SICS/R-90/9002

Kernel Andorra Prolog and its Computational Model

by

Seif Haridi and Sverker Janson

SICS research report
R90002
ISSN 0283-3638

KERNEL ANDORRA PROLOG AND ITS COMPUTATION MODEL

JANUARY 9, 1990

Seif Haridi and Sverker Janson
Swedish Institute of Computer Science
Box 1263, S-164 28 KISTA, Sweden

E-mail: seif@sics.se, sverker@sics.se

ABSTRACT

The logic programming language framework Kernel Andorra Prolog is defined by a formal computation model. In Kernel Andorra Prolog, general combinations of concurrent reactive languages and nondeterministic transformational languages may be specified. The framework is based on constraints.

The languages Prolog, GHC, Parlog, and Atomic Herbrand, are all executable in the Kernel Andorra Prolog computation model. There are instances of the framework in which all of these languages are embeddable.

1 INTRODUCTION

For some time now, the main efforts of logic programming language-design and implementation have been aimed towards either optimizations of Prolog, and AND/OR parallelizations thereof, or (more or less flat) concurrent committed choice languages. Preliminary research has shown that general combinations of these language types introduce new difficulties, affecting both language design and implementation.

Nevertheless, research in these two subareas is maturing, and the time has come to tackle the problem of their combination. Several languages of this kind have appeared recently, such as Andorra-I [Yang 89], flat Andorra Prolog [Haridi 89], and Pandora [Bahgat & Gregory 89], but none of these are fully general.

In this paper, a logic programming language framework called *Kernel Andorra Prolog* is defined. Kernel Andorra Prolog is specifically designed to include the Prolog and committed choice language paradigms, allowing the specification of fully general combinations.

§1.1 DESIGN GOALS

The following are the main design goals for the Kernel Andorra Prolog framework.

Formal Definition

Kernel Andorra Prolog and its computation model should be formally defined.

The motivation for the current description is primarily to be clear-cut, not to indulge in formalism. The model presented here will be further refined in the future, e. g. by including issues like granularity, and the control principles of the model (see section 5) will be formally specified.

Subsumption

The languages Prolog, GHC, Parlog, and Atomic Herbrand, should be subsumed by Kernel Andorra Prolog. There should be a single general instance of Kernel Andorra Prolog into which the majority of programs written in these languages are easily and automatically translatable. Such a language will be called the Andorra Prolog User Language.

This means that the programming paradigms from all camps are available in a single language—the reactive concurrent paradigm as well as the transformational.

Efficient Implementation

The Andorra Prolog User Language mentioned above should lend itself to efficient implementation on both single- and multi-processor architectures. In the latter case in such a way that the major forms of parallelism are exploitable.

Preliminary investigation suggests that this goal is quite feasible [Yang 89b, Brand 89], although a first implementation is likely to be slower than for existing languages.

Explicit Control

Control should be explicit in Kernel Andorra Prolog.

This is in order to simplify meta-programming, program transformation, and dataflow analysis.

Constraints

Kernel Andorra Prolog should be based on a constraint framework.

Our description uses the concept of constraints for generality, as does [Saraswat 89]. An instance of this framework using substitutions and unification is straightforward.

§1.2 OUR DESIGN

The proposed family of languages are guarded definite clause languages, with deep guards, and three guard operators (wait, cut, and commit).

In general, the machinery of deep guards is necessary in nondeterministic languages, for selecting a single solution, or collecting all solutions for a given goal. It particular the generalization to deep guards is essential to achieve the goal of simultaneously subsuming Prolog and exploiting independent and dependent parallelism. Deep guards can also be used to encapsulate nondeterministic transformational parts of a program while maintaining a reactive indeterministic computation at an outer level.

The computation model is a non-trivial generalization of the so called Andorra Model for pure definite clauses [Warren 87, Haridi & Brand 88]. The Andorra Model exploits implicit and-parallelism in the execution of definite clauses. The generalized model features a carefully controlled nondeterminism, which is available uniformly in a computation.

The framework is parameterized with the constraint system used, the chosen set of primitive operations, and their respective blocking conditions.

§1.3 OUTLINE OF CONTENTS

The paper is organized as follows:

In section 2, a setting is given, describing the simple view of constraints used, and a model for logic programming in general.

In section 3, the Andorra Model is presented. It gives priority to deterministic computation, which is seen as less speculative. Deterministic goals may be reduced in parallel, thus extracting implicit and-parallelism.

In section 4, our language and its computation model are shown. We generalize the Andorra Model to a language with deep guard evaluation. We also add the pruning operators cut and commit.

In section 5, the control of the computation model is described.

In section 6, some primitive constraint operations are introduced.

In section 7, several user-languages are summarized as instances of the kernel language when specific constraint operations are used.

Section 8 contains a short discussion.

2 A COMPUTATION MODEL FOR LOGIC PROGRAMS

A computation model for basic logic programming is described. It is then extended to reify clause search. We also introduce our (simple) view on the rôle of constraints in our language.

A computation in logic programming is a proof-tree, obtained by input resolution on Horn clauses. The states are negative clauses, called *goal clauses*. The program to be executed consists of an initial state, the *query*, and a (finite) set of program clauses. The computation proceeds from the initial state by successive reductions, transforming one state into another. In each step, an atomic goal of a goal-clause is replaced by the body of a program clause, producing a new goal clause. When an empty goal clause is reached, the proof is completed. As a by-product, a substitution for the variables occurring in the query has been produced.

§2.1 CONSTRAINTS

For generality, and to satisfy the design goals, the concept of constraint will be used instead of substitution. An *atomic constraint* is an atomic formula with a special predicate symbol. A *constraint* is a set of atomic constraints, which has a reading as their conjunction. The computation model is parameterized with a *constraint system*, which is a set of special predicate symbols from which atomic constraints may be built, some axioms describing properties of these symbols, and a mechanism that decides the satisfiability of a constraint. A constraint is *satisfiable* if the existential closure of the constraint is derivable from the axioms. Similarly, that an implication, e. g.

$$\theta \rightarrow \exists x\, \sigma,$$

holds, where θ and σ are constraints, means that the universal closure of the correspronding formula follows from the axioms.

A constraint system may also (optionally) be supplied with a mechanism that reduces a constraint to some simplified, sometimes *normal*, form. A mechanism combining the decision of satisfiability with the reduction to normal form is often called a *constraint solver*.

For example, the equality predicate =/2 is the only predicate in the usual Herbrand constraint system, where equality is axiomatized by Clark´s equality theory ([Lloyd 84] §14), expressing the free interpretation of terms, and the unification algorithm provides the constraint solver.

We disallow program clauses that define a predicate symbol in the constraint system.

§2.2 A BASIC MODEL FOR LOGIC PROGRAMMING

The notions of states and computation are now expressed in terms of a formal computation model. As the name suggests, we would like to model the actual computation, and reify aspects that are relevant in this context.

However, the model is abstract. This is mainly in the sense that some structures are implicit, which will necessarily be explicit in a real implementation, such as scheduling information, but also in the sense that some structures will be explicit, which can be made implicit in a real implementation, such as some copies of goals.

States are formalized in terms of objects called configurations. A *computation* is a sequence of configurations obtained by successive applications of rewrite rules that define valid state transitions. A *configuration* is, as described above, a sequence of atomic formu-

lae, called *atomic goals*, and an associated satisfiable *constraint*. This object is called an *and-box*, and it is defined and introduced into a configuration as follows.

⟨*configuration*⟩ ::= ⟨*and-box*⟩

⟨*and-box*⟩ ::= **and**(⟨*sequence of atomic goals*⟩ ; ⟨*constraint*⟩)

The Greek letters θ and σ denote constraints. An and-box with an *empty sequence of goals* is often denoted simply by its constraint, and an and-box with an *empty constraint* is denoted by **and**(⟨*sequence of atomic goals*⟩). The letters A and H denote atomic goals, and B, C, and D, denote sequences of atomic goals. The concatenation operation on sequences is ",". We will also overload the use of the letters A and H to denote a sequence with a single atomic goal. It should be clear from the context which use is intended.

We could, of course, mix constraints with other goals, but we choose not to do so, because when we will describe Kernel Andorra Prolog constraint goals will denote constraint operations rather than constraints.

The single rewrite rule that is used is called the *clausewise reduction operation*,

and(C, A, D; θ) ⇒ **and**(C, B, D; θ ∪ {A = H}),

where H :- B is a program clause for which the constraint θ ∪ {A = H} is satisfiable, in which variables are renamed apart from the variables in the configuration. Observe that H :- B is the abstract syntax of a clause where B might be an empty sequence.

The *initial configuration* is an and-box containing the query, the initial sequence of atomic goals, together with an empty constraint. The configurations that have empty sequences of goals are *final*. The (satisfiable) constraint of a final configuration is called an *answer*. An answer describes a set of assignments for variables for which the initial configuration holds, in terms of the chosen constraint system. In general, for an answer to be interesting, it is presented in some kind of normal form.

§2.3 A MODEL REIFYING NONDETERMINISM

For completeness, it is necessary to explore all final configurations that can be reached by reduction operations from the initial configuration. If this is done, the union of the sets of assignments satisfying the answers contains all possible assignments for variables that could satisfy the initial goal.

Especially, it is necessary to try all (relevant) program clauses for an atomic goal. This is quite implicit in the above formulation. The computation model is nondeterministic. This clause search nondeterminism will necessarily be made explicit in a real implementation. Therefore, to fulfil our design goals, the computation model is extended to make clause search nondeterminism explicit.

This is done by grouping the alternative and-boxes by so called or-boxes.

⟨*configuration*⟩ ::= ⟨*goal*⟩

⟨*goal*⟩ ::= ⟨*and-box*⟩ | ⟨*or-box*⟩

⟨*and-box*⟩ ::= **and**(⟨*sequence of atomic goals*⟩ ; ⟨*constraint*⟩)

⟨*or-box*⟩ ::= **or**(⟨*sequence of goals*⟩)

An occurrence of an or-box is called a *(global) fork*. The symbol "**fail**" denotes an empty or-box.

In the context of logic programming a *computation rule* will select atomic goals for which all clauses will be tried. This computation rule nondeterminism will remain in the model. We will reify the clause selection nondeterminism that appears when selecting possible clauses for an atomic goal.

The corresponding rewrite rule, called definitionwise reduction, creates a global fork, in which all the possible and-boxes that would be the result of clausewise reduction operations on a selected atomic goal are contained.

We rewrite by the *definitionwise reduction operation*,

$$\textbf{and}(C, A, D; \theta) \Rightarrow \textbf{or}(..., \textbf{and}(C, B_i, D; \theta \cup \{A = H_i\}), ...),$$

where H_i :- B_i are the clauses for which the constraint $(\theta \cup \{A = H_i\})$ is satisfiable. Note that rewrite rules may be applied to subexpressions of a configuration

When no clause is applicable, an or-box with no alternative and-boxes, the empty or-box or **fail**, is produced. The atomic goal is then said to *fail*.

Definitionwise reduction is sufficient to describe SLD-resolution. In section 4, a model that is extended with guard evaluation is introduced. First, the Andorra Model will be introduced, as its control principles are the main influence for the control of our extended model.

3 THE ANDORRA MODEL

The Andorra model gives priority to deterministic computation over nondeterministic computation, as nondeterministic steps could possibly (unnecessarily) multiply work.

The Andorra model divides a computation within and-boxes into *deterministic* and *nondeterministic phases*. First, all atomic goals for which it is known that at most one clause would succeed are reduced using a single clause (clausewise) during the so called deterministic phase. (These goals can be reduced in and-parallel.) Then, when no such goal is left, some goal is chosen for which all clauses are tried (definitionwise); this is called the nondeterministic phase. The computation then proceeds with a deterministic phase on each or-branch.

The key concept here is the notion of determinacy. An atomic goal is said to be *determinate* when there is at most one candidate clause that would succeed for the goal. As soon as it is known that an atomic goal has become determinate, the goal can either be reduced by a single clause, or fail, if it was known that no clause would apply. It is not considered to be an error if the mechanism for detecting the determinacy of goals fails to detect that a goal is determinate. In general, nothing less than complete execution will establish this property.

The Andorra model has a number of interesting consequences.

Firstly, the Andorra model allows determinate goals to be run in and-parallel, extracting implicit and-parallelism from the program.

Secondly, the notion of determinacy in the Andorra model gives a reasonably strong form of *synchronization*. As long as a goal is able to produce data deterministically, no consumer of this data is allowed to run ahead (if it does not know what to consume). This allows specification of concurrent processes.

Thirdly, the Andorra model reduces the search space by executing the determinate goals first. Goals can fail early, and the constraints produced by a reduction can reduce the number of alternatives for other goals. This has proved to be very relevant for the cod-

ing of constraint satisfaction problems [Kornfeld 89, Saraswat 89b, Bahgat& Gregory 89, Haridi 89, Yang 89a].

The Andorra model in items:

- An atomic goal fails if it is known that no clause would succeed for the goal.
- An atomic goal can be reduced using a single clause when it is known that all other clauses would necessarily fail for the goal.
- When no goal is known to be determinate, all clauses in its definition are tried for some goal.

Normally, a clause is known to fail for a goal if simple *primitive goals* occurring in the clause, like head unification, `=/2`, `</2`, `atomic/1`, and the like, are known to fail in the given context of the goal.

Example. Consider the well-known quick-sort program

```
qsort([],R,R).
qsort([X|L],R0,R) :-
     partition(L,X,L1,L2),
     qsort(L1,R0,[X|R1]),
     qsort(L2,R1,R).

partition([],C,[],[]).
partition([X|L],C,[X|L1],L2) :-
     X < C, partition(L,C,L1,L2).
partition([X|L],C,L1,[X|L2]) :-
     X >= C, partition(L,C,L1,L2).
```

Execution of a goal `qsort([2,3,1],L,[])` will be completely deterministic, and the Andorra model will extract parallelism as follows. The goals have some arguments suppressed for the sake of brevity, and the goals that are determinate are underlined. Determinate goals are reduced in one step.

```
?- qs([2,3,1]).
?- p([3,1]),   qs(L1),                qs(L2).
?- p([1]),     qs(L1),                qs([3|L2']).
?- p([]),      qs([1|L1']),           p([]),    qs(L1'''), qs(L2''').
?-             p([]), qs(L1''), qs(L2''),       qs([]),    qs([]).
?-                    qs([]),  qs([]).
?- □
```

The model extracts quite a lot of potential parallelism.

4 THE KERNEL ANDORRA PROLOG COMPUTATION MODEL

We now define a computation model that takes advantage of the principles underlying the Andorra model to control nondeterminism in a "deep" concurrent language.

First, the language and the configurations are defined. Then, the rewrite operations, which start "guard execution", perform commit, etc, are described. The Andorra Model principles will be clearly visible in section 5, where the pieces are put together.

§4.1 THE KERNEL ANDORRA PROLOG LANGUAGE

The kernel language clauses are basically definite clauses augmented with guard operators. Each clause contains exactly one guard operator.

⟨*guarded clause*⟩ ::= ⟨*head*⟩ **:-** ⟨*guard*⟩ ⟨*guard operator*⟩ ⟨*body*⟩

⟨*guard*⟩, ⟨*body*⟩ ::= ⟨*sequence of atomic goals*⟩

⟨*guard operator*⟩ ::= **':'** | **'!'** | **'|'**

Atomic goals are defined as usual. A *head* is an atomic goal with distinct variables as arguments. The guard operator ":" is called *wait*, "!" is called *cut*, and "|" is called *commit*. Cut and commit are called *pruning* operators.

For the generality of the argument, the following semantic description does not depend on how "head unification" is performed. Therefore, guarded clauses are assumed to be *normalized* in the sense that the head of a clause will have the form $p(v_1,\ldots,v_n)$ where v_i are distinct variables called *formal parameters,* and the terms that used to be in the head are moved into the guard where unification is performed by primitive constraint-operations. The primitive constraint operations are described in section 6.

A guarded clause defines predicate p/n if the head has the form $p(v_1,\ldots,v_n)$. A *definition* consists of a sequence of guarded clauses defining the *same* predicate, which all have the same guard operator.

§4.2 THE KERNEL ANDORRA PROLOG COMPUTATION MODEL

The computation model of Kernel Andorra Prolog allows arbitrarily deep guard evaluation. Guard evaluation is the evaluation of the guard part of the clauses which define the calling goal.

A box called *choice-box* is introduced, which holds a sequence of *guarded goals* being evaluated. As choice-boxes are now legal members of and-boxes, they are grouped with atomic goals, forming a new kind of goal called *local goal*. Since the sequence of guarded goals within a choice-box have the same guard operator, a choice-box will be qualified by the name of the guard, e.g. a commit choice-box.

⟨*configuration*⟩ ::= ⟨*goal*⟩

⟨*goal*⟩ ::= ⟨*and-box*⟩ | ⟨*or-box*⟩

⟨*local goal*⟩ ::= ⟨*atomic goal*⟩ | ⟨*choice-box*⟩

⟨*and-box*⟩ ::= **and**(⟨*sequence of local goals* ⟩ **;** ⟨*constraint*⟩)

⟨*or-box*⟩ ::= **or**(⟨*sequence of goals*⟩)

⟨*choice-box*⟩ ::= **choice**(⟨*sequence of guarded goals*⟩)

⟨*guarded goal*⟩ ::= ⟨*goal*⟩ ⟨*guard operator*⟩ ⟨*sequence of atomic goals* ⟩

In a guarded goal, the goals preceding the guard operator are called the *guard*, and the goals following it are called the *body*. Guarded goals are sometimes called *clauses*. The empty choice-box and the empty or-box are identified. The symbol "**fail**" denotes this object. An and-box with an empty sequence of local goals and a constraint σ is denoted by σ. An occurrence of an or-box is still called a *global fork*, whereas an occurrence of a choice-box is called a *local fork*. Each goal in a configuration has an *environment*. The environment of a goal is defined as the union of the constraints of all the and-boxes in which the goal occurs.

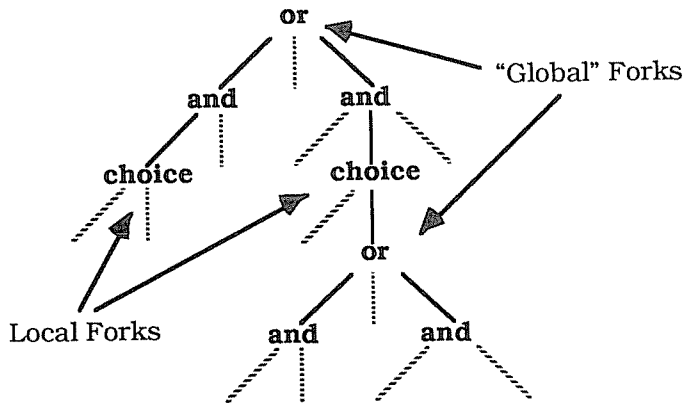


Figure 1. Local and global forks in a configuration.

In the following, the Greek letter β denotes a single goal. The letters B, C, and D denote sequences of local goals, P, Q and R denote sequences of goals, and S and T denote sequences of guarded goals. The symbol '%' denotes a guard operator.

Andorra computation is modelled by the following rewrite rules. The basic rewrite rules are those performing local forking and promotion. Local forking on an atomic goal is responsible for initiating local computations, while promotion communicates the results of a local computation to the siblings of the goal that started the local computation.

In the treatment that follows, there is a need to know whether a variable is local to an and-box or external to the box. This knowledge is necessary in order to understand the behaviour of the various constraint operations as well as some of the rewrite rules. The reason is that Andorra computation will be quite lazy on guessing (nonderministically) the value of external variables, and some primitive constraint operations will block if they try to impose constraints on external variables. Therefore and-boxes in our rewrite rules will be indexed by a set of variables which are the variables local to the box. We will sometimes omit the indexing of and-boxes when it is irrelevant.

§4.3 LOCAL FORKING

Local forking is the creation of a choice-box. An atomic goal in an and-box is replaced by a choice-box which contains the clauses defining the goal. The guards of these clauses are then available for further evaluation.

Local Forking Rule

Compared to section 2, head unification is reduced to simple parameter passing. Therefore we avoid the use of an equality constraint, and perform the substitution immediately. An atomic goal A, in the environment θ, may be rewritten by

$$\textbf{and}(C, A, D\ ;\ \sigma) \Rightarrow \textbf{and}(C, \textbf{choice}(\textbf{and}(G_1\pi_1)_{v1}\ \%\ B_1\pi_1, \ldots, \textbf{and}(G_n\pi_n)_{vn}\ \%\ B_n\pi_n), D\ ;\ \sigma),$$

using the sequence of clauses H_i :- G_i % B_i defining A (in order), in which variables are renamed apart from other variables in the configuration, and where

$$\pi_i = [\alpha_{i1}, \ldots, \alpha_{ik}/t_1, \ldots, t_k],\ A = p(t_1, \ldots, t_k),\ H_i = p(\alpha_{i1}, \ldots, \alpha_{ik}),$$

and v_i is the set of variables (other than head parameters) appearing in the guard or body of the corresponding clause.

§4.4 DETERMINISTIC PROMOTION

In a deep computation, inconsistent environments must be detected, failures must be propagated, and or-branches within a choice-box should be flattened. This is done by the following rules for environment synchronization, failure propagation, and or within a choice reduction.

Environment Synchronization Rule

and(B ; σ) ⇒ **fail**,

if the union of σ with the environment of the and-box is unsatisfiable.

Failure Propagation Rules

and(B, **fail**, C; θ) ⇒ **fail**

choice(S, (**fail** % B), T) ⇒ **choice**(S, T)

Or within a Choice Reduction Rule

choice(S, **or**(β, P) % B, T) ⇒ **choice**(S, β % B, **or**(P) % B, T)

The first promotion rule is for deterministic promotion which extracts the single guarded goal left in a choice-box, after completion of the guard execution. It promotes the produced constraints and moves the body of the guarded goal to the surrounding and-box.

Determinate Promotion Rule

and(C, **choice**(θ_V % B), D ; σ)$_W$ ⇒ **and**(C, B, D ; θ∪σ)$_{V\cup W}$,

if θ∪σ is satisfiable.

§4.5 QUIET INDETERMINISTIC PROMOTION

If the guard of a pruning choice-box is successful, its body may replace the choice-box by so called *indeterministic promotion.*

A successful guard execution is *quiet* if it reduces to a constraint σ_V which does not restrict external variables, i. e.

$\theta \rightarrow \exists_V(\sigma_V)$,

where θ is the environment of the choice-box. Here, and in the following, we use $\exists_V$ to denote existential quantification over the variables in v.

The key property of a quiet indeterministic promotion rule is that it is insensitive to any computation performed outside the pruning choice box.

Quiet Cut Rule

The cut operator "!" prunes branches to the right after a successful guard execution of the branch.

choice(P, σ_V ! B, Q) ⇒ **choice**(P, σ_V ! B),

if $\theta \rightarrow \exists_V(\sigma)$, where θ is environment of the choice-box.

Quiet Commit Rule

The commit operator "|" prunes all other branches after a successful guard execution of the branch.

choice(P, σ_V | B, Q) $\Rightarrow$ **choice**(σ_V | B),

if $\theta \rightarrow \exists_V(\sigma)$, where θ is environment of the choice-box.

The actual promotion is subsequently performed by the determinate promotion rule.

The above rules are called quiet pruning rules, since they prune quietly without adding constraints on external variables. The most obvious application is for safe clauses, which are known to bind only local variables. As will be seen quiet pruning can be ensured by the use of proper constraint operations.

The remaining promotion rules are sensitive to computation occurring outside the box to which they are applied and will be formulated later in the section dealing with controlling the computation model.

§4.6 NONDETERMINISTIC PROMOTION

A local fork is converted into a global fork by so called *nondeterministic promotion*. The sibling goals of a wait choice box (local fork) are distributed into a choice-box, and the choice-box is changed into an or-box.

Partial nondeterministic promotion extracts some of the successful guard-branches (i.e. not necessarily all branches) from a wait choice-box, and lets execution proceed into their bodies.

It sometimes matters whether branches are ordered or unordered. The semantics of cut requires that the branches in a global fork are ordered, otherwise this is unnecessary. We define the following concepts to keep track of these needs.

If the closest surrounding pruned choice is a cut choice, goals are in an *ordered context*. Otherwise, they are in an *unordered context*.

Partial Nondeterministic Promotion Rules

In an unordered context, we may rewrite an and-box by

and(C, **choice**(P, σ_V : B, Q), D ; $\theta)_W \Rightarrow$ *(unordered promotion)*
  or(**and**(C, B, D ; $\theta \cup \sigma)_{V \cup W}$,
    and(C, **choice**(P, Q), D ; $\theta)_W$),

if $\theta \cup \sigma$ is satisfiable. This rule selects any successful branch.

In an ordered context, we may rewrite an and-box by

and(C, **choice**(σ_V : B, Q), D ; $\theta)_W \Rightarrow$ *(ordered promotion)*
  or(**and**(C, B, D ; $\theta \cup \sigma)_{V \cup W}$,
    and(C, **choice**(Q), D ; $\theta)_W$),

if $\theta \cup \sigma$ is satisfiable. This rule selects the leftmost solution only. Note that this does not introduce any incompleteness as the cut operation needs a successful leftmost branch.

The *nondeterministic promotion rules* are the transitive closures of these rules. They make any number of successful branches extractable at once.

§4.7 NOISY INDETERMINISTIC PROMOTION

A guard executes *noisily* if it reduces to a constraint that restricts external variables. In general, noisy pruning requires checking the satisfiability of the combined constraints

Noisy Promotion Rule for Cut

The cut operator "!" prunes all branches to the right after a successful execution of a guard branch.

$$\textbf{and}(C, \textbf{choice}(\sigma_V\ !\ B, Q), D; \theta)_W \Rightarrow \textbf{and}(C, B, D\ ;\ \theta\cup\sigma)_{V\cup W},\quad \textit{(noisy cut)}$$

if $\theta\cup\sigma$ is satisfiable.

Noisy Promotion Rule for Commit

The commit operator "|" prunes all other clauses after a successful execution of a guard branch.

$$\textbf{and}(C, \textbf{choice}(P, \sigma_V\ |\ B, Q), D\ ;\ \theta)_W \Rightarrow \textbf{and}(C, B, D\ ;\ \theta\cup\sigma)_{V\cup W},\quad \textit{(noisy commit)}$$

if $\theta\cup\sigma$ is satisfiable.

5 Control of the Computation Model -- Stability

The model presented so far does not give any preference to one rewrite rule over another when both are applicable to a subexpression of a configuration. The Andorra computation model will favour deterministic reduction over nondeterministic and will defer guessing the value of variables as long as possible. To define the control of the model we need two notions, the notion of a guess-free rewrite rule and that of a stable box.

A rule is guess-free if it will not guess the value of an external variable, either by an indeterministic or a nondeterministic step. Guessing the value of an external variable by an indeterministic step may lead to failure and guessing by a nondeterministic step may lead to redundant speculative computations. We call all the rewrite rules presented above except: (i) nondeterministic promotion, and (ii) noisy deterministic promotion, guess-free rules. We include as guess-free the rules of primitive constraint operations to be described in the next section.

This gives us the **first control principle**:

- *Guess-free rules are applied whenever possible to any box in a configuration.*

Given a box within which all guess-free rules have been applied and its computation has ceased, but some rules that are not guess-free are still applicable to or within the box, we would like to know when we are going to apply the guessing rule. Intuitively these rules should be applied when no further rewrite operations anywhere external to the box can enable us to resume guess-free operations within the box. We refer to this state by saying that the box is *locally stable*. Unfortunately, local stability, as stated, is undecidable, since deciding local stability involves deciding the halting problem. Therefore, we make a conservative decidable definition of local stability but still leave room for better approximations.

A box is *locally stable* if all guess-free operations have been applied on or within the box and no constraints within the box are imposed on variables external to the box.

This gives us the **second control principle**:

- *Nondeterministic promotion and noisy inderministic promotions are applied to or within an locally stable box.*

Observe that if all guess-free operations terminate we will still have in the worst case one stable box since an outermost and-box will always be stable.

Another observation is that nondeterministic promotion is an expensive operation so one should perform global forking with the smallest scope as there is always the possibility of having nested boxes to all of which nondeterministic promotion is applicable.

This gives us the **third control principle**:

- *An innermost nondeterministic promotion within a stable box should be chosen first.*

This captures the essence of the Andorra Model in that all deterministic computation will be performed first. However, the addition of noisy pruning, and the possibility of nested guards, introduce new difficulties. Several rewrites may be applicable at the same time.

Since programmers should be able to specify a predictable behaviour for a program, we have defined a standard order in which the following operations are to be applied. As there is yet no programming experience from languages of this nondeterministic concurrent type, our first attempt is necessarily tentative, and could be changed in the future. We have essentially only our intuition from Prolog to build upon.

Conflicting rewrites are resolved as follows:

- If a noisy indeterministic promotion and a nondeterministic promotion are both applicable in an and-box, the leftmost is chosen.
- If several local forks in the same and-box are available for nondeterministic promotion, the leftmost is chosen.

There are cases not covered by this list that probably should have a standard order. We have chosen to leave them undefined until reasons for one choice or the other arises.

When several indeterministic promotions are available in the same and-box we deliberately allow arbitrary order. We do not seek to emulate Prolog behaviour in the general framework. This can be programmed, or further specified as a specific instance of the general framework. The arbitrary order left unspecified is necessary for multiple writers style programming.

6 THE PRIMITIVE OPERATIONS

The framework presented above is parameterized by the kind of primitive constraints and constraint operations that are allowed. Different constraint operations will in general have different kinds of blocking behaviour. We present a few operations, and the conditions under which they may be reduced. A specific instance of Kernel Andorra Prolog will have just a few of these operations combined in a disciplined way. In fact a random combination of these operation will lead to unpredictable behaviour since for instance some combinations are not commutative. Section 7 discusses possible user languages.

The only primitive constraint presented here is equality. The exposition is too general to allow for other primitive constraints. We show four operations on equality constraints, which have different blocking behaviour.

To each of our primitive operations corresponds a condition on the constraint produced by a primitive. The conditions express, using entailment, how constrained (or instantiated), the arguments are required to be before execution. These conditions are general enough to be applied also to other primitive constraints than equality.

Blocking Conditions

In the following description, the box α is the closest and-box in which the primitive is contained, β is the closest choice box containing α (if any), θ_α is the environment of α, θ_β is the environment of β, θ is the environment of the primitive, σ a "solution" constraint for the primitive, σ_α the constraint of α, and σ_β the constraint of β. The set of local variables of α is v_α and for β is v_β.

The notation $op(\sigma)$ denotes a constraint operation *op* applied to the primitive constraint σ. The conditions for primitives of a given class to reduce successfully are the following:

$\text{Tell}_\omega(\sigma)$ No condition.

$\text{Tell}_1(\sigma)$ σ imposes no new conditions on variables external to α and β, i.e.

$(\theta_\beta \cup \sigma_\alpha \cup \sigma_\beta) \rightarrow \exists_{v\alpha \cup v\beta}(\sigma \cup \sigma_\alpha \cup \sigma_\beta)$.

$\text{Tell}_0(\sigma)$ σ imposes no new conditions on variables external to α, i.e.

$(\theta_\alpha \cup \sigma_\alpha) \rightarrow \exists_{v\alpha}(\sigma \cup \sigma_\alpha)$.

$\text{Ask}(\sigma)$ The environment of the primitive entails σ, i. e. $\theta \rightarrow \sigma$.

A more operational reading follows.

Tell$_\omega$ A tell_ω-primitive may always go ahead.

Tell$_1$ A tell_1-primitive must block until it will not further restrict variables that are external to the "parent" and-box.

Tell$_0$ A tell_0-primitive must block until it will not further restrict variables that are external to the "current" and-box.

Ask An ask-primitive may never restrict variables.

These conditions are chosen so as to be able to emulate the behaviour of the languages Prolog, GHC, Atomic Herbrand, and Parlog. An arbitrary mix of constraint operations with various conditions might lead to non-commutative suspension behaviour, and is considered meaningless.

Another interesting variant of a tell-unification operation which we will used in the next section is:

$\text{tell}_{\text{cut/commit}}$ $\text{tell}_{\text{cut/commit}}$-unification can only bind variables that are local to the closest surrounding pruning choice-box or any wait choice-box within the pruning one.

This operation has a dynamic blocking behaviour.

Unification

The four corresponding kinds of unification operations are:

Tell$_\omega$ Tell_ω-unification is unrestricted unification, as in Prolog.

Tell$_1$ Tell_1-unification can bind local variables and variables in the closest surrounding environment. It is analogous to the "tell" operation of Atomic Herbrand.

Tell$_0$ Tell_0-unification can bind local variables only. It is the unification primitive of GHC.

Ask Ask-unification cannot bind variables. It is used by Parlog and ask in Atomic Herbrand.

Thus, for the above operations, an equality goal of a given kind may be rewritten by the following rules. The symbol θ denotes the environment of the constraint operation.

and(C, *op*($t_1 = t_2$), D ; σ)$_V$ ⇒ **and**(C, D; $\{t_1 = t_2\} \cup \sigma)_V$,

if the condition of *op*($t_1 = t_2$) is true, and $\{t_1 = t_2\} \cup \theta$ is satisfiable.

and(C, *op*($t_1 = t_2$), D ; σ) ⇒ **fail**,

if $\{t_1 = t_2\} \cup \theta$ is unsatisfiable. Otherwise, the equality goal will block.

7 POSSIBLE INSTANTIATIONS OF THE FRAMEWORK.

The following are possible user languages sharing the Kernel Andorra framework with restricted use of primitive operations. The following list is by no means complete and the presentation is very dense. A detailed treatment will be presented in another paper, where basic Prolog (Prolog without assert and retract) is treated. All languages presented may have a user-oriented syntax where the constraint operations are implicit. The constraint domain used is Herbrand.

The quiet directional Andorra Prolog

In the quiet directional user language, the only constraint operation used is $tell_0$. It is used both in the guard and the body. This is a nondeterministic language that subsumes GHC. The nondeterministic procedures using wait-guards are used in a specific input/output mode since the guard is quiet. The language can be implemented very efficiently because no satisfiability test is needed when constraints are combined by promotion rules, and the detection of stable boxes is simplified. Blocking on external variables is easy, because there is a single place where an external variable can become instantiated.

The pure Andorra Prolog

The basic Andorra model for definite clauses is achieved by the use of $tell_1$ operations. A definite clause is translated into a wait-guarded clause where the head unification and all primitive goals of the definite clause are extracted and their $tell_1$ version is formed and inserted as the flat guard of the corresponding guarded clause.

The quiet Andorra Prolog

A very useful language is achieved by a combination of the above two languages. Pure clauses translated as above combined with the quiet guarded language (the quiet directional Andorra Prolog) gives a nondirectional language and still preserves the property that all guards are quiet. Observe that this property holds even if pure procedures are called from within the deep guards.

The reactive quiet Andorra Prolog

The subset of the above language where goals in the body of commit or cut clauses are strictly calls to cut or commit procedures, and there are no restrictions on the type of procedures a goal in a guard may call is a reactive language. In this language nondeterministic computations are always encapsulated within guards, and no global nondeterminism is introduced.

The disciplined-noisy (or atomic) Andorra Prolog

For this language we introduce a sequencing operator '&' between goals. This includes 'quiet Andorra Prolog' as a subset. The language consists of guarded clauses where a guard now has the following form: G & t, where G is a quiet guard as before and t is a sequence of tell_1 operations on primitive constraints. Again, the combination preserves the property the the G-part of the guard is quiet. The language allows noisy pruning in a shallow way and still can be implemented reasonably efficiently. This language subsumes Atomic Herbrand and allows the full use of the Andorra model.

The core user language Andorra Prolog

The language currently called the Core User Language within the PEPMA Project allows wait-guards to be noisy and pruning guards to be quiet. It is obtained by using $\text{tell}_{\text{cut/commit}}$ operations uniformly in all guarded clauses.

8 DISCUSSION

We have presented a language framework that, at least in principle, will subsume the major families of logic programming languages. It is nevertheless reasonably compact and homogeneous.

A simple implementation of some fully general instance of this framework will not achieve the speed of the subsumed languages. It is our belief that these speeds (and better) will be reached by advanced compilation techniques, based on dataflow analysis, and recognition of cases corresponding for example to Prolog and FGHC. This remains to be shown.

Acknowledgements

The authors wish to thank David H. D. Warren, Vijay Saraswat, Bill Kornfeld, and Torkel Franzén, for many valuable comments and suggestions. This work is supported by the PEPMA ESPRIT Project (P2471).

REFERENCES

[Bahgat & Gregory 89] Reem Bahgat and Steve Gregory, *Pandora*, in: Proceedings of the ICLP-89, Portugal 1989.

[Brand 89] Per Brand, *Andorra Implementation Proposal*, Internal Report, SICS 1989.

[Haridi 89] Seif Haridi, *A Logic Programming Language Based on the Andorra Model*, New Generation Computing, forthcoming.

[Haridi & Brand 88] *Andorra Prolog, an Integration of Prolog and Committed Choice Languages*, in: Proceedings of the FGCS 1988, Japan.

[Kornfeld 89] William Kornfeld, *Constraint Programming in Andorra Prolog*, Presented at the Swedish-Japanese-Italian workshop, 1989.

[Lloyd 84] J. W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1989.

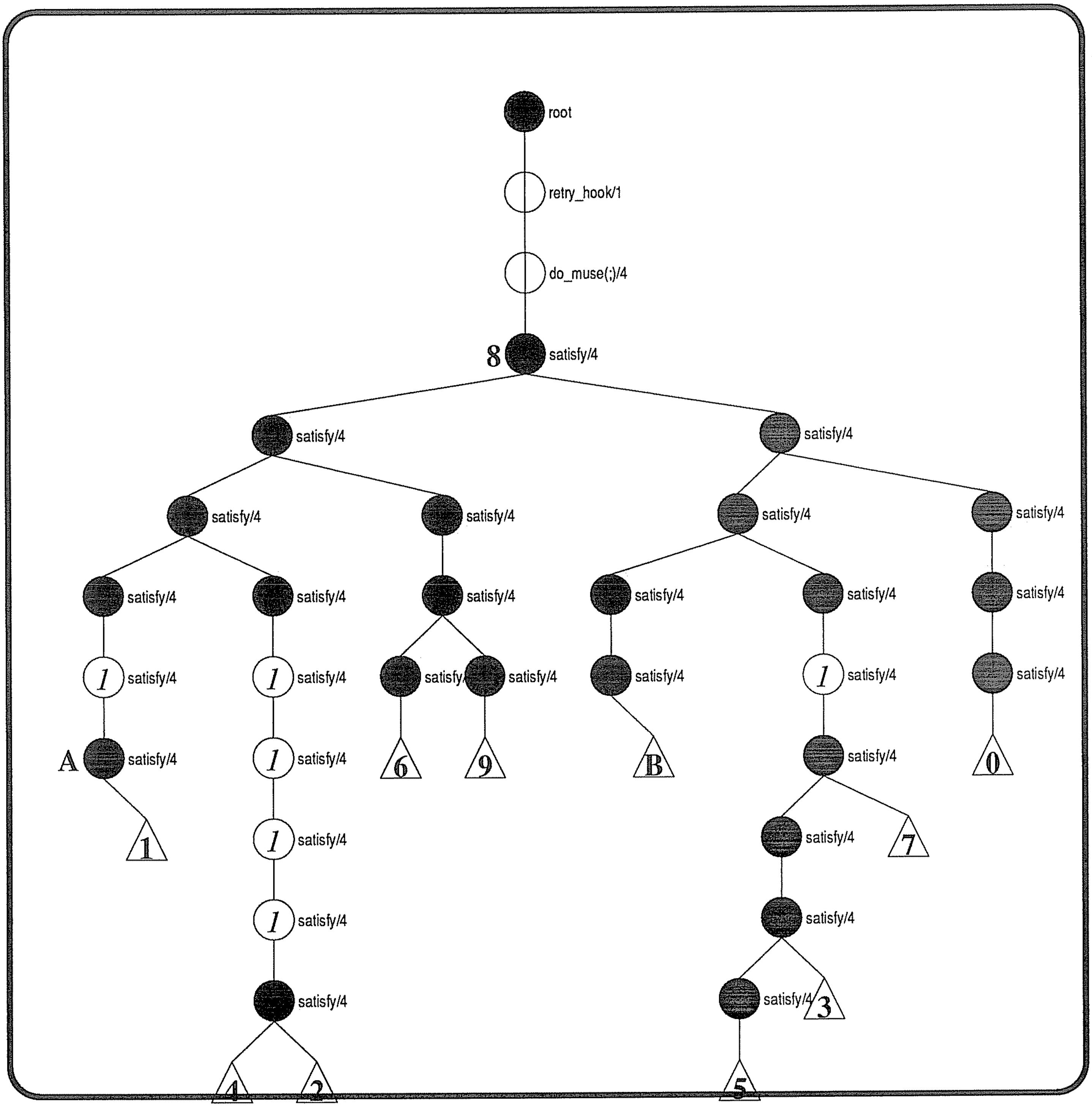
[Saraswat 89a] Vijay A. Saraswat, *Concurrent Constraint Programming Languages*, PhD thesis, Carnegie-Mellon University, January 1989.

[Saraswat 89b] Vijay A. Saraswat, *Programming in Andorra Prolog*, Xerox PARC 1989.

[Warren 87] David H. D. Warren, *The Andorra Principle*, Presented at Gigalips workshop, Stockholm, 1987.

[Yang 89a] Rong Yang, *Solving Simple Substitution Ciphers in Andorra-I*, in: Proceedings of the ICLP-89, Portugal, 1989.

[Yang 89b] Rong Yang, *Implementation Notes on Andorra-I*, Internal Report, University of Bristol.



c3_12_28685_001.mt