where U is the set of variables that were given as local variables to the corresponding initial configuration. This technicality is necessary for the notion of logical computation introduced below.

The logical interpretation of a given program Σ is a set of formulas Σ*, which is referred to as the *completion* of Σ. In the following, **T** stands for the union of Σ* and **TC**, the given theory of constraints.

4.8.2 Soundness

A computation $\gamma_0 \to \gamma_1 \to \cdots$ is *logical* if $\mathbf{T} \supset \gamma_i^* \equiv \gamma_{i+1}^*$ for every non-final i. A program is *logical* if every computation from that program is logical. The following properties clearly hold for logical programs.

PROPOSITION (soundness of answers).

If σ *is an answer to a query* A *then* $\mathbf{T} \supset \sigma \supset A^*$.

PROPOSITION (soundness of failure).

If there is a computation leading from a query A (*with free variables* V) *to a final configuration*

$$(\mathbf{and}()^{\sigma_1}_{V_1}, \ldots, \mathbf{and}()^{\sigma_n}_{V_n})$$

then $\mathbf{T} \supset A^* \supset \exists W_1\sigma_1 \vee \cdots \vee \exists W_n\sigma_n$, *where* W_i is $V_i \setminus V$.

In particular, if $n = 0$ *then* $\mathbf{T} \supset \neg A^*$.

AKL programs are logical if all conditional choice statements have *indifferent* guards, roughly meaning that if there are multiple solutions of a guard the result of the choice is the same for all, and if all committed choice statements have *authoritative* guards, roughly meaning that if more than one clause is applicable, then the choice between clauses (and solution within guards) will not matter for the result of the choice. For logical conditional choice, guards need not be indifferent.

One trivial case of indifference is that in which guards do not have multiple solutions simply because nondeterminate choice is not used. Another simple case of indifference is that in which no local variables are shared between the guard and the body. As a consequence, negation as failure is sound in AKL, since there are no variables that can be shared.

not_p(X) := (p(X) → fail ; true)

Together, these two cases cover a majority of the conditional choice statements which are written in practice.

The notion of authoritative guards is less useful. It covers some important cases, such as the use of commit for constraint propagation, where clauses have the same, or overlapping, logical content, exemplified by

```
and(X, Y, Z) :=
    ( X = 0 | Z = 0, ( Y = 0 ; Y = 1 )
    ; Y = 0 | Z = 0, ( Y = 0 ; Y = 1 )
    ; X = 1, Y = 1 | Z = 1
    ; Z = 0, X = 1 | Y = 0
    ; Z = 0, Y = 1 | X = 0
    ; Z = 1 | X = 1, Y = 1 ).
```

and also the following more complete form of negation.

```
not_p(X) := ( ( p(X) | true ) → fail ; true )
```

The use of commit in a program such as merge, where the choice gives rise to logically distinct solutions, is not authoritative.

4.8.3 Completeness

A computation is [redacted] computations terminate. A computation is *or-fair* if all non-terminal or-components are eventually rewritten.

PROPOSITION (completeness). [redacted]

Let Σ *be a program using only atoms, composition, hiding, and nondeterminate choice. If* Π *is an or-fair normal computation starting with a query* A*, and*

$\Sigma^* \supset A$ [redacted]

then some configuration in Π *has an or-component* σ_V *such that*

TC $\theta \supset \exists V \setminus U\sigma$

where U *are the free variables in* A.

4.8.4 Confluence

Two computations

Π: $\gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \cdots$

Π': $\gamma_0 \rightarrow \gamma'_1 \rightarrow \gamma'_2 \rightarrow \cdots$

starting from the same initial goal γ_0, are *confluent* if for every γ_i there is a γ'_j such that $\gamma_i = \gamma'_j$ or $\gamma_i \rightarrow^* \gamma'_j$ (where $\rightarrow^*$ is the transitive closure of $\rightarrow$), and conversely with Π and Π′ interchanged.

Informally, a *complete* computation is either finite, or no subgoal appears in the course of the computation such that (1) the subgoal survives unchanged throughout the computation, and (2) for infinitely many configurations in the computation, some rule is applicable to the subgoal.

PROPOSITION (confluence).

If computations are complete, the program contains no commit procedures or aggregates (and the applicability of choice splitting is somewhat further restricted to avoid interference between two possible such steps, where one step prevents the stability of another)*, then any two computations are confluent*.

One condition on choice splitting that guarantees confluence is to select the leftmost candidate in an innermost stable box.

4.9 RELATED WORK

Several more or less closely related computation models exist. The early models have in common that they are based on the constraint theory of trees and unification in an operational manner, which cannot easily be generalised to arbitrary constraints. Notions such as quietness and stability can then not be used for synchronisation. Instead more ad hoc notions are used that are related to bindings on individual variables. The models that do provide don't know nondeterminism do not provide aggregates.

Gregory [1987] describes a computation model for Kernel PARLOG, which is reminiscent of the computation model of AKL in that it models a language with deep guards using rewrite rules on an AND/OR tree. It does not deal with don't know nondeterminism nor with general constraints. On the other hand it provides sequential composition. A detailed comparison with PARLOG at the language level is given in Section 8.2.

Saraswat [1987a; 1987b; 1987c] presents a computation model for the to AKL closely related language CP[↓,|,&,;]. CP provides deep guards as well as don't know nondeterminism via the don't know guard operator '&'. It is not based on general constraints. Synchronisation is expressed with '↓' annotations on terms, which say that such terms may not be bound to external variables. Commit is not quiet, but relies on '↓' for synchronisation. Don't know nondeterminism is not synchronised, as with stability, but may happen at any time. There are no aggregates. The nature of '↓' makes a detailed comparison with AKL difficult, but AKL restricted to flat guards can be regarded as a subset of the more recent cc framework of Saraswat [1989], and their relation is discussed in Section 8.4.

Warren [1989] considers a computation model called the Extended Andorra Model (EAM), which is based on rewrite rules on a tree of and- and or-boxes closely related to those of AKL. The EAM is not based on constraints and does not encompass aggregates. It is mainly intended for parallel execution of full Prolog. This is clearly visible in its control scheme, which is based on a sequential flow of data from the left to the right in a configuration. The leftmost occurrence of a variable is regarded as a producer. Bindings attempted at other occurrences suspend. The EAM and AKL models evolved in the ESPRIT project PEPMA (EP 2471).

Haridi and Janson [1990] describes the Kernel Andorra Prolog (KAP) framework, which is an immediate antecedent of AKL. The main difference is that KAP is not based on quietness for the pruning guards. A similar effect is achieved by synchronising constraint operations, which move a constraint to the constraint store only if certain conditions hold on its relation to its environment w.r.t. variables at different levels. These operations destroy confluence and were abandoned in AKL. Atomicity of constraint operations, pruning, and

promotion was also a concern. In practice, such concepts do not carry over easily to constraint systems other than trees, and are not present in AKL.

Smolka [1994] presents a calculus for the deep-guard higher-order concurrent constraint language Oz. It is based on feature constraints, which are internalised by relative simplification rules. Constraints do not form environments, but "permeate" through the computation state by congruence rules. Synchronisation is via quietness and determinism, as in AKL. Oz does not use fixed form constructions such as boxes, but relies on congruence rules to provide multiple views that are exploited in the transition rules. A main difference is that Oz is higher-order. Programs do not exist outside the computation state, but are available on blackboards, which may be regarded as associated with and-boxes (in AKL terminology) in much the same way as constraint stores. A recent extension offers don't know nondeterminism in a manner that exploits the higher-orderness of the language [Schulte and Smolka 1994]. Oz is compared with AKL in more pragmatic terms in Section 8.5.

CHAPTER 5

AN EXECUTION MODEL

Write down an initial configuration. Make transitions with respect to the left-most possible goal. Stop when no transition is possible. This is a perfectly viable *execution model*, which respects the computation model but makes specific decisions about which step to make when, according to some control principles. By an *execution* we mean a computation following these principles.

The execution model presented below is an approximation of the one above. Goals are examined from the left to the right, and computation rules are applied where possible. When new constraints make rules applicable to goals that have already been examined, these goals are re-examined in any order.

The idea is that each operation should have the same cost as a corresponding operation in a real implementation (in an informal sense). To this end, auxiliary control information is added, which is not strictly necessary to express the control desired, but is necessary for efficiency. The execution model may be regarded as a "missing link" between the computation model and an abstract machine or other forms of implementations. It may also simply be regarded as an algorithm that produces computations.

5.1 OVERVIEW

The concept of a *worker* which performs execution steps and moves about in the configuration is useful as a mental model.

Goals are given *labels* which allow a worker to refer to occurrences of goals in configurations and to identify goals in subsequent configurations.

Lists of *tasks* and *contexts* are used to keep track of parts of a configuration that are to be examined.

Dependencies between constraint stores in the hierarchy formed by a configuration are maintained by *suspensions*.

The execution model is formalised as a transition system on *execution states*, where transitions are "driven" by tasks.

It is shown that the execution model produces a computation. It may, however, produce an infinite computation even if finite computations from the same initial configuration exist. It is not guaranteed to produce complete (fair) computations.

The execution model makes no assumptions about the choice of candidates.

5.2 WORKERS

Let us think of execution steps as being performed by a *worker*, an automaton that performs computation steps according to the given control principles. This notion plays no rôle in the formal definition of the execution model, but may be of help as a mental model.

In this dissertation, only *single worker execution* is explored, multiple worker execution being outside its scope, and the topic of related research [Montelius and Ali 1994]. Thus, the execution model presented is for one worker.

A worker is located in a box, the *current box*. Its constraint store is referred to as the *local (constraint) store* and variables in the set of local variables as the *local variables*. Its environment is referred to as the *environment*, and constraints stores and local variables of and-boxes containing the current box are referred to as *external (constraint) stores* and *external variables*, respectively.

The current box is changed by *moving* between boxes. When moving to a box that is contained within the current box, the worker is said to move *down*. When moving to a box containing the current box, the worker is said to move *up*. When moving, the worker moves step by step, to each intermediate box.

5.3 LABELLED GOALS

Goals will need a persistent identity, and are for this purpose given *labels*, a label being anything that may serve as an identifier (e.g., a numeral). Labels allow us to identify goals in different, subsequent configurations. The letters *i*, *j*, *k*, and *l* stand for labels. A *labelled goal* is written as *i*::G, where *i* is the label and G is the goal.

The definitions pertaining to configurations clearly carry over to *labelled configurations*, in which all goals are labelled. Thus, we will freely use contexts and computation rules with labelled goals.

Upon creation, each goal is given a label which is unique for the whole execution. Rewriting the interior of a goal does not change its label. How rules preserve labels should be quite clear except for the choice splitting rule, which copies goals. In this rule, new labels are given to all goals in the left branch.

Labelled goals occurring in a configuration are *live* with respect to this configuration. A goal which is not live is *dead*.

5.4 TASKS AND CONTEXTS

The *task* is the basic unit of work. A worker associates tasks with boxes. Tasks are thought of as being *in* boxes (from the point of view of a given worker). A worker keeps track of its tasks, and which tasks are in which boxes. A task is deleted when it has been processed, or when it is in a box which is deleted because of failure or promotion. Tasks associated with and-boxes are *and-tasks*, and tasks associated with choice-boxes or or-boxes are *choice-tasks*.

⟨task⟩ ::= ⟨and task⟩ | ⟨choice task⟩

⟨and task⟩ ::=	**s**(⟨label⟩)	(*statement*)
	a(⟨label⟩)	(*and-box*)
	w(⟨label⟩, ⟨label⟩)	(*wake*)
	in(⟨label⟩)	(*install*)
⟨choice task⟩ ::=	**c**(⟨label⟩)	(*clause*)
	o(⟨label⟩)	(*or-box*)
	cs(⟨label⟩)	(*choice splitting*)
	p(⟨label⟩)	(*promote*)

Tasks are processed in a *last-in first-out* fashion, and are kept in *lists*. These are grouped in contexts, corresponding to the choice-box and and-box levels.

⟨and list⟩ ::= ε | ⟨and task⟩.⟨and list⟩

⟨choice list⟩ ::= ε | ⟨choice task⟩.⟨choice list⟩

⟨context list⟩ ::= ε | (⟨and list⟩, ⟨choice list⟩).⟨context list⟩

The symbol ε stands for the empty list.

To each kind of task correspond *task execution rules*. The purpose of a rule is usually to evaluate the applicability of a computation rule to the goal with the associated label, and apply it if it is applicable. It may also create new tasks.

5.5 SUSPENSIONS AND WAKING

Constraint stores which are neither quiet, nor incompatible with their environments, may become so when new constraints are added to constraint stores in their environment. For computational efficiency, it is important to make a reasonably precise, but safe, guess on which stores are affected.

The standard technique is to tie this knowledge to the variables involved in terms of *suspensions*. Suspensions are expressions of the form *v.i*, where *v* is a variable and *i* is the label of an and-box. When a store somehow affects an external variable, its and-box is referred to by a suspension on this variable. (It is *suspended on* the variable.) When a new constraint somehow affects a variable, suspensions on this variable are *waked*, meaning that the associated stores are re-examined. This section attempts to give some formal meaning to the notion(s) of to “somehow affect”.

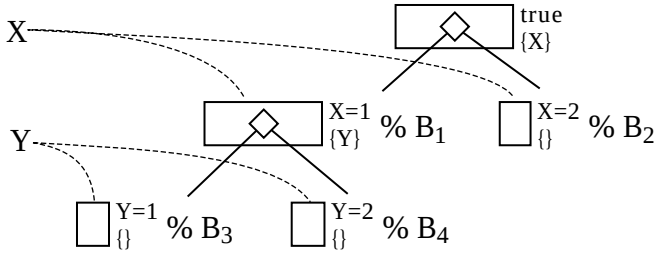

Figure 5.1. A hierarchy of constraint stores with suspensions

5.5.1 Conditions for Suspending and Waking

We discuss the problem in terms of constraints σ, θ, and τ, where σ is $\exists U \sigma'$ for the local variables U and local store σ' of an and-box in a configuration, θ is its environment, and τ is a constraint which will be added to the environment. It is known that $\exists(\theta \wedge \sigma)$ and $\exists(\theta \wedge \neg\sigma)$. Given that also $\exists(\theta \wedge \tau)$ and $\exists(\theta \wedge \neg\tau)$, the problem is to establish whether $\exists(\theta \wedge \tau \wedge \sigma)$ and $\exists(\theta \wedge \tau \wedge \neg\sigma)$, i.e., that σ is still neither incompatible nor quiet.

This is illustrated in Figure 5.2, which depicts a situation with the ranges of possible values for two variables over the real numbers. The addition of τ_1 will make σ entailed, and τ_2 will make it incompatible.

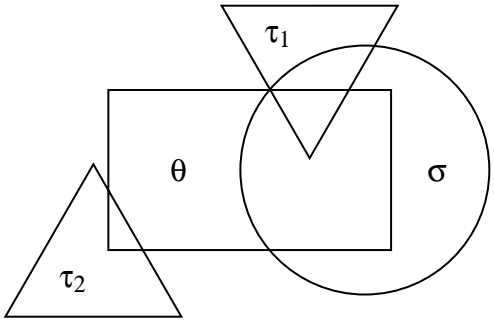

Figure 5.2. Ranges of possible values for variables

First, a way of using suspensions is given that is known to be sufficient on the basis of a simple logical relation between σ, θ, and τ. Then we discuss the special case of rational trees. In the following, the sets U and V partition the free variables in σ, θ, and τ.

A set V of variables can be *ignored for suspension* if

$$\forall V\, [\exists U\, \theta \supset \exists U\, (\theta \wedge \sigma) \wedge \exists U\, (\theta \wedge \neg\sigma)]$$

If a constraint τ only on V is added, σ will neither be incompatible nor quiet. Thus, it is sufficient to suspend the and-box of σ on all variables in U, and wake on all variables in τ.

For example, if θ is **true**, and σ is X = f(Y), then

$$\forall Y\ [\exists X\ (X = f(Y)) \wedge \exists X\ (X \neq f(Y))]$$

but not

$$\forall X\ [\exists Y\ (X = f(Y)) \wedge \exists Y\ (X \neq f(Y))]$$

and so {Y} can be ignored for suspension but not {X}.

The set which can be ignored is not unique. Any subset of such a set can be ignored, and they may also be disjoint or partially overlapping.

For example, if θ is **true**, and σ is X < Y, where < is over, e.g., the real numbers, then

$$\forall X\ [\exists Y\ (X < Y) \wedge \exists Y\ \neg(X < Y)]$$

and

$$\forall Y\ [\exists X\ (X < Y) \wedge \exists X\ \neg(X < Y)]$$

and so both {X} and {Y} can be ignored for suspension.

The above does not suffice to explain a suitably optimised suspension and waking scheme for rational trees. The following two schemes (given without proof) require multistage waking, which is waking also when re-entering a local store. This is not necessary for the above notion of ignored variables. We reason about the simplified form of constraints, using the relative simplification rules for rational tree constraints introduced in Section 4.4.7.

It is sufficient to suspend on *v* when adding a simplified equation of the form *v* = f(…), where *v* is an external (free) variable, and on *u* and *v* for equations of the form *u* = *v*, where both *u* and *v* are external (free) variables, and to wake on *v* when adding a simplified equation of the form *v* = *t*.

For example, if θ is $X_1 = f(Y_1) \wedge X_2 = f(Y_2)$ and σ is $X_1 = X_2$ then σ is simplified to Y_1 and Y_2, and it is sufficient to suspend on Y_1 and Y_2.

It is also sufficient to instead suspend only on *v* when adding a simplified equation of the form *v* = *t*, where *v* is an external (free) variable, and to wake on *v* when adding a simplified equation of the form *v* = f(…), and on *u* and *v* for equations of the form *u* = *v*. The previous scheme is preferred, however, for efficiency reasons.

Are there more general notions, similar in spirit to ignored variables, that cover also this scheme? The problem seems to be open, but it is likely that this cannot be achieved without taking into account the syntactic form of constraints and the behaviour of the constraint solver.

5.5.2 Establishing Stability

There are several possible schemes for establishing stability efficiently, which vary in precision. A common property is that with each and-box is associated information on suspensions on external variables that refer to stores in this and-box. When suspending an and-box on an external variable, it is recorded in this

and-box and in all ancestor and-boxes for which the variable is external. A box is deemed stable if no such suspensions are recorded.

The schemes vary in how well they take care of the waking of a suspension and the failure of an and-box. When a suspension is waked and it is discovered that the reason for suspending holds no more, this should be recorded in ancestor and-boxes. When an and-box fails, it should be recorded in all ancestor and-boxes that all its suspensions have disappeared. If either of these cases is not taken care of, boxes will unnecessarily be deemed unstable.

This conservative approach is permitted by the computation model. Observe that the trivial case of stability is the quiescence of a topmost and-box, a situation which is always detected.

We will not get into the details of formalising a scheme here. We assume that stability is established by inspection of pertinent suspensions. A simpler scheme for the abstract machine will be described and used in Chapter 6.

5.5.3 Operations on Suspensions

The execution rules manipulate suspensions as follows.

We will speak of sufficient sets of variables for suspension of stores in their environments. This can be, e.g., the complement of a set that can be ignored for suspension.

We will also speak of sufficient sets of variables for waking when adding new constraints to environments. This can be, e.g., all variables in the constraint, as in the scheme based on ignoring variables for suspension.

When promoting a store, it is necessary to wake as when adding the corresponding constraints, as this is new information w.r.t. other dependent stores. If existing suspensions that refer to the promoted and-box are replaced by suspensions referring to the destination and-box, there is no need to add more suspensions, since the incompatibility or quietness of either or both of the two stores involved is a consequence of the incompatibility or quietness of their combination.

When performing choice splitting, suspensions of copied and-boxes need to be duplicated. If the variable is “copied”, the new suspension is on the new variable, otherwise on the original. In both cases, the new suspension refers to the new and-box.

When re-entering a store via waking, the relation between this store and its environment has changed, and it becomes necessary to suspend it on new variables. When using ignored sets for suspending, it is not necessary to also wake, as if adding the local store as new. This is, however, necessary for the optimised treatment of rational trees.