0.91$ and $NE = 0.77$. Thus, the relative speedup and efficiency is almost identical for both versions while the normalized values are worse for the compiled program. We also note that hand coded **Nrev** is somewhat less efficient on the Sun machine than on the Sequent machine. (We have previously measured a normalized efficiency of 0.83 for **Nrev** on the Sequent.) This may be due to each worker being substantially faster on the Sun.

### 5.2.2 Quality of generated code

We compared the generated code with hand coded versions of the programs [9, 10], which were generated by manually introducing suspension and locking unification instructions.

In the case of **Map**, the code was identical to the hand coded version.

For **Nrev**, the compiler did not exploit type information to avoid generating a lock instruction, which was done when hand-coding the program. The code fragments where the programs differ are the following:

```
/*  Hand coded version            Compiled version
    ------------------            ---------------- */
    put_list X4                   lock_and_get_list X2 X4
    unify_value X3                unify_variable X5
    unify_variable X5             unify_variable X6
    get_value X2 X4               unlock X2 X4
    put_value X5 X2               get_value X3 X5
                                  put_value X6 X2
    execute append/3              execute append/3
```

Bevemyr (personal communication) has told us the reason for the poor performance of the compiled program is probably due to a bug in the execution engine, which makes locking unifications less efficient. We note that the compiler could have generated the same code as for the hand coded version if it had taken available type information into account.

The **Tsp** and **Match** programs were modified from the hand coded versions and so could not be compared directly. We note that in the **Match** program, suspension and locking instructions were used only when values were inserted in a shared tree of results. Since all other data were local, the compiler successfully eliminated all parallelization overhead.

In the **Tsp** benchmark, the recursion levels share a ground matrix of distances and the current shortest path and its cost. Process communication appears when the path on the current recursion level is compared to that of previous levels. Here, the compiler generates an instruction to await the arrival of the previous minimum, whereafter the process can continue without synchronization. The hand coded version of **Tsp** is less efficient: it employs **await** instructions in several places. The compiled version of **Tsp** had better execution times than the hand coded version, both on one and four workers. This is partly due to less suspension instructions and partly due to improvements in the compiler.

For the **Tree** program, the compiler generated better code than a hand-coded version, due to an unnecessary locking unification in the hand coded version. On the other hand, the compiler did not use type information to eliminate a superfluous primitive call, and emitted a useless suspension instruction. Since this occured in the base case, the impact on execution time was small.

The **Nrev**$*$ benchmark produced the same code as the hand coded program. This program probably suffers from the same bug as **Nrev**.

These results indicate two defects in the compilation algorithms: the compiler should take into account types when sequential code is generated, and should use global knowledge to eliminate repeated suspension on the same register and to use type information to realize when a suspension will be until leftmost.

The compiler was quite successful in eliminating synchronization and locking instructions. In two cases, the compiler could actually produce better code than hand coded versions of the program.

## 5.3 DISCUSSION

The performance of the entire compiler is at the time of writing mostly limited by the quality of analysis results. We summarize the good and bad points of the implementation below.

- Metalogic tests and predicates are handled poorly, for implementation reasons. If the abstract domain would trace whether a predicate instantiates a data structure, more precision could be obtained. The implementation currently keeps track of instantiation to some extent, but a more systematic treatment could be beneficial.
- The structure of terms other than lists is lost, which can give poor results for some programs, where parts of a term are always ground.
- Abstract indexing and open-ended list types sometimes interfere to yield a suboptimal type. This effect arises from lists having zero or more elements. Open lists thus include the free variables, while indexing determines that the list must be instantiated.
- Possible aliases are somewhat too pervasive at times. Certain aliases arise rarely, except for difference list programs.
- The quality of analysis results for real programs is very dependent on the results generated by primitive operations.
- Parallel analysis should take synchronization into account to yield better analysis results. (The annotation phase at present compensates this, but analysis results per se could make use of such extra information.)
- The linearity domain is often very efficient on the programs tested. Structures are often linear and arguments seldom share variables.

- The proposed list domain is quite efficient at analysing difference list programs. Apart from our benchmarks, precise results are obtained for programs such as the Dutch National Flag and Quicksort-with-difference-lists programs.
- The locality domain seems quite sufficient for the benchmark programs. Fragility and sharedness properties reflect the perfect results quite closely. Ground data are very efficient to remove fragility.
- Abstract indexing works very well to find determinism and restrict types so far. The limitation is that indexing is fixed prior to compilation. A generalization could be useful for better results. We note, however, that many programs already are written to be deterministic.

The compiler produces the expected results given the analysis results, apart from some implementation weaknesses. Extra suspension and locking instructions arise partly from the analyser deriving imperfect information, and partly from the compiler. The compiler does not reason in depth on types and suspension, and so generates suboptimal code at times. For instance, some suspensions are known to be until leftmost, but the compiler does at present not consider this. The result is that superfluous suspensions are generated. In the same way, the compiler may suspend several times on the same register, if the types of the register are weak. Both of these effects are implementation dependent and so could be corrected if they prove to practically limit performance.

So far, we have found the **await_strictly_nonvar** instruction to be generated rarely. Abstract indexing often eliminates a suspension and locking unification on each call, where shared data are used for indexing.

Our experience suggests that the general unification instructions with implicit suspension suggested in Chapter 4 can be useful and that the **await_leftmost** instruction should be eliminated.

Suspension and locking unification instructions are not very expensive when the data structure turns out to be instantiated. Hence, poor analysis results do not spell disaster. However, their presence means extra instruction decoding and (in the case of using **unlock**) worse code.

# Conclusion

## 6.1 SUMMARY

We have described a recursion-parallel programming language, Reform Prolog, and an execution model for Reform Prolog that employs the concepts of recursion-parallelism, safeness, Reform compilation and a balanced compiler-engine implementation to get an efficient and simple system.

We have developed an abstract domain that unifies type inference, aliasing and linearity information to provide precise analyses. We have furthermore introduced safeness analysis and locality analysis. The latter gives the compiler the means to strength-reduce expensive operations when a simpler implementation is implied by the available information. The simplicity of Reform Prolog, coupled with the sequential semantics, means the analysis of a parallel program becomes much more informative than in less tractable concurrent languages. Our abstract domain also handles difference lists with precision, which previous efforts, to our knowledge, have not reported.

We have built a compiler that takes advantage of the information from the analysis. The compiler attempts to remove sources of parallelization overhead by examining the locality and types of the involved terms, and can generate precisely the sequential code of a WAM when the involved terms are guaranteed to be local to the process. The compiler can eliminate instructions that lock heap cells as well as suspension instructions, even when terms are shared. Since the language is single-assignment in parallel, accessing bound terms never requires suspension.

## 6.2 FUTURE WORK

We outline possible future work in three areas: extending the execution model, improving the compiler and improving the analyser. From the pragmatic point of view, a thorough evaluation of the components must be made before we can comment on the practicality or usefulness of some of the proposals below.

**Execution model.** The execution model can be extended in several directions. First, we may consider extending the language. Currently, delayed computations are not supported by Reform Prolog. Adding freeze/2 and related constructs complicate semantics as well as execution; nevertheless, such a construct can be highly useful, e.g., to provide logically more appealing primitive operations. Extensions to handle suspended computations are projected in a joint COMPULOG project with the Gödel team at Bristol University and include support in the compiler as well as in the engine.

At the time of writing, we have not made an extensive evaluation of the system as a whole and the ease of programming in Reform Prolog. Should performance or expressiveness prove disappointing, there are several directions possible to extend the current system.

First, to extend the current model, suspension could be further differentiated. In particular, we distinguish only leftmost and non-leftmost when considering binding variables. Some programs only share between adjacent recursion levels and so could profit from refinement of the suspension criteria. Second, we could lift the restrictions of the language to incorporate nested recursion-parallelism. The changes to compiler and engine are likely to be extensive. Third, there are restrictions to the language to eliminate troublesome constructs, such as call/1, assert/1 and others. One could envision lifting these restrictions. Fourth, we could combine recursion-parallelism with other forms of parallelism. In particular, or-parallelism such as in Muse [2] might be a useful addition.

**Compilation.** One of the points of Reform Prolog is that absolute performance in a restricted execution model is preferrable to more general but slower models. In line with this observation, we consider working on native code compilation to develop a fast implementation competitive with top-of-the-line sequential systems. Included in this goal are exploring alternatives to the WAM model for Prolog, exploiting local instructions and building a system which can exploit the results of more powerful analyses than those developed in this thesis.

**Analysis.** Our compile-time analysis can be extended in several respects. First, for the purpose of native code generation, a compiler requires low-level information on whether trailing and dereferencing operations are necessary, and so on. Second, the types derived by our analyser are perhaps too simple. Janssens and Bruynooghe propose type graphs [38], that can describe arbitrary inductively defined terms. Suitably extended, e.g., to handle difference lists, such a concept could prove the basis of a better abstract domain. Third, from the practical perspective, our representation of aliasing is inexact when compared to other, non-transitive, approaches [52]. Should these provide a markedly better precision than ours, when used on real programs, it may be useful to investigate such better methods. Fourth, to analyse large programs, there are software engineering considerations in that analysis currently can use prohibitive amounts of memory. In order to evaluate large programs, we may have to consider more efficient implementations of analysers and abstract domains. In particular, the implementation of the abstract domain is sufficiently large and complex already to preclude easy modification. Further work on efficient representations and algorithms would certainly be useful in this respect.

## 6.3 RELATED WORK: IMPERATIVE LANGUAGES

We can compare our work with parallelizing compilers for imperative languages, and note that we exploit loop-parallelism with almost arbitrary procedure calls and arbitrarily complex pointer data structures. The power of our approach comes from the use of a declarative language and the semantic simplicity of Prolog when compared to Fortran or C. Indeed, the author conjectures that this simplicity is the key to performance greater than both Fortran and C: as programs become more complex, the programming idioms of Fortran (parallelism over arrays) become too restrictive, while the generality of C (arbitrary manipulation of arbitrary pointer structures) may hinder or even preclude informative analyses.

Using Prolog, our notion of data dependence is relaxed. Since Prolog is a single-assignment language, ground terms can be shared freely. Since unification is a logically pure operation, unification operations can be reordered more freely than assignments. Robust terms are the key concept here. In a sense, Reform Prolog's handling of shared data corresponds to disciplined parallel programming in imperative languages.

Prolog may not be the ideal choice of a language for the future; indeed, we have placed several restrictions on the language to make it useful as well as amenable to analysis. However, clean semantics as well as conciseness means analyses become simpler and programs smaller. Hence, more demanding analyses should be possible for a Prolog program in the same time as for analysing a Fortran or C program performing the same task.

What Fortran and C lose for analysis purposes, they regain at the compilation stage. Compilers for imperative languages generally produce superior code to even such efforts as Aquarius Prolog [74] or Parma [68], which define the state-of-the-art at the time of writing. The compilation of Prolog must be taken to the level where competitive code is produced for large Prolog programs written in a natural programming style by a knowledgeable Prolog programmer.

## 6.4 ACKNOWLEDGEMENTS

My thanks go to Johan Bevemyr and Håkan Millroth for making the work behind this thesis an interesting, challenging and satisfying task. Johan implemented the engine with daunting speed and performed the necessary hard-nosed reality checks. Did I mention he is a LaTeXwizard as well? Håkan managed to walk a fine line by keeping the project on track while at the same time encouraging innovation, which was no mean feat. He also proved to be a good editor and this work is much improved as a result.

Mattias Waldau deserves my thanks for posing a crucial question and I am grateful to Jonas Barklund and Sven-Olof Nyström for insightful comments on this work as well as computer science in general. Sten-Åke Tärnlund introduced me to the Reform Prolog project. The Computing Science Department of Uppsala University has provided an excellent working environment. The Swedish Institute of Computer Science generously lent us the use of a 26-processor Sequent Symmetry. Edinburgh University and, in particular, Malcolm Brown deserves thanks for their hospitality and the use of a Meiko Computing Surface in 1991. Rong Yang kindly answered my questions about P-Prolog. Gerda Janssens generously provided me with a prototype type graph inferencer. My thanks also to Marianne Ahrne, who proofread this thesis on short notice. NUTEK, COMPULOG and Uppsala University provided the funding for this research.

Finally, to my friends and family: this thesis would have been impossible without you, collectively and individually. Thank you.

# Bibliography

1. H. Aït-Kaci, *The WAM: A (Real) Tutorial*, MIT Press, 1991. {3, 48, 55}

2. K.A.M. Ali, R. Karlsson, The Muse or-parallel Prolog model and its performance, in *Proceedings of the North American Conference on Logic Programming*, MIT Press, 1990. {3, 90}

3. U. Banerjee, R. Eigenmann, A. Nicolau, D.A. Padua, Automatic program parallelization, Proceedings of the IEEE, vol. 81, no. 2, February, 1993 {8}

4. J. Barklund, J. Bevemyr, Executing bounded quantifications on shared memory multiprocessors, to appear at *PLILP'93* {6, 74}

5. J. Barklund, J. Bevemyr, Prolog with arrays and bounded quantifications, in *Logic Programming and Automated Reasoning*, LNCS 698, Springer Verlag, 1993 {80}

6. J. Barklund, H. Millroth, *Nova Prolog*, UPMAIL Technical Report 52, Computing Science Department, Uppsala University, 1988. {6}

7. J. Barklund, H. Millroth, Providing iteration and concurrency in logic programs through bounded quantifications, in International Conference on Fifth Generation Systems, 1992. {6}

8. J. Bevemyr, *A Recursion Parallel Prolog Engine*, Licentiate of Philosophy Thesis, Uppsala Theses in Computer Science 16/93, 1993. {11, 57, 76, 79}

9. J. Bevemyr, T. Lindgren, H. Millroth, Exploiting recursion-parallelism in Prolog, in *PARLE-93*, eds. A. Bode, M. Reeve, G. Wolf, LNCS 694, Springer Verlag, 1993. {12, 84}

10. J. Bevemyr, T. Lindgren, H. Millroth, Reform Prolog: The language and its implementation, in *Logic Programming: Proceedings of the Tenth International Conference*, MIT Press, 1993. {12, 84}

11. M. Bruynooghe, A practical framework for the abstract interpretation of logic programs, *Journal of Logic Programming* 10:91-124, 1991. {53}

12. H. Burkhardt, S. Frank, B. Knobe, J. Rothnie, *Overview of the KSR1 Computer System*, Technical report KSR-TR-9202001, Kendall Square Research, 1992 {11}

13. M. Carlsson, *Design and Implementation of an Or-Parallel Prolog Engine*, Ph.D. Thesis, SICS-RITA/02, 1990. {3}

14. K.L. Clark, F. McCabe, The control facilities of IC-Prolog, in *Expert Systems in the Micro-Electronic World* (ed. D. Michie), Edinburgh University Press, 1979. {5}

15. K.L. Clark, S. Gregory, A relational language for parallel programming, in *Proceedings ACM Symposium on Functional Programming and Computer Architecture*, 1981. {5}

16. K.L. Clark, S. Gregory, PARLOG: A parallel logic programming language, report DOC 83/5, Department of Computing, Imperial College, London, 1983. {5}

17. M. Codish, M. Falaschi, K. Marriott, Suspension analysis for concurrent logic programs, in *Logic Programming: Proceedings of the Eighth International Conference* {54}

18. M. Codish, A. Mulkers, M. Bruynooghe, M. Garcia de la Banda, M. Hermenegildo, Improving abstract interpretations by combining domains, in *Proceedings of the 1993 Symposium on Partial Evaluation and Program Manipulation*, ACM Press 1993. {53}

19. C. Codognet, P. Codognet, M.-M. Corsini, Abstract interpretation for concurrent logic languages, in *Logic Programming: Proceedings of the 1990 North American Conference*, MIT Press 1990 {54}

20. J.S. Conery, D.F. Kibler, Parallel interpretation of logic programs, in *Proceedings ACM Symposium on Functional Programming and Computer Architecture*, 1981. {4}

21. P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, 1977. {26}

22. P. Cousot, R. Cousot, Abstract interpretation and application to logic programs, Journal of Logic Programming, 1992:13:103-179. {26, 27, 37}

23. J. Crammond, The abstract machine and implementation of parallel Parlog, New Generation Computing 10 (1992), pp. 385-422, Springer Verlag 1992. {77, 79}

24. S.K. Debray, Static inference of modes and data dependencies in logic programs, ACM Transactions on Programming Languages and Systems, Vol. 11, No. 3, July 1989, pp. 418-450. {27, 28, 33, 40, 53}

25. S.K. Debray, Efficient dataflow analysis of logic programs, Journal of the ACM, Vol 39, No. 4, October 1992. {27, 30, 40, 43, 53, 54}

26. S.K. Debray, D.S. Warren, Automatic mode inference for logic programs, Journal of Logic Programming, Vol. 5, No. 3, 1988. {28, 30}

27. D. De Groot, Restricted AND-parallelism, in *Proceedings of the International Conference on Fifth Generation Systems*, North-Holland, Amsterdam, 1984. {4}

28. P. Feautrier, Dataflow analysis of array and scalar references, International Journal of Parallel Programming, Vol. 20, No. 1, February 1991, Plenum Press {7}

29. G. Filé, S. Rossi, Static analysis of Prolog with cut, in *Logic Programming and Automated Reasoning*, LNCS 698, Springer Verlag, 1993 {54}

30. G. Gupta, V. Santos Costa, R. Yang, M.V. Hermenegildo, IDIOM: Intergrating Dependent and-, Independent and- and Or-parallelism, in *Logic Programming: Proceedings of the 1991 International Symposium*, MIT Press, 1991. {4}

31. G. Gupta, M.V. Hermenegildo, ACE: And/Or-parallel copying-based execution of logic programs, in *Parallel Execution of Logic Programs*, LNCS 569, Springer Verlag, 1991. {4}

32. E. Hagersten, A. Landin, S. Haridi, DDM – A cache-only memory architecture, *IEEE Computer*, 25(9):44-54, Sept. 1992 {11}

33. W.L. Harrison III, The interprocedural analysis and parallelization of Scheme programs, *Lisp and Symbolic Computation*, Vol. 2, no. 3/4, 1989 {8}

34. B. Hausman, A. Ciepielewski, S. Haridi, Or-parallel Prolog made efficient on shared memory multiprocessors, in *1987 IEEE International Symposium on Logic Programming*, IEEE Press, 1987. {3}

35. M.V. Hermenegildo, An abstract machine for restricted AND-parallel execution of logic programs, in *Third International Conference on Logic Programming*, Springer LNCS 225, July 1986. {4, 78}

36. M.V. Hermenegildo, K.J. Greene, &-Prolog and its performance: exploiting independent and-parallelism, in *Proceedings of the Seventh International Conference on Logic Programming*, MIT Press 1990. {4}

37. M.V. Hermenegildo, F. Rossi, Non-strict independent and-parallelism, in *Proceedings of the Seventh International Conference on Logic Programming*, MIT Press 1990. {4}

38. G. Janssens, M. Bruynooghe, Deriving descriptions of possible values of program variables by means of abstract interpretation, Journal of Logic Programming 1992:13:205-258. {52, 90}

39. A. King, P. Soper, Schedule analysis of concurrent logic programs, in *Proceedings of the Joint International Symposium on Logic Programming*, MIT Press 1992. {6}

40. R.A. Kowalski, Predicate logic as a computer language, in *Information Processing 74*, pp. 569-574, North-Holland, Amsterdam, 1974. {3}

41. A.J.Kusalik, Serialization of process reduction in Concurrent Prolog, New Generation Computing 2(1984) 289-298. {77}

42. J.R. Larus, P.N. Hilfinger, Detecting conflicts between structure accesses, in *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, ACM Press, 1988 {9}

43. J.R. Larus, P.N. Hilfinger, Restructuring Lisp programs for concurrent execution, in *Proceedings of the ACM/SIGPLAN PPEALS 1988 Parallel Programming: Experience with Applications, Languages and Systems*, ACM Press 1988 {9}

44. B. Le Charlier, K. Musumbu, P. Van Hentenryck, A generic abstract interpretation algorithm and its complexity analysis (extended abstract), in *Logic Programming: Proceedings of the Eighth International Conference*, MIT Press, 1991. {53}

45. D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, M. Lam, The Stanford DASH Multiprocessor, *IEEE Computer*, 25(3):63-79, March 1992 {11}

46. J. Lloyd, *Foundations of Logic Programming* (2nd ed.), Springer Verlag, {3}

47. B.C. Massey, E. Tick, Sequentialization of parallel programs with modes, in *Logic Programming and Automated Reasoning*, LNCS 698, Springer Verlag, 1993 {6}

48. H. Millroth, *Reforming the compilation of logic programs*, Ph.D. Thesis, Uppsala Theses in Computer Science 10, 1991. {7, 17, 18}

49. H. Millroth, Reforming compilation of logic programs, in *Logic Programming: Proceedings of the Eighth International Conference*, MIT Press, 1991. {7}

50. H. Millroth, Reform compilation for non-linear recursion, *Proceedings of the International Conference on Logic Programming and Automated Reasoning*, LNCS 624, Springer Verlag, 1992. {7, 15}

51. H. Millroth, personal communication. {15}

52. K. Muthukumar, M.V. Hermenegildo, Compile-time derivation of variable dependency using abstract interpretation, Journal of Logic Programming, 1992:13:315-347. {54, 90}

53. L. Naish, Parallel NU-Prolog, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, MIT Press, 1988. {5, 20, 66}

54. C. Pancake, Multithreaded languages for scientific and technical computing, in *Proceedings of the IEEE*, Vol. 81, No. 2, February 1993. {8}

55. G.H. Pollard, Parallel execution of Horn clause programs, Ph.D. Thesis, Imperial College, London, 1981. {3}

56. V. Santos Costa, D.H.D. Warren, R. Yang, Andorra-I: A parallel Prolog system that transparently exploits both and- and or-parallelism, in *Third ACM SIGPLAN Symposium on Principles & Practices of Parallel Programming*, 1991. {4}

57. V. Santos Costa, D.H.D. Warren, R. Yang, The Andorra-I preprocessor: supporting full Prolog on the basic Andorra model, in *Logic Programming: Proceedings of the Eighth International Conference*, MIT Press 1991. {4, 53, 77, 78}

58. V. Santos Costa, D.H.D. Warren, R. Yang, The Andorra-I engine: a parallel implementation of the basic Andorra model, in *Logic Programming: Proceedings of the Eighth International Conference*, MIT Press 1991. {4, 77}

59. R. Sedgewick, *Algorithms (2nd edition)*, Addison-Wesley, 1989. {32}

60. E.Y. Shapiro, *A subset of Concurrent Prolog and its interpreter*, ICOT Technical Report TR-003, Institute for New Generation Computing Technology, Tokyo, 1983. {5}

61. K. Shen, Exploiting dependent and-parallelism in Prolog: the dynamic dependent and-parallel scheme (DDAS), in *Proceedings of the Joint International Symposium on Logic Programming*, MIT Press, 1992. {5}

62. K. Shen, *Studies of And-Or Parallelism*, Ph.D. Thesis, Cambridge University, revised June 1992. {5, 6, 78}

63. J.P. Singh, J.L. Hennessy, An empirical investigation of the effectiveness and limitations of automatic parallelization, in *Proceedings of the International Symposium on Shared Memory Multiprocessing*, Tokyo, April 1991 {8}

64. A. Singhal, Y.N. Patt, Unification parallelism: how much can we exploit?, in *Proceedings of the North American Conference on Logic Programming 1989*, MIT Press, 1989. {76}

65. H. Søndergaard, An application of abstract interpretation of logic programs: Occur check reduction, in *ESOP'86 Proceedings European Symposium on Programming*, LNCS 213, Springer Verlag, 1986. {53}

66. L. Sterling, E. Shapiro, *The Art of Prolog*, MIT Press, 1986 {3}

67. P. Tang & P.-C. Yew, Processor Self-Scheduling for Multiple Nested Parallel Loops, *Proc. 1986 Int. Conf. Parallel Processing*, August 1986. {8}

68. A. Taylor, *High Performance Prolog Implementation*, Ph.D. Thesis, Basser Department of Computer Science, Sydney University, 1991. {52, 53, 91}

69. E. Tick, *Parallel Logic Programming*, MIT Press, 1991. {6}

70. S-Å. Tärnlund, *Logic information processing*, TRITA-IBADB 1034, Department of Information Processing and Computer Science, Royal Institute of Technology and University of Stockholm, 1975. {3, 6}

71. S-Å. Tärnlund, Reform, unpublished manuscript, Computing Science Department, Uppsala University, 1991. {7}

72. K. Ueda, *Guarded Horn clauses*, ICOT Technical Report TR-103, Institute for New Generation Computing Technology, Tokyo, 1985. {5}

73. K. Ueda, M. Morita, A new implementation technique for flat GHC, in *Logic Programming: Proceedings of the Seventh International Conference*, MIT Press, 1990. {5}

74. P.L. Van Roy, *Can Logic Programming Execute as Fast as Imperative Programming?*, Ph.D. Thesis, UCB/CSD 90/600, Computer Science Division (EECS), University of California, Berkeley, 1990. {91}

75. A. Voronkov, Logic programming with bounded quantifiers, in *Logic Programming*, LNAI 592, Springer Verlag, 1992. {6}

76. D.H.D. Warren, *Implementing Prolog – compiling predicate logic*, DAI Technical Reports 39-40, Edinburgh University {48}

77. D.H.D. Warren, *An abstract Prolog instruction set*, Report 309, SRI International, Menlo Park, California, USA, 1983. {10, 55}

78. D.H.D. Warren, The SRI-model for or-parallel execution of Prolog, in *1987 IEEE International Symposium in Logic Programming*, IEEE Press 1987. {3}

79. D.H.D. Warren, The Andorra model, presented at Gigalips workshop, University of Manchester, 1988. {4}

80. M. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, 1989 {7}

81. R. Yang, *A Parallel Logic Programming Language and Its Implementation*, Ph.D. Thesis, Department of Electrical Engineering, Keio University, Yokohama, 1986. {4}

82. R. Yang, T. Beaumont, I. Dutra, V. Santos Costa, D.H.D. Warren, Performance of the compiler-based Andorra-I system, in *Logic Programming: Proceedings of the Tenth International Conference on Logic Programming*, MIT Press, 1993. {4}

UPMAIL
Computing Science Department
Uppsala University
Box 311
S-751 05 UPPSALA
Sweden

| | | |
|---|---|---|
| Phone: | Nat | 018 – 18 25 00 |
| | Int | +46 18 18 25 00 |
| Fax: | Nat | 018 – 52 12 70 |
| | Int | +46 18 52 12 70 |

ISSN 0283-359X