AKL

A Multiparadigm Programming Language

AKL

A Multiparadigm Programming Language Based on a Concurrent Constraint Framework

by
Sverker Janson

A Dissertation submitted
in partial completion of the requirements
for the Degree of Doctor of Philosophy
Uppsala University
Computing Science Department

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

Swedish Institute of Computer Science
Box 1263, S-164 28 Kista
Sweden

Uppsala Theses in Computing Science 19
ISSN 0283-359X
ISBN 91-506-1046-5

SICS Dissertation Series 14
ISRN SICS / D--14--SE
ISSN 1101-1335

Doctoral thesis at Uppsala University 1994
from the Computing Science Department

Abstract

Janson, S., 1994. AKL—A Multiparadigm Programming Language.
Uppsala Theses in Computing Science 19. 212 pp. Uppsala. ISSN 0283-359X.
ISBN 91-506-1046-5.

New programming languages conceived by adding yet another permutation of new features on top of established languages offer only complexity and confusion to software developers. New basic principles are necessary that support the desired functionality using a minimum of concepts.

This thesis reports on an investigation into principles for combining the constraint solving and don't know nondeterministic capabilities of Prolog and the constraint logic programming languages with the process-describing capabilities of concurrent logic languages such as GHC. The result, AKL, is a coherent language supporting multiple programming paradigms, such as concurrent, object-oriented, functional, logic, and constraint programming. In addition, AKL offers a large potential for automatic parallel execution.

The operational semantics of AKL is captured by a computation model, involving rewriting of “semi-logical” expressions that form the computation states. Constraints are used to express data and interaction. The computation model is then augmented with control, giving an execution model, which demonstrates how to perform computations in a systematic manner. Finally, an abstract machine brings us close to a real implementation. It is similar to the one underlying the AGENTS programming system for AKL developed at SICS by the author and his colleagues.

The research reported herein has been supported by the Swedish National Board for Industrial and Technical Development (NUTEK), and by the sponsors of SICS: Asea Brown Boveri AB, Ericsson Group, IBM Svenska AB, Nobel-Tech Systems AB, the Swedish Defence Material Administration (FMV), and Swedish Telecom.

Sverker Janson, Computing Science Department, Uppsala University,
Box 311, S-751 05 Uppsala, Sweden

ISSN 0283-359X
ISBN 91-506-1046-5

Printed in Sweden by Graphic Systems, Stockholm, 1994.

To Kia, Adam, and Axel

CONTENTS

PREFACE

This is not intended as a standardising document for AKL, which is a young language, in need of freedom. However, the basic investigations of language principles and corresponding implementation techniques have both reached a point of quiescence where they can benefit from an exposition of this kind.

Readers acquainted with early publications on AKL will notice a few modifications: various improvements of syntax and terminology which are regarded as a step forward, leaving old dross behind.

CONTRIBUTIONS

Research is an incremental activity; stone is laid upon stone. Although this makes it difficult to single out the unique aspects of any one particular work, an attempt is made to specify those ideas and achievements that are believed to be unique to the research reported in this dissertation.

The original contributions reported in this dissertation are:

- basic principles for don't know nondeterministic concurrent constraint logic programming languages that combine, coherently and uniformly, the *searching* and *constraint solving* ability of the constraint logic programming languages (e.g., Prolog) with the *process describing* ability of the concurrent logic programming languages (e.g., GHC)
- improved control and synchronisation for such languages, in particular the notion of *stability*
- new combinators for such languages, e.g., *logical conditional*, that are amenable to stronger logical interpretations
- a simple formal computation model for such languages
- an implementation methodology based on a stepwise refinement of the formal computation model via an execution model to an abstract machine
- basic implementation techniques for languages featuring don't know nondeterminism and hierarchical constraint stores, which can serve as a foundation for further refinements and optimisations

- the notion of *ports* for efficient process communication and object-oriented programming in concurrent constraint languages

SOURCE MATERIAL

This dissertation, although a monograph, is to a large extent based on a number of previous publications and reports:

- Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its Computation Model. In Warren and Szeredi, eds., *Logic Programming: Proceedings of the Seventh International Conference*, MIT Press, 1990.
- Sverker Janson and Seif Haridi. Programming Paradigms of the Andorra Kernel Language. In Saraswat and Ueda, eds., *Logic Programming: Proceedings of the 1991 International Symposium*, MIT Press, 1991.
- Seif Haridi, Sverker Janson, and Catuscia Palamidessi. Structural Operational Semantics for AKL. *Journal of Future Generation Computer Systems* **8**(1992).
- Torkel Franzén, Seif Haridi, and Sverker Janson. An Overview of AKL. In *ELP'91 Extensions of Logic Programming*, LNAI 596, Springer-Verlag, 1992.
- Sverker Janson and Johan Montelius. The Design of a Prototype Implementation of the Andorra Kernel Language, Deliverable Report, ESPRIT project 2471 (PEPMA), December 1992.
- Sverker Janson, Johan Montelius, and Seif Haridi. Ports for Objects. In Agha, Wegner, and Yonezawa, eds., *Research Directions in Concurrent Object-Oriented Programming*, MIT Press, 1993.
- Sverker Janson and Seif Haridi. An Introduction to AKL—A Multiparadigm Programming Language. In *Constraint Programming*, NATO-ASI Series, Springer-Verlag, forthcoming.

I thank my co-authors for generously allowing me to use also parts of the material where no obvious borderlines between the contributions of different authors exist.

ACKNOWLEDGEMENTS

First and foremost, I express my profound gratitude to my friend and advisor Seif Haridi for his unparalleled enthusiasm and continuous support. He has contributed to all parts of this research in too many ways to describe them all.

I would like to thank my co-worker Johan Montelius for his energy, cheerfulness, and “good vibes”. We have had fruitful co-operation on implementation as well as on programming paradigms.

The formal sections of this dissertation draw heavily upon the terminology introduced by Torkel Franzén in his quest to analyse the formal aspects of AKL. Some of his results are also summarised in this dissertation. I thank him for his

diligent reading of many versions of the present text, and the numerous comments that led to its improvement.

I also thank the other members of the Concurrent Constraint Programming Group—Khayri M. Ali, Kent Boortz, Per Brand, Björn Danielsson, Dan Sahlin, Thomas Sjöland, Galal Atlam, Björn Carlson, Ralph Clarke Haygood, Torbjörn Keisu, and Khaled Shaalan—for taking part in the development of AKL and AGENTS. It has been, and will continue to be, team work.

Other colleagues at SICS, in particular my colleagues in the former Logic Programming and Parallel Systems Laboratory, have provided a highly stimulating working environment. Many thanks to them all.

Of my colleagues in Uppsala, I thank Johan Bevemyr for implementing the first prototype of the AGENTS programming system, Jonas Barklund and Håkan Millroth for advice on the process of submitting a dissertation, and not least Sten-Åke Tärnlund for his struggle for computer science at Uppsala University.

This work was in part funded by the ESPRIT projects PEPMA (EP 2471) and ACCLAIM (EP 7195). Sponsors of PEPMA were the Swedish National Board for Industrial and Technical Development (NUTEK), Ericsson Group, and Swedish Telecom. Sponsor of ACCLAIM was NUTEK. The partners of PEPMA and ACCLAIM provided valuable feedback to the research reported herein.

SICS is a non-profit research foundation, sponsored by the Swedish National Board for Industrial and Technical Development (NUTEK), Asea Brown Boveri AB, Ericsson Group, IBM Svenska AB, NobelTech Systems AB, the Swedish Defence Material Administration (FMV), and Swedish Telecom. I thank these companies and organisations for taking an active interest in Swedish computer science research.

Finally, I thank my wife Kia and sons Adam and Axel for their patience and encouragement, and my parents for raising me to believe that writing a dissertation is something that people typically do.

CHAPTER 1

INTRODUCTION

This chapter discusses the notion of programming language design. A number of design criteria are presented: general design principles as well as specific design goals for the present context.

1.1 WHY DESIGN PROGRAMMING LANGUAGES?

Why design yet another programming language? Do we not already have all the languages we need? Maybe not, languages evolve, even the languages that form the foundation of the software industry. Recently, a new paradigm, object-oriented programming, worked its way into respectability. It started out as an exotic creature, Simula, and even more exotic was Smalltalk, on its exotic host, a personal work station, with a bit-mapped graphic display. Research then, mainstream now. There is already an object-oriented COBOL. Surely, the next FORTRAN standard will include an object-oriented extension.

Some application areas, such as knowledge information processing, still do not enjoy strong support in the major programming languages. Writing such programs requires an undue amount of effort, just as writing object-oriented software is awkward without proper linguistic support.

A number of research languages provide very good support for such tasks, and some of them are commercially available. However, these languages are typically single paradigm languages. Even though other programming paradigms, such as object-oriented programming, would add expressiveness desirable for other aspects of application programming, the question of how to achieve such an augmentation of expressive power, *while staying within the computational framework at hand*, is and will continue to be an active topic for research, and was a strong motivation for the research presented in this dissertation.

Furthermore, a new generation of powerful and inexpensive multiprocessor computer systems has recently emerged, but the range of existing applications able to benefit directly is very limited. Ideally, it should be possible to exploit

any parallelism inherent in programs, by automatically determining which parts may be run in parallel, and by doing so when appropriate. To determine this, today's programming languages in popular use require data-flow analysis, since they allow almost arbitrary interactions between statements. Not all parallelism can be discovered with today's analysis techniques, but even if they eventually are perfected, interactions between statements are frequent, and there is usually not much parallelism there to be discovered anyway. It is not likely that this approach will ever be useful as a general means of exploiting the potential of these machines. Too little parallel execution can be achieved.

There are two obvious roads to better utilisation of multiprocessor systems. One is to train highly qualified designers and programmers to perform the intricate task of producing good parallel software, something that is likely to be cost-effective only for standard software. Or, since new programs have to be written anyway, design new, even more structured, programming languages, amenable to automatic exploitation of potential parallelism. Both approaches carry an initial training cost, but the payoff of the latter is likely to be greater since the task of writing the actual parallel programs becomes so much easier.

Our interpretation of the current situation is that there is still a great need to do basic research in programming language design and implementation. Existing languages and programming systems have been found wanting.

1.2 BASIC DESIGN PRINCIPLES

It is easy to list a number of programming language design principles that are so general and so obviously appealing that any designer in his right mind would claim adherence to them. But, although such a list may therefore seem as simple-minded as the lyrics of a simple love song, firsthand experience can often give it both depth and meaning.

Any programming language should be simple, expressive, and efficient, at least from the programmer's point of view, but these criteria should also be applicable to the computation model, and its formal manipulation, as well as to the basic implementation technology.

- A *simple* language has few basic concepts, with simple interactions. A simple computation model and a simple basic implementation are often a natural consequence.
- An *expressive* language supports concise implementations of a wide range of algorithms, abstractions, and overall program structures. Anyone can design an expressive, but complex, language by “feature stacking”; e.g., PL/I, ADA, and CommonLisp. Anyone can design a simple Turing-complete language; e.g., (most) assembly languages and BASIC, which obviously lack expressiveness. Any language design is a compromise between expressiveness and simplicity.
- An *efficient* language allows algorithms to be implemented with their theoretical time and space requirements, with the “usual” constant factor for

the computer in question. Efficiency is often best achieved by directing implementation efforts to a small number of fairly low-level basic constructs, with simple interactions. However, efficiency can also be achieved by providing very high-level constructs that have efficient implementations.

Where to strike the balance between simplicity, expressiveness, and efficiency, is determined by the intended application area. In this dissertation, some simplicity and efficiency of the programming language will be traded for the expressiveness required by knowledge information processing applications.

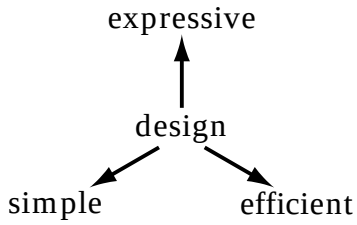


Figure 1.1. The balance between design principles

Another basic principle, still not as widely appreciated as one would expect considering that it has been well known since the early 1960s, is that the appearance (*syntax*) and the intended behaviour (*semantics*) of programs in a programming language should be *formally defined*. Some people in the business even seem to believe that formalism is an "egghead" notion, of little or no consequence in their daily life. Nevertheless, they can spend a large part of their life wrestling with the most rigid and formal of tools: computers and programming languages. Many of their problems, e.g., unexpected interactions between language features, stem from the fact that the definitions of these languages were not framed in a suitable formalism, which could have allowed an early pinpointing of difficulties in the design process. A suitable formalism allows the language designer to be precise, and thereby avoid inconsistency and ambiguity; it then allows the implementor to know exactly which behaviour to implement for any conceivable program; and it allows the programmer to know exactly what to expect in any situation.

Sometimes, simple formalisms can be used for the core of a language, e.g., SLD-resolution for Prolog, but do no longer suffice for the augmentations required to achieve a full-fledged programming language. The proper way to formalise Prolog behaviour has been debated for the better part of a decade. Indeed, there are many different approaches to defining programming languages.

A basic distinction is whether to have an *operational* or a *denotational* semantics. One tells us what programs do in a way that allows a corresponding implementation on a computer, the other what programs "mean" in terms of mathematical objects such as functions and relations. Having both is of course possible, but one should be regarded as the defining semantics.

Both have their advantages. A simple denotational semantics means that programs are simple from a mathematical point of view, which might simplify reasoning about a program. A simple operational semantics means that there is a simple basic implementation scheme, which might simplify predicting the actual behaviour of a program. In our experience, an operational semantics is to be preferred as the defining semantics, since it serves as a source of integrity for the language as a tool for programming. To also require properties such as soundness and completeness with respect to a particular semantics may be an additional constraint on the design.

Among operational semantics, we will make a distinction between *computation models* and *execution models*. A computation model defines computation states, transitions between computation states, and computations (as possibly nonterminating sequences of computation states derived by consecutive transitions). An execution model is a restriction of a computation model that should be capable of producing at least one, but not necessarily all, of the possible computations starting with a given state.

The *defining model* should be a computation model, which defines all legal computations for a given program. Associated with implementations are corresponding execution models, where particular choices may be made where the computation model offers a degree of freedom. For example, where the computation model may allow an arbitrary execution order (e.g., for the arguments to a function in a functional language), a particular order may be chosen in a sequential implementation, or that freedom may be exploited in a parallel implementation.

1.3 PARALLELISM

Programs should provide a potential for *parallel execution* that can be exploited automatically on multiprocessor computer systems, as argued in our initial discussion. Before we continue, here is some additional (informal) terminology that will be used throughout this dissertation:

- *Parallelism* will mean making use of several of the processors on a multiprocessor computer simultaneously, with the aim to make a program run faster, but without otherwise changing its observable behaviour.
- *Concurrency* will mean having components of a program that can proceed independently, with intermittent communication and synchronisation. Concurrency can often be exploited as parallelism.
- *Threading* will mean having concurrent components of a program that are given the opportunity to proceed at a certain well-defined pace, e.g., time-shared processes in an operating system. Threads can be used to satisfy real-time requirements.

Parallelism pertains to the implementation of a language, whereas concurrency and threading pertain to its definition.

The execution of a concurrent program becomes *nondeterministic* in that different components may make their move first, leading to a “random” choice of the next computation state. Such nondeterminism will usually not be visible in sequential implementations, but may be in parallel implementations due to the varying speed of the asynchronously working processors. Randomness is clearly in conflict with predictability. However, if the difference between alternative computation states is always such that they can “catch up” with one another in equivalent future computation states, then the choice doesn't matter. The same job will be done. Formally, this property is known as *confluence.*

Ideally, a concurrent language should have a confluent computation model, but there are reasons why this may not be possible. For example, in most many-to-one communication situations, messages have to be received from senders in an order that cannot be determined in advance. However, it is possible to localise such nondeterminism, which will be called *don't care nondeterminism*, to a few constructs in the language. If they are not used in a program, confluence should be guaranteed.

1.4 INTEROPERABILITY

A programming language and its implementation must be able to coexist and interact with their environment: the user, the hardware, the operating system, existing applications, and other computer systems. This property is often referred to as *interoperability.*

Interoperability can and should influence the language design process. Many new languages fail to reach a wider audience because of poor support of interoperability in the language and its programming system.

A program communicates with the user, other computers, and other programs through devices, secondary storage, shared memory, and process communication. A program communicates with subprograms written in other languages through procedure calls, potentially with updating of shared data, and message passing to objects. Such communication should fit into the computational framework of the new language, without ad hoc extensions, and without too roundabout modes of expression.

Many languages, typically found among the so called declarative languages in the functional and logical paradigms, lack proper models of state, thus crippling their interoperability with conventional languages, which rely on assignment to update data structures in a state.

1.5 PROGRAMMING PARADIGMS

Finally, let us attempt to characterise, very briefly, a number of basic language types that also stand for corresponding *programming paradigms.*

- An *imperative* language has the ability to mutate data objects. Given an arbitrary imperative language, its computation model may be understood as a transition relation, but no other reading is guaranteed to be possible.

- An *object-based* language associates operations with the types of data objects. In a *procedure-based* language, the definition of a generic operation has knowledge of all the data types to which it is applicable. In an object-based language, the operation may be defined for each data type separately.
- An *object-oriented* language is object-based. It also allows new data types to be defined based on existing types in such a way that all or selected parts of the operations on the existing type are *inherited* by the new type.
- A *functional* language allows programs to be read as function definitions, and the execution of a program as the evaluation of an applicative expression.
- A *logical* language allows programs to be read as defining predicates, and the execution of a program as finding a proof for a given statement.
- A *process-based* language provides notions of processes and message-passing between processes.

There is no conflict between these language types. Indeed, a language may be imperative, object-based, functional, logical, and process-based. Of course, a language representing a paradigm may sometimes be *pure* in that it will only admit constructs with the characteristic properties, e.g., a pure functional language will not admit operations that mutate data objects. However, single-paradigm languages are often crippled in their ability to interoperate with a multiparadigm environment.

Many (or most) languages provide elements of several paradigms, sometimes for efficiency, sometimes for expressiveness, and sometimes for all sorts of reasons. The problem is that the result may no longer be simple.

This dissertation describes a language that supports all the above paradigms, while still being surprisingly simple.

CHAPTER 2

LANGUAGE OVERVIEW

Although this is not an AKL textbook, an attempt has been made to provide a language introduction which is as readable as possible. Probably, the results in later chapters will be better appreciated given a firm basic understanding of AKL. Chapter 2, Language Overview, and Chapter 3, Programming Paradigms, form a self-contained informal introduction to AKL and its various aspects. A formal computation model is presented in Chapter 4.

2.1 THE DESIGN

The AGENTS Kernel Language[1], AKL, was conceived with *knowledge information processing* applications in mind, which motivates its particular compromise between simplicity, expressiveness, and efficiency. In addition, with a view to the next generation of multiprocessor computers, AKL provides a large potential for *parallel execution* that can be exploited automatically.

Among the most promising programming languages for knowledge information processing are the *logic programming* languages.

- Prolog and the related constraint logic programming languages have been commercially available for a number of years, and have been used, to great advantage, in a number of advanced industrial applications.
- The concurrent logic programming languages, e.g., Concurrent Prolog, PARLOG, GHC, and Strand, the last of which is commercially available, were designed to allow easy exploitation of multiprocessor computers.

However, both have shortcomings: Prolog lacks the expressiveness of the process-oriented framework, has no appropriate model of state and change, and has many properties that make parallelisation of programs more difficult than

[1] AKL was previously an acronym for the Andorra Kernel Language, but is now regarded as the kernel language of the AGENTS programming system.

necessary. The concurrent logic programming languages lack the expressiveness of don't know nondeterminism, which is useful for many knowledge information processing applications.

AKL integrates, in a coherent and uniform way, the don't know nondeterministic capabilities of Prolog with the process describing capabilities of languages such as GHC, thus remedying the shortcomings of both.

The rest of this chapter is an introduction to AKL.

First, the basic concepts of concurrent constraint programming are presented. Then, based on these concepts, the elements of AKL are introduced step by step, with examples illustrating the additional expressive power provided by each of the language constructs.

As a necessary complement to this fairly informal introduction, Chapter 4 provides a formal definition of the language and its computation model.

2.2 CONCURRENT CONSTRAINT PROGRAMMING

AKL is based on the concept of *concurrent constraint programming*, a paradigm distinguished by its elegant notions of communication and synchronisation based on constraints [Saraswat 1989].

In a concurrent constraint programming language, a computation state consists of a group of *agents* and a *store* that they share. Agents may add pieces of information to the store, an operation called *telling*, and may also wait for the presence in the store of pieces of information, an operation called *asking*.

The information in the store is expressed in terms of *constraints*, which are statements in some constraint language, usually based on first-order logic, e.g.,

$$X < 1,\ Y = Z + X,\ W = [a, b, c],\ \ldots$$

If telling makes a store inconsistent, the computation fails (more on this later). Asking a constraint means waiting until the asked constraint either is *entailed* by (follows logically from) the information accumulated in the store or is *disentailed* by (the negation follows logically from) the same information. In other words, no action is taken until it has been established that the asked constraint is true or false. For example, $X < 1$ is obviously entailed by $X = 0$ and disentailed by $X = 1$.

Constraints restrict the range of possible values of *variables* that are shared between agents. A variable may be thought of as a container. Whereas variables in conventional languages hold single values, variables in concurrent constraint programming languages may be thought of as holding the (possibly infinite) set of values consistent with the constraints currently in the store. This *extensional* view may be complemented by an *intensional* view, in which each variable is thought of as holding the constraints which restrict it. This latter view is often more useful as a mental model.

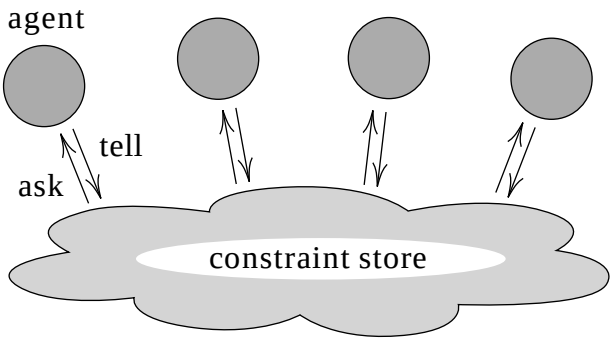


Figure 2.1. Agents interacting with a constraint store

The range of constraints that may be used in a program is defined by the current *constraint theory*, which in AKL, in principle, may be any first-order theory. In practice, it is necessary to ensure that the telling and asking operations used are computable and have a reasonable computational complexity. Constraint theories as such are not investigated in this dissertation. For the purpose of this dissertation, we will use a simple constraint theory with a few obvious constraints, which is essentially that of Prolog and GHC with arithmetic added.

Thus, *constraints* in AKL will be formulas of the form

⟨expression⟩ = ⟨expression⟩

⟨expression⟩ ≠ ⟨expression⟩

⟨expression⟩ < ⟨expression⟩

and the like. Equality constraints, e.g., X = 1, are often called *bindings*, suggesting that the variable X is ”bound” to 1 by the constraint. Correspondingly, the act of telling a binding on a variable is called *binding* the variable.

Expressions are either *variables* (alpha-numeric symbols with an upper case initial letter) , e.g.,

X, Y, Z, X_1, Y_1, Z_1, …

or *numbers*, e.g.,

1, 3.1415, -42, …

or *arithmetic expressions*, e.g.,

1 + X, -Y, X * Y, …

or *constants*, e.g.,

a, b, c, …

or *constructor expressions* of the form

⟨name⟩(⟨expression⟩, …, ⟨expression⟩)

where ⟨name⟩ is an alpha-numeric symbol with a lower case initial letter, e.g.,

s(s(0)), tree(X, L, R), …

There is also the constant [], which denotes the *empty list*, and the *list constructor* [⟨expression⟩ | ⟨expression⟩]. A syntactic convention used in the following is that, e.g., the expression [a | [b | [c | d]]] may be written as [a, b, c | d], and the expression [a | [b | [c | []]]] may be written as [a, b, c].

In addition we assume that constraints "true" and "false" are available, which are independent of the constraint system and may be identified with their corresponding logical constants.

Clearly, asking and telling numerical constraints can be computationally demanding (even undecidable). In this dissertation, only naïve capabilities are assumed, such as deriving X = 3 from X = Y + 1 and Y = 2 by simple propagation of values.

2.3 BASIC CONCEPTS

The agents of concurrent constraint programming correspond to *statements* being executed concurrently.

Constraints, as described in the previous section, are atomic statements known as *constraint atoms* (or just *constraints*). When they are asked and when they are told is discussed in the following.

A *program atom* of the form

⟨name⟩(X_1, …, X_n)

is a defined agent. In a program atom, ⟨name⟩ is an alpha-numeric symbol and *n* is the *arity* of the atom. The variables X_1, …, X_n are the *actual parameters* of the atom. Occurrences of program atoms in programs are sometimes referred to as *calls*. Atoms of the above form may be referred to as ⟨name⟩ / *n* atoms, e.g.,

plus(X, Y, Z)

is a plus / 3 atom. Occasionally, when no ambiguity can arise, " / *n*" is dropped.

The behaviour of atoms is given by (*agent*) *definitions* of the form

⟨name⟩(X_1, …, X_n) := ⟨statement⟩.

The variables X_1, …, X_n must be different and are called *formal parameters*. During execution, any atom matching the left hand side will be replaced by the statement on the right hand side, with actual parameters replacing occurrences of the formal parameters. A definition of the above form is said to define the ⟨name⟩ / *n* atom, e.g.,

plus(X, Y, Z) := Z = X + Y.

is a definition of plus / 3.

A *composition* statement of the form

⟨statement⟩, …, ⟨statement⟩

builds a composite agent from a set of agents. Its behaviour is to replace itself with the concurrently executing agents corresponding to its components.

A *conditional choice* statement of the form

(⟨statement⟩ → ⟨statement⟩ ; ⟨statement⟩)

is used to express conditional execution. Let us call its components *condition*, *then-branch*, and *else-branch*, respectively. (Later a more general version of this statement will be introduced.)

Let us, for simplicity, assume that the condition is a constraint. A conditional choice statement will *ask* the constraint in the condition from the store. If it is entailed, the then-branch replaces the statement. If it is disentailed, the else-branch replaces the statement. If neither, the statement will wait until either becomes known. If the condition is an arbitrary statement, the above described actions will take place when the condition has been reduced to a constraint or when it fails. The concept of failure is discussed in Section 2.5 and arbitrary statements as conditions in Section 2.6.

A *hiding* statement of the form

X_1, …, X_n : ⟨statement⟩

introduces variables with local scope. The behaviour of a hiding statement is to replace itself with its component statement, in which the variables X_1, …, X_n have been replaced by new variables.

Let us at this point establish some syntactic conventions.

- Composition binds tighter than hiding, e.g.,

  X : p, q, r

  means

  X : (p, q, r)

  Parentheses may be used to override this default, e.g.,

  (X : p), q, r

- Any variable occurring free in a definition (i.e., not as one of the formal parameters, nor introduced by a hiding statement) is implicitly governed by a hiding statement enclosing the right hand side of the definition, e.g.,

  p(X, Y) := q(X, Z), r(Z, Y).

  where Z occurs free, means

  p(X, Y) := Z : q(X, Z), r(Z, Y).

  in which hiding has been made explicit.

- Expressions may be used as arguments to program atoms, and will then correspond to bindings on the actual parameters, e.g.,

  p(X+1, [a, b, c])

  means

  Y, Z : Y = X+1, Z = [a, b, c], p(Y, Z)

where the new arguments have also been made local by hiding.

These syntactic conventions are always assumed, unless explicitly suppressed, for example in formal manipulations of the language.

It is now time for a first small example: an append/3 agent, which can be used to concatenate two lists.

append(X, Y, Z) :=
(X = [] → Z = Y
; X = [E | X_1], append(X_1, Y, Z_1), Z = [E | Z_1]).

It will initially suffice to think about constraints in two different ways, depending on the context in which they occur. When occurring as conditions, constraints are *asked*. Elsewhere, they are *told*.

In append/3, the condition X = [] is asked, which means that it may be read "as usual". If it is entailed, the then-branch is chosen, in which Z = Y is told. If the condition is disentailed, the else-branch is chosen. There, X = [E | X_1] is told. Since X is not [], it is assumed that it is a list constructor, in which E is equal to the head of X and X_1 equal to the tail of X. The recursive append call makes Z_1 the concatenation of X_1 and Y. The final constraint Z = [E | Z_1] builds the output Z from E and the partial result Z_1.

Note how variables allow us to work with incomplete data. In a call

append([1, 2, 3], Y, Z)

the parameters Z and Y can be left unconstrained. The third parameter Z may still be computed as [1, 2, 3 | Y], where the tail Y is unconstrained. If Y is later constrained by, e.g., Y = [], then it is also the case that Z = [1, 2, 3].

Variables are also indirectly the means of communication and synchronisation. If a constraint on a variable is asked, the corresponding agent, e.g., conditional choice statement, is suspended and may be restarted whenever an appropriate constraint is told on the variable by another agent. For example,

append([1 | W], Y, Z)

may be rewritten to

([1 | W] = [] → Z = Y
; [1 | W] = [E | X_1], append(X_1, Y, Z_1), Z = [E | Z_1])

by unfolding the append atom, then to

append(W, Y, Z_1), Z = [1 | Z_1]

by executing the choice statement, and substituting values for variables according to the told equality constraint, then to

(W = [] → Z_1 = Y
; W = [E | X_1], append(X_1, Y, Z_2), Z_1 = [E | Z_2]),
Z = [1 | Z_1]

by unfolding the append atom. At this point, the computation suspends. If another agent tells a constraint on W, e.g., W = [2, 3], the computation may be resumed and the final value Z = [1, 2, 3] can be computed. Thanks to the implicit ask synchronisation in conditional choice, the final result of a call to append does not depend on the order in which different agents are processed.

At this point it seems appropriate to illustrate the nature of concurrent computation in AKL. The following definitions will create a list of numbers, and add together a list of numbers, respectively.

```
list(N, L) :=
    ( N = 0 → L = []
    ; L = [N | L1], list(N - 1, L1) ).

sum(L, N) :=
    ( L = [] → N = 0
    ; L = [M | L1], sum(L1, N1), N = N1 + M ).
```

The following computation is possible. In the examples, computations will be shown by performing rewriting steps on the state (or configuration) at hand, unfolding definitions and substituting values for variables, etc., where appropriate, which should be intuitive. In this example we avoid details by showing only the relevant atoms and the collection of constraints on the output variable N. Intermediate computation steps are skipped. Thus,

list(3, L), sum(L, N)

is rewritten to

list(2, L_1), sum([3 | L_1], N)

by unfolding the list atom, executing the choice statement, and substituting values for variables according to equality constraints. This result may in its turn be rewritten to

list(1, L_2), sum([2 | L_2], N_1), N = 3 + N_1

by similar manipulations of the list and sum atoms. Further possible states are

list(0, L_3), sum([1 | L_3], N_2), N = 5 + N_2

sum([], N_3), N = 6 + N_3

N = 6

with final state N = 6.

The list / 2 call produces a list, and the sum / 2 call is there to consume its parts as soon as they are created. The logical variable allows the sum / 2 call to know when data has arrived. If the tail of the list being consumed by the sum / 2 call is unconstrained, the sum / 2 call will wait for it to be produced (in this case by the list / 2 call).

The simple set of constructs introduced so far is a fairly complete programming language in itself, quite comparable in expressive power to, e.g., functional programming languages. If we were merely looking for Turing completeness,

the language could be restricted, and the constraint systems could be weakened considerably. But then important aspects such as concurrency, modularity, and, of course, expressiveness would all be sacrificed on the altar of simplicity.

In the following sections, we will introduce constructs that address the specific needs of important programming paradigms, such as processes and process communication, object-oriented programming, relational programming, and constraint satisfaction. In particular, we will need the ability to choose between alternative computations in a manner more flexible than that provided by conditional choice.

2.4 DON'T CARE NONDETERMINISM

In concurrent programming, processes should be able to react to incoming communication from different sources. In constraint programming, constraint propagating agents should be able to react to different conditions. Both of these cases can be expressed as a number of possibly non-exclusive conditions with corresponding branches. If one condition is satisfied, its branch is chosen.

For this, AKL provides the *committed choice* statement

(⟨statement⟩ | ⟨statement⟩
; ⋯
; ⟨statement⟩ | ⟨statement⟩)

The symbol "|" is read *commit*. The statement preceding commit is called a *guard* and the statement following it is called a *body*. A pair

⟨statement⟩ | ⟨statement⟩

is called a (*guarded*) *clause*, and may be enclosed in hiding as follows.

X_1, …, X_n : ⟨statement⟩ | ⟨statement⟩

The variables X_1, …, X_n are called *local variables* of the clause.

Let us first, for simplicity, assume that the guards are all constraints. The committed-choice statement will *ask* all guards from the store. If any of the guards is entailed, it and its corresponding body replace the committed-choice statement. If a guard is disentailed, its corresponding clause is deleted. If all clauses are deleted, the committed choice statement fails. Otherwise, it will wait. Thus, it may select an arbitrary entailed guard, and commit the computation to its corresponding body.

If a variable Y is hidden, an asked constraint is preceded by the expression "for some Y" (or logically, "∃Y"). For example, in

X = f(a), (Y : X = f(Y) | q(Y))

the asked constraint is ∃Y(X = f(Y)) ("for some Y, X = f(Y)"), which is entailed, since there exists a Y (namely "a") such that X = f(Y) is entailed.

List merging may now be expressed as follows, as an example of an agent receiving input from two different sources.

```
merge(X, Y, Z) :=
    ( X = [] | Z = Y
    ; Y = [] | Z = X
    ; E, X1 : X = [E | X1] | Z = [E | Z1], merge(X1, Y, Z1)
    ; E, Y1 : Y = [E | Y1] | Z = [E | Z1], merge(X, Y1, Z1) ).
```

A merge agent can react as soon as either X or Y is given a value. In the last two guarded statements, hiding introduces variables that are used for "matching" in the guard, as discussed above. These variables are constrained to be equal to the corresponding list components.

2.5 DON'T KNOW NONDETERMINISM

Many problems, especially frequent in the field of Artificial Intelligence, and also found elsewhere, e.g., in operations research, are currently solvable only by resorting to some form of search. Many of these admit very concise solutions if the programming language abstracts away the details of search by providing don't know nondeterminism.

For this, AKL provides the *nondeterminate choice* statement.

```
( ⟨statement⟩ ? ⟨statement⟩
; ⋯
; ⟨statement⟩ ? ⟨statement⟩ )
```

The symbol "?" is read *wait*. The statement is otherwise like the committed choice statement in that its components are called (*guarded*) *clauses*, the components of a clause *guard* and *body*, and a clause may be enclosed in hiding.

Again we assume that the guards are all constraints. The nondeterminate choice statement will also *ask* all guards from the store. If a guard is disentailed, its corresponding clause is deleted. If all clauses are deleted, the choice statement fails. If only one clause remains, the choice statement is said to be *determinate*. Then the remaining guard and its corresponding body replace the choice statement. Otherwise, if there is more than one clause left, the choice statement will wait. Subsequent telling of other agents may make it determinate. If eventually a state is reached in which no other computation step is possible, each of the remaining clauses may be tried in different *copies* of the state. The alternative computation paths may be explored concurrently.

Let us first consider a very simple example, an agent that accepts either of the constants a or b, and then does nothing.

```
p(X) :=
    ( X = a ? true
    ; X = b ? true ).
```

The interesting thing happens when the agent p is called with an unconstrained variable as an argument. That is, we expect it to produce output. Let us call p together with an agent q examining the output of p.

```
q(X, Y) :=
    ( X = a → Y = 1
    ; Y = 0 ).
```

Then the following is one possible computation starting from

p(X), q(X, Y)

First p and q are both unfolded.

(X = a ? true ; X = b ? true), (X = a → Y = 1 ; Y = 0)

At this point in the computation, the nondeterminate choice statement is nondeterminate, and the conditional choice statement cannot establish the truth or falsity of its condition. The computation can now only proceed by trying the clauses of the nondeterminate choice in different copies of the computation state. Thus,

X = a, (X = a → Y = 1 ; Y = 0)

Y = 1

and

X = b, (X = a → Y = 1 ; Y = 0)

Y = 0

are the *two* possible computations. Observe that the nondeterminate alternatives are ordered in the order of the clauses in the nondeterminate choice statement. This ordering will also be used later.

Now, what could possibly be the use of having an agent generate alternative results? This we will try to answer in the following. It will help to think of the alternative results as a sequence of results. Composition of two agents will compute the intersection of the two sequences of results. This will be illustrated using the member agent, which examines membership in a list.

```
member(X, Y) :=
    ( Y1 : Y = [X | Y1] ? true
    ; X1, Y1 : Y = [X1 | Y1] ? member(X, Y1) ).
```

The agent

member(X, [a, b, c])

will establish whether the value of X is in the list [a, b, c]. When the agent is called with an unconstrained X, the different members of the list are returned as different possible results (in the order a, b, c, due to the way the program is written). The composition

member(X, [a, b, c]), member(X, [b, c, d])

will compute the X that are members in both lists. When two nondeterminate choice statements are available, the leftmost is chosen. In this case it will enumerate members of the first list, creating three alternative states

X = a, member(X, [b, c, d])

X = b, member(X, [b, c, d])

X = c, member(X, [b, c, d])

The members in the first list that are not members in the second are eliminated by the *failure* of the corresponding alternative computations. A computation that fails leaves no trace in the sequence of results, and the two final alternative states will be

X = b

X = c

In fact, the sequence of results may become empty, as in the case of the following composition

member(X, [a, b, c]), member(X, [d, e, f])

Such complete failure is also useful, as discussed below.

2.6 GENERAL STATEMENTS IN GUARDS

Although we have ignored it up to this point, any statement may be used as a guard in a choice statement. The behaviour presented above has been that of the special case when conditions and guards are constraints. Such guards are often referred to as *flat*. This will now be generalised to *deep* guards, which contain arbitrary statements.

Before we proceed, we introduce the general *conditional choice* statement.

(⟨statement⟩ → ⟨statement⟩
; ⋯
; ⟨statement⟩ → ⟨statement⟩)

The symbol "→" is read *then*. Again, the statement is otherwise like the other choice statements in that its components are called (*guarded*) *clauses*, the components of a clause *guard* and *body*, and a clause may be enclosed in hiding.

The previously introduced version of conditional choice is, of course, merely syntactic sugar for the special case

(⟨statement⟩ → ⟨statement⟩
; true → ⟨statement⟩)

The case where the guard of the last clause is "true" is common enough to warrant general syntactic sugar, thus

(⟨statement⟩ → ⟨statement⟩
; ⋯
; true → ⟨statement⟩)

may always be abbreviated to

```
( ⟨statement⟩ → ⟨statement⟩
; ···
; ⟨statement⟩ )
```

For the last time we make the simplifying assumption that the guards are all constraints. The conditional choice statement asks the first guard. If it is entailed, it and its corresponding body replace the choice statement. If it is disentailed, the clause is deleted, and the next clause is tried. If neither, the statement will wait. These steps are repeated as necessary. If no clauses remain, the conditional choice statement fails.

When a more general statement is used as a guard, it will first be executed *locally* in the guard, reducing itself to a constraint, after which the previously described actions take place. To illustrate this before we descend into the details, let us use append in a guard (a fairly unusual guard though).

```
( append(X, Y, Z) → p(Z)
; true → q(X, Y) )
```

If we supply constraints for X and Y, e.g., X = [1], Y = [2,3], a value will be computed *locally* for Z, and the resulting choice statement is

```
( Z = [1,2,3] → p(Z)
; true → q(X, Y) )
```

with its above described behaviour.

Formally, the computation in the guard is a separate computation, with local agents and its own local constraint store. Constraints told by local agents are placed in the local store, but constraints asked by local agents are asked from the union of the local store and external stores, known as their *environment*. Locally told constraints can thus be observed by local agents, but not by agents external to the guard.

When the local computation terminates successfully, the constraint asked for the guard is the conjunction of constraints in its local constraint store. This coincides with the behaviour in the special case that the guard was a constraint. In fact, the behaviour of a *constraint atom* statement is always to tell its constraint to the current constraint store.

If the local store becomes inconsistent with the union of external stores, the local computation fails. The behaviour is then as if the computation had terminated successfully, its constraint had been asked, and it had been found disentailed by the external stores.

The scope of don't know nondeterminism in a guard is limited to its corresponding clause. New alternative computations for a guard will be introduced as new alternative clauses. This will be illustrated using the following simple nondeterminate agent.

```
or(X, Y) :=
    ( X = 1 ? true
    ; Y = 1 ? true ).
```

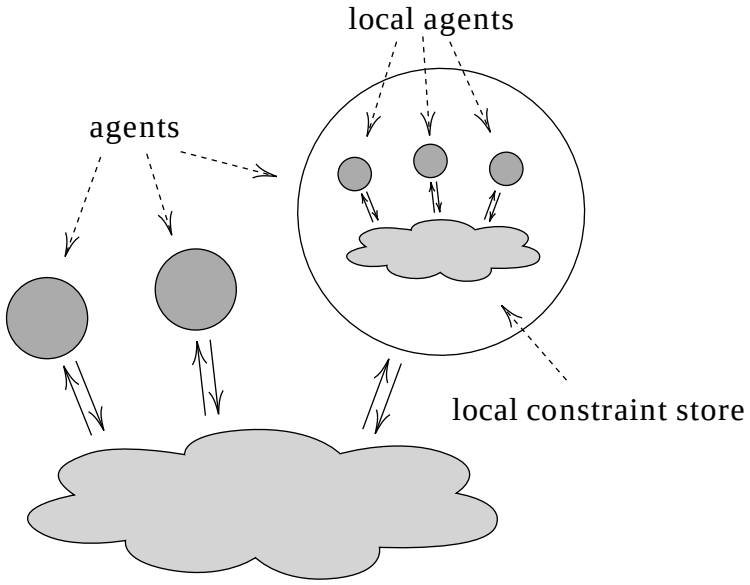


Figure 2.2. An agent with a local constraint store

Let us start with the statement

```
( or(X, Y) | q )
```

The or atom is unfolded, giving

```
( ( X = 1 ? true ; Y = 1 ? true ) | q )
```

Since no other step is possible, we may try the alternatives of the nondeterminate choice in different copies of the closest enclosing clause, which is duplicated as follows.

```
( X = 1 | q
; Y = 1 | q )
```

Other choice statements are handled analogously.

Before leaving the subject of don't know nondeterminism in guards, it should be clarified when alternatives may be tried. (A formal definition is given in Chapter 4.) A local store with associated agents is *stable* if no computation step other than copying in nondeterminate choice is possible (i.e., all agents are inactive), and no such computation step can be made possible by adding constraints to external constraint stores (if any). Alternatives may be tried for the leftmost possible nondeterminate choice in a stable state.

By only executing a nondeterminate choice in a stable state, don't know nondeterministic computations will be synchronised in a concurrent setting in a manner not unlike the synchronisation achieved by conditional or committed choice. For example, the agent

```
member(X, Y)
```

will unfold to

(Y_1 : Y = [X | Y_1] ? true
; X_1, Y_1 : Y = [X_1 | Y_1] ? member(X, Y_1))

By adding constraints to the environment of this agent, it is possible to continue execution without copying, e.g., by adding X = 1 and Y = [2 | W]. Thus, while there are active agents in its environment that may potentially tell constraints on Y, the above agent is unstable.

2.7 BAGOF

Finally, we introduce a statement which builds lists from sequences of alternative results. It provides powerful means of interaction between determinate and nondeterminate code. It is similar to the corresponding construct in Prolog, and a generalisation of the list comprehension primitive found in functional languages (e.g., [Hudak and Wadler 1991]).

A *bagof* statement of the form

bagof(⟨variable⟩, ⟨statement⟩, ⟨variable⟩)

builds a list of the sequence of alternative results from its component statement. The different alternatives for the variable in the first argument will be collected as a list in the variable in the last argument. The statement will be executed within the bagof statement in a manner not unlike the execution of a guard. It is required that the alternatives only restrict the given variable or variables introduced in the component statement. Don't know nondeterminism is not propagated outside it.

For example, the composition

member(X, [a, b, c]), member(X, [b, c, d])

has two alternative results X = b and X = c. By wrapping this composition in a bagof statement, collecting different alternatives for X in Y

bagof(X, (member(X, [a, b, c]), member(X, [b, c, d])), Y)

the result becomes

Y = [b, c]

as could be expected.

Relating this to our previous discussion of stability, the member atoms in

bagof(X, (member(X, Z), member(X, W)), Y)

are unstable (after unfolding) if there are active agents in their environment. If the constraints Z = [a | Z_1] and W = [b | W_1] are added, they become stable, and execution may proceed. After a while, a new unstable state is reached. Then, more information can be added, and so on. Thanks to stability, don't know nondeterministic computations are not affected by the order in which different agents are processed.

Bagof exists in two varieties: ordered (the default) and unordered. The don't know nondeterministic alternatives are, as usual, ordered in the order of clauses in the nondeterminate choice. Thus,

((X = a ; X = b) ; (X = c ; X = d))

generates alternatives for X in the order a, b, c, d. So,

bagof(X, ((X = a ; X = b) ; (X = c ; X = d)), Y)

yields Y = [a, b, c, d]. However,

unordered_bagof(X, ((X = a ; X = b) ; (X = c ; X = d)), Y)

ignores this order, and collects an alternative in the list as soon as it is available. Depending on the implementation, this could lead to a different order, e.g., Y = [d, c, b, a]. See Section 7.3.2 for an (unexpected) application of unordered bagof. Normally, ordered bagof is used. See Section 3.6 for a typical application.

2.8 MORE SYNTACTIC SUGAR

Analogously to what is usually done for functional languages, we now introduce syntactic sugar that is convenient when the guards in choice statements consist mainly of pattern matching against the arguments, as is often the case.

A definition of the form

$p(X_1, \ldots, X_n)$:=
(g_1 % b_1
; …
; g_k % b_k).

where % is either →, |, or ?, may be broken up into several *clauses*

$p(X_1, \ldots, X_n)$:- g_1 % b_1.
…
$p(X_1, \ldots, X_n)$:- g_k % b_k.

which together stand for the above definition. Each clause has a *head*.

The main point of this transformation into *clausal* definitions is that the following additional syntactic sugar may be introduced, which will be exemplified below: (1) Free variables are implicitly hidden, but here the hiding statement encloses the right hand side of the clause (i.e., to the right of ":-"), and not the entire definition. (2) Equality constraints on the arguments in the guard part of a clause may be folded back into the *heads* $p(X_1, \ldots, X_n)$ of these clauses. (3) If the remainder of the guard is "true", it may be omitted. (4) If the guard is "true", and the guard operator is wait "?", the guard operator may be omitted. (5) If the guard operator is omitted, and the body is "true", a clause may be abbreviated to a head.

As an example, the definition

```
member(X, Y) :=
    ( Y1 : Y = [X | Y1] ? true
    ; X1, Y1 : Y = [X1 | Y1] ? member(X, Y1) ).
```

may be transformed into clauses

```
member(X, Y) :-
        Y = [X | Y1]
    ?   true.
member(X, Y) :-
        Y = [X1 | Y1]
    ?   member(X, Y1).
```

where hiding is implicit according to (1). The equality constraints may then be folded back into the head according to (2), and the remaining null guards may be omitted in accordance with (3), giving

```
member(X, [X | Y1]) :-
    ?   true.
member(X, [X1 | Y1]) :-
    ?   member(X, Y1).
```

which may be further abbreviated to

```
member(X, [X | Y1]).
member(X, [X1 | Y1]) :-
        member(X, Y1).
```

according to (4) and (5).

We exemplify also with the append and merge definitions.

```
append([], Y, Z) :-
    → Y = Z.
append(X, Y, X) :-
    → X = [E | X1],
        Z = [E | Z1],
        append(X1, Y, Z1).
```

```
merge([], Y, Z) :-
    |   Y = Z.
merge(X, [], Z) :-
    |   X = Z.
merge([E | X], Y, Z) :-
    |   Z = [E | Z1],
        merge(X, Y, Z1).
merge(X, [E | Y], Z) :-
    |   Z = [E | Z1],
        merge(X, Y, Z1).
```

The examples should make it clear that some additional clarity is gained with the clausal syntax, which prevails in the logic programming community.

We end this section with a few additional remarks about the syntax.

As syntactic sugar, the underscore symbol "_" may be used in place of a variable that has a single occurrence in a clause. All occurrences of "_" in a definition denote different variables.

In an implementation of AKL, the character set restricts our syntax. The *then* symbol "→" is there written as "->", and subscripted indices are not possible. For example, append would be written as

```
append([], Y, Z) :-
   –>  Y = Z.
append(X, Y, Z) :-
   –>  X = [E | X1],
       append(X1, Y, Z1),
       Z = [E | Z1].
```

which is a program that can be compiled and run in AGENTS [Janson et al 1994]. However, to make programs as readable as possible, we will continue to use "→" and indices.

2.9 SUMMARY OF STATEMENTS

An informal summary of the different statements introduced follows.

Example	*Name*	*Section*
X = Y	Constraint atom	2.2, 2.3
append(X, Y, Z)	Program atom	2.3
p, q, r	Composition	2.3
X : p(X)	Hiding	2.3
(X = 0 → Y = 1 ; Y = 2)	Conditional choice	2.3, 2.6
(X = a | p(Z) ; Y = a | q(Z))	Committed choice	2.4
(X = 1 ? true ; X = 2 ? true)	Nondeterminate choice	2.5
bagof(X, member(X, [a,b,c]), L)	Bagof	2.7

A formal grammar is found in Chapter 4.

CHAPTER 3

PROGRAMMING PARADIGMS

Following the overview of AKL in the previous chapter, a number of representative examples illustrate the programming paradigms it provides.

3.1 A MULTIPARADIGM LANGUAGE

AKL is a *multiparadigm* programming language, supporting the following paradigms.

- processes and communication,
- object-oriented programming,
- functional and relational programming,
- constraint satisfaction.

These aspects of AKL are cleanly integrated, and provided using a minimum of basic concepts, common to them all. AKL agents will serve as processes, objects, functions, relations, or constraints, depending on the context (Figure 3.1).

Transformational aspects, such as the implicit search used for constraint satisfaction, co-exist harmoniously with *reactive* aspects, such as processes and process communication. Both may be used in the same program, with processes used for modelling the reactive interfaces to the user, external storage, and the outside world, and with a searching, don't know nondeterministic, component behaving as a process, encapsulating the nondeterminism.

As its name suggests, AKL is a programming language *kernel*. Some aspects of a complete programming language, a *user language*, have been omitted, such as type declarations and modules, a standard library, and direct syntactic support for some of the programming paradigms; but the *programming paradigms* and the basic *implementation technology* developed for AKL will carry over to any user language based on AKL.

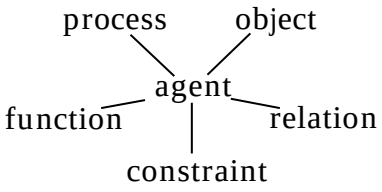


Figure 3.1. Multifaceted AKL agents

In the following sections, we will describe process programming in AKL, object-oriented programming in AKL, functional and relational programming in AKL, and constraint programming in AKL. Finally, it will be shown how these aspects may be integrated in an application.

3.2 PROCESSES AND COMMUNICATION

Agents may be thought of as processes, and telling constraints on shared variables may be thought of as communicating on a shared channel.

The basic principles supporting the idea of communicating processes were discussed in the AKL introduction (Section 2.3). Here we will expand the discussion by explaining many of the concurrent programming idioms. These are inherited from concurrent logic programming [Shapiro 1987; Taylor 1989; Tick 1991].

3.2.1 Communication and Streams

The underlying idea is that a logical variable may be used as a communication channel. On this channel, a message can be sent by a *producer* process by binding the variable to some value.

```
X = a
```

A conditional or a committed-choice statement may be used by a *consumer* process to achieve the effect of waiting for a message. By imposing suitable constraints on the communication variable in their guards, these statements will require the value of the variable to be defined before execution may proceed. Until the value has been produced, the statement will be suspended.

```
( X = a | this
; X = b | that )
```

However, as soon as the variable is constrained, the guard parts of these statements may be executed, and the appropriate action can be taken.

Message arguments can be transferred by binding the variable to a constructor expression.

```
X = f(Y)
```

Likewise, the argument can be received by matching against a constructor expression.

(Y : X = f(Y) | this(Y)
; Y : X = g(Y) | that(Y))

Again, note the scope of the hiding statement. It is limited to each guarded statement. If Y were given a wider scope, the first guard would instead be that the value of X should be equal to X = f(Y), for some given value of Y. The above use has the reading “if there exists a Y such that X = f(Y) …”, and it allows Y to be constrained by the guard.

Contrary to what is the case in the above examples, communication is not restricted to a single message between a producer and a consumer. A message can be given an argument that is the variable on which the next message will be sent. Usually, the list constructor is used for this purpose. The first argument of the list constructor is the message, and the second argument is the new variable. A sequence of messages M_i can be sent as follows.

$X_0 = [M_1 \mid X_1], X_1 = [M_2 \mid X_2], X_2 = [M_3 \mid X_3], \ldots$

The receiver waits for a list constructor, and expects the message to arrive in the first argument, and the variable on which further messages will be sent in the second argument. Observe that the above example is simply the construction of a list of messages. When used to transfer a sequence of messages between processes, a list is referred to as a *stream*. Just like a list, a stream may end with [], which indicates that the stream has been closed, and that no further messages will be sent.

Understood in these terms, the list-sum example in Section 2.3 is a typical producer-consumer example. The list agent produces a stream of messages, each of which is a number, and the sum agent consumes the stream, adding the numbers together.

3.2.2 Basic Stream Techniques

In the previous section, we discussed the notions of producers and consumers. The list-agent is an example of a producer, and the sum-agent is an example of a consumer. Further basic stream techniques are stream transducers, distributors, and mergers.

A stream *transducer* is an agent that takes one stream as input and produces another stream as output. This may involve computing new messages from old, rearranging, deleting, or adding messages. The following is a simple stream transducer computing the square of each incoming message.

```
squares([], Out) :-
    → Out = [].
squares([N | Ns], Out) :-
    → Out = [N*N | Out1],
      squares(Ns, Out1).
```

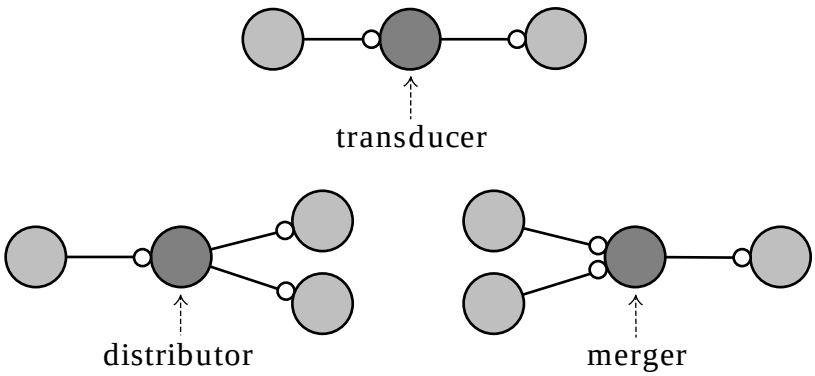


Figure 3.2. Transducer, distributor, and merger

A stream *distributor* is an agent with one input stream and several output streams that directs incoming messages to the appropriate output stream. The following is a simple stream distributor that sends apples to one stream and oranges to the other.

```
fruits([], As, Os) :-
    → As = [],
      Os = [].
fruits([F | Fs], As, Os) :-
      apple(F)
    → As = [F | As1],
      fruits(Fs, As1, Os).
fruits([F | Fs], As, Os) :-
      orange(F)
    → Os = [F | Os1],
      fruits(Fs, As, Os1).
```

A stream *merger* is an agent with several input streams and one output stream that interleaves messages from the input streams into the single output stream. The following is the standard binary stream merger, which was also shown in the language introduction (Section 2.4).

```
merge([], Ys, Zs) :-
    |  Zs = Ys.
merge(Xs, [], Zs) :-
    |  Zs = Xs.
merge([X | Xs], Ys, Zs) :-
    |  Zs = [X | Zs1],
       merge(Xs, Ys, Zs1).
merge(Xs, [Y | Ys], Zs) :-
    |  Zs = [Y | Zs1],
       merge(Xs, Ys, Zs1).
```

Note that all the above definitions can also be seen as simple list-processing agents. However, they are more interesting when one considers their behaviour as components in concurrent programs.

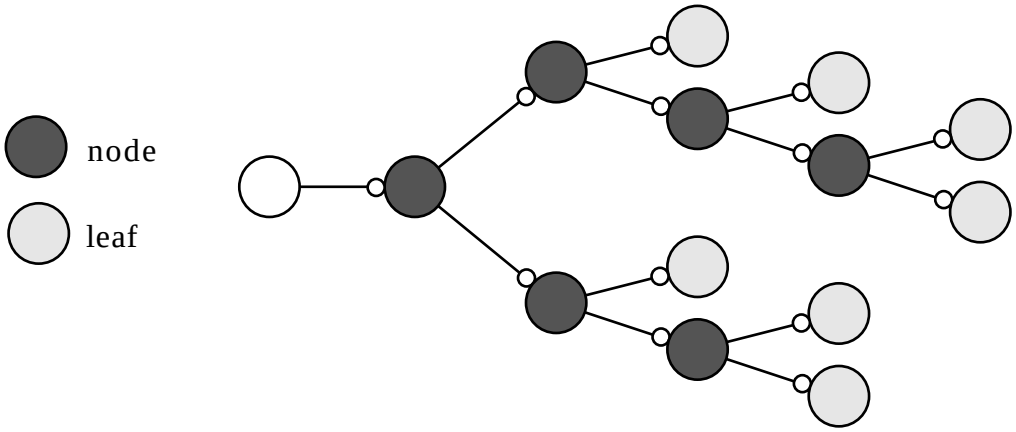


Figure 3.3. Tree of node and leaf processes

3.2.3 Process Structures

Process networks is a technique for storing data, and an example of an object-oriented reading of processes. The technique is best introduced by an example. We will show how a dictionary can be represented as a binary tree of processes.

The tree is built from leaf processes and node processes. A leaf process has one input stream from its parent. A node process has one input stream from its parent and two output streams to its children. In addition, it has two arguments for holding the key and the value of the data item stored in the node. Thus, the processes correspond to equivalent data-structures.

In their default state, these processes are waiting for messages on their input streams. The messages may be of the kind insert(Key, Value), with given key and value that should be inserted, lookup(Key, Result), with a given key and a sought for result (an unconstrained variable), and the closing of the stream which means that the tree should terminate (deallocate itself). Messages that include a variable for the return value are called *incomplete messages*.

The computed result is wrapped in the constructor found(Value), if a value corresponding to a key is found, and is otherwise simply the constant “not_found”.

When a node process receives a request, it compares the key to the key held in its argument, and either takes care of the request itself, or passes the request along to its left or right sub-tree, depending on the result of the comparison. A leaf process always processes a request itself.

```
dict(S) := leaf(S).

leaf([]) :-
    → true.
leaf([insert(K,V) | S]) :-
    → node(S, K, V, L, R),
      leaf(L),
      leaf(R).
leaf([lookup(K,V) | S]) :-
    → V = not_found,
      leaf(S).

node([], _, _, L, R) :-
    → L = [],
      R = [].
node([insert(K1, V1) | S], K, V, L, R) :-
    → (       K1 = K
          →   node(S, K, V1, L, R)
      ;       K1 < K
          →   L = [insert(K1, V1) | L1],
              node(S, K, V, L1, R)
      ;       K1 > K
          →   R = [insert(K1, V1) | R1],
              node(S, K, V, L, R1) ).
node([lookup(K1, V1) | S], K, V, L, R) :-
    → (       K1 = K
          →   V1 = found(V),
              node(S, K, V, L, R)
      ;       K1 < K
          →   L = [lookup(K1, V1) | L1],
              node(S, K, V, L1, R)
      ;       K1 > K
          →   R = [lookup(K1, V1) | R1],
              node(S, K, V, L, R1) ).
```

In the following section on object-oriented programming, we will relate this programming technique to conventional object-oriented programming and its standard terminology.

3.3 OBJECT-ORIENTED PROGRAMMING

In this section, the basic techniques that allow us to do object-oriented programming in AKL are reviewed. Like the programming techniques in the previous section, they are not a contribution of this dissertation, but belong by now to logic programming folklore.

There is more than one way to map the abstract concept of an object onto corresponding concepts in a concurrent constraint language. The first and most widespread of these will be described here in detail [Shapiro and Takeuchi

1983]. It is based on the process reading of logic programs. Several embedded languages have been proposed that support this style of programming (e.g., [Kahn et al. 1987; Yoshida and Chikayama 1988; Davison 1989]). They are typically much less verbose, and they also provide more explicit support for object-oriented concepts. More modern treatments of objects in concurrent logic languages exist (see, e.g., [Smolka, Henz, and Würtz 1993]). This is discussed in Chapter 7.

As will be seen, in this framework there is no real need for an *implementation* of objects, unlike the case when one is adding object-oriented support to a language such as C. Following an object-oriented style of programming is a very natural thing.

As a point of reference, we will adhere to the object-oriented terminology of [Snyder, Hill, and Olthoss 1989]. The meaning of this terminology will be summarised as it is introduced.

3.3.1 Objects

An *object* is an abstract entity that provides *services* to its clients. Clients explicitly *request* services from objects. The request *identifies* the requested service, as well as the object that is to perform the service.

Objects are realised as processes that take as input a stream (a list) of requests. The stream identifies the object. The data associated with the objects are held in the arguments of the process. An object definition typically has one clause per type of request, which performs the corresponding service, and one clause for terminating (or deallocating) the object. Thus, clauses correspond to *methods*. The requests are typically expressions of the form name(A, B, C), where the constructor “name” identifies the request, and A, B, and C are the arguments of the request.

The process description, the agent definition, is the *class*, the implementation of the object. The individual calls to this agent are the *instances*.

A standard example of an object is the bank account, providing withdrawal, deposits, etc.

```
make_bank_account(S) :=
        bank_account(S, 0).

bank_account([], _) :-
    → true.
bank_account([withdraw(A) | R], N) :-
    → bank_account(R, N - A).
bank_account([deposit(A) | R], N) :-
    → bank_account(R, N + A).
bank_account([balance(M) | R], N) :-
    → M = N,
        bank_account(R, N).
```

A computation starting with

make_bank_account(S),
S = [balance(B_1), deposit(7), withdraw(3), balance(B_2)]

yields

$B_1 = 0$, $B_2 = 4$

A bank account object is created by starting a process bank_account(S, 0) which is given as initial input an unspecified stream S (a variable) and a zero balance. The stream S is used to identify the object. A service deposit(5) is requested by binding S to [deposit(5) | S_1]. The next request is added to S_1, and so on.

In the above example, only one clause will match any given request. When it is applied, some computation is performed in its body and a new bank_account process replaces the original one. The requests in the above example are processed as follows. Let us start in the middle.

bank_account(S, 0), S = [deposit(7), withdraw(3), balance(B_2)].

The bank account process is reduced by the clause matching the first deposit-request, leaving some computation to be performed.

N = 0+7, bank_account(S_1, N), S_1 = [withdraw(3), balance(B_2)].

This leaves us with.

bank_account(S, 7), S1 = [withdraw(3), balance(B_2)].

The rest of the requests are processed similarly.

Finally, there are a few things to note about these objects. First, they are automatically *encapsulated*. Clients are prevented from directly accessing the data associated with an object. In imperative languages, this is not as self-evident, as the object is often confused with the storage used to store its internal data, and the object identifier is a pointer to this storage, which may often be used for any purpose.

Second, requests are entirely *generic*. The expression that identifies a request may be interpreted differently, and may therefore involve the execution of different code, depending on the object. This does not involve mandatory declarations in some shared (abstract or virtual) ancestor class, as in many other languages.

Third, *becoming* another type of object is extremely simple. Instead of replacing itself with an object of the same type, an object may pass its stream, and appropriate parameters, on to a new object. An example of this was given in the section on process structures, where a leaf process became a node process when a message was inserted into a binary tree.

3.3.2 Inheritance

In the object-oriented paradigm, objects can be classified in terms of the services they provide. One object may provide a subset of the services of another object. This way an *interface hierarchy* is formed.

It is of course important, from a software engineering point of view, that the descriptions of objects higher up in the hierarchy can be reused as parts of the descendant objects. This is called *implementation inheritance* or *delegation*.

Delegation is easily achieved in the framework we describe. However, since requests are completely generic, it is also possible to design an interface hierarchy without it, if so desired.

Delegation is achieved by creating instances of the ancestor objects. The object identifier of (the stream to) this ancestor object is held as an argument of the derived object. The object corresponding to the ancestor could appropriately be called a *subobject* of the derived object. The derived object filters incoming requests and delegates unknown requests to its subobject.

Delegation is not restricted to unknown requests. We may also define what is elsewhere known as *after-* and *before-methods* by filtering as well. The derived object may perform any action before passing a request on to a subobject.

Let us derive from the bank account class a kind of account that does some form of logging of incoming requests. Let us say that it also adds a get_log service that returns the log. This is easy.

```
make_logging_account(S) :=
        make_bank_account(O),
        make_empty_log(Log),
        logging_account(S, O, Log).

logging_account([get_log(L) | R], O, Log) :-
    → L = Log,
        logging_account(R, O, Log).
logging_account([Req | R], O, Log) :-
    → O = [Req | O1],
        add_to_log(Req, Log, Log1),
        logging_account(R, O1, Log1).
logging_account([], O, _) :-
    → O = [].
```

With delegation, it is cumbersome to handle the notion of *self* correctly. Modern forms of *multiple inheritance*, based on the principle of specialisation, are also difficult to achieve.

Instead, it is quite possible to view inheritance as providing the ability to share common portions of object definitions by placing them in superclasses, which are then implicitly copied into subclass definitions. To exploit this view, syntactic support has to be added to the language, e.g., along the lines of Goldberg,

Silverman, and Shapiro [1992]. This view corresponds closely to that of conventional object-oriented languages.

3.3.3 Ports for Objects

Ports are a special form of constraints, which, when added to AKL, or to any concurrent logic programming language, will solve a number of problems with the approach to object-oriented programming presented above. This section provides a preliminary introduction to ports. They, the problems they solve, and numerous examples of their use, are the topic of Chapter 7.

A *port* is a binary constraint on a bag (a multiset) of messages and a corresponding stream of these messages. It simply states that they contain the same messages, in any order. A bag connected to a stream by a port is usually identified with the port, and is referred to as a port. The open_port(*P*, *S*) operation relates a bag *P* to a stream *S*, and connects them through a port.

The stream *S* will usually be connected to an object. Instead of using the stream to access the object, we will send messages by adding them to the port. The send(*M*, *P*) operation sends a message *M* to a port *P*. To satisfy the port constraint, a message sent to a port will immediately be added to its associated stream, first come first served.

When a port is no longer referenced from other parts of the computation state, when it becomes garbage, it is assumed that it contains no more messages, and its associated stream is automatically closed. When the stream is closed, any object consuming it is thereby notified that there are no more clients requesting its services.

Thus, to summarise: A port is created with an associated stream (to an object). Messages are sent to the port, and appear on the stream in any order. When the port is no longer in use, the stream is closed, and the object may choose to terminate.

A simple example follows.

open_port(P, S), send(a, P), send(b, P)

yields

P = ⟨a port⟩, S = [a,b]

Here we create a port and a related stream, and send two messages. The messages appear in S in the order of the send operations in the composition, but it could just as well have been reversed. The stream is closed when the messages have been sent, since there are no more references to the port.

Ports solve a number of problems that are implicit in the use of streams. The following are the most obvious.

- If several clients are to access the same object, their streams of messages have to be merged into a single input stream. With ports, no merger has to be created. Any client can send a message on the same port.

- If objects are to be embedded in other data structures, creating e.g. an array of objects, streams have to be put in these structures. Such structures cannot be shared, since several messages cannot be sent on the same stream by different clients. However, several messages can be sent on the same port, which means that ports can be embedded.
- With naive binary merging of streams, message sending delay is variable. With ports, message sending delay is constant.
- Objects based on streams require that the streams are closed when the clients stop using them. This is similar to decrementing a reference counter, and has similar problems, besides being unnecessarily explicit and low-level. A port is automatically closed when there are no more potential senders, thus notifying the object consuming messages.

These and other problems and solutions are discussed in Chapter 7.

3.4 FUNCTIONS AND RELATIONS

Functions and relations are simple but powerful mathematical concepts. Many programming languages have been designed so that one of the available interpretations of a procedure definition should be a function or a relation. AKL has well-defined subsets that enjoy such interpretations, and provide the corresponding programming paradigms.

3.4.1 Functions

The functional style of programming is characterised by the *determinate* flow of control and by the *non-cyclic* flow of data. There is no don't care or don't know nondeterminism—a single result is computed. Agents do not communicate bidirectionally—an agent takes input from one agent and produces output to another agent. The latter point is weakened somewhat if the language has a non-strict semantics, in which case “tail-biting” techniques are possible.

Many of the AKL definitions are indeed written in the functional style. For example, the “append”, “squares” and “fruits” definitions in the preceding sections are essentially functional, although the latter two were introduced as components in a process-oriented setting.

The basic relation between functional programs and AKL definitions may be illustrated by an example, written in the non-strict, purely functional language Haskell [Hudak and Wadler 1991]. (The appropriate type declarations are supplied with the functional program for clarity.)

```
data (BinTree a) => (Leaf a) | (Node (BinTree a) (BinTree a))

flatten :: (BinTree a) -> [a] -> [a]

flatten (Leaf x) l = x:l
flatten (Node x y) l = flatten x (flatten y l)
```

In AKL, a corresponding program may be phrased as follows.

```
flatten(leaf(X), L, R) :-
    → R = [X | L].
flatten(node(X, Y), L, R) :-
    → flatten(Y, L, L1),
      flatten(X, L1, R).
```

The main differences are that an explicit argument has to be supplied for the output of the "function", and that nested function applications are unnested, making the output of one call the input of another.

AKL is not a higher-order language, but can provide similar functionality in a simple manner. The technique has been known in logic programming for a long time [Warren 1982; Cheng, van Emden, and Richards 1990]. A term representation is chosen for each definition in a program, and an agent apply is defined, which given such a term applies it to arguments and executes the corresponding definition.

Let a term p(n, t_1, …, t_m) represent a definition p / (n-m), which when applied to n-m arguments t_{m+1}, …, t_n calls p / n with p(t_1, …, t_n).

To give an example relating to the above programs, the term flatten(3) corresponds to the function flatten, and the term flatten(3, Tree) to the function (flatten tree) (where Tree and tree are equivalent trees). A corresponding agent

```
apply(flatten(3), [X,Y,Z]) :-
    → flatten(X, Y, Z).
apply(flatten(3,X), [Y,Z]) :-
    → flatten(X, Y, Z).
apply(flatten(3,X,Y), [Z]) :-
    → flatten(X, Y, Z).
apply(flatten(3,X,Y,Z), []) :-
    → flatten(X, Y, Z).
```

is also defined. In practice, it is convenient to regard apply as being defined implicitly for all definitions in a program, which is also easily achieved.

This functionality may now be used as in functional programs as follows. We define an agent map / 3, which maps a list to another list.

```
map(P, [], Ys) :-
    → Ys = [].
map(P, [X | Xs], Ys0) :-
    → Ys0 = [Y | Ys],
      apply(P, [X, Y]),
      map(P, Xs, Ys).
```

and may then call it with, e.g., map(append(3,[a]), [[b],[c]], Ys) and get the result Ys = [[a,b],[a,c]].

Although by no means necessary, expressions corresponding to lambda expressions can also be introduced. Let an expression

$(X_1, \ldots, X_k) \backslash A$

where A is a statement with free variables $Y_1, \ldots, Y_m$, stand for a term

$p((m+k), Y_1, \ldots, Y_m)$

where $p/(m+k)$ is a new agent defined as

$p(Y_1, \ldots, Y_m, X_1, \ldots, X_k) := A.$

We may now write, e.g., map((X,Y)\append(X, Z, Y), [[b],[c]], Ys) and get the result Ys = [[b | Z],[c | Z]]. Finally, the syntactic gap can be closed even further by introducing the notation

$P(X_1, \ldots, X_k)$

standing for

$apply(P, [X_1, \ldots, X_k])$

Obviously, the terms corresponding to functional closures may be given more efficient representations in an implementation.

3.4.2 Relations

The relational paradigm is known from logic programming as well as from database query languages. Most prominent of logic programming languages is Prolog (see, e.g, [Clocksin and Mellish 1987; Sterling and Shapiro 1986; O'Keefe 1990]), which is entirely based on the relational paradigm. A large number of powerful programming techniques have been developed. Prolog and its derivatives are used for data and knowledge base applications, constraint satisfaction, and general symbolic processing. AKL supports Prolog-style programming. This relation is discussed in Section 8.1.

Characteristic of the relational paradigm is the idea that programs interpreted as defining relations should be capable of answering queries involving these relations. Thus, if a parent relation is defined, the program should be able to produce all parents for given children and all children for given parents, enumerate all parents and corresponding children, and verify given parents and children.

The following definition clearly satisfies this condition.

```
parent(sverker, adam).
parent(kia, adam).
parent(sverker, axel).
parent(kia, axel).
parent(jan_christer, sverker).
parent(hillevi, sverker).
```

It is also satisfied by any non-recursive AKL program that does not use conditional choice, committed choice, or bagof.

Maybe less intuitive, but just as appealing, is the following: a simple parser of a fragment of the English language. The creation of a parse-tree is omitted.

$s(S_0, S) := np(S_0, S_1), vp(S_1, S).$

$np(S_0, S) := article(S_0, S_1), noun(S_1, S).$

article([a | S], S).
article([the | S], S).

noun([dog | S], S).
noun([cat | S], S).

$vp(S_0, S) := intransitive\_verb(S_0, S).$

intransitive_verb([sleeps | S], S).
intransitive_verb([eats | S], S).

The two arguments of each atom represent a string of tokens to be parsed as the difference between the first and the second argument. The following is a sample execution.

$s([a, dog, sleeps], S)$

$np([a, dog, sleeps], S_2), vp(S_2, S)$

$article([a, dog, sleeps], S_1), noun(S_1, S_2), vp(S_2, S)$

$noun([dog, sleeps], S_2), vp(S_2, S)$

$vp([sleeps], S)$

$intransitive\_verb([sleeps], S)$

$S = []$

The relation defined by s is

s([a, dog, sleeps | S], S)	s([a, dog, eats | S], S)
s([a, cat, sleeps | S], S)	s([a, cat, eats | S], S)
s([the, dog, sleeps | S], S)	s([the, dog, eats | S], S)
s([the, cat, sleeps | S], S)	s([the, cat, eats | S], S)

for all S, and will be generated as (don't know nondeterministic) alternative results from

$s(S_0, S)$

The idea of a pair of arguments representing the difference between lists is important enough to warrant syntactic support in Prolog, the DCG syntax, which allows the above definitions to be rendered as follows.

s --> np, vp.

np --> article, noun.

article --> [a].
article --> [the].

and so on.

The example is naive, since real examples would be unwieldy, but the state of the art is well advanced, and the literature on *unification grammars* based on the above simple idea is rich and flourishing.

3.5 CONSTRAINT PROGRAMMING

Many interesting problems in computer science and neighbouring areas can be formulated as *constraint satisfaction problems* (*CSPs*). To these belong, for example, Boolean satisfiability, graph colouring, and a number of logical puzzles (a couple of which will be used as examples). Other, more application oriented, problems can usually be mapped to a standard problem, e.g., register allocation to graph colouring. In general, these problems are NP-complete; any known general algorithm will require exponential time in the worst case. Our task is to write programs that perform well in as many cases as possible.

A CSP can be defined in the following way. A *(finite) constraint satisfaction problem* is given by a sequence of variables $X_1, \ldots, X_n$; a corresponding sequence of (finite) domains of values $D_1, \ldots, D_n$; and a set of *constraints* $c(X_{i_1}, \ldots, X_{i_k})$. A *solution* is an assignment of values to the variables, from their corresponding domains, which satisfies all the constraints.

For our purposes, a constraint can be regarded as a logical formula, where satisfaction corresponds to the usual logical notion, but formalism will not be pressed here. Instead, AKL programs are used to describe CSPs, and their intuitive logical reading provides us with the corresponding constraints. Each agent is regarded as a (user-defined) constraint, and will be referred to as such. These agents are typically don't know nondeterministic, and those assignments for which the composition of these agents does not fail are the solutions of the CSP.

The example to be used in this section is the *n*-queens problem: how to place *n* queens on an *n* by *n* chess board in such a way that no queen threatens another. The problem is very well known, and no new algorithm will be presented. The novelty, compared to solutions in conventional languages, lies in the way the algorithm is expressed. The technique used is due to Saraswat [1987b], and was also used by Bahgat and Gregory [1989].

Each square of the board is a variable V, which takes the value 0 (meaning that there is no queen on the square) or 1 (meaning that there is a queen on the square).

The basic constraint is that there may be *at most one* queen in each row, column, and diagonal. Given that *n* queens are to be placed on an *n* by *n* board, a derived constraint, which we will use, is that there must be *exactly one* queen in each row and column. Note that the exactly-one constraint can be decomposed into an at-least-one and an at-most-one constraint. We now proceed to define these constraints in terms of smaller components. The problem is not only to express the constraint, which is easy, but to express it in such a way that an appropriate level of propagation will occur, which will reduce the search space dramatically.

The *at-most-one* constraint can be expressed in terms of the following agent.

xcell(1, N, N).
xcell(0, _, _).

Note that this agent is determinate (Section 2.5) if the first argument is known, or if the last two arguments are known and different.

For a sequence of squares V_1 to V_k, we can now express that at most one of these squares is 1 using the xcell agent as follows.

xcell(V_1, N, 1),
xcell(V_2, N, 2),
…,
xcell(V_k, N, k)

If more than one V_i is 1, the variable N will be bound to two different numbers, and the constraint will fail. Let us call this constraint at_most_one(V_1, …,V_k), thus avoiding the overhead of having to write a program to create it.

An at_most_one constraint will clearly only have solutions where at most one square is given the value 1, but note also the following propagation effects. If one of the V_i is given the value 1, its associated xcell agent becomes determinate, and can therefore be reduced. When it is reduced, N is given the value i, and the other xcell agents become determinate, and can be reduced, giving their variables the value 0.

The *at-least-one* constraint can be expressed in terms of the following agent.

ycell(1, _, _).
ycell(0, S, S).

Note that this agent too is determinate if the first argument is known, or if the other two arguments are known and different.

For a sequence of squares V_1 to V_k, we can express that at least one of these squares is 1 using the ycell agent as follows.

S_0 = true,
ycell(V_1, S_0, S_1),
ycell(V_2, S_1, S_2),
…,
ycell(V_k, S_{k-1}, S_k),
S_k = false

If all the squares are 0, a chain of equality constraints, $S_0 = S_1$, $S_1 = S_2$, …, will connect the symbols ‘true’ and ‘false’, and the constraint will fail. This constraint we call at_least_one(V_1, …, V_k).

Again note the propagation effects. If a variable is given the value 0, then its associated ycell agent becomes determinate. When it is reduced, its second and third arguments are unified. If all variables but one are 0, the second argument of the remaining ycell agent will be ‘true’ and its third argument will be ‘false’,

and it will therefore be determinate. When it is reduced, its first argument will be given the value 1.

Thus, not only will these constraints avoid the undesirable cases, but they will also detect cases where information can be propagated. When no agent is determinate, and therefore no information can be propagated, alternative assignments for variables will be explored by trying alternatives for the xcell and ycell agents.

A program solving the *n*-queens problem can now be expressed as follows.

- For each column, row, and diagonal, consisting of a sequence of variables V_1, …, V_k, the constraint at_most_one(V_1, …, V_k) has to be satisfied.
- For each column and row, consisting of a sequence of variables V_1, …, V_n, the constraint at_least_one(V_1, …, V_n) has to be satisfied.
- The composition of these constraints is the program.

Note that when information is propagated, this will affect other agents, making them determinate. This will often lead to new propagation. One such case is illustrated in Figure 3.4.

1	0	0	0
0	0	V_{23}	V_{24}
0	V_{32}	0	V_{34}
0	V_{42}	V_{43}	0

Figure 3.4. A state when solving the 4-queens problem

The above grid represents the board, and in each square is written the variable representing it, or its value if it has one. We will now trace the steps leading to the above state. Initially, all variables are unconstrained, and all the constraints have been created. Let us now assume that the topmost leftmost variable (V_{11}) is given the value 1. It appears in the row V_{11} to V_{14}, in the column V_{11} to V_{41}, and in the diagonal V_{11} to V_{44}. Each of these is governed by an at_most_one constraint. By giving one variable the value 1, the others will be assigned the value 0 by propagation.

A second case of propagation is that in Figure 3.5 (next page), where V_{12} and V_{24} are assumed to contain queens, and propagation of the above kind has taken place.

Here we examine the propagation that this state will lead to. Notice that in row 3, all variables but V_{31} have been given the value 0. This triggers the at_least_one constraint governing this row, giving the last variable the value 1,

which in turn gives the variables in the same row, column, or diagonal (only V_{41}) the value 0. Finally, V_{43} is given the value 1 by reasoning as above.

0	1	0	0
0	0	0	1
V_{31}	0	0	0
V_{41}	0	V_{43}	0

Figure 3.5. Another state when solving the 4-queens problem

In comparison, *n*-queens programs written using *finite domain constraints* do not exploit the fact that both rows and columns should contain exactly one queen (e.g., [Van Hentenryck 1989; Carlson, Haridi, and Janson 1994]). They are, however, much faster since propagation is performed by specialised machinery.

A better solution can be obtained if all the xcell and ycell agents are ordered so that those governing variables closer to the centre of the board come before those governing variables further out. If at some step alternatives have to be tried for an agent, values will be guessed for variables at the centre first. This is a good heuristic for the *n*-queens problem.

3.6 INTEGRATION

So far, the different paradigms have been presented one at a time, and it is quite possibly by no means apparent in what relation they stand to each other. In particular the relational and the constraint satisfaction paradigms have no apparent connection to the process paradigm. Here, this apparent dichotomy will be bridged, by showing how a process-oriented application based on the solver for the *n*-queens problems could be structured.

The basic techniques for interaction with the environment (e.g., files and the user) are discussed first, and then an overall program structure is introduced.

3.6.1 Interoperability

The idea underlying interoperability is that an AKL agent sees itself as living in a world of AKL agents. The user, files, other programs, all are viewed as AKL agents. If they have a state, e.g., file contents, they are closer to objects, such as those discussed in Section 3.3. It is up to the AKL implementation to provide this view, which is inherited from the concurrent logic programming languages.

A program takes as parameter a port to the “operating system” agent, which provides further access to the functionality and resources it controls. An inter-

face to foreign procedures adds glue code that provides the necessary synchronisation, and views of mutable data structures as ports to agents.

The details of this form of interoperability have not yet been worked out. The examples use imaginary, although realistic, primitives, as in the following.

```
main(OS) :=
        send(create_window(W, [xwidth=100, xheight=100]), OS),
        send(draw_text(10, 10, 'Hello, world!'), W).
```

Here it is assumed that the agent main is supplied with the "operating system" port OS when called. It provides window creation, an operation that returns a port to the window agent, which provides text drawing, and so on.

For some kinds of interoperability, a consistent view of don't know nondeterminism can be implemented. For example, a subprogram without internal state, such as a numerical library written in C, does not mind if its agents are copied during the course of a computation. For particular purposes, it is even possible to copy windows and similar "internal" objects. But the real world does not support don't know nondeterminism. It would hardly be possible to copy an agent that models the actual physical file system; nor would it be possible to copy an agent that models communication with another computer.

The only solution is to regard this kind of incompleteness as acceptable, and either let attempts to copy such unwieldy agents induce a run-time error, or give statements a "type" which is checked at compile-time, and which shows whether a statement can possibly employ don't know nondeterminism.

3.6.2 Encapsulation

To avoid unwanted interaction between don't know nondeterministic and process-oriented parts of a program, the nondeterministic part can be *encapsulated* in a statement that hides nondeterminism. Nondeterminism is encapsulated in the guard of a conditional or committed choice and in bagof.

When encapsulated in the guard of a conditional or committed choice, a nondeterministic computation will eventually be pruned. In a conditional choice, the first solution is chosen. In a committed choice, any solution may be chosen.

When encapsulated in bagof, all solutions will be collected in a list.

More flexible forms of encapsulation can be based on the notion of *engines*. An engine is conceptually an AKL interpreter. It is situated in a server process. A client may ask the engine to execute programs, and, depending on the form of engine, it may interact with the engine in almost any way conceivable, inspecting and controlling the resulting computation. A full treatment of engines for AKL is future work.

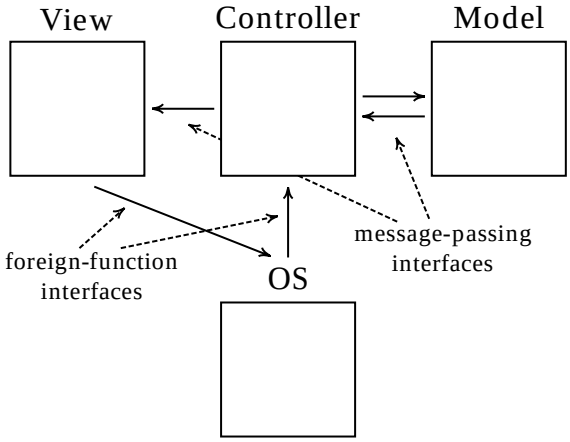


Figure 3.6. The basic program structure

3.6.3 A Program Structure

Responsibilities can be assigned to the components of a program as illustrated by Figure 3.6. The *view* presents output on appropriate devices. The *controller* interprets input and acts as a relay between the model and the view. The *model* is where the actual computation takes place, and this is also where don't know nondeterminism should be encapsulated.

To exemplify the above, we apply it to an *n*-queens application. We assume the existence of a don't know nondeterministic *n*-queens agent

n_queens(N, Q) := … .

which returns different assignments Q to the squares of an N by N chess board. It is easily defined by adding code for creation of constraints for different length sequences, and code for creating sequences of variables corresponding to rows, columns, and diagonals on the chess board. No space will be wasted on this trivial task here. We proceed to the program structure with which to support the application.

```
main(P) :=
        initialise(P, W, E),
        view(V, W),
        controller(E, M, S, V),
        model(M, S).
```

The initialise agent creates a window accepting requests on stream W and delivering events on stream E. The view agent presents whatever it is told to by the controller on the window using stream W. The model delivers solutions on the stream S to the *n*-queens problems submitted on stream M. The controller is driven by the events coming in on E. It submits problems to the model on stream M and receives solutions on stream S. It then sends solutions to the view agent on V for displaying.

Let us here ignore the implementation of the initialise, view, and controller agents. The interesting part is how the don't know nondeterminism is encapsu-

lated in the model agent. We assume that we are satisfied with being able to get either one or all solutions from the particular instance of the *n*-queens problem, or getting the reply that there are no solutions (for N = 2 or N = 3).

```
model([], S) :-
    → S = [].
model([all(N) | M], S) :-
    → bagof(Q, n_queens(N, Q), Sols),
      S = [all(Sols) | S1],
      model(M, S1).
model([one(N) | M], S) :-
    → (       Q : n_queens(N, Q)
          → S = [one(Q) | S1]
      ;       S = [none | S1] ),
      model(M, S1).
```

As described above, don't know nondeterminism is encapsulated in bagof and conditional choice statements.

CHAPTER 4

A COMPUTATION MODEL

Ideally, a computation model should serve a number of purposes:

- It should be a mental model for programmers.
- It should be a useful guide for implementors.
- It should be easy to use for investigating formally the properties of programs and the language.

The AKL computation model serves all of these purposes. It serves well as a mental model for programmers, it is an often useful level of abstraction of the behaviour of an implementation, and it has been used to show results about the language.

The computation model is formalised as a labelled transition system on configurations [Plotkin 1981]. It is defined by straightforward computation rules corresponding to different computation steps, and a structural rule propagating the effect of individual steps to the whole configuration. The major technicality is the introduction and control of don't know nondeterminism.

The model presented here has evolved from earlier models of KAP and AKL [Haridi and Janson 1990; Janson and Haridi 1991; Franzén 1991; Janson and Montelius 1992; Franzén 1994]. The main novelty of the current version is the use of the statement syntax for programs, as opposed to the clausal syntax of previous versions.

4.1 DEFINITIONS AND PROGRAMS

Assume given sets of *variables*, *constraint names*, and *program atom names*. A *constraint atom* is an expression of the form $c(X_1, \ldots, X_n)$, where c is a constraint name and $X_1, \ldots, X_n$ are variables. Similarly, a *program atom* is an expression of the form $p(X_1, \ldots, X_n)$, where p is a program atom name and $X_1, \ldots, X_n$ are different variables. An *atom* is a constraint atom or a program atom. The variables

in an atom are called *parameters*. The number of parameters—determined by *p* or *c*—is called the *arity* of the atom.

The remaining (abstract) syntactic categories pertaining to programs follow.

⟨definition⟩ ::= ⟨head⟩ := ⟨body⟩

⟨head⟩ ::= ⟨program atom⟩

⟨body⟩ ::= ⟨statement⟩

⟨statement⟩ ::= ⟨atom⟩ | ⟨composition⟩ | ⟨hiding⟩ | ⟨choice⟩ | ⟨aggregate⟩

⟨composition⟩ ::= ⟨statement⟩, ⟨statement⟩

⟨hiding⟩ ::= ⟨set of variables⟩ : ⟨statement⟩

⟨choice⟩ ::= ⟨sequence of clauses with the same guard operator⟩

⟨clause⟩ ::= ⟨set of vars⟩ : ⟨statement⟩ ⟨guard operator⟩ ⟨statement⟩

⟨guard operator⟩ ::= '→' | '|' | '?'

⟨aggregate⟩ ::= aggregate(⟨variable⟩, ⟨statement⟩, ⟨variable⟩)

The clauses of a choice statement have the same guard operator. To the guard operators correspond *conditional* choice ('→'), *committed* choice ('|'), and *nondeterminate* choice ('?') statements, respectively. The parameters of a head atom are called *formal parameters.* A variable is *bound* in a statement if all occurrences are in clauses, hiding, or aggregates, with corresponding occurrences in the corresponding hidden sets of variables or the first position of the aggregate. Otherwise, it is *free*. Variables in the body of a definition are either bound or formal parameters . A *program* is a finite set of definitions for different program atom names, satisfying the condition that every program atom name occurring in the program has a definition in the program.

In this chapter, bagof is generalised to the notion of an *aggregate*. The different alternative results for a don't know nondeterministic agent p(X) are grouped as follows. Let $\theta_1, \ldots, \theta_n$ be the results for p(X), in which local variables have been renamed apart, and let $X_1, \ldots, X_n$ be the different renamings of X. Let the operation *unit*(Y) bind Y to the unit of the aggregate (Y = [] in bagof), and let the operation *collect*(X, Y_1, Y) form an aggregate Y of a solution X and another aggregate Y_1 (Y = [X | Y_1] in bagof). Executing the statement

aggregate(X, p(X), Y)

will yield a result of the form

θ_1, *collect*(X_1, Y_1, Y), …, θ_n, *collect*(X_n, Y_n, Y_{n-1}), *unit*(Y_n)

For example,

bagof(X, member(X, [a,b,c]), Y)

yields the result

X_1 = a, Y = [X_1 | Y_1], X_2 = b, Y_1 = [X_2 | Y_2], X_3 = c, Y_2 = [X_3 | Y_3], Y_3 = []

Thus, the aggregate operation is defined by its unit and collect operations. Often, these are constraints, but in the following they may be arbitrary statements. In addition, the aggregate may be regarded as ordered or unordered, the unordered version being preferred if the collect operation is associative and commutative. For example, an unordered aggregate can reliably count the solutions, or add them up if they are numbers. There can also be operational reasons for preferring one or the other.

4.2 CONSTRAINTS

Constraint atoms are regarded as atomic formulas in some constraint language, for simplicity based on first order classical logic. Other logics might be useful, but such generality is not of interest in the present context.

The symbols σ, τ, and θ will be used for conjunctions of constraint atoms, called *constraints*. In the following ∃σ stands for the existential closure of σ, and ∃V for (any permutation of) the quantifier sequence $\exists X_1 \cdots \exists X_n$, where V is a set $\{X_1, \ldots, X_n\}$ of variables. We allow V to be empty, in which case ∃V is mere decoration.

We assume given some constraint theory **TC** defining the following logical properties of constraints:

1. σ is *satisfiable* iff **TC** ⊨ ∃σ
2. σ is *quiet w.r.t.* θ *and* V iff **TC** ⊨ θ ⊃ ∃Vσ, where the variables in V do not occur in θ
3. σ and θ are *incompatible* iff **TC** ⊨ ¬(σ ∧ θ)

A constraint which is not quiet is called *noisy*.

The symbols **true** and **false** denote variable-free atomic formulas for which it holds that **TC** ⊨ **true** and **TC** ⊨ ¬**false**. No further assumptions concerning the properties of **TC** will be made.

For the constraint theory of equalities between *rational trees* [Maher 1988], we can make the following observations. We need only consider constraints of the form **false**, or of the (satisfiable) *substitution* form $v_1 = t_1 \wedge \ldots \wedge v_n = t_n$, where v_i are different variables and t_i are variables not equal to any v_j or constructor expressions of the form $f(u_1, \ldots, u_k)$, where f is a tree constructor of arity k (≥ 0) and u_i are variables, since a constraint can always be reduced to either of these forms by *unification* [Lassez, Maher, and Marriott 1988]. A variable v is *bound* by a substitution if it is the left-hand or right-hand side of an equation. The constraint **true** is regarded as a special case of a substitution, where n is 0. A constraint is satisfiable if it can be reduced to a substitution. A constraint σ is quiet w.r.t. a satisfiable constraint θ in substitution form, and a set of variables V, if the reduced form τ of θ ∧ σ is satisfiable, and all variables bound by τ are either in V, bound to a variable in V, or are bound by θ. Constraints σ and θ are incompatible if θ ∧ σ can be reduced to **false**. See Section 4.4.7 for further discussion of rational tree constraints.

Other constraint theories that we will have reason to mention are those of *finite trees* [Maher 1988], *feature trees* [Aït-Kaci, Podelski, and Smolka 1992], *records* [Smolka and Treinen 1992], and *finite domains* [Van Hentenryck, Saraswat, and Deville 1992].

4.3 GOALS AND CONTEXTS

Goals are expressions built from statements and combinators called boxes. Their syntax is defined as follows.

$$
\begin{aligned}
\langle \text{goal} \rangle &::= \langle \text{global goal} \rangle \mid \langle \text{local goal} \rangle \\
\langle \text{global goal} \rangle &::= \langle \text{or-box} \rangle \mid \langle \text{and-box} \rangle \\
\langle \text{or-box} \rangle &::= \mathbf{or}(\langle \text{sequence of global goals} \rangle) \\
\langle \text{and-box} \rangle &::= \mathbf{and}(\langle \text{sequence of local goals} \rangle)^{\langle \text{constraint} \rangle}_{\langle \text{set of variables} \rangle} \\
\langle \text{local goal} \rangle &::= \langle \text{statement} \rangle \mid \langle \text{choice-box} \rangle \mid \langle \text{aggregate box} \rangle \\
\langle \text{choice-box} \rangle &::= \mathbf{choice}(\langle \text{sequence of guarded goals} \rangle) \\
\langle \text{guarded goal} \rangle &::= \langle \text{global goal} \rangle \, \langle \text{guard operator} \rangle \, \langle \text{statement} \rangle \\
\langle \text{aggregate box} \rangle &::= \mathbf{aggregate}(\langle \text{variable} \rangle, \langle \text{or-box} \rangle, \langle \text{variable} \rangle)
\end{aligned}
$$

The variables in the set of variables associated with an and-box are called the *local variables* of the and-box. The constraint associated with an and-box is called the *local constraint store* of the and-box.

In the following, the letters R, S, and T stand for sequences of goals. Sequences are formed from goals and other sequences using the associative comma operator. (No confusion is expected, even though comma is also used for composition.) Sequences may be empty. The symbol ε stands for the empty sequence, when the need arises to name it explicitly. The letter G stands for a goal, and the letters A and B for statements. The letters *u*, *v*, and *w*, stand for single variables, and the letters U, V, and W stand for sets of variables. Letters are decorated with indices and the like as needed. The symbol '%' stands for a guard operator.

An and-box of the form $\mathbf{and}()^{\sigma}_{V}$ is called *solved* and may be written as σ_V. An or-box of the form **or**() may be written as **fail**.

We will need two more concepts: that of a *context*, which is a goal with a "hole", and that of the *environment* of a context, which are the constraints "visible" from the hole.

We define *contexts* inductively as follows. The symbol λ denotes the hole, and the symbol χ denotes a context.

- λ is a context.
- if χ is a context then $\mathbf{and}(R, \chi, S)^{\sigma}_{V}$, **or**(R, χ, S), **choice**(R, (χ % A), S), and **aggregate**(*u*, χ, *v*) are contexts.

χ[G] denotes the expression obtained by substituting a goal G for λ in χ. χ may be referred to as the context of (this occurrence of) the goal G in χ[G]. Correspondingly, the expression $\chi[\chi']$ denotes the context obtained by substituting a context χ' for λ in χ. When λ occurs in a context χ of the form χ'[**and**(R, λ, S)$^{\sigma}_{V}$] or χ'[**or**(R, λ, S)], χ[] denotes the expression obtained by deleting λ.

The *environment* of a context, env(χ), is the conjunction of all constraints in all and-boxes surrounding the hole of χ.

- env(λ) = **true**
- env(**and**(R, χ, S)$^{\sigma}_{V}$) = env(χ) $\wedge$ σ
- env(**or**(R, χ, S)) = env(χ)
- env(**choice**(R, (χ % A), S)) = env(χ)
- env(**aggregate**(*u*, χ, *v*)) = env(χ)

By the environment of an occurrence of a goal G in another goal χ[G], we mean the environment of χ.

A goal G is a *subgoal* of any goal of the form χ[G].

4.4 GOAL TRANSITIONS

An *AKL goal transition system* w.r.t. a given program is a structure $\langle \Delta, \Rightarrow \rangle$, where Δ is the set of expressions generated by $\langle$goal$\rangle$, and $\Rightarrow \subseteq \Delta \times M \times C \times \Delta$ is a labelled transition relation, where M is the *mode* (one of D or N) and C is the set of contexts.

Read

$$G \underset{\chi}{\overset{m}{\Rightarrow}} G'$$

as saying that there is a transition from G to G' in mode *m* with context χ. The letters D and N stand for *determinate* and *nondeterminate* mode, respectively.

The rest of this section describes the rules defining the transition relation.

First, the *subgoal* rule

$$\frac{G \underset{\chi'[\chi]}{\overset{m}{\Rightarrow}} G'}{\chi[G] \underset{\chi'}{\overset{m}{\Rightarrow}} \chi[G']}$$

derives transitions of goals depending on transitions that can be made by the components of the goals.

4.4.1 Basic Statements

The *constraint atom* rule

$$\textbf{and}(R, A, S)^{\sigma}_{V} \underset{\chi}{\overset{D}{\Rightarrow}} \textbf{and}(R, S)^{A \wedge \sigma}_{V}$$

moves a constraint atom A to the constraint part of the and-box, thereby making it part of the constraint environment of the goals in the box.

The *program atom* rule

$$A \overset{D}{\underset{\chi}{\Rightarrow}} B$$

unfolds a program atom A using its definition A := B, where the arguments of A are substituted for the formal parameters, and bound variables in B are renamed as appropriate to avoid conflicts with the parameters of A.

The *composition* rule

$$\mathbf{and}(R, (A, B), S)^{\sigma}_{V} \overset{D}{\underset{\chi}{\Rightarrow}} \mathbf{and}(R, A, B, S)^{\sigma}_{V}$$

flattens compositions, making their components parts of the enclosing and-box.

The *hiding* rule

$$\mathbf{and}(R, (U : A), S)^{\sigma}_{W} \overset{D}{\underset{\chi}{\Rightarrow}} \mathbf{and}(R, B, S)^{\sigma}_{V \cup W}$$

introduces new variables. The variables of U in A are replaced by the variables in the set V, giving the new statement B. The set V is chosen to be disjoint from W, from all sets of local variables in R, S, and χ, and from the set of variables bound in A.

The *choice* rule

$$(U_1 : A_1 \,\%\, B_1 ; \cdots ; U_n : A_n \,\%\, B_n) \overset{D}{\underset{\chi}{\Rightarrow}} \mathbf{choice}(\mathbf{and}(A'_1)^{\mathbf{true}}_{V_1} \,\%\, B'_1, \ldots, \mathbf{and}(A'_n)^{\mathbf{true}}_{V_n} \,\%\, B'_n)$$

starts local computations in guards. The local variables U_i in A_i and B_i are replaced by the variables in the set V_i, giving new statements A'_i and B'_i. The sets V_i are chosen to be disjoint from each other, from all other sets of local variables in χ, and from the set of variables bound in A_i or B_i.

4.4.2 Promotion

The *promotion* rule

$$\mathbf{and}(R, \mathbf{choice}(\sigma_V \,\%\, B), S)^{\theta}_{W} \overset{D}{\underset{\chi}{\Rightarrow}} \mathbf{and}(R, B, S)^{\sigma \wedge \theta}_{V \cup W}$$

promotes a single remaining guarded goal with a solved guard and-box. If % is '$\rightarrow$' or '|' it is required that σ is quiet with respect to $\theta \wedge \mathrm{env}(\chi)$ and V.

4.4.3 Pruning

The *condition* rule

$$\mathbf{choice}(R, \sigma_V \rightarrow B, S) \overset{D}{\underset{\chi}{\Rightarrow}} \mathbf{choice}(R, \sigma_V \rightarrow B)$$

may be applied if S is non-empty and σ is quiet with respect to $\mathrm{env}(\chi)$ and V.

The *commit* rule

$$\textbf{choice}(R, \sigma_V \mid B, S) \overset{D}{\underset{\chi}{\Rightarrow}} \textbf{choice}(\sigma_V \mid B)$$

may be applied if at least one of R or S is non-empty and σ is quiet with respect to env(χ) and V.

4.4.4 Failure

The *environment failure* rule

$$\textbf{and}(R)^{\sigma}_{V} \overset{D}{\underset{\chi}{\Rightarrow}} \textbf{fail}$$

fails an and-box if σ and env(χ) are incompatible.

The *goal failure* rule

$$\textbf{and}(R, \textbf{choice}(), S)^{\sigma}_{V} \overset{D}{\underset{\chi}{\Rightarrow}} \textbf{fail}$$

fails an and-box if it contains an empty choice-box.

The *guard failure* rule

$$\textbf{choice}(R, \textbf{fail} \ \% \ B, S) \overset{D}{\underset{\chi}{\Rightarrow}} \textbf{choice}(R, S)$$

removes a failed guarded goal.

4.4.5 Nondeterminism

The *choice splitting* rule

$$\textbf{and}(S_1, \textbf{choice}(T_1, T_2), S_2)^{\sigma}_{V} \overset{N}{\underset{\chi}{\Rightarrow}}$$
$$\textbf{or}(\ \textbf{and}(S_1, \textbf{choice}(T_1), S_2)^{\sigma}_{V}\ ,$$
$$\textbf{and}(S_1, \textbf{choice}(T_2), S_2)^{\sigma}_{V}\ )$$

distributes nondeterminism in an inner nondeterminate choice-box over an and-box, creating alternatives in an outer box. The following two conditions must be satisfied.

- T_1 must be a single guarded goal of the form (θ_W ? A) and T_2 must be non-empty. Choice splitting is said to be performed *with respect to* T_1, and in a goal matching the left hand side of the rule, T_1 is called a *candidate* for choice splitting.
- The rewritten and-box must be a subgoal of a *stable* goal. The notion of stability is explained in Section 4.5.

The *guard distribution* rule

$$\textbf{choice}(R, \textbf{or}(G, S) \ \% \ B, T) \overset{D}{\underset{\chi}{\Rightarrow}} \textbf{choice}(R, G \ \% \ B, \textbf{or}(S) \ \% \ B, T)$$

distributes alternatives in a guard over a guarded goal, making the or-branches different guarded goals in the containing choice-box.

4.4.6 Aggregates

The *aggregate* rule

$$\text{aggregate}(u, \text{A}, v) \underset{\chi}{\overset{\text{D}}{\Rightarrow}} \textbf{aggregate}(w, \textbf{or}(\textbf{and}(\text{B})^{\textbf{true}}_{\{w\}}), v)$$

introduces an aggregate-box. The variable *u* in A is replaced with a variable *w*, giving the new statement B. The variable *w* is chosen to not occur in any set of local variables in the context χ, nor in the set of variables bound in A.

In the following, the expressions *unit*(*v*) and *collect*(*u*', *v*', *v*) stand for statements with one and three free variables, respectively. Nothing precludes having at the same time several different types of aggregates with different associated unit and collect statements, but for simplicity we restrict this exposition to one type of aggregate.

We may rewrite aggregate-boxes by the *unit* rule

$$\textbf{aggregate}(u, \textbf{fail}, v) \underset{\chi}{\overset{\text{D}}{\Rightarrow}} \mathit{unit}(v)$$

and by the *collect* rule

$$\textbf{and}(\text{R}_1, \textbf{aggregate}(u, \textbf{or}(\text{S}_1, \sigma_\text{V}, \text{S}_2), v), \text{R}_2)^{\theta}_{\text{W}} \underset{\chi}{\overset{\text{D}}{\Rightarrow}}$$
$$\textbf{and}(\text{R}_1, \textbf{aggregate}(u, \textbf{or}(\text{S}_1, \text{S}_2), v'), \mathit{collect}(u', v', v), \text{R}_2)^{\sigma' \wedge \theta}_{\text{V}' \cup \text{W}}$$

if σ is quiet w.r.t. $\theta \wedge \text{env}(\chi)$ and V. The local variables V in σ are replaced by the variables in the set V′, giving σ'. The set V' is chosen to be disjoint from W and all sets of local variables in R_1, R_2, and χ. In particular, the variable *u* is replaced by *u*'. If the aggregate is ordered it is also required that S_1 is empty.

The *or-flattening* rule

$$\textbf{or}(\text{R}, \textbf{or}(\text{S}), \text{T}) \underset{\chi}{\overset{\text{D}}{\Rightarrow}} \textbf{or}(\text{R}, \text{S}, \text{T})$$

unnests nested or-boxes, making the alternatives available for the collect rule.

4.4.7 Constraint Simplification

Simplification of constraints is not an essential part of the AKL model, but can be expressed as follows. With a constraint theory may be associated *constraint simplification rules* of the general form

$$\textbf{and}(\text{R})^{\sigma}_{\text{U}} \underset{\chi}{\overset{\text{D}}{\Rightarrow}} \textbf{and}(\text{R})^{\theta}_{\text{V}}$$

that replace [illegible] The set V contains all variables in U that occur in θ or R, and new variables that are in θ, but not in R, U or χ. The rules must satisfy the following condition

$$\textbf{TC} \quad \text{env}(\chi) \supset (\exists \text{W}\sigma \equiv \exists \text{W}\theta)$$

where W contains all variables in U and V not occurring in R, and they may not give rise to infinite sequences of simplifying transitions.

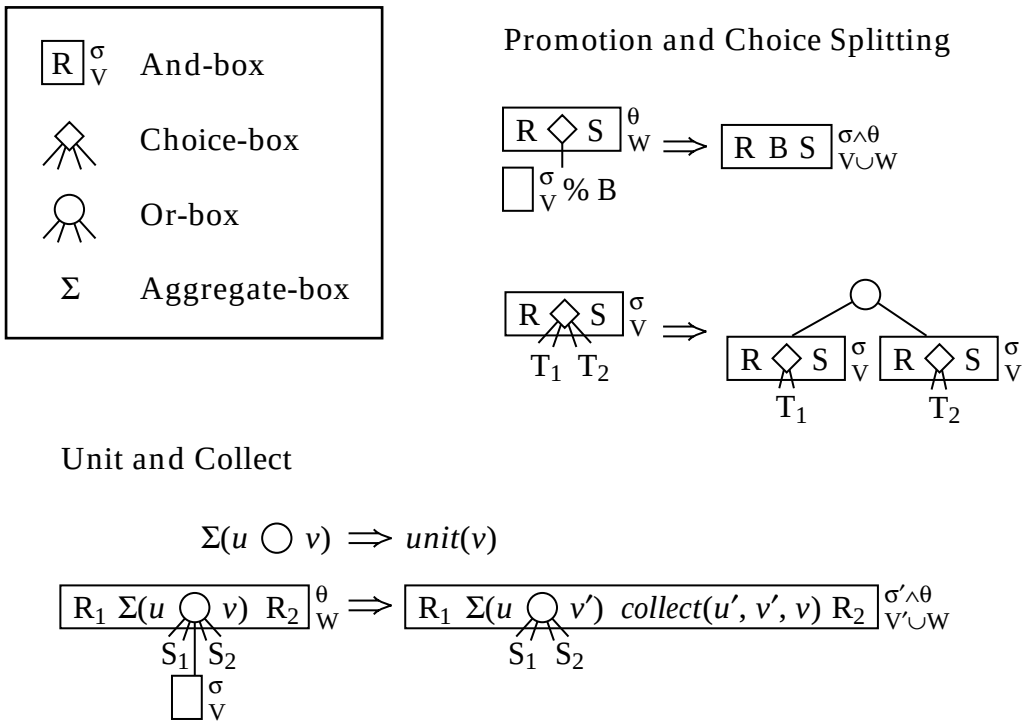


Figure 4.1. Graphical notation for boxes and rules

Simplification rules can be used to justify simplifications performed in an implementation, e.g., during constraint solving and garbage collection.

For the constraint system of rational trees, we have already, informally, introduced simplification by unification. A more precise description follows, for use in Chapter 5.

Simplification is relative, in that it is performed with respect to a given simplified constraint θ [Aït-Kaci, Podelski, and Smolka 1992]. As before, we assume that θ has substitution form, in which all equations of the form $u = v$ together form an acyclic relation over the variables. If we regard this relation over the variables as a partial order, the sets of variables known to be equal have unique maximal elements. Given a variable v, we refer to this as deref(v, θ). Below, u' is deref(u, θ) and v' is deref(v, θ).

We assume that σ∧**true** and **true**∧σ and σ∧**false** and **false**∧σ are always simplified to σ and **false**, respectively. Note below that if a constraint σ is simplified to σ′ relative to θ, then θ∧σ′ has the acyclic substitution form required above.

A constraint of the form $u = \mathrm{f}(v_1, \ldots, v_k)$ is simplified to **false** if u' is bound to a constructor expression with different constructors, to the simplified form of $u_1 = v_1 \wedge \cdots \wedge u_k = v_k$, if it has the same constructor and arguments u_i, and otherwise to $u' = \mathrm{f}(v_1, \ldots, v_k)$.

A constraint $u = v$ is simplified to **true** if u' is equal to v', to **false** if u' and v' are bound to constructor expressions with different constructors, to the simplified

form of $u_1 = v_1 \wedge \cdots \wedge u_k = v_k$ for their arguments u_i and v_i, if they have the same constructor, to $u' = v'$ if u' is not bound, and otherwise to $v' = u'$.

A constraint $\sigma_1 \wedge \sigma_2$ is simplified to $\sigma_1' \wedge \sigma_2'$, where σ_1 is simplified to σ_1' relative to θ and σ_2 is simplified to σ_2' relative to $\theta \wedge \sigma_1'$.

The above can be used for a rule that simplifies a constraint that has just been added by the constraint atom rule (relative to the conjunction of the remaining local store and the environment), and for a rule that simplifies an entire local store when constraints have been added to stores in its environment.

Simplification corresponding to garbage collection involves partitioning local stores σ in substitution form into two parts σ_1 and σ_2, where σ_2 binds variables in W that do not occur in σ_1. Clearly, $\exists W\sigma$ is equivalent to $\exists W\sigma_1$, and σ_2 can be garbage collected.

4.5 NONDETERMINISM AND STABILITY

Choice splitting potentially duplicates work. Therefore, it is delayed until it is necessary for the computation to proceed. Let us refer to the and-box which is to be rewritten by choice splitting as G_N and to its context as χ_N. Conditions that formalise the above "necessity" will be placed on a goal G and its context χ, for which

$$\chi_N = \chi[\chi'] \quad \text{and} \quad G = \chi'[G_N]$$

A first condition is to require that no determinate transitions are possible on G in χ. We say that a goal G is *quiescent in* χ if

$$\neg\exists G'(G \underset{\chi}{\overset{D}{\Rightarrow}} G')$$

However, future transitions in the context of a goal may produce constraints that either entail or are incompatible with constraints within the goal. Therefore, not only should it be required that no determinate transitions are applicable, but also that determinate transitions cannot become applicable as a result of further transitions in the context of the goal. Unfortunately, as stated, this condition is undecidable, and cannot be used for programming purposes.

Instead, we say that a goal is *stable* in a given context if it is quiescent, and remains so even if arbitrary constraints compatible with the environment, are added to the and-boxes in the context of the goal. We thereby avoid reasoning about the constraints that may be produced by future computations, and need only consider the relationship between [illegible] and its environment. It still depends on the constraint theory whether this condition is computationally tractable, but it is well-behaved for rational trees.

Formally, a goal G is stable in context χ if it is quiescent and

$$\forall\chi'\forall\tau\,(G{=}\chi'[] \supset [\mathbf{TC}\ \text{env}(\chi) \wedge \tau \supset \exists V\ \text{env}(\chi')])$$

$$\forall\chi'\forall\tau\,\forall\sigma\forall U\,(G{=}\chi'[\sigma] \supset [\mathbf{TC}\ \text{env}(\chi) \wedge \tau \supset \exists V\,(\text{env}(\chi') \wedge \neg\exists U\sigma)])$$

where V is the set of variables local to and-boxes in G.

Note that the notation χ[include solved and-boxes. The first condition states that environment failure will not become applicable. The second condition states that no solved and-box will become quiet. This is stronger than required, by only considering solved and-boxes for which quietness can cause further transitions. Even stronger, but probably more useful, conditions are

$$\forall\chi' \, (G{=}\chi'[] \supset [\mathbf{TC} \; \mathrm{env}(\chi) \supset \exists V \; \mathrm{env}(\chi')])$$

$$\forall\chi'\forall\sigma\forall U \, (G{=}\chi'[\sigma_U] \supset [\mathbf{TC} \; \mathrm{env}(\chi) \supset \exists V \, (\mathrm{env}(\chi') \wedge \neg\exists U\sigma)])$$

For the constraint theory of rational trees, the first of these conditions implies the second and is also equivalent with the above formulation [Franzén 1994]. If relative simplification is performed as suggested, a "constructive" formulation of this condition is that G should not contain equations of the form $v = f(\ldots)$ or $u = v$, where u and v are not in V.

Additional conditions may, and should, be imposed on the choice splitting rule, for example that it must be applied with respect to the leftmost candidate in an innermost stable box. However, it is not the intention that AKL should be based on a single such rule. Instead, it should be chosen for the given program. The topic of how to make this selection (e.g., using the concept of engines) is outside the scope of this dissertation, where we assume that a *leftmost* candidate be chosen.

4.6 CONFIGURATIONS AND COMPUTATIONS

Two sets of local variables in a goal *interfere* if their intersection is non-empty and either they occur in different goals in an and-box, or one occurs in the and-box of the other. A goal is *well-formed* if no sets of variables interfere.

A well-formed global goal is a *configuration* iff all variables occurring free anywhere in the goal occur in the set of local variables of some and-box in the goal. The letter γ stands for configurations.

OBSERVATION. *Transitions take well-formed goals to well-formed goals, and configurations to configurations.*

That goals are taken to goals is verified by inspection of the rules.

Variables are introduced in the hiding, choice, aggregate, and constraint simplification rules, and then as non-interfering sets of local variables.

Choice splitting duplicates sets of local variables, but not in a way that puts them in different goals in an and-box.

Promotion and the collect rule move variables between sets.

The set of local variables in the and-box promoted by the promotion rule does not interfere with the sets in the and-box to which it is promoted, since they are in different goals in this and-box, nor with the sets in any and-box containing it. Thus, the union will not interfere with any other set.

The set of local variables in the set promoted by the collect rule is renamed to avoid conflicts.

Other rules do not add or move variables. ♦

The *AKL computation model* is a structure $\langle \Delta, \rightarrow \rangle$, where Δ is the set of configurations, and $\rightarrow \subseteq \Delta \times \Delta$ is a transition relation, defined as the subset of goal transitions that are transitions on configurations.

$$\gamma \rightarrow \gamma' \equiv \gamma \overset{m}{\underset{\lambda}{\Rightarrow}} \gamma'$$

A *derivation* is a finite or infinite transition sequence

$$\gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \cdots$$

A configuration γ which satisfies $\neg\exists\gamma'.\ \gamma \rightarrow \gamma'$ is *terminal*.

Initial configurations are of the form

$$\mathbf{or}(\mathbf{and}(\mathrm{A})_{\mathrm{V}}^{\mathbf{true}})$$

where V is the set of variables occurring free in the statement A. The statement A is also referred to as the *query*.

Final configurations are of the form

$$\mathbf{or}(\mathbf{and}()_{\mathrm{V}_1}^{\sigma_1}, \ldots, \mathbf{and}()_{\mathrm{V}_n}^{\sigma_n})$$

in which all and-boxes are solved. A terminal configuration that is not final is *stuck*.

A *computation* is a derivation beginning with an initial configuration and, if finite, ending with a terminal configuration.

The sequence of and-boxes in a final configuration may be empty, in which case we talk of a *failed* computation. Otherwise, the constraints σ_i are referred to as the *answers* of the computation (also referred to as the answers to the query).

4.7 POSSIBLE EXTENSIONS

The computation model presented can be extended in many directions. By adding new guard operators, corresponding to new forms of choice statements, expressiveness is added to AKL with a minimum of effort, for programmers and implementors alike.

In the following, two variants of conditional choice are presented. The first, the logical condition, admits a cleaner logical interpretation. The second, cut, is unclean, but highly useful for executing Prolog programs in the AKL environment.

4.7.1 Logical Conditions

Due to the guard distribution rule, the condition operator of AKL prunes not only other clauses in the original choice, but also alternative solutions of the

guards. This has advantages, e.g., for a metainterpreter and for the implementation, but an operator which does not is subject to a less restrictive logical interpretation, as shown by Franzén [1994].

We introduce the *logical condition* operator $\rightarrow_L$ (called *soft cut* by Franzén) to which the guard distribution rule does not apply. To this operator corresponds a *logical conditional choice* statement.

We also need the notion of *or-component*, which is defined as follows. (1) If G is an and-box, G is an or-component of G. (2) If G is an or-component of G', then G is an or-component of **or**(R, G', S).

The *logical condition* rule

$$\textbf{choice}(G \rightarrow_L B, S) \underset{\chi}{\overset{D}{\Rightarrow}} \textbf{choice}(G \ ?\ B)$$

may be applied if σ_V is an or-component of G and σ is quiet with respect to env(χ) and V.

We may also add a rule

$$\textbf{choice}(R, G \rightarrow_L B, S) \underset{\chi}{\overset{D}{\Rightarrow}} \textbf{choice}(R, G \rightarrow_L B)$$

which prunes alternatives more eagerly if the above conditions hold and R and S are both non-empty.

4.7.2 Cut

Prolog provides the *cut* operation, which is noisy in the sense that pruning is performed without a quietness condition. In Prolog, synchronisation is given by the sequential execution order. Therefore, the state in which cut is applied is well-defined. (However, the noisiness of cut creates problems when *freeze* and similar coroutining constructs are added, as in the SICStus Prolog implementation [Carlsson et al 1993].)

In AKL, an operation very similar to cut can be introduced by adding a noisy form of the condition operator. This operation becomes very unpredictable unless some other form of synchronisation is added. The solution is to regard noisy pruning as a form of nondeterminate action, which is allowed in the same circumstances as choice splitting, i.e., in *stable* goals.

The syntax is augmented with a new guard operator, let us say '!', to which also corresponds a *cut choice* statement. The existing rules will work appropriately together with the following.

The *noisy cut* rule

$$\textbf{choice}(R, \sigma_V\ !\ B, S) \underset{\chi}{\overset{N}{\Rightarrow}} \textbf{choice}(R, \sigma_V\ !\ B)$$

may be applied within a stable goal if S is non-empty, and the *quiet cut* rule

$$\textbf{choice}(R, \sigma_V\ !\ B, S) \underset{\chi}{\overset{D}{\Rightarrow}} \textbf{choice}(R, C_V\ !\ B)$$

may be applied if S is non-empty and σ is quiet with respect to env(χ) and V.

The cut operation is highly useful when translating Prolog into AKL, and this application is discussed in Chapter 8.

4.8 FORMAL ASPECTS

Franzén [1994] presents soundness and completeness results for a logical interpretation of a subset of AKL, and a confluence result for a similar subset of AKL with certain restrictions on the choice splitting rule. These results are summarised here, partly informally, for the completeness of this exposition. Formal definitions and proofs are found in [Franzén 1994].

4.8.1 Logical Interpretation

Definitions and statements are interpreted (by *) as

$$(A := B)^* \Rightarrow A \equiv B^*$$

$$A^* \Rightarrow A \quad \text{(atom A)}$$

$$(A, B)^* \Rightarrow A^* \wedge B^*$$

$$(V : A)^* \Rightarrow \exists V\, A^*$$

$$(\, V_1 : A_1 \,\%\, B_1 \,;\, \cdots \,;\, V_n : A_n \,\%\, B_n \,)^* \Rightarrow$$
$$\exists V_1\, (A_1^* \wedge B_1^*) \vee \cdots \vee \exists V_n\, (A_n^* \wedge B_n^*) \quad (\% \in \{\,|\,, ?\})$$

$$(\, V_1 : A_1 \rightarrow B_1 \,;\, \cdots \,;\, V_n : A_n \rightarrow B_n \,)^* \Rightarrow$$
$$\exists V_1\, (A_1^* \wedge B_1^*) \vee \cdots$$
$$\vee ((\neg\exists V_1\, A_1^*) \wedge \cdots \wedge (\neg\exists V_{n\text{-}1}\, A_{n\text{-}1}^*) \wedge \exists V_n\, (A_n^* \wedge B_n^*))$$

and goals, correspondingly, as

$$\textbf{fail}^* \Rightarrow \textbf{false}$$

$$(\textbf{fail} \,\%\, B)^* \Rightarrow \textbf{false}$$

$$(\textbf{or}(G_1, \ldots, G_n) \,\%\, B)^* \Rightarrow (G_1 \,\%\, B)^* \vee \cdots \vee (G_n \,\%\, B)^* \quad (\% \in \{\,|\,, ?\})$$

$$(\textbf{or}(G_1, \ldots, G_n) \rightarrow B)^* \Rightarrow$$
$$(G_1 \rightarrow B)^* \vee \cdots \vee (\neg G_1^* \wedge \cdots \wedge \neg G_{n\text{-}1}^* \wedge (G_n \rightarrow B)^*)$$

$$\textbf{or}(G_1, \ldots, G_n)^* \Rightarrow G_1^* \vee \cdots \vee G_n^*$$

$$(\textbf{and}(G_1, \ldots, G_n)_V^{\sigma} \,\%\, B)^* \Rightarrow$$
$$\exists(V \setminus U)\, (\sigma \wedge G_1^* \wedge \cdots \wedge G_n^* \wedge B^*)$$

$$(\textbf{and}(G_1, \ldots, G_n)_V^{\sigma})^* \Rightarrow \exists(V \setminus U)\, (\sigma \wedge G_1^* \wedge \cdots \wedge G_n^*)$$

$$\textbf{choice}(G_1, \ldots, G_n)^* \Rightarrow G_1^* \vee \cdots \vee G_n^* \quad (\% \in \{\,|\,, ?\})$$

$$\textbf{choice}(G_1 \rightarrow B_1, \ldots, G_n \rightarrow B_n)^* \Rightarrow$$
$$(G_1 \rightarrow B_1)^* \vee \cdots \vee (\neg G_1^* \wedge \cdots \wedge \neg G_{n\text{-}1}^* \wedge (G_n \rightarrow B_n)^*))$$

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”.

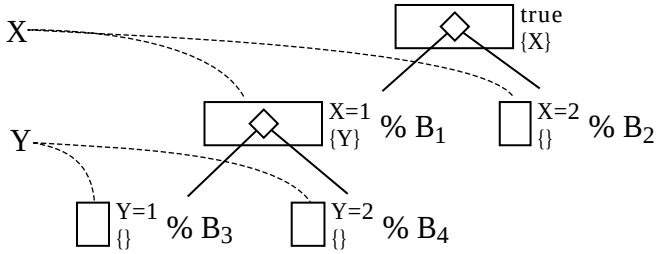


Figure 5.1. A hierarchy of constraint stores with suspensions

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.

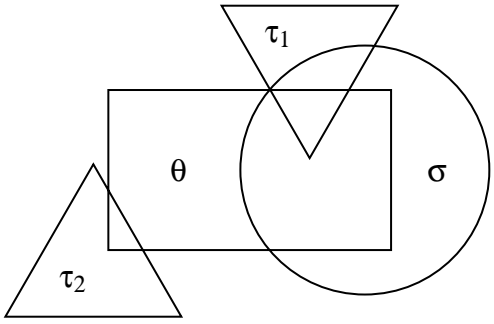


Figure 5.2. Ranges of possible values for variables

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.

5.6 EXECUTION STATES AND TRANSITIONS

An *execution state* is a tuple

$$(\gamma, sus, alst, clst, ctx)$$

where γ is a configuration, *sus* is a set of suspensions, *alst* is an and-list, *clst* is a choice-list, and *ctx* is a context list. The symbol ζ stands for execution states. The *execution model* is a structure $\langle E, \Rightarrow \rangle$, where E is the set of execution states and $\Rightarrow \subseteq E \times E$ is a transition relation on execution states.

The transition relation $\Rightarrow$ is defined by the derivation rules that follow. When a transition involves the application of one or several computation rules, this is explicitly stated, and also clearly reflected in the transition on the configuration part of the execution states involved. The context χ will be the same as in the corresponding computation rule, except for the choice splitting rule, which uses the contexts of Section 4.5 (Nondeterminism and Stability).

With each transition is associated a set of labels that are *new* in the transition, which will be used in Section 5.8.

5.6.1 Statement Tasks

Constraint Atom Success

$$\frac{(\chi[\ l::\mathbf{and}(\mathrm{R}, i::\mathrm{A}, \mathrm{S})^{\sigma}_{\mathrm{V}}\ ], sus, \mathbf{s}(i).alst, clst, ctx)}{(\chi[\ l::\mathbf{and}(\mathrm{R}, \mathrm{S})^{\mathrm{A}\wedge\sigma}_{\mathrm{V}}\ ], sus', \mathbf{w}(l, j_1)\ldots\mathbf{w}(l, j_m).alst, clst, ctx)} \quad (1)$$

The constraint atom rule is applied if env(χ) and A$\wedge\sigma$ are compatible.

The set *sus′* is $sus \cup \{v_1.l, \ldots, v_n.l\}$, where $\{v_1, \ldots, v_n\}$ is a sufficient set of variables for suspension of $\exists V(\sigma\wedge A)$ in an environment env(χ). The set $\{j_1, \ldots, j_m\}$ consists of all and-boxes in *l* suspended on variables in a sufficient set for waking when adding A to env(χ)$\wedge\sigma$. (Both *n* and *m* may be 0.)

Constraint Atom Failure

$$\frac{(\chi[\ j::\mathbf{and}(\mathrm{R}, i::\mathrm{A}, \mathrm{S})^{\sigma}_{\mathrm{V}}\ ], sus, \mathbf{s}(i).alst, clst, ctx)}{(\chi[\ k::\mathbf{fail}\ ], sus, \mathbf{a}(k), clst, ctx)} \quad (2)$$

The constraint atom rule and the environment synchronisation rule are applied if env(χ) and A$\wedge\sigma$ are incompatible. The label *k* is new.

Program Atom Execution

$$\frac{(\chi[\ i::\mathrm{A}\ ], sus, \mathbf{s}(i).alst, clst, ctx)}{(\chi[\ j::\mathrm{B}\ ], sus, \mathbf{s}(j).alst, clst, ctx)} \quad (3)$$

The program atom rule is applied, where B is as in the rule. An **s** (statement) task will execute the body. The label *j* is new.

Composition Execution

$$\frac{(\chi[\ \mathbf{and}(\mathrm{R},\ i{::}(\mathrm{A},\ \mathrm{B}),\ \mathrm{S})^{\sigma}_{\mathrm{V}}\ ],\ sus,\ \mathbf{s}(i).alst,\ clst,\ ctx)}{(\chi[\ \mathbf{and}(\mathrm{R},\ j{::}\mathrm{A},\ k{::}\mathrm{B},\ \mathrm{S})^{\sigma}_{\mathrm{V}}\ ],\ sus,\ \mathbf{s}(j).\mathbf{s}(k).alst,\ clst,\ ctx)} \tag{4}$$

The composition rule is applied, splitting a composition into its component statements. Two **s** (statement) tasks will execute the components. The labels *j* and *k* are new.

Hiding Execution

$$\frac{(\chi[\ \mathbf{and}(\mathrm{R},\ i{::}(\mathrm{U} : \mathrm{A}),\ \mathrm{S})^{\sigma}_{\mathrm{W}}\ ],\ sus,\ \mathbf{s}(i).alst,\ clst,\ ctx)}{(\chi[\ \mathbf{and}(\mathrm{R},\ j{::}\mathrm{B},\ \mathrm{S})^{\sigma}_{\mathrm{V}\cup\mathrm{W}}\ ],\ sus,\ \mathbf{s}(j).alst,\ clst,\ ctx)} \tag{5}$$

The hiding rule is applied, where V and B are as in the rule. An **s** (statement) task will execute its component. The label *j* is new.

Choice Execution

$$\frac{(\gamma,\ sus,\ \mathbf{s}(i).alst,\ clst,\ ctx)}{(\gamma',\ sus,\ \varepsilon,\ \mathbf{c}(j_1)\ldots\mathbf{c}(j_n).\mathbf{o}(k),\ (alst,\ clst).ctx)} \tag{6}$$

The choice rule is applied, where

$$\gamma = \chi[\ i{::}(\mathrm{U}_1 : \mathrm{A}_1\ \%\ \mathrm{B}_1\ ;\ \cdots\ ;\ \mathrm{U}_n : \mathrm{A}_n\ \%\ \mathrm{B}_n)\ ]$$

$$\gamma' = \chi[\ k{::}\mathbf{choice}((\mathbf{and}(j_1{::}\mathrm{A}'_1)^{\mathbf{true}}_{\mathrm{V}_1}\ \%\ \mathrm{B}'_1),\ \ldots,\ (\mathbf{and}(j_n{::}\mathrm{A}'_n)^{\mathbf{true}}_{\mathrm{V}_n}\ \%\ \mathrm{B}'_n))\ ]$$

and A'_i, B'_i, and V_i are as in the rule

A number of **c** (clause) tasks will execute the clauses, and an **o** (or) task will deal with promotion, failure, and suspension. The labels j_1, …, j_n, and *k* are new. The and-boxes introduced are also given new labels.

Aggregate Execution

$$\frac{(\chi[\ i{::}\mathrm{aggregate}(u,\ \mathrm{A},\ v)\ ],\ sus,\ \mathbf{s}(i).alst,\ clst,\ ctx)}{(\chi[\ \mathbf{aggregate}(w,\ j{::}\mathbf{or}(\mathbf{and}(k{::}\mathrm{B})^{\mathbf{true}}_{\{w\}}),\ v)\ ],\ sus,\ \varepsilon,\ \mathbf{c}(k).\mathbf{o}(j),\ (alst,\ clst).ctx)} \tag{7}$$

The aggregate rule is applied, where *w* and B are as in the rule. A **c** (clause) task will execute the goal, and an **o** (or) task will deal with suspension and the unit rule. The labels *j* and *k* are new. The aggregate box and the and-box are also given new labels.

5.6.2 Clause Tasks

Clause Execution

$$\frac{(\chi[\ i{::}\mathbf{and}(j{::}\mathrm{A})^{\mathbf{true}}_{\mathrm{V}}\ ],\ sus,\ \varepsilon,\ \mathbf{c}(j).clst,\ ctx)}{(\chi[\ i{::}\mathbf{and}(j{::}\mathrm{A})^{\mathbf{true}}_{\mathrm{V}}\ ],\ sus,\ \mathbf{s}(j).\mathbf{a}(i),\ clst,\ ctx)} \tag{8}$$

Whether the first (or single) alternative, or one tried as a result of failure or suspension of preceding guards, this rule initiates execution in an and-box. The **c** (clause) task on the choice-list is replaced by two tasks on the and-list: (1) an **s**

(statement) task for the statement in the and-box and (2) an **a** (and-box) task to deal with guards, aggregates, and the failure or suspension of an and-box.

5.6.3 And-Box Tasks

Promotion Execution

$$\frac{(\chi[\ l{::}\mathbf{and}(\mathrm{R}, \mathbf{choice}(i{::}\sigma_V\ \%\ \mathrm{B}), \mathrm{S})^{\theta}_{W}\ ],\ sus,\ \mathbf{a}(i),\ clst,\ (alst', clst').ctx)}{(\chi[\ l{::}\mathbf{and}(\mathrm{R}, j{::}\mathrm{B}, \mathrm{S})^{\sigma\wedge\theta}_{V\cup W}\ ],\ sus',\ \mathbf{w}(l, j_1)\ldots\mathbf{w}(l, j_n).\mathbf{s}(j).alst', clst', ctx)} \tag{9}$$

The promotion rule is applied. If % is '$\rightarrow$' or '$|$' it is required that σ is quiet with respect to $\theta \wedge \mathrm{env}(\chi)$ and V.

The set *sus′* is *sus* where suspensions *v.i* are replaced by suspensions *v.l*. The set $\{j_1, \ldots, j_n\}$ consists of all and-boxes in *l* suspended on variables in a sufficient set for waking when adding σ to $\mathrm{env}(\chi)\wedge\theta$. The label *j* is new.

Condition Execution

$$\frac{(\chi[\ j{::}\mathbf{choice}(\mathrm{R}, i{::}\sigma_V \rightarrow \mathrm{B}, \mathrm{S})\ ],\ sus,\ \mathbf{a}(i),\ clst,\ ctx)}{(\chi[\ j{::}\mathbf{choice}(\mathrm{R}, i{::}\sigma_V \rightarrow \mathrm{B})\ ],\ sus,\ \mathbf{a}(i),\ \mathbf{o}(j),\ ctx)} \tag{10}$$

The condition rule is applied if S is non-empty and σ is quiet with respect to $\mathrm{env}(\chi)$ and V.

Commit Execution

$$\frac{(\chi[\ j{::}\mathbf{choice}(\mathrm{R}, i{::}\sigma_V\ |\ \mathrm{B}, \mathrm{S})\ ],\ sus,\ \mathbf{a}(i),\ clst,\ ctx)}{(\chi[\ j{::}\mathbf{choice}(i{::}\sigma_V\ |\ \mathrm{B})\ ],\ sus,\ \mathbf{a}(i),\ \mathbf{o}(j),\ ctx)} \tag{11}$$

The condition rule is applied if at least one of R or S is non-empty and σ is quiet with respect to $\mathrm{env}(\chi)$ and V.

Collect Execution

$$\frac{(\gamma,\ sus,\ \mathbf{a}(i),\ clst,\ (alst', clst').ctx)}{(\gamma',\ sus,\ \varepsilon,\ clst,\ (\mathbf{s}(j).alst', clst').ctx)} \tag{12}$$

The collect rule is applied if σ is quiet with respect to $\mathrm{env}(\chi)$ and V, where

$$\gamma = \chi[\ k{::}\mathbf{and}(\mathrm{R}_1, \mathbf{aggregate}(u, \mathbf{or}(\mathrm{S}_1, i{::}\sigma_V, \mathrm{S}_2), v), \mathrm{R}_2)^{\theta}_{W}\ ]$$

$$\gamma' = \chi[\ k{::}\mathbf{and}(\mathrm{R}_1, \mathbf{aggregate}(u, \mathbf{or}(\mathrm{S}_1, \mathrm{S}_2), v'), j{::}collect(u', v', v), \mathrm{R}_2)^{\sigma\wedge\theta}_{V\cup W}\ ]$$

Observe that since the choice splitting rule is augmented with variable renaming, there is no need for renaming in the collection step. The variable *u′* is the copy corresponding to *u* in V, and *v′* does not occur in γ.

An **s** (statement) task will execute the collect operation. It is entered in the saved context, to be executed upon completion or suspension of the aggregate. The constraints are promoted immediately. No waking can occur since they are quiet. Suspensions referring to the and-box *i* need not be promoted. The label *j* is new.

Guard Failure Execution

$$\frac{(\chi[\ \mathbf{choice}(\mathrm{R},\ i{::}\mathbf{fail}\ \%\ \mathrm{B},\ \mathrm{S})\ ],\ sus,\ \mathbf{a}(i),\ clst,\ ctx)}{(\chi[\ \mathbf{choice}(\mathrm{R},\ \mathrm{S})\ ],\ sus,\ \varepsilon,\ clst,\ ctx)} \tag{13}$$

The guard failure rule is applied.

Failure in Or Execution

$$\frac{(\chi[\ \mathbf{or}(\mathrm{R},\ i{::}\mathbf{fail},\ \mathrm{T})\ ],\ sus,\ \mathbf{a}(i),\ clst,\ ctx)}{(\chi[\ \mathbf{or}(\mathrm{R},\ \mathrm{T})\ ],\ sus,\ \varepsilon,\ clst,\ ctx)} \tag{14}$$

The or-flattening rule is applied with S as the empty sequence.

Stable And-box Detection

$$\frac{(\chi[\ i{::}\mathrm{G}\ ],\ sus,\ \mathbf{a}(i),\ clst,\ ctx)}{(\chi[\ i{::}\mathrm{G}\ ],\ sus,\ \varepsilon,\ \mathbf{cs}(i).clst,\ ctx)} \tag{15}$$

if no suspensions in *sus* refer to the and-box *i* or and-boxes in *i*, i.e., *i* is stable, and the preceding and-box task rules (9–14) are not applicable.

And-Box Suspension

$$\frac{(\chi[\ i{::}\mathrm{G}\ ],\ sus,\ \mathbf{a}(i),\ clst,\ ctx)}{(\chi[\ i{::}\mathrm{G}\ ],\ sus,\ \varepsilon,\ clst,\ ctx)} \tag{16}$$

if the preceding and-box task rules (9–15) are not applicable.

5.6.4 Or-Box Tasks

Determinacy Detection

$$\frac{(\chi[\ i{::}\mathbf{choice}(j{::}\sigma_V\ \%\ \mathrm{B})\ ],\ sus,\ \varepsilon,\ \mathbf{o}(i),\ ctx)}{(\chi[\ i{::}\mathbf{choice}(j{::}\sigma_V\ \%\ \mathrm{B})\ ],\ sus,\ \mathbf{a}(j),\ \mathbf{o}(i),\ ctx)} \tag{17}$$

if % is ? or if σ is quiet with respect to env(χ) and V.

Determinacy of an or-box (due to failure of siblings) is detected. The single guard is re-entered, to perform promotion. No waking is possible since the and-box is solved.

Goal Failure

$$\frac{(\chi[\ \mathbf{and}(\mathrm{R},\ i{::}\mathbf{choice}(),\ \mathrm{S})^{\sigma}_{V}\ ],\ sus,\ \varepsilon,\ \mathbf{o}(i),\ (alst',\ clst').ctx)}{(\chi[\ j{::}\mathbf{fail}\ ],\ sus,\ \mathbf{a}(j),\ clst',\ ctx)} \tag{18}$$

The goal failure rule is applied. The label *j* is new.

Unit Execution

$$\frac{(\chi[\ \mathbf{aggregate}(u,\ i{::}\mathbf{fail},\ v)\ ],\ sus,\ \varepsilon,\ \mathbf{o}(i),\ (alst',\ clst').ctx)}{(\chi[\ j{::}unit(v)\ ],\ sus,\ \mathbf{s}(j).alst',\ clst',\ ctx)} \tag{19}$$

The unit rule is applied. The **s** (statement) task will execute the unit operation. The label *j* is new.

Choice-Box Suspension

$$\frac{(\chi[\ i::\mathbf{choice}(\mathrm{R})\ ],\ sus,\ \varepsilon,\ \mathbf{o}(i),\ (alst', clst').ctx)}{(\chi[\ i::\mathbf{choice}(\mathrm{R})\ ],\ sus,\ alst', clst', ctx)} \tag{20}$$

unless rules 17 or 18 are applicable.

Or-Box Suspension

$$\frac{(\chi[\ i::\mathbf{or}(\mathrm{R})\ ],\ sus,\ \varepsilon,\ \mathbf{o}(i),\ (alst', clst').ctx)}{(\chi[\ i::\mathbf{or}(\mathrm{R})\ ],\ sus,\ alst', clst', ctx)} \tag{21}$$

unless rule 19 is applicable.

5.6.5 Choice Splitting Tasks

As opposed to the other execution rules, in this section the contexts (χ and χ') are named as in Section 4.5 (Nondeterminism and Stability) instead of as in the corresponding computation rule (*choice splitting*).

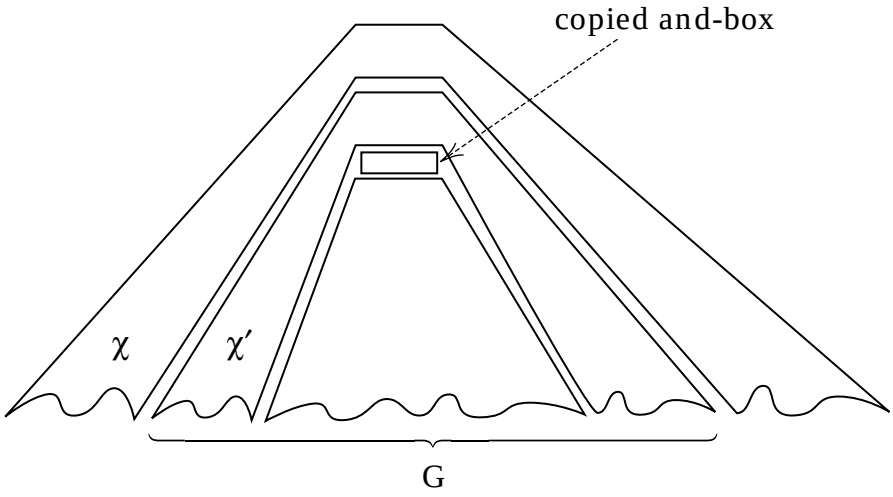


Figure 5.3. Contexts and goals involved in choice splitting

Choice Splitting Execution

$$\frac{(\chi[\ i::\mathrm{G}\ ],\ sus,\ \varepsilon,\ \mathbf{cs}(i).clst,\ ctx)}{(\gamma,\ sus',\ \varepsilon,\ clst',\ ctx')} \tag{22}$$

Since G is known to be stable, the choice splitting rule may be applied if G is of the form $\chi'[\ \mathbf{and}(S_1, \mathbf{choice}(T_1, T_2), S_2)^{\sigma}_{V}\ ]$, where $T_1 = (j::\theta_w\ ?\ A)$. (There is no need to commit to a particular choice of candidate.) If choice splitting is not applicable, let $\gamma = \chi[\ i::\mathrm{G}\ ]$, $sus' = sus$, $clst' = clst$, and $ctx' = ctx$, which is equivalent to and-box suspension.

Due to the handling of suspensions, the goals in the new branch need not only be duplicated, as suggested by the computation rule, but also *copied*, meaning that they will be given new labels and that local variables will be renamed. The copy we call G_1 and the copy of the candidate is labelled j'. All labels of goals in G_1 are new.

$G_1 = \mathbf{and}(S_1', k_1::\mathbf{choice}(j'::\theta'_{w'}\ ?\ A'), S_2')^{\sigma'}_{V'}$

The right branch retains the old labels.

$G_2 = \mathbf{and}(S_1, k_2::\mathbf{choice}(T_2), S_2)^{\sigma}_{V}$

Pertinent suspensions are duplicated.

$sus' = sus \cup sus_1 \cup sus_2$

$sus_1 = \{\ v.j \mid v.i \in sus,\ v$ external to $G_2,\ j$ copy of $i\ \}$

$sus_2 = \{\ v.j \mid u.i \in sus,\ u$ local to $G_2,\ j$ copy of $i,\ v$ renaming of $u\ \}$

If $\chi[\chi'] = \chi''[\ \mathbf{choice}(R_1, \lambda\ \%\ B, R_2)\ ]$, then splitting is applied in the scope of a choice-box. The guard distribution rule and guard failure rules are applied, removing immediately the new or-box created by splitting. The body B is *copied*, renaming variables V to corresponding variables in V′.

$\gamma = \chi''[\ l::\mathbf{choice}(R_1, G_1\ \%\ B', G_2\ \%\ B, R_2)\ ]$

The choice-box in which splitting is performed is labelled *l*.

If $\chi[\chi'] = \chi''[\ \mathbf{or}(R_1, \lambda, R_2)\ ]$, then splitting is applied in the scope of an or-box. The or-flattening rule is applied, removing the new or-box. Observe that the interaction with aggregates is explained with rule 12 (collect execution).

$\gamma = \chi''[\ l::\mathbf{or}(R_1, G_1, G_2, R_2)\ ]$

The or-box in which splitting is performed is labelled *l*.

If $\chi' = \lambda$ then choice splitting was performed with respect to the stable and-box. We promote the copied solution and remember to retry choice splitting. If the remaining choice-box is promotable, we promote it instead.

$ctx' = ctx$

if $T_2 = (\theta_W\ ?\ A'')$ then $clst' = \mathbf{p}(k_1).\mathbf{p}(k_2).clst$
else $clst' = \mathbf{p}(k_1).\mathbf{cs}(i).clst$

Otherwise, choice splitting was performed within the stable and-box. Again, we should promote the copied solution, from the point of view of the parent choice- or or-box, but choice splitting should not be retried at that point, since the boxes are not stable. However, if the remaining choice-box is promotable, it has to be promoted.

Let $i_0, j_0, \ldots, i_n, j_n$ be the labels of and-boxes (*i*s) and choice- or or-boxes (*j*s) in the path from the stable and-box labelled $i = i_0$ to the choice- or or-box in which splitting is performed labelled $l = j_n$.

$ctx' = (\mathbf{a}(i_n), \mathbf{o}(j_{n-1})).\ \ldots\ .(\mathbf{a}(i_1), \mathbf{o}(j_0)).(\mathbf{a}(i_0), clst).ctx$

if $T_2 = (\theta_W\ ?\ A'')$ then $clst' = \mathbf{p}(k_1).\mathbf{p}(k_2).\mathbf{o}(l)$
else $clst' = \mathbf{p}(k_1).\mathbf{o}(l)$

Stability guarantees that no failure will occur and no suspensions need be waked by moving down.

5.6.6 Choice Promotion Tasks

Promote after Splitting

$$\frac{(\chi[\, j::\mathbf{and}(\mathrm{R},\, i::\mathbf{choice}(k::\mathrm{G}\ ?\ \mathrm{B}),\, \mathrm{S})^{\sigma}_{\mathrm{V}}\,],\, sus,\, \varepsilon,\, \mathbf{p}(i).clst,\, ctx)}{(\chi[\, j::\mathbf{and}(\mathrm{R},\, i::\mathbf{choice}(k::\mathrm{G}\ ?\ \mathrm{B}),\, \mathrm{S})^{\sigma}_{\mathrm{V}}\,],\, sus,\, \mathbf{a}(j),\, \mathbf{o}(i),\, ctx')} \tag{23}$$

where

$ctx' = (\mathbf{a}(j),\, clst).ctx$

which makes it ready for rule 9 (promotion execution). No waking is possible, since we are in a goal which was stable before choice splitting, and if we are in a aggregate, collect operations have not yet been executed.

5.6.7 Wake-up Tasks

Woken Up

$$\frac{(\gamma,\, sus,\, \mathbf{w}(i,\, j).alst,\, clst,\, ctx)}{(\gamma,\, sus,\, alst,\, clst,\, ctx)} \tag{24}$$

if $i = j$ or j is the label of an and-box dead in γ.

Wake Up

$$\frac{(\chi[\, i::\mathrm{G}\,],\, sus,\, \mathbf{w}(i,\, j).alst,\, clst,\, ctx)}{(\chi[\, i::\mathrm{G}\,],\, sus,\, \mathbf{in}(k).\mathbf{w}(k,\, j).\mathbf{a}(k),\, \mathbf{o}(l),\, (alst,\, clst).ctx)} \tag{25}$$

if $i \neq j$ and j is the label of an and-box live in γ, where

$\mathrm{G} = \mathbf{and}(\mathrm{R}_1,\, l::\mathbf{choice}(\mathrm{S}_1,\, \mathrm{G}'\ \%\ \mathrm{B},\, \mathrm{S}_2),\, \mathrm{R}_2)^{\sigma}_{\mathrm{V}}$

or $\mathrm{G} = \mathbf{and}(\mathrm{R}_1,\, \mathbf{aggregate}(u,\, l::\mathbf{or}(\mathrm{S}_1,\, \mathrm{G}',\, \mathrm{S}_2),\, v),\, \mathrm{R}_2)^{\sigma}_{\mathrm{V}}$

and $\mathrm{G}' = \chi'[j::\mathrm{G}''] = k::\mathbf{and}(\mathrm{S})^{\theta}_{\mathrm{W}}$

Installation Success

$$\frac{(\chi[\, i::\mathbf{and}(\mathrm{R})^{\sigma}_{\mathrm{V}}\,],\, sus,\, \mathbf{in}(i).alst,\, clst,\, ctx)}{(\chi[\, i::\mathbf{and}(\mathrm{R})^{\sigma}_{\mathrm{V}}\,],\, sus',\, \mathbf{w}(i,\, j_1)\ldots\mathbf{w}(i,\, j_m).alst,\, clst,\, ctx)} \tag{26}$$

if env(χ) and σ are compatible.

The set sus' is $sus \cup \{v_1.i,\, \ldots,\, v_n.i\}$, where $\{v_1,\, \ldots,\, v_n\}$ is a sufficient set of variables for suspension of $\exists \mathrm{V}\sigma$ in an environment env(χ). The set $\{j_1,\, \ldots,\, j_m\}$ consists of all and-boxes in i suspended on variables in a sufficient set for waking when adding σ to env(χ). (It is only necessary to wake at this point if the suspension-waking scheme requires multistage waking, as for the optimised treatment of rational trees.)

Installation Failure

$$\frac{(\chi[\ i::\mathbf{and}(\mathrm{R})^{\sigma}_{\mathrm{V}}\ ],\ sus,\ \mathbf{in}(i).alst,\ clst,\ ctx)}{(\chi[\ j::\mathbf{fail}\ ],\ sus,\ \mathbf{a}(j),\ clst,\ ctx)} \tag{27}$$

The environment synchronisation rule is applied if env(χ) and σ are incompatible. The label *j* is new.

5.7 EXECUTIONS

A *partial execution* is a finite or infinite sequence of execution states

$$\zeta_0 \Rightarrow \zeta_1 \Rightarrow \zeta_2 \Rightarrow \cdots$$

in which labels that are new in the transition to ζ_i do not occur as labels of goals in any preceding state ζ_j ($j < i$).

A execution state ζ which satisfies $\neg\exists\zeta'.\ \zeta \Rightarrow \zeta'$ is *terminal*.

An *initial* execution state is of the form

$$(\mathbf{or}(\mathbf{and}(j::\mathrm{A})^{\mathbf{true}}_{\mathrm{V}}),\ sus,\ \varepsilon,\ \mathbf{c}(j),\ \varepsilon)$$

where *sus* is an empty set of suspensions.

A *final* execution state is of the form

$$(\gamma,\ sus,\ \varepsilon,\ \varepsilon,\ \varepsilon)$$

where γ is a *final* configuration. A terminal execution state that is not final is *stuck*.

An *execution* is a partial execution beginning with an initial execution state and, if finite, ending with a terminal execution state.

By the *goal to which a computation rule can be applied*, we mean (1) for statement rules, the statement itself, (2) for the pruning and collect rules, the solved and-box, (3) for the promotion rule, the solved and-box or the choice-box, (4) for the environment failure rule, the and-box replaced by fail, (5) for the goal failure rule, the empty choice-box, (6) for the guard failure rule, the failed guard, (7) for the choice splitting rule, the stable and-box, (8) for the unit rule, the empty or-box corresponding to fail, and (9) for the or-flattening rule, in the case of flattening an empty or-box, the empty or-box.

Below we assume that the scheme for suspension and waking is complete.

LEMMA 1. *In all execution states that do not contain* **in**, *all subgoals of the configuration to which a computation rule can be applied are referred to by tasks.*

PROOF. This is shown by induction on execution steps.

We first observe, by inspecting all rules, that in all task lists, tasks refers to subgoals of the box referred to by the **a** or **o** task that end them. We also observe, by inspecting the rules that introduce multiple choice-tasks (6 and 22), that tasks in choice-lists refer to subgoals of the pertaining or- or choice-box in left-to-right order, and that tasks are processed in this order.

The initial state contains a statement to which a computation rule can be applied, which is referred to by a **c** task.

It can be argued for each execution rule, by tedious inspection, that the goals to which computation rules can be applied after a corresponding step are either given new tasks or are already referred to by tasks. The more interesting cases are described.

The constraint atom and installation execution rules can lead (1) to the incompatibility with its environment of the local store, (2) to the quietness or incompatibility of constraint stores in subgoals of the and-box, or (3) to the and-box being solved. Either (zero or more) **w** tasks are entered that refer to all and-boxes for which constraint stores may have become quiet or incompatible, or an **a** task is left that refers to the failed and-box. In the first case, if the and-box is solved, it is referred to by the **a** task in the list. In the second case, the tasks removed refer to dead goals.

The condition and commit execution rules remove all siblings or siblings to the right of the solved and-box. This and-box was reached by what was the first task on the pertaining choice-list. Therefore, the other tasks, except **o**, refer to subgoals to the right, and may safely be removed in both cases.

The collect execution rule moves tasks in an unusual way, but adds an **s** task for the new statement, and there is already an **o** task for the or-box, in case it becomes empty.

The guard failure execution and failure-in-or execution rules remove goals, and have already **o** tasks for the choice- or or-box, in case they become determinate or empty.

The choice splitting execution rule can suspend due to the lack of a candidate. Otherwise, it makes promotion applicable to the copy of the candidate. Promotion may have become applicable in the right branch. If the copied and-box was stable, the right branch may still be stable. These possibilities are handled by tasks introduced.

The remaining rules rather trivially preserve the property. ♦

LEMMA 2. *A terminal execution state in an execution is of the form*

$$(\gamma, sus, \varepsilon, \varepsilon, \varepsilon)$$

where γ *is a terminal configuration*.

PROOF.

(1) All tasks have rules that apply in any execution state that may arise. In particular, the **a** and **o** tasks have suspension rules that pop the context stack. Thus, the task and context stacks are empty in a terminal state.

(2) If there are no tasks, γ is a terminal configuration, by Lemma 1. ♦

LEMMA 3. *Only a finite number of transitions on an execution state are made between transitions on its configuration*.

PROOF. The tasks **s**, **c**, **a**, **o**, **cs**, and **p** either lead to transitions, or are skipped by suspension rules and the like, and there is only a finite number of tasks to skip. Unless the configuration is changed by a transition, each **in** and **w** leads to a finite number of **w** and **in**, since the tree of and-boxes visited is finite. ♦

PROPOSITION (correctness). *The sequence of configurations in an execution, in which subsequent equal configurations are deleted, forms a computation.*

PROOF. An initial execution state contains an initial configuration. Transitions between execution states that modify the configuration do so by applying computation rules, which are implicitly propagated by the subgoal rule, and where we for copying make implicit use of a rule that allows us to rename the local variables of and-boxes. Terminal execution states contain terminal configurations by Lemma 2. Execution is guaranteed to make progress and either produce a finite or an infinite computation by Lemma 3. ♦

5.8 DISCUSSION

The execution model presented is not fair, in that a task can remain unattended indefinitely. For example,

```
p, a = b
```

where

```
p :- p.
```

will loop, not fail.

A fair execution model is arguably more appealing to the intuition. All agents are active, and have their own “virtual processor”. It is easier to reason about some properties of programs. However, the intuition can also be misleading.

A common misunderstanding is that a producer and a consumer will execute in a, more or less, constant size execution state (if garbage collection simplification is performed). This is not so. The producer can produce at, say, twice the speed of the consumer. To guarantee constant size, a bounded buffer programming technique has to be used (see, e.g., [Shapiro 1987]).

Another misunderstanding is that fairness models the notion of background processes, for example, background re-pagination of a document in a word processor while it is being edited. Basic fairness, however, has nothing to say about the resources devoted to each task. Re-pagination might very well use 90% of the time, and slow down editing, or it might use 0.1%, and never catch up.

These misunderstandings are usually beginner's problems.

A problem of a quite different nature is that by stipulating fairness we also stipulate “breadth-first” search in programs employing don't know nondeterminism. This could be relaxed, by giving (unfair) priority to the leftmost alternative in don't know choice, conditional choice, and ordered bagof, since the

difference cannot be observed. Such stipulations would be fundamental to the nature of the language.

A fair execution model for AKL is also considerably more complex. The investigation of AKL, and related languages, has not yet reached the point where it would be possible to argue conclusively for or against general fairness.

Meanwhile, it is obvious that, to be useful for, e.g., background processes, even a fair execution model has to be complemented with some form of metalevel control that allocates computational resources to different subcomputations. Such directives are available in the Oz language and system [Smolka et al 1994].

CHAPTER 6

AN ABSTRACT MACHINE

The abstract machine presented here is a refinement of the execution model, with some discrepancies related to waking. It specifies in more detail the representation of programs and execution states, and how tasks are executed on this representation. These representations are chosen to be suitable for implementation on a concrete machine, but are not discussed at the implementation level.

The abstract machine is particular to the constraint system of rational trees. The requirements of arbitrary constraint solvers are not sufficiently understood at this point to allow a fully general treatment. Carlson, Carlsson, and Diaz [1994] and Keisu [1994] have investigated particular constraint solvers—for finite domain constraints and rational tree constraints closed under all logical combinators.

A concrete implementation exists, the AGENTS system, that is based on a closely related abstract machine [Janson and Montelius 1992; Janson et al 1994]. The abstract machine of this presentation is for the most parts considerably simplified, but supports the *eager waking* behaviour of the execution model as well as the *lazy waking* of AGENTS 0.9. Some aspects of the concrete implementation are discussed.

No attempt has been made to describe an optimal machine. On the contrary, much has been sacrificed to keep the presentation simple. The purpose of this chapter is to introduce fundamental machinery for this particular style of abstract machine. Some important optimisations are described.

6.1 OVERVIEW

An execution state corresponds to a *machine state* as follows.

The (labelled) configuration is represented by *and-nodes* (corresponding to guarded goals with and-box guards or to and-boxes in or-boxes), *choice-nodes* (corresponding to choice-boxes, aggregate-boxes with or-boxes, or to the top-level or-box), *and-continuations* (corresponding to statements), *choice-continua-*

tions (corresponding to parts of choice-statements), *variables*, *symbols*, *tree constructors*, *constraints*, and *instructions* (representing statements)

The set of suspensions is represented by *suspensions* on variables and the *trail*.

The combination of and-list, choice-list, and context list is represented by the *and-stack*, the *choice-stack*, the *context stack*, and the *current and-node* register.

Agent definitions and their component statements are represented by *instructions*. The choice of instruction set is largely orthogonal to the rest of the abstract machine. An instruction set which is loosely based on the WAM is presented [Warren 1983], and some optimisations are suggested.

6.2 DATA OBJECTS

Objects are characterised by their types and attributes. All objects and some attributes of objects have unique *references* in a given machine state. For the representations of goals, these correspond to labels. It is assumed that there is a reference, called *null*, that is different from the reference of any object, and which may be used in the place of any reference. Objects reside in different data areas (which are described in Section 6.3). Occasionally, a reference will be referred to as the object and vice versa, and a reference to an attribute as a reference to the object, for the purpose of less verbose descriptions, and no confusion is expected from this relaxation.

6.2.1 And-Nodes

An *and-node* has the attributes

- *parent*: a reference to a choice-node
- *alternatives*: a pair of references to alternatives attributes
- *constraints*: a reference to a constraint (list)
- *goals*: a pair of references to goals attributes
- *continuation*: a reference to an and-continuation
- *forward*: a reference to an and-node
- *state*: one of live-stable, live-unstable, or dead

The and-node is on a double-linked list of alternatives, one link of which is in its parent choice-node. It has to support arbitrary insertion and deletion, and is double-linked for simplicity. Of the pair, one is called *left* and the other *right*.

The constraint list is null until the and-node is suspended for the first time (if ever). It will then contain bindings on external variables.

The and-node contains a double-linked list of goals, one link of which is in the and-node itself. Like the list of alternatives, it has to support arbitrary insertion and deletion, and is double-linked for simplicity. It is initially empty, i.e., its links refer to itself.

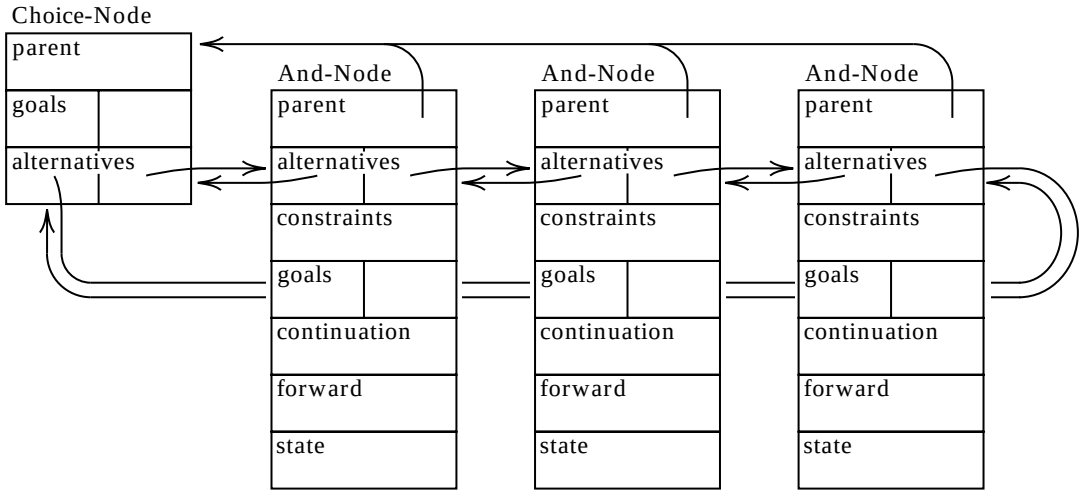


Figure 6.1. Tree of and- and choice-nodes

The continuation represents the guard operator and the body of a guarded goal, or serves as a special sentinel in and-nodes in the top-level and in aggregates. As the constraint list, it is null until the and-node is suspended.

Until promotion, the forward pointer is null. Then, it is set to refer to the and-node to which promotion took place.

The state of an and-node is either *live* or *dead*. The state of a live and-node is further subdivided into *live-stable* or *live-unstable*. In a node which is live-stable it is known that no constraint on external variables exists within the node. In a node which is live-unstable there might be such a constraint. An and-node is initially marked as *stable*. If a constraint is placed on a variable external to the and-node it is marked as *unstable*. An and-node that has been marked as unstable will never be marked as stable again. An and-node is marked as dead if it fails, is pruned, or is promoted.

6.2.2 Choice-Nodes

A *choice-node* has the attributes

- *parent*: a reference to an and-node
- *goals*: a pair of references to goals attributes
- *alternatives*: a pair of references to alternatives attributes

In the root choice-node, the parent and-node is null.

The choice-node is in a double linked-list of goals, one link of which is in the parent and-node.

The choice-node contains a double linked-list of alternatives, one link of which is in the choice-node itself.

Figure 6.1 illustrates how and- and choice-nodes are connect via parent and double links. In the figure, a choice-node is the parent and it and the and-nodes

are linked via an alternatives chain. The situation in which an and-node is the parent and the choice-nodes are linked via a goals chain is symmetric.

A node is *in* another node, if the latter can be reached from the former by following parent links. By the *grandparent* of a node is meant its parent's parent. By *empty* alternatives and goals, we mean that they contain no and-nodes or choice-nodes, respectively. An object in a double-linked list is known to be *left-most* (*right-most*) if its left (right) reference refers to an attribute in its parent. Familiar notions such as *inserting* and *unlinking* objects will be used on single- and double-linked lists.

6.2.3 Constraints and Variables

The abstract machine is specific to the constraint system of rational trees. There are three types of components in tree expressions: *symbols*, *tree constructors*, and *variables*. A *tree* is a reference to a tree component. In terms of the computation model, a tree is a variable which is constrained to be equal to the expression to which it refers.

A *symbol* has no attributes.

A *tree constructor* has the attributes

- *functor*: the name and arity
- *arguments*: a vector of trees

A *variable* has the attributes

- *value*: a tree
- *state*: one of bound or unbound
- *home*: a reference to an and-node
- *suspensions*: a reference to a suspension

In a new variable, suspensions is null and the state is unbound.

A *suspension* has the attributes

- *next*: a reference to a suspension (list)
- *suspended*: a reference to an and-node

The constraints in an and-node correspond to bindings to external variables.

A *constraint* has the attributes

- *next*: a reference to a constraint
- *variable*: a reference to a variable
- *value*: a tree

6.2.4 And-Continuations

An *and-continuation* has the attributes

- *continuation pointer*: a reference to a sequence of instructions

- *y-registers*: a vector of trees

The instructions represent a statement of which the trees in the vector are the (constrained) free variables.

And-continuations serve a dual purpose, as the representation of a statement and as the representations of corresponding **s** (statement) tasks.

6.2.5 Choice-Continuations

A choice-continuation has the attributes

- *continuation pointer*: a reference to a sequence of instructions
- *a-registers*: a vector of trees

The instructions represent unexplored guarded goals in a choice-box of which the trees in the vector are the (constrained) free variables.

Like and-continuations, choice-continuations serve a dual purpose, as the representation of guarded goals and as the representation of a corresponding sequence of **c** (clause) tasks.

6.2.6 Trail Entries

A *trail entry* has the attribute

- *variable*: a reference to a variable

A trail-entry refers to an external variable which has been bound locally.

6.2.7 Tasks

The tasks of the abstract machine correspond closely to the tasks of the execution model.

The **s** (statement) and **a** (and-box) tasks are represented by and-continuations, which also serve as representations of the pertaining statements. The **w** (wake) task is explicit also in the abstract machine.

The **c** (clause) tasks are represented by choice-continuations, which also serve as representations of the pertaining guarded goals. The **o** (or-box, here called no-choice), **cs** (choice splitting), and **p** (promote) tasks are explicit also in the abstract machine.

When regarded as tasks, and- and choice-continuations are referred to as tasks.

There is a need for new tasks related to waking—*resume* and *restore insertion point*. At the point of a call, the program counter and the x-registers represent a program atom that has to be made explicit when switching context. The insertion point corresponds to the label in the **s** tasks.

The *no-choice* task has no attributes.

The *wake*, *choice splitting*, and *promote* tasks have the attribute

- *node*: a reference to an and-node

The *resume* task has the attributes

- *continuation pointer*: a reference to a sequence of instructions
- *a-registers*: a vector of trees

The *restore insertion point* task has the attribute

- *goals*: a pair of references to goals attributes

6.2.8 Contexts

A *context* has the attributes

- *and*: a reference to an and-task or and-continuation
- *choice*: a reference to a choice-task or choice-continuation
- *trail*: a reference to a trail entry

Contexts serve the purpose of the context list in the execution model, namely to organise tasks in and- and choice-node levels. The trail attribute is a part of the representation of the set of suspensions.

6.3 DATA AREAS AND REGISTERS

6.3.1 Data Areas

The abstract machine has seven data areas: heap, trail stack, and-stack, choice stack, context stack, wake-up stack, and static store (Figure 6.2, next page).

The *heap* is an unordered collection of and-nodes, choice-nodes, tree constructors, variables, suspension lists, constraint lists, and and-continuations. New objects can be *created* on the heap and are then assigned unique references. Objects on the heap that are not referenced are removed by *garbage collection* (which is not discussed further).

A *stack* is a sequence of objects, regarded as extending upwards. New objects can be *pushed* on the stack and are then assigned unique references. For a given stack, the reference that would be assigned to a pushed object is known, and is called *top*. Any object in a stack can be *removed*, giving new references to all objects above it. When the top-most object is removed, it is *pop'ed*.

The *trail-stack* is a stack of trail entries. The *and-stack* is a stack of and-continuations and and-tasks. The *choice-stack* is a stack of choice-continuations and choice-tasks. The *context-stack* is a stack of contexts. The *wake-up* stack is a stack of references to and-nodes.

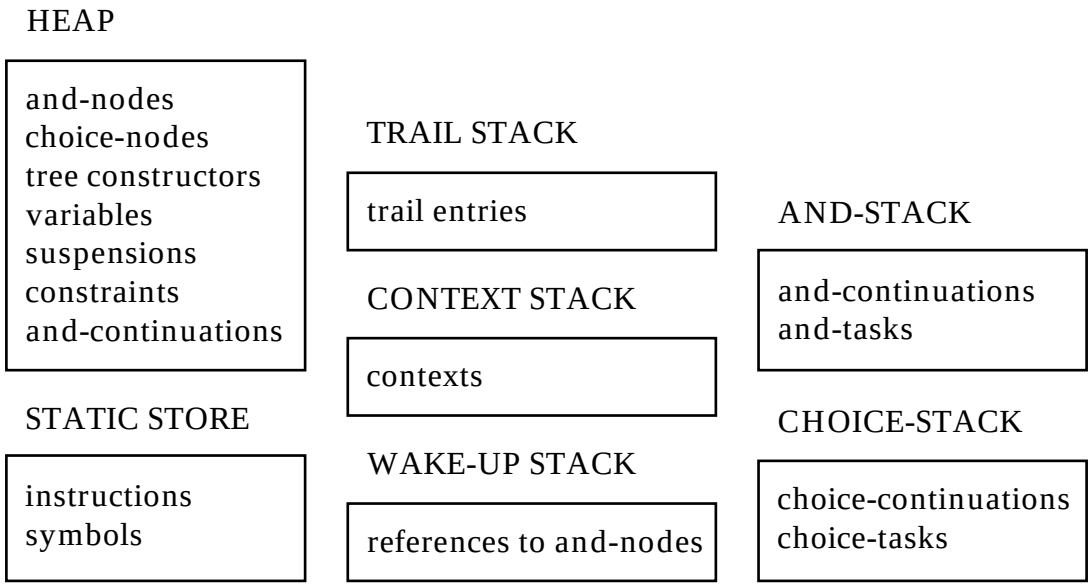


Figure 6.2. Data areas of the abstract machine

The *static store* contains symbols and sequences of instructions.

Since the data area can often be inferred from the context, it will not always be explicitly named.

6.3.2 Registers

Registers are implicit in the management of the data areas. We will speak of the *current* objects of stacks, meaning the objects immediately below their tops. Other registers are

- the *program counter*: a reference to a sequence of instructions
- the *current and-node*: a reference to an and-node
- the *insertion point*: a reference to a goals attribute
- the *x-registers*: a vector of trees
- the *current argument*: a reference to a tree
- the *unification mode*: one of read or write

By the notation x_i is meant the *i*th x-register, counting from 0 (in accordance with the tradition). It is assumed that all x-registers exist that are accessed by instructions currently in the static store.

By the notation y_i is meant the *i*th element of the y-registers attribute of the current and-continuation, counting from 0. By the *current choice-node* we mean the parent of the current and-node.

6.3.3 Initial States

In an initial state, the static store contains instructions and symbols for a program, as given by Section 6.6. All other data areas are empty. The current and-

node is null. The instruction pointer refers to the code for the initial statement, as given by Section 6.6.7. The machine is thus ready to enter the instruction decoding state.

6.3.4 Useful Concepts

An object in the and-stack, choice-stack, or trail stack is *in the current context* if it is equal to or above the object referred to by the corresponding attribute in the current context, or if the context stack is empty.

An and-node is *solved* when its goals attribute is empty and the program counter refers to a guard instruction.

An and-node is *quiet* when it is solved, the constraints list is empty, and there are no trail entries in the current context.

A variable is *local* if its home attribute, dereferenced, refers to the current and-node. The variable is *external* if the current and-node is in the and-node referred to. Otherwise the variable is unrelated (an insignificant case).

To *dereference an and-node,* if its forward attribute is non-null, dereference it, otherwise return the and-node.

An and-node is *dead* if it, dereferenced, has state dead or is in an and-node, the state of which is dead. Otherwise, it is *live.*

An and-node is *determinate* if it is the only alternative and the topmost element of the choice-stack is a no-choice task.

6.4 EXECUTION

During execution, the abstract machine moves between main states as illustrated by Figure 6.3 (next page). These are completely defined by

- contents of data areas
- values for pertaining registers

The abstract machine can be halted and restarted again at these points with no extra information. Other, here transient, states exist that could be made well-defined in an implementation with the aid of extra registers.

The registers needed to enter a state are

- for Decode Instructions, (potentially) all except unification mode
- for Fail, the current and-node
- for Back Up, the current and-node
- for Wake, (potentially) all except unification mode
- for Proceed, the current and-node, insertion pointer
- for Split?, the current and-node

The following sections describe these states and their pertaining actions.

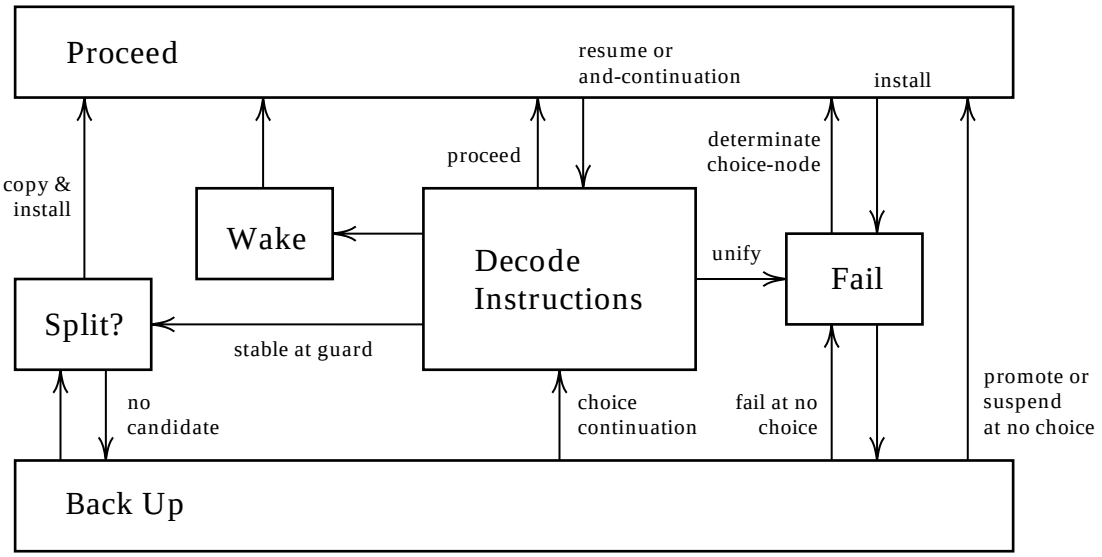


Figure 6.3. States of the abstract machine

6.4.1 Instruction Decoding

To *decode instructions*, interpret the instruction referenced by the program counter (according to Section 6.5). Unless the instruction moves to another state, continue decoding instructions, and, unless the instruction changes the program counter, move to the next instruction.

6.4.2 Unifying and Binding

To *unify x and y* perform the following. A set of pairs of references is used for unification of cyclic structures. It is assumed to be empty when unify is entered from instruction decoding. Dereference *x* and *y*. If *y* is a local variable, swap *x* and *y*, then select the first applicable case below.

- If *x* and *y* are identical references, do nothing.
- If *x* is a variable, bind *x* to *y*.
- If *y* is a variable, bind *y* to *x*.
- If *x* and *y* are tree constructors with the same functor, then, if *x.y* is found in the set of pairs, then do nothing, otherwise, add *x.y* to the set and unify each pair of corresponding arguments.
- Otherwise, fail.

To *bind x to y*, set the variable to state bound and store *y* in its value attribute, then select the applicable case below.

- If *x* is a local variable, unlink all suspensions on *x* and push references to the and-nodes on the wake stack.
- If *x* is an external variable, unlink all suspensions to and-nodes in the current and-node from *x*, and push corresponding references to the sus-

pended and-nodes on the wake stack. (It is also appropriate to unlink suspensions referring to dead and-nodes.) Push a trail entry for *x*.

Note that this is one point where the abstract machine differs from the execution model. Waking does not occur at the point of adding constraints, since it is not known which x-registers are used. Instead, waking is delayed until processing the next non-constraint statement.

To *dereference a tree x*, if *x* is a bound variable, dereference its value, otherwise return *x*. To *dereference a register x*, dereference the contents of *x*, and store the result in *x*.

6.4.3 Failing

To *fail the current and-node*, change its state to dead and unlink it from alternatives. Remove all trail entries in the current context, resetting their variables to unbound. Reset all variables in the constraint list to unbound. Remove all and-tasks in the current context. Remove all references from the wake-up stack. If there remains a single solved and determinate sibling and-node, install it and proceed. Otherwise, back up.

6.4.4 Suspending

To *suspend the current and-node*, unless it has a continuation, store the program counter in the continuation pointer of the current and-continuation, create a copy of it on the heap, making the copy the continuation of the current and-node, and pop it from the and-stack. Reset all variables in the constraint list to unbound. For each trail entry in the current context: Remove it and reset its variable to unbound. Create a constraint referring to the variable and its value. and add it to the current and-node. Suspend the and-node on the variable. If the dereferenced value is a variable, suspend the and-node also on that variable.

To *suspend an and-node on a variable*, add a suspension for this and-node to the variable and, for each and-node in the path from the current and-node to, but not including, the home and-node of the variable, change its state to unstable.

6.4.5 Backing Up

To *back up*, examine the top object of the choice-stack, and do one of the following depending on its type.

- If it is a choice continuation, decode instructions starting from its continuation pointer.
- If it is a promote task, remove it, install (simply) the and-node referred to from the parent choice-node, and promote it.
- If it is a choice splitting task, remove it, set the current and-node to the node referred to by the task, and attempt choice splitting.
- If it is a no-choice task, remove the task, and remove the top context. If the parent of the current choice-node is null, terminate the abstract machine.

Otherwise, if there are no alternatives, set the current and-node to the parent and fail. Otherwise, set the insertion point to the right sibling of the current choice-node, set the current and-node to the parent, and proceed.

Note that, to avoid testing if the parent of the choice-node is null, a special no-choice task can be used in the root node.

6.4.6 Waking

To *wake at call with N arguments*, push a resume task with the program counter as continuation pointer and the N x-registers as a-registers. Push a restore insertion point task and insert it at the insertion point. Wake and proceed.

To *wake at proceed*, push a restore insertion point task and insert it at the insertion point. Wake and proceed.

To *wake at a guard*, store the program counter in the continuation pointer of the current and-continuation, wake, and proceed.

To *wake*, for each reference to an and-node on the wake stack, remove it and, if the and-node is live, push a corresponding wake task.

Note that pushing a resume or insertion point task only to discover that no live node could be waked can easily be avoided in an implementation.

6.4.7 Promoting

To *promote* the current and-node, set the forward attribute to refer to the parent of the current choice-node, here called the *target and-node*. Unlink the current choice-node, setting the insertion point to its right sibling. Promote the trail and promote constraints. Set the current and-node to the target and-node. Remove the current context and remove the no-choice task. Decode instructions starting with the next instruction (following the current guard instruction).

To *promote simply*, do as above, but do not promote the trail and constraints, which are known to be empty.

To *promote the trail*, for each trail entry in the current context, select the first applicable case below.

- If the home of its variable is the target and-node, remove the trail entry. If suspensions on the variable refer to and-nodes in the target and-node, push references to these on the wake stack and unlink the suspensions. Unlink also all suspensions referring to dead nodes.
- If the dereferenced value of the variable is a variable, the home of which is the target and-node, bind this latter variable to the former after changing the state of the former to unbound. Remove the trail entry.
- Otherwise, do nothing.

To *promote constraints*, for each constraint in the current and-node, select the first applicable case below.

- If the home of its variable is the target and-node, then, if suspensions on the variable refer to and-nodes in the target and-node, push references to these on the wake stack and unlink the suspensions. If there are no such suspensions, do nothing.
- If the dereferenced value of its variable is a variable, the home of which is the target and-node, bind this latter variable to the former after changing the state of the former to unbound.
- Otherwise, move the constraint to the target and-node.

6.4.8 Pruning

To *prune alternatives,* unlink all alternatives to the right and left of the current and-node and mark them as dead. Delete all choice-tasks in the current context and push a no-choice task.

To *prune alternatives to the right,* do as above, but unlink only to the right.

6.4.9 Proceeding

To *proceed,* examine the top object of the and-stack, and do one of the following depending on its type.

- If it is an and-continuation, decode instructions starting from the continuation pointer.
- If it is a wake task, pop it. If it refers to a live and-node, dereference and install it. Proceed.
- If it is a restore insertion point task, pop it, unlink it, set the insertion point to the right of the goals attributes referred to, and proceed.
- If it is a resume task, pop it, move the a-registers to corresponding x-registers, and decode instructions starting from the continuation pointer.

6.4.10 Installing

To *install an and-node when proceeding,* for each of the and-nodes in the path from, but not including, the current and-node to the and-node to be installed, enter its parent choice-node and then enter itself. This process may be interrupted by failing when entering and-nodes.

To *install an and-node when failing, backing up, or collecting,* do as above, but start by entering the first and-node.

To *enter a choice-node,* push a context referring to the tops of the pertinent stacks, then a no-choice task.

To *enter an and-node,* make it the current and-node and make its goals attribute the insertion point. For each constraint, select the first applicable case below.

- If its dereferenced variable is equal to its dereferenced value, unlink the constraint.

- If its variable is unbound, set the value of this variable to the value in the constraint and change state to bound.
- If its variable is bound and its value is an unbound variable, swap the variable and the value of the constraint, and do as above.
- Otherwise, unlink the constraint and unify the value of the variable with the value of the constraint. This may fail, interrupting entering.

Then, push an and-continuation corresponding to the continuation of the and-node. Finally, wake.

Note that when *installing simply*, only case two is applicable when entering an and-node, since failure or entailment cannot occur due to stability.

6.4.11 Choice Splitting

To *attempt choice splitting*:

- Select a candidate. If none is found, back up.
- If the candidate's grandparent is the current and-node, then, if it has only one sibling, which is solved, push a promote task referring to the sibling, otherwise, push a choice splitting task referring to the current and-node.
- Copy w.r.t. the candidate.
- Install (simply) the copy of the candidate and proceed.

To *select a candidate*, look for solved and-nodes in the current and-node, the continuations of which refer to guard_nondet instructions. Select one according to the chosen scheme, for example, the left-most candidate. (See Section 6.7.8 for a some remarks on other possibilities.)

To *copy w.r.t. a candidate*, consider the and-node which is the grandparent of the candidate. Make an isomorphic copy of this node and its attributes, recursively, and insert it to the left of the grandparent, with the following three exceptions: (1) Do not copy the siblings of the candidate—make the copy of the candidate the only alternative. (2) Do not copy variables that are external to the copied node. (3) Do not copy suspensions that refer to dead and-nodes or to the candidate. For each external variable encountered, refer to it also from the copy, and for each of its suspensions that refers to a copied node, add a new suspension that refers to the copy, deleting the original suspension if it refers to the candidate. Finally, unlink the candidate and mark it as dead.

Note that it is permitted to dereference all variable chains during copying, since all local bindings are deinstalled, and the copied subtree is stable. (The single exception is for variables which are used as accumulators in aggregates.)

6.4.12 Collecting

To *collect* the current and-node, change its state to dead, unlink it from alternatives, and set the forward attribute to refer to the parent of the current choice-node. Store a reference to the next instruction in the current and-continuation,

and set the and-attribute in the current context to the top of the stack. If there is a left alternative, install it, otherwise install the right alternative. Proceed.

Note that by putting the continuation in the context of the parent and-node, it will be executed after having promoted the unit or after backing up. Note also that if there is a left alternative, it may be a collect alternative, whereas the right alternative might be the final unit alternative. (See Section 6.6.6 for how these relate to the aggregate statements.) Note, finally, that precisely this moving of an and-task "across" choice-tasks is what makes it necessary to have two separate task stacks instead of just one. (See Section 6.9.2 for a discussion of alternatives.)

6.5 AN INSTRUCTION SET

In the description of this instruction set, familiarity with the architecture of and compilation techniques for the Warren Abstract Machine (WAM) is assumed (such as can be acquired from Aït-Kaci [1991] and Carlsson [1990]).

6.5.1 Overview of Instructions

The instructions are divided into four groups: the *choice*, *and-*, *guard*, and *constraint* instructions. The constraint instructions are further divided into *get*, *put* and *unify* instructions.

Choice instructions are used in the code for choice statements, and-instructions for program atoms, composition, and clauses, and guard instructions for the guard operators.

Choice	*And*	*Guard*
switch_on_tree L_v L_s L_c try L N retry L trust L try_only L	call L N execute L N proceed allocate N deallocate fail	guard_condition guard_commit guard_nondeterminate guard_collect y_i x_j / y_j x_k / y_k guard_unit y_i x_j / y_j guard_top

Figure 6.4. Choice, and-, and guard instructions

Get, put, and unify instructions code tree equations in terms of registers that hold references to variables. Get instructions are used when these registers are already defined, put instructions define the value, and unify instructions are used for sub-trees in both cases.

Get	*Put*	*Unify*

get_variable x_i / y_i x_j get_value x_i / y_i x_j get_symbol S x_i get_constructor F x_i	put_variable x_i / y_i x_j put_value x_i / y_i x_j put_symbol S x_i put_constructor F x_i put_void x_i	unify_variable x_i / y_i unify_value x_i / y_i unify_symbol S unify_void

Figure 6.5. Get, put, and unify instructions

6.5.2 Choice Instructions

switch_on_tree L_v L_s L_c

Dereference x_0. Go (set the program counter) to label L_v, L_s, or L_c depending on its contents (variable, symbol, or constructor, respectively).

try L N

Create a choice-node, insert it at the current insertion point, and enter it. Create an and-node as one of its alternatives. Make it the current and-node and set the insertion pointer to refer to its goals. Push a choice-continuation with the continuation pointer referring to the next instruction (which is retry or trust), and with N a-registers getting their values from corresponding x-registers. Go to L.

retry L

Restore x-registers from corresponding a-registers in the current choice-continuation. Create an and-node and insert it as the last alternative in the current choice-node. Make this the current and-node and set the insertion pointer to refer to its goals. Go to L.

trust L

Do as in retry, but pop the choice-continuation and push a no-choice task before going to L.

try_only L N

Do as in try, but push a no-choice task instead of a choice-continuation.

6.5.3 And-Instructions

call L N

Set the continuation pointer of the current and-continuation to refer to the next instruction. Set the program counter to L. If the wake stack is non-empty, wake at call with N arguments.

execute L N

Set the program counter to L. If the wake stack is non-empty, wake at call with N arguments.

proceed

If the wake stack is non-empty, wake at proceed. Otherwise, proceed.

allocate N

Push an and-continuation with N y-registers.

deallocate

Pop the current and-continuation.

fail

Fail.

6.5.4 Guard Instructions

guard_condition

If the wake stack is non-empty, wake at a guard. If the current and-node is quiet, prune alternatives to the right. If it is left-most, promote simply. Otherwise, suspend it and attempt choice splitting.

guard_commit

If the wake stack is non-empty, wake at a guard. If the current and-node is quiet, prune alternatives and promote simply. Otherwise, suspend it and attempt choice splitting.

guard_nondeterminate

If the wake stack is non-empty, wake at a guard. If the current and-node is solved and determinate, promote. Otherwise, suspend it and attempt choice splitting.

guard_collect y_i x_j/y_j x_k/y_k

If the wake stack is non-empty, wake at a guard. If the current and-node is not quiet, or, if the aggregate is ordered, the current and-node is not left-most, suspend it and attempt choice splitting. Otherwise, move the value of the variable in y_i to x_k (y_k). Create a new variable with the grandparent and-node as home, store it in x_j (y_j), and make it the new value of the variable in y_i. Collect the current and-node.

guard_unit y_i x_j/y_j

If the wake stack is non-empty, wake at a guard. If the current and-node is not quiet, suspend it and attempt choice splitting. Otherwise, move the value of the variable in y_i to y_j (x_j). Promote the current and-node (simply).

guard_top

If the wake stack is non-empty, wake at a guard. Suspend the current and-node and attempt choice splitting.

6.5.5 Tree-Constraint Instructions

get_variable x_i x_j
get_variable y_i x_j

Move the contents of x_j to x_i (y_i).

get_value x_i x_j
get_value y_i x_j

Unify the contents of x_j and x_i (y_i).

get_symbol S x_i

Dereference x_i, and do one of the following depending on the result.

- variable: Bind x_i to S (a reference to a symbol).
- symbol equal to S: Continue.
- other: Fail.

get_constructor F x_i

Dereference x_i, and perform one of the following depending on the result.

- variable: Create a constructor with functor F. Bind x_i to a reference to it. Set the current argument to refer to its first argument. Enter write mode.
- constructor with functor F: Set the current argument to refer to its first argument. Enter read mode.
- other: Fail.

put_variable x_i x_j
put_variable y_i x_j

Create a variable and store a reference to it in x_i (y_i) and x_j.

put_value x_i x_j
put_value y_i x_j

Move the contents of x_i (y_i) to x_j.

put_symbol S x_i

Store S in x_i.

put_constructor F x_i

Create a constructor with functor F and store a reference to it in x_i. Set the current argument to refer to its first argument. Enter write mode.

put_void x_i

Create a variable and store a reference to it in x_i.

unify_variable x_i
unify_variable y_i

In read mode, move the contents of the current argument to x_i (y_i). In write mode, create a variable and store a reference to it in the current argument and in x_i (y_i). In both modes, step to the next argument.

unify_value x_i
unify_value y_i

In read mode, unify the contents of the current argument and the contents of x_i (y_i). In write mode, move the contents of x_i (y_i) to the current argument. In both modes, step to the next argument.

unify_symbol S

In read mode, dereference the contents of the current argument, and do one of the following depending on the result.

- variable: Bind it to S.
- symbol equal to S: Continue.
- other: Fail.

In write mode, store a reference to S in the current argument. In both modes, step to the next argument

unify_void

In write mode, create a variable and store a reference to it in the current argument. In both modes, step to the next argument.

6.6 CODE GENERATION

The code generation principles that follow are not optimal and should not be construed as a compiler. They serve as an introduction, or to confirm what should already be clear to a reader acquainted with compilation of Prolog to the WAM (see, e.g., [Carlsson 1990], which presents this in great detail).

Seen from the clausal point of view, code generation is similar to WAM code generation. The differences are the guard and try_only instructions. The guard instructions are like calls in that variables that should survive them are stored in y-registers. Since choice-nodes and and-nodes have to be created for guard execution, the try_only instruction has to be used when selecting a single clause.

6.6.1 Normal Programs

To simplify the description of code generation, it is assumed that definitions are *normal* in the following sense. Hiding occurs only as hiding over clauses in choice statements, statements in aggregate statements, and over bodies of definitions. The body of a definition is either a choice statement, aggregate statement, or does not contain choice or aggregate statements. The compositions are

then compositions of atoms, and are regarded as sequences composed by an associative operator. It is also assumed that all tree constraints are of the form $v = t$, where v is a variable.

Hiding is normalised by the following transformations (on the abstract syntax), renaming variables as necessary.

$$A, (V : B) \Rightarrow V : A, B$$

$$(V : A), B \Rightarrow V : A, B$$

$$U : (V : A) \Rightarrow U \cup V : A$$

$$U : (V : A) \,\%\, B \Rightarrow U \cup V : A \,\%\, B$$

$$U : A \,\%\, (V : B) \Rightarrow U \cup V : A \,\%\, B$$

A body of a definition properly containing a choice or aggregate statement B is normalised by replacing B with a program atom A, which is given a new name, and the parameters of which are the free variables of B. A definition A := B is then added to the program.

6.6.2 Composition and Chunks

A (normal) composition statement consists of a sequence of *chunks* of the form

$$A_1, \ldots, A_k, B$$

where A_i are constraint atoms and B is a program atom, possibly followed by a final chunk

$$A_1, \ldots, A_k$$

For composition statements enclosed in hiding, we speak of variables that are *parameters* and variables that are *local* (hidden), and otherwise ignore hiding. We may add new local (hidden) variables in transformations. A variable is *temporary* if it is local and occurs only in one of the chunks, or if it is a parameter and occurs only in the first chunk. We speak of *first* and *single* occurrences of variables. For the parameters, (unique) *parameter numbers* are known.

As a preparation for code generation we rewrite constraint atoms and program atoms in a sequence of chunks as follows, repeatedly if necessary.

If v is local and has its first occurrence in a constraint atom

$$v = f(t_1, \ldots, t_n)$$

it is rewritten to

$$u_{i1} = t_{i1}, \ldots, u_{im} = t_{im}, v = f(x_1, \ldots, x_n)$$

where t_{ij} are constructor expressions, u_{ij} are new local variables, and $x_k = u_k$, if $k = i_j$, and $x_k = t_k$, if $k \neq i_j$. Otherwise, it is rewritten to

$$v = f(x_1, \ldots, x_n), u_{i1} = t_{i1}, \ldots, u_{im} = t_{im}$$

with conditions as above.

A constraint $v = t$, where v is single, is rewritten to true. A constraint $u = v$, where v is a temporary variable and u is not, is rewritten to $v = u$. A constraint $v = t$, where neither v nor t are temporary, is rewritten to $(u = v, u = t)$, where u is a new local variable.

A definition of the form

$p(v_1, \ldots, v_N) := V : A$

where i-1 is the parameter number of v_i, may be compiled to code of the form

allocate N
⟨save parameters⟩
⟨code for body⟩
deallocate
proceed

where N is the number of permanent variables. To each permanent variable is assigned a (unique) y-register y_i, for i in 0, …, N–1. Rules for assigning x-registers to temporary variables are given in Section 6.6.8, as constraints on the code generated. For a permanent variable, R_v stands for the y-register which it is already assigned. For a temporary variable, R_v serves as a place-holder, which will be substituted by an x-register.

To save the parameters $v_1, \ldots, v_n$, produce the code

get_variable R_{v1} x_{i1}
…
get_variable R_{vn} x_{in}

where i_j is the parameter number of v_j.

6.6.3 Atoms

A constraint atom

$u = v$

is compiled to

put_variable R_v R_u

if both are first, to

put_value R_v R_u

if first occurrence of u, to

get_variable R_v R_u

if first occurrence of v, and to

get_value R_v R_u

otherwise.

A constraint atom

$v = S$

where S is a symbol, is compiled to

put_symbol S R_v

if v is first, and to

get_symbol S R_v

otherwise.

A constraint atom

$v = f(t_1, \ldots, t_n)$

is compiled to

X_constructor F R_v
Y_1
…
Y_n

where F is f/n, X is put if v is first, otherwise get, and Y_i is

unify_void

if t_i is a variable and single,

unify_variable R_u

if t_i is a variable u and first,

unify_value R_u

if t_i is a variable u and not first, and

unify_symbol S

if t_i is a symbol S.

The constraint true produces no code.

A program atom

$p(v_1, \ldots, v_N)$

is compiled to

Y_1
…
Y_n
call p_N N

where p_N is the label of the code for the definition of p / N, and where Y_i is

put_void x_{i-1}

if v_i is single,

put_variable R_{vi} x_{i-1}

if v_i is first, and

put_value R_{vi} x_{i-1}

otherwise.

6.6.4 Choice

A definition of the form

$$p(v_1, \ldots, v_N) := (\langle \text{clause}_1 \rangle ; \cdots ; \langle \text{clause}_n \rangle)$$

where i-1 is the parameter number of v_i, may be compiled to code of the form

p_N: switch_on_tree L_v L_s L_c
L_v: ⟨try-retry-trust N C_1, …, C_n⟩
L_s: ⟨try-retry-trust N C_{i1}, …, C_{ik}⟩
L_c: ⟨try-retry-trust N C_{j1}, …, C_{jm}⟩
C_1: ⟨code for clause$_1$⟩
…
C_n: ⟨code for clause$_n$⟩

where C_1, …, C_n, L_v, L_s, and L_c are local names that only pertain to the code for the choice statement at hand. Clauses other than C_{i1} to C_{ik} are known to fail (in the guard) if the contents of x_0 is a symbol. Clauses other than C_{j1} to C_{jm} are known to fail (in the guard) if the contents of x_0 is a tree constructor.

A zero length try-retry-trust chain is given as

fail

a chain C_i, of length one, as

try_only C_i

and longer chains, C_{i1}, …, C_{ik} as

try C_{i1} N
retry C_{i2}
…
trust C_{ik}

The try-retry-trust instructions build, incrementally, the choice-box and guarded goals for a choice statement. The switch_on_tree instruction makes a first choice, eliminating clauses that are known to be incompatible. Better clause selection is possible, e.g., by decision graphs [Brand 1994]. The switch_on_tree instruction only illustrates the concept.

6.6.5 Clauses

A clause is very similar to a composition statement. The chunks of a clause are the chunks of the guard and of the body. The parameters are the parameters of the head of the definition in which they occur, which also gives the parameter numbers. The local variables are given in the hiding over the clause, which is otherwise ignored. The notions of first, single, permanent, and temporary are otherwise as for composition.

A clause may be compiled as

```
        allocate N
        ⟨save parameters⟩
        ⟨code for guard⟩
    guard_%
        ⟨code for body⟩
        deallocate
        proceed
```

with the same conditions as for composition.

6.6.6 Aggregates

The treatment of aggregates is somewhat unclean[1], but adequate since it lends itself to simple implementation ("worse is better" [Gabriel 1994]).

Observe that the variable created and stored in x_M is bound to the contents of x_i. The collect instructions will replace the value of this variable repeatedly. The variable itself will not be copied by choice splitting in the aggregate, since it is external.

A definition of the form

$$p(v_1, \ldots, v_M) := \text{aggregate}(u, A, w)$$

where i-1 is the parameter number of v_i, and w is v_j, may be compiled to code of the form

```
p_N:  put_variable x_{M+1} x_M
      get_value x_{M+1} x_{j-1}
      try L_1 M+1
      trust L_2

L_1:  allocate N_1+1
      get_variable y_{N1} x_M
      ⟨save parameters⟩
      ⟨code for A⟩
    guard_collect y_{N1} R_v R_v'
      ⟨code for collect(u, v, v')⟩
      deallocate
      proceed

L_2:  allocate N_2+1
      get_variable y_{N2} x_M
    guard_unit y_{N2} R_v
      ⟨code for unit(v)⟩
      deallocate
      proceed
```

---

[1] Observe the appropriate section number.

The code generated corresponds to that for a choice with two special clauses. In the first are generated the solutions for *u* in A, which are collected by the guard, and the second ends with the unit when there are no more solutions.

The chunks of the collect clause are the chunks in A and in *collect*(*u*, *v*, *v′*). The chunks of the unit clause are the chunks in *unit*(*v*). Variables are classified and allocated accordingly, with the exception that *v* and *v*' have their first occurrences in the guard instructions and that parameters that occur in the unit are not temporary. N_1 and N_2 are the respective numbers of permanent variables.

6.6.7 Initial Statements

For simplicity, we assume that the initial statement is a program atom

initial(v_1, …, v_N)

This is no restriction, since it may be defined as anything.

```
        try_only L
L:      allocate N
        put_variable y0 x0
        …
        put_variable yN-1 xN-1
        call initial_N N
    guard_top
```

The guard_top instruction is given full freedom to report solutions for the variables in this statements, which are stored in y-registers, or other actions that would seem useful. For simplicity, we have given it the semantics of the execution model, where execution continues with other branches.

6.6.8 Register Allocation

The *lifetime* of a temporary variable is the sequence of instructions between, not including, the first and last instructions using R_v.

To *assign x-registers*, for each temporary variable *v*, select a register not used in its lifetime, and substitute it for R_v.

The objectives for this process are to minimise the number of x-registers used and to make possible the deletion of instructions as described in the next section

6.6.9 Editing

Having generated code according to the principles in preceding sections, some editing should be performed to improve execution efficiency.

The following instructions have no effect and can be deleted

get_variable x_i x_i

get_value x_i x_i

put_value x_i x_i

The instruction sequence

```
call L N
deallocate
proceed
```

should be replaced by

```
deallocate
execute L N
```

to ensure that tail recursive programs do not grow the and-stack.

The allocate instruction can be moved to just before the first instruction using a y-register, the first call, or the guard instruction, whichever comes first. The deallocate instruction can be moved to just after the last instruction using a y-register, the last call instruction, or the guard instruction, whichever comes last.

6.6.10 Two Examples

The familiar definition of append

```
append([], Y, Y).
append([H | X], Y, [H | Z]) :-
        append(X, Y, Z).
```

which is written as follows without syntactic sugar

```
append(X, Y, Z) :=
    (   X = [],
        Y = Z
    ?   true
    ;   H, X1, Z1 :
        X = [H | X1],
        Z = [H | Z1]
    ?   append(X1, Y, Z1) ).
```

can be translated to the following code using the principles above. Observe that ./2 is regarded as the functor for list constructors.

```
append_3:
        switch_on_tree Lv Ls Lc
Lv:     try C1 3
        trust C2
Ls:     try_only C1
Lc:     try_only C2

C1:     allocate 0
        get_symbol [] x0
        get_value x1 x2
    guard_nondeterminate
        deallocate
        proceed
```

```
C2:   allocate 3
      get_variable y1 x1
      get_constructor ./2 x0
      unify_variable x1
      unify_variable y0
      get_constructor ./2 x2
      unify_value x1
      unify_variable y2
  guard_nondeterminate
      put_value y0 x0
      put_value y1 x1
      put_value y2 x2
      deallocate
      execute append3
```

The guard instructions are given less indentation for readability.

In particular, code is generated for the second clause

```
      H, X1, Z1 :
      X = [H | X1],
      Z = [H | Z1]
  ?   append(X1, Y, Z1)
```

as follows. The variables X, Y, and Z, are parameters with numbers 0, 1, and 2, respectively. The variables X_1, Y, and Z_1 are classified as permanent and are assigned y-registers 0, 1, and 2, respectively. The variables X, H, and Z are temporary. The code before x-register allocation is

```
      allocate 3
      get_variable RX x0
      get_variable y1 x1
      get_variable RZ x2
      get_constructor ./2 RX
      unify_variable RH
      unify_variable y0
      get_constructor ./2 RZ
      unify_value RH
      unify_variable y2
  guard_nondet
      put_value y0 x0
      put_value y1 x1
      put_value y2 x2
      call append_3 3
      deallocate
      proceed
```

The variables X, H, and Z can now be allocated x-registers 0, 1, and 2, respectively, following the principles of lifetimes. Finally, two of the get_variable in-

structions can be deleted, and the call-deallocate-proceed instructions can be replaced by corresponding deallocate-execute instructions.

A call to bagof, such as the one discussed in Section 6.8.1,

```
p(L, S) := bagof(X, tail(X, L), S)
```

can be translated to the following code using the principles above

```
p_2:   put_variable x3 x2
       get_value x3 x1
       try L1 3
       trust L2

L1:    allocate 2
       get_variable y1 x2
       get_variable x1 x0
       put_variable y0 x0
       call tail_2 2
   guard_collect y1 x0 x1
       get_constructor ./2 x1
       unify_value y0
       unify_value x0
       deallocate
       proceed

L2:    allocate 1
       get_variable y0 x2
   guard_unit y0 x0
       get_symbol [] x0
       deallocate
       proceed
```

assuming that guard_collect is ordered. It is of course perfectly possible to provide both ordered and unordered versions of this instruction for use in different aggregates.

6.7 OPTIMISATIONS

This section lists a few optimisations that are available in the AGENTS system, and others that are believed to be obvious for future efficient implementations.

6.7.1 Flat Guards

Using only the simple machinery introduced so far, the execution speed of the AGENTS system is roughly a factor of four slower, for comparable programs without don't know nondeterminism, than a comparable Prolog implementation (such as SICStus Prolog with emulated code [Carlsson et al. 1993]), and much more memory is needed for execution. This is almost entirely due to the unnecessary creation of choice-nodes, and-nodes, and and-continuations for every choice statement.

However, a few simple instructions and notions of suspending and waking calls almost completely bridge the gap for a wide range of programs. The idea is to short-cut to promotion, suspension, or failure, for guards that only make simple tests on the first argument. This is a special case of the *flat* guards, that only contain constraints (with composition and hiding).

A *suspended call* has the attributes

- *parent*: a reference to an and-node
- *continuation pointer*: a reference to a sequence of instructions
- *a-registers*: a vector of trees
- *goals*: a pair of references to goals

Admit references to suspended calls in place of and-nodes in suspensions, in the wake stack, and in wake tasks.

To *suspend L with N arguments on a variable X*, create a suspended call with L as continuation pointer and N a-registers with contents from corresponding x-registers. Insert it at the insertion point, and add a suspension on the variable X referring to the suspended call.

To *proceed*, if the task is wake task referring to a suspended call, pop it. Install its parent and-node. Restore the program counter and x-registers from the suspended call, set the insertion pointer to its right sibling, unlink it, and decode instructions.

Add the following three instructions.

switch_on_constructor L_f F_1-L_1 … F_n-L_n
switch_on_symbol L_f S_1-L_1 … S_n-L_n

If x_0, dereferenced, is a constructor with functor F_i (is a symbol S_i), go to L_i. Otherwise, go to L_f.

suspend_call L N

Suspend L with N arguments on the variable in x_0. Proceed.

A definition such as

```
q(X) :=
        ( X = a → true
        ; Y: X = f(Y) → q(Y) )
```

which had to be encoded as

```
q_1:    switch_on_tree Lv Ls Lc
Lv:     try C1 1
        trust C2
Ls:     try_only C1
Lc:     try_only C2
```

```
C1:   allocate 0
      get_symbol a x0
  guard_nondet
      deallocate
      proceed

C1:   allocate 1
      get_constructor f/1 x0
      unify_variable y0
  guard_nondet
      put_value y0 x0
      deallocate
      execute q_1 1
```

can now be encoded as

```
q_1:  switch_on_tree Lv Ls Lc
Lv:   suspend_call q_1 1
Ls:   switch_on_symbol Lf a-C1
Lc:   switch_on_constructor Lf f/1-C2
Lf:   fail
C1:   proceed
C2:   get_constructor f/1 x0
      unify_variable x0
      execute q_1 1
```

Example	AGENTS without opt.	AGENTS with opt.	SICStus emulated
nreverse(300)	610 (4.5)	215 (1.6)	137 (1.0)
nreverse(1000)	7499 (2.9)	2433 (0.9)	2612 (1.0)
sort(medi)	394 (3.8)	250 (2.4)	105 (1.0)
sort(maxi)	8613 (4.1)	4031 (1.9)	2077 (1.0)

Figure 6.6. Performance of AGENTS w/wo optimisation vs. SICStus Prolog

The behaviour of the guards is entirely captured by the combination of switch instructions and the suspend_call instruction, and neither choice-nodes, and-nodes, nor and-continuations have to be created.

The impact on the performance of AGENTS for simple benchmarks where this coding of flat guards is applicable is shown in Figure 6.6. The comparison with SICStus only serves to indicate that the execution speed of AGENTS ends up in an acceptable order of magnitude.

The systems compared are AGENTS 0.9, with and without the above flat guard optimisation, and SICStus Prolog 2.1 #8 (emulated code) on a DECstation 5000/240. The times are in milliseconds and include the time for garbage collec-

tion. The benchmarks are the familiar naive reverse, which is called with 300 and 1000 elements, and merge sort by O'Keefe (system sort/2 in SICStus and in AGENTS), which is called with lists of integers of length 1000 (medi) and 11240 (maxi), formed by the decimals of π in groups of five.

The remaining difference in speed between AGENTS and SICStus seems to be mainly due to the more optimised emulator of SICStus. In the case of sort, register allocation in AGENTS is inferior and there are flat guards that cannot be optimised by the simple scheme presented. In both cases, the better ratio for the optimised case with larger inputs is probably due to the use of a copying garbage collector in AGENTS. For the small inputs, no garbage collection is needed.

The optimisation discussed deals only with a very simple case. A scheme for clause selection and quietness detection that optimises all flat guards is given by the decision graph method for arbitrary constraints by Brand [1994].

6.7.2 Small Variables

An unbound variable without suspensions needs only the home and state attributes. We call this a *small variable* (as opposed to *large* variables).

In an implementation with tagged pointers, a small variable can be stored in place of a reference. In particular, small variables can be used as arguments to constructors and in y-registers in and-continuations.

When binding a small variable, it is replaced by a reference to the value. If the variable is external, the home of the variable is still needed. There are two alternatives: (1) replace the small variable with a reference to a new large variable before binding, and (2) trail also the home of the variable. Both are feasible. In the latter case, the home and value attributes can share the same memory location also in the large variables.

If small variables are used as arguments to constructors, a tree built incrementally, by creating incomplete nodes with variable arguments that are later bound, will not contain any superfluous nodes. When they are not, e.g., in a list produced by append, there is one variable “between” every list cell. But, when they are used, care must be taken to make a reference to the argument if a small variable is encountered when accessing arguments.

If small variables are used in y-registers, the variables created for “return values” of calls are stored there. A non-suspending computation employing the flat guard optimisation will only use heap storage for building constructors. But, care must be taken to *globalise* these to avoid dangling references (cf., local variables and unsafe values in the WAM).

AGENTS has small variables, but does not (yet) use them in y-registers.

6.7.3 Separate Read and Write Code Streams

The instructions for tree constraints are suboptimal. Schemes based on separate code streams for reading and writing [Mariën and Demoen 1991] are more ef-

ficient and make it possible always to bind variables to complete tree constructors, instead of partially built tree constructors as in the get_constructor and put_constructor instructions. The latter feature solves two problems with the code described.

AGENTS provides *generic variables*, which are used to parameterise the system with different constraint systems. These are equipped with a method table, and when bound call a unify method. In principle, it is possible to generalise the constraint system of trees, e.g., to records [Smolka and Treinen 1992] or to RT [Keisu 1994], by introducing generic variables for these domains that "understand" what is means to be unified with the special case of a tree. However, for this it is necessary that variables are bound to complete tree constructors.

A parallel version of the abstract machine has to take care when binding variables. A common technique (e.g., [Crammond 1990]) is to atomic-swap the state and value attributes of the variable with the new value, and, if it was discovered that the variable was already bound, to unify the old and new values. For this it is also necessary to bind to complete tree constructors.

6.7.4 Trimming and Re-using And-Continuations

Non-tail-recursive definitions grow the and-stack. As described, the call instructions do not perform "trimming" of and-continuations, as of environments in the WAM. This would be easy to introduce, in particular since trimming can take place unconditionally. It is possible both to shrink and grow and-continuations as needed.

Tail-recursive definitions are in every sense like loops, but the and-continuations are pushed and popped in every cycle. This is shared with the WAM. Meier [1991] and others have suggested that environments (corresponding to and-continuations) should be reused, in the way a stack frame normally survives the entire procedure invocation in a procedural language. For this, Prolog requires determinacy analysis, since and-continuations are involved in nondeterminism, and may have to be shared between solutions to goals in the "loop". In AKL, and-continuations are never shared (not even in the sharing scheme discussed in the following section) and a corresponding scheme can be designed and implemented with less difficulty.

6.8 COPYING

The WAM shares parts of the goal clause between the branches in a search tree, using the trail, the choice-point stack, and backtracking. The abstract machine described here solves this by copying. Naturally, nondeterminism thus provided cannot be used like that of Prolog.

6.8.1 A Truly Bad Case

Observe, in Figure 6.7, the cost in AGENTS of enumerating all tails of a list using the following definition. This is intentionally a very bad case.

```
tail(X, X).
tail(X, [_ | Y]) :- tail(X, Y).
```

The testing environment is as for the benchmarks in Section 6.7.1. The times presented (in milliseconds) are the average of three consecutive runs of

```
bagof(X, tail(X, L), S)
```

A reference to S is live during execution. The heap is not reset between runs. The total is divided into times for copying, garbage collection, and other.

Length	Total	Copy	% Copy	GC	% GC	Other	% Other
300	309	237	77	38	12	34	11.1
600	1098	832	76	208	18	64	5.8
900	2518	1886	75	553	22	74	2.9
1200	4455	3441	77	878	20	137	3.1
1500	7268	5537	76	1560	21	171	2.4
1800	12293	8438	69	3583	29	272	2.2

Figure 6.7. AGENTS: Cost of enumerating all tails for different length lists

Length	Failure	Findall	Findall2	GC	% GC
300	3.3	334	333	0	0
600	6.0	1307	1283	0	0
900	9.6	2877	4058	1154	28
1200	12.9	5048	8247	3129	38
1500	15.6	7927	19769	11833	60
1800	18.6	11475	20946	9250	44

Figure 6.8. SICStus: Cost of enumerating all tails for different length lists

We now compare this with SICStus Prolog (Figure 6.8). The times (in milliseconds) are the averages of three runs. “Failure” is the total execution time (including GC) for a failure-driven loop of the form

```
tail(_, L), fail
```

“Findall” is the runtime for a goal of the form

```
findall(X, tail(X, L), S)
```

The heap is reset between runs, and there is no garbage collection. “Findall2” is the time when the heap is not reset, and garbage collection becomes visible.

AGENTS is, for this truly bad case, between two and three orders of magnitude slower than SICStus with plain failure-driven enumeration, and grows quadratically in the length of the list. The cost of activities other than copying and garbage collection grows more or less linearly with the length of the list, as could be expected.

In AGENTS, the elements are enumerated using bagof. Since copying has already been done, this carries no extra cost, whereas SICStus exhibits a dramatic, but non-surprising, slowdown when going from plain failure-driven enumeration to findall. (SICStus bagof is almost twice as slow.)

6.8.2 Some Better Cases

For problem solving purposes, the copying cost seems to be manageable, and stays around 40–65% of the total execution time (as illustrated by Figure 6.9).

The examples are: queens(N) is the solution for the N-queens problem presented in Section 3.5, using a centre out heuristic. money(dif) is the SEND+MORE=MONEY cryptarithmetic puzzle. zebra(cir) is the Five Houses puzzle. substitution and frequency are solutions to a substitution cipher from Yang [1989]. knights(6) and allknights(5) find one and all solutions to the knight's tour problem on boards of size 6 and 5, respectively. The programs are rather naïve in that they use tree constraints only. Solutions based on AGENTS with FD-constraints are quite competitive with CLP systems in speed, and have a similar copying overhead [Carlson, Haridi, and Janson 1994].) The columns are as before, with a new column for the number of copying steps performed.

Example	Total	Copy	%Cp	GC	%GC	Other	%Oth	Cp:s
queens(4)	16	4	25	0	0	12	75	1
queens(8)	89	31	35	7	8	51	57	4
queens(16)	608	286	47	120	20	202	33	8
money(dif)	260	172	66	9	3	79	31	152
zebra(cir)	88	55	62	5	6	28	32	29
substitution	1758	1186	67	263	15	309	18	182
frequency	229	127	55	47	21	54	24	45
knights(5)	131	38	29	6	5	87	66	24
knights(6)	973	518	53	80	8	375	39	201
allknights(5)	15870	4015	25	997	6	10858	69	3213

Figure 6.9. Copying cost for some problem solving programs

The reason for the reasonable behaviour of problem solving programs is that for most guessing steps most or all of the copy is actually used. This is particu-

larly true for all-solutions search. In one-solution search, if the solution is found early, more copying will be wasted.

6.8.3 A General Sharing Scheme

A remedy for the copying overhead could be a more general sharing scheme for machine states of the type described here. Such ingenious simplicity as that of the sharing scheme of the WAM cannot be hoped for. Already in the transition from plain Prolog execution to execution based on the Basic Andorra Model, as in Andorra-I [Santos Costa, Warren, Yang 1991a], a sharing scheme has to consider updates of objects other than variables. A sharing scheme for AKL, as discussed below, has to deal with a hierarchy in which a new sharing point can be introduced both "above" and "below" existing sharing points.

A complete scheme has not been designed, but the principles discussed here will probably apply. For simplicity, we assume a transition system over trees built of and- and or-nodes. Some transitions copy subtrees. For example,

$$\text{and}(A, \text{or}(B, C)) \Rightarrow \text{or}(\text{and}(A, B), \text{and}(A, C))$$

copies the subtree A. This is similar to the choice splitting rule of AKL. The idea of sharing is that the representation of A will be shared between the newly formed branches.

The first problem is how to make an update of a shared subtree specific to some part of the tree.

The differences between different occurrences of shared subtrees can be maintained in *diff trails*, which are associated with, but are not parts of, and-nodes.

A *diff* has the attributes

- *updated*: reference to a location
- *old*: old contents of location
- *new*: new contents of location

When entering an and-node, the diffs are *installed*, storing the new value in all updated locations. When exiting, the diffs are *deinstalled*, restoring the old value.

When an object that is shared with respect to some ancestor or-node is updated, a corresponding diff is associated with an ancestor and-node of the object that is in the or-node. Which and-node to choose is not obvious. Two possibilities are: (1) the and-node closest to the object and (2) the and-node closest to the or-node. In the first, installation is delayed until working close to the object. In the second, installation is performed once and for all while working in this branch. It is also conceivable to mix these with delayed deinstallation of diffs, where a diff could be installed upon entry of (1), but deinstalled upon exit of (2). For locations that are multiply updated, the scheme can be optimised to avoid multiple diff trailing, e.g., by marking the updated location with a time-stamp for the update. Observe that the diffs trails themselves are subject to sharing.

The next problem is how to distinguish shared subtrees from non-shared subtrees. An approximate scheme is described.

All objects in a tree that are subject to sharing are given *time-stamps*. A time-stamp is a number. Upon creation, or-nodes are given an *age of shared data*, which is equal to the greatest time-stamp in the part to become shared. The *max age* inside an or-node is the maximum age of shared data of it and all ancestor or-nodes. A new object is given a time-stamp that is greater than the max age where it is placed. Thus, any object which has a greater time-stamp than max age is not shared, and can be updated without diff trailing. Other objects may or may not be shared, and should be trailed for safety.

The time-stamp of an object may be represented naively using a slot in the object. A more involved scheme could use the actual address of the object. In both cases, a chunk of objects may be assigned a time-stamp as a group.

Clearly, a scheme of this kind carries a considerable cost. It is not unlikely that it would add an overhead on the execution times of many programs without don't know nondeterminism of, say, 25-50%, but as indicated by Figures 6.7, 6.8, and 6.9, this might pay off for nondeterministic programs, and for an AKL implementation mainly intended for such tasks, a sharing scheme should be considered. For AGENTS, the cost of copying is deemed acceptable, as don't know nondeterminism is mainly intended for constraint programming.

Length	Total	Copy	% Copy	GC	% GC	Other	% Other
300	59	34	57	0.0	0.0	26	43
600	117	65	55	3.2	2.7	49	48
900	166	90	54	6.2	3.7	70	42
1200	228	123	54	7.6	3.3	98	43
1500	290	148	51	16.2	5.6	126	43
1800	366	188	52	35.8	9.8	141	39

Figure 6.10. AGENTS: Cost of enumerating all tails with simple sharing

6.8.4 A Simple Sharing Scheme for Trees

Simply by adding a home attribute to tree constructors, and not copying trees which have constructors external to the copied and-node as roots since it is known that they cannot contain local variables, the truly bad case can be considerably improved. This scheme is available as an option in AGENTS, and the new times are shown in Figure 6.10. The testing conditions are exactly the same as in Section 6.8.1.

Execution time is linear in the length of input data. The execution time is more than one order of magnitude faster than that of SICStus for the same task.

6.8.5 Avoid Copying

An alternative to sharing is to avoid the need for copying. Potentially useful techniques include better selection of candidates, consistency checking, and constraint lifting (also known as constructive disjunction) [Abreu, Pereira, and Codognet 1992; Van Hentenryck, Saraswat, and Deville 1992; Moolenaar and Demoen 1994].

When choice splitting is attempted, the representation of potential candidates is quite explicit. They contain a list of constraints, and the constraints contain variables on which there are suspensions. A candidate can be given a weight according to how many variables it constrains and how many suspensions there are on these, factors that are likely to be correlated with a greater chance of failure of alternatives in sibling choice-nodes, and thereby less nondeterminism.

It is also possible to examine alternatives in sibling choice-nodes before copying, to see if there is some sibling that would fail for the constraints of the current candidate, in which case the candidate can fail without copying. Pertinent siblings can be found via the suspensions.

Finally, all alternatives in a nondeterminate choice-node can be examined, and if they share (i.e., all entail) a constraint, and this constraint is known to cause waking, it can be *lifted* to the parent and-node, instead of copying.

6.9 POSSIBLE VARIATIONS

A few, somewhat speculative, variations of the abstract machine are described and discussed. These variations await practical evaluation.

6.9.1 Eager vs. Lazy Waking

The execution model and the abstract machine presented employ *eager waking* in that the number of steps before waking is bounded. AGENTS 0.9 employs a *lazy waking* scheme. Waking takes place in the guard instructions only, where it is useful to establish whether all (currently known) work in an and-box has been performed. Even lazier waking is conceivable, for example to wake only when all other work available has been performed.

The abstract machine could be changed for lazy waking as follows. Add a wake attribute to contexts, which is set to the top of the wake stack at choice-node entry, analogously to the other attributes. When waking, wake only nodes (and suspended calls) in the current context. Finally, wake only in guard instructions.

Observe that there is no need for resume or restore insertion point tasks in this scheme. There are also other advantages as well as disadvantages.

The advantage of lazy waking is mainly that a producer is not interrupted only because a consumer is interested. In the list-sum example of Chapter 2, the ea-

ger waking model presented here will exhibit “thrashing” behaviour when faced with a goal of the form

```
sum(L, S), list(N, L)
```

The sum call will suspend, since there is no list. The list call will produce one list cell, and then suspend to wake the sum call, which will consume the single list cell and then suspend, and so on. This will be extremely inefficient, at least an order of magnitude slower in a native coded implementation. A lazy waking model does not have this problem, since the producer would complete its task uninterrupted.

The advantage of eager waking is mainly that when there is interaction, a process will reply to a request early, thereby giving the client input to its subsequent work, for which a more expensive suspension might occur if input was not present. For example, if write(Term, InStream, OutStream) suspends unless InStream is known and waking is lazy, a process that first tries to obtain a stream from a server and then proceeds to write all the elements in a list will suspend one call for each element in the list, since the server will not respond eagerly.

Thus, lazy waking is, in some sense, more stable, but is not so intuitive since it can be expected that a process will reply to a request as soon as possible. This is related to fairness (discussed in Section 5.9.1). An experienced programmer can control eager waking to achieve the desired effects and avoid thrashing, but can also properly synchronise programs in the lazy waking scheme. Programs that are written to work well in the lazy scheme are “better” in that they can be expected to work reasonably well with many different execution models.

6.9.2 Incremental vs. Batch Aggregates

As described, unless the generating goal suspends, aggregates will stack up all collect statements before executing them in a “batch”. In the case of the example in Section 6.8.1, the list produced was not made available to consumers, which could have reduced the overhead of GC by consuming the list as it was produced. Incremental execution can be achieved as follows.

Introduce a new choice-task *resume aggregate*, with attributes

- *aggregate*: a reference to a choice-node
- *beyond*: a reference to a choice-task

To *collect* the current and-node, change its state to dead, unlink it from alternatives, and set the forward attribute to refer to the parent of the current choice-node. Then, push a resume aggregate task, with the current choice-node as aggregate, and the choice attribute of the current context as beyond. Mark the grandparent of the current and-node as unstable. Finally, pop the current context, set the current and-node to its grandparent, and decode instructions from the next instruction.

Note that the resume aggregate task is found when backing up, after failure or after suspension. Choice splitting is inhibited explicitly. The and-node is unstable, since the aggregate is not completely executed.

To *back up*, if it is a resume aggregate task, do as follows. If the current and-node (containing the aggregate) is dead, remove all choice-tasks to and including beyond, and back up. Otherwise, install the first alternative in the aggregate referred to, set choice in the current context to beyond, pop the task, and proceed.

This scheme does not in anyway interfere with other execution. It is quite possible to distinguish incremental aggregates from other aggregates, and compile the former using special incremental guard_collect instructions. All tasks pertaining to the and-node containing the aggregate will be executed, and thus also any consumers it contains. An obvious disadvantage is that it, quite unnecessarily, suspends this and-node before resuming the aggregate, and that it has to be marked as unstable. The ideal position for this task would be as the last and-task, but (1) this position cannot be reached, as it is part of a sequence of instructions preceding the guard, and (2) it would involve moving any number of preceding tasks.

A different scheme can be achieved by placing the resume aggregate task on the and-stack after the and-continuation corresponding to the collect statement. There is no “room” there, but the continuation could be moved, since it is likely to be small, or room can be reserved for the task “in” the continuation. By so doing, there is no need for marking the and-node as unstable, but only consumers suspended on the output of the aggregate will be given a chance to execute. This scheme also does not interfere with other execution.

6.9.3 Message-Oriented Scheduling

In the abstract machine presented, agents are compiled more as procedures that expect input and produce output, than as objects, the normal state of which is suspended and which receive intermittent communication.

There is a compilation scheme based on a different point of view called *message-oriented scheduling* [Ueda and Morita 1992]. If it is known that a message sent on a certain stream has only one recipient, the recipient and the stream can be compiled in such a manner that sending a message is analogous to invoking a method of an object. For FGHC, this knowledge requires modes, since it is not otherwise known where streams are produced and consumed. For streams connected to ports (see Chapter 7), it can be easy to know, since the stream is produced by the port.

Messages to port-based built-in objects in the AGENTS system are handled as method calls, which gives a flavour of eager waking. It is probably undesirable to mix different styles of waking, since this complicates the mental model for programmers. Thus, message-oriented scheduling should probably be combined with eager waking.

6.10 RELATED WORK

Much work has been done on abstract machines for Prolog, the committed choice languages, and some on combinations. The committed-choice languages are often flat, or, if the guards are deep, do not use hierarchical binding environments (constraint stores for tree equality constraints). We distinguish between *goal-based* models, that create explicit representations of all constituents of a composition (in AKL terminology), and *continuation-based* models, that create them implicitly via a continuation, as in the abstract machine described here.

Miyazaki, Takeuchi, and Chikayama [1985] describe a sequential interpreter for Concurrent Prolog, with deep guards and hierarchical binding environments. The computation state is a tree of AND- and OR-processes, which correspond to choice- and and-nodes. Their *shallow binding scheme* is very close the one employed here. The model is goal-based.

Crammond [1990] reports on a parallel abstract machine for PARLOG (the JAM) and a corresponding implementation, which supports deep guards and is comparable to SICStus Prolog in speed. However, as an optimisation, it allows only one active deep guard at a time. There may be either one deep guard in clauses composed by parallel clause composition, or any number of deep guards in clauses composed by sequential clause composition. The model is goal-based. Since PARLOG does not distinguish between local and external variables, there is no need to support a hierarchical binding environment.

The trend for the “pure” committed-choice languages is strongly converging to light-weight flat languages and implementations that compile to C, such as jc [Gudeman, De Bosschere, and Debray 1992] and KLIC [Chikayama, Fujise, and Yashiro 1993].

Yang [1987] describes a parallel implementation scheme for P-Prolog, a language offering a combination of concurrent execution and don't know nondeterminism, using determinism for synchronisation. A rather involved scheme for parallel execution is described which focuses on the treatment of bindings in a parallel context, allowing no direct comparison.

Santos Costa, Warren, and Yang [1991a; 1991b] present Andorra-I, a Prolog implementation based on the Basic Andorra Model (BAM, see Section 8.1.8) which provides both dependent-and and or-parallelism on the Sequent Symmetry. The model is goal based. Execution states are not hierarchical, but share linked lists of processes in an or-tree. These lists are semi-double-linked, a technique which, compared to full double-links, reduces the need for locking when manipulating lists in a parallel implementation, and reduces the need for trailing when shared nodes are updated. This technique is difficult to adapt to the present context due to its incremental construction and suspension of goals. (But other techniques have similar effects [Montelius and Ali 1994].)

Palmer and Naish [1991] describe NUA-Prolog, an and-parallel implementation of Prolog based on the BAM, otherwise similar to Andorra-I, but without a pre-processor guaranteeing consistency with sequential Prolog behaviour.

Bahgat [1991] describes an abstract machine for Pandora, an extension of PAR-LOG based on the BAM, an extension of the JAM with a choice-point stack for sharing. The model does not maintain the order of goals, which are not in linked lists. Instead, programmer intervention is required for the choice of a goal for a don't know nondeterministic step.

CHAPTER 7

PORTS FOR OBJECTS

Can object-oriented programming be realised effectively in AKL? This chapter is an introduction to the concept of *ports*. It is argued that ports are highly useful for object-oriented programming in a concurrent logic programming language. It is also illustrated by examples how to use ports to emulate other communication schemes for concurrent programming.

7.1 INTRODUCTION

We regard *objects* for concurrent logic programming languages as processes, as first proposed by Shapiro and Takeuchi [1983], and later extended and refined in systems such as Vulcan [Kahn et al 1987], A'UM [Yoshida and Chikayama 1988], and Polka [Davison 1989] (Figure 7.1).

Some of these systems are embedded languages that only make restricted use of the underlying language. We are interested in full-strength combinations of the underlying language with an expressive concurrent object-oriented extension, with all the problems and opportunities this entails.

In this chapter we introduce *ports*, an alternative to streams, as communication support for object-oriented programming in concurrent constraint logic programming languages. From a pragmatic point of view, ports provide efficient many-to-one communication, object identity, means for garbage collection of objects, and opportunities for optimised compilation techniques for concurrent objects. It will also provide us with means for mixing freely objects and other data structures provided in concurrent logic programming languages. From a semantic point of view, ports preserve the monotonicity of the constraint store, which is a crucial property of all concurrent constraint languages.

In the next few sections we present some of the background of our work. First we examine some requirements on object-oriented systems. Then we discuss the notion of a communication medium, and review a number of proposals that do not meet our requirements.

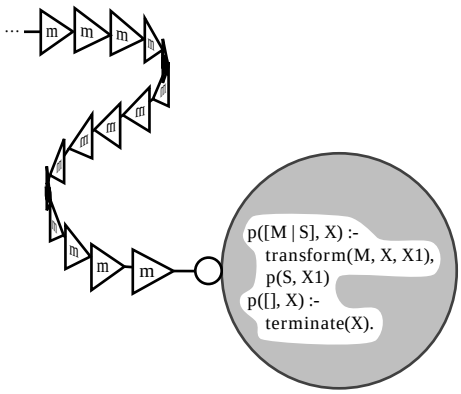


Figure 7.1. Objects as stream-consuming recursive processes

In the following sections, we introduce ports as a solution to these problems, one which also adds entirely new possibilities, such as a simple approach to optimised compilation of objects. We will also show that the Exclusive-read, Exclusive-write PRAM model of parallel computation can be realised quite faithfully in terms of space and time complexity using ports. This indirectly demonstrates that arbitrary parallel algorithms can be expressed quite efficiently. Finally, a number of examples illustrate how the functionality of many other languages is captured by ports.

7.2 REQUIREMENTS

Our starting point is a number of requirements on object-oriented languages, and we will let these guide our work. In so doing, we here only consider requirements on the object-based functionality, including requirements on the interaction between the host language and the concurrent object-oriented extension. Other aspects of object-oriented languages, such as inheritance, can be handled in many different ways (e.g., [Goldberg and Shapiro 1992]).

Since our goal is the integration of concurrent object and logic programming, we conform to the tradition of languages such as C++, where the object-oriented aspects are added to the underlying language and, in particular, allow objects and other data structures to be mingled freely. Thus, in a logic programming language, it should be possible to have objects *embedded* in a term data structure (Figure 7.2, next page). Since terms can be shared freely, this will allow concurrent objects to share, for instance, an array of concurrent objects.

Higher-level object-oriented languages provide automatic *garbage collection* of objects that are no longer referenced (Figure 7.3, next page). Programmers do not have to think about when objects are no longer in use, nor do they have to deallocate them explicitly. The high-level nature of logic programming languages makes it desirable to provide garbage collection of objects, just as of other data structures.

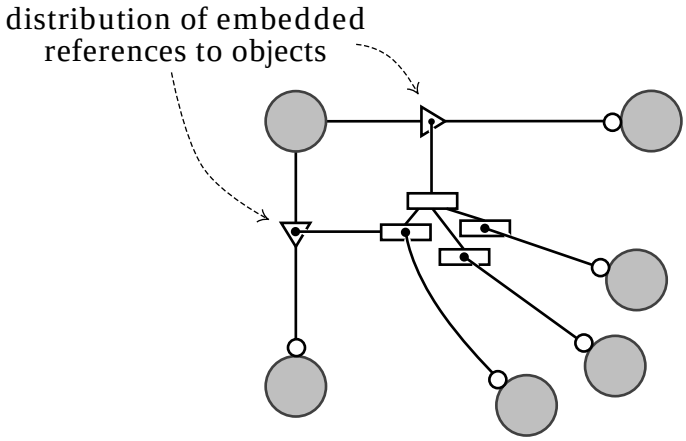


Figure 7.2. Objects embedded in terms

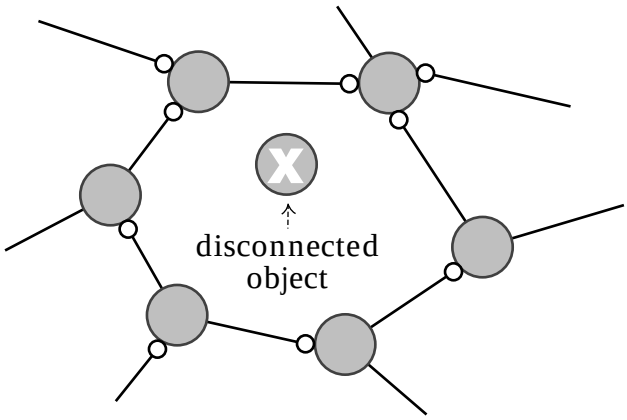


Figure 7.3. Garbage-collectable disconnected object

Note that, in the goal-directed view of logic programming, an object is a goal, which has to be proved; it cannot just be thrown away. However likely the assumption that a goal is provable without binding variables, it cannot be taken for granted. In a concurrent object-oriented setting, another dimension is added, in that an object may still be active, and affect its environment, although it is no longer referred to by other objects. Even if there are no incoming messages, an object may wish to perform some cleaning up before being discarded. Thus, garbage collection of objects in concurrent (logic) programs should involve notifying an object that it will no longer receive messages. It is up to the object to decide to terminate.

In addition, an implementation of objects should provide

- *light-weight* message sending and method invocation, the cost of which should preferably be similar to that of a procedure call,
- *compact* representation of objects, the size of which should be dominated by the representation of instance variables,
- *conservation* of memory, which means that a state transition should only involve modifying relevant instance variables—objects are reused.

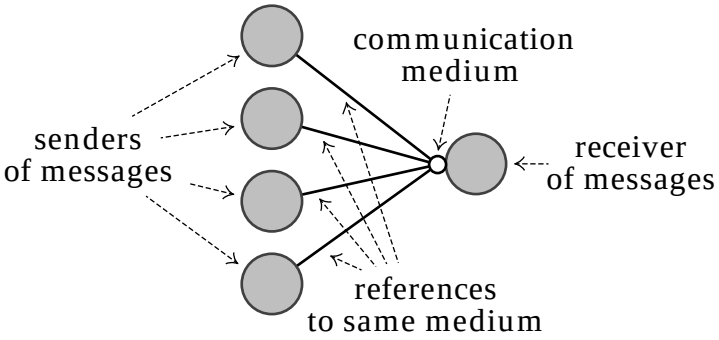


Figure 7.4. The communication medium

This chapter will address all but the last of the above requirements. The last requirement is generally solved in concurrent logic programming languages by providing a mechanism for detecting single references, and reusing the old instance variables.

7.3 COMMUNICATION MEDIA

A *communication medium* is a data type that carries messages between processes acting as objects (Figure 7.4).

The medium is used as a handle to an object, and is regarded as the *object identifier* from a programmer's point of view.

In the concurrent constraint view that we take, the communication medium is managed (described and inspected) using constraints. All constraints are added to a shared *constraint store*. To *send* a message means to impose a constraint on the medium which allows a process to detect the presence of a message, and *receive* the message by inspecting its properties. A sender of messages is a *writer* of the medium. A receiver of messages is a *reader* of the medium. A message that is received once and for all is said to be *consumed*. A medium can be *closed* by imposing constraints that disallow additional messages. A receiver can detect that a medium is closed.

An important property of the constraint store in concurrent (constraint) logic programming languages is that it is monotone. Addition of constraints will produce a new constraint store that entails all the information in the previous one. This property is important because it implies that once a process is activated by the receipt of a message, this activation condition will continue to hold until the message is consumed, regardless of the actions of other processes.

The discussion above leads us to the following requirements on our communication medium.

- A solution should preserve the monotonicity of the constraint store.
- The number of operations required to send a message (make it visible to an end receiver) should be constant (for all practical purposes), independent of the number of senders. All senders should be given equal oppor-

tunity, according to a first come first served principle. We call this the *constant delay property*.

- When a part of the medium that holds a message has been consumed, it should be possible to deallocate or reuse the storage it occupied, by garbage collection or otherwise.
- To provide completely automatic garbage collection of objects, it should be possible to apply the closing operation automatically (when the medium is no longer in use).
- To enable sending multiple messages to embedded objects, it should be possible to send multiple messages to the same medium.

The last requirement seems odd in conventional object-oriented systems. It is however a problem in all single-assignment languages including the current (constraint) logic programming languages.

In the remainder of this section we discuss a number of communication media that have been proposed for concurrent (constraint) logic programming languages.

7.3.1 Streams

The list is by far the most popular communication medium in concurrent logic programming. In this context lists are usually called *streams*. A background to streams and techniques for binary merging (see below) is given by Shapiro and Mierowsky [1984].

A message m is sent on the stream S by binding S to a list pair, $S = [m \mid S']$. The next message is sent on the stream S'. A stream S is closed by making it equal to the *empty list*, $S = []$. A receiver, which should be waiting for S to become equal to either a list pair or the empty list, will then either successfully match S against a list pair $[M \mid R]$, whereupon M will be made equal to m and R to S', in which case the next message can be received on S', or match S against [], in which case no further messages can be received. By the *end-of-stream* we mean a tail of the stream that is not yet known to be a list pair or an empty list.

To achieve the effect of several senders on the same stream, there are two basic techniques: (1) The stream is split into several streams, one for each sender, which are interleaved into one by a *merger* process. (2) The language provides some form of atomic “test-and-set” unification, which allows *multiple writers* to compete for the end-of-stream.

Merging is typically achieved either by a tree of binary mergers, or by a multiway merger. The binary-merge tree is built by splitting a stream as necessary (Figure 7.5).

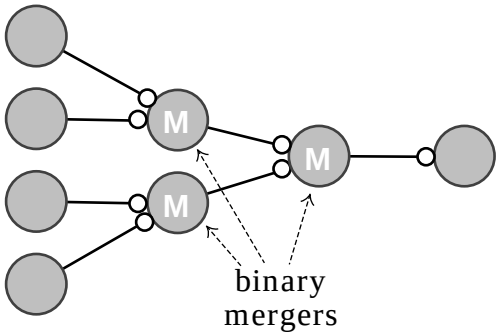


Figure 7.5. Binary merge network

Clearly, this technique does not have the constant delay property as the (best case) cost is O(log *m*) in the number of senders *m*. A *multiway merger* is a single merger process that allows input streams to be added and deleted dynamically (Figure 7.6).

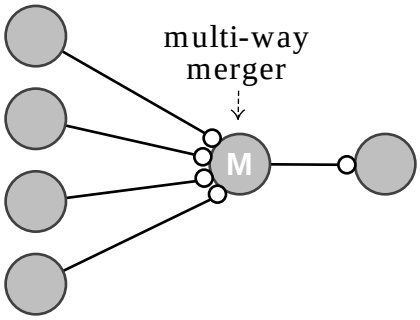


Figure 7.6. Multiway merger

A *constant delay multiway merger* cannot be expressed in most concurrent logic programming languages (AKL is an exception), but it is conceptually clean, and it is quite possible to provide one as a language primitive. We will assume that all multiway mergers have constant delay.

A number of disadvantages of merging follow.

- A merger process has to be created whenever there is the slightest possibility that several senders will send on the same stream. For many purposes this is not a problem. Once a multiway merger is created, adding and deleting input streams is fairly efficient. Yet, needing one feels like an overhead in an object-oriented context, where references to objects should be freely distributable.
- Explicit closing of all streams to all objects is necessary, since otherwise, either the program will eventually deadlock, or some objects will continue to be suspended, forever occupying storage.
- The merger process itself occupies storage, it also wastes storage when creating a new merged stream, and if it uses the standard mechanisms for suspension, it is likely to be (comparatively) inefficient.

- Messages have to be sent on the current end-of-stream variable, which is changed for every message sent. Assume that a process sends messages on streams which are embedded in a data structure, e.g., a tree containing objects. When a message has been sent on one stream, the new end-of-stream variable has to be "put back" into the tree. Usually this means building a new version of the tree, with the new end-of stream in place of the old, possibly reusing some unaffected parts of the old tree but necessarily allocating some new nodes. (However, if some form of single reference optimisation is employed by the language implementation new parts may reuse old storage: e.g., [Chikayama and Kimura 1987; Kahn and Saraswat 1990; Foster and Winsborough 1991]). Even worse, if this tree is to be shared with another process, where the possibility of the other process sending messages on the embedded streams cannot be excluded, two copies of the tree have to be created (allocating new nodes for at least one). All the streams have to be split in two (by merging), one for each copy.
- A serious problem is the *transparent message-distribution problem*. A message is usually a term $m(C_1, \ldots, C_n)$ where the C_i's are message components. Suppose we want to implement a transparent message-distributor object, which when it receives a message, of any kind, will distribute it to a list of other objects. Without prior knowledge of the components of messages, the distributor object cannot introduce the merging required for stream components.

An advantage with merging is that it allows list pairs to be reused in the merger and deallocated by the receiver as soon as a message has been consumed. In some cases, in a system with fairly static object-structure, explicit closing of all streams as a means for controlled termination of objects can be considered an advantage. Another general advantage of all stream communicating systems is the implicit sequencing of messages from a source object to a destination. This simplifies synchronisation in many applications.

Multiple writers can only be expressed in some languages with atomic "test-and-set" unification. The drawbacks of multiple writers are summarised as follows:

- The cost for multiple writers is typically O(m) per message, when there are m senders, and is therefore even further from the constant delay property than merging.
- The delay is proportional to the number of messages that have been sent.
- An inactive sender may hold a reference to parts of the stream that have already been consumed by the intended receiver, making deallocation impossible.
- It is difficult to close a stream. Some termination detection technique, such as *short-circuiting* (see, e.g., [Shapiro 1986a]), has to be used. In practice, this outweighs the advantages of multiple writers.

An advantage of multiple writers is that no merger has to be created. Moreover, several messages can be sent on the same stream, and not only by having explicit access to the end-of-stream variable.

Neither merging nor multiple writers provides a general solution to the problem of automatic garbage collection of objects. There are only special cases, as exemplified by A'UM [Yoshida and Chikayama 1988].

7.3.2 Multiway Merging

A constant delay merger written in AKL is presented in this section, as an indication of its expressiveness, and to illustrate a novel programming technique: bagof with feedback.

The way multiway mergers are normally used, a process requests a new input stream from the merger on one of the existing input streams, either by sending a special message which is caught by the merger, e.g., S = [new_stream(S_1) | S_2], or by generalising streams to *stream trees* using an additional constructor, e.g., S = split(S_1, S_2), which is the solution adopted here.

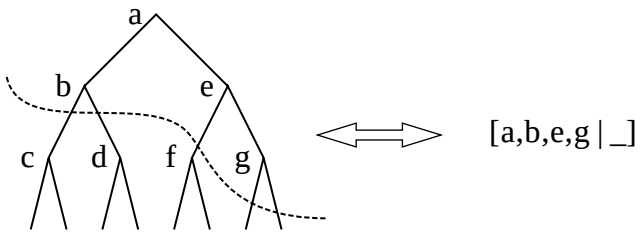


Figure 7.7. A stream tree with corresponding merged stream

In Figure 7.7, the relation between a (stylised) stream tree and a corresponding merged stream is shown. It is assumed that the merger has considered the messages up to the border drawn. Other messages (c, d, f) have been sent, i.e., are visible in the constraint store. It is easy to write a merger satisfying the first requirement using binary mergers, but the delay is then unbounded. The second requirement means that there should be a constant *k*, such that the number of computation steps required to detect the presence of and send the three known messages c, d, and f, should not exceed 3*k*.

To prepare the way for the complete solution, a simpler *ether* agent is shown first, which does not preserve the order of messages on input streams.

It is called with an input stream tree and an output stream. Ether uses unordered bagof to collect messages on the input streams (Figure 7.8).

```
ether(T, S) :=
        unordered_bagof(M, tree_member(M, T), S).
```

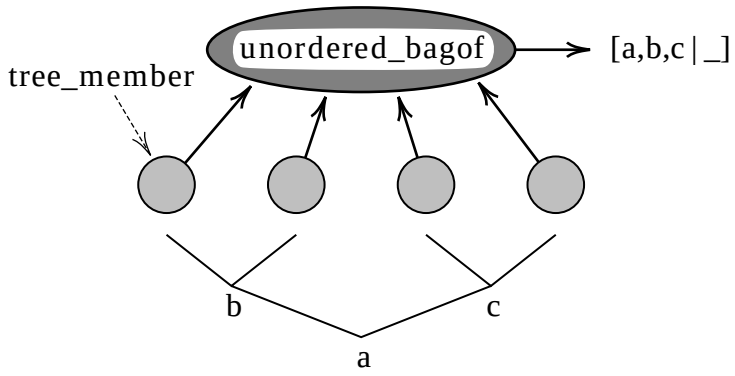


Figure 7.8. Ether merging

```
tree_member(M, split(S1, S2)) :-
    → ( true ? tree_member(M, S1)
      ; true ? tree_member(M, S2) ).
tree_member(M, [M1 | S]) :-
    → ( M = M1 ? true
      ; true ? tree_member(M, S) ).
tree_member([], Y) :-
    → fail.
```

The declarative reading of tree_member is that the message in its first argument is a member of the tree formed by split nodes and stream nodes in its second argument. The unordered bagof agent will collect these members to the output stream, when they become available along the fringe of this tree.

However, since the elements of an input stream will appear as different solutions, the unordered bagof may collect them in an arbitrary order, and thus their order is not preserved on the output stream. To remedy this, two processes are necessary, one bagof-based process to collect input streams, and one that guarantees sequentialisation of messages.

The improved merger consists of two co-operating agents: (1) a bagof agent that collects streams on which messages have been sent, and (2) a server that for each stream either removes a message (and feeds the rest to the generator in the bagof agent), splits the stream (and feeds the two new streams to the generator), or closes the stream (Figure 7.9).

```
merger(S0, S) :=
        server(B, A, S, 1),
        unordered_bagof(I, generator(I, [S0 | A]), B).
```

The generator process enters the streams as alternative solutions for the bagof agent. It constrains the streams to be known to have one of the possible forms, [], [_ | _], or split(_,_), before they are passed to the server.

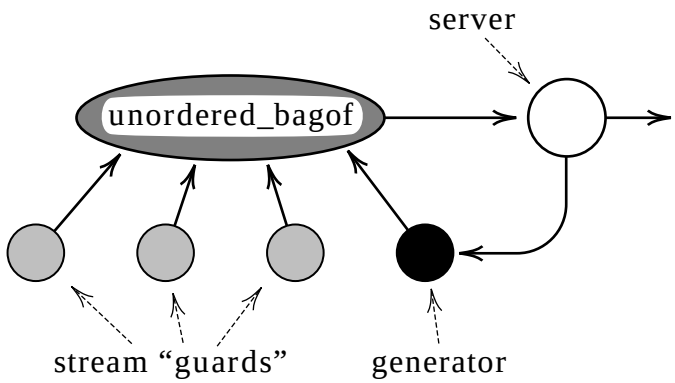


Figure 7.9. Constant delay multiway merger in AKL

```
generator(I, [S | R]) :-
    → (    true
         ? I = S,
           ( I = [] → true
           ; I = [_,_] → true
           ; I = split(_,_) → true )
       ;   true
         ? generator(I, R) ).
```

The server agent expects a stream of instantiated streams from the bagof agent. Apart from dealing with these in the above mentioned way, it also keeps track of the number of merged streams, and closes the output stream and terminates the bagof agent when none remain.

```
server([[E | R] | B], A, S, N) :-
    |   S = [E | S1],
        A = [R | A1],
        server(B, A1, S1, N).
server([split(S1,S2) | B], A, S, N) :-
    |   A = [S1,S2 | A1],
        server(B, A1, S, N+1).
server([[] | B], A, S, N) :-
    |   server(B, A, S, N-1).
server(B, A, S, 0) :-
    |   A = [],
        S = [].
```

The first clause deals with the “normal” case, when a message has been sent on the input stream. The message is added to the output stream, and the rest of the input stream is fed back to the generator. The second clause deals with the case when an input stream is split in two. Both are fed back to the generator, and the number of input streams is incremented by one. The third clause deals with the case when an input stream is closed. The number of input streams is decremented by one. The fourth clause deals with the case when there are no remain-

ing input streams. The output stream is closed, and [] is fed back into the generator to thereby terminate the bagof.

This section presented an implementation of a multiway merger in AKL, but does not provide new solutions to the problems discussed in the previous section. It can be suspected that the problems of merging and multiple-writers are inherent in streams, and people have therefore looked for alternative data types.

7.3.3 Mutual References

Shapiro and Safra [1986] introduced *mutual references* to optimise multiple writers, and as an implementation technique for constant delay multiway merging. The mental model is that of multiple writers.

A shared stream *S* is accessed indirectly through a *mutual reference Ref*, which is created by the allocate_mutual_reference(*Ref, S*) operation. Conceptually, the mutual reference *Ref* becomes an alias for the stream *S*. The message sending and stream closing operations on mutual references are provided as built-in operations. The stream_append(*X*, *Ref, New_Ref*) operation will bind the end-of-stream of *Ref* to the list pair [*X* | *S'*], returning *New_Ref* as a reference to the new end-of-stream. The stream *S* can be closed using the close_stream(*Ref*) operation, which binds the end-of-stream to the empty list [].

An advantage of this is that the mutual reference can be implemented as a pointer to the end-of-stream. When a message is appended, the pointer is advanced and returned. If a group of processes are sending messages on the same stream using mutual references, they can share the pointer, and sending a message will always be an inexpensive, constant time operation. Mutual references can be used to implement a constant delay multiway merger. Another advantage is that an inactive sender will no longer have a reference to old parts of the stream. This makes it possible to deallocate or reuse consumed parts of the stream.

A disadvantage is that we cannot exclude the possibility that the stream has been constrained from elsewhere, and that stream_append has to be prepared to advance to the real end-of-stream to provide multiple writers behaviour.

Otherwise, mutual references has the advantages of multiple writers, but the difficulty of closing the stream remains. It is also unfortunate that the mental model is still that of competition instead of co-operation.

7.3.4 Channels

Tribble et al [1987] introduced *channels* to allow multiple readers as well as multiple writers.

A channel is a partially ordered set of messages. The write(M, C_1, C_2) operation imposes a constraint that: (1) the message M is a member of the channel C_1, and (2) M precedes all messages in the channel C_2. The read(M, C_1, C_2) operation selects a minimal (first) element M of C_1, returning the remainder as the chan-

nel C_2. The empty(C) operation tests the channel C for emptiness. The close(C) operation imposes a constraint that the channel C is empty.

In the intended semantics, messages have to be labelled to preserve message multiplicity, and only minimal channels (without superfluous messages) satisfying the constraints are considered.

Channels seem to share most of the properties of multiple writers on streams. Thus, all messages have to be retained on an embedded channel, in case it will be read in the future. An inactive sender causes the same problem. Closing is just as explicit and problematic. The multiple readers ability can be achieved by other means. For example, a process can arbitrate requests for messages from a stream conceptually shared by several readers.

7.3.5 Bags

Kahn and Saraswat [1990] introduced *bags* for the languages Lucy and Janus.

Bags are multisets of messages. There is no need for user-defined merging, as this is taken care of by the Tell constraint bag-union $B = B_1 \cup \cdots \cup B_n$. A message is sent using the Tell constraint $B = \{m\}$. A combination of these two operations, $B = \{m\} \cup B'$, corresponds to sending a message on a stream, but without the order of messages being given by the stream. A message is received by the Ask constraint $B = \{m\} \cup B'$.

Note that bags can be implemented as streams, with a multiway merger as bag-union. Therefore, it is not surprising that bags have most of the disadvantages of streams with multiway merging. The host languages Lucy and Janus only allow single-referenced objects and therefore suffer less from these problems. It is possible to reuse list pairs in the multiway merger, and to deallocate (or reuse) list pairs when a bag is consumed.

7.4 PORTS

We propose *ports* as a solution to the problems with previously proposed communication media. In this section, we first define their behaviour. We then discuss their interpretation. Finally, we describe their implementation.

7.4.1 Ports Informally

A *port* is a connection between a bag of messages and a corresponding stream of these messages (Figure 7.10, next page). A bag which is connected via a port to a stream is usually identified with the port, and is referred to as a port.

The open_port(P, S) operation creates a bag P and a stream S, and connects them through a port. Thus, P becomes a port to S. The send(M, P) operation adds a message M to a port P. A message which is sent on a port is added to its associated stream with constant delay. When the port becomes garbage, its associated stream is closed. The is_port(P) operation recognises ports.

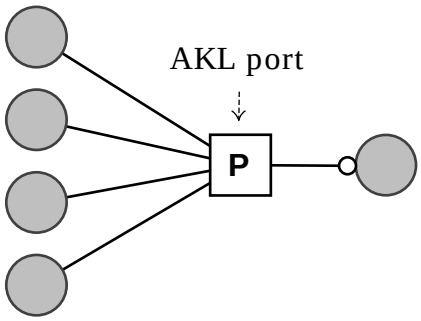


Figure 7.10. An AKL port

A first simple example follows. Given the program

```
p(S) := open_port(P, S), send(a, P), send(b, P).
```

calling

```
p(S)
```

yields the result

```
S = [a,b]
```

Here we create a port and a related stream, and send two messages. The order in which the messages appear could just as well have been reversed.

Ports solve all of the problems mentioned for streams and others.

- No merger has to be created; a port is never split.
- Several messages can be sent on the same port, which means that ports can be embedded.
- Message sending delay is constant.
- Senders cannot refer to old messages and thus prevent garbage collection.
- A port is closed automatically when there are no more potential senders.
- The transparent forwarding problem is solved, since messages can be distributed without inspection.
- Messages can be sequenced, as described in Section 7.5.

7.4.2 Ports as Constraints

We can provide a sound and intuitive interpretation of ports as follows. Ports are bags (also called multisets). The open_port and send operations on ports are constraints with the following reading. The open_port constraint states that all members in the bag are members in the stream and vice versa, with an equal number of occurrences. The send constraint states that an object is a member of the bag. The is_port constraint states that $\exists S$ open_port(*P*, *S*) for a bag *P*.

Our method for finding a solution to these constraints is don't care nondeterministic. Any solution will do. Therefore, the interpretation in terms of constraints is not a complete characterisation of the behaviour of ports, just as

Horn clauses do not completely characterise the behaviour of commit guarded clauses. In particular, it does not account for message multiplicity, nor for their "relevance", i.e. it does not "minimise" the ports to the messages that appear in a computation.

A logic with resources could possibly help, e.g., Linear Logic [Girard 1987]. It can easily capture the don't care nondeterministic and resource sensitive behaviour of ports. The automatic closing requires much more machinery. If such an exercise would aid our understanding remains to be seen.

The method is also incomplete. A goal of the form

$$\text{open\_port}(P_1, S_1), \text{open\_port}(P_2, S_2), P_1 = P_2$$

or of the form

$$\text{open\_port}(P_1, S_1), (\text{open\_port}(P_2, S_2), P_1 = P_2 \rightarrow \ldots)$$

cannot be solved. With the constraint interpretation, this would amount to unifying S_1 and S_2, but to be able to do this it is necessary to keep the whole list of messages sent throughout the lifetime of a port. Instead, this situation may be regarded as a runtime error.

Goals of the form

$$\text{open\_port}(P, S), (\text{send}(M, P) \rightarrow \ldots)$$

or of the form

$$\text{send}(M, P), (\text{open\_port}(P, S) \rightarrow \ldots)$$

can also not be solved. These situations do not make sense pragmatically, and are left unsolved. Both cases can be detected in an implementation, and could be reported as a runtime errors.

The AKL computation model does not, however, in its present form support don't care nondeterministic or incomplete constraint solving. An alternative approach is to describe the behaviour of port operations by *port reduction rules*, which assign unique identifiers to ports, and only make use of constraints for which it is reasonable for constraint solving to be complete [Franzén 1994]. This corresponds to the behaviour of the AGENTS implementation of ports, and is also the approach taken in the following formalisation.

7.4.3 Port Reduction Rules

Following Franzén [1994], we define the operational semantics of ports in terms of rewrites on port operations and on (parts of) the local constraint store of an and-box, to which correspond D-mode transitions in the AKL computation model. The syntactic categories are extended so that port operations may occur in the position of a program atom in programs and in configurations.

In the following, the letter *n* stands for a natural number.

A constraint

$$\text{port}(v, n, w, w')$$

means that v is a port with identifier n, associated list w, and w' is a tail of w. (To conform exactly with the syntax for constraints in the computation model, the parameter n would have to be expressed as a variable constrained to be equal to a number, but this is relaxed here.)

Observe that the following rules strictly accumulate information in the constraint store; the right hand side always implies the left hand side.

The first rule opens a port.

$$\textbf{and}(\text{R}, \text{open\_port}(v, w), \text{S})_{\text{V}}^{\sigma} \underset{\chi}{\overset{\text{D}}{\Rightarrow}} \textbf{and}(\text{R}, \text{S})_{\text{V}}^{\text{port}(v, n, w, w)\wedge\sigma}$$

where n is a closed term that does not occur in R, S, or χ.

The second rule enters a constraint that recognises ports.

$$\textbf{and}(\text{R}, \text{is\_port}(v), \text{S})_{\text{V}}^{\sigma} \underset{\chi}{\overset{\text{D}}{\Rightarrow}} \textbf{and}(\text{R}, \text{S})_{\text{V}}^{\exists u \exists w \exists w' \text{ port}(v, u, w, w')\wedge\sigma}$$

The third rule consumes a send to a port, moving the message to its associated stream. Observe that this rule monotonically adds information to the constraint store. Although the send constraint is removed, it is still implied by the presence of the message in the stream. Observe that the constraint store is composed by the associative and commutative conjunction operator ($\wedge$), and that constraint atoms may be reordered as appropriate for the application of rules.

$$\textbf{and}(\text{R}, \text{send}(u, v'), \text{S})_{\text{V}}^{\text{port}(v, n, w, w')\wedge\sigma} \underset{\chi}{\overset{\text{D}}{\Rightarrow}} \textbf{and}(\text{R}, \text{S})_{\text{V}}^{\text{port}(v, n, w, w'')\wedge w'=[u \mid w'']\wedge\sigma}$$

if $\sigma\wedge\text{env}(\chi)$ entails $v = v'$.

The fourth and final rule closes the associated stream when the port only occurs in a single port constraint.

$$\textbf{and}(\text{R})_{\text{V}}^{\text{port}(v, n, w, w')\wedge\sigma} \underset{\chi}{\overset{\text{D}}{\Rightarrow}} \textbf{and}(\text{R}, \text{S})_{\text{V}}^{w'=[]\wedge\sigma}$$

For this to become (nontrivially) applicable, it is necessary to have "garbage collection" rules. A simplification rule is given for a constraint theory **TC** with port constraints and rational tree equality constraints of the form $X = Y$ or $X = f(Y_1, \ldots Y_n)$, where variables may be ports. (For a discussion about such combinations of theories, see [Franzen 1994].)

A *garbage collection rule* for this theory is

$$\textbf{and}(\text{R})_{\text{V}}^{\sigma} \underset{\chi}{\overset{\text{D}}{\Rightarrow}} \textbf{and}(\text{R})_{\text{V}}^{\theta}$$

where the constraint atoms in θ is a strict subset of those in σ such that

$$\textbf{TC} \vdash \text{env}(\chi) \supset (\exists \text{W}\sigma \equiv \exists \text{W}\theta)$$

where W contains all variables in U not occurring in R, and where the set V contains all variables in U that occur in θ or R. This rule is clearly terminating, since there is a finite number of constraint atoms in a local constraint store.

For example, if χ is λ, R is p(X), σ is X = f(Y) ∧ Z = g(W,X), and U is {W, X, Y, Z}, then θ is X = f(Y) and V is {X,Y}, since

TC ∃W∃Y∃Z (X = f(Y), Z = g(W, X)) ≡ ∃W∃Y∃Z (X = f(Y))

This should correspond to our intuition for garbage collection.

7.4.4 Implementation of Ports

The implementation of ports, and of objects based on ports, gives us other advantages, which are first discussed in this section and then returned to in Section 7.5.2.

The implementation of ports can rely on the fact that a port is only read by the open_port/2 operation, and that the writers only use the send/2 and is_port/1 operations on ports, which are both independent of previous messages.

Therefore, there is no need to store the messages in the port itself. It is only necessary that the implementation can recognise a port, and add a message sent on a port to its associated stream. This can be achieved simply by letting the representation of a port point to the stream being constructed. In accordance with the port reduction rules, adding a message to a port then involves getting the stream, unifying the stream with a list pair of the message and a new "end-of-stream", and updating the pointer to refer to the new end-of-stream. Closing the port means unifying its stream with the empty list. Note that the destructive update is possible only because the port is "write only".

In this respect, ports are similar to mutual references. But, for ports there is conceptually no such notion as *advancing* the pointer to the end-of-stream. We are constructing a list of elements in the bag, and if the list is already given, it is unified with what we construct.

Other implementations of message sending are conceivable, e.g. for distributed memory multiprocessor architectures.

That a port has become garbage is detected by garbage collection, as suggested by the definition. If a copying garbage collector is used, it is only necessary to make an extra pass over the ports in the old area after garbage collection, checking which have become garbage (i.e., were not copied). Their corresponding streams are then closed.

From the object-oriented point of view, this is not optimal, as an object cannot be deallocated in the first garbage collection after the port becomes garbage, which means that it survives the first generation in a two-generation generational garbage collector. Note that for some types of objects, this is still acceptable, as their termination might involve performing some tasks, e.g. closing files. For other objects, it is not. In the next section we discuss compilation tech-

niques based on ports that allow us to differentiate between these two classes of objects, and treat them appropriately.

Reference counting is more incremental, and is therefore seemingly nicer for our purposes, but the technique is inefficient, it does not rhyme well with parallelism, and it does not reclaim cyclic structures. MRB [Chikayama and Kimura 1987] and compile-time GC (e.g., [Foster and Winsborough 1991]) are also of limited value, as we often want ports to be multiply referenced.

7.5 CONCURRENT OBJECTS

Returning to our main objective, object-oriented programming, we develop some programming techniques for ports, and discuss implementation techniques for objects based on ports.

Given ports, it is natural to retain the by now familiar way of expressing an object as a consumer of a message stream, and use a port connected to this stream as the object identifier.

```
create_object(P, Initial) :=
        open_port(P, S),
        object_handler(S, Initial).
```

In the next two sections we address the issues of synchronisation idioms, and of compilation of objects based on ports, as above.

7.5.1 Synchronisation Idioms

We need some synchronisation idioms. How do we guarantee that messages arrive in a given order? We can use continuations, as in Actor languages.

The basic sequencing idiom is best expressed by a program. In the following, the call / 1 agent is regarded as implicitly defined by clauses

$$call(p(X_1, \ldots, X_n)) :- \rightarrow p(X_1, \ldots, X_n).$$

for all p / *n* type (program and constraint) atoms in a given program.

```
open_cc_port(P, S) :=
        open_port(P, S0),
        call_cont(S0, S).

call_cont([], S) :-
    → S = [].
call_cont([(M & C) | S0], S) :-
    → S = [M | S1],
        call(C),
        call_cont(S0, S1).
call_cont([M | S0], S) :-
    → S = [M | S1],
        call_cont(S0, S1).
```

The (*Message* & *Continuation*) operator guarantees that messages sent because of something that happens in the continuation will come after the message. For example, it can be used as follows.

```
open_cc_port(P, S), send((a & Flag = ok), P), p(Flag, P).
```

The agent p may then choose to wait for the token before attempting to send new messages on the port P.

The above synchronisation technique using continuations can be implemented entirely on the sender side, with very little overhead. A goal send((*m* & *C*), *P*) is compiled as (send(*m*, *P*), *C*), with the extra condition that *C* should only begin execution after the message has been added to the stream associated with the port. It should be obvious that this is trivial, even in a parallel implementation.

Another useful idiom is the three-argument send, defined as follows.

```
send(M, P0, P) :=
    send((M & P = P0), P0).
```

which is useful if several messages are to be sent in sequence. If this is very common, the send_list operation can be useful.

```
send_list([], P0, P) :-
  → P = P0.
send_list([M | R], P0, P) :-
  → send(M, P0, P1),
    send_list(R, P1, P).
```

Both, of course, assume the use of the above continuation calling definition.

7.5.2 Objects based on Ports

If the object message-handler consumes one message at a time, feeding the rest of the message stream to a recursive call, e.g., as guaranteed by an object-oriented linguistic extension, then it is possible to compile the message handler using message oriented scheduling [Ueda and Morita 1992]. Instead of letting messages take the indirect route through the stream, this path can be shortcut by letting the message handling process pose as a special kind of port, which can consume its messages directly. There is then no need to save messages to preserve stream semantics. It is also easy to avoid creating the “top-level structure” of the message, with suitable parameter passing conventions. The optimisation is completely local to the compilation of the object.

Looking also at the implementation of ports from an object-oriented point of view, an object compiled this way poses as a port with a customised send-method. This view can be taken further by also providing customised garbage collection methods that are invoked when a port is found to have become garbage. If the object needs cleaning up, it will survive the garbage collection to perform this duty, otherwise the GC method can discard the object immediately.

Objects in common use, such as arrays, can be implemented as built-in types of ports, with a corresponding built-in treatment of messages. This may allow an efficient implementation of mutable data-structures. Ports can also serve as an interface to objects written in foreign languages.

Ports and built-in objects based on ports are available in AGENTS [Janson et al 1994]. An interesting example is that it provides an AKL engine is as a built-in object. A user program can start a computation, inspect its results, ask for more solutions, and, in particular, reflect on the failure or suspension of this computation. This facility is especially useful in programs with a reactive part and a (don't know nondeterministic) transformational part, where the interaction with the environment in the reactive part should not be affected by nondeterminism or failure, as exemplified by the AGENTS top-level and some programs with graphical interaction. In the future, this facility will also be used for debugging of programs and for metalevel control of problem solving.

In AGENTS, the open_port/2 operation is a specialised open_cc_port/2, which does not accept general continuations, but supports the send/3 operation.

7.6 PRAM

Shapiro [1986b] discusses the adequacy of concurrent logic programming for two families of computer architectures by simulating a RAM (Random Access Machine) and a network of RAMs in FCP (Flat Concurrent Prolog). However, a simulator for shared memory multiprocessor architectures, PRAMs (Parallel RAMs), is not given.

We conjecture that PRAMs cannot be simulated in concurrent logic programming languages without ports or a similar construct. This limitation could, among other things, mean that array-bound parallel algorithms, such as many numerical algorithms, cannot always be realised with their expected efficiency in these languages.

In the following we will show the essence of a simulator for an Exclusive-read Exclusive-write PRAM in AKL using ports.

7.6.1 PRAM with Ports

A memory cell is easily modelled as an object.

```
cell(P) :=
        open_cc_port(P, S),
        cellproc(S, 0).
```

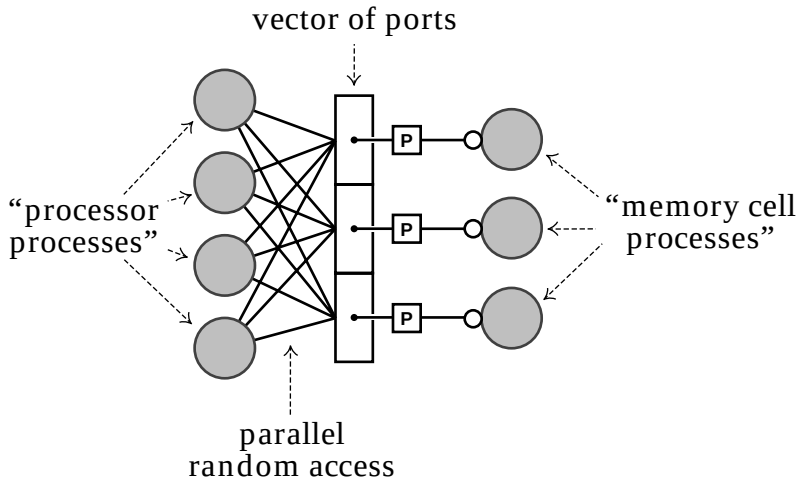


Figure 7.11. PRAM with AKL ports

```
cellproc([], _) :-
    → true.
cellproc([read(V0) | S], V) :-
    → V0 = V,
      cellproc(S, V).
cellproc([write(V) | S], _) :-
    → cellproc(S, V).
cellproc([exch(V1, V2) | S], V) :-
    → V2 = V,
      cellproc(S, V1).
```

PRAM is achieved by creating an array of ports to cells (Figure 7.11).

```
memory(M) :=
      M = m(C1, …, Cn),
      cell(C1), …, cell(Cn).
```

Any number of processes can share this array and send messages to its memory cells in parallel, updating them and reading them. The random access is achieved through the random access to slots in the array, and the fact that we can send to embedded ports without updating the array. Sequencing is achieved by the processors, using continuations as above.

7.6.2 RAM without Ports

Most logic programming languages do not even allow modelling RAM, as a consequence of the single-assignment property. Shapiro's simulator for a RAM [1986b] depends on a built-in *n*-ary stream distributor to access cell processes in constant time as above. In KL1 the MRB scheme allows a vector to be managed efficiently, as long as it is single-referenced [Chikayama and Kimura 1987].

```
memory(M) := new_vector(M, n).
```

A program may access (read) the vector using the

vector_element(*Vector*, *Position*, *Element*, *NewVector*)

operation (which preserves the single-reference property). Sequencing is achieved through continuing access on *NewVector*. Similarly, a program may modify (write) the array using the

set_vector_element(*Vector*, *Position*, *OldElement*, *NewElement*, *NewVector*)

operation (which also preserves the single-reference property). Sequencing can be achieved as above.

7.6.3 PRAM without Ports?

If MRB (or *n*-ary stream distributors) and multiway merging are available, they can be used to model PRAM, but with a significant memory overhead.

Each processor-process is given its own vector (or *n*-ary stream distributor) of streams to the memory cells. All streams referring to a single memory cell are merged. Sequencing is achieved as above. Thus we need O(*n m*) units of storage to represent a PRAM with *n* memory cells and *m* processors.

The setup of memories is correspondingly more awkward.

memories(M_1, …, M_m) :=
    memvector(M_1, C_{11}, …, C_{1n}),
    …,
    memvector(M_m, C_{m1}, …, C_{mn}),
    cell(C_1), …, cell(C_n),
    merge(C_{11}, …, C_{1m}, C_1),
    …,
    merge(C_{n1}, …, C_{nm}, C_n).

cell(S) :=
    cellproc(S, 0).

memvector(M, C_1, …, C_n) :=
    new_vector(M_1, *n*),
    set_vector_element(M_1, 1, _, C_1, M_2),
    …,
    set_vector_element(M_n, *n*, _, C_n, M).

Memory is accessed and modified as in a combination of the two previous models (Figure 7.12). A stream to a memory cell is accessed using the KL1 vector operations. A message for reading or writing the cell is sent on the stream, and the new stream is placed in the vector. An isomorphic structure can also be achieved using *n*-ary stream distributors.

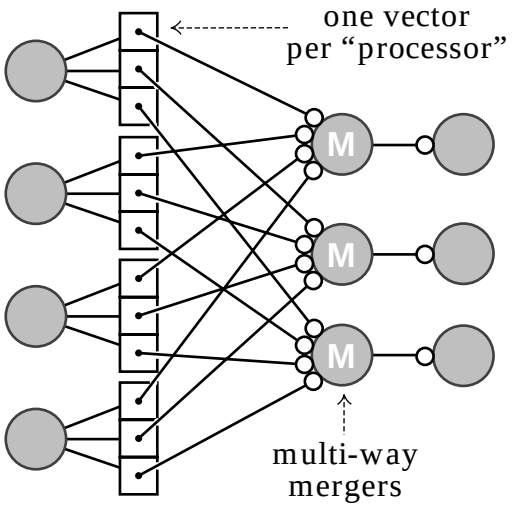


Figure 7.12. PRAM without AKL ports

7.7 EXAMPLES

The first two examples, which are due to Barth, Nikhil, and Arvind [1991], exhibit the need for *parallel random access* functionality in a parallel programming language. The two examples, histogramming a tree of samples and graph traversal, exemplify basic computation structures common to many different settings.

Barth, Nikhil, and Arvind contrast random access solutions with pure functional programs, showing clearly that the former are an improvement both in terms of the total number of computation steps and in terms of the length of the critical path (in “maximally” parallel executions). The compared programs can be expressed in AKL with and without ports, respectively. Only the parallel random access solution with ports is shown; its alternatives without ports can be expressed in many different ways.

The next two sections discuss Lisp and Id, which are functional languages extended with side effect operations, and suggest how these extensions can be simulated in AKL using ports.

The last three sections discuss concrete formalisms for concurrent programming, Actors, Linda, and Erlang, and show how their particular modes of communication can be realised in AKL using ports.

7.7.1 Histogramming a Tree of Samples

Given a binary tree in which the leaves contain numbers in the range 1, …, *n*, count the occurrences of each number.

In our solution, the counts for the numbers are in a PRAM memory of the kind defined above, which we assume has indices in the given range. The program traverses the tree, and, for each number found, increments the value in the cell

with this index. The short circuit guarantees that all nodes have been counted before returning the memory.

```
hist(T, M) :-
        memory(M0),
        count(T, M0, S0, S),
        ( S0 = S → M = M0 ).

count(leaf(I), M, S0, S) :-
    → arg(I, M, C),
        send((exch(K, K+1) & S = S0), C).
count(node(L,R), M, S0, S) :-
    → count(L, M, S0, S1),
        count(R, M, S1, S).
```

7.7.2 Graph Traversal

Given a directed graph in which the nodes contain unique identifiers and numbers, compute the sum of the numbers in nodes reachable from a given node. Assume that the identifiers are numbers in the range 1, …, *n*. Any number of computations may proceed concurrently on the same graph.

In our solution, the nodes which have been traversed are marked in a separate array. For simplicity we assume this to have the indices in the given range, whereas a better solution would employ hashing to reduce its size. Assume that the memory agent returns a memory of this size. A graph node is an expression of the form

```
node(I, K, Ns)
```

where I is the unique node identifier, K is a number (to be summed), and Ns is a list of neighbouring nodes. (Note that cyclic structures are not a problem in the constraint system of rational trees.)

```
sum(N, G, S) :=
        memory(M),
        traverse(N, G, M, S).

traverse(node(I, K, Ns), M, S) :-
        arg(I, M, C),
        send(exch(1, X), C),
        (       X = 1
            →   S = 0
        ;       traverse_list(Ns, M, S) ).

traverse_list([], _, S) :-
    → S = 0.
traverse_list([N | Ns], M, S) :-
    → traverse(N, M, S1),
        traverse_list(N, M, S2),
        S = S1 + S2.
```

7.7.3 Lisp

Many functional languages, such as Standard ML, CommonLisp, and Scheme, support various forms of *side effect* operations. Whether or not such practices are regarded with favour by purists, they have allowed the integration of object-oriented capabilities, e.g., CLOS into CommonLisp, which substantially enhance Lisp's expressive power.

AKL models side effects using *ports*. If it is assumed that assignment is confined to variables and to explicitly declared objects, an efficient translation of such Lisp programs into AKL is possible, which apart from allowing reuse of old code may also be seen as a parallelisation technique for Lisp programs.

Assume given a translation of the functional subset of Lisp along the lines presented in the previous section. Assume further that an "object-oriented" extension provides operations of the form "(put-field *x v*)" and "(get-field *x*)" for the assignable fields of its objects. Clearly, these operations have to be serialised in the order given by sequential execution of Lisp, but other operations can remain unaffected.

The translation of a Lisp expression of the form

(r … (p …) (q …))

will yield an AKL program of the form

…, p(…, R_1), q(…, R_2), r(…, R_1, R_2, R_3), …

If we assume that p, q, and r perform side effect operations, these can be serialised by adding two new arguments, chained in the order of execution of the original Lisp program, e.g.,

…, p(…, R_1, T_0, T_1), q(…, R_2, T_1, T_2), r(…, R_1, R_2, T_2, T_3), …

These arguments are used to pass a token between side effecting operations; each of these will wait for the token, perform the operation, and then pass the token along. Assuming that assignable fields in objects are memory cells with continuation-calling ports as defined in Section 7.5.1, the put-field and get-field operations may be written as

```
put_field(record(…, X, …), V, token, T) :-
    → send((write(V) & T = token), X).

get_field(record(…, X, …), V, token, T) :-
    → send((read(V) & T = token), X).
```

It should be quite obvious how to proceed to a general translation of such features, and no space will be wasted on this exercise here.

As an aside, the concept of *monads* provides a clean explanation of some well-behaved forms of side effect programming (among many other things) [Wadler 1990]. It remains to be seen if it suffices to explain, e.g., CLOS.

7.7.4 Id

In the taxonomy of Barth, Nikhil, and Arvind [1991], there are three approaches to parallel execution of functional languages:

> *Approach A*: purely functional (strict or non-strict) with implicit parallelism or annotations for parallelism.
>
> *Approach B*: strict functional, sequential evaluation, imperative extensions, and threads and semaphores for parallelism.
>
> *Approach C*: non-strict functional, implicitly parallel evaluation, M-structures, and occasional sequentialisation.

The style of functional programming discussed in Section 3.4.1 provided parallelism in a manner adhering to Approach A. Erlang, discussed for its process-oriented features below, is reminiscent of Approach B. M-structures are available in the parallel non-strict functional language Id, which is the only instance of approach C [Barth, Nikhil, and Arvind 1991]. Note that the approach presented in the previous section does not fall within any of these categories.

M-structures give Id the ability to express arbitrary PRAM-algorithms, just as ports do for AKL. The port examples shown above were modelled upon corresponding Id programs, using techniques that do not immediately correspond to M-structures. In this section, we show that M-structures can easily be expressed in terms of ports.

An *M-field* is a value cell in a structured data type, such as a record or an array. Such a cell is either *full* or *empty*. The *take* operation atomically reads the contents of a full cell and sets its state to empty. If the cell is empty, the take operation suspends. The *put* operation atomically stores a value in an empty cell and sets its state to full. If there are suspended take operations, one is resumed. If the cell is full, the put operation suspends. A data structure that contains M-fields is called an *M-structure*.

An M-field can be modelled in AKL as follows.

```
make_mfield(P) :=
        open_port(P, S),
        separate(S, Ps, Ts),
        match(Ps, Ts).

separate([], Ps, Ts) :-
    → Ps = [], Ts = [].
separate([put(U) | S], Ps, Ts) :-
    → Ps = [U | Ps1],
        separate(S, Ps1, Ts).
separate([take(V) | S], Ps, Ts) :-
    → Ts = [V | Ts1],
        separate(S, Ps, Ts1).
```

```
match([], _) :-
    | true.
match(_, []) :-
    | true.
match([U | Ps], [V | Ts]) :-
    | V = U,
      match(Ps, Ts).
```

The state of the M-field (*full* or *empty*) is implicit in the program. If there are available put requests, then it is full, otherwise it is empty.

Since Id provides implicit parallelism, the order of take and put operations is quite undefined. The synchronisation provided by these operations is often sufficient, but sometimes general *sequencing* is desirable and Id provides a sequencing operator for this purpose. Here short-circuiting would be appropriate for a general translation of Id, whereas *continuations* (Section 7.5.1) are sufficient for most programming purposes.

Finally, note that the implementation scheme for ports described in Section 7.4.4 allows simple low-level optimisations, by which the efficiency of the implementation of a port-based object such as an M-structure can be radically improved. In this case, it would be possible to use the implementation techniques proposed for Id.

7.7.5 Actors

The notion of Actors, as the intuitive notion of the essence of a process, was introduced by Hewitt and Baker [1977], and was further developed by Clinger [1981] and Agha [1986]. It was a source of inspiration for the conception of the concurrent logic programming languages. A mapping from Actors to AKL is shown here to make clear the ease with which the actor style of concurrency and “mailbox” communication can be expressed AKL.

A pure view of Actors is given. Any language based on Actors will be augmented by the power of the chosen host formalism, such as Scheme in the Act family of actor languages [Agha and Hewitt 1987], but the process-oriented component remains the same.

An *actor* has an *address* to which it may receive *communications*. We assume that communications have *identifiers* and *arguments*, which are the addresses of other actors. It has *acquaintances* which are other actors whose addresses it knows. Upon receiving a communication

1) communications may be sent in response, to its acquaintances or to arguments of the communication,
2) new actors may be created, with acquaintances chosen from the acquaintances of the actor, the arguments of the communication, and other new actors,
3) one of these actors may be appointed to receive further incoming communications on the original address.

An actor is defined by specifying its behaviour for each relevant type of communication.

It is easy to map the above notion of an actor into AKL using *ports* as follows.

The definition of an actor A is mapped to a definition of an AKL agent A. The address of an actor is a port (P), which is created for each new actor, and to which communications are sent. The acquaintances (F_1, …, F_k) of A are supplied in the other arguments of A.

A(P, F_1, …, F_k) :=
    open_port(P, S),
    A_behaviour(S, F_1, …, F_k).

The agent A_behaviour consumes the corresponding stream (S) of communications, which are constructor expressions of the form m(X_1, …, X_l), where X_i are ports. Its definition contains one clause of the following form for each possible type of incoming communication.

A_behaviour([m(X_1, …, X_l) | S], F_1, …, F_k) :-
  → send(M_1, …), …, send(M_m, …),
    new_N_1(P_1, …), …, new_N_n(P_n, …).
    A′_behaviour(S, …).

The communications sent are M_1, …, M_m. The new actors created are calls to new_N_1, …, new_N_n, and A′ receives further communications on S. The communications sent, their arguments, and the remaining arguments to the atoms representing new actors, are chosen from X_1, …, X_l, F_1, …, F_k, P_1, …, P_n, as given by the original agent definition.

Each definition also contains a clause

A_behaviour([], F_1, …, F_k) :- | true.

which terminates the actor when its address, the port, is no longer accessible, and has been closed automatically.

A major deficiency of actor languages in comparison with AKL, and other concurrent (constraint) logic programming languages, is that the elegance of implicitly concurrent programs such as quicksort is not readily reconstructed in an actor-based language, with its explicit and strictly unordered form of communication.

7.7.6 Linda

Linda is a general model for (asynchronous) communication over a conceptually shared data structure, the *tuple space* [Gelernter et al. 1985; Carriero and Gelernter 1989]. In a tuple space, *tuples* are stored, which are similar to constructor expressions in AKL, records in Pascal, and the like. There are three operations which access the tuple space. In the following *x* stands for an expression, *v* for a variable, and the notation {X | Y} for either X or Y.

    out(x_1, …, x_n)

stores a tuple in the tuple space,

in({var v_1 | x_1}, …, {var v_n | x_n})

retrieves (and deletes) a matching tuple from the tuple space, suspending if none was found, and

read({var v_1 | x_1}, …, {var v_n | x_n})

locates (but leaves in place) a matching tuple in the tuple space. The notation "var *v*" means that the variable is constrained by the matching operation; the other arguments should be equal to the corresponding slots in the matching tuple.

A tuple space can be modelled as an AKL agent. Processes access the tuple space using a port, the corresponding stream of which is consumed by the tuple space agent. The tuple space is represented as a list of tuples. Tuples may be arbitrary objects, which are selected using arbitrary conditions, not only plain matching. Note the use of the *n*-ary call primitive for this purpose. A condition is a constructor expression of the form

$p(x_1, \ldots, x_m)$

When "applied to" an argument using ternary *call*

call($p(x_1, \ldots, x_m)$, x, y)

it becomes the agent

$p(x_1, \ldots, x_m, x, y)$

which in the following context is expected to return a continuation on *y* if *x* satisfies the condition, otherwise it should fail. In general, all *n*-ary call operations may be regarded as implicitly defined by clauses

call(p(X_1, …, X_m), Y_1, …, Y_{n-1}) :- → p(X_1, …, X_m, Y_1, …, Y_{n-1}).

for all p / *n* type (program and constraint) atoms in a given program.

The *tuple space server* accepts a stream of "in" and "out" requests. ("read" is omitted, being a restricted form of "in".)

```
tuple_space([], _, []) :-
    → true.
tuple_space([in(P) | R], TS, S) :-
    → in(P, TS, S, TS1, S1),
      tuple_space(R, TS1, S1).
tuple_space([out(T) | R], TS, S) :-
    → out(T, R, TS, S, TS1, S1),
      tuple_space(R, TS1, S1).
```

The *in request* is served by scanning through the list of tuples looking for a matching tuple. If one is found, the continuation is called, otherwise the request is stored in the list of suspended requests.

```
in(P, [], S, TS1, S1) :-
    → TS1 = [],
      S1 = [P | S].
in(P, [T | TS], S, TS1, S1) :-
      call(P, T, C)
    → call(C),
      TS1 = TS,
      S1 = S.
in(P, [T | TS], S, TS1, S1) :-
    → TS1 = [T | TS2],
      in(P, TS, S, TS2, S1).
```

The *out request* is served by scanning through the list of suspended in requests looking for a matching request. If one is found, its continuation is called, otherwise the tuple is stored in the list of tuples.

```
out(T, TS, [], TS1, S1) :-
    → S1 = [],
      TS1 = [T | TS].
out(T, TS, [P | S], TS1, S1) :-
      call(P, T, C),
    → call(C),
      TS1 = TS,
      S1 = S.
out(P, TS, [M | S], TS1, S1) :-
    → S1 = [M | S2],
      out(P, TS, S, TS1, S2).
```

As a simple example, consider the following condition

```
integer(Y, X, C) :-
      integer(X)
    | C = (Y = X).
```

and the call

```
tuple_space([in(integer(Y)), out(a), out(1)], [], [])
```

which returns

```
Y = 1
```

after first suspending the in(integer(Y)) request, then trying it for out(a), which fails and is stored in the tuple space, then for out(1), which succeeds.

Of course, the above simple scheme can be improved. As it stands, the single tuple space server is a bottle neck, which unnecessarily serialises all accesses. This can be remedied, e.g., by distributing requests to different tuple-spaces using a hashing function, but the generality of the representation of conditions and tuples in the above program would have to be sacrificed to allow for a suitable hashing function.

```
send(Request, TupleSpace) :=
        hash(Request, Index),
        arg(Index, TupleSpace, Port),
        send(Request, Port).
```

To each port in the hash table is connected a tuple space server of the kind presented above.

7.7.7 Erlang

Erlang is a concurrent functional programming language designed for prototyping and implementing reliable (distributed) real time systems [Armstrong, Williams, and Virding 1991; 1993]. Although not restricted to this area, Erlang is mainly intended for telephony applications. That functional programming as in the functional component of Erlang is provided by AKL is illustrated in Section 3.4.1. Here it is argued that the process component, including its error handling capabilities, can be expressed in AKL.

Erlang is probably less well known than the other languages in this chapter. Therefore, its essential features will be summarised before the comparison.

Processes are expressed in a sequential first-order functional programming language. The functional component is augmented with side effect operations such as creation of processes and message passing. The operation which creates a process returns a process identifier (*pid*).

Pid = spawn(*Module*, *Function*, *Arguments*)

A process may acquire its own pid using the self() operation. Pids may be given *global names*, a feature not further discussed here. Messages are sent to a pid with the *Pid* ! *Message* operation. Messages are stored in the *mail-box* of a process, from which they may be retrieved at a later time using the *receive* operation

```
receive
        Pattern [when Guard] -> … ;
        …
         [after Time -> …]
end
```

If a message matching a pattern (and satisfying the guard condition) is found among the messages in the mail-box, it is removed. The pattern is a constructor expression with local variables which are bound to the corresponding components in the message. The last clause may specify a *time-out*, which will not be further discussed here.

For the purpose of error-handling, processes may be *linked*. [It is assumed that these links are unidirectional, i.e., *from* one process *to* another process.] Links may be added, deleted, and, in particular, associated with new processes atomically at the moment of spawning (in which case a link is established from the new process to the old process).

link(*Pid*)	(*Pid* to self)
link(Pid_1, Pid_2)	(Pid_2 to Pid_1)
unlink(*Pid*)	
unlink(Pid_1, Pid_2)	
Pid = spawn_link(*Module*, *Function*, *Arguments*)	(*Pid* to self)

If an error occurs in a process, exit-signals are automatically sent to the processes it is linked to. A process which receives an exit-signal will by default also exit. This can be overridden, whereby signals are trapped, and converted to ordinary messages, which can be received in the usual fashion. A process may simulate an exit, and may also force another process to exit.

process_flag(trap_exit, [true / false])	(trap exit-signals)
receive {'EXIT', *From*, *Message*} -> … ; end	
exit(*Reason*)	
exit(*Pid*, *Reason*)	(force exit of *Pid*)

Signals may also be caught and generated by *catch* and *throw* operations, which are not further discussed here.

We now proceed to express the above in AKL using ports. To simplify the presentation, the following restrictions are made:

- Communication in Erlang is somewhat similar to *tuple space* communication in Linda, which was discussed in the previous section, and does also provide for time-outs. This aspect is ignored here. Communication is sent on a plain port, and served in a first-come first-served manner.
- Erlang distinguishes between *signals* and *messages*. This distinction is not made here, where both are regarded as messages, as when trapping of exit signals is turned on in Erlang.
- In Erlang, processes notify linked processes also about successful completion. This can be detected using short-circuiting, as in Section 7.7.1, and is not deemed important here.

An Erlang process identifier is mapped to an AKL port. Erlang processes have coarser granularity than AKL processes; to each Erlang process corresponds a *group* of AKL processes, performing the computations corresponding to the (sequential) functional component of Erlang. It is assumed that the definitions of these processes are written in the flat committed choice subset of AKL, where the guards may only contain simple tests.

The processes in a group share a pair p(V, P), the first component of which is either unconstrained or equal to the constant “exit”, and the second component of which is a port. It is assumed that all atoms, except the simple tests in guards, are supplied with this pair in an extra argument. If an error occurs in a primitive operation, a message of the form exit(*Reason*) is sent on the port in the pair, otherwise no action is taken. If the first component is equal to “exit”, an operation may terminate with no further action.

All choice statements in the definitions of processes in a group should have an extra case which examines the state of the first component of the pair, terminating if it is equal to “exit”.

```
p(…, F) :- …
…
p(…, p(exit,_)) :- | true.
```

A process group is created as follows.

```
group(Args, L, P) :=
        descriptor(P, Args, D),
        error_handler(L, P, S, F),
        processes(S, Args, F, P).
```

The *descriptor* agent forms a suitable descriptor of the current process to be sent to linked processes in the event of a failure, and may be defined in any appropriate manner. The definitions of the following agents are generic, and can be shared between all process groups.

```
error_handler(L, P, S, F) :=
        open_port(P, S),
        open_port(FP, Cs),
        F = p(_,FP),
        distribute(S, FP, Ms),
        linker(Cs, D, L).
```

The *distribute* agent filters out the *special* messages to the group, concerning linking and exiting, which are sent to the linker.

```
distribute([], C, Ms) :-
    → Ms = [].
distribute([special(C) | S], FP, Ms) :-
    → send(C, FP, FP1),
        distribute(S, FP1, Ms).
distribute([M | S], FP, Ms) :-
    → Ms = [M | Ms1],
        distribute(S, FP, Ms1).
```

The *linker* agent holds the list of pids (ports) of processes which the group is linked to, and an initial list (L) may be provided at the time of creation of the group, as in the *spawn_link* operation. It informs linked processes in the event of forced exit or failure.

```
linker([link(P1) | Cs], D, L) :-
    → linker(Cs, D, [P1 | L]).
linker([unlink(P1) | Cs], D, L) :-
    → delete(P1, L, L1),
        linker(Cs, D, L1).
linker([exit(R) | Cs], D, L) :-
    → F = p(exit,_),
        send_ports(L, exit(D, R)).
```

```
send_ports([], _) :-
    → true.
send_ports([P | Ps], M) :-
    → send(M, P),
      send_ports(Ps, M).
```

The *processes* agent creates the group of processes that perform the actual computation, receive and send messages, etc.

As demonstrated above, the degree of safety achieved with Erlang error handling can also be achieved in AKL, but at the cost of slightly more verbose programs: some process group set-up, and an extra argument to all atoms. Syntactic sugar could easily remedy this, if such capabilities warrant special treatment.

The computational overhead is the time to set up the error handler and the storage it occupies, the filtering of messages, the passing around of the extra argument, and (occasionally) suspending on the extra argument.

As it is currently defined, the error_handler will remain also when computation in the corresponding process group has terminated. It will terminate only when there are no more references to the port of the group.

7.8 DISCUSSION

The notion of ports provides AKL with a model of mutable data, which is necessary for effective object-oriented programming and effective parallel programming. Ports can serve as an efficient interface to foreign objects, e.g., objects imported from C++. They also support a variety of process communication schemes. Thus, ports reinforce otherwise weak aspects of AKL, and can also be introduced and used for the same purpose in other concurrent (constraint) logic programming languages.

Other useful communication mechanisms are conceivable that do not fall into the category discussed here. For example, the *constraint communication* scheme of Oz [Smolka, Henz, and Würtz 1993], has similar expressive power (except for automatic closing), but has to be supported by the additional concept of a blackboard. It can, however, be emulated by ports, using code equivalent to that for M-fields in Id (Section 7.7.4).

CHAPTER 8

RELATED WORK

Which are the ancestors and siblings of AKL and how does it compare to them? As a complement to the comments and small sections on related work scattered over the other chapters, this chapter offers more extensive comparisons between AKL and its main ancestors, Prolog, CLP, GHC, and the cc framework, and with a couple of selected languages that address similar problems.

8.1 AKL VS. PROLOG

Prolog is the grand old man of logic programming languages, conceived in 1973 and still going strong (see, e.g., [Clocksin and Mellish 1987; Sterling and Shapiro 1986; O'Keefe 1990]). AKL was designed so as to encompass as much as possible of the good sides of Prolog, while leaving out the not so good.

This section attempts to illuminate the relation between AKL and Prolog/CLP from several different perspectives. First, a simple computation model for constraint logic programming is shown, and a comparison is made with the AKL model. Then, we look at the issue of definite clause programming in AKL, how to translate a definite clause program to AKL, and how to interpret it using AKL. Prolog extends definite clause logic programming in a number of ways. Some of these can be dealt with in AKL, some cannot. It is shown how to interpret a logic program with side effects in AKL. Then, it is suggested how AKL provides the different forms of parallel execution suggested for Prolog as concurrency, potentially parallel execution, in its computation model. Finally, the Basic Andorra Model for the parallel execution of definite clause logic programs, a major source of inspiration for AKL, is described and discussed.

8.1.1 A Constraint Logic Programming Model

A computation in constraint logic programming (CLP) is a proof-tree, obtained by an extended form of input resolution on Horn clauses [Jaffar and Lassez 1987; Lloyd 1989]. 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, an *answer constraint* for the variables occurring in the query has been produced.

Syntax of Programs

The syntax of CLP programs is defined as follows.

⟨program clause⟩ ::= ⟨head⟩ :- ⟨sequence of constraint atoms⟩, ⟨body⟩

⟨head⟩ ::= ⟨program atom⟩

⟨body⟩ ::= ⟨sequence of program atoms⟩

The letter C stands for a sequence of constraint atoms, and the expression $\sigma(C)$ stands for the conjunction of these constraints (or **true** if the sequence is empty). The *local variables* of a clause are the variables occurring in the clause but not in its head.

The computation states are goal clauses, but for the sake of easier comparison with the AKL model, we use a simplified and-box.

⟨and-box⟩ ::= $\mathbf{and}(\langle\text{sequence of program atoms}\rangle)^{\langle\text{constraint}\rangle}$

The *clausewise reduction* rule

$\mathbf{and}(R, A, S)^{\theta} \Rightarrow \mathbf{and}(R, B, S)^{\theta \wedge \sigma(C)}$

is applicable if $\theta \wedge \sigma(C)$ is satisfiable and if a clause

A :- C, B

can be produced by replacing the formal parameters of a program clause with the actual parameters of the program atom A, and replacing the local variables of the clause by variables not occurring in the rewritten and-box.

Syntax of Goals

A *configuration* is an and-box. The *initial* configuration is of the form

$\mathbf{and}(R)^{\mathbf{true}}$

containing the query (the sequence of program atoms R). Configurations of the form $\mathbf{and}()^{\theta}$ are called *final*. The constraint θ 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 given constraint system.

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 a program atom. (This is quite implicit in the above formulation.) The computation model is nondeterministic. This clause search nondeterminism will necessarily be

made explicit in an implementation. Therefore, the computation model is extended to make clause search nondeterminism explicit.

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

⟨global goal⟩ ::= ⟨and-box⟩ | ⟨or-box⟩

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

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

The corresponding rewrite rule, called definitionwise reduction, creates an or-box containing all and-boxes that are the result of clausewise reduction operations on a selected program atom.

The *definitionwise reduction* rule

$$\mathbf{and}(R, A, S)^{\theta} \Rightarrow \mathbf{or}(\mathbf{and}(R, B_1, S)^{\theta \wedge \sigma(C1)}, \ldots, \mathbf{and}(R, B_n, S)^{\theta \wedge \sigma(Cn)})$$

unfolds a program atom A using those of its clauses

$$A :- C_1, B_1, \ldots, A :- C_n, B_n$$

for which $\theta \wedge \sigma(C_i)$ is satisfiable (the *candidate clauses*). Again, the actual parameters of A have been substituted for the formal parameters of the program clauses, and the local variables of the clause have been replaced by variables not occurring in the and-box. When there are no candidate clauses, an or-box with no alternative and-boxes **or**(), the empty or-box or **fail**, is produced.

Definitionwise reduction is sufficient to model the behaviour of SLD-resolution and related constraint logic programming models.

8.1.2 Definite Clauses in AKL

We now relate AKL to SLD-resolution, the computation model underlying Prolog, by a translation from definite clauses into AKL and by an interpreter for definite clauses written in AKL.

Translating Definite Clauses

We will show that this basic Prolog functionality is available in AKL by mapping definition clauses and SLD-resolution on corresponding AKL concepts. No formal proof is given, but the discussion should make the correspondence quite clear.

Let *definite* (*program*) *clause*, *definite goal*, *SLD-derivation*, *SLD-refutation*, and related concepts be defined as in Lloyd [1989]. Only the computation rule that selects the leftmost atom in the goal, as in the case of Prolog, will be considered.

For emulation of SLD, we use the constraint system of finite trees.

We further assume that all definitions have at least two clauses. If this is not the case, dummy clauses of the form p ← fail are added. This is necessary only to

get *exactly* the same derivations as in SLD-resolution, and may otherwise be ignored when programming relational programs in AKL.

Let a *definite clause*

$p(T_1, \ldots, T_n) \leftarrow B$

be translated into AKL as

$p(X_1, \ldots, X_n)$:- true ? $X_1=T_1, \ldots, X_n=T_n$, B.

putting equality constraints corresponding to the head arguments in the body, and interpreting the sequence of goals in the body B as AKL composition. Note that the arguments of this AKL clause are different variables, and that the guard is empty.

Similarly, let *definite goals*

$\leftarrow B$

be translated into AKL as

B

As an example, assume in the following the above translation of the definitions of some predicates p, q, and r, which have at least two clauses each. The definition of p is

p :- true ? B_1.
p :- true ? B_2.

The execution of the goal

p, q, r

proceeds as follows. Computation begins by unfolding each of the atoms with its definition, and with further computation within the resulting choice statements. However, as the guards are empty, these are immediately solved.

(true ? B_1 ; true ? B_2), “choice for q”, “choice for r”

Since q and r have at least two clauses (this is the reason for adding dummy clauses to unit clause definitions), the and-box is also stable. Computation may now proceed by trying the alternatives B_1 and B_2.

B_1, “choice for q”, “choice for r”

B_2, “choice for q”, “choice for r”

Computation will then proceed in both alternatives by unfolding the atoms in the bodies B_i, and by telling the equality constraints corresponding to head unification. Failure will be detected before trying alternatives again.

If we analyse the above computation in terms of SLD-resolution, we see that unfolding essentially corresponds to finding variants of program clauses for the atom in question. Nondeterminate choice corresponds to a first (imaginary) stage in the resolution step, where a clause is chosen, and the subsequent telling of the constraints to a second stage, where the substitution is applied.

If we identify AKL computation states of the form

$\theta_1, \ldots, \theta_i$, "choice for p_1", $\theta_{i+1}, \ldots, \theta_j$, "choice for p_k", $\theta_{j+1}, \ldots, \theta_k$

with the definite goals

$(\leftarrow p_1, \ldots, p_k)\theta_1 \cdots \theta_i\theta_{i+1} \cdots \theta_j\theta_{j+1} \cdots \theta_k$

the sequence of *stable* AKL computation states in one alternative branch of the computation corresponds to an *SLD-derivation* using the original definite clauses.

If we further identify a failed AKL computation state

fail

with the empty clause, the failed branches will correspond to *SLD-refutations.*

Of course, the soundness and completeness results for SLD-resolution carry over to this subset of AKL, but stronger results, summarised in Chapter 4, hold for a more interesting logical interpretation of a larger subset of AKL.

Interpreting Definite Clauses

In the translation of definite clauses in the previous section, a nondeterminate choice operation was necessary for each resolution step. Instead, the effect of branching in an SLD-tree can be achieved by recursion in different guards, as shown in the following interpreter, adapted from Ken Kahn's "or-parallel Prolog interpreter in Concurrent Prolog" [Shapiro 1986a].

```
solve([]).
    → true.
solve([A | As]) :-
        clauses(A, Cs)
    → resolve(A, Cs, As).

resolve(A, [(A :- Bs) | Cs], As) :-
        append(Bs, As, ABs),
        solve(ABs)
    ?   true.
resolve(A, [C | Cs], As) :-
        resolve(A, Cs, As)
    ?   true.

append([], Y, Z) :-
    → Y = Z.
append([E | X], Y, Z0) :-
    → Z0 = [E | Z],
        append(X, Y, Z).
```

Definite clauses are represented as follows.

```
clauses(member(_, _), Cs) :-
    → Cs = [ (member(E, [E | _]) :- []),
             (member(E, [_ | R]) :- [member(E,R)]) ].
clauses(ancestor(_,_), Cs) :-
    → Cs = [ (ancestor(X,Y) :- [parent(X,Y)]),
             (ancestor(X,Y) :- [parent(X,Z), ancestor(Z,Y)]) ].
clauses(parent(_,_), Cs) :-
    → Cs = [ (parent(a,ma) :- []),
             (parent(a,fa) :- []),
             (parent(b,mb) :- []),
             (parent(b,fb) :- []),
             (parent(fa,c) :- []),
             (parent(mb,c) :- []) ].
```

At the time of its discovery it was regarded as a "death blow" to Concurrent Prolog, as the then existing or-parallel implementations of Prolog were very inefficient. Since then, efficient or-parallel implementations have appeared, such as Aurora [Lusk et al. 1988; Carlsson 1990] and Muse [Ali and Karlsson 1990; Karlsson 1992].

The technique used for the above interpreter can also be used for programming in AKL, but it is sometimes awkward. For simple programs, such as member, with only one call in the body, the technique is fairly straight-forward, but the resulting program loops in AKL.

```
member(E, [E | R]).
member(E, [_ | R]) :- member(E, R) ? true.
```

The interpreter uses a *success continuation* with the remaining calls in the body, which is passed to each recursive instance of solve. In the member definition, the continuation is empty and can be ignored. The ancestor definition (in the clauses/2 definition) has two calls in its body, and can be translated as follows.

```
ancestor(X, Y, C) :- parent(X, Y, C) ? true.
ancestor(X, Y, C) :- parent(X, Z, ancestor(Z, Y, C)) ? true.

parent(a, ma, C) :- call(C) ? true.
…
```

Ordinary enumeration can be combined with guard nesting, e.g., as follows.

```
ancestor(X, Y) :- parent(X, Y) ? true.
ancestor(X, Y) :- ( parent(X, Z) ? ancestor(Z, Y) ) ? true.
```

To this particular case, with nondeterminate choice being performed without siblings in the guard, correspond particularly efficient implementation techniques, since no copying (or sharing) of active goals is necessary.

8.1.3 Prolog Particulars

Prolog extends SLD-resolution with

- the *pruning* operation cut

- *side effects* such as input and output operations on files and operations on the internal database
- *metalogical* operations such as var / 1

These extensions have no interpretations in terms of SLD-resolution.

We will discuss how Prolog programs with cut and side effects can be executed in AKL. Metalogical operations are not provided for; why will be discussed. A familiarity with Prolog will be assumed (e.g., as can be acquired from [Clocksin and Mellish 1987], [Sterling and Shapiro 1986], or [O'Keefe 1990]).

8.1.4 Cut

The *cut* pruning operation in Prolog is similar to conditional choice in AKL. For example, a Prolog definition

```
p(a, Y) :- !, q(Y).
p(X, Y) :- r(Y).
```

may be translated to AKL as

```
p(a, Y) :- → q(Y).
p(X, Y) :- → r(Y).
```

if in each of its uses, the first argument is known to be equal to ‘a’ or known to be different from ‘a’. In this case, cut may be performed without having added constraints to the arguments of p / 2 calls. This is called *quiet pruning*. If, on the other hand, the first argument is unknown, the Prolog cut operation would still take effect, even though it is not known that the first argument is equal to ‘a’. This is called *noisy pruning*. The corresponding AKL program would suspend in this case. For the purpose of translation into AKL, it is desirable to transform, if possible, programs using noisy pruning to programs using only quiet pruning. For example, if the above Prolog program is always used with a variable as the first argument, it can be safely transformed into

```
p(X, Y) :- !, X = a, q(Y).
p(X, Y) :- r(Y).
```

where the use of cut is quiet. If there are occurrences of p / 2 atoms in the program that are always used in one way, and other occurrences that are always used in the other, then two different definitions can be used to make both cases quiet. The full functionality of noisy cut pruning is only required if the same occurrence is used in both ways. (An experimental extension of AKL with a (*noisy*) *cut choice* statement, with the ability to perform noisy pruning, is described in Section 4.7.2.)

It is possible to make translation from Prolog into AKL completely automatic along the lines outlined in this section using data-flow analysis by abstract interpretation [Bueno and Hermenegildo 1992]. The noisy cut extension is then used when no other translation is possible. The use of noisy cut also involves using means to sequentialise the program to avoid back-propagation of values that violates Prolog semantics. Bueno and Hermenegildo also observed that the

performance of Prolog programs was improved by transforming from noisy to quiet cut.

Although similar to conditional choice, cut in Prolog may be used quite arbitrarily, e.g., as in

```
p :- q, !, r, !, s.
p :- t.
p :- u, !.
```

To correspond more closely to AKL definitions, each clause should contain exactly one cut or no clause should contain cut. We first observe that the repeated use of cut in the first clause can be factored out, as in

```
p :- q, !, p1.
```

with an auxiliary definition

```
p1 :- r, !, s.
```

We then factor out a cut-free definition containing the second clause, and a third cut-definition with the third clause. The resulting program is

```
p :- q, !, p1.
p :- !, p2.

p1 :- r, !, s.

p2 :- t.
p2 :- p3.

p3 :- u, !.
```

which, if the uses of cut are quiet, corresponds to the AKL definition

```
p :- q → p1.
p :- → p2.

p1 :- r → s.

p2 :- ? t.
p2 :- ? p3.

p3 :- u → true.
```

Such awkward translations are rarely necessary in practice. Our experience of translating Prolog programs is that elegant translations are usually available in specific cases. Typically, Prolog definitions correspond more or less directly to a conditional choice or to a nondeterminate choice. It should be noted that there is no fully general translation of cut inside a disjunction. However, it is the opinion of the author that this abominable construct should be avoided anyway.

Unfortunately, the quietness restriction also makes some pragmatically justifiable programming tricks impossible in AKL that are possible in Prolog. These tricks depend on the sequential flow of control, and the resulting particular instantiation patterns of the arguments of a cut-procedure in specific execution

states. There are usually work-arounds that do not involve noisy pruning. However, in some cases, the translation is quite non-trivial, and cannot be readily automated.

A typical case is presented together with suggestions for alternative solutions in AKL. The principles underlying these alternative solutions can be adapted to other similar cases. The following lookup/3 definition in Prolog relies on noisy cut to add a key-value pair automatically at the end of a partially instantiated association list if a pair with a matching key is not found.

```
lookup(K, V, [K=V1 | R]) :- !, V = V1.
lookup(K, V, [_ | R]) :- lookup(K, V, R).
```

Clearly, the corresponding AKL definition would only be able to find existing occurrences of the key. There are several work-arounds for this problem. It is for example possible to manage a complete list of key-value pairs, as in the following AKL program.

```
lookup(K, V, L, NL) :-
        lookup_aux(K, V1, L)
    → V = V1,
        NL = L.
lookup(K, V, D, ND) :-
    → ND = [K=V | D].

lookup_aux(K, V, [K=V1 | D]) :-
    → V = V1.
lookup_aux(K, V, [X | D]) :-
    → lookup_aux(K, V, D).
```

Note that the AKL definition corresponding to the above Prolog program is used as an auxiliary definition. The cost of searching and adding new elements remains the same.

However, when using this program, it is necessary to pass the list around more explicitly than in the Prolog solution. Note also that access is serialised. A list cannot be updated concurrently. The following solution allows concurrent lookups. It uses an incomplete list as in the Prolog solution, but access to the dictionary is managed by a dictionary server which synchronises additions to the tail of the list. The server is accessed through a port.

```
alist(P) :-
        open_port(P, S),
        alist_server(S, D, D).

alist_server([lookup(K, V) | S], D, T) :-
    → lookup(K, V, D, T, NT),
        alist_server(S, D, NT).
alist_server([], D, T) :-
    → true.
```

```
lookup(K, V, [K1=V1 | R], T, NT) :-
        K = K1
    |   V = V1,
        T = NT.
lookup(K, V, [K1=V1 | R], T, NT) :-
        K ≠ K1
    |   lookup(K, V, R, T, NT).
lookup(K, V, T, T, NT) :-
    |   T = [K=V | NT].
```

The lookup definition is like the previous ones but for the last two arguments and the third clause. The extra arguments hold the tail of the list and the new tail of the list. If nothing is inserted, the old tail is returned. The third clause detects that the list is equal to its tail, inserts a new pair, and returns the new tail of the list. The definition has to be don't care nondeterministic, since the guards of the preceding clauses have not been refuted. Each lookup request is given the new tail of the preceding one, thus serialising updates, but permitting concurrent lookups.

We can note that this could also have been an FGHC program, but for the fact that FGHC implementations usually do not consider variable identity as a quiet case, and therefore the third lookup-clause will not recognise the uninstantiated tail as intended.

The above program can be extended to perform a checking lookup which does not add the new element, even though the list ends with a variable. In Prolog a metalogical primitive would be needed to achieve the same effect.

The incomplete structure technique is more useful when the lookup-structure is organised as a tree. Unless sophisticated memory management techniques are used (such as reference counting, producer-consumer language restrictions, or compile-time analysis), alternative solutions with complete structures will require O(log(N)) allocated memory for each new addition to the tree, but they are also less satisfactory because they require that access is serialised.

The following Prolog program will only allocate the new node.

```
treelookup(K, V, t(K, V1, L, R)) :- !, V = V1.
treelookup(K, V, t(K1, _, L, R)) :- K < K1, !, treelookup(K, V, L).
treelookup(K, V, t(_, _, L, R)) :- treelookup(K, V, R).
```

In AKL, the tree can be represented by a tree of processes, e.g., as shown in Section 3.2.3. It can be argued that this is less efficient than the Prolog solution, but a compilation technique for FGHC programs that optimises such programs, *message-oriented scheduling*, suggests that this inefficiency is not an inherent problem [Ueda and Morita 1992].

Finally, we observe that the troublesome *negation as failure*, expressed as

```
not(P) :- call(P), !, fail.
not(_).
```

in Prolog, is always *sound* in AKL when expressed as

```
not(P) :- call(P) → fail.
not(_) :- → true.
```

(see Section 4.8.2). A more complete, and still sound, definition of negation is achieved by wrapping the called goal in a committed choice, as in

```
not(P) :- (call(P) | true) → fail.
not(_) :- → true.
```

It is now possible to call

```
not(member(X, [a, X, b]))
```

using member defined as in Section 2.5, and get the desired failure.

8.1.5 Side Effects

It is possible to model definite clauses with side effects in AKL, while still interpreting programs in a true "metainterpreter" style, mapping object-level nondeterminism to metalevel nondeterminism. The program as such is probably not very useful, but it illustrates the versatility of AKL, and also introduces programming techniques with wider applicability.

The program makes use of a technique related to that used for the constant delay multiway merger defined in Section 7.3.2.

An ordered bagof agent encapsulates the or-tree formed by the interpreter. Solutions are communicated to a server process, which is also in charge of side effects. In the example interpreter, the only possible side effects are reading and writing a value held in the server, but this can be extended, e.g., to a scheme reminiscent of assert/retract or to include I/O.

When the interpreter wishes to perform a side effect, it communicates a special "solution" to bagof which is then sent to the side effect server. This solution is either a read or a write message. In a write, the value to be written is simply given as its argument.

Read is more tricky. Since the bagof collection operation renames local variables, a read result cannot easily be returned using local variables. The read operation must have access to a *unique* global variable that is sent to the server and on which the result can be returned. This is achieved as follows. The interpreter is given an extra external variable as an argument. Upon every branching performed, a redundant "solution" is returned, which communicates to the server that a split has been performed, sending its external variable as an argument. The server then binds this variable to a structure split(_,_). The two new branches select one new external variable each, and continue. The read operation sends the current external variable. The result is returned by binding the external variable to a structure read(V,_), where the first argument is the value read, and the second argument is a new external variable for future communications.

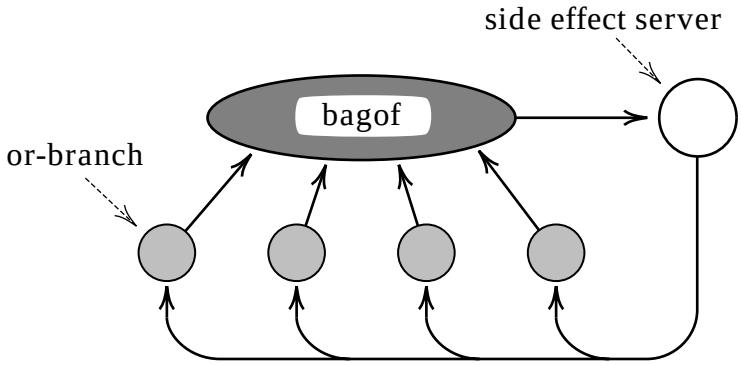


Figure 8.1. Definite clause interpreter with side effects

The complete interpreter follows. As explained above, the interpreter consists of a bagof and a server process.

```
solve :-
        bagof(X, solve(X, _), Y),
        server(Y, []).

solve(X, E) :-
        G = ⟨goal to be called⟩,
        solve([G], X, G, E).
```

The server deals with the different forms of “solutions” that may be produced: write, read, split, and a real solution.

```
server([], _) :-
    → true.
server([write(U) | Xs], V) :-
    → server(Xs, U).
server([read(U) | Xs], V) :-
    → U = read(V,_), server(Xs, V).
server([split(E) | Xs], V) :-
    → E = split(_, _), server(Xs, V).
server([solution(_) | Xs], V) :-
    → server(Xs, V).
```

The interpreter catches the special cases, dispatching to special procedures for read and write, and to clause trying for other goals.

```
solve([], X, G, E):-
    → X = solution(G).
solve([write(U) | As], X, G, E) :-
    → solve_write(U, As, X, G, E).
solve([read(U) | As], X, G, E) :-
    → solve_read(U, As, X, G, E).
solve([A | As], X, G, E) :-
        clauses(A, Cs)
    → try(A, Cs, As, X, G, E).
```

Write and read send messages to the server in special “solutions”. The read message, with the value and a new variable, is received in the second branch.

```
solve_write(U, As, X, G, E) :-
    ?  X = write(U).
solve_write(U, As, X, G, E) :-
    ?  solve(As, X, G, E).

solve_read(U, As, X, G, E) :-
    ?  X = read(E).
solve_read(U, As, X, G, E) :-
    ?  receive_read(E, U, As, X, G).

receive_read(read(V,E), U, As, X, G) :-
    →  U = V,
       solve(As, X, G, E).
```

A split is performed if there are two or more clauses to try. When splitting, a message is sent to the server, to allow it to generate new external variables that will correspond to the E's in the second and third clauses.

```
try(A, [_,_|_], As, X, G, E) :-
    ?  X = split(E).
try(A, [(A:-Bs)|_], As, X, G, split(E,_)) :-
    ?  append(Bs, As, ABs),
       solve(ABs, X, G, E).
try(A, [_,C|Cs], As, X, G, split(_,E)) :-
    ?  try(A, [C|Cs], As, X, G, E).
```

A limitation of the scheme presented here compared to Prolog is that read terms containing variables may not be manipulated freely. Upon writing, a term is promoted to the external environment, making any variable external. Such variables may not be bound in the interpreter. Effectively, this means that the read and write operations are restricted to ground terms.

8.1.6 Metalogical Operations

Prolog supports a number of so called *metalogical* operations that cannot be explained in terms of the basic computation model. All have in common that they regard unbound variables as objects. They can establish that a variable is unbound, test variables for equality, and compare them using the standard term ordering. One such notorious operation is var/1, which tests whether a given variable is unbound.

Such an operation does not rhyme well with concurrency. It destroys the property that all conditions are monotone. Once a guard is quiet or incompatible with its environment, this will continue to hold. The behaviour of var is anti-monotone, changing from success to failure if its argument is bound. Although pragmatically justifiable uses can be found, the very availability of such an operation encourages a poor programming style, and it is quite possible to do

without it. For this reason, all operations of this kind have been omitted from AKL.

8.1.7 Prolog and Parallelism

Considerable attention has been given to the topic of making Prolog programs run in parallel. Roughly, there are three basic approaches: *or-parallelism*, where the different branches in the search tree are explored in parallel [Lusk et al. 1988; Carlsson 1990; Karlsson 1992], *independent and-parallelism*, where goals may run in parallel if they do not disagree on shared variables [DeGroot 1984; Hermenegildo and Greene 1990], and *dependent and-parallelism*, where goals may be reduced in parallel, although they may potentially bind shared variables, as in the committed choice languages [Naish 1988; Santos Costa, Warren, and Yang 1991a; 1991b]. There are also various combinations of the above that await evaluation [Gupta et al. 1991; Gupta and Hermenegildo 1991; 1992].

The concurrency of AKL provides a potential for parallel execution. In the following sections, it is briefly discussed how concurrency (a potential for parallel execution) corresponding to the above three forms of parallelism may be identified in AKL programs.

When a computation is split into alternative computations by performing a nondeterminate choice, this forms a tree in a manner quite analogous to the search tree formed by Prolog execution.

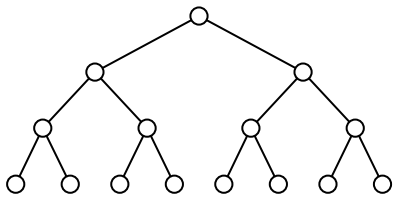


Figure 8.2. An or-tree

The different branches are quite independent, and computation steps may be performed concurrently. Clearly, this provides AKL with a potential for *or-parallel execution*.

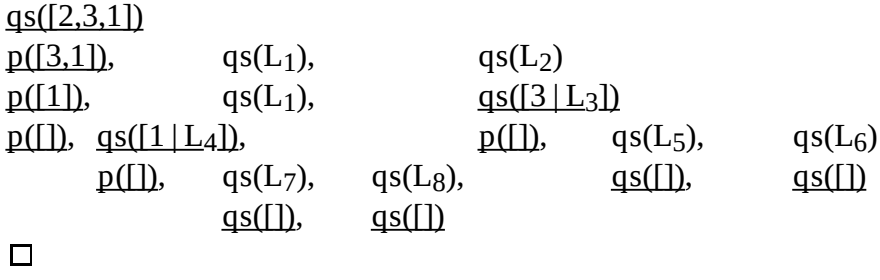
Execution mainly consists of unfolding goals, some pruning of alternatives in choices, followed by a choice of the single remaining alternative. The only interaction between such goals is through shared variables. This corresponds to committed choice execution and can be exploited as *dependent and-parallelism* in the usual manner.

Consider the well-known quicksort 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 determinate, and the AKL is able to extract concurrency 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.



□

Quite a lot of potential parallelism is extracted.

Various forms of execution corresponding to the *independent and-parallelism* exploited for Prolog are conceivable. To emphasise the similarity with Prolog, it is shown how pure definition clauses can be translated in such a way that a potential for independent and-parallelism appears.

Assume that in the clause

p(X) :- q(X, Y), r(X, Z), s(Y, Z).

the goals q and r are found to be independent. In the context of restricted and-parallelism, this means that the program that uses p calls it with an argument X such that neither of the goals q or r will instantiate X further. The following translation into AKL enables independent parallel execution of q and r as soon as X is sufficiently instantiated by its producers.

$p(X)$:- true ? $q_1(X, Y)$, $r_1(X, Z)$, s(Y, Z).

$q_1(X, Y)$:- $q(X, Y_1)$? $Y = Y_1$.

$r_1(X, Z)$:- $r(X, Z_1)$? $Z = Z_1$.

By putting the goals in guards and extracting the output argument, unless the goals attempt to restrict X, all computation steps are always admissible, including choice splitting.

This style of translation can make use of the tools developed for restricted and-parallelism, such as compile-time analysis of independence, making it completely automatic [Bueno and Hermenegildo 1992].

8.1.8 The Basic Andorra Model

The *Basic Andorra Model* (BAM) is a control strategy for the definite clause computation model with the appealing property of giving priority to deterministic work over nondeterministic work [Haridi and Brand 1988]. The BAM was first proposed by D. H. D. Warren at a GigaLIPS meeting in 1987, and was also discovered independently by Smolka [1993], who dubbed it *residuation* and described it for general constraints.

The BAM divides a computation within and-boxes into *determinate* and *nondeterminate phases*. First, all program atoms with at most one candidate (*determinate* atoms) are reduced during the determinate phase. Then, when no determinate atom is left, an atom is chosen for which all candidates are tried; this is called the nondeterminate phase. The computation then proceeds with a determinate phase on each or-branch.

The BAM has a number of interesting properties.

Firstly, all reductions in the deterministic phase may be executed in parallel, thereby extracting implicit dependent and-parallelism from pure definite clause programs, as in NUA-Prolog [Palmer and Naish 1991]. Andorra-I provides both dependent and- and or-parallelism on the Sequent Symmetry [Santos Costa, Warren, and Yang 1991a; 1991b].

Secondly, the notion of determinacy is a form of synchronisation. While data is being produced during the determinate phase, consumers of this data are unable to run ahead (since this would make them nondeterminate). This makes it possible to program with a notion of concurrent processes.

Thirdly, executing the determinate goals first reduces the search space, and thus, in general, the execution time. Goals can fail early, and the constraints produced by a reduction can reduce the number of alternatives for other goals. This is very relevant for the coding of constraint satisfaction problems [Bahgat and Gregory 1989; Haridi 1990; Yang 1989; Gregory and Yang 1992].

In AKL, the principle underlying the BAM has been generalised to a language with deep guards, and is embodied in the stability condition. However, the BAM itself is available as a special case. Below is shown a simple translation from program clauses into AKL. The translated programs will behave exactly as dictated by the BAM.

By putting the constraints in the guard of a wait-clause, local execution in the guard will establish whether a clause is a candidate. Other rules have priority over choice splitting, and therefore execution in AKL will conform to the BAM.

A program clause

A :- C, B.

where C is a constraint, is translated into a corresponding AKL clause as

A :- C ? B.

according to the above suggested scheme.

8.2 AKL VS. COMMITTED-CHOICE LANGUAGES

The field of concurrent logic programming was initiated by Clark and Gregory [1981] with the Relational Language, and attracted considerable interest after strong promotion of Concurrent Prolog by Shapiro [1983]. Later developments include PARLOG [Clark and Gregory 1986; Gregory 1987], GHC [Ueda 1985], KL1 [Ueda and Chikayama 1990], and Strand [Foster and Taylor 1989; 1990].

With Concurrent Prolog, Shapiro first demonstrated the expressiveness of concurrent logic programming, and the process-oriented programming techniques developed have become standard. However, the synchronisation mechanism of Concurrent Prolog is based on the notion of *read-only variables*. It in no way corresponds to the notion of *asking*, and a comparison will not be attempted.

AKL is closer in spirit to more recent languages, and in Sections 8.2.2, 8.2.3, and 8.2.4, AKL is compared to GHC, KL1, and PARLOG, respectively.

8.2.1 A Committed-Choice Computation Model

In this section, a computation model for a committed choice language closely resembling GHC is presented.

A program is a finite set of *guarded clauses*.

⟨guarded clause⟩ ::= ⟨head⟩ :- ⟨guard⟩ ‘|’ ⟨body⟩

⟨head⟩ ::= ⟨program atom⟩

⟨guard⟩, ⟨body⟩ ::= ⟨sequence of atoms⟩

We keep the and-boxes of the CLP-model, which are augmented with sets of local variables, leave out the or-boxes, but instead introduce choice-boxes and guarded goals for guard computations.

⟨goal⟩ ::= ⟨local goal⟩ | ⟨and-box⟩

⟨and-box⟩ ::= $\mathbf{and}(\langle\text{sequence of local goals}\rangle)^{\langle\text{constraint}\rangle}_{\langle\text{set of variables}\rangle}$

⟨local goal⟩ ::= ⟨atom⟩ | ⟨choice-box⟩

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

⟨guarded goal⟩ ::= ⟨and-box⟩ ‘|’ ⟨statement⟩

The model is a labelled transition system of the same type as that used in the AKL model. We use the constraint atom rule, the promotion rule, and the commit rule. Failure rules are not needed (but are harmless).

The single new rule is the *local forking* reduction rule

$$A \overset{D}{\underset{\chi}{\Rightarrow}} \mathbf{choice}(\mathbf{and}(G_1)^{\mathbf{true}}_{V_1} \mid B_1, \ldots, \mathbf{and}(G_n)^{\mathbf{true}}_{V_n} \mid B_n)$$

which unfolds a program atom A with its definition

$$A :- G_1 \mid B_1, \ldots, A :- G_n \mid B_n$$

where the arguments of A are substituted for the formal parameters, and the local variables of the *i*th clause are replaced by the variables in the set V_i. The sets V_i are chosen to be disjoint from each other and from all other sets of local variables in the context χ.

8.2.2 GHC

GHC was proposed by Ueda as a rational reconstruction of PARLOG, Concurrent Prolog, and similar languages at the time [Ueda 1985].

A GHC program consists of *guarded clauses* of the following form.

h :- g_1, …, g_m | b_1, …, b_n.

Each of the components (h, g_i, b_i) is an *atom*, of the form

$p(t_1, \ldots, t_k)$

where t_i are *terms*, expressions built from variables, numbers, constants, and constructors, and p is an alpha-numeric symbol, as in AKL atoms.

It should be apparent that GHC is a syntactic subset of AKL with rational tree constraints.

The comparison can be taken further in that we may describe GHC computations in terms of their difference from AKL computations. In fact, an AKL program written in the GHC subset will execute almost as prescribed by the GHC definition. The same results will be produced, but possibly (for contrived programs) at a higher or a lower cost, as discussed below.

The most obvious difference is that GHC avoids binding external variables. Roughly, a binding X=t is only made visible if X is a local variable. If X is external, the binding is said to *suspend*, waiting for X to become bound in an external environment. The advantage of this restriction is that no local binding environments have to be maintained, but there are also disadvantages as discussed below.

Local bindings will sometimes detect failure, for example in

p(X) :- X=1, X=2 | true.

In AKL an agent p(Z), with an unconstrained variable Z, would fail, whereas GHC would make it suspend. Worse is that GHC may randomly fail or suspend, depending on whether the guard in the following clause is executed from the left or from the right, respectively.

p(X) :- Y = 1, Y = 2, X = Y | true.

Although no sane programmer would write code like this, the situation could appear as the result of a deep guard execution, or as the result of a program transformation. This problem is almost immaterial in theoretical GHC, as the difference between suspension and failure plays no essential rôle for the semantics of a program. In KL1, where failure can be detected, the problem is more serious.

There are two alternative behaviours for GHC, which both seem to be in accordance with its definition: (1) If a binding operation is suspended, the whole guard is suspended, since the computation cannot be completed until a corresponding external variable has been produced. (2) Only the binding operation is suspended; the other agents in the guard may continue.

Consider a GHC program of the following form.

```
p(X, Y) :- X = a, q(Y, X, Z) | r(Z).
p(X, Y) :- … .
```

The first guard not only awaits a binding, but also performs a computation. Assume that the execution of q is time-consuming and that the binding X = a is not yet available when p is first entered.

In some implementations of behaviour (1) the binding will be tried before, or very early in, the execution of q. Then most of q will suspend and will only be available for execution after the binding has been produced from elsewhere. This can lead to fewer computation steps than in AKL, where it is possible that steps are spent on q before trying other clauses, which may very well be applicable immediately. However, the program can execute slower in a parallel implementation of this kind than in a parallel implementation of AKL if processors are in good supply. In AKL, the producer of the binding X = a and q may execute in parallel. Behaviour (1) would make them execute in sequence, which could take longer time.

Behaviour (2) is closer to that of AKL, but it can be more wasteful, as follows. Again consider the above example. If, while q is being executed and no other clause is applicable, a conflicting binding X = b is suddenly produced by q, then the AKL guard would fail. GHC (2) would not detect the conflict, and work would be wasted on q.

If nothing else, this discussion shows that the differences between AKL and GHC are of the same calibre as the differences between alternative interpretations of GHC itself. It has even been claimed that the behaviour of the corresponding subset of AKL is indeed a viable interpretation of the GHC definition.

8.2.3 KL1

KL1 is based on Flat GHC (FGHC), a theoretically cleaner language, which, in its turn, is a special case of GHC [Ueda and Chikayama 1990; Chikayama 1992]. The “flatness” refers to a restriction on the guard part of a clause, which may not contain program atoms defined by the user program.

Thus, in KL1, in a clause of the form

$h :- g_1, \ldots, g_m \mid b_1, \ldots, b_n.$

the atoms g_i in the guard are restricted to (tree equality) constraints and certain built-in operations. This restriction will make the computation state flat. There will be no nested “choice-statements” as when the guard may contain arbitrary statements, and this simplifies the implementation considerably.

KL1 allows “pragma” of the form

% otherwise

and

% alternatively

which may be placed between the clauses in a definition.

The first, *otherwise*, means that the clauses following it should only be tried if the guards in all preceding clauses fail. The effect is similar to that of conditional choice in AKL, and even more similar to the sequential clause operator of PARLOG, for which a mapping into AKL is shown in the next section. When primitives that detect failure are introduced, such as otherwise, the order-dependence of the GHC suspension rule has to be taken into account. For KL1 the solution is to impose a sequential left-to-right execution order in guards, which gives predictable, if not ideal, behaviour.

The second, *alternatively*, means that clauses following it should only be tried if the guards in all preceding clauses have failed or are *currently* suspended at the time of attempting reduction. There is no similar construct in AKL.

KL1 also provides new high-level constructs, such as the *sho-en*, which is similar to the *engine* concept proposed for AKL.

A low-level facility of KL1 is the MRB optimisation, which allows data structures such as arrays to be updated in place in constant time if they are single-referenced. As only one reference may be held at any given instant, access is serialised. By splitting an array into two disjoint parts, the two may be updated in parallel, to be joined at a later time. AKL provides more flexible models of mutable data based on ports (Chapter 7).

8.2.4 PARLOG

PARLOG provides a number of constructs of potential interest [Clark and Gregory 1986; Gregory 1987]. Here we discuss Kernel PARLOG, the standard form without mode declarations and with explicit head unification. We will not discuss the don't know extension, the simple functionality of which is entirely subsumed by AKL. Familiarity with PARLOG is assumed.

A PARLOG definition is a sequence of clauses, grouped by *parallel* (“.”) and *sequential* (“;”) *clause composition* operators.

$$\langle \text{def} \rangle ::= \langle \text{clause} \rangle \mid (\langle \text{def} \rangle \,.\, \langle \text{def} \rangle) \mid (\langle \text{def} \rangle \,;\, \langle \text{def} \rangle)$$

$$\langle \text{clause} \rangle ::= \langle \text{atom} \rangle \leftarrow \langle \text{goal} \rangle : \langle \text{goal} \rangle$$

$$\langle \text{goal} \rangle ::= \langle \text{atom} \rangle \mid (\langle \text{goal} \rangle, \langle \text{goal} \rangle) \mid (\langle \text{goal} \rangle \,\&\, \langle \text{goal} \rangle)$$

The symbols “<-” and “:” correspond to “:-” and “ | ” in AKL. The operators “,” and “&” are *parallel* and *sequential conjunction*, respectively. The former corresponds to the composition statement of AKL; the latter is discussed below.

PARLOG in AKL

The difference between parallel and sequential clause composition is similar to that between committed and conditional choice in AKL. Thus, a definition

H <- $G_1 : B_1$.
…
H <- $G_n : B_n$.

which uses only ".", can be translated to

H^* :- $G_1^* \mid B_1^*$.
…
H^* :- $G_n^* \mid B_n^*$.

where * denotes a mapping from PARLOG goals to AKL statements, and a corresponding definition

H <- $G_1 : B_1$;
…
H <- $G_n : B_n$.

which uses only ";", can be translated to

H^* :- $G_1^* \rightarrow B_1^*$.
…
H^* :- $G_n^* \rightarrow B_n^*$.

However, when parallel and sequential clause composition operators are arbitrarily nested, the translation becomes less straightforward. The following is a mapping from definitions with arbitrary nesting of the clause composition operators to corresponding committed choice and conditional choice statements. It is awkward, and never needed for "real" programs, but included for the sake of completeness.

Number the guards and bodies of the clauses c_1 to c_n in the (Kernel) PARLOG definition g_1 to g_n and b_1 to b_n, correspondingly. The clause c_i has local variables $X_{i,1}$ to X_{i,k_i}. We assume that all heads are equal (with different variables as arguments); let H represent these.

A PARLOG definition is mapped to an AKL definition with a single commit clause. The guard of this clause computes a continuation which identifies chosen body, and which contains local variables shared between guard and body.

$$D^* \Rightarrow H^* \text{ :- } D^{\#(1)} \mid \; (X_{1,1}, \ldots, X_{1,k1} : B_0 = \text{cont}(1, X_{1,1}, \ldots, X_{1,k1}) \rightarrow b_1^* \; ; \ldots \; ; X_{n,1}, \ldots, X_{n,kn} : B_0 = \text{cont}(n, X_{n,1}, \ldots, X_{n,kn}) \rightarrow b_n^*).$$

The following rules map sequential and parallel clause composition to conditional and committed choice, respectively.

$$(D_1 ; \ldots ; D_m)^{\#(n)} \Rightarrow (B_n : D_1^{\#(n+1)} \rightarrow B_{n-1} = B_n \; ; \ldots \; ; B_n : D_m^{\#(n+1)} \rightarrow B_{n-1} = B_n)$$

$$(D_1 \,.\, \ldots \,.\, D_m)^{\#(n)} \Rightarrow$$
$$(B_n : D_1^{\#(n+1)} \mid B_{n-1} = B_n$$
$$; \cdots$$
$$; B_n : D_m^{\#(n+1)} \mid B_{n-1} = B_n)$$

If a guard succeeds, it returns a continuation for its corresponding body.

$$C_i^{\#(n)} \Rightarrow (X_{i,1}, \ldots, X_{i,k_i} : g_i^* \mid B_{n-1} = \mathrm{cont}(i, X_{i,1}, \ldots, X_{i,k_i}))$$

A complete mapping of sequential conjunction requires distributed termination techniques, for example *short-circuiting* (cf., [Shapiro 1986a]). This is demonstrated by the interpreter in the next section. Certain restricted cases of sequencing G_1 & G_2 can be expressed as (G_1 ? G_2) using nondeterminate choice. The difference is that the guard G cannot communicate with other agents until it has been completely evaluated.

PARLOG Interpreter in AKL

The above style of translation of sequential and parallel composition can also be expressed in terms of an interpreter for (a subset of) Kernel PARLOG in AKL. The sequential and parallel clause composition operations are represented by the constructors seq(C,D) and par(C,D), respectively. Here is also included a mapping of sequential conjunction in terms of short-circuiting. Observe the need for a special treatment of primitives.

```
parlog(true, C0, C) :-
    → C0 = C.
parlog((P,Q), C0, C) :-
    → parlog(P, C0, C1),
      parlog(Q, C1, C).
parlog((P&Q), C0, C) :-
    → parlog(P, D0, D),
      ( D0 = D | parlog(Q, C0, C) ).
parlog(H, C0, C) :-
      is_primitive(H)
    → primitive(H, C0, C).
parlog(H, C0, C) :-
      definition(H, D)
    → try(D, H, B),
      parlog(B, C0, C).

try(seq(C,D), H, B) :-
    → ( B0 : try(C, H, B0) → B = B0
      ; B0 : try(D, H, B0) → B = B0 ).
try(par(C,D), H, B) :-
    → ( B0 : try(C, H, B0) | B = B0
      ; B0 : try(D, H, B0) | B = B0 ).
```

```
try(clause(C), H, B) :-
        instance(C, (H <- G : B0)),
        parlog(G, _, _)
    → B = B0.
```

An example encoding of a definition (concat) follows.

```
definition(concat(X,Y,Z), Def) :-
    → Def = par(concat_1(X,Y,Z), concat_2(X,Y,Z)).

instance(concat_1(X,Y,Z), Clause) :-
    → Clause = (concat(X,Y,Z) <- X = [] : Z = Y).
instance(concat_2(X,Y,Z), Clause) :-
    → Clause = (concat(X,Y,Z) <- X = [E | X1] : Z = [E | Z1], concat(X1,Y,Z1)).

is_primitive((X = Y)) :- → true.

primitive((X = Y), C0, C) :-
    → X = Y, C = C0.
```

In the case of (equality) constraints, sequencing has no meaning in AKL.

Discussion

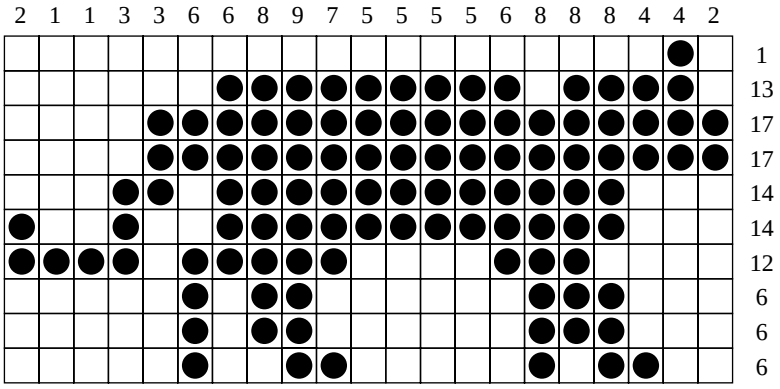
One might ask if it is necessary or even desirable to have general sequencing as a primitive in this kind of language. One common use is to sequence side effect operations, which have been added to the language in place of properly integrated support for interoperability. Other uses are to improve efficiency and resource allocation by explicit tampering with the execution order.

If a producer and a consumer can execute sequentially, they can be sequenced with a sequencing operator. Efficiency can be gained; there is no context switching between processes. If the consumer cannot take any action until its corresponding producer has terminated, efficiency is always gained. Efficiency can also be lost; nothing will be consumed while the producer is working, and the working set will grow, decreasing locality and increasing the time spent on garbage collections. Of course, potential parallelism is also lost.

AKL does not provide sequencing, since it does not add to the expressiveness of the language and there is good hope that sufficient efficiency can be achieved without such interventions by the programmer, adjusting this, adjusting that.

For example, unless the language requires fairness, there is no need for unnecessary context switching between processes. On the other hand, the language implementation is free to, for example, switch to the consumer when memory consumption of the producer exceeds a certain threshold.

Finally, we can note that a commercially available implementation of what is essentially Flat PARLOG is the language Strand [Foster and Taylor 1989, 1990]. A further development of Strand is PCN [Foster and Tuecke 1991].



Upper-left to lower-right diagonals 0, 0, 0, 0, 0, 1, 2, 5, 6, 4, 4, 6, 7, 7, 8, 8, 5, 6, 6, 6, 6, 5, 5, 3, 3, 1, 1, 1, 0, 0
Upper-right to lower-left diagonals 0, 1, 2, 3, 3, 3, 2, 4, 5, 5, 7, 9, 9, 6, 6, 5, 4, 5, 5, 4, 5, 4, 2, 2, 2, 2, 1, 0, 0, 0

Figure 8.3. A sample scanner grid

8.3 AKL VS. CONSTRAINT LOGIC PROGRAMMING

In this section, it is demonstrated by examples how AKL goes beyond basic constraint logic programming (CLP).

When constraint programming in AKL, the underlying constraint system serves as the basic terminology, which may stand for a lesser or a greater portion of the solution depending on its strength. But AKL may also be used to program constraint solving. For example, finite domain constraint solving can be expressed in AKL with the constraint system of trees (implied by work by Bahgat and Gregory [1989] and Gregory and Yang [1992]). Whereas the logical content of the corresponding constraints could also be expressed in, e.g., a constraint logic programming (CLP) language, the necessary constraint propagation would not be achieved.

In Section 3.5, a solution of the N-Queens problem illustrated programmable constraint satisfaction techniques. It employed a combination of the short-circuit and the mutual exclusion variable techniques to express the *exactly-one* constraint, and the propagation effect achieved corresponds to that of *forward checking* in CLP [Van Hentenryck 1989].

To show techniques applicable also in many other situations, a new example is used, called the Scanner[1]. The problem is to reveal the contents of a grid. Each square may be filled or non-filled. The input is the number of filled squares along each row, column, and diagonal. In general, there may be no, one, or several solutions for a given input. Figure 8.3 shows a sample input and solution.

---

[1] The Scanner was implemented in AKL by Johan Montelius. It was originally described by A.K. Dewdney in Scientific American (September, 1990, p. 124).

Analogously to how the exactly-one and at-most-one constraints were used for the N-Queens problem, the Scanner problem can be reduced to an *exactly-N* constraint, which is used to express all constraints.

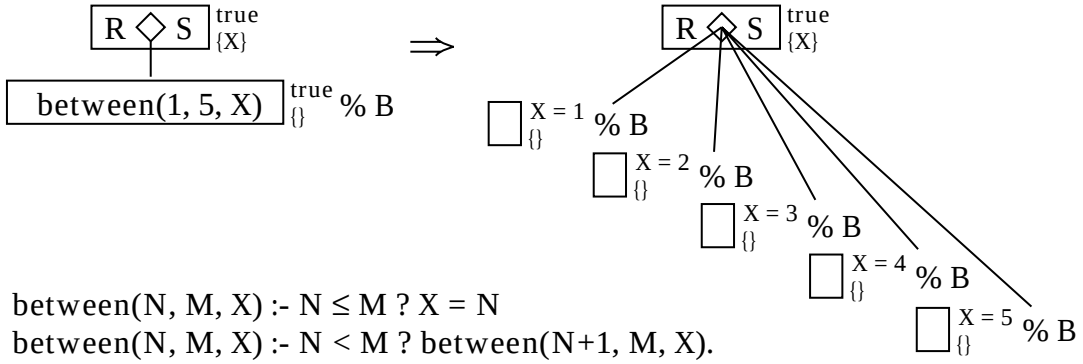


Figure 8.4. Local enumeration of possible values

The constraint exactly_n(N, K, L) is read “N out of a total K elements in the list L are 1; the rest are 0”. The propagation effect desired is that whenever N elements are known to be 1, the rest are known to be 0, and whenever K-N are known to be 0, the rest are known to be 1. In the grid in Figure 8.3, we would first know that there are no filled squares in column 1; we would then know that the rest of the squares in the top-left to bottom-right diagonal were filled; we would then know that the rest of the squares in the right-most and bottom-most rows were blank; and so on.

This constraint cannot be expressed in the same simple way as the exactly-one constraint. Two techniques will be shown, that of *local execution* and the very general *monitor-controller* technique. But first we specify the relation in almost CLP style and discuss its properties.

```
exactly_n(0, 0, []).
exactly_n(N, K, [1 | L]) :-
        N > 0
    ?   exactly_n(N-1, K-1, L).
exactly_n(N, K, [0 | L]) :-
        K > N
    ?   exactly_n(N, K-1, L).
```

Logically, it should be clear that the above program expresses the exactly-N constraint. It does not, however, perform propagation as desired. As written, an exactly_n call suspends if the first element of the list is not given. The values of other elements cannot be taken into account, and the two desired forms of propagations are not performed.

Instead of suspending, we could of course consider generating all possible lists. *Local execution* means enumerating alternatives for a variable (or a group of variables) locally in the guard of a nondeterminate choice statement (Figure 8.4). Propagation will be automatic by the pruning of failed alternatives. When the

choice becomes determinate, that solution will be promoted, and propagation will occur.

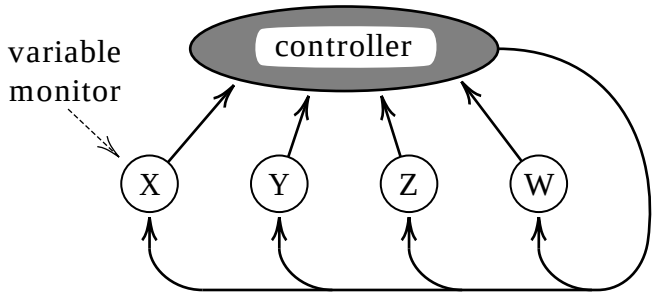


Figure 8.5. Monitors and controller

The exactly-N constraint can be expressed as follows using the local execution technique.

```
exactly_n(N, K, L) :-
        exactly_n_aux(N, K, L)
    ?   true.

exactly_n_aux(0, 0, L) :-
    ?   L = [].
exactly_n_aux(N, K, L0) :-
        N > 0
    ?   L0 = [1 | L],
        exactly_n_aux(N-1, K-1, L).
exactly_n_aux(N, K, L0) :-
        K > N
    ?   L0 = [0 | L],
        exactly_n_aux(N, K-1, L).
```

Again, it should be clear that the above program expresses the exactly-N constraint, the only differences being the placement of the goal in the guard and the moving of constraints on the list to the body. It does perform propagation as desired, but at a great cost. If nothing is known about the list, $\binom{N}{K}$ alternative lists are created. Thus, in this case, local execution should typically not be used (but see Section 8.3.2 for a case where it can).

Monitor-controller means having a monitor process for each variable, which reports back to a controller when certain constraints are told on its variable (Figure 8.5). The controller is able to propagate information back to the monitors, which are usually addressed collectively by broadcasting.

The exactly-N constraint can be expressed as follows using the monitor-controller technique.

```
exactly_n(N, K, L) :=
        open_port(P, S),
        spawn_monitors(L, P, B),
        controller(S, N, K, B).
```

```
spawn_monitors([], P, B) :-
    → true.
spawn_monitors([V | L], P, B) :-
    → monitor(V, P, B),
      spawn_monitors(L, P, B).

monitor(V, P, B) :-
      data(V)
    | send(V, P).
monitor(V, P, B) :-
      data(B)
    | V = B,
      send(V, P).

controller(S, 0, K, B) :-
    | B = 0,
      turned(S, K, 0).
controller(S, K, K, B) :-
    | B = 1,
      turned(S, K, 1).
controller([1 | S], N, K, B) :-
      N > 0
    | controller(S, N-1, K-1, B).
controller([1 | S], N, K, B) :-
      K > N
    | controller(S, N, K-1, B).

turned(0, S, V) :-
    → S = [].
turned(N, [V | S], V):-
      N > 0
    → turned(N-1, S, V).
```

This program solves the problem illustrated in Figure 8.3 deterministically.

8.3.1 The Cardinality Operator

An elegant generalisation of the monitor-controller technique is the implementation in AKL of the *cardinality operator* [Van Hentenryck and Deville 1991]. The computation rules given are straight-forwardly realised in AKL as the different clauses of a controller.

The cardinality operator is an agent #(L, U, Cs), where L and U are natural numbers and Cs is a given list of (representations of) constraints. It succeeds if the number of constraints that are true is in the range L to U, the lower and upper bounds. Furthermore, its propagation effect is that if it at some point is known that the remaining constraints must be all false, or all true, this knowledge is published immediately.

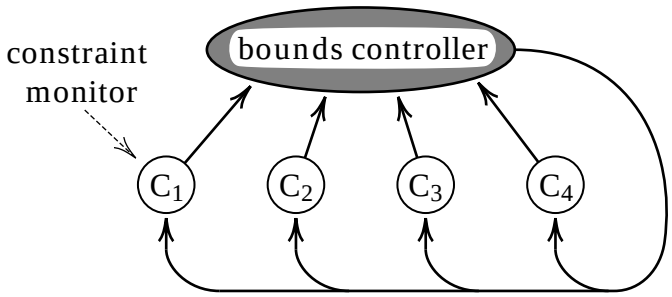


Figure 8.6. Monitor-controller for the cardinality operator

Clearly, the at-most-one, at-least-one, and exactly-one constraints are subsumed by this general construct, which is expressed in AKL as follows.

```
#(L, U, Cs) :=
        open_port(P, S),
        constraints(Cs, N, P, F),
        bounds(L, U, N, S, F).
```

The *trivial satisfaction* rule:

```
bounds(L, U, N, S, F) :-
        L =< 0, N =< U
    |   F = any.
```

The *positive satisfaction rule*:

```
bounds(L, U, N, S, F) :-
        L =< U, L = N
    |   F = true,
        allequal(N, true, S).
```

The *negative satisfaction rule*:

```
bounds(L, U, N, S, F) :-
        L =< U, U = 0
    |   F = false,
        allequal(N, false, S).
```

The *positive reduction rule*:

```
bounds(L, U, N, [true | S], F) :-
        ( 0 < L, L < N, L =< U
        ; 0 < U, U < N, L =< 0 )
    |   bounds(L-1, U-1, N-1, S, F).
```

The *negative reduction rule*:

```
bounds(L, U, N, [false | S], F) :-
        ( 0 < L, L < N, L =< U
        ; 0 < U, U < N, L =< 0 )
    |   bounds(L, U, N-1, S, F).
```

The constraints agent spawn monitors for the constraints.

```
constraints([], 0, _, F) :-
    → true.
constraints([C | Cs], N, P, F) :-
    → constraint(C, P, F),
      constraints(Cs, N1, P, F),
      N = N1+1.
```

For simplicity, we assume that the representation of constraints is callable. We also use general negation to express the published negative constraint. This could be replaced by some general means of interpreting the representation.

```
constraint(C, P, F) :-
        call(C)
    |   send(true, P).
constraint(C, P, F) :-
        not(call(C))
    |   send(false, P).
constraint(C, P, true) :-
    |   call(C).
constraint(C, P, false) :-
    |   not(call(C)).
constraint(C, P, any) :-
    |   true.
```

The allequal agent verifies that all incoming messages have the same value as that broadcast to the remaining monitors.

```
allequal(0, X, S) :-
    → S = [].
allequal(N, X, S) :-
        N > 0
    → allequal(N-1, X, S1),
      S = [X | S1].
```

8.3.2 Local Search

Closely related to the notion of independent and-parallelism achieved by local execution in guards is a technique for writing search programs in AKL.

In this example, due to D.H.D. Warren, we illustrate both the incremental growth of or-trees in AKL, and how the notion of local execution in guards makes it possible to write elegant search programs. It is a program that finds common sub-lists of two lists. We start by giving a Prolog/CLP program for this task.

```
sublist([], Y).
sublist([E | X], [E | Z]) :- sublist(X, Z).
sublist(X, [E | Y]) :- X = [_ | _], sublist(X, Y).

?- sublist(L, [c,a,t,s]), sublist(L, [l,a,s,t]).
```

This program will repeat the execution of the second goal for each solution of the first goal. One could imagine evaluating the first or the second goal in advance, and combining its results with the solutions of the other. But this is not a good solution. The goals have a large number of solutions, many of which are completely irrelevant. Clearly, there is a trade-off between the drawback of repeating work and the drawback of wasting work.

However, it is reasonably safe to execute each goal locally until the first point at which the execution for this goal could fail (in an "interesting" way). This happens when the first element of a sublist is generated. To achieve this effect, the above definition is transformed into the following.

```
sublist([], Y).
sublist([E | X], Y) :- suffix([E | Z], Y) ? sublist(X, Z).

suffix(X, X).
suffix(X, [E | Y]) :- suffix(X, Y).
```

Let us start with the initial goal

```
sublist(L, [c,a,t,s]), sublist(L, [l,a,s,t])
```

The sublist goals are first unfolded (into the implicit nondeterminate choice statements).

```
( L=[] ? true
; E, L1, Z : L=[E | L1], suffix([E | Z], [c, a, t, s]) ? sublist(L1, Z) ),
( L=[] ? true
; E, L1, Z : L=[E | L1], suffix([E | Z], [l, a, s, t]) ? sublist(L1, Z) )
```

The guards (preceding the wait operators "?") will now execute locally, finding all solutions. Without getting into details, it should be fairly intuitive that, after a while, the following configuration is reached.

```
( L = [] ? true
; L1 : L = [c | L1] ? sublist(L1, [a, t, s])
; L1 : L = [a | L1] ? sublist(L1, [t, s])
; L1 : L = [t | L1] ? sublist(L1, [s])
; L1 : L = [s | L1] ? sublist(L1, []) ),
( L = [] ? true
; L1 : L = [l | L1] ? sublist(L1, [a, s, t])
; L1 : L = [a | L1] ? sublist(L1, [s, t])
; L1 : L = [s | L1] ? sublist(L1, [t])
; L1 : L = [t | L1] ? sublist(L1, []) )
```

The solutions that have been found for the suffix/2 goal give rise to different alternative guarded goals in the choice statement. At this stage, these alternatives have to be tried. Further computation will eventually lead to a "Cartesian product" of the alternatives in the two choice statement. When the combinations that fail have been removed, the combined alternatives are as follows.

$L = []$

$L = [a \mid L_1]$, sublist(L_1, [t, s]), sublist(L_1, [s, t])

$L = [t \mid L_1]$, sublist(L_1, [s]), sublist(L_1, [])

$L = [s \mid L_1]$, sublist(L_1, []), sublist(L_1, [t])

We have one complete solution L = []. Execution will now continue within each of the remaining alternatives, but we will stop here, hopefully having demonstrated the main point.

8.4 AKL VS. THE CC FRAMEWORK

The concept of *concurrent constraint programming*, due to seminal work by Saraswat [1989; Saraswat and Rinard 1990; Saraswat, Rinard, and Panangaden 1991], evolved from earlier work in constraint logic programming [Jaffar and Lassez 1987] and in concurrent logic programming [Maher 1987; Saraswat 1987]. One source of inspiration was the logical view of committed-choice language provided by Maher [1987], where the entailment-based view of synchronisation was introduced.

For his analysis of the concurrent constraint programming paradigm, Saraswat developed a family of languages called the cc framework, which are in many ways the immediate ancestors of AKL. Saraswat has pursued the study of these languages in new directions, for example basing them on Linear Logic [Girard 1987], but the comparison made here focuses on his original cc framework [Saraswat 1989].

The core of the cc family is a language called cc(↓,→,⇒), providing blocking Ask "↓", Atomic Tell "★", indeterminate (don't care) prefixing "→", and nondeterministic (don't know) prefixing "⇒". The cc family is open-ended, in that other combinators can be conceived and included, such as Eventual Tell and groups, which are of interest for the following comparison with AKL.

Many of the ideas in the cc languages recur in AKL, although rearranged in a smaller set of higher level primitives with improved control and synchronisation. Saraswat proposed a control for cc languages which is based on the Basic Andorra Model (Section 8.1.8). AKL generalises this form of control by also taking into account the interactions between constraint stores in the hierarchy formed by the nesting of choice and bagof statements. AKL control would also be applicable to cc languages.

In the following sections the most important combinators of the cc family are compared with corresponding constructs in AKL, and finally a mapping from a restriction of the language cc(↓,*,→,⇒) to AKL is shown.

8.4.1 Basic Operations

Some basic operations correspond to implicit operations on constraints in AKL.

The *Ask* ("↓") operation corresponds closely to the ask operation implicitly employed by AKL for various purposes. For example, the behaviour of a cc agent

($c\downarrow \rightarrow A$) may be translated into AKL as ($c \rightarrow A^*$), where * is a mapping from cc agents to AKL statements. The difference between this and the cc operation is that asking is done with respect to local variables, which are existentially quantified. An AKL agent ($x : c(x) \mid A(x)$) asks $\exists x\, c(x)$, and may then use x in A. This behaviour can be achieved with a cc agent ($(\exists x\, c(x))\downarrow \rightarrow (c(x)* \rightarrow A(x))$), but at the cost of repetition, which, apart from being verbose, could correspond to similar repetitious behaviour in the manipulation of constraints in an implementation.

The *Eventual Tell* ("$*$") operation is similar to the tell operation implicitly employed by AKL. The difference is that the told constraints are not buffered in AKL; instead, the inconsistency of the resulting constraint store may be detected at a later time. AKL uses telling on constraint atoms and for promotion of constraints computed in guards.

The *Atomic Tell* ("$\star$") operation has no corresponding operation in AKL. Similar effects can be achieved with the *cut* operation (proposed as an extension of AKL in Section 4.8). This operation requires that the state is stable. There is also an ordering of agents in AKL which will allow no competition in this situation: the leftmost will be chosen. Some uses of atomic tell can be emulated, but with less concurrency, and hence less potential parallelism. The most important reason for abstaining from atomic tell is the difficulty of efficient parallel implementation. The atomicity requirement means that an arbitrary number of variables have to be locked. With the stability condition it is guaranteed that no other agents can interfere, but at the cost of less concurrency.

Saraswat also proposes so called *atomic groups*: the "guard" and the "wfguard" constructs. They are basic operations, and may thus be used as the prefix of a prefixing operation. Their definition is not compatible with the rest of the framework (without modification), but intuitively they provide user-defined atomic transactions. AKL guard execution is always atomic in this sense, due to the separation of constraint stores. The implementation scheme proposed for atomic groups involves copying the entire constraint store, and is hardly realistic for the intended uses.

In addition the cc framework has the Global Ask, Ask-and-Instantiate, Ask-and-fix, Inform, and Check basic operations, which have no counterparts in AKL, in which the constraint operations are monotone and do not introduce nondeterminism.

8.4.2 Prefixing Combinators

Prefixing combinators are similar to the guard operators of AKL. The comparisons are done with respect to the choice statements corresponding to these guard operators.

Nondeterministic prefixing ("$\Rightarrow$") is similar to nondeterminate choice in AKL. Any basic operation may be used as prefix. When the operation in the prefix succeeds, a don't know nondeterministic split may be performed, with no additional synchronisation requirements. This is to be compared with AKL which

uses a form of telling in the guard of the nondeterminate choice statement, and which may only perform don't know nondeterministic actions in stable states, when nothing can be done to avoid it. Nondeterministic prefixing is "parallel" in that the different alternative computations may be explored concurrently. Saraswat also proposes a sequential variety, which explores one alternative at a time, the others being blocked until it terminates. This does not correspond to anything in AKL; the reasons for including it or avoiding it are similar to those for sequential composition.

Indeterminate prefixing ("→") is similar to committed choice in AKL. Again, any basic operation may be used as prefix. The effect achieved when Ask is used was discussed above. The effect achieved when Atomic Tell is used was also discussed above. The effect achieved when Eventual Tell is used is not likely to be useful.

8.4.3 Coarse-Grained Combinators

Saraswat recognises the need to encapsulate nondeterministic computations, and proposes a few coarse-grained interleaved combinators for this purpose.

The *single-solution group* ("oneg") is similar to a guard of a committed choice statement in that it selects a single solution of an otherwise encapsulated computation. The differences are that there is no entailment requirement, nor a "continuation" in the form of a body, which performs a particular action with respect to the chosen solution. The lack of synchronisation makes the choice of solution more arbitrary than necessary.

The *all-solution group* ("allg") is similar to a guard of a nondeterminate choice statement in that all solutions of an otherwise encapsulated computation will be tried nondeterministically. Again, the lack of synchronisation makes the choice of solutions to be tried quite arbitrary.

The *reconciliation* combinator ("⇑") provides the ability to take the "Cartesian product" of two deadlocked groups (of the same kind). A related effect can be achieved in AKL by explicit decomposition of problems into what should be executed locally, and what should be a residue to be combined (Section 8.6).

The *synthesise* combinator ("synthesize") communicates nothing until all solutions of an encapsulated computation have been found, whereupon it combines these into a single disjunctive constraint, which is told to the store. No corresponding operation is available in AKL (although a number of candidates have been discussed).

The definition of the above combinators is not compatible with the rest of the framework (without modification). The resulting computation model would have a flavour reminiscent of AKL, in that it provides a hierarchy of constraint stores.

8.4.4 Essential Combining Forms

And-parallel composition ("||") corresponds to composition in AKL.

Or-selection ("□") corresponds to, if anything, the ";" operator separating the guarded statements in choice statements. In AKL the guarded statements in a choice statement must all be of the same kind, i.e., the behaviour is associated with the choice statement. Although the cc framework allows combinations of different prefixing combinators, this ability is not much used.

Existential quantification ("^") corresponds to hiding in AKL.

8.4.5 cc Programs in AKL

We will map a subset of cc(↓,∗,→,⇒), with Eventual Tell instead of the usual Atomic Tell, to corresponding AKL programs. We assume the algebraic syntax of cc programs in the sense that or-selection separates the clauses in a definition, otherwise the standard syntax is used. We further observe that, with Eventual Tell, $(c\downarrow : c* \rightarrow A)$ is equivalent to $(c\downarrow \rightarrow (c* \rightarrow A))$.

Thus, we assume definitions in the following form.

$$A ::= \text{stop} \mid \text{fail} \mid g \mid A \parallel A$$

$$B ::= c* \mid c\downarrow$$

$$D ::= X\,\hat{}\,D \mid B \Rightarrow A \mid B \rightarrow A \mid D \,\square\, D$$

$$K ::= g :: D$$

We further add the restriction that all clauses in a definition must have the same prefixing combinator. The mapping * assigns a corresponding AKL program to a given cc(↓,∗,→,⇒) program with the given restriction. (Please excuse the overloading of the "⇒" arrow.)

$$(\text{stop} \mid \text{fail} \mid g)^* \Rightarrow (\text{true} \mid \text{fail} \mid g)$$

$$(A \parallel A)^* \Rightarrow (A^*, A^*)$$

$$(c\downarrow \rightarrow A)^* \Rightarrow c \mid A^*$$

$$(c* \rightarrow A)^* \Rightarrow \text{true} \mid c, A^*$$

$$(c\downarrow \Rightarrow A)^* \Rightarrow (c \mid \text{true})\,?\,A^*$$

$$(c* \Rightarrow A)^* \Rightarrow c\,?\,A^*$$

$$(X\,\hat{}\,D)^* \Rightarrow (X : D^*)$$

$$(D_1 \,\square\, \cdots \,\square\, D_n)^* \Rightarrow (D_1{}^* ; \cdots ; D_n{}^*)$$

$$(g :: D)^* \Rightarrow g := D^*$$

For example, the cc program

```
max(X, Y, Z) ::
    ( X ≤ Y↓ ⇒ equal(Y, Z)
    □ Y ≤ X↓ ⇒ equal(X, Z) ).
```

```
equal(Y, Z) ::
    (Y = Z)* ⇒ stop.
```

translates into AKL as

```
max(X, Y, Z) :=
    ( (X ≤ Y | true) ? equal(Y, Z)
    ; (Y ≤ X | true) ? equal(X, Z) ).

equal(Y, Z) :=
    ( (Y = Z) ? true ).
```

The reason for asking in cc is often synchronisation. In AKL, the program would be written as

```
max(X, Y, Z) :=
    ( X ≤ Y % Y = Z
    ; Y ≤ X % X = Z ).
```

where '%' is either → (normally), | (hardly), or ? (possibly), depending on the desired behaviour. In the latter case synchronisation would be given by the nondeterminacy of the choice.

8.4.6 Conclusion

AKL extends the expressive power of the cc family by improving the functionality which was aimed at by the coarse-grained combinators. Improved control, given by stability, and aggregates such as bagof, enable the use in AKL of don't know nondeterminism in otherwise reactive programs.

Saraswat has reconstructed the cc family in terms of Linear Logic [Girard 1987], yielding the Lcc and HLcc family of *linear* and *higher-order linear* concurrent constraint languages. This clearly adds expressiveness, but possibly at the price of sacrificing simplicity. The path chosen for AKL is to augment the language where a need has been perceived, such as adding *ports* for process communication, which rhymes well with the constraint-based communication scheme.

8.5 AKL VS. OZ

Oz is a deep-guard higher-order concurrent constraint programming language based on the constraint system of records [Smolka, Henz, and Würtz 1993; Smolka 1994; Smolka et al. 1994]. Oz is a descendant of AKL, and is its closest relative, but goes beyond it by providing new functionality such as *higher-orderness* and *processes* (threads). The standard formulation of Oz does not, however, provide don't know nondeterminism, although such extensions are being considered, and will surely become part of the language [Schulte and Smolka 1994].

Kernel Oz, the essential sublanguage to which other constructs can be reduced, has the following syntax.

E	::=	**false** | **true** | $x = s$	constraints
	|	$x = l(y_1 \ldots y_n)$	tuple construction
	|	$x = l(l_1 : y_1 \ldots l_n : y_n)$	record construction
	|	**proc** $\{ x \ \overline{y} \}$ E **end**	procedure definition
	|	$\{ x \ \overline{y} \}$	procedure application
	|	$E_1 \ E_2$	concurrent composition
	|	**local** $\overline{x}$ **in** E **end**	variable declaration
	|	**if** C_1 [] … [] C_n **else** E **fi**	conditional
	|	**or** C_1 [] … [] C_n **ro**	disjunction
	|	**process** E **end**	process creation
C	::=	$\overline{x}$ **in** E_1 **then** E_2	clause
x, y, z	::=	⟨variable⟩	
l	::=	x | ⟨atom⟩	
s	::=	l | ⟨number⟩	
$\overline{x}, \ \overline{y}$	::=	⟨possibly empty sequence of variable⟩	

Kernel procedures provide for constraint communication and additional constraint operations. The operational semantics is given by the Oz calculus, which is a rewrite system modulo structural congruence [Smolka 1994]. The Oz user language provides a plethora of syntax for different modes of expression in an ALGOL-like style.

The Oz conditional is a combination of AKL committed and conditional choice. It may commit to any of the clauses, but if all fail, the else-branch is chosen. It can be modelled in AKL in a manner analogous to the treatment of the sequential and parallel clause composition of PARLOG.

The Oz disjunction is similar to AKL nondeterminate choice, but will promote not only a clause with a solved guard, but any single remaining clause. This exact behaviour cannot be modelled in AKL. In the standard formulation, Oz does not have choice splitting, but in the extension for encapsulated search, disjunction plays the rôle of nondeterminate choice for splitting operations. This extension does not, however, make don't know nondeterminism orthogonal as in AKL. Splitting does not distribute over guards, but is only performed in a special *solve* statement. This makes don't know nondeterminism unsuitable for uses such as simple tests in guards, and the language loses some logic programming flavour. The intended use of the solve statement is that different branches of a search tree are explored by concurrent composition (“conjunction”). This makes it difficult to model single solution search without restricting the potential for parallelism, a problem shared with the committed-choice languages [Gregory 1993]. It is, however, also possible to explore the different branches in the guards of a conditional, in which case this problem does not arise. Finally, it is not clear how to achieve independent and-parallelism for search problems. The above comments pertain to a proposed extension, and may not be true of the final version [Schulte and Smolka 1994].

The essence of the Oz higher-order procedure definitions and applications can be modelled in AKL as discussed in Section 3.4.1. (Oz, however, makes different instances of abstractions unequal using its ability to generate unique names.) In a language based on higher-orderness such as Oz, abstractions play a different rôle, however. For example, the extensive support for object-oriented programming offered by Oz relies on it. There is no need for a separate concept of modules, since variable hiding works for procedure names. The view of a program is more flexible.

The put and get operations of Oz constraint communication can be modelled by ports in the same way as the put and take operations of Id M-structures in Section 7.7.4.

The Oz process is a way of assigning a “virtual processor” to an expression. Each expression belongs to a process, and each process is given an opportunity to make progress in a round-robin fashion, much the same way as processes in an operating system. AKL as presented has no processes. A similar effect could be achieved by stipulating that the execution model should be fair, but this would not be enough for practical purposes (as discussed in Section 5.8).

It does not seem altogether unlikely that AKL, in the near future, will evolve in the direction of Oz, and support records, higher-orderness, and processes.

AD LIBITUM [1]

In the following, AKL is assessed in the form of a dialogue between researchers who represent quite different approaches to research in programming language design and implementation, as will be seen. Any resemblance between these fictional researchers and real persons is purely coincidental.

Cast:

S. Worker (a lowly graduate student)

Prof. H. A. Riddle (his supervisor)

Sir Cheswhat (a crusader against *ad hoc*'ery)

Prof. Warden (a guardian of relics)

S. Worker is giving a presentation of AKL, likely to continue for hours due to frequent interruptions by Prof. Riddle, clarifying various points of interest. He has just commented upon the language design, presented early in the talk. We enter the ensuing discussion …

Sir Cheswhat: … but what is your design methodology?

S. Worker: Our starting points were the languages Prolog and GHC, and the conceptual framework provided by concurrent constraint programming.

We tried to combine their essence, operationally, in a manner that allowed using the programming paradigms of both as well as exploiting the potential for parallelism of both. As usual we wanted the result to be simple, efficient, and expressive – you know, the usual stuff …

---

[1] **ad libitum** (L.) at pleasure

Sir Cheswhat: Yes, I know, but this makes your design entirely *ad hoc*[1]!

S. Worker: Indeed! AKL was designed *for this* very purpose. The AKL computation model was designed to make the synthesis of Prolog and GHC capabilities as smooth as possible. This could not have been achieved by using another, otherwise more general, formalism.

Prof. Warden: You claim that AKL is simple, but I just cannot agree. Comparing with Prolog, you introduce notions such as guard operators, suspension, and not least concurrency. This can hardly be called simple!

S. Worker: AKL is based on comparatively few notions. Based on these, we can model most of the functionality of modern dialects of Prolog. They are not simple. They provide different forms of side effects. They provide cut, inside disjunction and inside call. Many also provide suspension, which interacts strangely with other language features. Compared with this, I call AKL simple.

Sir Cheswhat: I don't understand this preoccupation with Prolog and GHC. Why not simply recognise that they are both deficient languages of questionable parentage and that it is high time to move to a new beginning?

S. Worker: We are interested in innovation but also in consolidation. With Prolog, commercially available since many years and with a large community of users, and GHC, the basis of the fifth generation project in Japan, are associated many well-developed programming techniques, as well as a large number of applications. AKL encompasses both, while making a number of improvements. One could call it backward compatibility.

Sir Cheswhat: But is this science?

S. Worker: The premise that two given languages are to be synthesised in this manner does not make the problem easier nor less interesting. The underlying principles and implementation technology needed to support such a combination are quite novel, and of general interest for related languages. As regards the scientific method, the corroboration of these ideas involves making them suitably precise, and, in particular, to verify that they meet pragmatic requirements in terms of expressiveness and implementation efficiency.

Prof. Warden: Can you actually run all Prolog programs in AKL?

S. Worker: Not in the way I believe you mean. Whereas GHC is harboured within AKL as a syntactic subset, the exact behaviour of the execution-order sensitive side effect and metalogical operations of Prolog is not faithfully replicated.

---

[1] **ad hoc** (L.) for this

This doesn't mean that corresponding programs can't be written. It simply means that they have to be written manually, whereas the majority of Prolog programs can be translated into AKL automatically, with the aid of data-flow analysis techniques.

Prof. Warden: If backward compatibility is a concern, shouldn't you give Prolog at least the same amount of attention as GHC? After all, Prolog is by far the more wide-spread language.

S. Worker: The decision was a compromise between innovation and consolidation. It would be possible to augment the AKL computation model with notions of side effects and metalogical operations, and their sequential execution, and thereby achieve better compatibility, but such augmentations would stand out clearly as foreign to the spirit of the model, and would also complicate implementations.

Instead, the augmentation with the process-oriented paradigm provides much better ways of doing things like input/output and manipulation of state. In particular, it does so in a way that interacts with nondeterminism in a meaningful way, something which can't be said about side effect operations.

Prof. Warden: But control in AKL is so explicit; this makes programming less declarative. I would prefer implicit control which exploits, intelligently, the logical structure of the problem representation.

S. Worker: AKL provides a strong logical reading for a large class of programs. To understand what such a program (potentially) computes, its declarative reading is sufficient. To understand how it is done, to understand the algorithmic behaviour of a program, an operational understanding is necessary, and there are simple computation rules to follow to get this understanding.

If the program text is changed, the behaviour is more or less changed, even if the "logic" remains the same. The behaviour of a program is not given by its declarative semantics. An "implicit" control regime may have weak points, where small changes lead to notable differences. That a programmer is not expected to understand the details of the computation model makes the problem worse. In AKL, a highly predictable control regime has been chosen, which the programmer can understand and control.

Prof. Warden: But even so, the control for a particular program is not what I want. For example, can you get these behaviours in AKL?

(He writes down a few small don't know nondeterministic logic programs and an outline of their desired behaviours on a piece of paper.)

(Some time for deliberation passes.)

S. Worker: Well, it seems that you can, for these examples, but in some cases my formulation is somewhat distorted, due to the need to express the particular control desired.

In fact, I wouldn't be surprised if it occasionally turned out to be very difficult to program a particular behaviour, in the worst case at the cost of working entirely at the metalevel, on a representation of the original logic program.

In many ways, the functionality you are looking for is that of an intelligent problem solver. AKL is a programming language, which supports the programming of problem solving programs.

(S. then continues his talk, hoping to get home in time for dinner.)

* * *

S. Worker has managed to fend off most interruptions, and has covered a major section of his talk, the description of the various programming paradigms provided by AKL, when Prof. Riddle cannot hold back his enthusiasm any longer …

Prof. Riddle (spreading his arms in a wide gesture): You see? What did I tell you? You can do anything in AKL!

Sir Cheswhat: In fact, I would have been more surprised if you couldn't write all programs in AKL, considering that it is likely to be Turing complete…

S. Worker: This is true, but the point made is that some languages that provide certain features, e.g., parallelism or don't know nondeterminism, lack the ability to use them to their full potential.

First, not all languages intended for parallel implementation allow arbitrary parallel random access algorithms to be expressed with appropriate efficiency. Prolog is not intended as a general purpose language for parallel machines. GHC is, but lacks the ability to simulate a PRAM efficiently, as do "declarative" languages in general. AKL solves the problem through *ports*, an extension in the spirit of the rest of the language, which can be added to GHC as well.

Second, not all languages with don't know nondeterminism have means to *encapsulate* nondeterminism within a suitable part of a program. In Prolog, nondeterminism is global. For example, the state of a user-interface to a nondeterministic program has to be stored in the database. Backtracking will erase any other state the program might have. AKL provides single and all solutions encapsulation in the form of *deep guards* and bagof, and more controllable encapsulation in the form of *engines.*

Sir Cheswhat: What are these deep guards anyway? Why do you think you need them? Wouldn't in fact engines suffice for encapsulation purposes?

S. Worker: Engines are very powerful, and can be added to any language which can model a state, even to Prolog with freeze, but they certainly

do not replace deep guards. Guards provide user-definable conditions and negation. They also allow us to use don't know nondeterminism at a much smaller scale than that of program modules.

Prof. Warden: I always thought that deep guards were inefficient, and that this was the reason for moving, for example, from *full* GHC to *flat* GHC?

S. Worker: There is an overhead, but it is very small. It is comparable to the overhead of supporting backtracking in Prolog. I believe that no scheme for encapsulating nondeterminism could be made cheaper. Since we provide encapsulated nondeterminism, we can also provide deep guards.

It is possible to virtually eliminate even this overhead for code executing outside any encapsulation, at the price of possibly having two compiled versions of agents which are used inside and outside encapsulation, respectively. The gain would probably be negligible.

Prof. Warden: You say that Prolog is essentially provided, but isn't a program with deep conditional guards more speculative than the corresponding Prolog program with cut?

S. Worker: It is possible, but only in contrived programs, and then only if the implementation is in conflict with our recommendations.

Sir Cheswhat: I can't give my favourite *xyz* semantics to AKL!

S. Worker: We can give different semantics, or readings, to different subsets of AKL. If a particular reading is important in a certain context, then stick to the corresponding subset. AKL's lack of an *xyz* semantics for the whole language does not in any way prevent, nor does it seem to make more difficult, formal manipulations such as data flow analyses and program transformations. If and when languages with *xyz* semantics prove their overall advantages, we will have to reconsider our design.

Sir Cheswhat: I still believe one could do much better. There are a number of unexplored possibilities, which could give us the expressiveness you are looking for while resting upon a firm semantical foundation.

S. Worker: I hope you are right! But we are not there yet. Meanwhile, AKL is definitely a step forward compared to Prolog, GHC, and many other languages.

Sir Cheswhat: Well, it's all there in my *Tractatus* anyway…

Prof. Warden: I still don't understand what's wrong with Prolog…

(Here the exhausted researchers break for the evening…)

BIBLIOGRAPHY

Abreu, Salvador, Luís Moniz Pereira, and Philippe Codognet [1992]. Improving backward execution in the Andorra family of languages. In *Logic Programming: Proceedings of the Joint International Conference and Symposium on Logic Programming*. The MIT Press.

Agha, Gul [1986]. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press.

Agha, Gul and Carl Hewitt [1987]. Actors: a conceptual foundation for concurrent object-oriented programming. In [Shriver and Wegner 1987].

Aït-Kaci, Hassan [1991]. *Warren's Abstract Machine: A Tutorial Reconstruction*. The MIT Press.

Aït-Kaci, Hassan, Andreas Podelski, and Gert Smolka [1992]. A feature-based constraint system for logic programming with entailment. In *Fifth Generation Computer Systems 1992*. IOS Press.

Ali, Khayri M. and Roland Karlsson [1990]. The Muse approach to or-parallel Prolog. *International Journal of Parallel Programming* **19**(2):129-162. (Also in [Karlsson 1992].)

Armstrong, Joe L., S. Robert Virding, and Mike C. Williams [1991]. Erlang user's guide & reference manual, version 3.1. Computer Science Laboratory, Ellemtel Telecommunications Systems Laboratories, Älvsjö, Sweden.

Armstrong, Joe L., Mike C. Williams, and S. Robert Virding [1993]. Concurrent Programming in Erlang. Prentice Hall.

Bahgat, Reem and Steve Gregory [1989]. Pandora: non-deterministic parallel logic programming. In *Logic Programming*: *Proceedings of the Sixth International Conference*. The MIT Press.

Bahgat, Reem [1991]. Pandora: non-deterministic parallel logic programming, Ph.D. diss., Department of Computing, Imperial College of Science and Technology, London.

Barth, Paul S., Rishiyur S. Nikhil, and Arvind [1991]. M-structures: extending a parallel, non-strict, functional language with state. In *Functional Programming and Computer Architecture '91*.

Brand, Per [1994]. Decision graph compilation, Draft SICS Research Report, Swedish Institute of Computer Science.

Bueno, Francisco and Manuel Hermenegildo [1992]. An automatic translation scheme from Prolog to the Andorra Kernel Language. In *Fifth Generation Computer Systems 1992*. IOS Press.

Carlson, Björn, Mats Carlsson, and Daniel Diaz [1994]. Entailment of finite domain constraints. In *Logic Programming: Proceedings of the Eleventh International Conference*. The MIT Press. (To appear.)

Carlson, Björn, Seif Haridi, and Sverker Janson [1994]. AKL(FD)—A concurrent language for FD programming. Unpublished manuscript. Swedish Institute of Computer Science.

Carlsson, Mats [1990]. Design and implementation of an or-parallel Prolog engine. Ph.D. diss., RIT(KTH) TRITA-CS-9003, Department of Telecommunication and Computer Systems, The Royal Institute of Technology, Stockholm, and SICS Dissertation Series 02, Swedish Institute of Computer Science.

Carlsson, Mats, Johan Andersson, Stefan Andersson, Kent Boortz, Hans Nilsson, Thomas Sjöland, and Johan Widén [1993]. SICStus Prolog user's manual, SICS Technical Report T93:01, Swedish Institute of Computer Science.

Carriero, Nicholas and David Gelernter [1989]. How to write parallel programs: a guide to the perplexed. *ACM Computing Surveys* **21**(3):323–357.

Cheng, M. H. M., M. H. van Emden, and B. E. Richards [1990]. On Warren's method for functional programming in logic. In *Logic Programming: Proceedings of the Seventh International Conference*. The MIT Press.

Chikayama, Takashi and Y. Kimura [1987]. Multiple reference management in Flat GHC. In *Logic Programming: Proceedings of the Fourth International Conference*. MIT Press.

Chikayama, Takashi [1992]. Operating system PIMOS and kernel language KL1. In *Fifth Generation Computer Systems 1992*. IOS Press.

Chikayama, Takashi, Tetsuro Fujise, and Hiroshi Yashiro [1993]. A portable and reasonably efficient implementation of KL1 (poster abstract). In *Logic Programming: Proceedings of the Tenth International Conference on Logic Programming*. The MIT Press.

Clark, Keith L. and Steve Gregory [1981]. A relational language for parallel programming. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*. ACM Press. (Reprinted in [Shapiro 1987].)

Clark, Keith L. and Steve Gregory [1986]. PARLOG: parallel programming in logic. *ACM Transactions on Programming Languages and Systems* **8**(1). (Revised version in [Shapiro 1987].)

Clinger, W. D. [1981]. Foundations of Actor semantics. AI-TR-633, MIT Artificial Intelligence Laboratory.

Clocksin, William F. and Christopher S. Mellish [1987]. *Programming in Prolog*. 3rd ed. Springer-Verlag.

Crammond, Jim [1990]. Scheduling and variable assignment in the parallel Parlog implementation. In *Logic Programming: Proceedings of the 1990 North American Conference*. The MIT Press.

Davison, Andrew [1989]. POLKA: A PARLOG object-oriented language, Ph.D. diss., Department of Computing, Imperial College, London.

DeGroot, Doug [1984]. Restricted and-parallelism. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1984*. Ohmsha, Ltd., Japan.

Foster, Ian and Stephen Taylor [1989]. Strand: a practical parallel programming language. In *Logic Programming: Proceedings of the North American Conference 1989*. MIT Press.

Foster, Ian and Stephen Taylor [1990]. *Strand, New Concepts in Parallel Programming*. Eaglewood Cliffs.

Foster, Ian and Steven Tuecke [1991]. Parallel programming with PCN, version 1.2. Technical Report ANL-91/32, Argonne National Laboratory, University of Chicago.

Foster, Ian and Will Winsborough [1991]. Copy avoidance through compile-time analysis and local reuse. In *Logic Programming: Proceedings of the 1991 International Symposium*. The MIT Press.

Franzén, Torkel [1991]. Logical aspects of the Andorra Kernel Language. SICS Research Report R91:12, Swedish Institute of Computer Science.

Franzén, Torkel [1994] Some formal aspects of AKL. Draft SICS Research Report, Swedish Institute of Computer Science.

Franzén, Torkel, Seif Haridi, and Sverker Janson [1992]. An overview of AKL. In *ELP'91 Extensions of Logic Programming*. LNAI (LNCS) 596, Springer-Verlag.

Gabriel, Richard P [1994]. Lisp: good news, bad news, how to win big. Lucid, Inc. Unpublished.

Gelernter, David, Nicholas Carriero, S. Chandran, and S. Chang [1985]. Parallel programming in Linda. In *Proceedings of the International Conference on Parallel Processing* (St. Charles, Ill., Aug.). IEEE Computer Society Press.

Girard, Jean-Yves [1987]. Linear logic. *Journal of Theoretical Computer Science* **50**(1).

Goldberg, Yaron, William Silverman, and Ehud Shapiro [1992]. Logic programs with inheritance. In *Fifth Generation Computer Systems 1992*. IOS Press.

Gregory, Steve [1987]. *Parallel Logic Programming in PARLOG*. Addison-Wesley.

Gregory, Steve and Rong Yang [1992]. Parallel constraint solving in Andorra-I. In *Fifth Generation Computer Systems 1992*. IOS Press.

Gregory, Steve [1993]. Experiments with speculative parallelism in Parlog. In *Logic Programming: Proceedings of the 1993 International Symposium*. The MIT Press.

Gudeman, David, Koenraad De Bosschere, and Saumya K. Debray [1992]. jc: an efficient and portable sequential implementation of Janus. In *Logic Programming: Proceedings of the Joint International Conference and Symposium on Logic Programming*. The MIT Press.

Gupta, Gopal and Manuel Hermenegildo [1991]. ACE: and/or-parallel copying-based execution of logic programs, Technical Report TR-91-25, Department of Computer Science, University of Bristol.

Gupta, Gopal and Manuel Hermenegildo [1992]. Recomputation-based implementations of and/or-parallel Prolog. In *Fifth Generation Computer Systems 1992*. IOS Press.

Gupta, Gopal, Vitor Santos Costa, Rong Yang, and Manuel Hermenegildo [1991]. IDIOM: a model integrating dependent-, independent-, and or-parallelism. In *Logic Programming: Proceedings of the 1991 International Symposium*. The MIT Press.

Haridi, Seif and Per Brand [1988]. Andorra Prolog, an integration of Prolog and committed choice languages. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*. Ohmsha, Ltd., Japan.

Haridi, Seif [1990]. A logic programming language based on the Andorra model. *Journal of New Generation Computing* **8**(7):109–125. (Revised version of [Haridi and Brand 1988].)

Haridi, Seif and Sverker Janson [1990]. Kernel Andorra Prolog and its computation model. In *Logic Programming: Proceedings of the Seventh International Conference*. The MIT Press. (Revised version of SICS Research Report R90002.)

Haridi, Seif and Catuscia Palamidessi [1991]. Structural operational semantics for Kernel Andorra Prolog. In *Proceedings of PARLE'91*. LNCS, Springer-Verlag.

Haridi, Seif, Sverker Janson, and Catuscia Palamidessi [1992]. Structural operational semantics for AKL. *Journal of Future Generation Computer Systems* **8**(1992).

Hermenegildo, Manuel and K. Greene [1990]. &-Prolog and its performance: exploiting independent and-parallelism. In *Logic Programming: Proceedings of the Seventh International Conference*. The MIT Press.

Hewitt, Carl E. and Henry Baker [1977]. Actors and continuous functionals. In *Proceedings IFIP Working Conference on Formal Description of Programming Concepts*.

Hudak, Paul and Philip Wadler [1991], eds. Report on the programming language Haskell: a non-strict, purely functional language, version 1.0. Yale University and University of Glasgow. (haskell@cs.yale.edu, haskell@cs.glasgow.ac.uk)

Jaffar, Joxan and Jean-Louis Lassez [1987]. Constraint logic programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press.

Janson, Sverker and Seif Haridi [1991]. Programming paradigms of the Andorra Kernel Language. In *Logic Programming: Proceedings of the 1991 International Symposium*. The MIT Press. (Revised version of SICS Research Report R91:08.)

Janson, Sverker and Johan Montelius [1992]. The design of a prototype implementation of the Andorra Kernel Language. Public Deliverable Report, ESPRIT project PEPMA (EP 2471).

Janson, Sverker, Johan Montelius, and Seif Haridi [1993]. Ports for objects. In *Research Directions in Concurrent Object-Oriented Programming*, MIT Press.

Janson, Sverker and Seif Haridi [1994]. An introduction to AKL—a multiparadigm programming language. In *Constraint Programming*, NATO-ASI Series, Springer-Verlag.

Janson, Sverker, Johan Montelius, Kent Boortz, Per Brand, Björn Carlson, Ralph Clarke Haygood, Björn Danielsson, and Seif Haridi [1994]. AGENTS user manual, Draft SICS Research Report, Swedish Institute of Computer Science.

Kahn, Kenneth M., Eric Dean Tribble, Mark S. Miller, and Daniel G. Bobrow [1987]. Vulcan: logical concurrent objects. In [Shriver and Wegner 1987]. (Also in [Shapiro 1987].)

Kahn, Kenneth M. and Vijay A. Saraswat [1990]. Actors as a special case of concurrent constraint programming. In *OOPSLA/ECOOP '90 Conference Proceedings*. ACM Press.

Karlsson, Roland [1992]. A high-performance or-parallel Prolog system, Ph.D. diss., TRITA-TCS-LPS-9202, Department of Telecommunication and Computer Systems, The Royal Institute of Technology, Stockholm, and SICS Dissertation Series 07, Swedish Institute of Computer Science.

Keisu, Torbjörn [1994]. Tree Constraints, Draft Ph.D. diss., Department of Teleinformatics, The Royal Institute of Technology, Stockholm, and Swedish Institute of Computer Science.

Lassez, Jean-Louis, Michael J. Maher, and Kimbal G. Marriott [1988]. Unification revisited. In *Foundations of Deductive Databases and Logic Programming*. Morgan-Kauffman.

Lloyd, John W. [1989]. *Foundations of Logic Programming*. 3d ed. Springer-Verlag.

Lusk, Ewing, R. Butler, T. Disz, R. Olson, R. Overbeek, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, S. Haridi, P. Brand, M. Carlsson, A. Ciepilewski, B. Hausman [1988]. The Aurora or-parallel Prolog system. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*. Ohmsha, Ltd., Japan. (Also in [Carlsson 1990].)

Maher, Michael J. [1987]. Logic semantics for a class of committed choice programs. In *Logic Programming: Proceedings of the Fourth International Conference*. The MIT Press.

Maher, Michael J. [1988]. Complete axiomatizations of the algebra of finite, rational and infinite trees, Technical Report, IBM Thomas J. Watson Research Center.

Mariën, André and Bart Demoen [1991]. A New Scheme for Unification in the WAM. In *Logic Programming: Proceedings of the 1991 International Symposium*. The MIT Press.

Meier, Micha [1991]. Recursion vs. iteration in Prolog. In *Logic Programming: Proceedings of the Eighth International Conference*. The MIT Press.

Miyazaki, Toshihiko, Akikazu Takeuchi, and Takashi Chikayama [1985]. A sequential implementation of Concurrent Prolog based on the shallow binding scheme. In *1985 Symposium on Logic Programming*. IEEE Computer Society Press.

Montelius, Johan, and Khayri M. Ali [1994]. An and/or-parallel abstract machine for AKL (extended abstract). In Parallel Logic Programming and its Programming Environments, Technical Report CIS-TR-94-04, University of Oregon.

Moolenaar, Remco and Bart Demoen [1993]. A parallel implementation for AKL, In *Programming Language Implementation and Logic Programming* (*PLILP '93*). Springer-Verlag.

Moolenaar, Remco and Bart Demoen [1994]. Hybrid tree search in the Andorra model. In *Logic Programming: Proceedings of the Eleventh International Conference*. The MIT Press. (To appear.)

Naish, Lee [1988]. Parallelizing NU-Prolog. In *Logic Programming: Proceedings of the Fifth International Conference and Symposium*. The MIT Press.

O'Keefe, Richard A. [1990]. *The Craft of Prolog*. The MIT Press.

Palmer, Doug and Lee Naish [1991]. NUA-Prolog: an extension to the WAM for parallel Andorra. In *Logic Programming: Proceedings of the Eighth International Conference*. The MIT Press.

Plotkin, Gordon D. [1981]. A structural approach to operational semantics, DAIMI FN-19, Computer Science Department, Aarhus University.

Santos Costa, Vitor, David H. D. Warren, Rong Yang [1991a]. The Andorra-I engine: a parallel implementation of the Basic Andorra Model. In *Logic Programming: Proceedings of the Eighth International Conference*. The MIT Press.

Santos Costa, Vitor, David H. D. Warren, and Rong Yang [1991b]. The Andorra-I preprocessor: supporting full Prolog on the Basic Andorra Model. In *Logic Programming: Proceedings of the Eighth International Conference*. The MIT Press.

Saraswat, Vijay A. [1987a]. The concurrent logic programming language CP: definition and operational semantics. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press.

Saraswat, Vijay A. [1987b]. CP as a general-purpose constraint language. In *Proceedings of the National Conference on Artificial Intelligence*. American Association of Artificial Intelligence.

Saraswat, Vijay A. [1987c]. The language GHC: operational semantics and comparison with CP(↓, |). In *Proceedings 1987 Symposium on Logic Programming*. IEEE Computer Society Press.

Saraswat, Vijay A. [1989]. Concurrent constraint programming languages, Ph.D. diss., Carnegie-Mellon University. (Also available in Doctoral Dissertation Award and Logic Programming Series, MIT Press, 1993.)

Saraswat, Vijay A. and Martin Rinard [1990]. Concurrent constraint programming. In *Conference Record of the Seventh Annual ACM Symposium on Principles of Programming Languages*. ACM Press.

Saraswat, Vijay A., Martin Rinard, and Prakash Panangaden [1991]. Semantic foundation of concurrent constraint programming. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press.

Schulte, Christian, and Gert Smolka [1994]. Encapsulated search in Oz, Draft Research Report, DFKI.

Shapiro, Ehud [1983]. A subset of Concurrent Prolog and its interpreter, Technical Report TR-003, ICOT. (Revised version in [Shapiro 1987].)

Shapiro, Ehud and Akikazu Takeuchi [1983]. Object-oriented programming in Concurrent Prolog. *Journal of New Generation Computing* **1**(1). (Revised version in [Shapiro 1987]).

Shapiro, Ehud and Colin Mierowsky [1984]. Fair, biased, and self-balancing merge-operators: their specification and implementation in Concurrent Prolog. *Journal of New Generation Computing* **2**(3). (Also in [Shapiro 1987].)

Shapiro, Ehud [1986a]. Concurrent Prolog: a progress report. *IEEE Computer* **8**(19). (Also in [Shapiro 1987].)

Shapiro, Ehud [1986b]. A test for the adequacy of a language for an architecture, Technical Report CS86-01, Weizmann Institute of Science. (Revised version in [Shapiro 1987].)

Shapiro, Ehud and Shmuel Safra [1986]. Multiway merge with constant delay in Concurrent Prolog. *Journal of New Generation Computing* **4**(3). (Revised version in [Shapiro 1987].)

Shapiro, Ehud [1987], ed. *Concurrent Prolog: Collected Papers*. 2 vols. The MIT Press.

Shapiro, Ehud [1989]. The family of concurrent logic programming languages. *ACM Computing Surveys* **21**(3):412–510.

Shriver, Bruce and Peter Wegner [1987], eds. *Research Directions in Object-Oriented Programming*. The MIT Press.

Smolka, Gert and Ralf Treinen [1992]. Records for logic programming. In *Logic Programming: Proceedings of the Joint International Conference and Symposium on Logic Programming*. The MIT Press.

Smolka, Gert [1993]. Residuation and guarded rules for constraint logic programming. In *Constraint Logic Programming: Selected Research*. The MIT Press.

Smolka, Gert, Martin Henz, and Jörg Würtz [1993]. Object-oriented concurrent constraint programming in Oz, Research Report RR-93-16, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany.

Smolka, Gert [1994]. A calculus for higher-order concurrent constraint programming with deep guards, Research Report RR-94-03, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany.

Smolka, Gert, et al. [1994]. The Oz handbook, Research Report, German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany.

Snyder, A., W. Hill and W. A. Olthoss [1989]. A glossary of common object-oriented terminology, STL-89-26, Software Technology Laboratory, Hewlett-Packard Laboratories.

Sterling, Leon and Ehud Shapiro [1986]. *The Art of Prolog*. The MIT Press.

Taylor, Stephen [1989]. *Parallel logic programming techniques*. Eaglewood Cliffs.

Tick, Evan [1991]. *Parallel Logic Programming*. The MIT Press.

Tribble, Eric Dean, Mark S. Miller, Kenneth M. Kahn, Daniel G. Bobrow, Curtis Abbott, and Ehud Shapiro [1987]. Channels: a generalisation of streams. In *Logic Programming: Proceedings of the Fourth International Conference*. The MIT Press. (Also in [Shapiro 1987].)

Ueda, Kazunori [1985]. Guarded Horn Clauses, TR-103, ICOT, Tokyo. (Revised version in [Shapiro 1987]).

Ueda, Kazunori and Takashi Chikayama [1990]. Design of the kernel language for the parallel inference machine. *The Computer Journal* **33**(6).

Ueda, Kazunori and Masao Morita [1992]. Message-oriented parallel implementation of moded Flat GHC. In *Fifth Generation Computer Systems 1992*. IOS Press.

Van Hentenryck, Pascal [1989]. *Constraint Satisfaction in Logic Programming*. The MIT Press.

Van Hentenryck, Pascal and Yves Deville [1991]. The cardinality operator: a new logical connective and its application to constraint logic programming. In *Logic Programming: Proceedings of the Eighth International Conference*. The MIT Press.

Van Hentenryck, Pascal, Vijay Saraswat, and Yves Deville [1992]. Constraint logic programming over finite domains: the design, implementation, and applications of cc(FD), Technical report, Computer Science Department, Brown University.

Wadler, Philip [1990]. Comprehending monads. In *ACM Conference on Lisp and Functional Programming*. ACM Press.

Warren, David H. D. [1982]. Higher-order extensions to Prolog: are they needed? In *Machine Intelligence 10*, Ellis Horwood with John Wiley and sons, Lecture Notes in Mathematics 125.

Warren, David H. D. [1983]. An abstract Prolog instruction set, Technical Note #309, Artificial Intelligence Center, SRI International.

Warren, David H. D. [1989]. The Extended Andorra Model with implicit control. Presentation at the ICLP'90 Preconference Workshop on Parallel Logic Programming in Eilat, Israel. Workshop record available from SICS, Box 1263, S-164 28 KISTA, Sweden.

Yang, Rong [1987]. *P-Prolog – A Parallel Logic Programming Language*. Series in Computer Science, Vol. 9, World Scientific.

Yang, Rong [1989]. Solving simple substitution ciphers in Andorra-I. In *Logic Programming*: *Proceedings of Sixth International Conference*. The MIT Press.

Yoshida, Kaoru and Takashi Chikayama [1988]. A'UM – A stream-based concurrent object-oriented language. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1988*. Ohmsha, Ltd., Japan.

INDEX

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Computing Science Department, Uppsala University
Uppsala Theses in Computing Science (Doctoral Dissertations)

03. Göran Hagert, *Logic Modeling of Conceptual Structures: Steps Towards a Computational Theory of Reasoning*, 1986
09. Jonas Barklund, *Parallel Unification*, 1990
10. Håkan Millroth, *Reforming Compilation of Logic Programs*, 1990
15. Andreas Hamfelt, *Metalogic Representation of Multilayered Knowledge*, 1992
19. Sverker Janson, *AKL—A Multiparadigm Programming Language*, 1994

Swedish Institute of Computer Science
SICS Dissertation Series

01. Bogumil Hausman, *Pruning and Speculative Work in OR-parallel PROLOG*, 1990
02. Mats Carlsson, *Design and Implementation of an OR-parallel Prolog Engine*, 1990
03. Nabiel A. Elshiewy, *Robust Coordinated Reactive Computing in SANDRA*, 1990
04. Dan Sahlin, *An Automatic Partial Evaluator for Full Prolog*, 1991
05. Hans A. Hansson, *Time and Probability in Formal Design of Distributed Systems*, 1991
06. Peter Sjödin, *From LOTOS Specifications to Distributed Implementations*, 1991
07. Roland Karlsson, *A High Performance OR-parallel Prolog System*, 1991
08. Erik Hagersten, *Toward Scalable Cache Only Memory Architectures*, 1991
09. Lars-Henrik Eriksson, *Finitary Partial Inductive Definitions and General Logic*, 1993
10. Mats Björkman, *Architectures for High Performance Communication*, 1993
11. Steven Pink, *Measurement, Implementation and Optimization of Internet Protocols*, 1993
12. Martin Aronsson, *GCLA: The Design, Use and Implementation of a Program Development System*, 1993
13. Christer Samuelsson, *Fast Natural-Language Parsing Using Explanation-Based Learning*, 1994
14. Sverker Janson, *AKL—A Multiparadigm Programming Language*, 1994

AKL
A Multiparadigm Programming Language
Based on a Concurrent Constraint Framework

Printed by Graphic Systems, Stockholm, Sweden, using a camera-ready copy typeset in Adobe Palatino by the author, using a LaserWriter Pro 810, Microsoft Word 5.1, Adobe Illustrator 5.0, and the Macintosh Quadra 700 and Macintosh PowerBook 170 personal computers.