Kernel Andorra Prolog and its Computation Model

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 optimisations of Prolog, and AND/OR parallelisations 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 basic Andorra [12, 11], flat Andorra Prolog [3], and Pandora [1], 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 instance of Kernel Andorra

Prolog into which the majority of programs written in these languages are easily and automatically translatable. One such 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 [11, 5], although a first implementation is likely to be somewhat slower than state-of-the-art implementations of existing languages.

Explicit Control

Control should be explicit in Kernel Andorra Prolog.

Explicit control will 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 [8]. An instance of this framework using substitutions and unification is straightforward.

1.2 Our Design

The languages in the proposed family 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. In particular the generalisation 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 generalisation of the Andorra Model for pure definite clauses [10, 4]. The Andorra Model exploits implicit and-parallelism in the execution of definite clauses. The generalised model features a carefully controlled nondeterminism, which is available uniformly in a computation.

The framework has as parameters the constraint system used, and the chosen set of constraint operations with their respective activation conditions. Also, in some specific cases, sequential ordering between goals is necessary to achieve the desired synchronisation effects.

1.3 Outline of Contents

The paper is organised 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 basic 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. Our language adds deep guard evaluation and the pruning operators cut and commit to the basic model.

In section 5, the control of the computation model is described. It is a generalisation of the Andorra Model.

In section 6, some constraint operations are introduced.

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

Section 8 contains a short discussion.

2   A Computation Model for Logic Programs

In this paper, we will develop a computation model for logic programs that allows a high degree of parallelisation. This is done in a stepwise manner. First, 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. A *constraint* is a formula in some constraint language. The constraint formulas are closed under conjunction. The computation model presupposes a *constraint system*, which is a set of special predicate symbols from which constraints may be built, a constraint theory $(\mathcal{TC})$ describing properties of these predicates, and mechanisms that decide the following two properties—satisfiability and restriction ("entailment"). A constraint σ is *satisfiable* if $\mathcal{TC} \models \exists(\sigma)$. The constraint σ does not *restrict* θ (*outside V*) if $\mathcal{TC} \models \theta \rightarrow \exists V(\sigma)$. (Here, and in the following, we use $\exists V$ to denote existential quantification over the variables in V.) The *true* constraint is a variable-free constraint formula **true** which is true in the constraint theory, i. e. $\mathcal{TC} \models \textbf{true}$.

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 axiomatised by Clark's equality theory, 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 expressed in terms of a formal computation model. 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 formalised in terms of objects called configurations. A *computation* is a (possibly infinite) 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 formulæ, 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.

$$\langle configuration \rangle ::= \langle and\text{-}box \rangle$$
$$\langle and\text{-}box \rangle ::= \textbf{and}(\langle sequence\ of\ atomic\ goals \rangle\ ;\ \langle constraint \rangle)$$

The Greek letters θ and σ denote constraints. An and-box of the form $\textbf{and}(;\ \sigma)$ will normally be written as σ, and an and-box with the true constraint is normally written as $\textbf{and}(\langle sequence\ of\ atomic\ goals \rangle)$. 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 the *clausewise reduction operation*,

$$\textbf{and}(C, A, D\ ;\ \theta) \Rightarrow \textbf{and}(C, B, D\ ;\ \theta \wedge (A = H)),$$

where H :- B is a program clause for which the constraint $\theta \wedge (A = H)$ is satisfiable, in which variables are renamed apart from the variables in the configuration. Observe that B might be an empty sequence.

The *initial configuration* is an and-box containing the query, the initial sequence of atomic goals, together with the true constraint. The configurations that have empty sequences of goals are *final*. The 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 or-boxes.

$$\langle configuration \rangle ::= \langle goal \rangle$$
$$\langle goal \rangle ::= \langle and\text{-}box \rangle \mid \langle or\text{-}box \rangle$$
$$\langle and\text{-}box \rangle ::= \mathbf{and}(\langle sequence\ of\ atomic\ goals \rangle \ ; \langle constraint \rangle)$$
$$\langle or\text{-}box \rangle ::= \mathbf{or}(\langle sequence\ of\ goals \rangle)$$

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*,

$$\mathbf{and}(C, A, D \ ; \theta) \Rightarrow \mathbf{or}(\ldots, \mathbf{and}(C, B_i, D \ ; \theta \wedge (A = H_i)), \ldots),$$

where $H_i :\text{-} B_i$ are the clauses for which the constraint $\theta \wedge (A = H_i)$ is satisfiable, in which variables are renamed apart from the variables in the 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 A 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. In the next section, 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 are likely to 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 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 *deterministic* 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 deterministic, 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 deterministic. 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 deterministic 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 *synchronisation*. 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 deterministic 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 coding of constraint satisfaction problems [6, 9, 1, 4, 12].

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 deterministic, 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. The goals with a nonvariable argument are deterministic. All deterministic 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 Extended Computation Model

We now define a computation model that will take 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 that start guard execution, perform commit, etc, are described. The control of the computation model is defined in section 5, where its relation to the Andorra Model will be clearly visible.

4.1   The Kernel Andorra Prolog Language

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

$$\langle guarded\ clause\rangle ::= \langle head\rangle :- \langle guard\rangle \langle guard\ operator\rangle \langle body\rangle$$

$$\langle head\rangle ::= \langle variable\text{-}pure\ atomic\ goal\rangle$$

$$\langle guard\rangle, \langle body\rangle ::= \langle sequence\ of\ atomic\ goals\rangle$$

$$\langle atomic\ goal\rangle ::= \langle variable\text{-}pure\ atomic\ goal\rangle \mid \langle constraint\ goal\rangle$$

$$\langle guard\ operator\rangle ::= \text{':'} \mid \text{'!'} \mid \text{'|'}$$

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 or even on the appearance of terms. Therefore, guarded clauses are assumed to be normalised in the sense that all user-defined atomic goals will have the variable-pure form $p(v_1, \ldots, v_n)$. In the head, v_i are different variables called *formal parameters*. In the body, v_i may be repeated, and are there called *actual parameters*. The variables in a clause that do not occur among the formal parameters are called *local variables*.

Unification and the like are performed by primitive constraint operations. A constraint operation may block its constraint until its *activation condition* is satisfied. Some constraint operations are described in section 6.

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

4.2   The Kernel Andorra Prolog Computation Model

The computation model of Kernel Andorra Prolog allows arbitrarily deep guard evaluation.

In the treatment that follows, there is a need to know whether a variable is local or external to an and-box. This knowledge is necessary in order to specify 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 (nondeterministically) the value of external variables, and some constraint operations will block if they try to impose constraints on external variables. Therefore and-boxes 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.

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

$$\langle configuration\rangle ::= \langle goal\rangle$$
$$\langle goal\rangle ::= \langle and\text{-}box\rangle \mid \langle or\text{-}box\rangle$$
$$\langle local\ goal\rangle ::= \langle atomic\ goal\rangle \mid \langle choice\text{-}box\rangle$$
$$\langle and\text{-}box\rangle ::= \mathbf{and}(\langle seq.\ of\ local\ goals\rangle\ ;\ \langle constraint\rangle)_{\langle set\ of\ variables\rangle}$$
$$\langle or\text{-}box\rangle ::= \mathbf{or}(\langle sequence\ of\ goals\rangle)$$
$$\langle choice\text{-}box\rangle ::= \mathbf{choice}(\langle sequence\ of\ guarded\ goals\rangle)$$
$$\langle guarded\ goal\rangle ::= \langle goal\rangle\langle guard\ operator\rangle\langle sequence\ of\ atomic\ goals\rangle$$

In a guarded goal, the goal preceding the guard operator is called the *guard*, and the sequence of atomic goals following it are called the *body*. The symbol "**fail**" will be used to denote the empty or-box and the empty choice-box regarded as collapsing to the same object. An occurrence of an or-box is still called a *global fork*, whereas an occurrence of a choice-box is called a *local fork* . Finally, the following syntactic category is convenient.

$$\langle subconfiguration\rangle ::= \langle goal\rangle \mid \langle local\ goal\rangle \mid \langle guarded\ goal\rangle$$

The *subgoals* of a subconfiguration G are defined as follows:

1. G is a subgoal of G,

2. the subgoals of $G_1,\ldots,G_n$ are subgoals of $\mathbf{and}(G_1,\ldots,G_n\ ;\ \theta)_V$, of $\mathbf{or}(G_1,\ldots,G_n)$, and of $\mathbf{choice}(G_1,\ldots,G_n)$,

3. the subgoals of G are subgoals of G % B, where "%" is a guard operator.

Each subgoal has an *environment*, defined as the conjunction of the constraints of all the and-boxes in which the goal occurs.

Computation is modelled by rewrite rules defined in the following sections. The rewrite rules are applied to occurrences of subgoals of a configuration. Thus, a rewrite rule

$$Left \Rightarrow Right$$

is to be understood as defining a transition

$$G[Left] \Rightarrow G[Right]$$

on a configuration G, in which one occurrence of subgoal *Left* is replaced with *Right*.

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 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. The letters V and W are sets of variables.

4.3　Primitive Constraint Rules

A constraint subgoal may be rewritten by one of the following two rules, corresponding to successful execution and failure.

Some constraint operations, corresponding to the actions of existing languages like Prolog, GHC, Parlog, and Atomic Herbrand, are described in section 6.

The notation $op(\sigma)$ denotes a constraint operation op applied to the primitive constraint σ. (The constraint σ may optionally be the result of a "built-in" procedure, working in the environment of the constraint operation.)

Constraint Imposition

An and-box may be rewritten by

$$\mathbf{and}(C, op(\sigma), D \; ; \theta) \Rightarrow \mathbf{and}(C, D \; ; \sigma \wedge \theta),$$

if the activation condition of $op(\sigma)$ is satisfied, and $\sigma \wedge \theta$ is satisfiable.

Constraint Failure

An and-box may be rewritten by

$$\mathbf{and}(C, op(\sigma), D \; ; \theta) \Rightarrow \mathbf{fail},$$

if σ is inconsistent with the environment of $op(\sigma)$.

4.4　Local Forking

Local forking is the creation of a choice-box. An atomic subgoal is replaced by a choice-box which contains guarded goals that are derived from the clauses defining the goal. The guards of these guarded goals are then available for further evaluation.

Compared to section 2, head unification is reduced to simple parameter passing. Therefore we perform the substitution immediately.

Local Forking

An atomic subgoal A may be rewritten by

$$A \Rightarrow \mathbf{choice}(\mathbf{and}(G_1 \; ; \mathbf{true})_{V_1} \; \% \; B_1, \ldots, \mathbf{and}(G_n \; ; \mathbf{true})_{V_n} \; \% \; B_n),$$

using the sequence of clauses H_i :- $G_i \; \% \; B_i$ defining A (in order), in which the actual parameters of A are substituted for the formal parameters of H_i, and where the local variables V_i are renamed apart from other variables in the configuration.

4.5　Deterministic Promotion

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

Deterministic Promotion

An and-box may be rewritten by

$$\mathbf{and}(C, \mathbf{choice}(\theta_V \ \% \ B), D \ ; \sigma)_W \Rightarrow \mathbf{and}(C, B, D \ ; \theta \wedge \sigma)_{V \cup W}$$

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

4.6 Pruning and Indeterministic Promotion

If the guard of a guarded goal in a pruning choice-box is successful, some or all of its siblings will be pruned. Eventually, its body might replace the choice-box by deterministic promotion. The combination is called *indeterministic promotion*.

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

Cut and Commit

$$\mathbf{choice}(P, \sigma \ ! \ B, Q) \Rightarrow \mathbf{choice}(P, \sigma \ ! \ B)$$
$$\mathbf{choice}(P, \sigma \ | \ B, Q) \Rightarrow \mathbf{choice}(\sigma \ | \ B)$$

The actual promotion is performed by the deterministic promotion rule.

4.7 Nondeterministic Promotion

The successful guard branches in a wait choice-box are extracted by *nondeterministic promotion*. A new or-box is created, which has two branches. In the first branch, the choice-box is replaced with the body of the extracted solution. In the second branch, the rest of the choice-box remains.

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.

A subgoal is in an *ordered context* if it occurs in a cut choice-box which is also the closest surrounding pruning choice-box. Otherwise, it is in an *unordered context*.

Nondeterministic Promotion

We may rewrite an and-box by

$$\mathbf{and}(C, \mathbf{choice}(P, (\sigma_V : B), Q), D \ ; \theta)_W \Rightarrow$$
$$\mathbf{or}(\mathbf{and}(C, B, D \ ; \theta \wedge \sigma)_{V \cup W}, \mathbf{and}(C, \mathbf{choice}(P, Q), D \ ; \theta)_W),$$

if $\theta \wedge \sigma$ is satisfiable. In an unordered context, this rule may select any successful branch. In an ordered context, the sequence P is restricted to be empty. Then the rule selects the leftmost solution. Note that this does not introduce any incompleteness as the cut operation needs a successful leftmost branch.

4.8 Normalisation Rules

In a deep computation, inconsistent environments must be detected, failures must be propagated, and alternative solutions of guards must be dealt with. This is done by the following rules for environment synchronisation, failure propagation, and or within a choice reduction.

Environment Synchronisation

$$\mathbf{and}(B \; ; \sigma) \Rightarrow \mathbf{fail},$$

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

Failure Propagation

$$\mathbf{and}(B, \mathbf{fail}, C \; ; \sigma) \Rightarrow \mathbf{fail}$$

$$\mathbf{choice}(S, (\mathbf{fail} \% B), T) \Rightarrow \mathbf{choice}(S, T)$$

Or within Choice Reduction

$$\mathbf{choice}(S, \mathbf{or}(\beta, P) \% B, T) \Rightarrow \mathbf{choice}(S, \beta \% B, \mathbf{or}(P) \% B, T)$$

These rules propagate the consequences of the local forking and the promotion rules.

5 Control of the Computation Model

In this section, the control of the Kernel Andorra Prolog computation model is defined. A precise definition is given in terms of admissible computation steps. First, the relationship with the basic Andorra Model is discussed.

5.1 Determinacy Detection

The heuristic of the Andorra Model is that any deterministic step should be preferred to a nondeterministic step (within the same and-box), as the latter is likely to multiply work.

In the Andorra Model, there is a determinacy detector that examines all atomic goals. It tries to establish that a goal is deterministic, basing its conclusion on compile time information extracted from the definition for the goal.

In the Kernel Andorra Prolog computation model, the equivalent of determinacy detection for an atomic goal is local forking of the goal and the subsequent rewrite steps applied to and within the resulting choice-box. Any of these steps can potentially make the choice-box reducible by deterministic promotion or normalisation.

Thus, in the generalised model, the deterministic phase of an and-box should involve performing local forking, and as much computation as possible within the resulting choice-boxes, to detect determinacy, and also all deterministic promotions and normalisation rewrites that become applicable as a result of the determinacy detection.

5.2 Blocking and Pruning

The Kernel Andorra Prolog language has two extensions that complicate determinacy detection: blocking constraint operations, and pruning guard operators.

A natural extension of the Andorra Model is that nondeterministic promotion of an and-box should be postponed until all deterministic computation has ceased in siblings and parents that might affect constraints in the box.

Indeterministic promotion is a potential source of (unnecessary) incompleteness. Decisions are easily made that are inconsistent with what is produced by computation on a sibling or parent level.

A natural extension of the Andorra Model is that pruning of a choice-box is postponed until all deterministic computation has ceased in siblings and parents that might affect constraints in the box, and thereby the determinacy of the choice, or the constraint on which the indeterministic choice is based.

A constraint σ_V is *quiet* if it does not restrict external variables, i. e.

$$\theta \rightarrow \exists V(\sigma_V),$$

where θ is the environment of σ_V. Otherwise, the constraint is *noisy*. A guard execution is *quiet* (*noisy*) if it reduces to a quiet (noisy) constraint. An application of a pruning rule is *quiet* (*noisy*) if the successful guard is quiet (noisy).

The key property of quiet pruning is that the constraint on which it is based cannot be affected by computation performed outside the pruned choice box. As will be seen, quiet pruning can be ensured by the use of proper constraint operations.

5.3   Kernel Andorra Prolog Computations

We summarise the above discussion in precise terms as follows.

Local forking, deterministic promotion, normal form rules, and constraint rules are called *guess-free* rules. Nondeterministic promotion and pruning are called *guessing* rules.

A subgoal G is *stable* if no guess-free rules are applicable to or within G, and no constraint σ of an and-box or a constraint operation occurring in G restricts variables outside V, where V is the union of variables local to and-boxes in G in which σ occurs.

The following rule applications are *admissible*.

- An application of a guess-free rule is always admissible.

- An application of nondeterministic promotion is admissible iff

    1. it is applied to (or to a subgoal of) a stable subgoal, and

    2. there are no admissible applications of guessing rules to proper subgoals of the rewritten subgoal, i. e. it is innermost.

- An application of pruning is admissible iff

    1. it is applied to (or to a subgoal of) a stable subgoal, or

    2. the solution on which the pruning is based is quiet.

A *Kernel Andorra Prolog computation* is a (possibly infinite) sequence of configurations. It is obtained by successive admissible applications of rewrite rules, starting with an *initial configuration* of the form $\mathbf{and}(A_1, \ldots, A_n \ ; \ \mathbf{true})_V$, where V contains the variables that occur in $A_1, \ldots, A_n$.

This definition provides the *minimum* control which is characteristic for our computation model. Further restrictions, such as sequential execution of goals and leftmost nondeterminate promotion (as in Prolog) may be imposed to achieve desired control effects.

6 The Primitive Constraint Operations

In this section, some primitive constraint operations are introduced. They are defined by their *activation conditions*. These conditions express, using entailment, how constrained (or instantiated), the arguments are required to be before execution.

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 might lead to unpredictable behaviour, since for instance some combinations are not commutative. Section 7 discusses possible user languages.

We introduce four constraint operations. They have their motivation as being the operations used by the languages we seek to subsume.

Ask The Ask operation is only activated if its constraint is quiet. This operation is used by the Parlog $==/2$ operation and by Ask operations in Atomic Herbrand.

Tell$_0$ The Tell$_0$ operation may impose new constraints on local variables (the zeroth level). It is used by GHC.

Tell$_1$ The Tell$_1$ operation may impose new constraints on local variables and on any variables that are local to the closest parent and-box (on the first parent level). It is similar to Tell in Atomic Herbrand.

Tell$_\omega$ The Tell$_\omega$ operation is an unrestricted constraint operation, as found in Prolog. It may impose new constraints on any variable at any time.

Formally, this is expressed as follows. The and-box α contains the constraint operations $op(\sigma)$,

$$\alpha = \mathbf{and}(\ldots, op(\sigma), \ldots; \sigma_\alpha)_{V_\alpha}.$$

The and-box α has a closest parent and-box β (where we assume the or-reduced form),

$$\beta = \mathbf{and}(\ldots, \mathbf{choice}(\ldots, \alpha \% B, \ldots), \ldots; \sigma_\beta)_{V_\beta}.$$

When there is no parent and-box its constraint is assumed to be true and its set of local variables empty.

The activation conditions for the above constraint operations follow. The environment of the constraint operation $op(\sigma)$ is θ.

Ask The constraint σ imposes no new constraints on any variable, i. e. $\theta \rightarrow \theta \wedge \sigma$.

Tell$_0$ The constraint σ imposes no new constraints on variables external to α, i. e. $\exists V_\alpha \theta \rightarrow \exists V_\alpha (\theta \wedge \sigma)$.

Tell$_1$ The constraint σ imposes no new constraints on variables external to α and β, i. e. $\exists (V_\alpha \cup V_\beta) \theta \rightarrow \exists (V_\alpha \cup V_\beta)(\theta \wedge \sigma)$.

Tell$_\omega$ No condition.

The conditions are not shown in their minimal logical form, but rather in a form that shows how the scope of the operations grows from Ask to Tell$_1$. Another interesting variant of a tell-unification operation which will be used in the next section is the following.

Tell$_{(!/|)}$ A constraint operation of this type may only restrict variables that are local to and-boxes within the closest surrounding pruning choice-box.

This operation has a dynamic blocking behaviour.

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. For verifications of the remarks below on the formal properties of these languages, see [2].

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 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.

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.

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.

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.

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 that the G-part of the guard is quiet. The language allows noisy pruning in a shallow way and can still 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 while constraining pruning guards to be quiet. It is obtained by using $\mathrm{Tell}_{(!/|)}$ operations uniformly in all guarded clauses. We have written several nice examples, finite domain constraint and the like, in this language.

8 Related Work

Vijay Saraswat defined a language CP [7] that provides among other things deep guards, an operator called "don't know commit", related to our "wait" operator, and the concept of blocks, which are similar to and-boxes. One of the main differences between CP and our work is our control of *when* to promote nondeterminism. This is also true of Saraswat's thesis [8]. Also, we emphasise fully interleaved execution in a language with deep guards. However, Kernel Andorra Prolog is definitely a concurrent constraint language.

The main influence for the Kernel Andorra Prolog model was the basic Andorra Model proposed by David H. D. Warren [10]. The final result owes much to the mutual exchange of ideas within the PEPMA Esprit Project, where Warren has been considering an Extended Model, based on the basic Andorra Model, which is closely related to Prolog, and in which the kind of control that is expressed using constraint operations and wait guards in our language is largely implicit.

9 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.

Simple implementations of some of the instances of this framework will not achieve the speed of some implementations of more restricted 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. Our preliminary implementation study suggests that the execution speed of a first general single-processor implementation of Atomic Kernel Andorra Prolog will be close to the speed of Prolog for deterministic flat computation within an and-box. This will be the topic of future papers.

Acknowledgements

References

[1] Reem Bahgat and Steve Gregory. Pandora. In *Proceedings of the ICLP-89*, 1989.

[2] Torkel Franzén. Formal aspects of Kernel Andorra Prolog. SICS Research Report in preparation, 1990.

[3] Seif Haridi. A logic programming language based on the Andorra model. *New Generation Computing*, 1990. Forthcoming issue.

[4] Seif Haridi and Per Brand. Andorra Prolog, an integration of Prolog and committed choice languages. In *Proceedings of the FGCS*, 1988.

[5] Sverker Janson and Johan Montelius. Implementing Kernel Andorra Prolog. SICS Research Report in preparation, 1990.

[6] William Kornfeld. Constraint programming in Andorra Prolog. Presented at the Swedish-Japanese-Italian workshop, 1989.

[7] Vijay A. Saraswat. The concurrent logic programming language CP: Definition and operational semantics. In *Proceedings of POPL*, 1987.

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

[9] Vijay A. Saraswat. Programming in Andorra Prolog. Technical report, Xerox PARC, 1989.

[10] David H. D. Warren. The Andorra principle. Presented at the Gigalips workshop, Stockholm, 1987.

[11] Rong Yang. Implementation notes on Andorra-I. Internal Report, 1989.

[12] Rong Yang. Solving simple substitution ciphers in Andorra-I. In *Proceedings of the ICLP-89*, 1989.