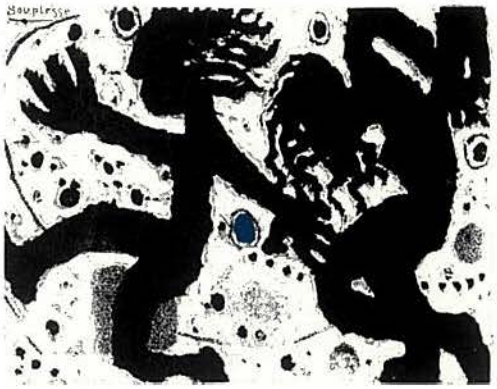
DLP

A language for distributed logic programming



A. Eliëns

Design, semantics and implementation

Stellingen behorende bij het proefschrift van A. Eliëns

Stellingen

1. De informatica zou baat hebben bij een lijst met (de) tien open problemen van het vakgebied.

2. Formalismen die naast een declaratieve component een besturingscomponent met de functionaliteit van een programmeertaal bieden verdienen de voorkeur voor de ontwikkeling van expertsystemen. Vergelijk [M. Bezem, *Consistency of rule-based expert systems*, CADE-9, LNCS 310 Springer (1988) pp. 151-162], [A. Eliëns, *Expert systems as deductive systems*, Centre for Mathematics and Computer Science CS-R8825], en dit proefschrift.

3. Het object-georiënteerde paradigma is in hoge mate onafhankelijk van de computationele aspecten van een programmeertaal. Vergelijk [A. Snyder, *Encapsulation and inheritance in object oriented programming languages*, Proc. OOPSLA 1986, pp. 31-45]

4. En/of bomen dienen nader onderzocht te worden als model voor de representatie van parallelle berekeningen met non-deterministisch communicatiegedrag. Vergelijk [A. Eliëns, *Semantics for Occam*, Centre for Mathematics and Computer Science CS-N8606]

5. Het bewijs van de equivalentie van de operationele en denotationele semantiek van de taal $\mathcal{B}_2$, een abstractie van DLP voor het modelleren van gedistribueerd backtracken, rust op de vergelijking

   $$\mathcal{C}[\![< \alpha, F >: C]\!] = \mathcal{C}[\![C]\!] \rhd_\alpha \mathcal{F}[\![F]\!]\alpha$$

   waarin syntactische proces-continuaties C gerelateerd worden aan hun semantische tegenhangers $\mathcal{C}[\![C]\!]$ middels de insertie-operator $\rhd_\alpha$.
   Vergelijk [Hoofdstuk 6 van dit proefschrift]

6. Het gemak waarmee een taalconcept door een programmeur gehanteerd wordt laat zich niet relateren aan de ingewikkeldheid van de semantische beschrijving daarvan. Vergelijk [P. America, *The practical importance of formal semantics*, in: J.W. de Bakker, 25 jaar semantiek, Liber Amicorum, Centre for Mathematics and Computer Science (1989) pp. 31-40]

7. Programmacorrectheid wordt bevorderd door hulpmiddelen die de programmeur in staat stellen leesbare programma's te schrijven. Vergelijk [D. Knuth *Literate programming*, The Computer Journal, Vol. 27, No. 2, 1984, pp. 97-111]

8. De productie van kunst laat zich niet mechaniseren. Vergelijk [A. Eliëns, *Computational Art*, Suppl. Issue on Electronic Art of Leonardo da Vinci, Pergamon Press, 1988, pp. 21-25]

9. Het schematische denken vormt een wezenlijk onderdeel van het creatieve proces. Vergelijk [A. Eliëns, *Creativity, reflection and involvement*, Proc. 9th Int. Conf. of Aesthetics, Dubrovnik 1980, pp. 263-267]

10. De (maatschappelijke) erkenning van de verzorgende aspecten van het vaderschap noodzaakt tot een nadere overweging van de arbeidsmoraal.

DLP - A language for Distributed Logic Programming

Design, semantics and implementation

DLP - A language for Distributed Logic Programming

Design, semantics and implementation

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Universiteit van Amsterdam,
op gezag van de Rector Magnificus
prof. dr. P.W.M. de Meijer
in het openbaar te verdedigen in de Aula der Universiteit
(Oude Lutherse Kerk, ingang Singel 411, hoek Spui),
op vrijdag 15 februari 1991 te 15.00 uur

door ANTONIUS PETRUS WILHELMUS ELIËNS
geboren te 's-Hertogenbosch

Promotores:	prof. dr. P. Klint
	prof. dr. J.W. de Bakker
Co-promotor:	dr. J.J.M.M. Rutten
Faculteit:	Wiskunde en Informatica

Acknowledgements I thank my promotores, Paul Klint for his encouraging attitude during the process of writing this thesis, and Jaco de Bakker for taking an early interest in my work. Together with Jan Rutten, my co-promotor, they discussed the successive drafts of my thesis. Their amiable scrutiny significantly improved the presentation of my ideas.

Acknowledgements are due to the members of the reading comittee, Luís Monteiro for being willing to judge my work, Peter van Emde Boas for common dilations on the subject of my thesis, and Marc Bezem for imposing high scientific standards.

Further I like to mention Carel van den Berg for giving technical support, Peter Lucas for sharing his knowledge of expert systems, Jean-Marie Jacquet for reading the manuscript, Wouter Mettrop for bibliographical explorations, and, as friends and colleagues, Alban Ponse and Louis Kossen.

This work has been carried out at the Centre for Mathematics and Computer Science. I have enjoyed working there.

The cover design is by E.C. Wielenga.

Preface

This thesis deals with distributed logic programming, from a variety of perspectives. In particular it concerns the extension of logic programming (LP) with notions of object oriented programming (OO) and parallelism (||). Its theme can be summarized by the pseudo-equation

$$\text{DLP} = \text{LP} + \text{OO} + ||$$

as will become clear along the way. Partly by coincidence and partly because of a deliberate choice, this treatment of distributed logic programming is not restricted to a study of how to integrate the semantic foundations of logic programming, object oriented programming and parallelism into a unifying framework. Such a framework, at least to a large extent, has already been developed by J.W. de Bakker and his co-workers. Cf. [America and Rutten, 1989]. Rather, the present study deals with issues of language design, formal semantics, and applications in the area of distributed knowledge based systems.

The starting point of this investigation of distributed logic programming was the development of a prototype system extending Prolog with primitives for parallelism. As the system evolved gradually a shift in emphasis occurred toward integrating object oriented features. Finally, the effort of developing a prototype has resulted in a language proposal, embodying the concept of a distributed logic programming language.

The final proposal has been rather influenced by the concepts developed for the language POOL [America, 1987]. With some lenience one could speak of the language DLP, introduced in this book, as being a member of a family of POOL-like languages. The major difference between the distributed logic programming language to be presented here and the language POOL, as dealt with in [America and Rutten, 1989], however is that the latter takes a Pascal-like language as its base language. The most obvious and perhaps important problem that arises concerns the (distributed) backtracking induced by the choice for a Prolog-like language as the base language. Nevertheless, with regard to the formal semantics, the resemblance with POOL enables in a significant degree to lean on the foundational efforts presented in [America and Rutten, 1989]. In developing the formal semantics for distributed logic programming, another significant influence has been the research reported in [de Vink, 1989] and [de Bruin and de Vink, 1988], dealing with operational and denotational (continuation) semantics for Prolog with cut.

A common (metric) framework that allows to integrate the topics of process creation, communication and backtracking is conceived in [de Bakker, 1988]. The work on the formal semantics of distributed logic programming presented may be regarded as a (non-trivial) application of the foundational results presented in the sources mentioned before. The primary concern here has been to use formal semantic techniques to elucidate the dynamics of distributed logic programming, notably process creation and communication in the presence of backtracking.

The design of a programming language is a delicate issue. Apart from the possibility of providing a formal semantics, there remains a rather subjective criterion of a more aesthetic nature. Is it pleasing to use the language? And for what applications? For the author the answer to the first question is naturally yes. Nonetheless, a first attempt has been made to convince the reader of the applicability and the expressiveness of the language by presenting a number of examples dealing with problems in parallel programming, knowledge representation and distributed problem solving. A distributed medical (toy) expert system has been developed to illustrate the power of our language.

How intricate the relation between formal semantics and actual systems can be is demonstrated by the fact that the core of the prototype consists of an interpreter based on a continuation semantics for sequential Prolog, adapted from [Allison, 1986]. We shown how the actual implementation of the Prolog interpreter is derived from this particular semantics; and also what extensions (read hacks, but positively) are needed to arrive at the full functionality of the prototype system. Many loose ends can still be detected in this transition from a formal semantics to an actual system. Perhaps the only valuable remark that can be made here is that the pleasure in experimenting with such a complex prototype system was definitely enhanced by taking a well understood formal description of a constituting part of the final system as a starting point.

An (almost) complete listing of the code of the DLP prototype is contained in the appendix. From a certain perspective, one may regard this part as the actual thesis, and the rest of the book as providing additional comment.

A leading interest behind this work has been the question of parallelism in expert system reasoning. Distributed logic programming seems to offer a solution, in that it seems a suitable vehicle for the implementation of knowledge based systems, including expert systems, and systems for distributed problem solving. Moreover, I wish to state my conviction that the object oriented programming paradigm will grow in importance, both for knowledge representation and the implementation of knowledge based systems. Parallelism, moreover is a very desirable feature, also for AI-applications. I regard this extension of Prolog to a distributed logic programming language as a first study in augmenting a logic based language with constructs for parallel object oriented programming. I am looking forward to similar extensions of, say, a resolution based theorem prover like LMA/ITP [Wos et al, 1984].

Contents

Part I

Design

Chapter 1

Introduction

- The activity of the intuition consists in making spontaneous judgements which are not the results of conscious trains of reasoning ... -
A. Turing, from Andrew Hodges The Enigma of Intelligence

Distributed logic programming (DLP) may be characterized by the pseudo-equation

$$\text{DLP} = \text{LP} + \text{OO} + \|$$

stating that distributed logic programming combines logic programming (LP), object oriented programming (OO) and parallelism (||).

The application area we have in mind for DLP encompasses the implementation of knowledge-based systems, including expert systems and systems for distributed problem solving.

Logic Programming, because of its declarative nature, has established itself as an important implementation method for problems in Artificial Intelligence, knowledge based systems, and in particular expert systems. C.f. [Butler Cox, 1983]. Its major representative is Prolog. See [Clocksin and Mellish, 1981].

Object Oriented Programming has recently become a paradigm for developing complex systems. Objects integrate *data* and *methods* that operate on these data in a protected way. This allows to organize a program as a collection of objects, representing the conceptual structure of the problem. Part of the popularity of object oriented languages is due to the facilities for code sharing as offered by the inheritance mechanism. See [Wegner, 1987].

Parallelism is somehow of independent interest. Most of the developments in parallel logic programming are based on exploiting the parallelism inherent in the computation model of logic programming languages. Parallelism in DLP is achieved by extending the notion of object to that of a process, in a similar fashion as the approach taken for POOL [America, 1987].

In developing the language DLP we have strived for compatibility with the backtracking behavior of Prolog. Our approach is in this respect distinct from the efforts of extending concurrent logic programming languages with object oriented features, since these support only the so-called *don't care*, or committed choice, non-determinism inherited from their base languages, instead of the *don't know* non-determinism of Prolog.

In the next section we present the basic notions around which DLP is built. A summary of the contents of this thesis is given in section 1.2.

1.1 Objects, processes and communication

Objects play a central role in DLP, both as a means for modularization and in providing protection for (local) data. Non-logical variables may be used to store the data encapsulated by an object. The clauses defined for an object act as methods, in that they have exclusive access to these data.
Apart from clauses for methods, so-called *constructor* clauses may be declared that specify the own activity of an object. An object declaration in DLP is of the form

object *name* {
var *variables.*
constructor clauses
method clauses
}

Constructor clauses are used when creating a new instance of a declared object, to start the own activity of the newly created object. Objects without constructor clauses are passive objects, without activity of their own. Each instance of an object has private copies of the declared non-logical variables.

Inheritance in DLP is static. When in a declaration the name of the object is followed by one or more object names, as in name : base, then both the non-logical variables and the clauses of the object base are copied into (each instance of) the object name.

Processes come into existence on two occasions. When an object is created a *constructor* process is started, executing the own activity of an object. When an object receives a method call, a process is created for handling the rendez-vous, that is for evaluating the goal and to return the answer substitutions to the invoking process. Both constructor processes and processes evaluating a goal are sequential Prolog processes.

A method call may result in multiple answer substitutions. Backtracking over these answer substitutions produced in a rendez-vous is done lazily, initiated by

the invoking process. We call this backtracking *global* since multiple processes are involved.

An object may state its willingness to answer a method call by means of an *accept statement.* When an object is not willing to answer a method call, the evaluation of the goal is suspended until the object is willing to do so.

The language DLP is an extension of Prolog with a number of special forms for dealing with non-logical variables, object creation and for engaging in a rendez-vous.[1]
The special forms include:

- $x := t$ - to assign t as a value to the non-logical variable x
- $O = new(c(t))$ - to create a new active instance of c, evaluating $c(t)$
- $O!m(t)$ - to call the method $m(t)$ of object O
- $accept(m_1, ..., m_n)$ - to answer one of the methods $m_1, ..., m_n$

In the list above t stands for a term, x for a non-logical variable, O for a logical variable, c for an object name, and $m, m_1, ..., m_n$ for method names.

Each occurrence of a non-logical variable x is reduced to the value of x, unless it occurs on the left hand side of an assignment. We also support the simplification of arithmetical expressions.

The evaluation of $O = new(c(t))$ results, apart from the creation of a new instance of object c for which the constructor $c(t)$ is evaluated, in binding O to a pointer to the object.

Communication takes place by means of a (synchronous) rendez-vous. A method call is of the form $O!m(t)$, where O must be bound to an object. When the object is willing to accept the method, s indicated by the occurrence of a goal $accept(..., m, ..)$, a process is created for evaluating the call. The calling process waits until it receives an answer, or a message of failure. Since non-logical (instance) variables may be assigned values to, no new evaluation process is started for the object, when the call is accepted, until the first answer or failure is delivered. While backtracking over alternative answer substitutions, other method calls may become active. The distinction between objects and processes is necessitated by the possibility of global (distributed) backtracking. When only local backtracking would be allowed, the evaluation of a method call could be done by the object itself.

An example of an object is given in the following declaration.

```
object ctr {
var n = 0.
ctr() :- accept(any), ctr().
```

[1] We treat here only the subset of DLP that we consider most important. An overview of DLP is contained in chapter 3.

```
inc() :- n := n + 1.
value(N) :- N = n.
}
```

The use of such a counter is illustrated by the goal

```
:-
    O = new(ctr()),
    O!inc(),
    O!value(X).
```

where a counter is created that is subsequently asked to answer the method calls *inc*() and *value*(*X*). Evaluating the goal $O = new(ctr())$ results in the creation of a new instance of the declared object, evaluating the constructor. The constructor of the counter states that the object is prepared to answer any call, indefinitely.

The DLP target machine is assumed to be a parallel processor consisting of a limited number (less than 100, say) of processor nodes that are connected with each other by a packet switching network, in such a way that the distance between each node never exceeds a fixed number (3 or 4) of intermediate nodes. For reasons of optimal utilization such machines are considered to support coarse grain parallelism, which means that the ratio of communication and computation must be in favor of the latter. For effectively using such a processor we have included facilities for the allocation of objects and processes.

1.2 An overview of this thesis

Our research covers the design, the semantics and the implementation of the language DLP. This book is divided in twelve chapters distributed over three parts. We have included an appendix that contains an almost complete listing of the prototype implementation.

Part I: Design

1. *Introduction*: Some motivational background is given. The notions of dynamic process creation, communication and backtracking are introduced.

2. *Extending Prolog to a parallel object oriented language*: We introduce a number of constructs for distributed logic programming and illustrate these by examples taken from the literature dealing with parallel programming. Among other celebrities the classic Dining Philosophers appear in our gallery.

3. *The language DLP*: A summary is given of the constructs by which DLP extends Prolog. Further we isolate the subsets to which we will devote a semantic study in part II.

4. *Knowledge representation and inheritance*: It is shown how knowledge representation problems can be solved with the object oriented features, inheritance and message-delegation, as provided by the DLP language. As an application we present the implementation of a distributed medical (toy) expert system

in DLP, illustrating the principles of hierarchic knowledge organization and distributed inference.

5. *Design perspectives*: We will reflect on the motivations underlying our approach to distributed logic programming. In addition we will discuss some of the related work on integrating the paradigms of object oriented programming and logic programming.

Part II: Semantics

6. *Process creation and communication in the presence of backtracking*: We introduce the technical framework needed for our semantic enterprise. Moreover, we present a semantic treatment of the major phenomena figuring in distributed logic programming.

7. *An abstract version of DLP and its operational semantics*: We will give a detailed operational characterization of the core of DLP. In contrast to the previous chapter, our description here includes the (logic) programming aspects of DLP.

8. *Comparative semantics for DLP*: We extend the semantic study started in chapter 6 to the various subsets of DLP, as selected in chapter 3.

Part III: Implementation

9. *An implementation model for DLP*: We sketch how we have combined the computation models of object oriented programming and logic programming in our implementation of DLP. We also give a short overview of the actual prototype implementation discussed in chapter 11.

10. *Deriving a Prolog interpreter*: The implementation of the Prolog part of DLP has been derived from a formal continuation semantics for Prolog. We treat this semantics and discuss the derivation of the code for the interpreter.

11. *The implementation of the prototype*: A rather detailed description of the implementation of the prototype is given. This description corresponds to the listing of the code given in the appendix.

12. *Conclusions and future research*: We draw some conclusions on the current status of the language and its implementation. Further, we provide some recommendations for improving the efficiency of the DLP system; and explore the possibility of providing a declarative semantics for a distributed logic programming language like DLP.

Chapter 2

Extending Prolog to a parallel object oriented language

- A sceptical lady patient has a rather long dream, in which certain persons tell her of my book on Wit, and praise it highly. Then something is said about a 'channel', perhaps another book in which 'channel' occurs, or something else to do with 'channel'... she doesn't know; it is quite vague -
Sigmund Freud, The interpretation of dreams

We will investigate a number of constructs that may be useful for extending Prolog to a language suited for parallel and distributed computation, fitting in the framework imposed by the object oriented programming paradigm. The constructs introduced are all incorporated in the language DLP, of which an overview is given in chapter 3. This chapter is of an exploratory nature. We will reflect on the design considerations in chapter 5.

In section 2.1 we will introduce objects as a means for modularization, and for encapsulating data that are accessible by methods defined as clauses. We will briefly discuss inheritance among objects. A distinction is made between passive and active objects that are objects having own activity. Section 2.2 may be regarded as an intermezzo exploring communication between active objects via channels. In section 2.3 we treat a synchronous rendez-vous mechanism for handling method calls to active objects. A discussion of the distributed backtracking that may occur in a rendez-vous is given in section 2.4. Section 2.5 deals with a construct for the conditional acceptance of method calls. In section 2.6 we treat

some low level primitives for process creation that give the programmer additional control over the parallel evaluation of goals. Finally, in section 2.7 we deal with the constructs needed for the allocation of objects and processes.

Since the primary intent here is to give an intuition for the mechanisms needed for parallel object oriented logic programming, and to motivate the constructs proposed by examples, the description of the constructs itself will be rather informal.

2.1 Objects and processes

We start with introducing the notion of objects. Throughout, a program is a Prolog-like program and a collection of object declarations.

In its most simple form an object declaration looks as follows.

object *name* {
clauses
}

As we continue, we will gradually introduce features giving more functionality to an object.

2.1.1 Objects as modules

The first view that we will take of objects is simply as modules, containing a collection of clauses. As an example of such an object, look at the declaration for a library of list manipulation predicates.

lib

```
object lib {

member(X,[ X | _ ]).
member(X,[ _ | T ]) :- member(X,T).

append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).

}
```

Clauses can be activated by a goal of the form

- *name*!*goal*

that must be read as asking for the evaluation of *goal* using the clauses defined for the object with that *name*.

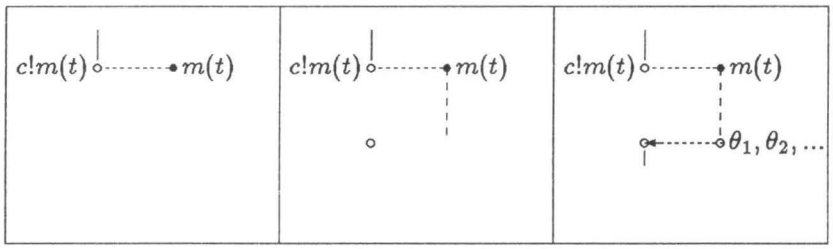
An example of the use of such a goal could be

```
:- lib!member(X,[amsterdam, paris, london]).
```

which, following ordinary Prolog evaluation rules, would bind the logical variable X successively to the cities mentioned in the second argument of *member*.

The intended semantics for an object as declared above does not deviate in any significant way from the semantics of an ordinary Prolog program. In other words, evaluating the goal $lib!member(X, L)$ will give the same results as evaluating the goal $member(X, L)$ when the clauses for *member* are directly added to the program.

Below we have pictured the way how communication with an object takes place.



Assume that we have an object c. The goal $c!m(t)$ asks the object c to evaluate $m(t)$, where m is a predicate name and t represents the arguments of the call. We use a predicate name m since, adopting standard terminology, we will speak of the methods of an object, which in our case are ordinary clauses. While the goal $m(t)$ is being evaluated, the caller waits for an answer. Backtracking over the results, indicated by $\theta_1, \theta_2, ...$, may take place as long as there are alternative solutions. Backtracking is initiated by the object that called for the method.

The obvious advantage of having the clauses for a predicate assembled in a module-like object is that, when a different functionality of these predicates is required, simply another object can be asked to do the evaluation.

We may extend the facilities for modular programming by allowing an object to use the clauses of another object. For example when defining a predicate $inboth(X, L_1, L_2)$, which checks whether X occurs both in list L_1 and L_2, it is convenient to be able to use the definition for *member* directly by using the clauses of *lib*, instead of explicitly addressing each call of *member* at the object *lib*. This is realized in the declaration for the object *check*.

check

```
object check {
use lib.
inboth(X,L1,L2) :-  member(X,L1), member(X,L2).
}
```

2.1.2 Objects with states

Modules of the kind treated above, how handy they may be, do not deserve to be mentioned objects, since they do not contain any private data or an internal state. Below we will introduce *non-logical variables*, for which we allow destructive assignment.[1] Also we will introduce a facility to make instances, or rather copies,

[1] Non-logical variables are usually called *instance variables* in object oriented terminology.

of declared objects. Furthermore, we will briefly discuss how objects may inherit non-logical variables and clauses of other objects.

Non-logical variables Objects may contain private data. We introduce *non-logical variables* for storing such data. As an example consider the declaration for the object *travel*.

travel

```
object travel {
use lib.
var city = [amsterdam, paris, london].
reachable(X) :- member(X, city).
}
```

We may ask such an object to evaluate the goal *reachable*(*tokyo*) as in

```
:- travel!reachable(tokyo).
```

for which the answer is, perhaps unfortunately, no. When the goal *reachable*(*tokyo*) is evaluated we assume that the non-logical variable *city* is replaced by its value, the list of cities to which it is initialized. Moreover, due to the backtracking provided by Prolog, we could ask the object *travel* to list all reachable cities.

The advantage of overloading predicate names comes into view when we imagine the situation that we have a number of travel agencies, implemented by the objects $travel_1, ..., travel_n$, similar to the object *travel* but with possibly a different value for the non-logical variable *city*, which allows us to ask

```
:- lib!member(O,[travel_1, ..., travel_n]), O!reachable(tokyo).
```

that may get us where we want to be, after all.

Both the features of non-logical variables, storing persistent data, and backtracking, enabling search over these data, are of relevance for the implementation of knowledge based systems. For such a toy example it may not seem worth to introduce non-logical variables, but in a real life situation these data may be stored in a large database. Only the clauses declared for an object have access to the non-logical variables. This justifies our speaking of clauses as methods, since the clauses provide an interface to the object encapsulating these data.

Assignment Having non-logical variables, the question immediately comes up whether we may change the value of such a variable or not. It seems unnatural to have no as an answer, since say a travel agency may decide to change its offer once in a while. We introduce a goal of the form

- *variable* := *term*

for assigning values to non-logical variables. The use of such a goal is illustrated in the following version of *travel*.

travel

```
object travel {
use lib.
var city = [amsterdam, paris, london].
reachable(X) :- member(X,city).
add(X) :- append([X],city,R), city := R.
}
```

So, as an example, when we have as a goal

```
:- travel!add(berlin).
```

each successive request to *travel* includes *berlin* as a reachable city. For convenience we have assumed that the list of destinations always grows longer. In general, assignment to a non-logical variable is destructive, in that the previous value is lost.[2]

Instances of objects Objects with mutable states introduce the need for having instances of objects of a particular kind. For example, we might like to have a number of instances of the object *travel*, that may differ in the destinations that they offer.

Each *instance* of an object contains both a copy of the non-logical variables of the object and a copy of its clauses. The non-logical variables of an instance are initialized to the current value of the non-logical variables of the object. Apart from the clauses declared for the object, also a copy is made of the clauses contained in the objects occurring in the *use* list.

For creating an instance of an object we introduce a goal

- $O = new(name)$

that results in binding the newly created instance of the object to the logical variable O. Its use is illustrated by a goal like

```
:-
  O1 = new(travel), O2 = new(travel),
  O1!add(berlin), O2!add(antwerpen).
```

in which two instances of the object *travel* are created, that differ in respectively including *berlin* and *antwerpen* in their offer of reachable destinations. Notice that instances of objects are objects as well.[3]

[2] We will discuss the protection needed in the presence of concurrency in section 2.3, where we treat the rendez-vous mechanism.

[3] We have deviated from standard terminology, in not speaking of objects as instances of classes, since both the named object declared in the program and its instances (that is copies) may be used as objects.

Inheritance As we have seen, an object may use the clauses of the objects it contains in its *use* list. We propose another feature to enable an object to inherit the non-logical variables of other objects. This type of inheritance is exemplified in the declaration

```
object travel {
var city = [amsterdam, paris, london].
}
object agency {
isa travel.
...
}
```

This declaration effects that the object *agency*, and all its instances, will have a non-logical variable *city*, initialized to the list above.

In most cases the inheritance relation is such that the inheriting object contains both the non-logical variables and the clauses of the objects it inherits. We have provided the notation

```
object a:b { ... }
```

as a shorthand for

```
object a {
isa b.
use b.
...
}
```

As an example, consider the declaration below

agency

```
object travel {
use lib.
var city = [amsterdam, paris, london].
reachable(X) :- member(X,city).
}
object agency : travel {
book(X) :- reachable(X), price(X,Y), write( pay(Y) ).
price(amsterdam,5).
...
}
```

The object *agency* may use all the clauses of *travel*, and in addition has access to the non-logical variable *city*.

Inheritance is effected by code-sharing, in a static way. Conceptually, the inheriting object contains a copy of the objects it inherits. We will discuss how we deal with clashes that may arise in multiple inheritance in chapter 4, where we will also provide some examples of how inheritance may be used for knowledge representation.

2.1.3 Active objects

Thus far, we have not given any clue how we conceive of concurrent programming in our (yet to be proposed) language. The first idea that comes to mind is to make passive (instances of) objects active, by letting them have activity of their own. Having a number of objects concurrently executing some activity of their own is, however, not of much help when there is no means to communicate with these objects. Therefore, apart from allowing to create active instances of objects, it is also necessary to provide a way by which their activity can be interrupted in order to evaluate some goal.

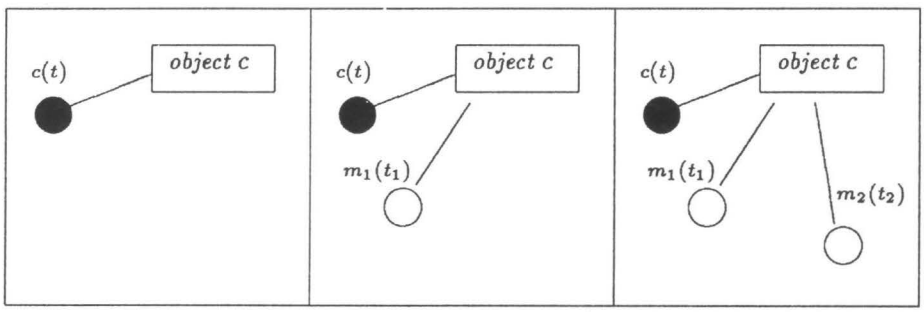
Active objects are created by a goal of the form

- $O = new(name(t_1, ..., t_n))$

where *name* is the name of the declared object, and $t_1, ..., t_n$ are arbitrary terms.

The term $name(t_1, ..., t_n)$ is called the *constructor*, since when creating a new object a process is started evaluating the goal $name(t_1, ..., t_n)$. In order not to fail, clauses must be defined by which the constructor can be evaluated. The predicate name of the head of these clauses, that for obvious reasons we call *constructor clauses*, is identical to the name of the declared object.

Multiple processes may refer to a single object. Apart from the constructor process, a process is created for each method call in order to keep track of the backtracking over the results of that call. We have pictured an object and some processes referring to it below.



The call $O = new(c(t))$ results in a new instance of the object c together with a constructor process evaluating $c(t)$. Both the calls $O!m_1(t_1)$ and $O!m_2(t_2)$, that may come from different processes, result in a separate process for evaluating these calls.

The constructor clauses specify, what may be called the *body* of an object, its own activity. For interrupting this own activity we provide the goal

- $accept(any)$

that forces the object to wait until it is requested to evaluate a goal.[4] When this has happened, that is when the goal is evaluated and an answer has been sent back, the accept goal succeeds and the object may continue with its own activity.

[4]Later on we will encounter accept goals of a more complex nature.

As an example, consider the object declaration for an *agency* that, in some naive way, implements the fusion of a number of travel agencies of the old style.

agency

```
object agency {
use lib.
var cities = [].
agency(L) :-
        member(O,L),
        O!reachable(X),
        add(X),
        fail.
agency(_) :- run().
run() :- accept(any), run().
reachable(X) :- member(X,cities).
add(X) :- append([X],cities,R), cities := R.
}
```

The declaration for *agency* differs from the declaration for the object *travel* only in having constructor clauses and an auxiliary clause for *run*(), that define the own activity of each instance of an *agency*.

Suppose now that we wish to combine four travel agencies, $travel_1, ..., travel_4$ of the old style into two new agencies, then we may state as a goal

```
:-
  O1 = new(agency([travel1, travel2])),
  O2 = new(agency([travel3, travel4])),...
```

the result of which is that both agencies, start initializing their list of cities concurrently. The body of an agency consists, after initialization, of a tail-recursive loop stating the willingness to accept any goal. Each time the accept goal is reached, the object waits until it is requested to evaluate a goal. A request to evaluate a goal, in its turn, must wait until the object is willing to accept such a request.

2.1.4 Communication and synchronization

We have sketched here the most simple form of the evaluation of a goal by an object. We call this *remote goal evaluation* since we have not provided the means yet to be selective in what is acceptable as a request.

Clearly, apart from the initialization and the fact that the own activity of an object must explicitly be interrupted, the semantics of an active object must be similar to that of a passive object. Conceptually, we may regard a passive object *obj* to be executing its constructor *obj*(), defined by

```
obj() :- accept(any), obj().
```

For active objects, however, only one method call may be active with producing its first answer. We do not wish to impose any restrictions on the internal concurrency displayed by passive objects.[5] The programmer, however, must take care to provide the protection necessary for safely accessing non-logical variables. Active objects do allow a limited form of internal concurrency, namely for backtracking over multiple answers to a method call.

Backtracking A question we have not addressed when treating the remote evaluation of a goal by an active object is how we deal with the possible occurrence of backtracking over the resulting answers, as imposed by our intention to have a semantics that does not deviate from the one for ordinary Prolog, with respect to backtracking.

In our proposal we deal with the backtracking that may occur in a method call by creating a new process for each request to evaluate a goal. The backtracking information needed for finding all solutions for the goal is maintained by that process.

As a related question we may ask: when do we allow an object to continue with its own activity? In the absence of backtracking the natural choice is: immediately after the answer has been delivered. In our case we might wish to deliver all answers before allowing an object to continue its own activity. We feel however that this is overly restrictive and we will provide arguments why it is sensible to stick to the natural choice for non-backtracking languages, despite backtracking.

Another thorny issue, in the presence of backtracking, is the existence of non-logical variables that may be assigned values in an imperative way. Must these assignments be undone on backtracking or not? The examples given already suggest the latter alternative. Indeed this is the alternative chosen, and defended later on.

Synchronization We may consider remote goal evaluation as an important means for objects to communicate with each other. Moreover, by requiring to state explicitly whether an object is willing to accept a request, we have provided some means for synchronizing the behavior of objects.

However, we may wish to be more selective in what to accept as a request. For instance what is acceptable may depend on the state of the object, or even on conditions imposed on the arguments of the call. When the object is selective in this sense, it seems more apt to speak of a rendez-vous, since both the object and the process that request the evaluation of a goal participate establishing in the communication.

Summarizing, what we have described until now is more or less a full-fledged object oriented language. We may regard the clauses defined for an object as methods, having access to private data stored in the non-logical variables. Calling a method is to engage in a rendez-vous, when the object is willing to accept the call.

[5] When multiple processes referring to a single object are active concurrently we speak of internal concurrency.

Before continuing our description of this approach, however, we wish to reflect on the possibility to realize objects with states that communicate by means of message passing, in a simpler way. Do we need non-logical variables to implement states? And, do we need a synchronous rendez-vous to communicate with objects? We will deal with these questions in the next section, where we explore the possibility of using channels as the medium of communication between active objects.

2.2 Communication over channels

We may implement objects as continuously running processes communicating with each other over channels. C.f. [Shapiro and Takeuchi, 1983], [Pereira and Nasr, 1984]. Before going into details we will present the language constructs involved. First of all we need a facility to create processes. We will use a goal of the form

- $new(c(t_1, ..., t_n))$

to create an active instance of the object c.
For creating new channels we use a goal of the form

- $C = new(channel)$

that results in binding the newly created channel to the logical variable C.
Further we need, what we call an output statement of the form

- $C!t$

where C refers to a channel and t is an arbitrary term.
Also, we need an input statement of the form

- $C?t$

where C is assumed to refer to a channel and t is an arbitrary term.

We will characterize the semantics of communication over channels by giving a simple example, adapted from [Pereira and Nasr, 1984], but originally given in [Shapiro and Takeuchi, 1983]. Assume that we wish to implement a counter, that allows us to ask for its value and to increment its value. Clearly, we must have some means to store the state of the object, and also some means to send it the corresponding messages. With the constructs introduced, our implementation looks as follows.

ctr

```
object ctr {
ctr(C) :- run(C,0).
run(C,N) :-
        C?inc(),
        N1 = N + 1,
        run(C,N1).
run(C,N) :-
        C?value(N),
        run(C,N).
}
```

The first clause encountered is the constructor for an instance of *ctr*. The argument C is assumed to refer to a channel. Evaluating the constructor results in calling $run(C, 0)$, initializing the (logical) state variable holding the value of the counter to zero. The remaining two clauses define the body of the object. The first clause contains the input statement $C?inc()$ that is used to increment the value of the counter. The second clause contains the input statement $C?value(N)$ that is used to answer requests for the value of the counter. The actual value of the counter is maintained in a proper way by passing it as an argument to the tail-recursive call to *run*.

A typical example of the use of such a counter is the goal

```
:-
 C = new(channel),
 new(ctr(C)),
 C!inc(),
 C!value(X).
```

that effects the binding of X to one.

The example given illustrates the use of such objects to implement server processes. Let us now give a more detailed description of the semantics of communication over channels.

Communication over channels is synchronous, in that either side waits until there is a complementary communication intention for that channel. For the example above this means that the body of the counter will stick to the goal $C?inc()$ until the user process reaches an output statement. We call a communication successful if the term at the input side, or more briefly the input term, is unifiable with the output term, the term at the output side. When these terms do not unify, as is the case for $inc()$ and $value(X)$, the input side is allowed to backtrack until it finds another input statement for that channel and the procedure is repeated.. As long as the input side is backtracking the output side waits with its request to communicate.

The asymmetry with respect to backtracking is sufficiently motivated by the example above. We must remark however that Delta Prolog adopted a communication mechanism that is symmetric in its backtracking behavior, but rather complex.

We stress the fact that both in Delta Prolog and in our proposal communication over channels is bi-directional, in the sense that variables both in the input term and the output term may receive a value through unification.

As an example consider the following object declaration

```
object a {
a(C) :- run(C,0).
run(C,N) :- C?f(N,Y), run(C,Y).
}
```

and the goal

```
:- C = new(channel), new(a(C)), C!f(X,1).
```

which results in binding X to zero and Y in the body of a to one.

We conclude this intermezzo with an example of a situation where the number of processes can grow indefinitely large. Below we present our implementation of the solution to the problem of generating primes given in [Hoare, 1978].

primes

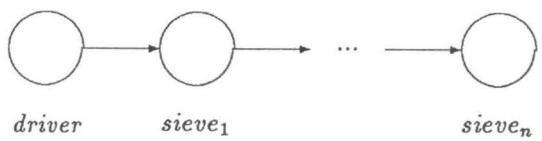
```
object driver {
driver(I) :-
        C = new(channel),
        new(sieve(C)),
        drive(C,I).
drive(C,I) :-
        C!I, J = I + 2, drive(C,J).
}
object sieve {
sieve(C) :-
        C?P,
        collect(P),
        Cout = new(channel),
        new(sieve(Cout)),
        run(P,C,Cout).
run(P,C,Cout) :-
        C?I,
        ( I//P ≠ 0 → Cout!I ; true ),
        run(P,C,Cout).
collect(I) :- write(I).
}
```

The program is started by the goal

```
:- new(driver(3)).
```

The output can be collected by sending each prime with which a new sieve is initialized to a special process. The goal $I//P \neq 0$ is evaluated by simplifying $I//P$ to I *modulo* P followed by a test whether the result is unequal to zero.

The structure of these processes may be pictured as



where each arrow represents a channel.

As sketched here, communication over channels offers a rather limited functionality. In particular since we have not included guarded commands or annotated variables, synchronization must rely purely on the synchronous nature of communication. Another important limitation is that no backtracking over the results of a communication is allowed, once a successful communication is achieved.

2.3 Communication by rendez-vous

In the previous section we have seen how we may implement objects with states without the use of non-logical variables to store the state of such objects. The approach sketched there has a number of limitations. Instead of augmenting the proposal of the previous section, we will take up the main thread of this chapter and will investigate how we may achieve object oriented behavior by regarding clauses as methods.

Below we summarize the language features that we treat in this section.

- $x := t$ - to assign the value of a term to the non-logical variable x;
- $O = new(c(t_1, ..., t_n))$ - to create an active instance of c, to which O will refer;
- $O!m(t_1, ..., t_n)$ - to call the method m of the object to which O refers;
- $accept(m_1, ..., m_n)$ - to state the willingness to accept calls for $m_1, ..., m_n$.

States As we have indicated in the section introducing objects we may use non-logical variables to store persistent data. We rephrase the declaration of the counter presented in the previous section to recall their use.

ctr

```
object ctr {
var n = 0.
ctr() :- accept(any), ctr().
inc() :- n := n + 1.
value(N) :- N = n.
}
```

This solution differs from the previous one in that the state of the object is not maintained by keeping the value of the counter as an argument in a tail-recursive loop, but as an explicit non-logical variable that can be updated by assignment.[6] A typical use of such a counter is exemplified by the goal

```
:-
    C = new(ctr()),
    C!inc(),
    C!value(X).
```

[6] We allow for arithmetic simplifications both in the right hand side of assignments to non-logical variables and in the left and right hand side of an equality.

where a counter is created, which subsequently receives the method calls *inc*() and *value*(X). Despite the notational similarity with communication over channels, the calls $C!inc()$ and $C!value(X)$ are now method calls, that is goals that are evaluated by the object. The evaluation of these goals is taken care of by the clauses defined for *inc* and *value*.

An instance of a counter is an active object. Method calls for an active object must be explicitly accepted by an accept statement. To protect the access to non-logical variables, mutual exclusion between method calls is provided, for active objects, by not allowing any method call to be accepted as long as the evaluation of a method call has not lead to returning an answer. After having delivered the first answer, other method calls may become active. As we will see, for passive objects we do not wish to provide such protection.

Suspension The mutual exclusion provided by a counter is only meant to protect the access to non-logical variables. At any time, any method call is acceptable. It is conceivable, however, that whether a method call is acceptable depends on the state of the object, as expressed in its non-logical variables.

A typical example of such an object is the semaphore, given below.

sema

```
object sema {
var n = 0.

sema(N) :- n := N, run.

p() :- n := n - 1.
v() :- n := n + 1.

run :-
      ( n ≡ 0 → accept(v) ; accept(p,v) ),
      run.
}
```

The constructor for *sema* causes the semaphore to loop over a conditional that tests the value of the non-logical variable n. When the value of n is zero, calls to $p()$ will be suspended, due to the statement *accept*(v); otherwise both $p()$ and $v()$ may occur, since when n is not zero the statement *accept*(p, v) is evaluated.

A semaphore of the kind above may be used to regulate the concurrent evaluation of goals by passive objects. To illustrate this we present a modified version of the travel agency described in section 1.2.

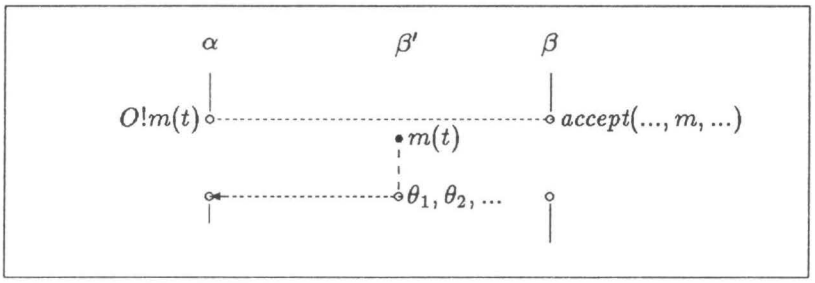
travel

```
object travel {
use lib.
var s = new(sema(1)).
var city = [amsterdam, paris, london].
reachable(X) :- member(X,city).
add(X) :- s!p(), append([X],city,R), city := R, s!v().
}
```

The object *travel* implements a multiple readers/single writers protocol, since adding an item to the city list is embedded in the semaphore calls $p()$ and $v()$. The initialization of the non-logical variable s to an active instance of *sema* occurs exactly once for each instance of *travel*.

Lets now take a closer look at the semantics of the accept statement. By the way, we only allow accept statements to occur in active objects since we wish method calls for passive objects to be evaluated concurrently.[7]

Calling a method of an active object results in a rendez-vous, when that object is willing to accept the call. The interaction between the processes involved is pictured below.



Say that we have a process α that calls $O!m(t)$, with O bound to the object of which β evaluates the constructor. When β reaches an accept statement the activity of β is interrupted, and a new process β' is started which refers to the same object as β, to evaluate $m(t)$. Process β resumes the evaluation of the constructor as soon as the first answer, say θ_1, is produced. Thereafter β' continues to produce alternative solutions, as may be requested by α.

Operationally, when an accept statement is reached, the evaluation of the current goal is suspended until a method allowed by the arguments of the accept statement is called for. We call the argument of the accept statement the *accept list*, and by convention take *any* to stand for all methods of the object. When a method not occurring in the accept list is called for, the call is suspended and the object waits until another call satisfying the accept list occurs. The suspended call will result in a process evaluating the method call whenever the accept list is changed in such a way that the call is allowed. For handling suspended calls the

[7] For active objects we do allow accept statements to occur (possibly nested) also in processes for evaluating a method call. In our semantic description, however, we impose the requirement that accept statements may occur only in the constructor process of an object.

object maintains a so-called *accept queue*. All suspended calls are stored in the accept queue, in the order in which they arrive. When the accept list is changed, the object first searches the queue and takes the first call satisfying the new accept list. If no such call is present, the object waits for other incoming calls. This procedure guarantees fairness in handling method calls, since due to the FIFO behavior of the accept queue, no allowed call will be suspended forever. C.f. [America, 1989b].

Dining philosophers Our pièce de rédistance is an implementation of a solution for the problem of the *dining philosophers*. Cf. [Dijkstra, 1971].

Five philosophers must spent their time *thinking* and *eating*. When a philosopher wants to eat he must get hold of two forks. Since only five forks are available, each philosopher must await his turn. To avoid the situation where each philosopher sits waiting with one fork, only four philosophers at a time are allowed in the dining room.

Since a philosopher needs to know no more than his name, the dining room and his proper forks, after creation he may proceed to enter the cycle of thinking, entering the dining room, picking up his forks, eating and leaving the dining room to start thinking, again.

philosopher

```
object philosopher {
var name.

philosopher(Name,Room,Lf,Rf) :-
        name := Name,
        proceed(Room,Lf,Rf).

think :-  write(thinking(name)).
eat :-  write(eating(name)).

proceed(Room,Lf,Rf) :-
        think,
        Room!enter(),
        Lf!pickup(), Rf!pickup(),
        eat,
        Lf!putdown(), Rf!putdown(),
        Room!exit(),
        proceed(Room,Lf,Rf).
}
```

A philosopher is admitted to the dining room when less than four guests are present, otherwise he must wait for one of his colleagues to exit.

room

```
object room {
var occupancy = 0.

room() :-
        (
        occupancy ≡ 0 → accept(enter) ;
        occupancy ≡ 4 → accept(exit) ;
        accept(enter, exit)
        ),
        room().

enter() :- occupancy := occupancy + 1.
exit() :- occupancy := occupancy − 1.

}
```

Forks can only be picked up and then put down.

fork

```
object fork {

pickup().
putdown().

fork() :-
        accept( pickup ),
        accept( putdown ),
        fork().

}
```

The ceremony is started by assigning the philosophers their proper forks and showing them the dining room. We omit the details of their initiation.

The example fully demonstrates the synchronization enforced by accept statements. Such behavior could not be effected by using synchronous communication over channels. In fact, the synchronization and suspension capability of the rendezvous mechanism makes communication over channels superfluous. However, communication involving accept statements and non-logical variables is semantically considerably more complex, and seems to preclude a declarative semantics. Apart from this, communication over channels is more efficient.

2.4 Distributed backtracking

Distributed backtracking is an important issue for systems that wish to support *don't know* nondeterminism, in contrast to *don't care* nondeterminism where once a choice is made all alternatives are thrown away. The examples presented previously were deterministic in the sense that only one solution needed to be produced. The object presented in the following example, however, may produce an infinite number of solutions.

nat

```
object nat {
number(0).
number(s(X)) :- number(X).
}
```

Its use is illustrated by

```
:- nat!number(X), write(X), fail.
```

which will print all natural numbers, eventually. Backtracking is done lazily, in that on backtracking the object evaluating *number* will start to produce the next solution.
Note that the goal

```
:- nat!number(X), X = s(s(0)).
```

differs from the goal

```
:- nat!number(s(s(0))).
```

in that the latter only communicates success, whereas the former has to communicate three bindings for X.

Backtracking in the presence of non-logical variables In the presence of non-logical variables mutual exclusion is needed for reasons of protection. Mutual exclusion takes effect for active objects only. This protection however lasts until the first answer is requested and received. This procedure is motivated by the assumption that any important change of the state of the object can be done during the period that the first solution is produced. Backtracking over the second and remaining solutions can be done while other processes are active.

We will show how the state of an object can be fixed by binding it to a non-logical variable. Consider another variant of our, by now familiar, travel agency.

travel

```
object travel {
use lib.
var city = [amsterdam, paris, london].
travel() :- accept(any), travel().
reachable(X) :- L = city, member(X,L).
add(X) :- append([X],city,R), city := R.
}
```

In the clause for *reachable* we have made the binding of the second argument of *member* to the value of the city list explicit. Since an instance of *travel* now is an active object the mutual exclusion mechanism prevents the city list to be changed while the first answer for *reachable* is being produced. After that, *member* may

backtrack, but with the value of the city list bound to L, as it was at the time of the call.

Now suppose that we declare *travel* in the following somewhat contrived manner.

travel

```
object travel {
use lib.
var city = [amsterdam, paris, london].

travel() :- accept(any), travel().

reachable(X) :- length(city,N), get(N,X).

add(X) :- append([X],city,R), city := R.

get(N,X) :- element(N,city,X).
get(N,X) :- N > 1, N1 = N - 1, get(N1,X).

length([],0).
length([H|T],N) :- length(T,N1), N = N1 + 1.

element(1,[ X | _ ],X).
element(N,[ _ | T ],X) :-
        N > 1,
        N1 = N - 1,
        element(N1,T,X).
}
```

The reader may check that now the city list may be updated, by adding elements to the front, while the process evaluating *reachable* is backtracking over the possible solutions. In general such interference is not desirable, but as has been shown, can easily be avoided by binding the state of an object to a logical variable.

Deterministic objects In many cases we do not need the full power of backtracking over the results in a rendez-vous. For instance, asking the value of a counter results in precisely one answer. Such deterministic behavior is obvious when the call contains no unbound logical variables, since the answer will then be either failure or success.

To cope with the cases in which this cannot so easily be decided we have provided a way to declare an object to be deterministic. Our counter clearly is a deterministic object.

ctr

```
deterministic object ctr {
var n = 0.

ctr() :- accept(any), ctr().

inc() :- n := n + 1.
value(N) :- N = n.

}
```

The prefix *deterministic* enforces that only one solution will be returned for each method call.

2.5 Conditional acceptance

The accept statement that we have considered only allows to mention the names of methods for which a call is acceptable. We will now introduce a much more powerful mechanism that allows, among other things, to impose arbitrary conditions on the arguments of the call.

The format of the *conditional accept statement* is

- $accept(..., m(t_1, ..., t_n) : guard \rightarrow goal, ...)$

The semantics of the conditional accept statement is similar to the semantics of the accept statement treated previously, except that instead of a method name m, also expressions of the form

$$m(t_1, ..., t_n) : guard \rightarrow goal$$

may be stored in the accept list. When a method is called, say by $m(t'_1, ..., t'_n)$ then if the call can be unified with the expression $m(t_1, ..., t_n)$ and if in addition the *guard* can be successfully evaluated, the goal *goal* will be evaluated. Both the *guard* and the *goal* may contain variables occurring in $m(t_1, ..., t_n)$. The evaluation of *goal* replaces the actual (local) method call. It is easy to see that the conditional accept statement subsumes the original accept statement since the statement $accept(m)$ has a meaning identical to

$$accept(m(t_1, ..., t_n) : true \rightarrow m(t_1, ..., t_n))$$

We call the arguments of the accept statement *accept expressions*. Accept expressions may take the following forms.

- m - accepts all calls for method m
- $m(t_1, ..., t_n)$ - accepts all calls that unify with $m(t_1, ..., t_n)$
- $m : guard$ - accepts all calls for m if the *guard* holds
- $m(t_1, ..., t_n) : guard$ - accepts all calls that unify with $m(t_1, ..., t_n)$ for which the *guard* holds
- $m : guard \rightarrow goal$ - executes *goal* for all calls to m if the *guard* holds

In addition we have

- $m(t_1, ..., t_n) : guard \rightarrow goal$

that executes *goal* for all calls unifying with $m(t_1, ..., t_n)$ for which the *guard* hold.

To illustrate the power of the generalized accept statement we rephrase some of the examples presented earlier.

We will first give an alternative declaration for the object *sema*.

sema

```
object sema {
var n = 0.
sema(N) :- n := N, run.
p() :- n := n - 1.
v() :- n := n + 1.
run :- N = n, accept(v:N ≥ 0, p:N > 0), run.
}
```

The behavior of an instance of *sema* is identical to the behavior of the object *sema* as defined before. The present declaration differs from the previous one in that the Prolog conditional goal

$$n \equiv 0 \rightarrow answer(v); answer(p, v)$$

is replaced by the conditional accept statement

$$accept(v : N \geq 0, p : N > 0)$$

with N bound to n, in which the guards contain the conditions under which the method calls may be accepted.

Perhaps somewhat surprisingly, we no longer need to use non-logical variables to maintain the state of an (active) object. We will first illustrate this by (re) declaring our familiar counter.

ctr

```
object ctr {
ctr() :- run(0).
run(N) :- accept(
          inc():true → N1 = N + 1,
          value(N):true → N1 = N
          ),run(N1).
}
```

The state is passed as an argument in a tail-recursive call to *run*, that implements the body of the object.

In a similar way we can implement a semaphore, as shown below.

sema

```
object sema {
sema(N) :- accept(
           v:N ≥ 0 → N1 = N + 1,
           p:N > 0 → N1 = N - 1
           ), sema(N1).
}
```

which, to our mind, is a rather elegant way of coding a semaphore. Notice that we do not have to specify clauses for the methods but may specify the functionality of a method in the *goal* part of an accept expression.

It is a kind of pity that enlarging the functionality of the accept statement renders such a nice feature as the non-logical variable, to record the state of an object, obsolete. However, with respect to expressiveness, there is a price to pay. As we will see in chapter 4, logical state variables maintained as an argument in a tail-recursive loop may not be inherited, whereas non-logical state variables may be inherited among objects, thus allowing a rather concise description of the functionality of a collection of objects.

Another reason to stick to non-logical variables has to do with efficiency. Transferring a complex state as an argument after each method call is clearly less efficient.

No need to say that the generalized accept statement preserves backtracking over the possible answers delivered in a rendez-vous, as illustrated by our rephrasing of the declaration of a travel agency.

travel

```
object travel {
use lib.
travel() :- run([amsterdam,paris,london]).
run(L) :- accept(
          reachable(X): L1 = L → member(X,L),
          add(X): true → append([X],L,L1)
          ), run(L1).
}
```

Just as before we may generate all reachable cities by stating the goal

```
:- O = new(travel()),(O!reachable(tokio); O!reachable(X)).
```

Since the *goal* in a conditional accept expression may fail, care must be taken to update the state variable in a proper way, as has been done in the example above.

2.6 Process creation and resumptions

The constructs presented thus far are all rather transparent in the sense that we have taken care to preserve the semantics of ordinary Prolog programs to the extent possible. We will now present some extensions by which the programmer may be able to profit more from the potential parallelism in a problem.[8] For process creation, one may use the goal statement

[8]The constructs treated in this section and the following are fairly low level. We encourage the reader interested in only the major concepts of DLP to skip these sections and to continue with the next chapter.

- $Q = O!G$

to request the object to which O refers to create a process to evaluate the goal G. The variable Q thereafter refers to that process. And we introduce the goal

- $Q?$

to request the answers delivered by the process to which Q refers.

The semantics of these statements can be easily explained by remarking that the method call $O!G$ may considered to be implemented as

```
O!G :- Q = O!G, Q?.
```

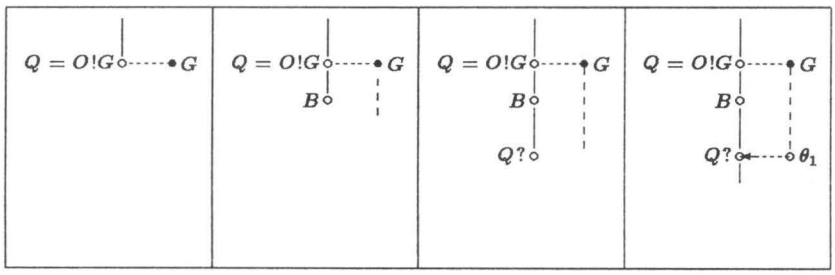
Using a somewhat contradictory terminology, a call of the form $Q = O!G$ may be regarded as an asynchronous method call, since receiving the answers requires an explicit request of the form $Q?$. The variable Q is bound to a pointer to the process evaluating G. The goal G may be any goal, for instance one that must be explicitly accepted by an accept goal of the object to which O refers. We call the goal $Q?$ a resumption request, since it delivers a resumption containing the answer substitutions that result from the call. Evaluating the resumption effects the possible variable bindings of these answers in the current context. See also section 9.2.

Asynchronous calls allow the programmer to achieve some extra parallelism, since the newly created process runs independently of the invoking process, that does not have to wait for an answer. The overlapping of these processes is reflected in the goal sequence

$$Q = O!G,\ B,\ Q?$$

In between the creation of the process and stating the resumption request to collect the answers, the invoking process can perform whatever action suitable.

Below we have depicted the steps that are taken in evaluating a goal as the one above.



When the call $Q = O!G$ is accepted, a process for evaluating G is created. The caller may proceed with evaluating B, whereafter it must wait for an answer for G. The resumption goal $Q?$ succeeds when the answer is received.

This mechanism of process creation and resumption allows to define and-parallelism in a rather straightforward way, as

```
A&B :- Q = self!A, B, Q?.
```

where *self* refers to the object evaluating the goal $A\&B$. Such goals may however occur only in passive objects, since only passive objects allow internal concurrency.

Note that, somewhat counter-intuitively, backtracking will be done first over A and then over B, which, semantically, gives rise to the inequality $A\&B \neq A, B$. A typical example of the usage of this kind of parallelism is the following, familiar, quicksort program.

qsort

```
qsort([],[]).
qsort([X|T],S) :-
  split(X,T,L1,L2),
  ( qsort(L1,S1) & qsort(L2,S2) ),
  append(S1,[X|S2],S).

split(X,[],[],[]).
split(X,[Y|T],[Y|L1],L2) :-
        X > Y,!,
        split(X,T,L1,L2).
split(X,[Y|T],L1,[Y|L2]) :-
        split(X,T,L1,L2).
```

Each nonempty list is divided into two sublists, one with values less than the values in the other, that are sorted in parallel and then appended.

The advantage of this approach is that the programmer may restrict the cases where parallel evaluation occurs by imposing extra conditions (cf. [DeGroot, 1984]) as in

```
A&B :-  ground(A),!, Q =  self!A, B, Q?.
A&B :-  A, B.
```

where splitting of a new process is allowed only when A is ground. Note that the cut in the first clause is used to avoid unwanted backtracking over the second solution of $A\&B$.

2.7 Allocation of objects and processes

In the examples given, no attention has been paid to the issue of allocating (instances of) objects and the actual distribution of computation over the available resources.

When a new instance of an object is created, it can be allocated on a particular processor node, by a statement of the form

- $O = new(obj@N)$

where N is a so-called *node expression* denoting a particular processor node and *obj* either the name of an object or a constructor, that is an object name with arguments.

Also it is possible to split off a process to evaluate a goal on a particular node by the statement $G@N$ for G a goal, and N a node number. Conceptually, the meaning of a goal $G@N$ is given by the clause

```
G@N :- O = new(self@N), Q = O!G, Q?.
```

Such a goal may be used only for passive objects.

Node expressions refer to a processor of the parallel machine on which the system runs. The parallel machine for which our system is intended, is assumed to have a limited number of processor nodes that are connected with each other by a communication network. See section 1.1. The programmer knows each processor by its number, $0...n-1$, for n processors. To permit a more refined strategy of allocating processes and resources, *node expressions* may be used, from which a processor number can be calculated. Apart from viewing the network as a linear sequence of processors, the programmer may look at the configuration as an (imaginary) tree, or as being organized as a matrix, a grid of processors.

Tree organization A node expression of the form

$$I_0 : I_1 : ... : I_n$$

denotes, with the branching degree (by default) fixed to two, the processor associated with the node in the tree reached by following the path $I_1, ..., I_n$ from I_0. The association of processor numbers with nodes of the tree is done by counting the nodes of tree breadth first. For example the expression $0 : 1 : 2 : 1$, giving the path $1, 2, 1$ from 0, results in processor number 9.

As an example of how such node expressions can be used to distribute the load of computation over the available processors, consider the following variant of the quicksort program presented earlier.

qsort

```
qsort(L,R) :-  qsort(0,L,R).

qsort(N,[],[]).
qsort(N,[X|T],S) :-
  split(X,T,L1,L2),
  ( sort(N:1,L1,S1)@N:1 & sort(N:2,L2,S2)@N:2 ),
  append(S1,[X|S2],S).
```

When splitting off two processes for sorting the sublists, the processes are allocated on the successor nodes of the current node, as long as sufficient processors are available. More refined strategies may be encoded by including tests on the length of the lists.

Matrix organization The processor topology may also be viewed as a matrix with a certain width (say 4 for 16 processors). The programmer can index this matrix by expressions of the form

$$N_1 \# N_2$$

which allows to distribute the load over the available processors by moving, for instance north from $N_1 \# N_2$ to $N_1 \# N_2 + 1$, over the matrix. Abbreviations such

as *north*, *west* and so on are provided to facilitate these kind of turtle movements over the matrix of processors. Cf. [Shapiro, 1984].

Chapter 3

The language DLP

- Perhaps a difference is to be sought in the opposite direction: perhaps expression is more direct and immediate than representation. -
Nelson Goodman, Languages of Art

The constructs that we have explored in previous chapter are part of the language DLP, a language for distributed logic programming, that has been the result of the research reported in this thesis. In this chapter we wish to provide an overview of the language DLP, and we will also isolate the subsets of DLP chosen for studying the semantics of the language. We will start with a summary of the constructs introduced in the previous chapter.

3.1 Declarations

An object declaration is of the form

object *name* {
use *objects.*
var *variables.*

clauses
}

Among the clauses may be *constructor* clauses, that enable to create active instances of an object. The clauses of an object act as methods, in that they have exclusive access to the non-logical variables of an object. Clauses are ordinary Prolog clauses, except for the possible occurrence of *special forms* by which DLP extends Prolog.

Inheritance of the non-logical variables of objects is achieved by including

isa *objects*

in the object declaration. The declared object then contains a copy of all the non-logical variables of the objects mentioned in the *isa* list.

A declaration of the form

object a:b { ... }

must be read as

object a {
isa b.
use b.
...
}

Inheritance is static. It is effected before an object becomes active evaluating a goal. In other words, there is no interaction between objects, except on the occasion of an explicit communication.[1]

3.2 Statements

The core of DLP consists of the special forms

- $v := t$ - to assign (the value of) the term t to the non-logical variable v
- $O = new(c(t))$ - to create an active instance of the object c
- $O!m(t)$ - to call the method $m(t)$ for the object O
- $accept(m_1, ..., m_n)$ - to state the willingness to accept methods $m_1, ..., m_n$

These are the extensions that play a prominent role in part II, treating the semantics of DLP. Moreover, the constructs listed above are sufficient to solve a number of knowledge representation problems, as illustrated in chapter 4.

In addition, however, we have special forms that allow to communicate over channels.

- $C = new(channel)$ - to create a new channel
- $C!t$ - to put output term t on channel C
- $C?t$ - to put t as an input term on channel C

Apart from the regular accept statement we have introduced a generalized accept statement, that may contain *conditional accept expressions* of the form

- $method : guard \rightarrow goal$

[1] See chapter 4 for a more precise characterization of inheritance, and the resolution of possible name conflicts.

that result in the *goal* being evaluated, if *method* agrees in some way with the call and moreover the *guard* holds. (C.f. section 2.5)

For those wishing to squeeze out the last drop of parallelism we have included the primitives

- $Q = O!G$ - to create a process to evaluate G
- $Q?$ - to ask for the results of process Q

These are the primitives that have been used to implement the synchronous rendezvous.

Lastly, since effective parallelism requires some strategy of allocating resources we have provided *node-expressions* that can be used to allocate an object or a process, evaluating a goal. The special forms

- $O = new(obj@N)$
- $G@N$

where *obj* is either an object name or a constructor, respectively allocate an object or a goal at the processor node denoted by N.

Since we have strived for full compatibility with Prolog, we have retained the backtracking behavior to the extent possible. We support both *local* backtracking, that occurs within the confines of a process, and *global* (or *distributed*) backtracking in which multiple processes are involved.

We have made a distinction between *active* objects, which have own activity that must explicitly be interrupted to answer method calls; and *passive* objects that may answer a method any time. Active objects are allowed to have a limited amount of internal concurrency, in that multiple processes may be active backtracking over the results of a method call. Passive objects, on the other hand, display full internal concurrency. Method calls may be evaluated concurrently. Providing the needed protection is in this case the duty of the programmer.

3.3 DLP as a family of languages

For studying the semantic aspects of the language, it has appeared profitable to isolate a number of subsets of DLP. Due to its static nature we may ignore inheritance. For all the sub-languages DLP_0 - DLP_2 (see below) we have provided both an operational and a denotational (continuation) semantics in chapter 8. We have not included a semantic characterization of conditional acceptance since, despite the elegance of the construct, this seems rather complex. Features such as allocation are omitted since they are simply not interesting from a semantic point of view. Neither have we included a treatment of the cut, nor a treatment of assert and retract statements.

Backtracking and non-logical variables. We consider DLP_0 to be the base language of DLP. It extends Prolog (without cut) with a construct to assign values to non-logical variables.

DLP_0 = *Prolog* + *non-logical variables*		
non-logical variables	$v := t$	assigns the term t to the non-logical variable v

Moreover, in unification goals of the form $t_1 = t_2$, and in the right hand side of assignments, non-logical variables are replaced by their values and arithmetical expressions are simplified.

Communication over channels. The sub-language DLP_1 extends DLP_0 by a construct to create active instances of objects. Also, communication over channels is introduced.

DLP_1 = DLP_0 + *communication over channels*		
object creation	$new(c(t))$	creates an object executing $c(t)$
channels	$C = new(channel)$	creates a new channel
	$C!t$	output statement for term t over channel C
	$C?t$	input statement for term t over channel C

Communication over channels is relatively simple, since the backtracking that occurs during a communication is a local phenomenon, restricted to the process at the input side of the channel.

Communication by rendez-vous. Our next extension is called DLP_2 . It extends DLP_0 by constructs for synchronizing on method calls.

DLP_2 = DLP_0 + *communication by rendez-vous*		
object creation	$O = new(c(t))$	creates an active instance of object c,
rendez-vous	$O!m(t)$	calls the method $m(t)$ for object O
	$accept(m_1, ..., m_n)$	states the willingness to accept the methods $m_1, ..., m_n$

As a restriction for DLP_2 , which does not hold for DLP, we require that accept statements occur only in constructor processes.

We have given an operational semantics of DLP_2 in chapter 7. The backtracking that may occur during a rendez-vous is global. We speak of global or distributed backtracking, since the process calling the method may force the answering process to backtrack over any remaining answers.

The language DLP is simply the collection of all constructs introduced thus far. See table 3.1. It supports cuts, and the assert and retract statements as

may be encountered in Prolog. Also inheritance is included. Clearly, some of the constructs introduced in the previous chapter may be regarded as abbreviations.

Table 3.1:

DLP - an overview		
non-logical variables	$v := t$	assigns the term t to the non-logical variable v
object creation	$O = new(c)$	creates a passive instance of object c
	$O = new(c(t))$	creates an active instance of object c
allocation	$O = new(c@N)$	creates a passive instance of object c allocated at node N
	$O = new(c(t)@N)$	creates an active instance of object c, allocated at node N
channels	$C = new(channel)$	creates a new channel
	$C!t$	output statement for term t over channel C
	$C?t$	input statement for term t over channel C
process creation	$Q = O!G$	requests the evaluation of G by the object to which O refers
resumptions	$Q?$	requests the results of a remote goal evaluation
synchronization	$accept(e_1, ..., e_n)$	accepts any call satisfying an accept expression e_i

Chapter 4

Knowledge representation and inheritance

- It seemed easy when I thought of it... Not words. An act. I won't write anymore. -
Cesar Pavese, This business of living

An important contribution of object oriented programming has been the use of inheritance, in defining objects that are almost alike, or more generally speaking in specifying the relations between objects.

Inheritance may occur in a number of guises. Most simply, inheritance may be viewed as code sharing. For the inheriting object, extra code must be specified to effect that the object in some sense specializes the inherited object. In a typed setting we would be tempted to speak of the inheriting object as belonging to a subtype of the type of the inherited object. However, in [America, 1987a] it is adequately observed that subtyping is a far more abstract notion, that encompasses inheritance by code sharing. Objects in DLP may inherit non-logical variables from other objects as well as clauses, by code sharing.

A rather different form of inheritance, found for instance in Actor languages [Agha, 1986], is achieved by delegating messages to the inherited objects, whenever the inheriting object does not provide any specific functionality for dealing with a method call. Examples will show how inheritance by message delegation may be programmed in DLP.

In chapter 2 it has been shown how to extend Prolog with primitives for parallelism that fit within the framework of object oriented programming. The objective of this chapter will be to illustrate how to use object oriented constructs, such as

inheritance and method delegation, for solving problems of knowledge representation.

4.1 Specialization by inheritance

An object may specialize another object by inheriting its code, and adding some specific functionality. In section 2.1.2 we have introduced the notation

object a:b { ... }

for indicating that object a inherits both the non-logical variables and the clauses of object b. Inheritance is static, in that the non-logical variables and the clauses of the inherited objects are just added to those of the inheriting object.

We may now define a semaphore, as respectively a specialization of a number and a counter, by declaring

semaphore

```
object number {
var n = 0.

put(N) :- n := N.
value(N) :- N = n.
}

object counter : number {

inc() :- n := n + 1.
dec() :- n := n - 1.
}

object semaphore : counter {

semaphore(N) :- put(N), run.

p() :- dec().
v() :- inc().

run :-
      value(N),
      ( N ≡ 0 → accept(v); accept(p,v) ),
      run.
}
```

Hence a *semaphore* is a kind of *counter*, which itself is a kind of *number*. In this, somewhat artificial example, the only active object is a semaphore, active in the sense that it contains a constructor clause. The non-logical variable n is inherited from the object *number*, via the object *counter* which provides the methods for incrementing or decrementing n.

Conceptually, when creating an instance of *semaphore*, an object is created that contains n as a local non-logical variable and as clauses all clauses occurring in the objects involved. As a consequence, when calling the constructor for *semaphore*, the non-logical variable of the newly created object is assigned a value by the call to *put*.

Multiple inheritance is allowed. We may put

```
object a:(b,c) { ... }
```

to effect that object a inherits both b and c. Moreover inheritance is recursively applied, in that also objects inherited by inherited objects are inherited. Cycles in the inheritance graph may occur. Nevertheless, each object is inherited only once, since it is checked whether it has already been inherited. We say that an object occurs earlier than another object in the inheritance list, if in applying inheritance recursively (depth first, in left to right order) it is dealt with before the other object. For dealing with cycles and possible clashes it is convenient to assume that the inheriting object itself occurs as the first object in the inheritance list. As an example, the declaration

```
object a:(b,c) { ... }
object b:(d,a) { ... }
object d:e { ... }
```

results in the list $a \cdot b \cdot d \cdot e$ as the inheritance list for a, the list $b \cdot d \cdot e \cdot a$ for b and $d \cdot e$ for d.

Clashes may occur, when using such a copying method for inheritance, between non-logical variables with the same name, and between clauses defining a similar predicate.

Clashing non-logical variables are treated in a very simple-minded way. When encountering multiple non-logical variables with the same name, and possibly different initializations, the one belonging to the object occurring first in the inheritance list is chosen. Since the inheriting object is considered to occur as the first object in the inheritance list the initializing expressions for variables defined for the object itself take precedence over other initializations.

Overriding versus backtracking For clauses that define a similar predicate, occurring in different objects, there is the choice between inheriting them in an overriding fashion (by overwriting them) or to allow backtracking over all clauses occurring in objects in the inheritance list.

Overwriting clauses implies that whenever a clause from the inheriting object succeeds, no clause of the inherited objects is tried for solving the (sub)goal. Since DLP is intended for knowledge based applications the clauses of the inherited objects are added to the clauses defined for the object itself, in the order in which the inherited objects occur in the inheritance list.

4.2 Message delegation

Extensive research has been invested in using Horn clause logic for querying and updating a database. Partly this research has been motivated by the flexibility of Horn clause logic for defining virtual attributes, the values of which are not given by actual data items but must be computed from the information stored. A somewhat radical extrapolation of these attempts is represented by approaches

using objects to store information and to guide the retrieval. Such retrieval may partly take place by evaluating clauses. Cf. [Houtsma and Balsters, 1988].

As an example of an object oriented approach to (for instance) modeling the hierarchic organization of a firm, consider the following program fragment, telling us that an employee has a manager, and that in his turn a manager is an employee.

firm

```
object employee {
var manager.
employee(M) :- manager := M, run.
run :- accept(any), run.
request(S,A) :- manager!approve(S,A).
}
object manager : employee {
manager(M) :- employee(M).
approve(S,approved) :- unimportant(S).
approve(S,rejected) :- unreasonable(S).
unimportant(domestic_trip).
unimportant(presentation).
unreasonable(double_salary).
}
```

For an employee to undertake any action he/she must request his/her manager for approval. This is implemented by asking the manager stored in the non-logical variable of the employee to approve of that particular action. Since the manager is an employee, he himself may have to ask for approval of his manager. Chaining upwards in the hierarchy a manager must exist that has no obligation to ask for approval. This manager may be created with *self* as an argument for the constructor, which makes asking for approval a local affair, depending on what the top manager himself considers reasonable and important. Consider the goal

```
:-
  M = new(manager(self)),
  E = new(employee(M)),
  E ! request(domestic_trip, A).
```

as an example of creating a one level hierarchy.

Backtracking Applying inheritance in integrating *data* and *knowledge* may require that knowledge is used in an additive fashion. In other words, the pieces contained in the individual objects may all contribute to the solution. We have adapted the example given in [Houtsma and Balsters, 1988] to illustrate the need for using all knowledge available.

For convenience, we introduce an abbreviation for accessing the value of the non-logical variables of a particular object. We use the expression

- *O@name*

to stand for the value of the non-logical *name* of the object *O*.

The idea of the example is that there are two kinds of researchers: associates and professors. They have similar interests, in that they both know of a number of topics.

A researcher has a name and works in some field, that is given as a list of topics. A researcher knows of a topic whenever he has visited a conference that somehow concerns the topic. Naturally, a researcher knows all of his field.[1]

researcher

```
object researcher : environment {
var name, field.

researcher(N,F) :-
      name := N,
      field := F,
      work().

work() :- accept(any), work().

knowsof(Topic) :- member(Topic,field).

knowsof(Topic) :-
            visitor(name, Conference),
            Conference ! concerns(Topic).

}
```

The constructor for *researcher* stores the name of the fellow and his *field* of interest, and then calls for *work*() that is interrupted only to answer what he knows of. A researcher inherits a number of things from an *environment*. For instance, whether a researcher is a *visitor* of a conference is assumed to be known by its environment.

An associate is a researcher that works under a professor. Apart from what he knows of as a researcher he is also assumed to know of everything his professor knows of.

[1] We assume that both *member* and *append* are available as system predicates.

associate

```
object associate : researcher {

var prof.

associate(N,F,P) :-
        prof := P,
        researcher(N,F).
associate(R,P) :-
        prof := P,
        researcher(R@name, R@field).

knowsof(Topic) :- prof ! knowsof(Topic).

}
```

In case a new associate already is a researcher of some unidentified kind his non-logical variables of interest may simply be copied, as expressed in the second clause for *associate*.[2] The other constructor explicitly takes his name and field of interest.

A professor is a researcher and apart from what he knows of as a researcher he also knows of the topics of all conferences for which he is a member of the program committee.

professor

```
object professor : researcher {

professor(R) :- researcher(R@name, R@field).
professor(N,F) :- researcher(N,F).

knowsof(Topic) :-
        committee( name, Conference),
        Conference ! concerns(Topic).
}
```

Similarly as for an associate, two constructors are provided to be able to cope with his past. In the clause for *knowsof*, the non-logical variable *name* comes from being a *researcher*. Being a researcher, a professor inherits the *environment* declared for researchers. Whether a professor is a member of a program committee must be known by the environment.

Next we define a conference, for instance one concerning parallel processing.

[2]Since inheritance is static we cannot take the object itself. Instead we must copy its contents. The resulting *associate* object is an object that contains both the non-logical variable *prof* and the non-logical variables of the *researcher* object.

conference

```
object parallel_processing_conference {
var topics = [ computers , concurrency ].

concerns(Topic) :-   member(Topic, topics).
}
```

And all that remains to be done is to declare in what *environment* our actors live. We assume that an environment is just a collection of facts by which, among other things, it may be established that one is a member of a program committee or a visitor of a conference.

environment

```
object environment {
var conferences = [ parallel_processing_conference ].

committee(knuth, parallel_processing_conference ).
committee(turing, parallel_processing_conference ).

visitor(newman, parallel_processing_conference ).
visitor(overbeek, parallel_processing_conference ).
}
```

Now the reader is invited to check what associate *newman* knows of.

```
:- P = new( professor( turing, [computing] ) ),
   A = new( associate( newman, [engineering], P )),
   A !bagof(X, knowsof(X), Topics).
```

As a remark, we may ask an associate for all topics he or she knows of, since being a researcher he or she is willing to accept anything that interrupts the work.

4.3 Distributed problem solving

The motivation for proposing DLP for implementing problem solving and reasoning systems is twofold.

First, there is the obvious need to structure knowledge in a, what once was called, frame-like system. That is were the constructs derived from object oriented programming come in. Cf. [Aikins, 1980].

Secondly, there is the wish to distribute the reasoning among independent parts of the system in order to gain efficiency by exploiting parallelism. Cf. [Davis, 1980].

In the realm of medical diagnosis systems an early attempt at distributing knowledge can be found in [Gomez and Chandrasekaran, 1981]. Apart from proposing how to distribute knowledge as a hierarchy of medical concepts, it is suggested there to regard each concept as a specialist that is busy checking whether the

observed symptoms justify the diagnosis of the particular disease represented by the concept.

With respect to the organization of knowledge the key issues are the efficiency with which a diagnosis is found and how to focus and control the search. Although lacking any medical knowledge whatsoever we will sketch in global lines what the architecture of such a system might look like in DLP. The knowledge we do have of rule based medical expert systems comes to a great extent from [Lucas, 1986].

The toy system that we will propose here is rule based, in the sense that the knowledge concerning a particular disease is stored in production rules that operate on a collection of established data. The declaration for the object containing the data is given below.

facts

```
object facts {
var data = [].

facts(D) :- data := D, run.

run :- accept(any), run.

holds(X) :- member(X,data).
add(X) :-
      append([X],data,R),
      data := R.
}
```

Clearly, one can ask whether a particular data item holds, that is if it is contained in the list of data. Also, data items may be added. For a particular patient the facts are accessible by all doctors that are trying to assess some diagnosis.

The inference engine is the part responsible for the actual derivation of a possible diagnosis. It operates on a collection of facts, by applying knowledge contained in rules.

infer

```
object infer {
var knowledge, facts.

infer() :-
        accept(facts),
        accept(knowledge),
        run.

run :- accept(any), run.

knowledge(K) :- knowledge := K.
facts(F) :- facts := F.

derive(H) :- facts!holds(H),!.
derive(H) :-
        knowledge!rule(P,H),
        test(P),
        facts!add(H).

test([]).
test([X|R]) :-
        derive(X),
        test(R).

}
```

Before it may start to derive new facts, it must have the possession over an initial collection of facts and a body of knowledge. This requirement is met by including the sequence of accept statements in the constructor, and by providing the corresponding methods.

The derivation itself proceeds by backward chaining. When a particular hypothesis is not yet established as a fact, its derivation is attempted by (recursively) applying some knowledge rule, stating that the hypothesis holds if certain premisses are fulfilled. If the derivation succeeds, the hypothesis is added to the facts already established.

Now we come to defining a *doctor* as an entity that, having the capacity to infer, tries to establish a diagnosis on the basis of the facts and her knowledge. Somewhat naively, our doctor needs to refresh his knowledge with each case that presents itself. Due to this knowledge she knows what she is looking for. Establishing a diagnosis amounts to checking the possible hypotheses.

doctor

```
object doctor {
var hypotheses, inference.

doctor() :- accept(inform), accept(diagnosis), doctor().

inform(K,F) :-
        I = new(infer()),
        I!facts(F), I!knowledge(K),
        hypotheses := K@diagnoses,
        inference := I.

diagnosis(D) :-
        member(D,hypotheses),
        check(D).

check(D) :- inference!derive(D).
}
```

Knowledge representation To illustrate the work of a medical doctor we must supply some (medical) knowledge. Since our intent was to give a clue how to organize knowledge in a hierarchic fashion we will start with specifying a generic object disease.

disease

```
object disease {
var diagnoses = [disease],
    causes = [liver,lungs].

rule([high_temperature],fever).
rule([fever],disease).
}
```

Apart from the rules that contain the knowledge needed to establish a diagnosis, the object contains also a list of possible diagnoses, which for the generic case simply states that a patient may have a disease. The rules inform us that a patient has a disease if he has fever, that is a high temperature.

The knowledge concerning a disease contains also an indication of its possible causes. However, such knowledge is only necessary when one wants to consult a specialist.

As an example of refining the generic object disease to a particular case consider the declaration for a liver disease.

liver

```
object liver : disease {
var diagnoses = [liver_disease],
    causes = [intrahepatic, extrahepatic].
rule([disease,yellow_skin],liver_disease).
}
```

Notice that, in accordance with our discussion in section 4.1, the values of the variables *diagnoses* and *causes* are determined by the object containing the knowledge pertaining to a liver disease.

Before going to a specialist, lets see what a doctor can do.

```
:-
    F = new(facts([high_temperature,yellow_skin])),
    M = new(doctor()), M!inform(liver,F).
    M!diagnosis(D).
```

After being informed of the symptoms of our patient, the doctor will on the basis of her knowledge concerning liver diseases, correctly diagnose a liver disease.

Before being able to continue, we must enlarge our body of knowledge, according to the possible causes indicated for each disease.

knowledge

```
object lungs : disease {
var diagnoses = [tuberculosis, asthma],
    causes = [].
rule([coughing,bleeding],tuberculosis).
rule([coughing,red_eyes],asthma).
}


object intrahepatic : liver {
var diagnoses = [intrahepatic],
    causes = [].
rule([liver_disease,sweating],intrahepatic).
}


object extrahepatic : liver {
var diagnoses = [extrahepatic],
    causes = [].
rule([liver_disease,bleeding],extrahepatic).
}
```

The knowledge that we have added contains some fictional rules concerning lung diseases, and some knowledge refining what we know about liver diseases. Neither for lung diseases nor for the intrahepatic and extrahepatic variants of liver diseases we seem to know any causes.

Cooperation and communication Our objective is to let a number of specialists cooperate in treating a patient. A *specialist* is a doctor that is specialized in some field of (medical) knowledge. Given a case, a specialist may state a diagnosis and advice what other specialists a patient must consult.

specialist

```
object specialist : doctor {
var field.

specialist(K) :-
        field := K,
        practice.

practice :- accept(any), practice.

advice(F,D,A) :-
        inform(field,F),
        diagnosis(D),
        A = field@causes.
}
```

For treating a patient, we create a *clinic*. We assume that a clinic has an unlimited supply of specialists in all possible disciplines.

clinic

```
object clinic {

case(C,D) :-
        F = new(facts(C)),
        treat(disease,F,D).

treat(K,F,[D|R]) :-
        M = new(specialist(K)),
        M!advice(F,D,A),
        try(F,A,R).

treat(K,_,[]).

try(F,[],[]).

try(F,[K|T],[D|R]) :-
        treat(K,F,D) & try(F,T,R).

}
```

When a patient comes in, his complaints are filed and his treatment is started by assuming that he has a disease. The result of the treatment will be a list of

possible diagnoses. The treatment is initially taken care of by a specialist that must assess whether the patient has a disease.

If a specialist succeeds in establishing a diagnosis, then her advice with respect to the possible causes of the disease is used to try further treatment. If the list of possible causes is empty, then no possible diagnoses will be added, naturally. On the other hand, each possible cause may give rise to a new treatment, that may be executed in parallel with the exploration of alternative causes.

Below we present the worst case that we can imagine.

```
:- clinic!case([high_temperature,yellow_skin,sweating,coughing,bleeding],D),
   write(D).
```

Please notice that apart from treating multiple cases simultaneously, we also may encounter parallelism in the exploration of possible causes, due to the parallel and-operator introduced in section 2.6. A more effective distribution of processing may be obtained by using the allocation primitives introduced in section 2.7.

Chapter 5

Design perspectives

- I have a commission for you... A literary matter. I know my limitations; the skill of rhymed malice, the arts of metrical slander, are quite beyond my powers, You understand. -
Salman Rushdie, The Satanic Verses

The primary motivation in developing DLP was to provide an environment suited for implementing distributed knowledge based systems, such as expert systems and systems for distributed problem solving. It is widely recognized that, for modeling a domain of expertise or problem solving, preference should be given to a declarative language for knowledge representation. Nevertheless, although we have clearly stated such a preference by taking Horn clause logic as a base language for DLP, it is our (strongly held) opinion that a mere declarative language does not provide sufficient means to control search and inference to the extent necessary for building systems with non-trivial functionality. Partly, this control can be effected by relying on a standard way of evaluation, such as the left-to-right evaluation of Prolog goals, or by defining a strategy adequate for the problem at hand as described in [Wos et al, 1984]. A language such as Prolog, however, allows more drastic means for effecting control, notably by means of the cut and the *assert* and *retract* statements, the use of which is justifiably frowned upon by those taking a strictly declarative stand.

When thinking about parallelism the ideological gap between a pragmatic approach and what is theoretically justifiable seems to widen, inevitably. Serious difficulty in exploiting the parallelism inherent in the computation model of logic programming languages has been caused by the need to restrict the number of processes created for evaluating a goal, and to avoid excessive communication between processes. Cf. [Conery, 1987]. Moreover, for reasons of synchronization several ad hoc solutions have been proposed, such as the *wait* statement in Par-

log, and the annotated variable in Concurrent Prolog. Cf. [Ramakrishnan, 1986]. Rather than adhering to some established conception of concurrent logic programming (cf. [Ringwood, 1988]), we have decided to investigate what, to our mind, is needed for distributed problem solving, and more specifically for distributed logic programming.

With respect to issues of knowledge representation we have taken the stand that, although we wish to support a declarative language, we also wish to provide a programmer with sufficient means for programming a suitable solution, even if this would mean leaving the declarative realm. For efficiently exploiting parallelism, this seems unavoidable; for knowledge representation, although not unavoidable, providing non-declarative constructs inspired by the paradigm of object oriented programming simply means following a trend, that is giving in to the need expressed by programmers in the field.

Having sketched the motivation for developing the language DLP, we wish to take a step back, in order to reflect on the concept of distributed programming and the origins of DLP in the design space of computer programming languages. In the end we will ask ourselves the question whether there really is a need for such a language.

5.1 The concept of distributed programming

As a starting point, lets try to delineate the notion of a distributed programming language:

> Distributed programming languages support computation by multiple autonomous processes that communicate by message passing rather than shared variables and may be implemented by geographically separated networks of communicating processes. [Bal et al, 1989]

Clearly, at first sight our language DLP meets this general criterion; but, since we are not so much interested in reassurance as in characterizing the design principles of our language, let us take a look at the features that characterize distributed programming languages and the alternatives facing a designer for each of these features. A list of the languages that we refer to in this section is given in the table below.

Ada	[DoD, 1982]
C++	[Stroustrup, 1986]
Concurrent C	[Tsujino et al, 1984]
Concurrent Prolog	[Shapiro, 1986]
CSP	[Hoare, 1978]
Emerald	[Black et al, 1987]
Linda	[Gelernter et al, 1986]
Occam	[Inmos, 1984]
Parlog	[Clark and Gregory, 1986]
POOL	[America, 1987]
Smalltalk	[Goldberg and Robson, 1983]

As a first characteristic, we encounter, following [Bal et al, 1989], the requirement of *parallelism*. There seems to be abundant choice in what to take as the unit of parallelism: processes (CSP), tasks (Ada), active objects (POOL), multithreaded objects (Emerald), clauses (Concurrent Prolog and Parlog), or even statements (Occam). One of the questions that we must keep in mind when studying distributed languages, is the support a language offers for mapping parallel computations on an actual distributed architecture. An important issue also, is the grain of parallelism of the processes created to perform a computation.

Another decision that must be made concerns the way *communication* is dealt with. As alternatives we encounter data sharing and message passing. Despite the apparent contradiction with the general characterization given in the quotation above, we mention Linda as an interesting example of data sharing.[1] Also Concurrent Prolog and Parlog, utilizing shared logical variables as the medium of communication, deserve to be classified among the distributed languages. Choosing for message passing we may employ point to point connections (CSP), channels (Occam, Delta Prolog) or broadcasting. Communication may to a certain extent be non-deterministic. For example, both the select statement of Ada and the guarded Horn clauses of Concurrent Prolog and Parlog result in a choice for a particular communication, ignoring alternatives.

As an additional feature, some of the languages mentioned in [Bal et al, 1989] handle *partial failure* by offering exceptions, atomic sections or recovery mechanisms. Such failure may be due to, for example, hardware errors or the violation of integrity constraints. We wish to remark that such failures are rather different from the failure encountered in a language such as Prolog. Failure in Prolog is one of the possible outcomes of a computation, it may even be used to generate all the solutions to a particular goal. In developing DLP we have simply paid no attention to partial failure or error recovery.

In the literature we came across three computation models underlying distributed programming. The most basic of these is that of communicating sequential processes, first presented in the influential paper [Hoare, 1978]. Object based concurrent languages may be regarded as extending this basic model, by generalizing communication and providing additional features for synchronization and protection. Finally, the model underlying concurrent logic programming languages is perhaps the most flexible of these, since it allows to mimic the two previous ones.

5.1.1 Communicating sequential processes

The basic model of a distributed programming language is that of a group of sequential processes that run in parallel and communicate by message passing. By a sequential process we mean a process with a single thread of control.

The prime example of a language supporting communicating sequential processes is CSP. [Hoare, 1978]. In CSP we may create a fixed number of parallel

[1] The contradiction is resolved by making a distinction between *physical* data sharing and *logical* data sharing. Obviously, we mean the latter here. Logical data sharing provides the programmer with the illusion of common data by hiding the physical distribution of the data.

processes by using a parallel operator. Each process consists of a name and a body, that is a sequence of statements. Communication between processes is achieved by using *send* and *receive* statements. As an example, consider the program

$$[p_1 :: p_2!3 \parallel p_2 :: p_1?n]$$

where process p_1 is about to execute the statement $p_2!3$, sending the value 3 to process p_2 and p_2 is about to execute $p_1?n$, to receive a value from p_1 that is assigned to the (integer) variable n. The proposal in [Hoare, 1978] also provides for pattern matching in communication. Also, a guarded command is offered, that allows to select a particular alternative dependent on the possibility of a communication. Due to its synchronous nature, communication in CSP is said to subsume synchronization mechanisms such as semaphores, events, conditional critical regions and monitors. C.f. [Andrews and Schneider, 1983].

Occam is a language that embodies a number of the features of CSP. [Inmos, 1984]. A noticeable difference with CSP is that communication statements do not address processes, but that instead communication takes place over channels. Occam is the machine language of the transputer. Transputers may be connected into a network. The language provides a mechanism for mapping symbolic channels to the actual hardware channels implementing the network. In contrast to CSP, Occam does also provide facilities for mapping processes on processing units.

Perhaps the major advantage of such a language is that it is efficiently implementable, giving the programmer full control over the hardware resources. C.f. [Bal et al, 1989]. From the point of view of program design, however, the necessity of such control may be considered a disadvantage. From this perspective, languages with inherent parallelism seem more suitable. As alternatives to languages supporting the basic model we have: object oriented languages, that support concurrently executing active objects; functional languages, that allow parallel evaluation due to the absence of side-effects; and logic programming languages, that enable to work in parallel on parts of the AND/OR proof tree.

5.1.2 Objects and concurrency

The notion of an object based language covers a wide range of languages, including Ada, POOL, Emerald, Smalltalk and C++. An object may be characterized as an entity that has a collection of operations and a state that remembers the effect of operations.[2] C.f. [Wegner, 1987].

Apart from providing a construct to group operations, and a facility for defining an abstract interface to a collection of data, an object provides an additional protection mechanism since access and modification of the data it encapsulates is allowed only through the use of the operations defined for the object, the so-called methods.

[2] In accordance with the literature, we speak of object based languages and reserve the phrase object oriented language for the languages offering inheritance as an additional mechanism. See [Wegner, 1987] for a detailed discussion of the dimensions of object based language design.

Even when considering method calls as (synchronous) message passing, object based languages may fit well in a sequential model of computation, assuming that an object is passive except when answering a method call.

Extending the sequential object model to include parallelism may be achieved simply by allowing an object to be active on its own account, that is when not answering a message. As alternative ways to obtain parallelism, we mention the possibility to employ asynchronous communication as encountered in the Actor languages [Hewitt, 1977], [Agha, 1986]; or to add processes as an orthogonal concept to the language. A drawback of the last solution however is the need to provide extra facilities for synchronization and mutual exclusion. Active objects seem in this respect to be a much more natural solution, since such protection is already offered by the method interface, assuming that only one method is answered at a time. C.f. [America, 1989b].

The notion of active objects, that may be created dynamically, has been adopted by the language POOL. See also section 9.5.1. Each object may have own activity, called the body of the object, that is started as soon as the object is created. The own activity of the object is interrupted to answer a method call when a so-called *answer statement* is encountered. The answer statement introduces a certain degree of non-determinism, since although a number of method calls may be considered acceptable only one of these will be chosen.

The communication model of method calls in POOL has been derived from the *rendez-vous* as encountered in Ada. The rendez-vous, as an interaction between processes, has a two-way nature. It generalizes in this respect the primitives provided by for example CSP, that allow only one-directional point-to-point communication. In the terminology of POOL, the rendez-vous model is based on three concepts: *a method declaration*, that is like the declaration of a procedure having the right to access the private data of an object; *a method call*, that is like a procedure call but with an object as an additional parameter; and an *answer statement*, to interrupt the own activity of an object and to state the willingness to accept a particular method call.[3] Answer statements allow to suspend the acceptance of a method call, dependent on the state of the object. Even stronger acceptance conditions may be imposed in Concurrent C that allows to inspect the actual parameters of a call to determine acceptance.

The language POOL enables the programmer to locate a newly created object on a particular processor by so-called pragmas, listing the set of processors from which the system may choose.

As another object based distributed language, we wish to mention Emerald. Just as POOL, Emerald offers the possibility to create active objects dynamically. An important difference between POOL and Emerald however is that Emerald allows multiple threads of control: one object can be active answering a number of method calls. Moreover, the processes created for answering a method call run in parallel with the process executing the own activity of the object. A monitor construct

[3] In the context of Ada one speaks of respectively an *entry declaration*, an *entry call* and an *accept statement*.

is provided to enable synchronization and protection when accessing local data shared by processes active for the object.

In addition to a facility for mapping objects and processes to processors, Emerald supports the migration of objects and processes by allowing them to move or to be moved from one processor to another.

5.1.3 Concurrent logic programming

The model underlying concurrent logic programming forms a rather radical departure from the two previous models in that communication is essentially effected through shared logical variables. Parallelism is inherent in the computation model of logic programming languages, because of their declarative nature. Basically, two kinds of parallelism can be distinguished: AND-parallelism, due to the parallel evaluation of the atoms in a compound goal; and OR-parallelism, that arises from trying multiple clauses simultaneously for finding a solution to a goal atom.

Although there are a number of attempts at implementing parallel Prolog this way, the two major representatives of concurrent logic programming, Concurrent Prolog and Parlog, have based their approach on the additional assumption of committed choice non-determinism and restricted unification.

Unlimited OR-parallelism, required to find all solutions to a goal, may result in an uncontrollable amount of processes. To restrict OR-parallelism, guarded Horn clauses were introduced. A guarded Horn clause is a clause of the form

$$A :- G_1, ..., G_n \mid B_1, ..., B_m.$$

where A is the head of the clause, $G_1, ..., G_n$ the guard goals and $B_1, ..., B_m$ the actual body of the clause. When a goal atom is evaluated, all clauses of which the head unifies with the atom are selected and the guards of these clauses are evaluated in parallel. The first clause of which the guard is evaluated successfully is committed to. The alternative solutions to the goal atom, embodied in the competing clauses, are thrown away. Since only one clause is chosen, backtracking over alternative solutions is impossible, once the commitment to that particular clause is made. What is allowed as a guard influences the expressiveness of the language in a significant degree, and for that matter the difficulty of implementing it. See [Shapiro, 1989] for an extensive discussion of this topic.

Unrestricted AND-parallelism, that is the parallel evaluation of the atoms in a compound goal, may result in incompatible bindings of the logical variables involved. To handle this problem, both Concurrent Prolog and Parlog require to indicate which atom acts as the producer of a binding to a variable and which atoms are merely consuming the binding. Concurrent Prolog uses annotations to indicate the variables that must be bound to a term to enable the evaluation of the atom in which they occur to proceed. Parlog, on the other hand, uses mode declarations, indicating the input/output behavior of the arguments of a predicate.

Despite the absence of backtracking and the restrictions on unification concurrent logic programming languages offer a very versatile mechanism for implementing distributed systems. C.f. [Shapiro, 1989]. In particular these languages allow to implement active objects with state variables in a very elegant way. This is achieved by defining clauses for objects according to the scheme presented below.

```
obj(State, [Message|Messages]) :-
        handle Message,
        update State to State',
      obj(State',Messages).
```

An object is implemented as a tail-recursive processes, that receives messages and updates its state if necessary. C.f. [Shapiro and Takeuchi, 1983]. As an example, consider the clauses implementing a counter in Concurrent Prolog. For DLP variants of this counter see sections 2.2 and 2.3.

ctr

```
ctr(N,[inc() | T]) :- N1 = N + 1, ctr(N1,T).
ctr(N,[value(N) | T]) :- ctr(N,T).
```

The first argument of *ctr* represents the state of the object, that is passed as an argument to the recursive call, appropriately modified if necessary. The second argument represents the stream of incoming messages, with the tail unspecified to await later binding.

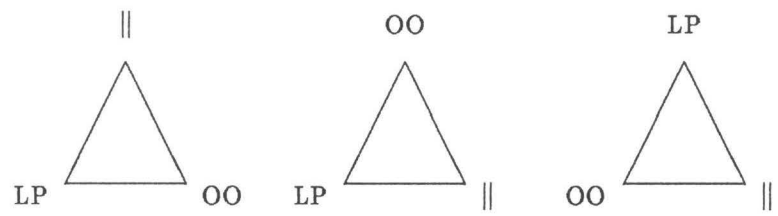
Based on this scheme, two languages combining logic programming and object oriented programming have been proposed: Vulcan and Polka. See section 5.2.2.

Concurrent logic programming languages offer fine grained parallelism. As an additional feature for dynamically mapping computations to processes [Shapiro, 1984] proposes a turtle notation for executing Concurrent Prolog programs on a grid of processors. See also section 2.7.

5.2 DLP = LP + OO + ||

Now that we have explored the design space of distributed programming languages we must trace the origins of our language DLP and motivate the decisions made.

Before trying to justify our own decisions we may well look at the alternatives we have in combining the three components of our language: logic programming, object oriented programming and parallelism. The diagrams below suggest three possible ways of arriving at such a combination.



Either we may start from a combination of logic programming and object oriented programming (LP + OO) and add concurrency to it; or we may take a

combination of logic programming and concurrency (LP + ||) and extend this to include object oriented programming; or we may take a concurrent object oriented language (OO + ||) and lift this in order to support logic programming.

According to this scheme we may classify a number of related approaches as in the table below.

	Language	LP	OO	||	References
LP + OO	Logical Objects	+	+	−	[Conery, 1988]
	SPOOL	+	+	−	[Fukunaga and Hirose, 1986]
	Communicating Prolog Units	+	+	+	[Mello and Natali, 1986]
LP + ||	Vulcan	+	+	+	[Kahn et al, 1986]
	Polka	+	+	+	[Davison, 1989]
	Delta Prolog	+	−	+	[Pereira and Nasr, 1984]
OO + ||	MultiLog	+	+	+	[Karam, 1988]
	OrientK/84	+	+	+	[Ishikawa and Tokoro, 1986]

To indicate where DLP must be located in the table above, we may remark that our language is very much alike MultiLog. In contrast to MultiLog, however, DLP supports backtracking over the results of a method call, a property it shares with the languages falling under the heading LP + OO, combining logic programming and object oriented programming.

In the sections that follow, we will investigate the merits of DLP with respect to our classification scheme. For each of the combinations LP + OO, LP + || and OO + || we will try to delineate the central issue tackled and the problems encountered in completing the triangle. After describing the related approaches that exemplify that particular combination we will discuss the solutions provided in DLP.

5.2.1 LP + OO

The central question in combining logic programming and object oriented programming is how to implement objects with internal states. Such states must be hidden. An object, in other words, must guarantee a certain protection with respect to the access and modification of its state by providing a suitable method interface.

Logical Objects provide an extension of Prolog with so-called *object clauses.* [Conery, 1988]. An object clause is a generalized Horn clause of the form

```
method(X), object(S) :- ..., object(S1).
```

Such a clause is activated when a compound goal contains both a literal unifying with method(X) and a literal unifying with object(S). In this way, after having created an object with an initial state, goal literals may be interpreted as method calls to an object, that as a result may change its state in an invisible way.

A characteristic feature of this proposal is that it allows the usual Prolog backtracking over method calls. The advantage of the approach sketched is that "it does minimal damage to the pure logic programming foundations".

As a drawback, we may mention that *referential transparency* is lost: the meaning of an atom is no longer a fixed relation but relative to all possible states of the object for which the atom is a method.

Two implementations exist. The first is a meta-interpreter, which maintains a list of object states. The second exists in the form of a preprocessor translating object clauses to Prolog clauses. An implementation based on an extended version of the Warren Abstract Machine is planned.

SPOOL is a Prolog based object oriented language. See [Fukunaga and Hirose, 1986]. It extends Prolog with classes, instance variables, and message passing to enable the communication between instances of clauses.

The clauses defined for a class act as methods. In a method call of the form

send(Receiver,Message)

both the *Receiver* and the *Message* may be variables, thus allowing backtracking both with respect to the destination of the message and the method that will be applied. This rather unwieldy backtracking behavior is made possible by embedding methods in Prolog clauses representing a class of objects. See [Yokoi, 1986] for the details of the compilation of SPOOL to Prolog.

The advantage of this approach is that it combines the expressiveness of a logic programming language with the organizational capabilities of object oriented programming.

To our mind, the backtracking that may arise from anonymous method calls, where the destination is a variable, seems somewhat extravagant.

In [Fukunaga and Hirose, 1986] an application of SPOOL for producing program annotations is described.

Communicating Prolog Units extend Prolog with object oriented programming constructs by employing meta-programming techniques to define and handle the interaction between *units* of Prolog clauses. [Mello and Natali, 1986].

Communication between units is possible by using a meta-predicate of the form

send(*Destination*, *Goal*, *Answer*)

where *Destination* represents the unit by which the *Goal* must be evaluated. The results are collected in the *Answer* argument.

Units act as objects, since an internal state may be represented by clauses, and modified by asserting or retracting clauses.

A process, that is an instance of a unit, may influence the way external requests are handled by providing synchronization clauses of the form

entry(...), *accept*(...) :- *body*(...).

When a goal in a *send* statement unifies with the *entry* part of the head of the clause, the evaluation of the *body* is postponed until the *accept* part of the head is satisfied. The functionality of these synchronization clauses resembles that of a rendez-vous.

The approach followed here has resulted in a number of constructs with great expressive power, due to the application of meta-programming techniques. A clear advantage is that the backtracking behavior of Prolog is retained.

Currently, only a prototype implementation in Prolog does exist.

In comparison with the approach taken for Logical Objects, that remains within the logic programming framework, the language SPOOL is of a more hybrid nature. A common characteristic of the three approaches sketched above is that they allow backtracking over method calls, which seems partly to be a result of embedding object oriented constructs in Prolog. The extent to which this is a deliberate design decision or an accidental quality due to the implementation is not altogether clear.

Both Logical Objects and SPOOL are sequential languages. For Communicating Prolog Units, concurrency is introduced by allowing units, that are like objects, to be active.

DLP supports objects with states primarily by what we have called non-logical variables. Passive objects merely respond to method calls, that may be regarded as the evaluation of a goal by an object. With respect to the backtracking behavior of such method calls we have decided to strive for full compatibility with Prolog. In other words, it must make no difference when a goal is evaluated by means of a method call or in the ordinary way, provided the clauses needed for evaluating the goal are available.

Having objects with non-logical variables complicates matters a bit. We have decided that backtracking over the results of a method call does not undo any modification to the non-logical variables of the object to which the call was addressed. This decision needs some justification.

As a first observation, we wish to state that from the outside the non-logical variables of an object are invisible. Calling a method results in binding logical variables to some value, possibly in a number of alternative ways, or failure. How these results are computed is the responsibility of the object.

Secondly, we may remark that we conceive of non-logical variables as an abstract representation of entities such as a database. Undoing modifications to such entities at every attempt at backtracking over a method call may lead to serious problems, in particular since in a logic programming context failure is a natural outcome of a computation.

We have already made clear that we wish to distinguish between partial failure in the sense of hardware errors or the violation of integrity constraints and failure in the logic programming sense. To our mind, automatically undoing modifications to non-logical variables may be better handled by additional features such as atomic sections or recovery mechanisms. C.f. [Klint, 1985].

In the third place, we like to point out that although for a sequential language automatic recovery on backtracking may seem a feasible solution, when introducing concurrency a number of problems arise due to the interaction with other objects. What is the scope for which we must guarantee protection? And, how do we handle the possible interference by other method calls.

5.2.2 LP + ||

In section 5.1 we have introduced concurrent logic programming as one of the paradigms of distributed computing. We have also sketched how to use a concurrent logic programming language to implement objects. In this section we will treat two languages that provide object oriented extensions to a concurrent logic programming language.

Vulcan provides linguistic support for object oriented programming based on the computation model of concurrent logic programming. [Kahn et al, 1986].

To accommodate the need of programmers to use convenient program cliches, a preprocessor has been built that allows to dispose of the verbosity adhering to programming object oriented programs directly in Concurrent Prolog.

A program in Vulcan consists of clauses for classes and clauses for methods, possibly mingled with plain Concurrent Prolog clauses. The object oriented features supported include message sending, class inheritance and inheritance by delegation.

The advantage of such an approach is that it allows to offer the programmer a variety of constructs, that are nevertheless based on a simple and clean semantics. Another advantage may lay in the concurrent nature of the language.

As a drawback, in comparison with the two previous approaches, we may mention that backtracking is not supported; more specifically backtracking over method calls is not possible due to the committed choice character of Concurrent Prolog.

Polka is a hybrid language, combining the paradigms of concurrent logic programming and object oriented programming. [Davison, 1989]. Based on Parlog, it introduces object oriented constructs such as classes, instance variables, inheritance and self-communication. These abstractions enable to write a "higher level Parlog", reducing the verbosity of ordinary Parlog.

A very powerful mechanism is provided by the incorporation of meta-level programming constructs that allow to treat classes as first order entities. In terms of expressiveness, this last feature may be considered an advantage of this particular approach at combining logic programming and object oriented programming. The operational semantics provided for the language suggests that the semantic complexity of this hybrid language is still manageable.

It is observed that the absence of backtracking, due to the committed choice character of Parlog, may be considered an advantage or a disadvantage, dependent on the needs of the application. Polka has been used to implement, among other things, a blackboard expert system. The implementation of Polka is described in [Davison, 1989].

A restriction adhering to both languages is the absence of backtracking over the results of a method call. As a remedy one could re-introduce backtracking by integrating a concurrent language with Prolog. C.f. [Shapiro, 1989]. Merely providing an interface between the two languages is not satisfactory, simply because it does not provide an integration of the two computation models underlying

these languages. Embedding Prolog in a concurrent logic programming language, although feasible for non-flat versions allowing arbitrary goals in the guard, leads to efficiency problems. Moreover, as [Shapiro, 1989] observes, the implementation techniques for concurrent logic programming languages lag far behind those developed for sequential Prolog. C.f. [Hermenegildo, 1986]. Another option might be to extend Prolog to a language combining logic programming and concurrency. An example of such an approach is the language Delta Prolog.

Delta Prolog is a distributed logic programming language that extends Prolog to include AND-parallelism and synchronous communication with two way pattern matching. C.f. [Pereira and Nasr, 1984] and [Pereira et al, 1986]. Parallel processes may be created by using the operator || for the parallel composition of goals. Processes may synchronize and communicate with each other by means of so-called *event goals*, a construct based on the Distributed Logic notion of event. [Monteiro, 1984].

Event goals are goals of the form $T?E$ or $T!E$, where E represents an event and T a term, the event pattern. As an example, the compound parallel goal

$$T_1!E \;||\; T_2?E$$

succeeds if T_1 and T_2 unify.

Delta Prolog supports fully distributed backtracking, that occurs when one of the components of a parallel goal does not succeed. The backtracking order is then from right to left, just as in ordinary Prolog. Another occasion on which globally controlled backtracking occurs is when two compatible event goals fail to result in a communication.

The advantage of Delta Prolog is, first of all, that it includes full Prolog as a subset, and secondly, that it supports backtracking on the results of a communication. As a disadvantage, we may note that it does not provide modular features that may be of help in program design. In comparison with the rendez-vous construct, communication via events is less general, since the suspension of attempts at communication is not supported.

An implementation of Delta Prolog exists, as an extension to C-Prolog partly written in C and partly in Prolog, that allows to execute Delta Prolog programs on a network of processors.

When we compare Delta Prolog with the object oriented languages based on the concurrent logic programming model, the most obvious difference is that communication in Delta Prolog must be stated explicitly, whereas communication in the concurrent logic programming languages is mediated by shared logical variables. Another difference is that processes created for executing a concurrent logic program are usually fine-grained, whereas Delta Prolog gives rise to coarse grain parallelism.

We note that Delta Prolog allows to implement objects in a similar way as the concurrent logic programming languages. We do not consider Delta Prolog to be object oriented, however.

DLP supports concurrency in basically two ways. The first, perhaps most natural way, is to create active objects that execute their own activity in parallel with the activity of other objects. The second way is to let an object evaluate a number of method calls simultaneously.

To implement this, a notion of processes has been introduced, distinct from objects. Each active object has a so-called constructor process associated with it, that handles the evaluation of the body of the object, as expressed in the constructor clauses. Moreover, for each method call a process is created to handle the backtracking information needed to generate all answers to the call, and to communicate these to the invoking process.

When multiple processes are active for some object, we speak of internal concurrency, or multi-threaded objects. Passive objects allow unlimited internal concurrency. On the other hand, for active objects we have allowed such internal concurrency only for backtracking over alternative answers, after having delivered the first answer. The reason for this policy is that we wish to guarantee mutual exclusion between method calls for the time needed to produce the first answer. That is to say, no two method calls will be active with producing their first results simultaneously. A method call may however become active when other processes are still busy backtracking over the answers of a call.

The language DLP provides primitives for synchronous communication over channels, that are rather alike those offered by Delta Prolog. However, the backtracking that may arise during communication over channels in DLP is much more limited than the distributed backtracking supported by Delta Prolog, in that it occurs locally within the confines of the process stating the input goal. The rendez-vous supported by DLP does not suffer from such a restriction. Moreover, we claim that our rendez-vous allows to impose synchronization constraints that can not be expressed in Delta Prolog.

Additional parallelism may be achieved in DLP by using the primitives for process creation and resumption requests directly. These constructs allow to join the results of two independently running processes sharing logical variables. We have not encountered any such mechanism in the literature! We remark that these primitives have been used to implement the synchronous rendez-vous arising from a method call.

To enable active objects to engage in a rendez-vous we have provided an accept statement for interrupting the own activity and to state the willingness to answer particular method calls. This construct has been inspired by languages combining the object oriented programming paradigm with concurrency.

A notable difference between DLP on the one hand and Delta Prolog and the two object oriented languages based on the concurrent logic programming model on the other hand is that states of objects in DLP are kept in non-logical (instance) variables whereas in the three other approaches states are maintained as the argument of a tail-recursive predicate. We note that the conditional accept statement introduced in section 2.5 allows a similar implementation of objects.

5.2.3 OO + ||

A radically different approach at combining logic programming, object oriented programming and concurrency is to take an existent parallel object oriented language as a starting point and to attempt to lift it to a language supporting logic based computation. Such an approach is exemplified by the language MultiLog. As another example we wish to mention the language OrientK/85.

MultiLog is a multi-tasking, object oriented Prolog. [Karam, 1988]. It is intended to be used for prototyping concurrent, embedded systems. MultiLog supports both passive and active objects, instance variables, methods, classes and inheritance.

Active objects are large grain sequential processes, that may communicate with each other by a rendez-vous like method call. To engage in such a rendez-vous an active object must interrupt its activity by an Ada-like accept statement.

Once a rendez-vous is successfully completed, that is when an answer has been delivered, all contact with the object to which the call was addressed is broken off. When the invoking process backtracks, no 'hidden communication' takes place to generate alternative solutions. Instead, the logical variables that have been bound in evaluating the method call simply become unbound.[4] MultiLog, however, does support local backtracking, that may be needed to select the appropriate clauses for answering a method call. This design decision, for not supporting global or distributed backtracking, is motivated by the intended use of the language for prototyping embedded systems: an equivalent mechanism does not exist in target languages such as Ada!

Currently, research is being done to improve on the efficiency of the MultiLog system, and to provide a suitable user interface.

Orient84/K combines the paradigms of object oriented, logic based, demon oriented and concurrent programming. [Ishikawa and Tokoro, 1986].

Concurrency is achieved by having active objects executing their own behavior. Apart from a *behavioral* part, each object contains a *knowledge base* part consisting of clauses, and a *monitor* part specifying the synchronization conditions of the object.

OrientK/84 supports instance variables, classes, inheritance, prioritized execution of objects, prioritized answering of method calls and *trigger* predicates, that become activated when certain conditions are met.

In [Ishikawa and Tokoro, 1986] considerable attention is paid to the design of a virtual machine for efficiently executing concurrent objects. The issues considered include the management of concurrently and sequentially executable objects, and the compilation of Orient84/K programs into an efficient intermediate code.

[4] In [Davison, 1989] MultiLog is incorrectly classified among the languages that support backtracking over the results of a communication.

Both MultiLog and OrientK/84 support single-threaded objects only. One of the possible advantages of lifting a parallel object oriented language may be that the mechanisms for process creation and communication are to a certain extent available. However, implementing a Prolog interpreter is not altogether a trivial matter.

DLP is, apart from a number of notational differences, quite alike MultiLog. A notable exception to this similarity however is that, unlike MultiLog, DLP does support global backtracking over the results of a rendez-vous.

In a sense our language DLP may be regarded as the result of lifting the language POOL to a logic programming language. The problem we had to solve was to find the proper constructs for process creation and communication between processes. As a consequence, our initial design goal has been *the extension of Prolog to a parallel object oriented language.* Unlike MultiLog and the object oriented languages based on concurrent logic programming, we did not wish to give up compatibility with Prolog. DLP therefore supports the *don't know* non-determinism of backtracking.

5.3 Who needs distributed logic programming?

We have boldly stated that DLP promises to be a suitable vehicle for implementing a variety of distributed knowledge based systems. Our argument in supporting this claim, however, merely consisted in showing that our language is sufficiently expressive for implementing the examples figuring in the literature as somehow important.

A different approach to supporting this claim would consist of trying to establish the usefulness of DLP for (Distributed) AI by enumerating the problems and requirements of this field of research and check whether DLP satisfies these needs. Instead of an exhaustive treatment, however, we will highlight one issue of crucial importance and provide some pointers to the literature.

Distributing control By far the most difficult problem to solve in designing a (truly) distributed system is not how to distribute the data but how to tackle the problem of distributing control. Apart from solving problems in a certain domain of expertise, the system must somehow decide how to decompose a complex task in subtasks. Also, in many cases, a final answer must be constructed from the pieces resulting from processing the subtasks. Several metaphors exist that suggest solutions to these problems, ranging from the image of a hierarchic (military-like) organization of processes to a heterarchical (market-like) configuration where processes buy or sell the needed capabilities. Cf. [Fox, 1981].

A very interesting solution to the effective mating of tasks and capabilities is provided in [Davis, 1980] where the active participation of both processes needing and processes offering a certain capability is described as *contract negotiation.* The idea is that a process that needs some task to be performed broadcasts a message, inviting processes to subscribe to the task and state their price. After a certain time then the process that initiated the subscription decides what offer, if any, is

the most suitable. Further negotiation may follow if the process does not accept the task after all, or if it needs some further information or capability to perform the task effectively. In the latter case the initiating process may send the required information or capability.

Our distributed logic programming language, as we have presented it, offers no facilities for broadcasting messages. It may be worthwhile to consider incorporating a restricted broadcast facility, for instance to send a message to all instances of a named object. How to deal with shared instantiations of logical variables however is not immediately obvious. Moreover, although a bit tedious, such a broadcast facility can be programmed in DLP as it is.

Sending information or clauses, on the other hand, does not cause any problem, since we have provided an *assert* statement, that enables to dynamically modify the functionality of an object.

Applications When asking the question what possible applications exist for which a distributed logic programming language as presented does provide a better tool than any of the other languages, at present we may only point at the medical expert system presented in section 4.3. Most of the other examples are implementable in at least one other language. For example competing implementations of the Dining Philosophers exist in Parlog and MultiLog. See [Ringwood, 1988] and [Karam, 1989]. We claim, however, that only our language is sufficiently expressive to provide a natural solution to all programming examples encountered.

Obviously, our language DLP may well be used for prototyping distributed systems, as well as for the implementation of knowledge based systems.

We can however also imagine the use of DLP as a high level specification language in a software engineering context, since the language combines declarative features with mechanisms for process creation and communication in a semantically sound way.

Part II

Semantics

Chapter 6

Process creation and communication in the presence of backtracking

- I shall offer you
the rich burnt fat
of a white goat.
Yes, I shall leave it behind for you. -
Sappho

A formal semantic analysis provides a touchstone by which to judge the validity of the constructs proposed in a language design. For our language DLP, or rather abstract versions thereof, we will provide both an operational semantics, characterizing the behavior of a program, and a denotational semantics, giving the meaning of a program in terms of a mapping to some domain of mathematical objects. We use the phrase *comparative semantics* for our effort to prove these characterizations equivalent.

Before studying the semantics of our language DLP in any detail, however, we wish to provide the reader with an introduction to the techniques used in our semantic description. In particular we wish to treat in this chapter the mathematical background to (metric) denotational semantics, in order to be able to concentrate in following chapters on issues concerning the language rather than on the tools used in the analysis. Our treatment, here and in subsequent chapters, owes in a considerable extent to the foundational work presented in [America and Rutten, 1989], notably [America and Rutten, 1989a] and [Kok and Rutten, 1988]; as well as the work on analyzing flow of control in logic programming languages reported

in [de Bakker, 1988]. For any details omitted in our sketch of it, we refer the reader to the various sources.

Another, equally important, reason for including this introductory chapter is to give the reader some feeling for the way we tackle the problems in providing a semantics for the distributed logic programming language DLP, problems caused by the complex interaction between communication and backtracking. To this end we define three abstract languages $\mathcal{B}_0$, $\mathcal{B}_1$ and $\mathcal{B}_2$ that illustrate the creation of processes, communication and backtracking that may occur in DLP. These languages are similar to the languages $\mathcal{L}_0$, $\mathcal{L}_1$ and $\mathcal{L}_2$ used in chapter 8 for giving the comparative semantics of DLP, except that they abstract from all details having to do with the logic programming aspects of DLP, such as the renaming of logical variables and unification.[1] What we are primarily interested in here is an analysis of the flow of control and the communication behavior displayed by programs in DLP-like languages, omitting as many details as possible.

We will first sketch the mathematical background of our semantic enterprise, that is, we will define metric spaces, and we will characterize the kind of mathematical domains we use in defining our semantic models. Next, after having briefly described the structure of the equivalence proofs that we use to relate an operational and a denotational characterization, we will provide the semantics for the language $\mathcal{A}$, a simple language with choice, illustrating the techniques introduced.

To meet our second goal, of giving an introductory account of the problems encountered in a distributed logic programming language we define the languages

$\mathcal{B}_0$ - a simple language with backtracking,

$\mathcal{B}_1$ - the language $\mathcal{B}_0$ extended with dynamic object creation and communication with local backtracking, and

$\mathcal{B}_2$ - the language $\mathcal{B}_0$ extended with dynamic object creation and communication with global backtracking.

For these languages we give both an operational and a denotational semantics, and relate these by proving their equivalence. We encourage those interested only in the operational semantics of DLP to jump to chapter 7.

6.1 Mathematical preliminaries

We will model the behavior of our language(s), denotationally, on a so-called process domain. Process domains may be given by reflexive equations of the form $\mathbb{P} \cong \mathcal{F}(\mathbb{P})$, stating the equivalence between $\mathbb{P}$ and a composite domain derived from $\mathbb{P}$. The framework of complete metric spaces has proved very useful for solving this kind of equations. The technique for solving reflexive domain equations introduced in [de Bakker and Zucker, 1982] has been generalized in [America and Rutten, 1989a] in order to cope with the need to solve equations

[1] An overview of DLP and the variants introduced for studying the semantics aspects of the language is given in chapter 3.

of the form $\mathbb{IP} \cong \mathbb{IP} \rightarrow \mathbb{IP}$ that arose in developing a semantics for POOL. Cf. [America et al, 1989].

6.1.1 Metric spaces

A *metric space* is a pair (M, d) with M a non-empty set and d a mapping $d : M \times M \rightarrow [0, 1]$ that assigns to each two elements from M a distance in the range $[0, 1]$. The *metric* d must satisfy for arbitrary elements $x, y \in M$ the properties that (a) $d(x, y) = 0 \Longleftrightarrow x = y$, (b) $d(x, y) = d(y, x)$, and for each $z \in M$, (c) $d(x, y) \leq d(x, z) + d(y, z)$. When instead of (c) d satisfies (c') $d(x, y) \leq max\{d(x, z), d(y, z)\}$ then d is called an *ultra-metric*.

As an example, for A an arbitrary set, the so-called discrete metric on A is defined by $d(x, y) = 0$ if $x = y$ and 1 otherwise. As a second example, for A an alphabet and $A^\infty = A^\star \cup A^\omega$, the set of finite and infinite words over A, we may put $d(x, y) = 2^{-sup\{n \mid x(n)=y(n)\}}$, where $x(n)$ denotes the prefix of length n in case the length of x exceeds n and x otherwise. By convention $2^{-\infty} = 0$. Now (A^∞, d) is an ultra-metric space.

We are interested primarily in *complete* metric spaces. Let (M, d) be a metric space, and let $(x_i)_i$ be a sequence in M, then $(x_i)_i$ is a Cauchy sequence whenever for every $\varepsilon > 0$ there is a $N \in \mathbb{N}$ such that $\forall n, m > N.d(x_n, x_m) < \varepsilon$. In other words past a certain point the distance between two elements in the sequence is smaller than this arbitrary ε. We call an element $x \in M$ the limit of a sequence $(x_i)_i$ whenever for arbitrary ε we can find an $N \in \mathbb{N}$ such that $\forall n > N.d(x_n, x) < \varepsilon$. We call such a sequence convergent and write $lim_{i\rightarrow\infty} x_i = x$. A metric space (M, d) is called complete whenever each Cauchy sequence converges to an element of M.

The importance of complete metric spaces derives from the possibility to characterize unique fixed points of functions within such a space. Let (M_1, d_1), (M_2, d_2) be metric spaces. For a function $f : M_1 \rightarrow M_2$, we call f continuous whenever we have that $lim_i f(x_i) = f(x)$ for an arbitrary sequence $(x_i)_i$ in M_1 with limit x. We write $f : M_1 \rightarrow^\varepsilon M_2$, for arbitrary constant ε whenever, for all $x, y \in M_1$ it holds that $d_2(f(x), f(y)) \leq \varepsilon \cdot d_1(x, y)$. We call f *non distance increasing (ndi)* when $0 \leq \varepsilon \leq 1$. The function f is called *contracting* when $0 \leq \varepsilon < 1$. We will write $f : M_1 \rightarrow^1 M_2$ for f *ndi*. Note that each contracting function is *ndi*.

Theorem 6.1.1

a. Let (M_1, d_1) and (M_2, d_2) be metric spaces, then each ndi $f : M_1 \rightarrow^1 M_2$ is continuous.

b. [Banach] *For (M, d) a complete metric space, each contracting $f \in M \rightarrow M$ has a unique fixed point which equals $lim\ f^i(x)$ for arbitrary $x \in M$, where by definition $f^0(x) = x$ and $f^{i+1}(x) = f(f^i(x))$.*

Having characterized complete metric spaces, we provide the means to construct complete metric spaces from given complete metric spaces. Let (M_i, d_i) be complete metric spaces, for $i = 1, ..., n$. We may construct the (complete) function space $(M_1 \rightarrow M_2, d_{M_1 \rightarrow M_2})$ by defining

$$d_{M_1 \to M_2}(f_1, f_2) = sup_{x \in M_1}\{d_2(f_1(x), f_2(x))\},$$

that is as the supremum of the distances of the function values in M_2. The (complete) union space $(M_1 \cup ... \cup M_n, d_{M_1 \cup ... \cup M_n})$ is obtained by defining

$$d_{M_1 \cup ... \cup M_n}(x, y) = d_i(x, y) \text{ if } x, y \in M_i$$

and 1 otherwise, assuming that $M_1, ..., M_n$ are disjoint. Similarly, we obtain the (complete) product space $(M_1 \times .. \times M_n, d_{M_1 \times ... \times M_n})$ by letting

$$d_{M_1 \times ... \times M_n}((x_1, ..., x_n), (y_1, ..., y_n)) = max_i\{d_i(x_i, y_i)\}.$$

For (M, d) a metric space, we call a subset X of M closed whenever each converging sequence with elements in X has its limit in X. Let $\mathcal{P}_{nc}(M) = \{X \mid X \subset M, X$ non-empty and closed $\}$. From (M, d) a complete metric space, we may construct the (complete) power space $(\mathcal{P}_{nc}(M), d_{\mathcal{P}_{nc}(M)})$ by defining

$$d_{\mathcal{P}_{nc}(M)}(X, Y) = max\{sup_{x \in X}\{d(x, Y)\}, sup_{y \in Y}\{d(y, X)\}\},$$

where $d(x, Y) = inf_{y \in Y}\{d(x, y)\}$. The distance $d_{\mathcal{P}_{nc}(M)}$ is usually called the Hausdorff distance and is written as d_H. By convention $sup\emptyset = 0$ and $inf\emptyset = 1$. A similar construction holds for compact power sets, defined by $\mathcal{P}_{co}(M) = \{X \mid X \subset M, X$ compact $\}$. Finally, for (M, d) a metric space, we may define the complete metric space $id_\varepsilon(M, d)$, for $0 < \varepsilon < 1$, as (M, d') where $d'(x, y) = \varepsilon \cdot d(x, y)$ for $x, y \in M$. In all the cases above we may uniformly replace *metric* by *ultra-metric.*

6.1.2 Domains

The mathematical domains for our denotational semantics are complete metric spaces satisfying a reflexive domain equation of the form $\mathbb{P} \cong \mathcal{F}(\mathbb{P})$, where $\mathcal{F}(\mathbb{P})$ is an expression composed of some given fixed spaces by applying one or more of the constructions for obtaining complete metric spaces. In [America and Rutten, 1989a] it is shown that such an equation is solvable if $\mathcal{F}$, taken as a functor on the category of metric spaces, is contracting in the sense that the elements of the image of the functor, including (category theoretically speaking) both objects and arrows, are in some sense nearer to each other than the elements from which they were derived. They also show that the completion procedure described in [de Bakker and Zucker, 1982] can be generalized to a direct limit construction for the sequence obtained by applying $\mathcal{F}$ to an arbitrary one point space. The requirement that $\mathcal{F}$ is contracting, as a functor on complete metric spaces, is essential for the existence of such a limit.

We will illustrate our notion of domains by giving some examples. For A an arbitrary set, with typical elements a, the domain $\mathbb{P}_1$, defined by

- $\mathbb{P}_1 \cong A \cup A \times \mathbb{P}_1$

consists, intuitively speaking, of all sequences over A. As examples of elements from $\mathbb{P}_1$ we have a, $< a_1, a_2 >$ and $< a_1, < a_2, ... >>$. For notational convenience we use an infix pairing operator $\cdot$ to be able to write pairs of the form $< a, \rho >$,

with $\rho \in \mathbb{P}_1$, as $a \cdot \rho$. Our examples above may now be written as a, $a_1 \cdot a_2$, $a_1 \cdot (a_2 \cdot ...)$.

We may use elements of $\mathbb{P}_1$ to describe sequences of action labels, with each label denoting an action by which a computation goes from one state to another. From this perspective, the expression $< a_1, < a_2, ... >>$ corresponds to the (deterministic) computation

$$\cdot \xrightarrow{a_1} \cdot \xrightarrow{a_2} ...$$

The domain $\mathbb{P}_1$ is isomorphic to A^∞, the set of strings over A, defined by $A^\infty = A^\star \cup A^\omega$. To prove this we may define a concatenation operator $\circ : A^\infty \times A^\infty \to A^\infty$ for strings w over A, by the equations $w_1 \circ w_2 = w_1 w_2$ if w_1 is of finite length and w_1 otherwise. We will denote the empty string by ε and state $\varepsilon w = w = w\varepsilon$.

To solve the equation for $\mathbb{P}_1$ we must, technically speaking, write

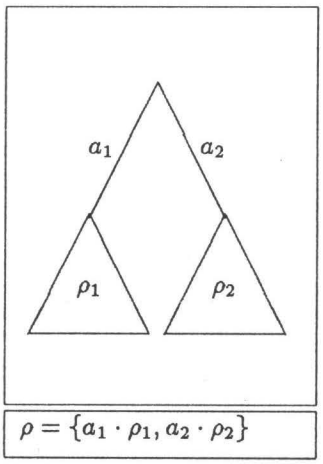
$$\mathbb{P}_1 \cong A \cup A \times id_{\frac{1}{2}}(\mathbb{P}_1)$$

and construct the corresponding metric, which can then be proven equal to the metric on A^∞. The functor implicitly defined by the right hand side of the isometry equation above is contracting, which is (intuitively) a consequence of the fact that the occurrence of $\mathbb{P}_1$ is preceded by a factor $id_{\frac{1}{2}}$. In the sequel we will leave this factor out.

An example of a domain that allows to model non-deterministic computations is the domain $\mathbb{P}_2$, with typical elements ρ, defined by

- $\mathbb{P}_2 = \mathcal{P}_{co}(A \cup A \times \mathbb{P}_2)$

The domain $\mathbb{P}_2$ exemplifies a branching structure, that may be regarded as consisting of rooted trees, of which the branches are labeled with elements from an alphabet A. As examples of elements of $\mathbb{P}_2$ we have $\{a\}$, $\{a, a \cdot \{a_1, a_2\}\}$, and $\{a_1 \cdot \rho_1, a_2 \cdot \rho_2\}$, the latter of which may be depicted as



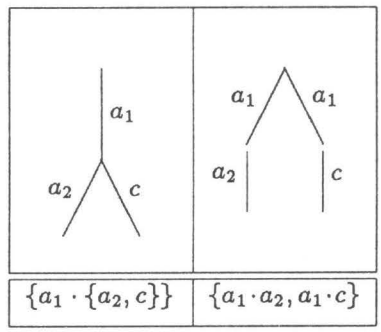
A tree as the one above models non-deterministic behavior in that either a_1 may be chosen after which ρ_1 follows, or a_2 followed by ρ_2. These trees satisfy the additional property of being commutative, in the sense that the order of the elements

in the set representing a process is unimportant, and absorptive, in the sense that multiple occurrences of an item in such a set are collapsed. The condition requiring $\mathbb{P}_2$ to contain only compact sets corresponds, intuitively, with the requirement that the tree is finitely branching, or in other words that the non-determinism in the computation is bounded.

Our motivation to use a branching structure, instead of a domain consisting of sets of strings over an alphabet, may be elucidated by means of our next example domain given by

- $\mathbb{P}_3 = \mathcal{P}_{co}(A \cup C \cup A \times \mathbb{P}_3 \cup C \times \mathbb{P}_3)$

where, in addition to A we have a set C, with elements c, consisting of what we like to call communication intentions. The branching nature of our domain allows us to distinguish between the processes $\{a_1 \cdot \{a_2, c\}\}$ and $\{a_1 \cdot \{a_2\}, a_1 \cdot \{c\}\}$, as depicted below.



Now, if we have a choice between an action a and a failure due to an unresolved communication intention c, obviously we would like to choose for the action, and forget about the communication. If however we have no other choice then we must admit that we have reached a dead-end. Naively, using a function $trace : \mathbb{P}_3 \rightarrow \mathcal{P}((A \cup C)^\infty)$, the sets of traces resulting from these processes would be

$$trace(\{a_1 \cdot \{a_2, c\}\}) = \{a_1 a_2, a_1 c\} = trace(\{a_1 \cdot \{a_2\}, a_1 \cdot \{c\}\});$$

however, putting the considerations above in effect we obtain a function $trace^\star$ that yields the traces not ending in an unresolved communication intention.

$$trace^\star(\{a_1 \cdot \{a_2, c\}\}) = \{a_1 a_2\} \neq \{a_1 a_2, a_1 c\} = trace^\star(\{a_1 \cdot \{a_2\}, a_1 \cdot \{c\}\}),$$

We cannot make such a distinction in a structure, such as $\mathcal{P}_{nc}((A \cup C)^\infty)$, which is a linear structure. See also [de Bakker et al, 1984].

What we have treated thus far are domains for modeling so-called *uniform* languages, i.e. languages that may be modeled without the use of a state parameter. In contrast, non-uniform languages need a state for modeling the effect of actions, as for example assignments in an imperative language. The next two example domains correspond with respectively $\mathbb{P}_1$ and $\mathbb{P}_3$, except that they depend on elements of a set Σ.

- $\mathbb{P}_4 = \Sigma \to (A \cup A \times \mathbb{P}_4)$
- $\mathbb{P}_5 = \Sigma \to \mathcal{P}_{co}(A \cup C \cup A \times \mathbb{P}_5 \cup C \times \mathbb{P}_5)$

In the present chapter we will not use these domains, we will do so however when modeling the languages $\mathcal{L}_0$, $\mathcal{L}_1$ and $\mathcal{L}_2$ that also cover the (logic) programming aspects of DLP.

We may remark that we do not have the need to use an equation of the form

$$\mathbb{P} \cong \mathbb{P} \to \mathbb{P}$$

for which the generalization in [America and Rutten, 1989a] was intended. However, in giving the semantics of global (distributed) backtracking, we have avoided the use of it at the expense of a quite intricate operator, as we will see in section 6.5.2.

6.1.3 Using contractions in proving the equivalence of semantic models

The use of contractions to characterize a variety of models has been advocated in [Kok and Rutten, 1988], and has proven its usefulness in proving the equality of operational and denotational semantics. In short, when we succeed in proving that both the operational semantics and the denotational semantics are a fixed point of a higher order mapping, then by the uniqueness of fixed points as stated in theorem 6.1.1 they are equal.

In somewhat more detail, when we have, say, a language $\mathcal{L}$ we may specify a (labeled) transition system describing the possible computation steps of a program in $\mathcal{L}$. With $\mathbb{R}$ a set containing the possible results, for instance sets of strings of action labels, we may define an operational semantics $\mathcal{O}[\![\cdot]\!] : \mathcal{L} \to \mathbb{R}$ as the fixed point of a higher order mapping $\Psi : (\mathcal{L} \to \mathbb{R}) \to (\mathcal{L} \to \mathbb{R})$, collecting all possible computation sequences. Next we define $\mathcal{D}[\![\cdot]\!] : \mathcal{L} \to \mathbb{P}$ for a domain $\mathbb{P}$. Now taking a projection $\pi : \mathbb{P} \to \mathbb{R}$, mapping elements from $\mathbb{P}$ to $\mathbb{R}$, we must show that $\pi \circ \mathcal{D}$, the composition of π and $\mathcal{D}[\![\cdot]\!]$ is a fixed point of the mapping Ψ, and by theorem 6.1.1 we are done.

We have used this technique for structuring our equivalence proofs, following [de Bakker, 1988].

6.2 A simple language with choice

In order to illustrate the technical notions introduced in the previous section we will treat a simple language $\mathcal{A}$ that, basically, may be used to execute actions from a set of actions A. Statements in the language $\mathcal{A}$ are goals, either to execute an action, to evaluate a recursively defined goal, or to evaluate a compound goal. We speak of goals here for conformity with the languages $\mathcal{B}_0$, $\mathcal{B}_1$ and $\mathcal{B}_2$. We may note that $\mathcal{A}$ is a context-free language.

Syntax We assume to have the set A of actions, mentioned above, with typical elements a, and a set Pvar of procedure variables, with typical elements p.

We may then define *goals* $g \in$ Goal by

$$g ::= a \mid p \mid g_1; g_2 \mid g_1 + g_2$$

A goal of the form $g_1; g_2$ stands for sequential composition and a goal of the form $g_1 + g_2$ represents a choice between goal g_1 and g_2.

We define *procedure declarations* as being of the form $p \leftarrow g$ for $p \in$ Pvar and $g \in$ Goal. We call g the *body* of the procedure p.

We let *declarations* D take the form

$$D = \{p \leftarrow g \mid p \in \mathsf{Pvar}, g \in \mathsf{Goal}\}$$

A *program* in $\mathcal{A}$ is a tuple $< D \mid g >$, with D a declaration and g a goal.

6.2.1 Operational semantics

We will define the operational semantics of the language $\mathcal{A}$ in terms of the behavior that may be observed when executing a program. We define a set of labels Λ, with typical elements η, such that

$$\Lambda = A \cup \{\star\}$$

contains the actions $a \in A$ and the special label $\star$. The label $\star$ will be used to denote behavior that is visible, but silent in the sense that it cannot further be inspected. The empty label, denoted by the empty word ε, will be used to denote invisible behavior and consequently will disappear in the operational semantics. We take $\Lambda^\infty = \Lambda^\star \cup \Lambda^\omega$ as the set of finite and infinite strings over Λ. The concatenation operator $\circ : \Lambda^\infty \times \Lambda^\infty \to \Lambda^\infty$ is defined as the usual string concatenation for strings in Λ^∞, which satisfies $\varepsilon \circ w = w = w \circ \varepsilon$. We moreover define

$$\mathbb{R} = \mathcal{P}_{nc}(\Lambda^\infty)$$

and extend the concatenation operator for strings in Λ^∞ by defining $w \circ X = \{w \circ x : x \in X\}$.

We will specify the computation steps that may occur when executing a program by means of a Plotkin-style transition system, defined by a number of transition rules. Our transition rules are axioms of the form $\Gamma \xrightarrow{\eta} \Gamma'$, stating that the configuration Γ may be taken to Γ' while displaying η. The label η may be empty, denoting the fact that the transition to which the rule gives rise is unobservable. We will call transitions with the empty label also unlabeled.

Configurations Γ represent, in a rather straightforward way, the part of the goal that still must be evaluated. In order to deal with termination we define syntactic continuations $S \in$ SynCo by

$$S ::= \surd \mid g; S$$

an let a configuration Γ be an element of Conf = SynCo.

The transition rules below specify a transition system $(\mathsf{Conf}, \Lambda, \longrightarrow)$ with $\longrightarrow \subset \mathsf{Conf} \times \Lambda \times \mathsf{Conf}$, relative to some declaration D.

$\mathcal{A}$

$a; S \xrightarrow{a} S$	*Action*
$p; S \xrightarrow{\star} g; S$ for $p \leftarrow g$ in D	*Rec*
$(g_1; g_2); S \longrightarrow g_1; (g_2; S)$	*Seq*
$(g_1 + g_2); S \longrightarrow g_1; S$ $\longrightarrow g_2; S$	*Choice*

As an explanation, the axiom for *Action* states that a configuration that has an action a as its first goal may be taken to that configuration without the action, while displaying the label a. The axiom *Rec* states that a call for p is replaced by the body for p, while displaying the special label $\star$. Applying the axiom *Seq* enables a further analysis of the first goal, simply by changing the brackets. The axiom *Choice* allows to continue with either one of the goals g_1 or g_2. Both the axioms *Seq* and *Choice* give rise to unlabeled transitions.

As an example, starting with the goal $a_1; a_2$ we have the computation

$$(a_1; a_2); \surd \longrightarrow a_1; (a_2; \surd) \xrightarrow{a_1} a_2; \surd \xrightarrow{a_2} \surd \qquad \text{(a)}$$

As another example, a possible computation resulting from the goal $(a_1; a_2) + (a_1; a_3)$ is

$$((a_1; a_2) + (a_1; a_3)); \surd \longrightarrow (a_1; a_3); \surd \longrightarrow a_1; (a_3; \surd) \xrightarrow{a_1} a_3; \surd \xrightarrow{a_3} \surd \qquad \text{(b)}$$

As the observable behavior of the goal evaluated in (a) we take the set $\{a_1 a_2\}$, and for the one evaluated in (b) we take $\{a_1 a_2, a_1 a_3\}$. We formalize our notion of observability below.

Computation sequences Since we are interested only in the labeled transitions, which represent steps in which non-trivial computation occurs, we must in some way abstract from the unlabeled transitions. We define two configurations Γ and Γ' to be related by $\longrightarrow\!\!\!\!\!\rightarrow$, notation $\Gamma \longrightarrow\!\!\!\!\!\rightarrow \Gamma'$, if there is a (possibly empty) sequence of unlabeled transitions from Γ to Γ'. Next we define $\Gamma \xrightarrow{\eta}\!\!\!\!\!\rightarrow \Gamma'$ to hold if there is a configuration Γ'' for which $\Gamma \longrightarrow\!\!\!\!\!\rightarrow \Gamma''$ and moreover $\Gamma'' \xrightarrow{\eta} \Gamma'$ for a non-empty label η. In other words $\Gamma \xrightarrow{\eta}\!\!\!\!\!\rightarrow \Gamma'$ whenever there is a transition with label η preceded by zero or more unlabeled transitions. We will say that Γ *blocks* if there is no label η and configuration Γ' for which $\Gamma \xrightarrow{\eta}\!\!\!\!\!\rightarrow \Gamma'$. An alternative way to deal with empty transitions would be to define transition rules with non-empty premisses, allowing to ignore the empty labels. We prefer to use axioms, however, for their conciseness.

Now we can define the *operational semantics* of a program $< D \mid g > \in \mathcal{A}$ as a mapping $\mathcal{O}[\![\cdot]\!] : \mathcal{A} \rightarrow \mathbb{R}$.

Definition 6.2.1 $\mathcal{O}[\![< D \mid g >]\!] = \mathcal{T}[\![g; \surd]\!]$ *where* $\mathcal{T}$ *is given by*

$$\mathcal{T}[\![\Gamma]\!] = \begin{cases} \{\varepsilon\} & \text{if } \Gamma \text{ blocks} \\ \bigcup\{\eta \circ \mathcal{T}[\![\Gamma']\!] : \Gamma \xrightarrow{\eta}\!\!\!\!\rightarrow\ \Gamma'\} & \text{otherwise} \end{cases}$$

Above we have defined $\mathcal{T}$ by a reflexive equation. As it stands it is not clear whether $\mathcal{T}$ exists. Before showing that, we will illustrate our approach by defining the function computing the factorial. The function $fac : \mathbb{N} \rightarrow \mathbb{N}$ may be written as

$$fac(n) = \quad \textit{if } n = 0 \textit{ then } 1 \textit{ else } n \star fac(n-1)$$

The proof that *fac* is well-defined proceeds by induction on n. An alternative way to characterize the function *fac* is to define a higher order mapping $\Phi : (\mathbb{N} \rightarrow \mathbb{N}) \rightarrow (\mathbb{N} \rightarrow \mathbb{N})$, taking functions over the natural numbers to functions over the natural numbers, by

$$\Phi(\phi) = \lambda n.\textit{if } n = 0 \textit{ then } 1 \textit{ else } n \star \phi(n-1)$$

and state that $fac = \mathsf{fix}\Phi$. We may obtain the fixed point of Φ by iterating the application of Φ to an arbitrary function f an indefinite number of times. As an example $\Phi^2(f)$, the two-fold application of Φ to f looks as

$$\lambda n.\textit{if } n = 0 \textit{ then } 1 \textit{ else } n \star (\lambda n'.\textit{if } n' = 0 \textit{ then } 1 \textit{ else } n' \star f(n'-1))(n-1)$$

We may check that $\Phi(fac) = fac$ by observing that $\Phi(fac)(n)$ equals $\textit{if } n = 0 \textit{ then } 1 \textit{ else } n \star fac(n-1)$ which clearly equals $fac(n)$.

In order to show that $\mathcal{T}$ exists we define a higher order mapping $\Psi : (\mathsf{Conf} \rightarrow \mathbb{R}) \rightarrow (\mathsf{Conf} \rightarrow \mathbb{R})$ by

$$\Psi(\phi)[\![\Gamma]\!] = \begin{cases} \{\varepsilon\} & \text{if } \Gamma \text{ blocks} \\ \bigcup\{\eta \circ \phi[\![\Gamma']\!] : \Gamma \xrightarrow{\eta}\!\!\!\!\rightarrow\ \Gamma'\} & \text{otherwise} \end{cases}$$

and define $\mathcal{T} = \mathsf{fix}\Psi$.

We state that Ψ is well-defined, in that it takes *ndi* functions to *ndi* functions. Also Ψ is contracting since, intuitively, with each application a label is added.[2] Hence Ψ has a fixed point, which moreover is unique due to theorem 6.1.1. As a remark, we must note that we have suppressed the dependency of $\mathcal{T}$, and Ψ, on the declaration D.

6.2.2 Denotational semantics

The distinguishing feature of a denotational semantics is its compositionality. A compositional semantics allows to give mathematical meaning to parts of a program, that may be glued together by semantic operators to yield the meaning of the composite program.

[2]Since recursive procedure call results in a label, we do not have to impose the requirement that bodies of procedures are guarded to prevent infinite sequences of invisible transitions to occur.

Instead of giving a direct denotational semantics we take a slightly devious route in assigning mathematical meaning to the language $\mathcal{A}$, by giving a continuation semantics instead.

We will use semantic continuations $R \in \mathsf{SemCo} = \mathbb{R}$. Semantic continuations are introduced here for the reader to become familiar with the kind of continuation semantics used for analyzing the languages with backtracking, not because they are more convenient for this particular case. The usefulness of continuation semantics for modeling a variety of (more or less) exotic language features has been eloquently defended in [de Bruin, 1986], to which we refer the reader for a more extensive account of this brand of denotational semantics.

Without further ado we define the function $\mathcal{D}[\![\cdot]\!] : \mathsf{Goal} \to \mathsf{SemCo} \to \mathbb{R}$.

$\mathcal{A}$

(i) $\mathcal{D}[\![a]\!]R = a \circ R$

(ii) $\mathcal{D}[\![p]\!]R = \star \circ \mathcal{D}[\![g]\!]R$ for $p \leftarrow g$ in D

(iii) $\mathcal{D}[\![g_1; g_2]\!]R = \mathcal{D}[\![g_1]\!](\mathcal{D}[\![g_2]\!]R)$

(iv) $\mathcal{D}[\![g_1 + g_2]\!]R = \mathcal{D}[\![g_1]\!]R \cup \mathcal{D}[\![g_2]\!]R$

The difference with a direct denotational semantics comes to light in the equation (iii) for sequential composition, where we put the result of evaluating g_2 in the continuation.

Those familiar with denotational semantics may have frowned upon the equations for recursive procedure call in $\mathcal{D}[\![\cdot]\!]$. We take the opportunity to remark that the metric setting developed thus far may also be profitably used in characterizing functions as $\mathcal{D}[\![\cdot]\!]$ as the fixed point of a higher order mapping $\Phi : (\mathsf{Goal} \to \mathsf{SemCo} \to \mathbb{R}) \to (\mathsf{Goal} \to \mathsf{SemCo} \to \mathbb{R})$.
We define Φ by

$$\Phi(\phi)[\![a]\!]R = a \circ R$$

$$\Phi(\phi)[\![p]\!]R = \star \circ \phi[\![g]\!]R \text{ for } p \leftarrow g \text{ in } D$$

$$\Phi(\phi)[\![g_1; g_2]\!]R = \Phi(\phi)[\![g_1]\!](\Phi(\phi)[\![g_2]\!]R)$$

$$\Phi(\phi)[\![g_1 + g_2]\!]R = \Phi(\phi)[\![g_1]\!]R \cup \Phi(\phi)[\![g_2]\!]R$$

and must now prove that Φ is contracting and well-defined in the sense that it maps *ndi* functions to *ndi* functions. Although by no means necessary, this approach is convenient since we no longer have the need for an environment parameter, as traditionally encountered in denotational semantics, to cope with recursion.

Definition 6.2.2 *For a program* $< D \mid g >$ *the meaning* $\mathcal{M}[\![< D \mid g >]\!] = \mathcal{D}[\![g]\!]\{\varepsilon\}$.

We give two examples.

$$\mathcal{M}[\![a_1; a_2]\!] = \mathcal{D}[\![a_1; a_2]\!]\{\varepsilon\} = \mathcal{D}[\![a_1]\!](\mathcal{D}[\![a_2]\!]\{\varepsilon\}) = a_1 \circ (a_2 \circ \{\varepsilon\}) = \{a_1 a_2\}.$$

$$\mathcal{M}[\![(a_1; a_2) + (a_1; a_3)]\!] = \mathcal{D}[\![(a_1; a_2) + (a_1; a_3)]\!]\{\varepsilon\} =$$
$$\mathcal{D}[\![a_1; a_2]\!]\{\varepsilon\} \cup \mathcal{D}[\![a_1; a_3]\!]\{\varepsilon\} = \{a_1 a_2, a_1 a_3\}.$$

6.2.3 Equivalence between operational and denotational semantics

We wish the observable behavior of a program $< D \mid g >$ assigned to it by the operational semantics $\mathcal{O}[\![\cdot]\!]$ to be adequately characterized by the mathematical meaning given by the function $\mathcal{M}[\![\cdot]\!]$.

Theorem 6.2.3 $\mathcal{O}[\![< D \mid g >]\!] = \mathcal{M}[\![< D \mid g >]\!]$

We will give an outline of the proof and promise to fill in the details later.

$$\mathcal{O}[\![< D \mid g >]\!] = \mathcal{T}[\![g; \surd]\!] \text{ by definition 6.2.1}$$
$$= \mathcal{R}[\![g; \surd]\!] \text{ by corollary 6.2.7}$$
$$= \mathcal{D}[\![g]\!]\{\varepsilon\} \text{ by definition 6.2.4}$$
$$= \mathcal{M}[\![< D \mid g >]\!] \text{ by definition 6.2.2}$$

In the proof we make use of an intermediate semantic function $\mathcal{R} : \mathsf{SynCo} \rightarrow \mathsf{SemCo}$ mapping syntactic continuations to semantic continuations.

Definition 6.2.4

a. $\mathcal{R}[\![\surd]\!] = \{\varepsilon\}$

b. $\mathcal{R}[\![g; S]\!] = \mathcal{D}[\![g]\!]\mathcal{R}[\![S]\!]$

We may state the following property.

Lemma 6.2.5 $\mathcal{R}[\![(g_1; g_2); S]\!] = \mathcal{R}[\![g_1; (g_2; S)]\!]$

Proof: The proof amounts to repeatedly applying the definition of $\mathcal{R}$ and D as shown.
We have that $\mathcal{R}[\![g_1; (g_2; S)]\!] = \mathcal{D}[\![g_1]\!]\mathcal{R}[\![g_2; S]\!] = \mathcal{D}[\![g_1]\!](\mathcal{D}[\![g_2]\!]\mathcal{R}[\![S]\!]) =$
$\mathcal{D}[\![g_1; g_2]\!]\mathcal{R}[\![S]\!] = \mathcal{R}[\![(g_1; g_2); S]\!]$ as desired. □

The key step in the proof that $\mathcal{O} = \mathcal{M}$ consists of showing that the function $\mathcal{R}$ is a fixed point of the operator Ψ that characterizes the operational meaning of a program. (See section 6.2.1)

Lemma 6.2.6 $\Psi(\mathcal{R}) = \mathcal{R}$

Proof: In order to be able to use induction we define a complexity measure c on configurations of the form $g; S$ by stating

$c(g; S) = c(g)$ with $c(a) = c(p) = 1$ and
$c(g_1; g_2) = c(g_1 + g_2) = c(g_1) + c(g_2) + 1$.

We must now prove for each Γ that $\Psi(\mathcal{R})[\![\Gamma]\!] = \mathcal{R}[\![\Gamma]\!]$. As induction hypothesis we assume that whenever $c(\Gamma) > c(\Gamma')$ then $\Psi(\mathcal{R})[\![\Gamma']\!] = \mathcal{R}[\![\Gamma']\!]$. When $\Gamma = \surd$ the result is immediate. For Γ of the form $g; S$ the proof proceeds by a case analysis on g.

- If $g \equiv a$ then $\Psi(\mathcal{R})[\![a; S]\!] =$ (by the definition of $\Psi(\mathcal{R})$)
  $a \circ \mathcal{R}[\![S]\!] =$ (by the definition of $\mathcal{D}$)
  $\mathcal{D}[\![a]\!]\mathcal{R}[\![S]\!] =$ (by definition 6.2.4)
  $\mathcal{R}[\![a; S]\!]$.

- If $g \equiv p$ then $\Psi(\mathcal{R})[\![p; S]\!] =$ (for $p \leftarrow g'$ in D)
  $\star \circ \mathcal{R}[\![g'; S]\!] =$ (by definition 6.2.4)
  $\star \circ \mathcal{D}[\![g']\!]\mathcal{R}[\![S]\!] =$ (by the definition of $\mathcal{D}$)
  $\mathcal{D}[\![p]\!]\mathcal{R}[\![S]\!] =$ (again by definition 6.2.4)
  $\mathcal{R}[\![p; S]\!]$.

- If $g \equiv g_1; g_2$ then $\Psi(\mathcal{R})[\![(g_1; g_2); S]\!] =$ (by the definition of $\Psi(\mathcal{R})$)
  $\Psi(\mathcal{R})[\![g_1; (g_2; S)]\!] =$ (by applying the induction hypothesis)
  $\mathcal{R}[\![g_1; (g_2; S)]\!] =$ (by lemma 6.2.5)
  $\mathcal{R}[\![(g_1; g_2); S]\!]$.

- For $g \equiv g_1 + g_2$ we have $\Psi(\mathcal{R})[\![(g_1 + g_2); S]\!] =$ (by the definition of $\Psi(\mathcal{R})$)
  $\Psi(\mathcal{R})[\![g_1; S]\!] \cup \Psi(\mathcal{R})[\![g_2; S]\!] =$ (by applying the induction hypothesis)
  $\mathcal{R}[\![g_1; S]\!] \cup \mathcal{R}[\![g_2; S]\!] =$ (by definition 6.2.4)
  $\mathcal{D}[\![g_1]\!]\mathcal{R}[\![S]\!] \cup \mathcal{D}[\![g_2]\!]\mathcal{R}[\![S]\!] =$ (by the definition of $\mathcal{D}$)
  $\mathcal{D}[\![g_1 + g_2]\!]\mathcal{R}[\![S]\!] =$ (again by definition 6.2.4)
  $\mathcal{R}[\![(g_1 + g_2); S]\!]$. □

Our corollary follows from the fact that the fixed point of Ψ is unique.

Corollary 6.2.7 $\mathcal{T} = \mathcal{R}$

The proof of the equivalence $\mathcal{O} = \mathcal{M}$ given above hinges on finding a semantic counterpart for the syntactic continuation employed in giving the operational semantics. In the sections that follow we will apply this proof technique to the semantics of considerably more complex languages.

6.3 A language with backtracking

In the previous section we have illustrated the major ingredients of our semantic approach by giving the comparative semantics for a very simple language. We will apply these techniques now to languages that become progressively more complex with respect to their backtracking behavior.

We will start with the language $\mathcal{B}_0$, a simple language with backtracking.[3] The language $\mathcal{B}_0$ differs from the language $\mathcal{A}$ in that we have replaced the choice construct $g_1 + g_2$ by a construct $g_1 \,\square\, g_2$, for alternative composition, that instead of choosing between g_1 and g_2 enforces to backtrack over the alternatives.

Although backtracking may be used for mimicking non-deterministic computation, the behavior of backtracking itself is deterministic in the sense that the order in which the evaluation of alternative goals takes place is completely determined. The determinate nature of backtracking is reflected in the fact that the behavior of a program in $\mathcal{B}_0$ may be characterized by a single string of action labels, instead of a set of such strings as for programs in $\mathcal{A}$.

Another difference with the language $\mathcal{A}$ is that actions in $\mathcal{B}_0$ may fail. Failure, as determined by an abstract interpretation function, gives rise to backtracking.

Syntax Again we assume to have actions $a \in A$, and procedure variables $p \in$ Pvar. As special actions we assume to have an action *skip* and an action *fail*.

We define *goals* $g \in$ Goal by

$$g ::= a \mid p \mid g_1; g_2 \mid g_1 \,\square\, g_2$$

and let *declarations* D take the form

$$D = \{p \leftarrow g \mid p \in \mathsf{Pvar}, g \in \mathsf{Goal}\}$$

A *program* is a tuple $< D \mid g >$, with D a declaration and g a goal.

6.3.1 Operational semantics

We will characterize the behavior of a program in $\mathcal{B}_0$ as a string over a set of labels Λ that we define by

$$\Lambda = A \cup \{\star\}$$

with $\star$ given the interpretation of a silent action. We also use ε for an unobservable action, as explained previously. Since the behavior of a program in $\mathcal{B}_0$ is determinate we may map its meaning to an element of $\Lambda^\infty = \Lambda^\star \cup \Lambda^\omega$.

As for $\mathcal{A}$ we use syntactic continuations in specifying the transition rules. In order to deal with backtracking, however, we must distinguish between syntactic success continuations and syntactic failure continuations, as explained below. A success continuation represents the part of the program that must be executed to find a solution. Success continuations $R \in$ SuccCo are alike to the syntactic continuations used for $\mathcal{A}$. We define

$$R ::= \surd \mid g; R$$

Success continuations R correspond to the sequential evaluation of a goal, without backtracking. They are used as in $g; \surd$, for an arbitrary goal g. When g is successfully evaluated then the empty success continuation $\surd$ is reached.

[3] Our treatment closely follows the treatment of a very similar language given in [de Bakker, 1988].

Evaluating a choice in the language $\mathcal{A}$ allowed us to forget about the alternative. Backtracking, however, requires to keep account of what alternatives remain to be evaluated. For this purpose we use so-called failure continuations $F \in \mathsf{FailCo}$ defined by

$$F ::= \Delta \mid R : F$$

The empty failure continuation Δ represents the situation that all possible solutions have been explored. Intuitively, a failure continuation $R : F$ is a stack having a success continuation R on top. If R for some reason fails the failure continuation F, the remainder of the stack, may be used to find a solution. The evaluation of a goal g is started by the failure continuation $g; \surd : \Delta$. When during the execution an alternative goal, say of the form $g_1 \,\square\, g_2$ is encountered, as represented by $(g_1 \,\square\, g_2); R : F$, then the resulting failure continuation is $(g_1; R) : (g_2; R) : F$, having the success continuation $g_1; R$ on top of the stack and $g_2; R$ as the first alternative to be explored.

As configurations we now take failure continuations and let Γ range over $\mathsf{Conf} = \mathsf{FailCo}$. We assume to have an abstract interpretation function $I : A \longrightarrow \{true, false\}$ defining for each action whether it is successful or fails. The function I must satisfy $I(skip) = true$ and $I(fail) = false$.

$\mathcal{B}_0$

$a; R : F \xrightarrow{a} R : F$ if $I(a)$ $\longrightarrow F$ otherwise	*Action*
$p; R : F \xrightarrow{\star} g; R : F$ for $p \leftarrow g$ in D	*Rec*
$(g_1; g_2); R : F \longrightarrow g_1; (g_2; R) : F$	*Seq*
$(g_1 \,\square\, g_2); R : F \longrightarrow (g_1; R) : (g_2; R) : F$	*Alt*
$\surd : F \longrightarrow F$	*Tick*

Notice that whenever an action fails, as expressed in the axiom for *Action*, the failure stack is popped in order to search for an alternative solution. Since we like to find all solutions, when a solution is reached as indicated by the occurrence of a $\surd$, the stack is also popped in order to evaluate the remaining alternatives.

As an example, consider the goals $a_1 \,\square\, a_2$ and $a_1; a_2$ that give rise to the computations

$$(a_1 \,\square\, a_2); \surd : \Delta \longrightarrow (a_1; \surd) : (a_2; \surd) : \Delta \xrightarrow{a_1} \surd : (a_2; \surd) : \Delta$$
$$\longrightarrow a_2; \surd : \Delta \xrightarrow{a_2} \surd : \Delta \longrightarrow \Delta$$

$$(a_1; a_2); \surd : \Delta \longrightarrow a_1; (a_2; \surd) : \Delta \xrightarrow{a_1} a_2; \surd : \Delta \xrightarrow{a_2} \surd : \Delta \longrightarrow \Delta$$

Both computations result in the same observable behavior. Consider however the goal $(a_1 \,\square\, a_2); a_3$ for which we have

$$((a_1 \,\square\, a_2); a_3); \surd : \Delta \longrightarrow (a_1 \,\square\, a_2); (a_3; \surd) : \Delta$$

$$\longrightarrow (a_1; (a_3; \surd)) : (a_2; (a_3; \surd)) : \Delta$$

$$\xrightarrow{a_1} (a_3; \surd) : (a_2; (a_3; \surd)) : \Delta \xrightarrow{a_3} \surd : (a_2; (a_3; \surd)) : \Delta$$

$$\longrightarrow a_2; (a_3; \surd) : \Delta \xrightarrow{a_2} a_3; \surd : \Delta \xrightarrow{a_3} \surd : \Delta \longrightarrow \Delta$$

The behavior of this computation is characterized by the string $a_1 a_3 a_2 a_3$ which is clearly different from the behavior resulting from the evaluation of $(a_1; a_2); a_3$.

Finally we give some examples of failing actions.

$$\mathit{fail}; \mathit{skip}; \surd : \Delta \longrightarrow \Delta$$

$$\mathit{skip}; \mathit{fail}; \surd : \Delta \xrightarrow{\mathit{skip}} \mathit{fail}; \surd : \Delta \longrightarrow \Delta$$

$$\mathit{fail} \,\square\, \mathit{skip}; \surd : \Delta \longrightarrow \mathit{fail}; \surd : \mathit{skip}; \surd : \Delta \longrightarrow \mathit{skip}; \surd : \Delta \xrightarrow{\mathit{skip}} \surd : \Delta \longrightarrow \Delta$$

$$\mathit{skip} \,\square\, \mathit{fail}; \surd : \Delta \longrightarrow \mathit{skip}; \surd : \mathit{fail}; \surd : \Delta \xrightarrow{\mathit{skip}} \surd : \mathit{fail}; \surd : \Delta$$
$$\longrightarrow \mathit{fail}; \surd : \Delta \longrightarrow \Delta$$

Using the relation $\xrightarrow{\eta}\!\!\!\!\rightarrow$ as given for $\mathcal{A}$ we may define the operational semantics of a program $< D \mid g >$ in $\mathcal{B}_0$ as a function $\mathcal{O}[\![\cdot]\!] : \mathcal{B}_0 \to \Lambda^\infty$, as below.

Definition 6.3.1 $\mathcal{O}[\![< D \mid g >]\!] = \mathcal{T}[\![g; \surd : \Delta]\!]$ *where for* $\Psi : (\mathsf{Conf} \to \Lambda^\infty) \to (\mathsf{Conf} \to \Lambda^\infty)$ *and*

$$\Psi(\phi)[\![\Gamma]\!] = \begin{cases} \varepsilon & \text{if } \Gamma \text{ blocks} \\ \eta \circ \phi[\![\Gamma']\!] \text{ if } \Gamma \xrightarrow{\eta}\!\!\!\!\rightarrow \Gamma' & \text{otherwise} \end{cases}$$

we let $\mathcal{T} = \mathsf{fix}\Psi$.

6.3.2 Denotational semantics

As the domain for our denotational semantics we take Λ^∞, and let ρ range over it.

With respect to the semantic continuations we must make a distinction, similar to the one made when specifying the transition rules, between semantic success continuations and semantic failure continuations. We define the set of semantic failure continuations to be Λ^∞ itself, by defining

$$\mathsf{Fail} = \Lambda^\infty$$

Semantic success continuations are defined as the function space

$$\mathsf{Succ} = \mathsf{Fail} \to \Lambda^\infty$$

We use typical elements $R \in \mathsf{Succ}$ and $F \in \mathsf{Fail}$, not to be confused with their syntactic counterparts. The idea of applying (semantic) success continuations to failure continuations to model backtracking behavior is due to [de Bruin, 1986].

Our denotational semantics $\mathcal{D}[\![\cdot]\!] : \mathsf{Goal} \to \mathsf{Succ} \to \mathsf{Fail} \to \Lambda^\infty$ is defined by the equations below.

$\mathcal{B}_0$

(i) $\mathcal{D}[\![a]\!]RF = I(a) \to a \circ RF, F$

(ii) $\mathcal{D}[\![p]\!]RF = \star \circ \mathcal{D}[\![g]\!]RF$ for $p \leftarrow g$ in D

(iii) $\mathcal{D}[\![g_1; g_2]\!]RF = \mathcal{D}[\![g_1]\!](\mathcal{D}[\![g_2]\!]R)F$

(iv) $\mathcal{D}[\![g_1 \,\square\, g_2]\!]RF = \mathcal{D}[\![g_1]\!]R(\mathcal{D}[\![g_2]\!]RF)$

As a comment, the expression $I(a) \to a \circ RF, F$ must be read as

if $I(a)$ *then* $a \circ (R(F))$ *else* F

where $R(F)$ denotes the application of R to F. Note that only in (i) R must be applied to F. In the other equations R and F are just taken along, the brackets are used there only for disambiguation.

The difference between success continuations and failure continuations comes to light in the equations for the goals $g_1; g_2$ and $g_1 \,\square\, g_2$. For the latter, the meaning corresponding to the second component is put in the failure continuation, while for the first it is put in the success continuation.

Definition 6.3.2 *Let* $R_0 = \lambda\rho.\rho$ *and* $F_0 = \varepsilon$ *then* $\mathcal{M}[\![< D \mid g >]\!] = \mathcal{D}[\![g]\!]R_0F_0$.

Again some examples. We assume that no action fails.

$$\mathcal{D}[\![a_1; a_2]\!]R_0F_0 = \mathcal{D}[\![a_1]\!](\mathcal{D}[\![a_2]\!]R_0)F_0 = a_1 \circ (a_2 \circ \varepsilon) = a_1a_2$$

$$\mathcal{D}[\![a_1 \,\square\, a_2]\!]R_0F_0 = \mathcal{D}[\![a_1]\!]R_0(\mathcal{D}[\![a_2]\!]R_0F_0) = a_1 \circ (a_2 \circ \varepsilon) = a_1a_2$$

We also have

$$\mathcal{D}[\![(a_1 \,\square\, a_2); a_3]\!]R_0F_0 = \mathcal{D}[\![a_1 \,\square\, a_2]\!](\mathcal{D}[\![a_3]\!]R_0)F_0 =$$

$$\mathcal{D}[\![a_1]\!](\mathcal{D}[\![a_3]\!]R_0)(\mathcal{D}[\![a_2]\!](\mathcal{D}[\![a_3]\!]R_0)F_0) = a_1 \circ (a_3 \circ (a_2 \circ (a_3 \circ \varepsilon)) = a_1a_3a_2a_3$$

6.3.3 Equivalence between operational and denotational semantics

We state our objective in giving the comparative semantics for $\mathcal{B}_0$ as a theorem.

Theorem 6.3.3 $\mathcal{O}[\![< D \mid g >]\!] = \mathcal{M}[\![< D \mid g >]\!]$

The proof has a structure similar to the one given for $\mathcal{A}$. (section 6.2.3)

$$\begin{aligned}\mathcal{O}[\![< D \mid g >]\!] &= \mathcal{T}[\![g; \surd : \Delta]\!] \text{ by definition 6.3.1}\\ &= \mathcal{F}[\![g; \surd : \Delta]\!] \text{ by corollary 6.3.7}\\ &= \mathcal{D}[\![g]\!]R_0F_0 \text{ by definition 6.3.4}\\ &= \mathcal{M}[\![< D \mid g >]\!] \text{ by definition 6.3.2}\end{aligned}$$

In the actual proof we will use the intermediate semantic functions

$\mathcal{R}$: SuccCo $\rightarrow$ Succ
$\mathcal{F}$: FailCo $\rightarrow$ Fail

mapping syntactic success and failure continuations to their semantic counterparts.

Definition 6.3.4

a. $\mathcal{R}[\![\surd]\!] = \lambda\rho.\rho$

b. $\mathcal{R}[\![g; R]\!] = \mathcal{D}[\![g]\!]\mathcal{R}[\![R]\!]$

c. $\mathcal{F}[\![\Delta]\!] = \varepsilon$

d. $\mathcal{F}[\![R : F]\!] = \mathcal{R}[\![R]\!]\mathcal{F}[\![F]\!]$

The following lemma states that unobservable transitions do no harm.

Lemma 6.3.5 *if* $\Gamma \longrightarrow \Gamma'$ *then* $\mathcal{F}[\![\Gamma]\!] = \mathcal{F}[\![\Gamma']\!]$

Proof: Consider first the case that $\Gamma = a; R : F$ and assume that $\Gamma \longrightarrow \Gamma'$, which occurs when $I(a)$ is *false*. Now $\mathcal{F}[\![a; R : F]\!] = \mathcal{R}[\![a; R]\!]\mathcal{F}[\![F]\!] = \mathcal{D}[\![a]\!]\mathcal{R}[\![R]\!]\mathcal{F}[\![F]\!] = \mathcal{F}[\![F]\!]$, simply by applying the definitions.

As a second case we give the proof for *Alt*. Let $\Gamma = g_1 \,\square\, g_2; R : F$.
Then $\mathcal{F}[\![(g_1 \,\square\, g_2); R : F]\!] = \mathcal{R}[\![(g_1 \,\square\, g_2); R]\!]\mathcal{F}[\![F]\!] = \mathcal{D}[\![g_1 \,\square\, g_2]\!]\mathcal{R}[\![R]\!]\mathcal{F}[\![F]\!] = \mathcal{D}[\![g_1]\!]\mathcal{R}[\![R]\!](\mathcal{D}[\![g_2]\!]\mathcal{R}[\![R]\!]\mathcal{F}[\![F]\!]) = \mathcal{R}[\![g_1; R]\!]\mathcal{F}[\![g_2; R : F]\!] = \mathcal{F}[\![g_1; R : g_2; R : F]\!]$. □

Again, the key step in the proof that $\mathcal{O} = \mathcal{M}$ consists of showing that the function $\mathcal{F}$ is a fixed point of the operator Ψ that characterizes the operational meaning of a program. (Section 6.3.1)

Lemma 6.3.6 $\Psi(\mathcal{F}) = \mathcal{F}$

We introduce a complexity measure c on $\Gamma \in$ FailCo such that whenever $\Gamma \longrightarrow \Gamma'$ we have that $c(\Gamma) > c(\Gamma')$, by defining

$c(\Delta) = 0$ and $c(g; R : F) = c(g) + c(F)$,
$c(a) = c(p) = c(\surd) = 1$, and
$c(g_1; g_2) = c(g_1 \,\square\, g_2) = c(g_1) + c(g_2) + 1$,

As an illustration

$$c((g_1 \,\square\, g_2); R : F) = c(g_1 \,\square\, g_2) + c(F) =$$
$$c(g_1) + c(g_2) + 1 + c(F) > c(g_1) + c(g_2) + c(F) = c(g_1; R : g_2; R : F).$$

We must prove for each Γ that $\Psi(\mathcal{F})[\![\Gamma]\!] = \mathcal{F}[\![\Gamma]\!]$. If Γ blocks then the result is clear. Assume that Γ is of the form $g; R : F$ and that for some η we have $\Gamma \xrightarrow{\eta}\!\!\!\!\rightarrow \Gamma'$. The proof proceeds by a case analysis on g.

- If $g \equiv a$ we have by the definition of Ψ if $I(a)$ is true that $\Psi(\mathcal{F})[\![a; R : F]\!] = a \circ \mathcal{F}[\![R; F]\!] =$ (by applying definition 6.3.4) $\mathcal{F}[\![a; R : F]\!]$. If $I(a)$ is not true then $a; R : F \longrightarrow F$ and the result follows from the induction-hypothesis, stating that for Γ' satisfying $c(\Gamma) > c(\Gamma')$ it holds that $\Psi(\mathcal{F})[\![\Gamma']\!] = \mathcal{F}[\![\Gamma']\!]$.
- If $g \equiv p$ then we have that $\Psi(\mathcal{F})[\![p; R : F]\!] = \star \circ \mathcal{F}[\![g; R : F]\!] =$ (by applying the definition of $\mathcal{D}$ and $\mathcal{F}$) $\star \circ \mathcal{D}[\![g]\!]\mathcal{R}[\![R]\!]\mathcal{F}[\![F]\!]$ and the result follows immediately.
- For $g \equiv g_1; g_2$ we have that $\Psi(\mathcal{F})[\![(g_1; g_2); R : F]\!] = \Psi(\mathcal{F})[\![g_1; (g_2; R) : F]\!]$ by the definition of Ψ and by induction this is equal to $\mathcal{F}[\![g_1; (g_2; R) : F]\!]$ which by lemma 6.3.4 equals $\mathcal{F}[\![(g_1; g_2); R : F]\!]$.
- The case $g \equiv g_1 \,\square\, g_2$ is similar. □

Corollary 6.3.7 $\mathcal{T} = \mathcal{F}$

6.4 Dynamic object creation and communication with local backtracking

In the sections that follow we extend the language $\mathcal{B}_0$ with primitives for dynamic object creation and communication. An object is a process that is evaluating a goal. A number of objects may be active concurrently. Our first extension $\mathcal{B}_1$ allows communication with local backtracking, local in the sense that the backtracking occurs within the confines of an object without affecting the behavior of other objects. To be able to treat $\mathcal{B}_1$ as a uniform language (c.f. section 6.1.2) we have delegated the task of inventing identifiers for newly created objects to the programmer, by providing him with a collection of names for objects.

Syntax As for $\mathcal{B}_0$ we have actions $a \in A$, and procedure variables $p \in \mathsf{Pvar}$. Moreover, we assume to have a set of object names O, for which we use typical elements $\hat{\alpha}$ and $\hat{\beta}$. Further we have a set C of communication intentions, with typical elements c. A communication intention may be either an input statement $\overline{c}$ or an output statement that we also write as c. Take note that dependent on the context we may use c for denoting an output statement or for a communication intention in general, including input statements.

We define the extensions $e \in E$ by

$$e ::= new(\hat{\alpha}) \mid c \mid \overline{c}$$

and *goals* $g \in \mathsf{Goal}$ by

$$g ::= a \mid p \mid g_1; g_2 \mid g_1 \,\square\, g_2 \mid e$$

A declaration D is of the form

$$D = \{p \leftarrow g \mid p \in \mathsf{Pvar}, g \in \mathsf{Goal}\}$$

and, in order to be able to specify the own activity of an object we associate with each $\hat{\alpha}$ a procedure variable $p \in \mathsf{Pvar}$ and assume that we have stated

$$body(\hat{\alpha}) = p$$

relating each $\hat{\alpha}$ to some p.

A *program*, again, is a tuple of the form $< D \mid g >$.

6.4.1 Operational semantics

Our operational semantics is set up in a similar way as for $\mathcal{B}_0$. As the result domain for characterizing the behavior of a program we take

$$\mathbb{R} = \mathcal{P}_{nc}(\Lambda^{\infty})$$

since the concurrent execution of objects gives rise to indeterminate behavior.

We assume to have a set of objects Obj, with typical elements α and β. These objects correspond in a natural way with the object names from O, in that for each $\alpha \in \mathsf{Obj}$ we have $\hat{\alpha} \in O$ and vice versa.

Communication between objects may be either successful or failing. We assume that each output statement c has a number of matching input statements that we write as $\overline{c}$, $\overline{c}_1$, $\overline{c}_2$,... Moreover we assume to have defined a function $eval : C \times C \rightarrow \mathsf{Goal}$, and say that the communication between c and $\overline{c}$ is successful if $eval(c, \overline{c}) = skip$ and failing if $eval(c, \overline{c}) = fail$. We also use a function $\overline{eval} : C \times C \rightarrow \mathsf{Goal}$ that depends on $eval$ in the following way.

$$\overline{eval}(c, \overline{c}) = \begin{cases} c & \text{if } eval(c, \overline{c}) = fail \\ skip & \text{otherwise} \end{cases}$$

The function $eval$ is used at the input side to determine whether backtracking must occur in case the communication fails. If the communication is failing applying $\overline{eval}$ results in repeating the request for communication at the output side. Otherwise, if the communication is successful the output side may proceed, and $\overline{eval}$ results in *skip*. As a comment, the interplay of $eval$ and $\overline{eval}$ models the communication over channels in DLP. There we allow the input side to backtrack until a successful communication is possible. The output side, on the other hand, must wait until the input side has found a successful input statement.

We use syntactic success continuations $R \in \mathsf{SuccCo}$ and syntactic failure continuations $F \in \mathsf{FailCo}$ as introduced for $\mathcal{B}_0$:

$$R ::= \surd \mid g; R$$

$$F ::= \Delta \mid R : F$$

Since each object may be active evaluating a goal, we use a tuple of the form $< \alpha, F >$ which we will also refer to as the object α. A configuration $\Gamma \in \mathsf{Conf} = \mathcal{P}(\mathsf{Obj} \times \mathsf{FailCo})$ consists of a set of such objects.

The transition rules for $\mathcal{B}_1$ listed below, must be augmented with a general rule

$$\Gamma \xrightarrow{\eta} \Gamma' \Longrightarrow X \cup \Gamma \xrightarrow{\eta} X \cup \Gamma'$$

stating that whenever a transition is possible from Γ to Γ' then any set of objects disjoint with Γ and Γ' may be added without affecting the possibility of a transition.

We have refined the interpretation function I, in order to let the outcome depend on the object by which the evaluation takes place.

$\mathcal{B}_1$

$\{< \alpha, a; R : F >\} \xrightarrow{a} \{< \alpha, R : F >\}$ if $I(a)(\alpha)$ — *Action*

$\longrightarrow \{< \alpha, F >\}$ otherwise

$\{< \alpha, p; R : F >\} \xrightarrow{\star} \{< \alpha, g; R : F >\}$ for $p \leftarrow g$ in D — *Rec*

$\{< \alpha, (g_1; g_2); R : F >\} \longrightarrow \{< \alpha, g_1; (g_2; R) : F >\}$ — *Seq*

$\{< \alpha, (g_1 \,\square\, g_2); R : F >\} \longrightarrow \{< \alpha, (g_1; R) : (g_2; R) : F >\}$ — *Alt*

$\{< \alpha, new(\hat{\beta}); R : F >\} \xrightarrow{\star} \{< \alpha, R : F >, < \beta, body(\hat{\beta}); \surd : \Delta >\}$ — *New*

$\{< \alpha, c; R_1 : F_1 >, < \beta, \overline{c}; R_2 : F_2 >\}$ — *Comm*

$\xrightarrow{\star} \{< \alpha, \overline{eval}(c, \overline{c}); R_1 : F_1 >, < \beta, eval(c, \overline{c}); R_2 : F_2 >\}$

$\{< \alpha, \surd : F >\} \longrightarrow \{< \alpha, F >\}$ — *Tick*

Apart from the fact that a configuration is now a collection of processes, the transition system differs from the one for $\mathcal{B}_0$ only by including the axioms *New* and *Comm*. Applying the axiom *New* results in adding an object to the collection of active objects, executing the body of that object. Applying *Comm* has as effect that the output statement c and the input statement $\overline{c}$ are replaced by respectively the outcome of $\overline{eval}(c, \overline{c})$ and $eval(c, \overline{c})$.

As an example of creating a new object, let $body(\hat{\beta}) = p$ and $p \leftarrow a$ in D, then we may have the computation

$$\{< \alpha, new(\hat{\beta}); \surd : \Delta >\}$$

$$\xrightarrow{\star} \{< \alpha, \surd : \Delta >, < \beta, p; \surd : \Delta >\} \longrightarrow \{< \alpha, \Delta >, < \beta, p; \surd : \Delta >\}$$

$$\xrightarrow{\star} \{< \alpha, \Delta >, < \beta, a; \surd : \Delta >\}$$

$$\xrightarrow{a} \{< \alpha, \Delta >, < \beta, \surd : \Delta >\} \longrightarrow \{< \alpha, \Delta >, < \beta, \Delta >\}$$

Next we give an example showing the backtracking that may occur in communication.
Let

$$D = \{p_1 = c, p_2 = \bar{c}_1 \,\square\, \bar{c}_2\}.$$

When we moreover define

$$eval(c, \bar{c}_1) = fail \text{ and } eval(c, \bar{c}_2) = skip$$

then we may have the computation

$$\{< \alpha, p_1; \surd : \Delta >, < \beta, p_2; \surd : \Delta >\}$$

$$\xrightarrow{\star} \ldots \xrightarrow{\star} \{< \alpha, c; \surd : \Delta >, < \beta, (\bar{c}_1 \,\square\, \bar{c}_2); \surd : \Delta >\}$$

$$\longrightarrow \{< \alpha, c; \surd : \Delta >, < \beta, (\bar{c}_1; \surd) : (\bar{c}_2; \surd) : \Delta >\}$$

$$\xrightarrow{\star} \{< \alpha, c; \surd : \Delta >, < \beta, (fail; \surd) : (\bar{c}_2; \surd) : \Delta >\}$$

$$\longrightarrow \{< \alpha, c; \surd : \Delta >, < \beta, \bar{c}_2; \surd : \Delta >\}$$

$$\xrightarrow{\star} \{< \alpha, skip; \surd : \Delta >, < \beta, skip; \surd : \Delta >\}$$

$$\xrightarrow{skip} \ldots \xrightarrow{skip} \ldots \longrightarrow \{< \alpha, \Delta >, < \beta, \Delta >\}$$

Notice that after the first attempt at communication, between c and $\bar{c}_2$ the output side does not change and the input side $\bar{c}_1$ is replaced by *fail*. The second attempt, however, succeeds and results in *skip* on both sides.

We may now define the operational semantics of a program $< D \mid g >$ as a function $\mathcal{O}[\![\cdot]\!] : \mathcal{B}_1 \to \mathbb{R}$, with $\mathbb{R} = \mathcal{P}_{nc}(\Lambda^\infty)$.

Definition 6.4.1 $\mathcal{O}[\![< D \mid g >]\!] = \mathcal{T}[\![< \alpha_0, g; \surd : \Delta >]\!]$ *for some initial object* α_0, *where for* $\Psi : (\mathsf{Conf} \to \mathbb{R}) \to (\mathsf{Conf} \to \mathbb{R})$ *and*

$$\Psi(\phi)[\![\Gamma]\!] = \begin{cases} \{\varepsilon\} & \text{if } \Gamma \text{ blocks} \\ \bigcup\{\eta \circ \phi[\![\Gamma']\!] : \Gamma \xrightarrow{\eta} \Gamma'\} & \text{otherwise} \end{cases}$$

we let $\mathcal{T} = \mathsf{fix}\Psi$.

As a difference with the definition given for $\mathcal{B}_0$ it must be noted that now a set of strings over Λ is delivered. The apparent non-determinism of a program is not due to its backtracking behavior, but comes from the indeterminacy introduced by the concurrent behavior of objects.

6.4.2 Denotational semantics

Since we wish to be able to ignore unresolved attempts at communication, as long as they do not block the computation, taking $\mathbb{R}$ as our domain for giving a denotational semantics no longer suffices.

Therefore we define the process domain $\mathbb{P}$, with typical elements ρ, by the equation

$$\mathbb{P} \cong \{\delta\} \cup \mathcal{P}_{co}(\Lambda \times \mathbb{P} \cup C \times (C \to \mathbb{P}))$$

For technical convenience, we use a special empty process δ. Non-empty processes in $\mathbb{P}$ are sets containing pairs of the form $\eta \cdot \rho$, with η a non-empty label, and pairs of the form $c \cdot f$, with c a communication intention from C and f an element from $C \to \mathbb{P}$. Remember that we use the pairing operator $\cdot$ to write pairs $< x, y >$ as $x \cdot y$. We will use syntactic variables ξ to range over elements of a set ρ. Intuitively, a pair $\eta \cdot \rho$ represents a computation step labeled by η and followed by a resumption ρ. A pair $c \cdot f$, on the other hand, represents the intention to communicate, that is to wait for a matching communication intention.

We model the concurrent behavior of a computation by arbitrary interleaving, and define the *merge* operator $\|: \mathbb{P} \times \mathbb{P} \to \mathbb{P}$ that must satisfy $\delta \| \rho = \rho = \rho \| \delta$ and for non-empty ρ_1 and ρ_2

$$\rho_1 \| \rho_2 = \rho_1 \lfloor\!\lfloor \rho_2 \cup \rho_2 \lfloor\!\lfloor \rho_1 \cup \rho_1 \mid \rho_2$$

where $\lfloor\!\lfloor$ is the so-called *left merge* operator defined by

$$\rho_1 \lfloor\!\lfloor \rho_2 = \{\eta \cdot (\rho \| \rho_2) : \eta \cdot \rho \in \rho_1\} \cup \{c \cdot \lambda c'.(f(c') \| \rho_2) : c \cdot f \in \rho_1\}$$

that merges the resumption of the items contained in ρ_1 with ρ_2. C.f. [de Bakker et al, 1984].
We define the communication operator $\mid$ by

$$\rho_1 \mid \rho_2 = \bigcup\{\xi_i \mid \xi_j : \xi_i \in \rho_i, \xi_j \in \rho_j \ \ for \ \ i,j \in \{1,2\} \ \ and \ \ i \neq j\}$$

and

$$\xi_1 \mid \xi_2 = \begin{cases} \{\star \cdot (f_1(\bar{c}) \| f_2(c))\} & \text{if } \xi_1 = c \cdot f_1 \text{ and } \xi_2 = \bar{c} \cdot f_2 \\ \emptyset & \text{otherwise} \end{cases}$$

Technically speaking, we must prove the merge operator to exist and be continuous by showing that it is the fixed point of a suitable operator, as has been explained before.

Similar as for $\mathcal{B}_0$ we use semantic success continuations $R \in \mathsf{Succ} = \mathsf{Fail} \to \mathbb{P}$ and semantic failure continuations $F \in \mathsf{Fail} = \mathbb{P}$. The function $\mathcal{D}[\![\cdot]\!] : \mathsf{Goal} \to \mathsf{Obj} \to \mathsf{Succ} \to \mathsf{Fail} \to \mathbb{P}$ is defined by the equations below. Let $R_0 = \lambda\rho.\rho$ and $F_0 = \delta$.

$\mathcal{B}_1$

(i) $\mathcal{D}[\![a]\!]\alpha RF = I(a)(\alpha) \rightarrow \{a \cdot RF\}, F$

(ii) $\mathcal{D}[\![p]\!]\alpha RF = \{\star \cdot \mathcal{D}[\![g]\!]\alpha RF\}$ for $p \leftarrow g$ in D_α

(iii) $\mathcal{D}[\![g_1; g_2]\!]\alpha RF = \mathcal{D}[\![g_1]\!]\alpha(\mathcal{D}[\![g_2]\!]\alpha R)F$

(iv) $\mathcal{D}[\![g_1 \,\square\, g_2]\!]\alpha RF = \mathcal{D}[\![g_1]\!]\alpha R(\mathcal{D}[\![g_2]\!]\alpha RF)$

(v) $\mathcal{D}[\![new(\hat{\beta})]\!]\alpha RF = \{\star \cdot (RF \,||\, \mathcal{D}[\![body(\hat{\beta})]\!]\beta R_0 F_0)\}$

(vi) $\mathcal{D}[\![c]\!]\alpha RF = \{c \cdot f\}$ with $f = \lambda \bar{c} \in C.\mathcal{D}[\![\overline{eval}(c, \bar{c})]\!]\alpha RF)$

(vii) $\mathcal{D}[\![\bar{c}]\!]\alpha RF = \{\bar{c} \cdot f\}$ with $f = \lambda c \in C.\mathcal{D}[\![eval(c, \bar{c})]\!]\alpha RF)$

Recall that the expression RF both in (i) and (v) denotes the application of R to F.

Definition 6.4.2 $\mathcal{M}[\![< D \mid g >]\!] = \mathcal{D}[\![g]\!]\alpha_0 R_0 F_0$ *for some initial object* α_0.

6.4.3 Equivalence between operational and denotational semantics

In contrast to the equivalence results for our previous languages we are not able now to state a direct equivalence between $\mathcal{O}$ and $\mathcal{M}$. We must use a projection π that takes elements from IP to IR, yielding sequences of actions. Moreover, our projection eliminates unresolved communication intentions that do not block the computation.

Theorem 6.4.3 $\mathcal{O}[\![< D \mid g >]\!] = \pi \circ \mathcal{M}[\![< D \mid g >]\!]$ *for a suitable projection* π.

Assuming that we have such a projection π we may outline the proof as

$$\begin{aligned}
\mathcal{O}[\![< D \mid g >]\!] &= \mathcal{T}[\![\{< \alpha_0, g; \surd : \Delta >\}]\!] \text{ by definition 6.4.1} \\
&= \pi \circ \mathcal{G}[\![\{< \alpha_0, g; \surd : \Delta >\}]\!] \text{ by corollary 6.4.7} \\
&= \pi \circ \mathcal{F}[\![g; \surd : \Delta]\!]\alpha_0 \text{ by definition 6.4.5} \\
&= \pi \circ \mathcal{D}[\![g]\!]\alpha_0 R_0 F_0 \text{ by definition 6.4.5} \\
&= \pi \circ \mathcal{M}[\![< D \mid g >]\!] \text{ by definition 6.4.2}
\end{aligned}$$

Apart from the semantic functions $\mathcal{F}$: FailCo $\rightarrow$ Fail and $\mathcal{R}$: SuccCo $\rightarrow$ Succ, mapping syntactic failure continuations and syntactic success continuations to their semantic counterparts, we also need to have a function $\mathcal{G}$: Conf $\rightarrow$ IP mapping configurations Γ, which are sets of objects, to an element from IP.

Definition 6.4.4

a. $\mathcal{R}[\![\surd]\!]\alpha = \lambda\rho.\rho$

b. $\mathcal{R}[\![g; R]\!]\alpha = \mathcal{D}[\![g]\!]\alpha\mathcal{R}[\![R]\!]\alpha$

c. $\mathcal{F}[\![\Delta]\!]\alpha = \delta$

d. $\mathcal{F}[\![R : F]\!]\alpha = \mathcal{R}[\![R]\!]\alpha\mathcal{F}[\![F]\!]\alpha$

e. $\mathcal{G}[\![\{< \alpha_1, F_1 >, ..., < \alpha_n, F_n >\}]\!] = \mathcal{F}[\![F_1]\!]\alpha_1 \parallel ... \parallel \mathcal{F}[\![F_n]\!]\alpha_n$

As we may see from the definition above, the meaning of a set of objects is simply the merge of the meaning of the individual elements.

Again we have the property that unobservable transitions do not affect the (mathematical) meaning of a configuration.

Lemma 6.4.5 *if* $\Gamma \longrightarrow \Gamma'$ *then* $\mathcal{G}[\![\Gamma]\!] = \mathcal{G}[\![\Gamma']\!]$

Proof: We will treat the case that $\Gamma = \{< \alpha, a; R : F >\}$, assuming that $I(a)(\alpha)$ is *false*.
$\mathcal{G}[\![\Gamma]\!] = \mathcal{F}[\![a; R : F]\!]\alpha = \mathcal{D}[\![a]\!]\alpha\mathcal{R}[\![R]\!]\alpha\mathcal{F}[\![F]\!]\alpha =$ (since $I(a)(\alpha)$ is *false*)
$\mathcal{F}[\![F]\!]\alpha = \mathcal{G}[\![< \alpha, F >]\!]$ as desired. □

In order to show that $\pi \circ \mathcal{G}$ is a fixed point of the higher order mapping characterizing $\mathcal{T}$ we will first define the projection operator π that reduces the unresolved communication intentions to $\{\varepsilon\}$, the equivalent of the empty process δ.

First we define a function $rem : \mathbb{IP} \rightarrow \mathbb{IP}$ by $rem(\delta) = \emptyset$ and

$$rem(\rho) = \rho \backslash \{c \cdot f : c \cdot f \in \rho\}$$

which defines $rem(\rho)$ to be what remains after removing all unresolved communication intentions and occurrences of δ. We then define $\pi : \mathbb{IP} \rightarrow \mathbb{IR}$ by

$$\pi\ \rho\ = \begin{cases} \{\varepsilon\} & \text{if } rem(\rho) = \emptyset \\ \bigcup\{\eta \circ (\pi\ \rho') : \eta \cdot \rho' \in rem(\rho)\} & \text{otherwise} \end{cases}$$

A property of π that we will use below is expressed by $\bigcup\{\eta \circ (\pi\ \rho')\} = \pi\ \{\eta \cdot \rho'\}$, which can easily be verified.

Lemma 6.4.6 $\Psi(\pi \circ \mathcal{G}) = \pi \circ \mathcal{G}$

Proof: We use a complexity measure similar as the one introduced for $\mathcal{B}_0$, but extended in order to cope with configurations containing a collection of objects. What we need is that whenever $\Gamma \longrightarrow \Gamma'$ then $c(\Gamma) > c(\Gamma')$. We define $c(\Gamma) = \Sigma_{x \in \Gamma}\, c(x)$ and let $c(< \alpha, F >) = c(F)$. We further extend the complexity measure for $\mathcal{B}_0$ by taking $c(e) = 1$ for $e \in E$.

We must now prove for all Γ that $\Psi(\pi \circ \mathcal{G})[\![\Gamma]\!] = \pi \circ \mathcal{G}[\![\Gamma]\!]$. It may be observed, since we have defined $\mathcal{G}[\![\{< \alpha_1, F_1 >, ..., < \alpha_n, F_n >\}]\!] = \mathcal{F}[\![F_1]\!]\alpha_1 \parallel ... \parallel \mathcal{F}[\![F_n]\!]\alpha_n$, that the equality $\mathcal{G}[\![\Gamma \cup \Gamma']\!] = \mathcal{G}[\![\Gamma]\!] \parallel \mathcal{G}[\![\Gamma']\!]$ holds, due to the obvious associativity and commutativity of the parallel merge operator. Hence $\pi \circ \mathcal{G}[\![\Gamma \cup \Gamma']\!] = \pi(\mathcal{G}[\![\Gamma]\!] \parallel$

$\mathcal{G}[\![\Gamma']\!]$). This observation allows us, for non-blocking configurations, to restrict our attention to the (syntactic) processes in the configuration Γ that really play a role in the transition taken. If a configuration Γ blocks then, since this is either due to termination or to an unresolved communication intention, the equality $\Psi(\pi \circ \mathcal{G})[\![\Gamma]\!] = \pi \circ \mathcal{G}[\![\Gamma]\!]$ clearly holds. We will treat some selected cases of non-blocking configurations. Let $\overline{R_i} = \mathcal{R}[\![R_i]\!]\alpha_i$ and $\overline{F_i} = \mathcal{F}[\![F_i]\!]\alpha_i$, with subscript i possibly empty.

- If $\Gamma = \{< \alpha, a; R : F >\}$ then in case $I(a)(\alpha)$ we have
  $\Psi(\pi \circ \mathcal{G})[\![\Gamma]\!] = \bigcup\{a \circ (\pi \circ \mathcal{G})[\![\{< \alpha, R : F >\}]\!]\} = \pi\ \{a \cdot \mathcal{G}[\![\{< \alpha, R : F >\}]\!]\}$
  which, by the definition of $\mathcal{R}$ and $\mathcal{D}$, is equal to $\pi \circ \mathcal{G}[\![\Gamma]\!]$.
  If $I(a)(\alpha)$ is *false* then $\Gamma \rightarrow \Gamma'$ with $\Gamma' = \{< \alpha, F >\}$. Since $c(\Gamma') < c(\Gamma)$ we have by induction $\Psi(\pi \circ \mathcal{G})[\![\Gamma']\!] = \pi \circ \mathcal{G}[\![\Gamma']\!]$ and by lemma 6.4.5 this is equal to $\pi \circ \mathcal{G}[\![\Gamma]\!]$.

- If $\Gamma = \{< \alpha, new(\hat{\beta}); R : F >\}$ then
  $\Psi(\pi \circ \mathcal{G})[\![\Gamma]\!] = \bigcup\{\star \circ (\pi \circ \mathcal{G})[\![\{< \alpha, R : F >, < \beta, body(\hat{\beta}); \surd : \Delta >\}]\!]\} =$
  (by the property of π)
  $\pi\ \{\star \cdot \mathcal{G}[\![\{< \alpha, R : F >, < \beta, body(\hat{\beta}); \surd : \Delta >\}]\!]\} =$ (by the definition of $\mathcal{G}$)
  $\pi\ \{\star \cdot (\overline{RF} \parallel \mathcal{D}[\![body(\hat{\beta})]\!]\beta\mathcal{R}[\![\surd]\!]\beta\mathcal{F}[\![\Delta]\!]\beta)\} =$ (by the definition of $\mathcal{D}$)
  $\pi\ \mathcal{D}[\![new(\hat{\beta})]\!]\alpha\overline{RF} =$ (by applying definition 6.4.4)
  $\pi \circ \mathcal{G}[\![\{< \alpha, new(\hat{\beta}); R : F >\}]\!]$ which proves the result.

- Let $\Gamma = \{< \alpha_1, c; R_1 : F_1 >, < \alpha_2, \overline{c}; R_2 : F_2 >\}$.
  Then $\Psi(\pi \circ \mathcal{G})[\![\Gamma]\!] =$ (by applying *Comm*)
  $\pi\ \{\star \cdot \mathcal{G}[\![\{< \alpha_1, \overline{eval}(c, \overline{c}); R_1 : F_1 >, < \alpha_2, eval(c, \overline{c}); R_2 : F_2 >\}]\!]\} =$
  (by definition 6.4.4)
  $\pi\ (\{\star \cdot (\mathcal{D}[\![\overline{eval}(c, \overline{c})]\!]\alpha_1\overline{R}_1\overline{F}_1 \parallel \mathcal{D}[\![eval(c, \overline{c})]\!]\alpha_2\overline{R}_2\overline{F}_2)\}) =$
  (by the definition of communication)
  $\pi\ (\{c \cdot \lambda\tau.\mathcal{D}[\![\overline{eval}(c, \tau)]\!]\alpha_1\overline{R}_1\overline{F}_1\} \parallel \{\overline{c} \cdot \lambda\tau.\mathcal{D}[\![eval(\tau, \overline{c})]\!]\alpha_2\overline{R}_2\overline{F}_2\}) =$
  (by the definition of $\mathcal{D}$)
  $\pi\ (\mathcal{D}[\![c]\!]\alpha_1\overline{R}_1\overline{F}_1 \parallel \mathcal{D}[\![\overline{c}]\!]\alpha_2\overline{R}_2\overline{F}_2) =$ (by applying definition 6.4.4 again)
  $\pi \circ \mathcal{G}[\![\{< \alpha_1, c; R_1 : F_1 >, < \alpha_2, \overline{c}; R_2 : F_2 >\}]\!]$. □

We may now simply collect our result.

Corollary 6.4.7 $\mathcal{T} = \pi \circ \mathcal{G}$

6.5 Dynamic object creation and communication with global backtracking

The language $\mathcal{B}_2$, our second extension of $\mathcal{B}_0$, is similar to $\mathcal{B}_1$ in that a number of objects may become concurrently active, but differs from it in the way communication is dealt with. The model of communication employed is that of a synchronous rendez-vous. A difference with the rendez-vous mechanism as encountered in Ada or POOL however is the possible occurrence of backtracking over the resulting

answers. In order to maintain this backtrack information a process is created for each rendez-vous. Therefore we no longer identify objects and processes but instead say that multiple processes may refer to a single object. We distinguish between two kinds of processes, namely *constructor* processes that execute the own activity of an object and processes that are created for handling a rendez-vous. As for $\mathcal{B}_1$ we put the burden of creating unique object and process identifiers on the programmers shoulders, who may use a collection of object and process names for this purpose.

We regard the rendez-vous as the evaluation of a method call by an object and therefore will speak of methods instead of procedures. For a rendez-vous to take place an object must state its willingness to accept a method call. Acceptance is non-deterministic in that the object may choose to accept one of a number of possible requests.

Syntax Again we assume to have a sets of actions A, with typical elements a. We also have method variables $m \in \mathsf{Mvar}$, that take the place of procedure variables. Further we introduce a set of process names P, with typical elements $\hat{\alpha}$ and $\hat{\beta}$, and assume to have also a set of object names O, with typical elements $\underline{\alpha}$ and $\underline{\beta}$, We take O disjoint from P. Since processes refer to an object, for each element $\hat{\alpha}$ in P, we state that there is an object name $\underline{\hat{\alpha}}$ in O.

We assume to have a set of results R, with typical elements $\overline{r}$, and moreover assume that with each $m \in \mathsf{Mvar}$ is associated an $\overline{r}_m$ to be used in communicating the results of a rendez-vous. Further we have a set Comm of communication intentions, consisting of elements of the form $\hat{\alpha}!m$, $\overline{m}$, $\hat{\alpha}?$ and $\overline{r}$, of which the meaning will be explained later.

We define extensions $e \in E$ by

$$e ::= new(\underline{\hat{\alpha}}) \mid \hat{\alpha}!m \mid [m_1, ..., m_n] \mid \hat{\alpha}? \mid \overline{r}$$

and define *goals* $g \in \mathsf{Goal}$ by

$$g ::= a \mid m \mid g_1; g_2 \mid g_1 \,\square\, g_2 \mid e$$

A *declaration* D is of the form

$$D = \{m \leftarrow g \mid m \in \mathsf{Mvar}, g \in \mathsf{Goal}\}$$

and we assume to have a function *body* defining $body(\underline{\hat{\alpha}}) = m$ for each $\underline{\hat{\alpha}}$.

A *program* is a tuple $< D \mid g >$.

6.5.1 Operational semantics

As the result domain for characterizing the behavior of a program we take

$$\mathbb{R} = \mathcal{P}_{nc}(\Lambda^\infty)$$

just as for $\mathcal{B}_1$.

We now however use a set of processes Proc, with typical elements α and β, and further assume to have a set of objects Obj, with typical elements $\underline{\alpha}$ and $\underline{\beta}$. We state that Obj $\subset$ Proc, and moreover associate with each element $\alpha \in$ Proc an

element $\underline{\alpha} \in \mathsf{Obj}$ that we call the object to which α refers. We let Obj be contained in Proc since we may identify the process executing the constructor for an object with the object itself. For each object name $\hat{\underline{\alpha}}$ we have an object $\underline{\alpha}$ in Obj, and for each process name $\hat{a}$ we have a process α in Proc. Note that, since O and P are disjunct, the process β derived from the process name $\hat{\beta}$, is distinct from the process $\underline{\beta}$ corresponding to the object name $\hat{\underline{\beta}}$, although the process β does refer to the object $\underline{\beta}$.

Communication by rendez-vous embodies the most important and yet most difficult aspect of our language $\mathcal{B}_2$. A complication in treating this communication is the possible occurrence of backtracking over the answers produced in a rendez-vous. For this reason we have decomposed a synchronous rendez-vous in a method call of the form $\hat{\beta}!m$ and a request for the answers resulting from evaluating m, of the form $\hat{\beta}?$. A method call $\hat{\beta}!m$ is considered to be addressed to the object $\underline{\beta}$, that is the process $\underline{\beta}$ executing the constructor $body(\hat{\underline{\beta}})$. The caller is assumed to provide the process name $\hat{\beta}$ for the process β that is created to evaluate m.

We have pictured a typical rendez-vous below and in figure 6.1.

As soon as the call $\hat{\beta}!m$ is accepted, due to the occurrence of $[..., m, ...]$ in $\underline{\beta}$, process α requests for an answer by stating $\hat{\beta}?$. The process β, which refers to the object $\underline{\beta}$, starts evaluating the goal m. Process α must wait until process β has computed an answer. When the first answer is computed it is sent to α. The process $\underline{\beta}$, which is the constructor process for the object $\underline{\beta}$, may then proceed with the evaluation of the constructor $body(\hat{\underline{\beta}})$.

Similar as for $\mathcal{B}_1$ we use a function $eval : \mathsf{Comm} \times \mathsf{Comm} \rightarrow \mathsf{Goal}$ by which we decide what course of action must be taken by the process requesting the answer after the answer has been communicated. We say that there are no more answers when $eval(\hat{\beta}?, \overline{r}) = fail$, which must be the case when $\overline{r} = \overline{r}_\delta$; and that a solution is found when $eval(\hat{\beta}?, \overline{r}) = skip \,\square\, \hat{\beta}?$, which allows for backtracking over alternative answers.

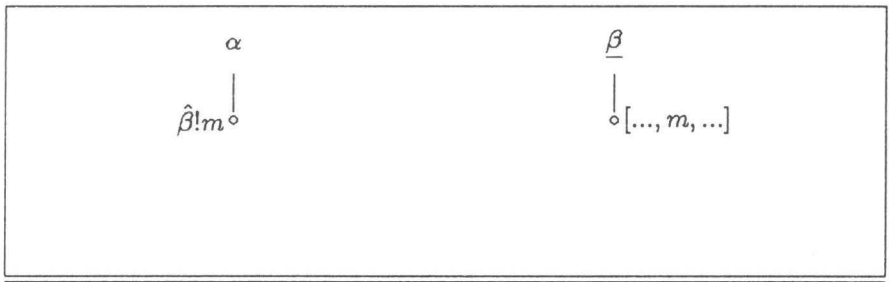
When backtracking occurs in α, the goal $\hat{\beta}?$ succeeds as many times as there are alternative answers produced by β.

Our treatment thus far covers both the synchronization and the transfer of results taking place in a rendez-vous. Before we give a more detailed description of the global backtracking as it may occur in a rendez-vous we must comment that for DLP we have adopted a protocol of mutual exclusion that guarantees that no other method call becomes active until the first answer is delivered. After having communicated the first result, the process evaluating the method becomes independent, and immediately starts producing an alternative answer concurrently with the own activity of the object to which the method call was addressed.

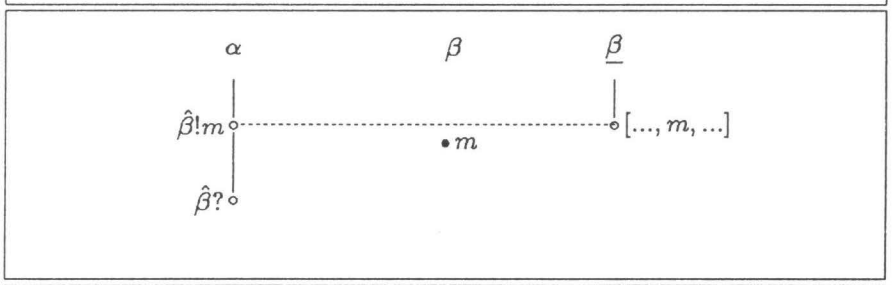
The backtracking that occurs locally in a process is modeled by using syntactic failure continuations, which are stacks of syntactic success continuations, that we have defined as

$$R ::= \surd \mid g; R$$
$$F ::= \Delta \mid R : F$$

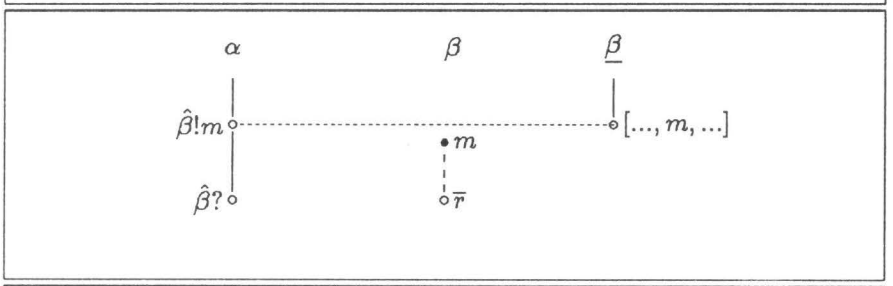
Figure 6.1: Matching communication pairs in a rendez-vous



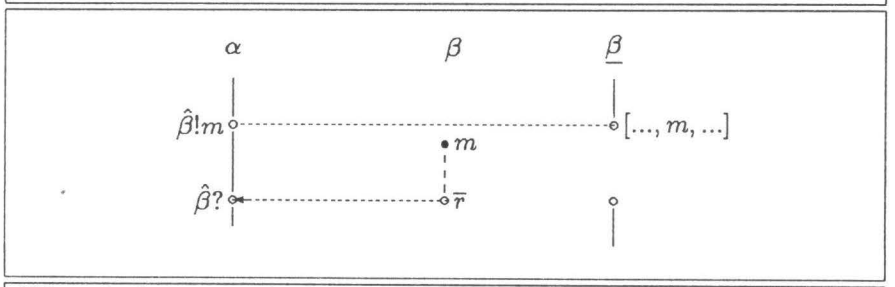
Process α is calling a method m, with O bound to $\underline{\beta}$. Process $\underline{\beta}$ is willing to accept a call for method m.



A new process β is created to evaluate the method m. Process α may proceed as soon as the call is accepted.



The evaluation of m continues until β arrives at a $\overline{r}$. Process α must wait until the request $\hat{\beta}?$ is answered.



When the first answer has been sent to α, both process α and process $\underline{\beta}$, that has accepted the method call, may continue.

Without distributed backtracking we could have processes of the form $< \alpha, F >$, with α a process identifier and F a failure continuation, just as for $\mathcal{B}_1$. However, the possible occurrence of distributed backtracking complicates matters, in that we need to introduce yet another kind of syntactic continuations. We define *process continuations* C by

$$C ::= \Xi \mid < \alpha, F >: C$$

The empty process continuation Ξ is simply a process that has nothing left to do. So we may forget about its name. As an example, a process continuation of the form $< \alpha, F >: \Xi$ is simply the process evaluating the failure continuation F. Now suppose that for process $\underline{\beta}$, evaluating the constructor $body(\hat{\beta})$, we have a process continuation $< \underline{\beta}, [..., m, ...]; R : F >: \Xi$ then, as we explained previously, a process that for the moment we indicate by $< \beta, m; \overline{r}_m \,\square\, \overline{r}_\delta; \surd : \Delta >$, evaluating m, may be started when another process calls for $\hat{\beta}!m$. We can however not have the process $\underline{\beta}$ running concurrently with the process β, since we must block the acceptance of any other method call for $\underline{\beta}$ until β has delivered its first answer. Our solution to modeling this protocol of mutual exclusion is simply to put $< \beta, (m; \overline{r}_m \,\square\, \overline{r}_\delta); \surd : \Delta >$ in front of the process continuation $< \underline{\beta}, R : F >: \Xi$, thus blocking the execution of $\underline{\beta}$. In other words a process continuation may be regarded as a process stack, of which the top is the active process. Then, as soon as a solution has been computed, which is represented by the process continuation $< \beta, \overline{r}_m; R' : F' >:< \beta, R : F >: \Xi$ we may communicate the answer to the process asking for it. From then on, we have two processes running concurrently, namely $< \beta, R' : F' >: \Xi$ to explore any alternative solutions contained in the failure continuation $R' : F'$, and $< \underline{\beta}, R : F >: \Xi$, evaluating the constructor for $\underline{\hat{\beta}}$, which may now accept new method calls.

As indicated, configurations $\Gamma \in \mathsf{Conf} = \mathcal{P}(\mathsf{ProcCo})$ are sets of syntactic process continuations.

The transition rules are listed below. Again we assume to have a general rule of the form

$$\Gamma \xrightarrow{\eta} \Gamma' \Longrightarrow X \cup \Gamma \xrightarrow{\eta} X \cup \Gamma'$$

for $X \in \mathsf{Conf}$ disjoint from Γ and Γ'. We now let I depend on the process by which the evaluation occurs.

$\mathcal{B}_2$

$$\{< \alpha, a; R : F >: C\} \xrightarrow{a} \{< \alpha, R : F >: C\} \text{ if } I(a)(\alpha) \qquad \textit{Action}$$
$$\longrightarrow \{< \alpha, F >: C\} \text{ otherwise}$$

$$\{< \alpha, m; R : F >: C\} \xrightarrow{\star} \{< \alpha, g; R : F >: C\} \text{ for } m \leftarrow g \text{ in } D \qquad \textit{Rec}$$

$$\{< \alpha, (g_1; g_2); R : F >: C\} \longrightarrow \{< \alpha, g_1; (g_2; R) : F >: C\} \qquad \textit{Seq}$$

$$\{< \alpha, (g_1 \,\square\, g_2); R : F >: C\} \longrightarrow \{< \alpha, (g_1; R) : (g_2; R) : F >: C\} \qquad \textit{Alt}$$

$$\{< \alpha, \textit{new}(\underline{\beta}); R : F >: C\} \qquad \textit{New}$$
$$\xrightarrow{\star} \{< \alpha, R : F >: C, < \underline{\beta}, \textit{body}(\hat{\underline{\beta}}); \surd : \Delta >: \Xi\}$$

$$\{< \alpha, \hat{\beta}!m; R_1 : F_1 >: C_1, < \underline{\beta}, [..., m, ...]; R_2 : F_2 >: C_2\} \qquad \textit{Method}$$
$$\xrightarrow{\star} \{< \alpha, \hat{\beta}?; R_1 : F_1 >: C_1,$$
$$< \beta, (m; \overline{r}_m \,\square\, \overline{r}_\delta); \surd : \Delta >:< \underline{\beta}, R_2 : F_2 >: C_2\}$$

$$\{< \alpha, \hat{\beta}?; R_1 : F_1 >: C_1, < \beta, \overline{r}; R_2 : F_2 >: C_2\} \qquad \textit{Result}$$
$$\xrightarrow{\star} \{< \alpha, \textit{eval}(\hat{\beta}?, \overline{r}); R_1 : F_1 >: C_1, < \beta, R_2 : F_2 >: \Xi, C_2\}$$

$$\{< \alpha, \surd : F >: C\} \longrightarrow \{< \alpha, F >: C\} \qquad \textit{Tick}$$

Apart from adding a process continuation component to every tuple of the form $< \alpha, F >$ the only difference with the transition rules for $\mathcal{B}_1$ is the replacement of the axiom *Comm* by the axioms *Method* and *Result*.

We will illustrate their use with an example. Let

$$D = \{m \leftarrow a_1 \,\square\, a_2\}$$

then

$$\{< \alpha, \hat{\beta}!m; \surd : \Delta >: \Xi, < \underline{\beta}, [..., m, ...]; \surd : \Delta >: \Xi\} \text{ (by applying } \textit{Method} \text{)}$$

$$\xrightarrow{\star} \{< \alpha, \hat{\beta}?; \surd : \Delta >: \Xi, < \beta, ((m; \overline{r}_m) \,\square\, \overline{r}_\delta); \surd : \Delta >:< \underline{\beta}, \surd : \Delta >: \Xi\}$$

The call $\hat{\beta}!m$ is replaced by the request for an answer $\hat{\beta}?$, where $\hat{\beta}$ is the name for the process evaluating the method m. The process β itself is pushed on the stack for the object $\underline{\beta}$, thus blocking any further activity of $\underline{\beta}$. The next transitions come from the evaluation of m, after putting $\overline{r}_\delta$, the reply in case of failure, as the last alternative on the failure stack of β.

$$\longrightarrow \{< \alpha, \hat{\beta}?; \surd : \Delta >: \Xi, < \beta, ((m; \overline{r}_m); \surd) : (\overline{r}_\delta; \surd) : \Delta >:< \underline{\beta}, \surd : \Delta >: \Xi\}$$

$$\longrightarrow \{< \alpha, \hat{\beta}?; \surd : \Delta >: \Xi, < \beta, m; (\overline{r}_m; \surd) : (\overline{r}_\delta; \surd) : \Delta >:< \underline{\beta}, \surd : \Delta >: \Xi\}$$

$$\xrightarrow{\star} \{< \alpha, \hat{\beta}?; \surd : \Delta >: \Xi,$$
$$< \beta, (a_1 \,\square\, a_2); (\overline{r}_m; \surd) : (\overline{r}_\delta; \surd) : \Delta >:< \underline{\beta}, \surd : \Delta >: \Xi\}$$

Process β starts the evaluation of the body of m.

$$\longrightarrow \{< \alpha, \hat{\beta}?; \surd : \Delta >: \Xi,$$
$$< \beta, a_1; (\overline{r}_m; \surd) : a_2; (\overline{r}_m; \surd) : (\overline{r}_\delta; \surd) : \Delta >:< \underline{\beta}, \surd : \Delta >: \Xi\}$$

$$\xrightarrow{a_1} \{< \alpha, \hat{\beta}?; \surd : \Delta >: \Xi,$$
$$< \beta, (\overline{r}_m; \surd) : a_2; (\overline{r}_m; \surd) : (\overline{r}_\delta; \surd) : \Delta >:< \underline{\beta}, \surd : \Delta >: \Xi\}$$

The process β is ready to communicate its first result.

$$\xrightarrow{\star} \{< \alpha, (skip \,\square\, \hat{\beta}?); \surd : \Delta >: \Xi, < \beta, \surd : a_2; (\overline{r}_m; \surd) : (\overline{r}_\delta; \surd) : \Delta >: \Xi,$$
$$< \underline{\beta}, \surd : \Delta >: \Xi\}$$

The object $\underline{\beta}$ may now continue its own activity and β may concurrently search for alternative answers.

$$\longrightarrow \ldots \longrightarrow \{< \alpha, \hat{\beta}?; \surd : \Delta >: \Xi,$$
$$< \beta, a_2; (\overline{r}_m; \surd) : (\overline{r}_\delta; \surd) : \Delta >: \Xi, < \underline{\beta}, \Delta >: \Xi\}$$

$$\xrightarrow{a_2} \{< \alpha, \hat{\beta}?; \surd : \Delta >: \Xi, < \beta, (\overline{r}_m; \surd) : (\overline{r}_\delta; \surd) : \Delta >: \Xi, < \underline{\beta}, \Delta >: \Xi\}$$

Now β is ready to communicate its second answer.

$$\xrightarrow{\star} \{< \alpha, (skip \,\square\, \hat{\beta}?); \surd : \Delta >: \Xi, < \beta, \surd : (\overline{r}_\delta; \surd) : \Delta >: \Xi, \Xi, < \underline{\beta}, \Delta >: \Xi\}$$

$$\xrightarrow{skip} \ldots \longrightarrow \{< \alpha, \hat{\beta}?; \surd : \Delta >: \Xi, < \beta, \overline{r}_\delta; \surd : \Delta >: \Xi, \Xi, < \underline{\beta}, \Delta >: \Xi\}$$

Finally β communicates failure, since no more answers are available.

$$\xrightarrow{\star} \{< \alpha, fail; \surd : \Delta >: \Xi, < \beta, \surd : \Delta >: \Xi, \Xi, < \underline{\beta}, \Delta >: \Xi\}$$

$$\longrightarrow \ldots \longrightarrow \{< \alpha, \Delta >: \Xi, < \beta, \Delta >: \Xi, \Xi, < \underline{\beta}, \Delta >: \Xi\}$$

When an answer is communicated the process that has produced the answer is popped from the process stack in order to enable both the process and the remainder of the stack to run concurrently. In practice the stack contains never more than two non-empty processes. In the case that it contains only one non-empty process, the empty process is added to the collection of processes.

The *operational semantics* of a program $< D \mid g >$ is defined in the usual way.

Definition 6.5.1 $\mathcal{O}[\![< D \mid g >]\!] = \mathcal{T}[\![< \alpha_0, g; \surd : \Delta >: \Xi]\!]$ *for an initial process* α_0,

where for

$$\Psi(\phi)[\![\Gamma]\!] = \begin{cases} \{\varepsilon\} & \text{if } \Gamma \text{ blocks} \\ \bigcup\{\eta \circ \phi[\![\Gamma']\!] : \Gamma \xrightarrow{\eta} \Gamma'\} & \text{otherwise} \end{cases}$$

we let $\mathcal{T} = \mathsf{fix}\Psi$.

6.5.2 Denotational semantics

We construct our domain in a similar way as for $\mathcal{B}_1$ but, since communication in $\mathcal{B}_2$ is more complex the part taking care of the communication intentions is correspondingly more involved.

We first define a set $C = \mathsf{Proc} \times \mathsf{Comm}$, with typical elements c. In other words C has elements of the form $[\alpha, \hat{\beta}!m]$, $[\alpha, \overline{m}]$, $[\alpha, \hat{\beta}?]$ and $[\alpha, \overline{r}]$. We further assume to have a set $T = \mathsf{Proc} \cup \mathsf{Comm}$ and define the domain $\mathbb{IP}$, with typical elements ρ and q by

$$\mathbb{IP} \cong \{\delta\} \cup \mathcal{P}_{co}(\Lambda \times \mathbb{IP} \cup C \times \mathbb{IP} \cup C \times (T \to \mathbb{IP}))$$

We let ξ range over elements of a set ρ, and f over functions from $T \to \mathbb{IP}$.

As an example of a process in $\mathbb{IP}$ consider the set $\{a \cdot \rho, c_1 \cdot \rho', c_2 \cdot f\}$, containing one action a with resumption ρ and two communication intentions, one with resumption from $\mathbb{IP}$ and one with a resumption from $T \to \mathbb{IP}$.

We have to adapt the *parallel merge* operator given for $\mathcal{B}_1$ to the present situation.

The operator $\|$: $\mathbb{IP} \times \mathbb{IP} \to \mathbb{IP}$ must satisfy $\delta \,\|\, \rho = \rho = \rho \,\|\, \delta$ and for non-empty ρ_1 and ρ_2

$$\rho_1 \,\|\, \rho_2 = \rho_1 \between \rho_2 \cup \rho_2 \between \rho_1 \cup \rho_1 \mid \rho_2$$

with the *left merge* operator $\between$ defined by

$$\begin{aligned} \rho_1 \between \rho_2 = &\{\eta \cdot (\rho \,\|\, \rho_2) : \eta \cdot \rho \in \rho_1\} \cup \\ &\{c \cdot (\rho \,\|\, \rho_2) : c \cdot \rho \in \rho_1\} \cup \\ &\{c \cdot \lambda\tau.(f(\tau) \,\|\, \rho_2) : c \cdot f \in \rho_1\} \end{aligned}$$

and the communication operator $\mid$ by

$$\rho_1 \mid \rho_2 = \bigcup\{\xi_i \mid \xi_j : \xi_i \in \rho_i, \xi_j \in \rho_j \;\; \textit{for} \;\; i,j \in \{1,2\} \;\; \textit{and} \;\; i \neq j\}$$

and

$$\xi_1 \mid \xi_2 = \begin{cases} \{\star \cdot (\rho \,\|\, f(\beta))\} & \text{if } \xi_1 = [\alpha, \hat{\beta}!m] \cdot \rho \text{ and } \xi_2 = [\underline{\beta}, \overline{m}] \cdot f \\ \{\star \cdot (f(\overline{r}) \,\|\, \rho)\} & \text{if } \xi_1 = [\alpha, \hat{\beta}?] \cdot f \text{ and } \xi_2 = [\beta, \overline{r}] \cdot \rho \\ \emptyset & \text{otherwise} \end{cases}$$

Process insertion When a request for a method call is granted, the process evaluating the call has exclusive access to the object to which the method call was directed, exclusive in the sense that no other process referring to that object will be active until the first answer or failure is returned to the invoking process. However, when this happens, that is when the invoking process receives the first answer, the (constructor) process that accepted the request for a rendez-vous may continue its computation, in parallel with everything else that is going on. In order to let the accepting process be active immediately after the first result has been returned we need an *insertion operator* that grafts the resumption of this process at the right place in the (semantic) process due to evaluating the method call.

We define the insertion operator $\rhd_\alpha : \mathrm{IP} \times \mathrm{IP} \to \mathrm{IP}$ by $\delta \rhd_\alpha \delta = \delta$, $\delta \rhd_\alpha \rho = \rho$, $\rho \rhd_\alpha \delta = \delta$ and for non-empty ρ and q

$$\begin{aligned}\rho \rhd_\alpha q = {} & \{\eta \cdot \rho \rhd_\alpha q' : \eta \cdot q' \in q\} \cup \\ & \{[\alpha, \overline{r}] \cdot (\rho \parallel q') : [\alpha, \overline{r}] \cdot q' \in q\} \cup \\ & \{c \cdot \rho \rhd_\alpha q' : c \cdot q' \in q \ \ for \ \ c \neq [\alpha, \overline{r}]\} \cup \\ & \{c \cdot \lambda\tau.\rho \rhd_\alpha f(\tau) : c \cdot f \in q\}\end{aligned}$$

In other words the first time that the intention for communicating an answer substitution of the form $[\alpha, \overline{r}] \cdot q'$, is encountered the inserted process ρ is put in parallel with the resumption awaiting a successful communication. We may then not put $[\alpha, \overline{r}] \cdot \rho \rhd_\alpha q'$ in also. In definition 6.5.4 we will encounter the equivalence $\mathcal{C}[\![< \alpha, F >: C]\!] = \mathcal{C}[\![C]\!] \rhd_\alpha \mathcal{F}[\![F]\!]\alpha$, where $\mathcal{C}[\![C]\!]$ is the meaning of the (syntactic) process continuation C and $\mathcal{F}[\![F]\!]\alpha$ the meaning of the (syntactic) failure continuation F. We regard this equation to embody the intuition behind the insertion operator.

Again, we remark that in order to prove the operators $\parallel$ and $\rhd_\alpha$ do exist and behave properly by being continuous, we may define them using contracting higher order mappings.

The semantic function $\mathcal{D}[\![\cdot]\!] : \mathrm{Goal} \to \mathrm{Proc} \to \mathrm{Succ} \to \mathrm{Fail} \to \mathrm{IP}$ is given by the equations below. Let $R_0 = \lambda\rho.\rho$ and $F_0 = \delta$.

$\mathcal{B}_2$

(i) $\mathcal{D}[\![a]\!]\alpha RF = I(a)(\alpha) \rightarrow \{a \cdot RF\}, F$

(ii) $\mathcal{D}[\![m]\!]\alpha RF = \{\star \cdot \mathcal{D}[\![g]\!]\alpha RF\}$ for $m \leftarrow g$ in D

(iii) $\mathcal{D}[\![g_1; g_2]\!]\alpha RF = \mathcal{D}[\![g_1]\!]\alpha(\mathcal{D}[\![g_2]\!]\alpha R)F$

(iv) $\mathcal{D}[\![g_1 \,\square\, g_2]\!]\alpha RF = \mathcal{D}[\![g_1]\!]\alpha R(\mathcal{D}[\![g_2]\!]\alpha RF)$

(v) $\mathcal{D}[\![new(\underline{\hat{\beta}})]\!]\alpha RF = \{\star \cdot (RF \parallel \mathcal{D}[\![body(\underline{\hat{\beta}})]\!]\underline{\beta}R_0F_0)\}$

(vi) $\mathcal{D}[\![\hat{\beta}!m]\!]\alpha RF = \{[\alpha, \hat{\beta}!m] \cdot \rho\}$ with $\rho = \mathcal{D}[\![\hat{\beta}?]\!]\alpha RF$

(vii) $\mathcal{D}[\![[m_1, ..., m_n]]\!]\alpha RF = \{[\alpha, \overline{m_i}] \cdot f_i : i = 1, ..., n\}$
with $f_i = \lambda\beta \in \mathsf{Proc}.RF \rhd_\beta \mathcal{D}[\![m_i; r_{m_i} \,\square\, \overline{r}_\delta]\!]\beta R_0F_0$

(viii) $\mathcal{D}[\![\hat{\beta}?]\!]\alpha RF = \{[\alpha, \beta?] \cdot f\}$
with $f = \lambda c \in \mathsf{Comm}.\mathcal{D}[\![eval(\hat{\beta}?, c)]\!]\alpha RF$

(ix) $\mathcal{D}[\![\overline{r}]\!]\alpha RF = \{[\alpha, \overline{r}] \cdot RF\}$

We leave the example $\mathcal{D}[\![\hat{\beta}!m]\!]\alpha R_0F_0 \parallel \mathcal{D}[\![[m]]\!]\underline{\beta}R_0F_0$ with $D = \{m \leftarrow a_1 \,\square\, a_2\}$ as an exercise to the reader.

Definition 6.5.2 $\mathcal{M}[\![< D \mid g >]\!] = \mathcal{D}[\![g]\!]\alpha_0R_0F_0$ *for some initial process* α_0.

6.5.3 Equivalence between operational and denotational semantics

We augment π (see section 6.4.3) by redefining, for non-empty ρ

$$rem(\rho) = \rho\backslash(\{c \cdot \rho' : c \cdot \rho' \in \rho\} \cup \{c \cdot f : c \cdot f \in \rho\})$$

and we state, in a by now familiar fashion

Theorem 6.5.3 $\mathcal{O}[\![< D \mid g >]\!] = \pi \circ \mathcal{M}[\![< D \mid g >]\!]$

the proof of which is summarized by

$$\begin{aligned}
\mathcal{O}[\![< D \mid g >]\!] &= \mathcal{T}[\![\{< \alpha_0, g; \surd : \Delta >: \Xi\}]\!] \text{ by definition 6.5.1} \\
&= \pi \circ \mathcal{G}[\![\{< \alpha_0, g; \surd; \Delta >: \Xi\}]\!] \text{ by corollary 6.5.7} \\
&= \pi \circ \mathcal{F}[\![g; \surd : \Delta]\!]\alpha_0 \text{ by definition 6.5.4} \\
&= \pi \circ \mathcal{D}[\![g]\!]\alpha_0R_0F_0 \text{ by definition 6.5.4} \\
&= \pi \circ \mathcal{M}[\![< D \mid g >]\!] \text{ by definition 6.5.2}
\end{aligned}$$

In addition to the functions $\mathcal{R}$: SuccCo $\rightarrow$ Succ and $\mathcal{F}$: FailCo $\rightarrow$ Fail we need a function

$$\mathcal{C} : \mathsf{ProcCo} \rightarrow \mathbb{IP}$$

to give meaning to syntactic process continuations. As for $\mathcal{B}_1$ we also have a function $\mathcal{G}$: Conf $\rightarrow$ IP, mapping configurations to processes in IP.

Definition 6.5.4

a. $\mathcal{R}[\![\surd]\!]\alpha = \lambda\rho.\rho$

b. $\mathcal{R}[\![g; R]\!]\alpha = \mathcal{D}[\![g]\!]\alpha\mathcal{R}[\![R]\!]\alpha$

c. $\mathcal{F}[\![\Delta]\!]\alpha = \delta$

d. $\mathcal{F}[\![R : F]\!]\alpha = \mathcal{R}[\![R]\!]\alpha\mathcal{F}[\![F]\!]\alpha$

e. $\mathcal{C}[\![\Xi]\!] = \delta$

f. $\mathcal{C}[\![< \alpha, F >: C]\!] = \mathcal{C}[\![C]\!] \rhd_\alpha \mathcal{F}[\![F]\!]\alpha$

g. $\mathcal{G}[\![\{C_1, ..., C_n\}]\!] = \mathcal{C}[\![C_1]\!] \parallel ... \parallel \mathcal{C}[\![C_n]\!]$

We can state the following property.

Lemma 6.5.5 *if* $\Gamma \longrightarrow \Gamma'$ *then* $\mathcal{G}[\![\Gamma]\!] = \mathcal{G}[\![\Gamma']\!]$

Proof: As an example, let $\Gamma = \{< \alpha, (g_1; g_2); R : F >: C\}$.
We have that $\mathcal{G}[\![\Gamma]\!] = \mathcal{C}[\![C]\!] \rhd_\alpha \mathcal{F}[\![(g_1; g_2); R : F]\!]\alpha =$ (similar as for $\mathcal{B}_1$)
$\mathcal{C}[\![C]\!] \rhd_\alpha \mathcal{F}[\![g_1; (g_2; R) : F]\!]\alpha =$ (according to definition 6.5.4)
$\mathcal{G}[\![\{< \alpha, g_1; (g_2; R) : F >: C\}]\!]$ as desired. □

For proving our equivalence result we take a similar route as for $\mathcal{B}_1$ and characterize $\pi \circ \mathcal{G}$ as follows.

Lemma 6.5.6 $\Psi(\pi \circ \mathcal{G}) = \pi \circ \mathcal{G}$

Proof: We extend the complexity measure given for $\mathcal{B}_1$ to syntactic process continuations by stating $c(< \alpha, F >: C) = c(F) + c(C)$ with $c(\Xi) = 0$. It is easy to establish that the required property of c stating that if $\Gamma \rightarrow \Gamma'$ then $c(\Gamma) > c(\Gamma')$ holds. For instance $c(< \alpha, (g_1; g_2); R : F >: C) = c((g_1; g_2); R : F) + c(C) > c(g_1; (g_2; R) : F) + c(C)$ (by applying c as defined for FailCo) $= c(< \alpha, g_1; (g_2; R) : F >: C)$. Now we must prove for all Γ that $\Psi(\pi \circ \mathcal{G})[\![\Gamma]\!] = \pi \circ \mathcal{G}[\![\Gamma]\!]$. For blocking configurations the result is immediate. For non-blocking configurations we treat the cases corresponding to *Action* and *Result*. Let $\overline{R}_i = \mathcal{R}[\![R_i]\!]\alpha_i$, $\overline{F}_i = \mathcal{F}[\![F_i]\!]\alpha_i$ and $\overline{C}_i = \mathcal{C}[\![C_i]\!]$. Below we will use the fact that $\bigcup\{\eta \circ (\pi\ \rho)\} = \pi\ \{\eta \cdot \rho\}$.

- If $\Gamma = \{< \alpha, a; R : F >: C\}$ then if $I(a)(\alpha)$ is *true* we have
  $\Psi(\pi \circ \mathcal{G})[\![\Gamma]\!] =$ (by the definition of Ψ)
  $\bigcup\{a \circ (\pi \circ \mathcal{G})[\![\{< \alpha, R : F >: C\}]\!]\} =$ (by the property of π)
  $\pi\ \{a \cdot \mathcal{G}[\![\{< \alpha, R : F >: C\}]\!]\} =$ (by definition 6.5.4

$\pi\ \{a \cdot (\mathcal{C}[\![C]\!] \rhd_\alpha \mathcal{F}[\![R : F]\!]\alpha)\} =$ (by the definition of $\rhd_\alpha$)
$\pi\ (\mathcal{C}[\![C]\!] \rhd_\alpha \{a \cdot \mathcal{F}[\![R : F]\!]\alpha\}) =$ (by applying the definition of $\mathcal{D}$ and $\mathcal{F}$)
$\pi\ (\mathcal{C}[\![C]\!] \rhd_\alpha \mathcal{F}[\![a; R : F]\!]) =$
$\pi\ \mathcal{C}[\![< \alpha, a; R : F >: C]\!] =$
$\pi \circ \mathcal{G}[\![\{< \alpha, a; R : F >: C\}]\!]$.
In case $I(a)(\alpha)$ is not *true* we use the induction hypothesis as in the proof for $\mathcal{B}_1$.

- If $\Gamma = \{< \alpha_1, \hat{\alpha}_2?; R_1 : F_1 >: C, < \alpha_2, \overline{r}; R_2; F_2 >: C_2\}$ then $\Psi(\pi \circ \mathcal{G})[\![\Gamma]\!] =$
$\pi\ \{\star \cdot \mathcal{G}[\![\{< \alpha_1, \mathit{eval}(\hat{\alpha}_2?, \overline{r}); R_1 : F_1 >: C_1, < \alpha_2, R_2 : F_2 >: \Xi, C_2\}]\!]\}) =$
(by definition 6.5.4)
$\pi\ \{\star \cdot (\mathcal{C}[\![< \alpha_1, \mathit{eval}(\hat{\alpha}_2?, \overline{r}); R_1 : F' >: C_1]\!] \parallel \mathcal{C}[\![< \alpha_2, R_2 : F_2 >: \Xi]\!] \parallel \overline{C}_2)\}) =$
(taking $f = \lambda\tau.\overline{C}_1 \rhd_{\alpha_1} \mathcal{F}[\![\mathit{eval}(\hat{\alpha}_2?, \tau); R_1 : F_1]\!]\alpha_1$)
$\pi\ \{\star \cdot (f(\overline{r}) \parallel \overline{R}_2\overline{F}_2 \parallel \overline{C}_2)\} =$ (by the definition of communication)
$\pi\ (\{[\alpha_1, \hat{\alpha}_2?] \cdot f\} \parallel \{[\alpha_2, \overline{r}] \cdot (\overline{R}_2\overline{F}_2 \parallel \overline{C}_2)\}) =$
(by the definition $\rhd_{\alpha_1}$ and $\rhd_{\alpha_2}$ with $f' = \lambda\tau.\mathcal{F}[\![\mathit{eval}(\hat{\alpha}_2?, \tau); R_1 : F_1]\!]\alpha_1$)
$\pi\ ((\overline{C}_1 \rhd_{\alpha_1} \{[\alpha_1, \hat{\alpha}_2?] \cdot f'\}) \parallel (\overline{C}_2 \rhd_{\alpha_2} \{[\alpha_2, \overline{r}] \cdot \overline{R}_2\overline{F}_2\})) =$
$\pi\ ((\overline{C}_1 \rhd_{\alpha_1} \mathcal{D}[\![\hat{\alpha}_2?]\!]\alpha_1\overline{R}_1\overline{F}_1) \parallel (\overline{C}_2 \rhd_{\alpha_2} \mathcal{D}[\![\overline{r}]\!]\alpha_2\overline{R}_2\overline{F}_2)) =$
$\pi\ (\mathcal{C}[\![< \alpha_1, \hat{\alpha}_2?; R_1 : F_1 >: C_1]\!] \parallel \mathcal{C}[\![< \alpha_2, \overline{r}; R_2 : F_2 >: C_2]\!]) = \pi \circ \mathcal{G}[\![\Gamma]\!]$ as desired. □

Corollary 6.5.7 $\mathcal{T} = \pi \circ \mathcal{G}$

Chapter 7

An abstract version of DLP and its operational semantics

- The slow return from the fire...
the pause when the bell strikes suddenly again,
the listener on the alert -
Walt Whitman, Leaves of Grass

Now we wish to put some flesh on the skeleton provided in the previous chapter, where we gave the semantics of process creation, communication and backtracking in an abstract, uninterpreted, setting.

As a first step, we will give an operational characterization of an abstract language $\mathcal{L}$ that represents what we consider to be the core of DLP, and that is referred to as DLP_2 in chapter 3. In the next chapter, we will extend the comparative semantics given for the uniform languages $\mathcal{B}_0$, $\mathcal{B}_1$ and $\mathcal{B}_2$ to the comparative semantics for the languages $\mathcal{L}_0$, $\mathcal{L}_1$ and $\mathcal{L}_2$ that represent various subsets of DLP. The operational semantics for the language $\mathcal{L}$ is identical to the operational semantics for $\mathcal{L}_2$ given there. The reader, not interested in a more intuitive account of the operational semantics of DLP_2 is invited to skip this chapter and to continue with the comparative semantics for DLP, in chapter 8.

The DLP language is an extension of Prolog with *objects* and *special forms* for dealing with non-logical variables, object creation, and engaging in a rendezvous. Objects in DLP are module-like entities with private non-logical (instance) variables and methods that have access to these variables in a protected way.

Methods are defined by clauses that are written as Prolog clauses; these clauses may however contain as goals the special forms mentioned. We will treat here the subset DLP_2 , covering the special forms listed below. An important restriction of DLP_2 is that *accept* goals may occur only in the constructor process.

For reasons of syntactic clarity we no longer overload the equality symbol but introduce separate symbols for goals having side-effects, the assignment to non-logical variables and the creation of objects and processes. The special forms include:

- $v := t$ - to assign (the value of) the term t to the non-logical variable v
- $O :: new(c(t))$ - to create a new instance of the object c and start the constructor process evaluating the goal $c(t)$, O thereafter refers to the newly created object
- $O!m(t)$ - to engage in a rendez-vous with the object to which O refers, for evaluating the method call $m(t)$
- $accept(m_1, ..., m_n)$ - to state the willingness to accept methods $m_1, ..., m_n$

When assigning a term t to a non-logical variable, all non-logical variables in the term are first replaced by their values. The same applies when unifying two terms.

In particular we will concentrate on the issues of object creation and the communication occurring in a rendez-vous. Instances of objects are created by using the special form $O :: new(c(t))$ that results in binding O to the newly created instance of object c. When creating the object the constructor process evaluating the goal $c(t)$ is started for it. A rendez-vous is initiated by a method call of the form $O!m(t)$, with O bound to an object. For a rendez-vous to take place the object must state its willingness to answer a method call by an accept statement of the form $accept(m_1, ..., m_n)$.

The distinguishing feature of DLP, compared with other approaches, is the possible occurrence of distributed backtracking in a rendez-vous. In the absence of distributed backtracking we may identify an object with a process, since an object then is either executing its own activity or evaluating a method call. Backtracking however requires to keep administration of what part of the search space has been explored. A notion of processes, separate from objects, is thus motivated by the need to keep record of this information. Processes are created for each method call to enable backtracking over the answers.

The intent of this chapter is to provide with a more precise definition of objects and processes, and a description of their behavior during a rendez-vous. Before going into any details, however, we will briefly repeat the informal account given of objects, processes and how backtracking may interact with the use of non-logical variables.

7.1 Objects, processes and backtracking

Objects serve as a means for modularization and provide protection for (local) data. Non-logical variables may be used to store the data encapsulated by an ob-

ject. The clauses defined for an object act as methods, in that they have exclusive access to these data.

An object declaration in DLP contains the names of the non-logical variables of (instances of) the object and a definition of both the constructor clauses and method clauses. The clauses are written in standard Prolog format, but may contain as goals the special forms listed previously. A clause is considered a constructor clause when the predicate name of the head of the clause is identical to the name of the declared object. A method for an object is defined by all clauses having that method name as the predicate name of their head.

When a new object is created, by a goal of the form $O :: new(c(t))$, then this newly created object contains a copy of the non-logical variables of the declared object and also (conceptually) a copy of all its clauses. We will consider only active objects, that is objects with own activity arising from evaluating the constructor for that object. Since inheritance is effected by copying the non-logical variables and clauses from the inherited objects, for giving a semantic characterization of the interpretation of DLP programs it suffices to look at DLP without inheritance.

Processes are created on the occasion of creating a new object, and when requesting a rendez-vous. When a new object is created, the constructor process, defined by clauses for the predicate with the name of the object, is started. When a rendez-vous is requested by a method call, a process is started for evaluating the method call. Creating a new process for evaluating such a goal is necessitated by our approach to distributed backtracking. Both the constructor process and the processes started for evaluating a goal by an object are said to refer to that object.

Backtracking may occur when multiple answer substitutions result from evaluating a method call. As an example, a method call to the object declared below may generate an indefinite number of solutions.

```
object nat {
nat() :- accept(num), nat().
num(0).
num(s(X)) :- num(X).
}
```

The constructor clause for the object states that any call to num will be accepted.

Evaluating the goal

```
:- O::new(nat()), O!num(X), write(X), fail.
```

results in printing all natural numbers, eventually. The process created for evaluating num(X) backtracks each time that backtracking over O!num(X) is tried due to the occurrence of fail in the goal.

In the example above, multiple processes could safely be active for the object simultaneously. Obviously, in the presence of non-logical variables, protection is needed from concurrently changing the value of a such a variable, by disallowing method calls to be simultaneously active. To what extent however do we wish to

guarantee such mutual exclusion? In dealing with the semantics of our language we proceed from the assumption that any state change due to an assignment to a non-logical variable may safely occur before the first answer. For backtracking over the remaining answers, the state may be fixed in a non-logical variable in order to avoid interference from other method calls.[1] We thus allow multiple processes referring to a single object to backtrack concurrently over answer substitutions generated in a rendez-vous, since by a decision of design, we do not wish to exclude other processes from having a method call evaluated by that object, any longer than until the first answer substitution or failure has been delivered.

Related to the issue of mutual exclusion is the question whether in backtracking any assignment made to non-logical variables must be undone. Again, as a matter of design, we have decided that assignments to non-logical variables are permanent. Non-logical variables were introduced for storing persistent data, shared by all processes referring to an object. We may observe moreover that restoring a state may be programmed, using additional non-logical variables.[2] From the point of view of a semantic description, neither choice seems to present any serious difficulties.

7.2 The language $\mathcal{L}$

The language $\mathcal{L}$ that we will define here extends the language $\mathcal{B}_2$ to allow an interpretation of the actions and the primitives for process creation and communication.

A special feature of our abstract language is, apart from the special forms for object creation and communication, the occurrence of a goal for unbinding variables, that may have become bound previously. Goals to unbind variables occur in what we have called method declarations and are also inserted to allow backtracking over alternative solutions in a rendez-vous. We have chosen for introducing undo actions to avoid the use of stacks of substitutions recording the bindings on backtrack points, that, although it is perhaps conceptually more elegant, burdens the notation considerably.

Terms are either empty, variables (logical and non-logical) or function terms built from a function symbol and zero or more terms. Constants are considered zero-ary function terms. As special constants we introduce a set of object names Objname, with typical elements c, and a set of method names Method, with typical elements m. We let Objname $\subset$ Method. Furthermore, we assume to have a set Var of logical variables, with typical elements denoted by capitals X, O and Q; a set Nlvar of non-logical variables, with typical elements v; and a set Term of terms, with typical elements t, constructed according to

$$t ::= \varepsilon \mid X \mid v \mid f(t_1, ..., t_n) : f \text{ a function symbol, } n \geq 0$$

[1] See the discussion in section 2.4.

[2] We motivate this decision in section 5.2.1, when discussing the design perspectives.

The set of function symbols include the set of object names and method names. Constants are zero-ary functions. When $t = \varepsilon$ we write $f()$ instead of $f(\varepsilon)$.

Actions $a \in$ Action may take one of the following forms.

$$a ::= u(t) \mid v := t \mid t_1 = t_2$$

Elementary actions a can be goals of the form $u(t)$, for undoing the bindings of the logical variables in a term t, non-logical variable assignments of the form $v := t$, or unification goals of the form $t_1 = t_2$. Occasionally, we will use an action *fail*, that always fails.

Primitives $e \in$ Prim are needed for object creation and engaging in a rendezvous. These primitives, by which $\mathcal{L}$ extends Prolog, are defined as

$$e ::= O :: new(c(t)) \mid O!m(t) \mid accept(m_1, ..., m_n) \mid Q? \mid !t$$

where m ranges over the set of method names, and c over the set of object names. Goals of the form $O :: new(c(t))$ are used for creating a new object. A goal of the form $O!m(t)$ is used for calling a method $m(t)$ for the object denoted by O. The accept goal $accept(m_1, ..., m_n)$ indicates the willingness to accept a method call for m_i, $i = 1, ..., n$. As a restriction to the use of accept goals we require that accept goals may occur in the constructor process only. Goals of the from $Q?$ are used to request the answers resulting from the evaluation of a goal. Finally, a goal of the form $!t$ is used to send the result of evaluating a method call to the invoking process.

Goals $g \in$ Goal are given by

$$g ::= a \mid m(t) \mid g_1; g_2 \mid g_1 \square g_2 \mid e$$

Goals of the form $m(t)$ are used for a kind of (recursive) procedure call. These goals correspond to both local method goals and constructor goals in DLP_2 . For convenience we assume that calls $m(t)$ have a single argument t, with t possibly equal to ε. The goals $g_1; g_2$ and $g_1 \square g_2$ stand for, respectively, sequential composition and alternative composition of goals g_1 and g_2. Dynamically, alternative composition amounts to backtracking, in a sequential fashion.

Method declarations take the form $m \leftarrow b$ where $b \equiv \lambda\tau.g$ for some g is called the body of m.
As has been explained in our description of DLP, a method definition for an object consists of a number of clauses, that differ from common Prolog clauses only by the possible occurrence of the special forms by which DLP extends Prolog. We show how such clauses correspond to a method declaration in our language $\mathcal{L}$. A predicate definition of the form The clauses for a predicate appear as components

of an alternative composition, that we will also call disjuncts.[3] The goals $\tau = t_i$ take care of unifying the parameters of the clause with the arguments of the call. We note that in a DLP clause only the actions $v := t$, and $t_1 = t_2$ may occur. Moreover only the primitives $O :: new(c(t))$, $O!m(t)$ and $accept(m_1, ..., m_n)$ are allowed. The communication of answers resulting in a rendez-vous is modeled by the dynamic insertion of goals of the form $Q?$ for requesting an answer, and $!t$ for sending a result. These goals are hidden from the user of the language.

We assume that the components of the alternative composition share no logical variables. However, since these alternatives will be executed sequentially, the unbinding of the logical variables of the goal must be taken care of. For this purpose we insert the action $u(\tau)$ in each disjunct, starting with the second, in order to undo any binding resulting from the evaluation of a previous disjunct.

Object declarations are 3-tuples of the form

$$(c \leftarrow b_c, < v_i >_{i=1,...,r}, < m_i \leftarrow b_i >_{i=1,...,k})$$

containing the *name* c of an object, its *constructor* $c \leftarrow b_c$, a possibly empty sequence of r *non-logical variables* and a possibly empty sequence of k *methods*, that is groups of clauses that define a predicate. The constructor $c \leftarrow b_c$ may be called in the same way as a method.

Programs are tuples of the form $< D \mid g >$, where D is a collection of object declarations and g a goal statement.

7.3 Configurations

Evaluating a program $< D|g >$ may result in a collection of processes concurrently active in evaluating their goal, and occasionally entering in communication with other processes. In defining the possible computation steps that such a collection of processes may take, we use a syntactic representation of this collection together with a state on which these processes operate, for instance by changing the value of a variable.

Representing objects and processes An *object* is a tuple (c, n) containing the *name* c of a declared object and its *instance number* n, indicating that it is the n'th instance of c. Hence we define the set of objects as

$$\text{Obj} = \{(c, n) : c \text{ an object name}, n \in \mathbb{N}\}$$

A *process* is a tuple (c, n, k) containing the *name* and *instance number* of the object it refers to, and a counter k that uniquely identifies the k'th process referring to the object (c, n). We define the set of processes, with typical elements α and β, by

[3] The translation given here applies also to the languages $\mathcal{L}_0$, $\mathcal{L}_1$ and $\mathcal{L}_2$ treated in the next chapter. The only significant difference with the Prolog clause format is the explicit use of an alternative construct of the form $g_1 \square g_2$ to model the behavior of alternative clauses for a predicate, and the insertion of undo actions for unbinding variables.

$$\mathsf{Proc} = \{(c, n, k) : (c, n) \in \mathsf{Obj}, k \in \mathbb{N}\}$$

With each object (c, n) is associated a process $(c, n, 0)$ executing the constructor $b_c(t)$ for some t. The other processes referring to the object (c, n), created for answering a method call, are represented by (c, n, k), with $k \geq 1$.

In the sequel we will identify the constructor process $(c, n, 0)$ with the object (c, n) to which it refers. In particular, the request for a rendez-vous addressed at the process $(c, n, 0)$ will be interpreted as a method call for the object (c, n). To access the object to which a process refers we define a function *obj* by $obj(c, n, k) = (c, n)$.

We have introduced α, β as typical elements of the set Proc of process identifiers. We introduce no syntactic variables for objects, but will use the function *obj* for which it holds that $obj(\alpha) \in \mathsf{Obj}$ when $\alpha \in \mathsf{Proc}$.

Further we extend the set of terms by redefining t as

$$t ::= \varepsilon \mid \delta \mid X \mid v \mid \alpha \mid f(t_1, ..., t_n) : f \text{ a function symbol}, n \geq 0$$

where δ is a special term to be used for communicating the absence of answers for a method call. The extension also allows process identifiers to occur as terms.

Logical variables may become bound to terms. The binding of logical variables is recorded in a substitution of type $\mathsf{Subst} = \mathsf{Var} \rightarrow \mathsf{Term}$. We will write $t\theta$ to denote the term t modified by applying the substitution θ, where we assume that θ is extended to terms in a standard fashion. We say that a substitution is a unifier of two terms if the two terms become syntactically equal after applying the substitution. When the terms involved are process identifiers this means that they must be pointing to the same process. A unifier θ is more general than a unifier θ' if there is a substitution ξ such that $\xi \circ \theta = \theta'$. It is a well-known fact that if a unifier exists for two terms, then there exists also a most general unifier. Most general unifiers are unique, up to renaming variables. We assume to have a function $mgu : \mathsf{Term} \times \mathsf{Term} \rightarrow \mathsf{Subst}$ that computes an idempotent most general unifier of two terms, if it exists. Composition of unifiers $\xi \circ \theta$ is defined by $t(\xi \circ \theta) = (t\theta)\xi$.

States The actions of our language operate on states. States contain the dynamic information that must be kept during the execution of a program, such as the bindings of logical variables due to the evaluation of a goal by a process, the values of the non-logical variables of each object and global information concerning the objects and processes that are in use.

We define states $\sigma \in \Sigma$ as triples (θ, s, i) containing respectively a substitution for each process, a store for the values of the non-logical variables for each object, and a *count* function for keeping record of the number of instances of an object and the like.

The **substitution** function θ has type $\mathsf{Proc} \rightarrow \mathsf{Subst}$; by θ_α we denote the substitution that delivers the bindings of the variables for process α.

The **store** function s has type $\mathsf{Obj} \rightarrow \mathsf{Store}$, with $\mathsf{Store} = \mathsf{NIvar} \rightarrow \mathsf{Term}$. Similarly $s_{obj(\alpha)}$ is the function that delivers the values of the non-logical variables of the object to which α refers. The non-logical variables of an object (c, n) are thus shared by all processes (c, n, k) referring to that object.

In order to be able to replace the occurrence of a non-logical variable in a term by its value we interprete $s_{obj(\alpha)}$ as to be extended to terms.

The **count** function i is used to take care of a number of administrative details. It consists of a function of type Objname $\rightarrow$ IN, that is used to store for each declared object c its current instance number n. It contains moreover a function of type Obj $\rightarrow$ IN that gives for each object (c, n) the number of processes created for answering a method call. Also i contains a function of type Proc$\times$Method $\rightarrow$ IN, that is needed for renaming logical variables. It keeps count of the number of times a method m is called (locally) for process α. This number, called the invocation depth of m in α, is represented by $i(\alpha, m)$ and is updated with each call of the form $m(t)$. More precisely, i is the union of these functions, and is of type

$$\begin{array}{l}\mathsf{Objname} \rightarrow \mathbb{N} \ \cup \\ \mathsf{Obj} \rightarrow \mathbb{N} \ \cup \\ \mathsf{Proc} \times \mathsf{Method} \rightarrow \mathbb{N}\end{array}$$

A renaming function is a function of type Var $\rightarrow$ Var that substitutes (possibly renamed) variables for variables, when applied to a term or a goal. We assume to have a function $\nu \in$ Proc $\times$ Method $\times$ IN $\rightarrow$ Var $\rightarrow$ Var that given a process name, a method name and a natural number indicating the invocation depth, delivers a renaming function working on variables. Such renaming functions are extended to terms in a natural way. Applying a renaming function $\nu_{\alpha mi(\alpha,m)}$ results in renaming the variables of a term as shown in the example $f(a, X)\nu_{\alpha mi(\alpha,m)}$ that gives $f(a, X_{(\alpha,m,n)})$, when $i(\alpha, m) = n$. On the level of the implementation we assume to have an injective mapping from indexes of the form (α, m, n) to storage locations for logical variables. Since renaming is typically applied to the body of a method we define $b\nu_{\alpha mi(\alpha,m)} = \lambda\tau.g\nu_{\alpha mi(\alpha,m)}$ for $b = \lambda\tau.g$. Obviously, the renaming does not affect the parameter τ.

Interpretation The effect of actions on states can be described by defining a function effect : Proc $\rightarrow$ Goal $\rightarrow \Sigma \rightarrow \Sigma$. As an abbreviation we write $\sigma(a)(\alpha)$ for effect$(\alpha)(a)(\sigma)$, which denotes the state resulting from executing a for α in state σ. The effect function is partial.

To model the state changes we define function variants of a function f by

$$f\{x/v\}(y) = \begin{cases} v & \text{if } x \equiv y \\ f(y) & \text{otherwise} \end{cases}$$

As abbreviations we define

$$\begin{array}{l}\theta\{\alpha/X \leftarrow t\} = \theta\{\alpha/\theta_\alpha\{X/t\}\}, \text{ and} \\ s\{obj(\alpha)/v \leftarrow t\} = s\{obj(\alpha)/s_{obj(\alpha)}\{v/t\}\}\end{array}$$

denoting, respectively, the modification of the substitution θ_α by binding X to t, and the modification of the store for the object to which α refers by setting the value of the non-logical variable v to t.

An example of the function effect is given by the interpretation of the goal $u(t)$ that is introduced to undo the bindings of variables.

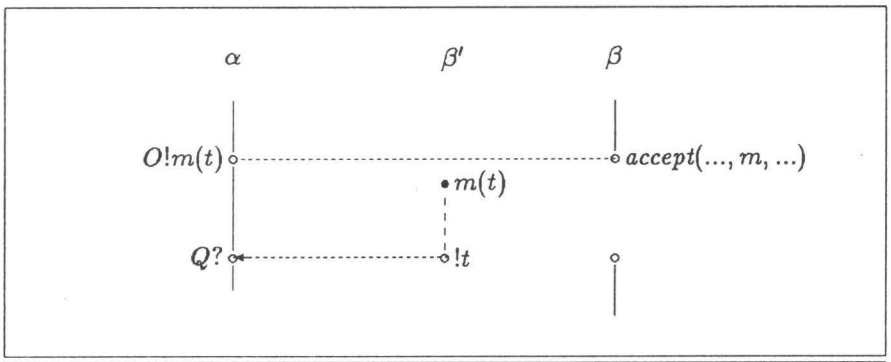
- $\sigma(u(t))(\alpha) = (\theta', s, i)$ for $\sigma = (\theta, s, i)$ with $\theta' = \theta\{\alpha/X \leftarrow X\}_{X \in varsof(t)}$

thereby defining the effect of evaluating the goal as undoing the possible bindings of the variables occurring in the term t, by changing θ_α to behave as the identity for these variables.

Communication by rendez-vous We have introduced the major ingredients of our operational semantics. Executing a program will be modeled by a collection of processes, each evaluating a goal, and possibly modifying the state when executing an action.

Communication between processes is restricted to a synchronous rendez-vous as occurs in calling a method. Since backtracking over the resulting answers may occur, we decompose a synchronous rendez-vous in a method call of the form $O!m(t)$, with O assumed to be bound to the identifier for the constructor process of an object, and a (dynamically inserted) request for the answer substitutions resulting from evaluating $m(t)$, of the form $Q?$, with Q bound to (a pointer to) the process that evaluates $m(t)$. By an answer substitution we mean the substitution computed in finding a solution for the goal $m(t)$ restricted to the unbound variables occurring in t.

In the figure below we picture the communication occurring in a rendez-vous.



We assume that O is bound to $\beta = (c, n, 0)$, the constructor process for the object (c, n). Process $\beta' = (c, n, k)$, for some k, is a variant of β in that $obj(\beta) = obj(\beta')$. In accepting the call Q becomes bound to β'.

As soon as the call $O!m(t)$, with O bound to β, is accepted α inserts the goal $Q?$, that is a request for the answer substitutions resulting from the evaluation of $m(t)$. The process β', which is a variant of the process β in that they refer to the same object, starts evaluating $m(t)$. Process α must wait until the process β' has computed an answer substitution. When the first answer substitution is computed, it is sent to α; and the goal $Q?$ succeeds. The process β, which is the constructor process $(c, n, 0)$ for the object $obj(\beta) = (c, n)$, may then proceed with the evaluation of the constructor goal. When backtracking occurs in α, the goal $Q?$ succeeds as many times as there are alternative answer substitutions produced by β'. Backtracking in β' runs concurrently with the activity of process β.

A successful rendez-vous consists of two communication steps. The step that initiates the rendez-vous is represented by the pair

$$(O!m(t),\ accept(..., m, ...))$$

containing both the call $O!m(t)$ and the accept goal $accept(..., m, ...)$, stating the willingness to accept the method m.

For returning the results we will use the pair

$$(Q?, !t)$$

composed of the request for an answer $Q?$, and the goal $!t$, that indicates that a solution has been found.

To model the effect of a communication on a particular state we extend the effect function by a function of type $\mathsf{Proc} \times \mathsf{Proc} \rightarrow \mathsf{Prim} \times \mathsf{Prim} \rightarrow \Sigma \rightarrow \Sigma$, for which we use the obvious abbreviation $\sigma(e_1, e_2)(\alpha, \beta)$ for $e_1, e_2 \in \mathsf{Prim}$.

Syntactic continuations are used to model the backtracking behavior of a program. We introduce three types of syntactic continuations.

The first type of continuations are called *success continuations* $R \in \mathsf{SuccCo}$, defined by

$$R ::= \surd \mid g; R$$

which correspond to the sequential evaluation of a goal, without backtracking.

We use *failure continuations* $F \in \mathsf{FailCo}$, defined by

$$F ::= \Delta \mid R : F$$

to model the backtracking local to a process. Intuitively, a failure continuations may be regarded as a stack of success continuations, with Δ the empty stack.

To model distributed backtracking we define *process continuations* $C \in \mathsf{ProcCo}$ by

$$C ::= \Xi \mid < \alpha, F >: C$$

which allows to create a stack of processes of which the process on top is considered to be active. The empty process continuation Ξ may be regarded as the empty stack.

Finally, we may define configurations (Γ, σ), as consisting of a set $\Gamma \in \mathcal{P}(\mathsf{ProcCo})$ of process continuations and a state $\sigma \in \Sigma$.

7.4 Transition rules

The transition system below, describing the computation rules for evaluating a goal g given a declaration D, consists of axioms of the form $(\Gamma, \sigma) \xrightarrow{\eta} (\Gamma', \sigma')$ meaning that the configuration (Γ, σ) may be taken to (Γ', σ'), while displaying label η. The label η may be empty or an element of the set $\Lambda = \mathsf{Action} \cup \{\star\}$. In case the label is empty we also speak of unlabeled transitions. Although we make no further use of these labels here, we may comment that the empty label represents invisible

behavior. The special label $\star$ represents visible but in a sense silent behavior, that cannot further be inspected. Labels from Action merely represent the action being executed.

An axiom may be applied only if the effect function used is defined for its arguments. We indicate the state modified by the effect function by $\sigma^\star$. In other words $\sigma^\star$ abbreviates $\sigma(g)(\alpha)$ or $\sigma(e_1, e_2)(\alpha, \beta)$. The axiom for Action is conditional, in that whenever $\sigma^\star$ is defined the first transition must be applied, and if $\sigma^\star$ is not defined then the alternative transition must be applied.

We assume to have a general rule of the form

$$(\Gamma, \sigma) \xrightarrow{\eta} (\Gamma', \sigma') \Longrightarrow (X \cup \Gamma, \sigma) \xrightarrow{\eta} (X \cup \Gamma', \sigma')$$

for $X \in \mathcal{P}(\mathsf{ProcCo})$ disjoint from Γ and Γ'.

Let $\sigma = (\theta, s, i)$.

Actions can be either goals of the form $u(t)$ to undo the bindings of logical variables, non-logical variable assignments of the form $v := t$, or unification goals of the form $t_1 = t_2$. In both the latter goals the terms t, t_1, t_2 are simplified using $s_{obj(\alpha)}$ in order to replace non-logical variables by their values.

$\mathcal{L}$

$$(\{< \alpha, a; R : F >: C\}, \sigma) \xrightarrow{a} (\{< \alpha, R : F >: C\}, \sigma^\star) \qquad \textit{Action}$$
$$\longrightarrow (\{< \alpha, F >: C\}, \sigma)$$

with for $a \in \{u(t), v := t, t_1 = t_2\}$ the interpretation

- $\sigma(u(t))(\alpha) = (\theta', s, i)$
  where $\theta' = \theta\{\alpha/X \leftarrow X\}_{X \in varsof(t)}$
- $\sigma(v := t)(\alpha) = (\theta, s', i)$
  where $s' = s\{obj(\alpha)/v \leftarrow s_{obj(\alpha)}(t)\theta_\alpha\}$
- $\sigma(t_1 = t_2)(\alpha) = (\theta', s, i)$
  where
  $\theta' = \theta\{\alpha/mgu(s_{obj(\alpha)}(t_1)\theta_\alpha, s_{obj(\alpha)}(t_2)\theta_\alpha) \circ \theta_\alpha\}$

Local method calls amount to replacing calls of the form $m(t)$ by the goal $b(t)$ when $m \leftarrow b$ occurs as a method for c in D, assuming that $\alpha \equiv (c, n, k)$. Before inserting the body as a goal, the renaming function $\nu_{ami(\alpha,m)}$ must be applied.

When a local method call is executed the argument is instantiated with the current substitution. Any variable that is then unbound may receive a binding due to evaluating the body of the method. Hence when disjuncts of a body are executed sequentially, any binding resulting from the evaluation of a previous disjunct must be undone. Recall that in our representation of method declarations we have inserted undo statements for unbinding these variables. Since the argument is instantiated, only the variables that were unbound are visible as variables. All other variables are replaced by their instantiation and hence not affected by the undo statements.

$(\{< \alpha, m(t); R : F >: C\}, \sigma)$ *Rec*

$\xrightarrow{\star} (\{< \alpha, b\nu_{ami(\alpha,m)}(t\theta_\alpha); R : F >: C\}, \sigma^\star)$

for $m \leftarrow b$ in D for the object of α, with the interpretation

- $\sigma(m(t))(\alpha) = (\theta, s, i\{(\alpha, m)/i(\alpha, m) + 1\})$

Sequential composition and backtracking are defined as in [de Bakker, 1988]. A sequential statement is decomposed to enable to look at the first component.

When an alternative statement is encountered, a new success continuation containing the first alternative is put on top of the failure stack. The part that has to follow after a successful execution of either statement remains the same for both statements.

$(\{< \alpha, (g_1; g_2); R : F >: C\}, \sigma) \longrightarrow (\{< \alpha, g_1; (g_2; R) : F >: C\}, \sigma)$ *Seq*

$(\{< \alpha, (g_1 \square g_2); R : F >: C\}, \sigma)$ *Alt*

$\longrightarrow (\{< \alpha, (g_1; R) : (g_2; R) : F >: C\}, \sigma)$

New statements give rise to the execution of a new process executing the constructor for the newly created object. As an effect of executing a new statement the state is modified with respect to the counter for the number of copies of a named object. In the interpretation given below we have omitted the initialization of non-logical variables, since we assume that the constructor process takes care of this.

$(\{< \alpha, O :: new(c(t)); R : F >: C\}, \sigma)$ *New*

$\xrightarrow{\star} (\{< \alpha, R : F >: C, < \beta, F_\beta >: \Xi\}, \sigma^\star)$

where $F_\beta \equiv c(s_{obj(\alpha)}(t)\theta_\alpha); \surd : \Delta$ with the interpretation

- $\sigma(O :: new(c(t)))(\alpha) = (\theta', s, i')$
  where $\theta' = \theta\{\alpha/O \leftarrow \beta\}$ with $\beta = (c, i'(c), 0)$
  and $i' = i\{c/i(c) + 1\}$

Method calls result in creating a new item on top of the process stack of the object that receives the request. Accordingly the state is changed by raising the process counter for the object. The newly created process β' evaluates the goal $m(t'); !t' \square !\delta$. The sequence $m(t'); !t'$ represents the evaluation of the method call $m(t')$ followed by the goal $!t'$ to communicate the results to the invoking process. The alternative $!\delta$ will be evaluated when there are no more solutions for $m(t')$.

In the invoking process, the call $O!m(t)$ is replaced by the goal $Q?$, with Q a fresh variable bound to β', to request the answers that result from evaluating the call. The system will block when the method is not defined for that object.

$(\{< \alpha, O!m(t); R_1 : F_1 >: C_1,$ *Method*

$< \beta, accept(..., m, ...); R_2 : F_2 >: C_2\}, \sigma)$

$\xrightarrow{\star} (\{< \alpha, Q?; R_1 : F_1 >: C_1,$

$< \beta', F_{\beta'} >:< \beta, R_2 : F_2 >: C_2\}, \sigma^\star)$

if $\beta = O\theta_\alpha$, where $F_{\beta'} \equiv (m(t'); !t' \,\square\, !\delta); \surd : \Delta$ for $t' = s_{obj(\alpha)}(t)\theta_\alpha$

with the interpretation

- $\sigma(O!m(t), accept(..., m, ...))(\alpha, \beta) = (\theta', s, i')$
  where for $\beta = (c, n, 0)$ and Q a fresh variable it holds
  that $\beta' = (c, n, i'(c, n))$, $i' = i\{(c, n)/i(c, n) + 1\}$
  and $\theta' = \theta\{\alpha/Q \leftarrow \beta'\}$

Results are communicated back to the process that requested the rendez-vous, by stating the goal $Q?$, on occurrence of the goal $!t$ in the process evaluating the call. A failure arises in the process that requested the result when no more answers can be generated, that is when $t = \delta$. Otherwise, the request is replaced by the goal $t\theta_\alpha = t\theta_\beta \,\square\, u(t\theta_\alpha); Q?$. The first component $t\theta_\alpha = t\theta_\beta$ effects the resulting answer substitution in the context of the process that requested the answer.

To allow backtracking over multiple answers the second component contains a repeated request for an answer. The variable bindings of t that result from a previous answer must however first be undone, as expressed by the inclusion of the statement $u(t\theta_\alpha)$.

A similar argument as given for *Rec* holds for the occurrence of the undo action in the axiom *Result*. Only the variables that are unbound at the moment of requesting the rendez-vous are affected by the undo action inserted before the repeated request for an answer substitution. These variables may receive a binding due to evaluating the method call, and hence must be freed from this binding when looking for alternative solutions.

$(\{< \alpha, Q?; R_1 : F_1 >: C_1, < \beta, !t; R_2 : F_2 >: C_2\}, \sigma)$ *Result*

$\xrightarrow{\star} (\{< \alpha, F_1 >: C_1, < \beta, R_2 : F_2 >: \Xi, C_2\}, \sigma)$ if $t = \delta$

$\xrightarrow{\star} (\{< \alpha, (t\theta_\alpha = t\theta_\beta \,\square\, u(t\theta_\alpha); Q?); R_1 : F_1 >: C_1,$

$< \beta, R_2 : F_2 >: \Xi, C_2\}, \sigma)$

provided that $Q\theta_\alpha = \beta$

An operational semantics, in terms of the observable behavior of a program, may now readily be defined as the set of possible transition sequences. We will pursue this issue in the following chapter, where we give both an operational and denotational semantics for various subsets of DLP.

7.5 An example

To give some feeling for what occurs in the execution of a program we present (a part of) the steps taken by the program computing all natural numbers.

```
object nat {
nat() :- accept(num), nat().
num(0).
num(s(X)) :- num(X).
}
```

As a goal we state

```
:- O :: new(nat()), O!num(Y), fail.
```

Translating this program into the language $\mathcal{L}$ gives the declaration

$$D = \{< nat \leftarrow \lambda t.(accept(num); nat()),$$
$$< \varepsilon >,$$
$$< num \leftarrow \lambda t.(t = 0 \ \square \ (u(t); t = s(X); num(X))) >$$
$$>\}$$

where the component $< \varepsilon >$ indicates the absence of non-logical variables.
Our goal translates to

$$O :: new(nat()); O!num(Y); fail$$

and we may make the derivation below, assuming that *fail* always fails.
We will omit applications of the axiom *Seq*, since this merely amounts to reshuffling the brackets.
Let $\sigma = (\theta, s, i)$ and $\sigma_k = (\theta_k, s_k, i_k)$ for $k = 1, 2, ...$
Starting in an initial state σ, with α the process evaluating the goal, we have

1. $(\{ < \alpha, new(nat()); O!num(Y); Q?; fail; \surd : \Delta >: \Xi \}, \sigma)$

   Applying *New* leads to

2. $\xrightarrow{*} (\{ < \alpha, O!num(Y); fail; \surd : \Delta >: \Xi,$
   $< \beta, nat(); \surd : \Delta >: \Xi$
   $\}, \sigma_1)$

with $\sigma_1 = (\theta\{\alpha/O \leftarrow \beta\}, s, i\{nat/i(nat)+1\})$.
Here $\beta = (nat, 1, 0)$, and the instance counter for *nat* is increased by one.
By expanding $nat()$, using the axiom *Rec*, we arrive at

3. $\xrightarrow{\star}$ ({ $< \alpha, O!num(Y); fail; \surd : \Delta >: \Xi$,
$< \beta, accept(num); nat(); \surd : \Delta >: \Xi$
}, σ_2)

with $\sigma_2 = (\theta_1, s_1, i_1\{(\beta, nat)/i(\beta, nat)+1\})$.
The increase of the count for (β, nat) is due to the renumbering of variables, which is irrelevant in this case.
Now we may apply *Method*, which results in

4. $\xrightarrow{\star}$ ({ $< \alpha, Q?; fail; \surd : \Delta >: \Xi$,
$< \beta', (num(Y); !Y \,\square\, !\delta); \surd : \Delta >:$
$< \beta, nat(); \surd : \Delta >: \Xi$
}, σ_3)

with $\sigma_3 = (\theta_2\{\alpha/Q \leftarrow \beta'\}, s_2, i_2\{(nat, n)/i_2(nat, n)+1)\})$,
for $\beta = (nat, n, 0)$, thus binding Q to $\beta' = (nat, n, 1)$.
Notice that the process $< \beta', \ldots >$ is put on the process stack containing $< \beta, \ldots >$ and Ξ.
Applying *Alt* for β' gives

5. $\longrightarrow$ ({ $< \alpha, Q?; fail; \surd : \Delta >: \Xi$,
$< \beta', num(Y); !Y; \surd : (!\delta; \surd) : \Delta >:$
$< \beta, nat(); \surd : \Delta >: \Xi$
}, σ_3)

Applying the axiom *Rec* for β', to call $num(Y)$, gives

6. $\xrightarrow{\star}$ ({ $< \alpha, Q?; fail; \surd : \Delta >: \Xi$,
$< \beta', (Y = 0 \,\square\, (u(Y); Y = s(X'); num(X'))); !Y; \surd : (!\delta; \surd) : \Delta >:$
$< \beta, nat(); \surd : \Delta >: \Xi$
}, σ_4)

with $\sigma_4 = (\theta_3, s_3, i_3\{(\beta', num)/i_3(\beta', num)+1\})$.
We use X' to denote the renamed variable X.
The variable, X' is the variable X indexed by $(\beta', num, i_3(\beta', num))$.
By applying *Alt* for β' we get

7. $\longrightarrow$ ({ $< \alpha, Q?; fail; \surd : \Delta >: \Xi$,
$< \beta', (Y = 0; !Y; \surd) : (u(Y); Y = s(X'); num(X'); !Y; \surd) : (!\delta; \surd) : \Delta >:$

$$< \beta, nat(); \surd : \Delta >: \Xi$$
$$\}, \sigma_4)$$

which allows us to apply *Action* for β', giving

8. $\xrightarrow{Y=0}$ ({ $< \alpha, Q?; fail; \surd : \Delta >: \Xi$,
$< \beta', !Y; \surd : (u(Y); Y = s(X'); num(X'); !Y; \surd) : (!\delta; \surd) : \Delta >:$
$< \beta, nat(); \surd : \Delta >: \Xi$
$\}, \sigma_5)$

with $\sigma_5 = (\theta_4\{\beta'/Y \leftarrow 0\}, s_4, i_4)$.
Now β' is ready to send its first answer substitution.
Applying *Result* gives

9. $\xrightarrow{\star}$ ({ $< \alpha, (Y = 0 \,\square\, u(Y); Q?); fail; \surd) : \Delta >: \Xi$,
$< \beta', (u(Y); Y = s(X'); num(X'); !Y; \surd) : (!\delta; \surd) : \Delta >: \Xi$,
$< \beta, nat(); \surd : \Delta >: \Xi$
$\}, \sigma_5)$

From now on β is free to accept requests for a rendez-vous again and β' may continue to generate further answer substitutions, indefinitely. Applying *Alt* for α and *Action* for β' leads to

10. $\longrightarrow \ldots \xrightarrow{u(Y)}$ ({ $< \alpha, (Y = 0; fail; \surd) : (u(Y); Q?; fail; \surd) : \Delta >: \Xi$,
$< \beta', (u(Y); Y = s(X'); num(X'); !Y; \surd) : (!\delta; \surd) : \Delta >: \Xi$,
$< \beta, nat(); \surd : \Delta >: \Xi$
$\}, \sigma_6)$

with $\sigma_6 = (\theta_5\{\beta/Y \leftarrow Y\}, s_5, i_5)$.
Note that the substitution value of Y in β has already been substituted.
Applying *Action* for α gives

11. $\xrightarrow{Y=0}$ ({ $< \alpha, fail; \surd : (u(Y); Q?; fail; \surd) : \Delta >: \Xi$,
$< \beta', (Y = s(X'); num(X'); !Y; \surd) : (!\delta; \surd) : \Delta >: \Xi$,
$< \beta, nat(); \surd : \Delta >: \Xi$
$\}, \sigma_7)$

with $\sigma_7 = (\theta_6\{\alpha/Y \leftarrow 0\}, s_6, i_6)$.

Since *fail* is assumed to fail, applying *Action* results in popping the failure stack for α. Then the variable Y must be unbound both for process α. When we moreover evaluate $Y = s(X')$ for β' we arrive at the configuration

12. $\longrightarrow$ ({ $< \alpha, Q?; fail; \surd : \Delta >: \Xi$,
$< \beta', num(X'); !Y; \surd : (!\delta; \surd) : \Delta >: \Xi$,
$< \beta, nat(); \surd : \Delta >: \Xi$
}, σ_8)

with $\sigma_8 = (\theta_7\{\alpha/Y \leftarrow s(X')\}, s_7, i_7)$, that resembles the configuration having state σ_3 except for the binding of Y to $s(X')$.

Chapter 8

Comparative semantics for DLP

- We see therefore at first the picture as a whole with its individual parts still more or less kept in the background; we observe the movements, transitions, connections, rather than the things that move, combine and are connected. -
Friedrich Engels

In chapter 6 we have given the comparative semantics for the languages $\mathcal{B}_0$, $\mathcal{B}_1$ and $\mathcal{B}_2$, three uniform abstract languages with backtracking, with special attention to process creation and communication. We now introduce the languages $\mathcal{L}_0$, $\mathcal{L}_1$ and $\mathcal{L}_2$, that may be regarded as augmenting $\mathcal{B}_0$, $\mathcal{B}_1$ and $\mathcal{B}_2$ with the details necessary to model the actual (logic) programming aspects of DLP, such as the assignment to non-logical variables, parameter passing and unification. The languages $\mathcal{L}_0$, $\mathcal{L}_1$ and $\mathcal{L}_2$ correspond in an obvious way to the subsets of DLP distinguished in chapter 3 and listed in the table below.

DLP_0 = *Prolog* + *non-logical variables*		
non-logical variables	$v := t$	assigns the value of the term t to the non-logical variable v
DLP_1 = DLP_0 + *object creation* + *communication over channels*		
object creation	$new(c(t))$	creates an active instance of object c
channels	$C :: new(channel)$	creates a new channel
	$C!t$	output statement for term t over channel C
	$C?t$	input statement for term t over channel C
DLP_2 = DLP_0 + *object creation* + *method call by rendez-vous*		
object creation	$O :: new(c(t))$	creates an active instance of object c
rendez-vous	$O!m(t)$	calls the method $m(t)$ for O
	$accept(m_1, ..., m_n)$	accept methods $m_1, ..., m_n$

One subset, called $\mathcal{L}_0$ covers an abstraction of the base language Prolog, and represents a simple logic programming language with backtracking but without cut. The language $\mathcal{L}_0$ extends a Prolog-like language by non-logical variables that may be assigned terms as values, in an imperative way. Our treatment may be regarded as a warming-up, preparing the way for a semantic treatment of the remaining subsets.

The second subset $\mathcal{L}_1$ covers dynamic object creation and communication over channels. For $\mathcal{L}_1$ objects are identical to processes, since each object is active only with executing its so-called constructor process. The semantics given for this sublanguage extends the semantics for $\mathcal{L}_0$ with features for dealing with process creation and communication. A provision had to be made for the backtracking that may occur in an attempt at communication, when selecting an appropriate input statement. As for $\mathcal{L}_0$, we give both an operational semantics and a denotational semantics. The equivalence proof relating them extends the proof given for $\mathcal{L}_0$ in a rather straightforward way.

Our third subset $\mathcal{L}_2$ covers, apart from dynamic object creation, also method call by rendez-vous. In order to deal with the distributed backtracking that may occur when a method call gives rise to multiple answer substitutions we had to make an explicit distinction between objects and processes, and as a consequence we had to invent an appropriate semantic operator that relates the meaning of processes created for handling a rendez-vous to the meaning of the process executing the own activity of the object.

8.1 Backtracking

The first language $\mathcal{L}_0$ that we deal with is a simple sequential logic programming language, without cut. Modeling the base language of DLP, it provides a starting point for the treatment of backtracking, access to non-logical variables and unification.

The language $\mathcal{L}_0$ extends the language $\mathcal{B}_0$ by introducing terms. Actions, involving terms, may be interpreted as undoing variable bindings, assigning a term t to a non-logical variable, or the unification of two terms. Another difference with $\mathcal{B}_0$ is that recursive procedure calls now take a term as a parameter. Correspondingly we now use procedure declarations of the form $p \leftarrow \lambda\tau.g$ to represent the clauses for a predicate p. The only significant difference with the Prolog clause format is the explicit use of an alternative construct and the insertion of undo actions for unbinding variables. We have chosen to employ undo actions for undoing bindings on backtracking for reasons of notational convenience. See also section 7.2.

Syntax As *constants* we assume to have a set Procname of procedure names, with typical elements p. For *variables* we will distinguish between logical variables $X \in$ Var and non-logical variables $v \in$ Nlvar. Now we can define terms comprising the set Term, with typical elements t, by

$$t ::= \varepsilon \mid X \mid v \mid f(t_1, ..., t_n) : f \text{ a function symbol, } n \geq 0$$

We will write $f()$ for terms of the form $f(\varepsilon)$.

As *actions* we define elements $a \in$ Action by

$$a ::= u(t) \mid v := t \mid t_1 = t_2$$

representing, respectively, undoing variable bindings, the assignment of a term t to the non-logical variable v, and the unification of terms t_1 and t_2. In the sequel we also assume to have an action *fail*.

Now *goals* $g \in$ Goal are given by

$$g ::= a \mid p(t) \mid g_1; g_2 \mid g_1 \,\square\, g_2$$

where a is an action, $p(t)$ a procedure call to p with argument t, $g_1; g_2$ the sequential composition of goals g_1 and g_2, and $g_1 \,\square\, g_2$ the alternative composition of these goals.

The clauses defining a predicate are given as *procedure declarations* of the form $p \leftarrow b$, where p is a predicate name and b of the form $\lambda\tau.g$, for g a goal, which is called the body of a procedure declaration.

A *program* is a tuple $< D \mid g >$, with D a collection of procedure declarations and g a goal statement

8.1.1 Operational semantics

We will describe the computation steps taken by the system when evaluating a program in $\mathcal{L}_0$ by means of a *transition system*, that specifies for each kind of *configuration* (which is more or less the state of a system in execution) the steps, if any, by which to reach a subsequent configuration.

A *substitution* $\theta \in \mathsf{Subst} = \mathsf{Var} \to \mathsf{Term}$ is given as a function from (logical) variables to terms. Applying a substitution to a term amounts to replacing all logical variables by their substitution values. In the sequel we assume to have a function $mgu : \mathsf{Term} \times \mathsf{Term} \to \mathsf{Subst}$ that computes an idempotent most general unifier of two terms, if it exists.

We also employ a *store* $s \in \mathsf{Store} = \mathsf{NIvar} \to \mathsf{Term}$, which is a function assigning terms to non-logical variables. Applying a store to a term results in replacing all non-logical variables by their values.

For renaming logical variables we need a *count* $i \in \mathsf{Count} = \mathsf{Procname} \to \mathbb{N}$, that assigns a natural number to a procedure name.

As part of a configuration we use *states* Σ, with typical elements σ, defined by

$$\Sigma = \{(\theta, s, i) \mid \theta \in \mathsf{Subst}, s \in \mathsf{Store}, i \in \mathsf{Count}\}$$

to record the substitution, store and count at a certain point in the computation. We will apply the function variant notation for changing states, as in $f\{x/v\}$ which for an argument y has as value v if $x = y$ and $f(y)$ otherwise.

Variable renaming is effected by a function $\nu \in \mathsf{Procname} \times \mathbb{N} \to \mathsf{Var} \to \mathsf{Var}$, that takes a procedure name, a natural number (given by the count for this procedure name) and a variable for which it delivers a fresh variable. See section 7.3. Applying a renaming $\nu_{pi(p)}$ is illustrated for example by $f(a, X)\nu_{pi(p)} = f(a, X_{(p,i(p))})$, where a new variable X with index $(p, i(p))$ is created. Renaming the body of a procedure declaration is defined by $b\nu_{pi(p)} = \lambda\tau.g\nu_{pi(p)}$, for $b = \lambda\tau.g$.

To model how a state is modified due to the evaluation of a goal we define an *effect function* $\mathsf{effect} : \mathsf{Goal} \times \Sigma \to \Sigma$, by equations of the form $\sigma(a) = \sigma'$, where $\sigma(a)$ abbreviates $\mathsf{effect}(a)(\sigma)$. The function effect is partial.

The operational semantics of a program is given in terms of the observable behavior displayed when executing a program. For this reason we have labeled the transition rules with *labels* from a set

$$\Lambda = \mathsf{Action} \cup \{\star\}.$$

The label $\star$ is used to indicate that on expanding a procedure call with its body, a computation step is made that we cannot further inspect. We consider all actions a visible in the sense that they may be observed by an external observer. The empty label, denoted by the empty word ε, is used to indicate invisible behavior and is usually omitted. We take

$$\Lambda^\infty = \Lambda^\star \cup \Lambda^\omega$$

as the set of strings over Λ and use a concatenation operator $\circ : \Lambda^\infty \times \Lambda^\infty \to \Lambda^\infty$ defined by $w_1 \circ w_2 = w_1 w_2$ if w_1 is of finite length and w_1 otherwise. We extend this operator to sets of strings over Λ by defining

$$w \circ X = \{w \circ x : x \in X\}$$

We moreover require that $\varepsilon \circ w = w = w \circ \varepsilon$.

Syntactic continuations are used in order to specify the backtracking behavior of a program.
We distinguish between *success continuations* $R \in \mathsf{SuccCo}$ defined by

$$R ::= \surd \mid g; R$$

with $\surd$ the empty success continuation, denoting success, and $g; R$ the sequential composition of a goal and a success continuation; and *failure continuations* $F \in \mathsf{FailCo}$ given by

$$F ::= \Delta \mid R : F$$

with Δ the empty failure continuation, denoting failure and $R : F$ representing a stack with a success continuation on top, modeling in a sense the alternative composition of goals.

Syntactic continuations of this kind are also used in [de Vink, 1989], although there the substitution resulting from the computation thus far is part of the success continuation. Since later on we have to deal with dynamic object and process creation, we have a (global) state parameter in our configurations to represent the substitution and have introduced an auxiliary action for undoing the bindings of logical variables.

Transition rules specifying the behavior of the system, are given as axioms of the form $(\Gamma, \sigma) \xrightarrow{\eta} (\Gamma', \sigma')$, meaning that the configuration (Γ, σ) may be taken to the configuration (Γ', σ') while the label η is displayed. The label η may be empty. Transitions with the empty label ε will also be called unlabeled transitions. A configuration $(\Gamma, \sigma) \in \mathsf{Conf}$ consists of a (syntactic) failure continuation Γ and a state σ. The axioms specifying the possible computation steps, for a particular declaration D define the relation $\longrightarrow \subset \mathsf{Conf} \times \Lambda \times \mathsf{Conf}$. A *transition system* for D then is a triple $(\mathsf{Conf}, \Lambda, \longrightarrow)$.

In the rules below, we have used the notation $\sigma^\star$ to indicate the state σ modified by the effect function, in other words $\sigma^\star$ abbreviates $\sigma(g)$ for a goal g. We allow for a branching behavior of the computation to the extent that whenever the function effect is defined then the step in which the effect function occurs must be taken. Only if the function effect is undefined the alternative step may be taken. Such a branching behavior is exemplified in the rule *Action*.

Let $\sigma = (\theta, s, i)$.

$\mathcal{L}_0$

$$(a; R : F, \sigma) \xrightarrow{a} (R : F, \sigma^\star) \qquad \textit{Action}$$
$$\longrightarrow (F, \sigma)$$

with for $a \in \{u(t), v := t, t_1 = t_2\}$ the interpretation

- $\sigma(u(t)) = (\theta', s, i)$ where $\theta' = \theta\{X/X\}_{X \in varsof(t)}$
- $\sigma(v := t) = (\theta, s', i)$ where $s' = s\{v/s(t)\theta\}$
- $\sigma(t_1 = t_2) = (\theta', s, i)$ if $\theta' = mgu(s(t_1)\theta, s(t_2)\theta) \circ \theta$

$(p(t); R : F, \sigma) \xrightarrow{\star} (b\nu_{pi(p)}(t\theta); R : F, \sigma^{\star})$ for $p \leftarrow b$ in D *Rec*

with the interpretation

- $\sigma(p(t)) = (\theta, s, i\{p/i(p)+1\})$

$((g_1; g_2); R : F, \sigma) \longrightarrow (g_1; (g_2; R) : F, \sigma)$ *Seq*

$((g_1 \,\square\, g_2); R : F, \sigma) \longrightarrow ((g_1; R) : (g_2; R) : F, \sigma)$ *Alt*

$(\surd : F, \sigma) \longrightarrow (F, \sigma)$ *Tick*

The axioms for this simple language are quite obvious. A brief comment must suffice. Actions may be executed when the effect function $\sigma(a)$ is defined, otherwise failure arises and the success continuation R on top of the stack is popped to enable the execution of the remaining failure continuation F. The interpretation of the various actions is straightforward. For instance $\sigma(u(t))$ delivers the state σ where all (logical) variables occurring in t have become unbound. The effect of a unification statement $t_1 = t_2$ consists of modifying the substitution component of the state by the most general unifier of (the instantiated versions of) t_1 and t_2. Expanding a procedure call requires to rename the variables occurring in the body of the procedure by a renaming function set by the procedure name and its current invocation depth as stored in the *count* parameter of the state. The count for p must accordingly be increased by one. The axioms for *Action* and *Rec* are the only axioms with a non-empty label. Having a non-empty label intuitively denotes a non-trivial computation step, that takes some time to perform. The axiom for sequential composition simply states that the compound goal $g_1; g_2$ is decomposed so that the goal g_1 may be executed with g_2 part of the success continuation. The axiom for alternative composition specifies that to execute $g_1 \,\square\, g_2$, the goal g_1 must be executed while storing g_2 as an alternative. A new success continuation is created for this purpose and inserted in the failure stack. On failure, backtracking over g_2 may occur.

Computation sequences consist of the steps that represent observable behavior. We define a relation $\xrightarrow{\eta}\!\!\!\!\!\rightarrow$, for η non-empty, by stating that $(\Gamma, \sigma) \xrightarrow{\eta}\!\!\!\!\!\rightarrow (\Gamma', \sigma')$ whenever there is a transition with label η preceded by zero or more unlabeled transitions. We will say that (Γ, σ) *blocks* if there is no label η and configuration (Γ', σ') for which $(\Gamma, \sigma) \xrightarrow{\eta}\!\!\!\!\!\rightarrow (\Gamma', \sigma')$.

As our result domain, for characterizing the behavior of a program in $\mathcal{L}_0$, we take

$$\mathbb{R} = \Sigma \rightarrow \Lambda^{\infty}$$

and define the *operational semantics* of a program $< D \mid g >$ by the function $\mathcal{O} : \mathcal{L}_0 \to \mathbb{R}$ as below.

Definition 8.1.1 $\mathcal{O}[\![< D \mid g >]\!] = \mathcal{T}[\![g; \surd : \Delta]\!]$ *where for* $\Psi : (\mathsf{FailCo} \to \mathbb{R}) \to (\mathsf{FailCo} \to \mathbb{R})$ *and*

$$\Psi(\phi)[\![\Gamma]\!](\sigma) = \begin{cases} \varepsilon & \text{if } (\Gamma, \sigma) \text{ blocks} \\ \eta \circ \phi[\![\Gamma']\!](\sigma') \text{ if } (\Gamma, \sigma) \xrightarrow{\eta} (\Gamma', \sigma') & \text{otherwise} \end{cases}$$

we let $\mathcal{T} = \mathrm{fix}\Psi$.

8.1.2 Denotational semantics

For $\mathcal{L}_0$ we may take $\mathbb{R}$, with typical elements ρ, as a domain. To model the backtracking behavior we will employ semantic continuations. We use failure continuations $F \in \mathsf{Fail}$, with $\mathsf{Fail} = \mathbb{R}$. Semantic success continuations $R \in \mathsf{Succ} = \mathsf{Fail} \to \mathbb{R}$ take a failure continuation as an argument to compute the resulting behavior. We define the semantic function $\mathcal{D}[\![\cdot]\!] : \mathsf{Goal} \to \mathsf{Succ} \to \mathsf{Fail} \to \mathbb{R}$ by the equations below. In the equations for $\mathcal{D}[\![\cdot]\!]$ we use the interpretation function as defined for the transition system. We again assume that $\sigma = (\theta, s, i)$.

$\mathcal{L}_0$

(i) $\mathcal{D}[\![a]\!]RF = \lambda\sigma.\sigma(a)\,\mathit{defined} \to a \circ RF\sigma(a), F\sigma$

(ii) $\mathcal{D}[\![p(t)]\!]RF = \lambda\sigma. \star \circ \mathcal{D}[\![b\nu_{pi(p)}(t\theta)]\!]RF\sigma(p(t))$ for $p \leftarrow b$ in D

(iii) $\mathcal{D}[\![g_1; g_2]\!]RF = \mathcal{D}[\![g_1]\!](D[\![g_2]\!]R)F$

(iv) $\mathcal{D}[\![g_1 \,\square\, g_2]\!]RF = \mathcal{D}[\![g_1]\!]R(D[\![g_2]\!]RF)$

The conditional form of the axiom *Action* is reflected in the conditional expression on the right hand side of the equation for actions a, abbreviating *if* $\sigma(a)$ *is defined then* $a \circ R(F)(\sigma(a))$ *else* $F\sigma$. Note the similarity with the definition in section 6.3.2.

Remark: As we have explained in the chapter introducing metric semantics (section 6.2.2), we may define a higher order mapping $\Phi : (\mathsf{Goal} \to \mathsf{Succ} \to \mathsf{Fail} \to \mathbb{R}) \to (\mathsf{Goal} \to \mathsf{Succ} \to \mathsf{Fail} \to \mathbb{R})$ and state that $\mathcal{D} = \mathrm{fix}\Phi$, in order to establish that $\mathcal{D}[\![\cdot]\!]$ is well-defined.

We may now define the meaning of a program $< D \mid g >$.

Definition 8.1.2 $\mathcal{M}[\![< D \mid g >]\!] = \mathcal{D}[\![g]\!]R_0F_0$ *with* $R_0 = \lambda\rho.\rho$ *and* $F_0 = \lambda\sigma.\varepsilon$.

8.1.3 Equivalence between operational and denotational semantics

For a program $< D \mid g >$ the operational semantics characterizes the observable behavior of a program. A denotational semantics gives, in a compositional way, mathematical meaning to the constructs of the language. We wish both characterizations to be equivalent.

Theorem 8.1.3 $\mathcal{O}[\![< D \mid g >]\!] = \mathcal{M}[\![< D \mid g >]\!]$

We will present an outline of the proof and fill in the details later.

$\mathcal{O}[\![< D \mid g >]\!] = \mathcal{T}[\![g; \surd : \Delta]\!]$ by definition 8.1.1

$= \mathcal{F}[\![g; \surd : \Delta]\!]$ by corollary 8.1.7

$= \mathcal{D}[\![g]\!] R_0 F_0$ by definition 8.1.4

$= \mathcal{M}[\![< D \mid g >]\!]$ by definition 8.1.2

In the proof we make use of an auxiliary function $\mathcal{F}$: FailCo $\rightarrow$ Fail that maps syntactic failure continuations to semantic failure continuations. Also we will make use of a function $\mathcal{R}$: SuccCo $\rightarrow$ Succ for mapping syntactic success continuations to semantic success continuations.

Definition 8.1.4

a. $\mathcal{R}[\![\surd]\!] = \lambda\rho.\rho$

b. $\mathcal{R}[\![g; R]\!] = \mathcal{D}[\![g]\!]\mathcal{R}[\![R]\!]$

c. $\mathcal{F}[\![\Delta]\!] = \lambda\sigma.\varepsilon$

d. $\mathcal{F}[\![R : F]\!] = \mathcal{R}[\![R]\!]\mathcal{F}[\![F]\!]$

Now we can state the following property.

Lemma 8.1.5 *if* $(F, \sigma) \longrightarrow (F', \sigma)$ *then* $\mathcal{F}[\![F]\!]\sigma = \mathcal{F}[\![F']\!]\sigma$

The key step in the proof that $\mathcal{O} = \mathcal{M}$ consists of showing that the function $\mathcal{F}$ is a fixed point of the operator that characterizes the operational meaning of a program.

Lemma 8.1.6 $\Psi(\mathcal{F}) = \mathcal{F}$

Proof: We introduce a complexity measure c on $\Gamma \in$ FailCo such that whenever $(\Gamma, \sigma) \longrightarrow (\Gamma', \sigma)$ we have that $c(\Gamma) > c(\Gamma')$, by defining

$c(a) = c(p(t)) = c(\surd) = 1,$
$c(g_1; g_2) = c(g_1 \,\square\, g_2) = c(g_1) + c(g_2) + 1,$
$c(\Delta) = 0$ and $c(g; R : F) = c(g) + c(F).$

We must prove for each Γ and σ that $\Psi(\mathcal{F})[\![\Gamma]\!]\sigma = \mathcal{F}[\![\Gamma]\!]\sigma$. If (Γ, σ) blocks then the result is clear. Assume that Γ is of the form $g; R : F$ and that for some η we have $(\Gamma, \sigma) \xrightarrow{\eta}\!\!\!\!\rightarrow (\Gamma', \sigma')$. The proof proceeds by a case analysis on g. We will treat the case that $g \equiv a$.

- If $g \equiv a$ we have by the definition of Ψ if $\sigma(a)$ is defined that
  $\Psi(\mathcal{F})[\![a; R : F]\!]\sigma = a \circ \mathcal{F}[\![R; F]\!]\sigma(a) =$ (by applying definition 8.1.4)
  $\mathcal{F}[\![a; R : F]\!]\sigma$.
  If $\sigma(a)$ is not defined then $(a; R : F, \sigma) \longrightarrow (F, \sigma)$ and the result follows from the induction-hypothesis, stating that for Γ' satisfying $c(\Gamma) > c(\Gamma')$ it holds that $\Psi(\mathcal{F})[\![\Gamma']\!]\sigma = \mathcal{F}[\![\Gamma']\!]\sigma$.

Corollary 8.1.7 $\mathcal{T} = \mathcal{F}$

8.2 Dynamic object creation and communication over channels

The language $\mathcal{L}_1$ extends $\mathcal{L}_0$ by providing constructs for dynamic object creation and communication over channels. It extends $\mathcal{B}_1$, apart from the introduction of terms, by the occurrence of primitives for the creation of channels and the explicit use of channel names.

A program is organized as a collection of object declarations. For the language $\mathcal{L}_1$, creating an instance of an object means creating a process (that executes the *constructor* for that object). Hence, we may identify objects and processes. Apart from objects, an indefinite number of channels may be created. Point to point connections exist when two processes have access to the same channel. Communication over channels allows a limited form of backtracking, namely the input side is allowed to backtrack until an input term is found that is unifiable with the output term.

Syntax As *constants* we need a set of object names Objname, with typical elements c, and a set of procedure names $p \in$ Procname. We assume that each object has a constructor, and hence that Objname $\subset$ Procname.

Terms and actions are as for $\mathcal{L}_0$. For logical variables that will become bound to a channel we will use by convention a capital C.

The *primitives* $e \in$ Prim by which $\mathcal{L}_1$ extends $\mathcal{L}_0$, are defined by

$$e ::= C :: new(channel) \mid new(c(t)) \mid C!t \mid C?t$$

In addition to $\mathcal{L}_0$ we thus have a statement to create a new channel, a statement to create a new (active) object, and the communication statements $C!t$ for output and $C?t$ for input.

The set of *goals* $g \in$ Goal is given by

$$g ::= a \mid p(t) \mid g_1; g_2 \mid g_1 \,\square\, g_2 \mid e$$

which differs from that given for $\mathcal{L}_0$ only by the extensions.

The clauses for a predicate p are, as for $\mathcal{L}_0$, collected in *procedure declarations* of the form $p \leftarrow b$, with b of the form $\lambda\tau.g$.

We assume to have *object declarations* of the form

$$(c \leftarrow b_c, < v_i >_{i=1,...,r}, < p_i \leftarrow b_i >_{i=1,...,k})$$

for named objects c, with b_c defining the constructor for c, v_i the non-logical variables of c, and $p_i \leftarrow b_i$ the procedure declarations corresponding to the clauses of the object c.

A *program* $< D \mid g >$ consists of a collection of object declarations D and a goal g.

8.2.1 Operational semantics

Since the language $\mathcal{L}_1$ allows for dynamic object creation the transition system that specifies how to execute a program must deal with sets of objects, rather than a single object as for $\mathcal{L}_0$. Moreover, we need to distinguish between two distinct objects even if they show the same behavior.

To identify *objects* it suffices to have an object name and its instance number. We define the set Obj, with typical elements α and β by

$$\mathsf{Obj} = \{(c, n) \mid c \text{ an object name}, n \in \mathbb{N}\}$$

For an object $\alpha = (c, n)$ we will write D_α to denote the object declaration for c. We will use $\gamma \in \mathsf{Channel} = \mathbb{N}$ for referring to channels.

Similarly as for $\mathcal{L}_0$ we extend the set of terms by redefining

$$t ::= \varepsilon \mid X \mid v \mid \gamma \mid f(t_1, ..., t_n) : f \text{ a function symbol}, n \geq 0$$

where we introduce terms γ to be able to bind variables to channels.

Since the state of a configuration no longer concerns a single (sequential) process but a collection of processes executing in parallel, we use generalized *substitutions* $\theta \in \mathsf{Subst}^+ = \mathsf{Obj} \rightarrow \mathsf{Subst}$ that depend on an object $\alpha \in \mathsf{Obj}$, and we let $\theta_\alpha \in \mathsf{Subst}$ denote a substitution that belongs to a particular object α.

Similarly, we generalize the notion of a *store* $s \in \mathsf{Store}^+ = \mathsf{Obj} \rightarrow \mathsf{Store}$, and let $s_\alpha \in \mathsf{Store}$ denote the store belonging to the object α.

We must keep a *count* $i \in \mathsf{Count}^+ = (\mathsf{Objname} \cup \{ch\} \cup \mathsf{Obj} \times \mathsf{Procname}) \rightarrow \mathbb{N}$, for administrative reasons. The count function may be split into a function from $\mathsf{Objname} \rightarrow \mathbb{N}$, to keep track of the number of instances of a particular object, a function from $\{ch\} \rightarrow \mathbb{N}$, to generate new channels, and a function from $\mathsf{Obj} \times \mathsf{Procname} \rightarrow \mathbb{N}$ to be used in renaming variables.

The states σ that we use in the configurations come from the set

$$\Sigma = \{(\theta, s, i) \mid \theta \in \mathsf{Subst}^+, s \in \mathsf{Store}^+, i \in \mathsf{Count}^+\}.$$

For modifying states we use function variants (again) and introduce the abbreviations

$$\theta\{\alpha/X \leftarrow t\} = \theta\{\alpha/\theta_\alpha\{X/t\}\}$$
$$s\{\alpha/v \leftarrow t\} = s\{\alpha/s_\alpha\{v/t\}\}$$

For *renaming* variables a function $\nu \in \mathsf{Obj} \times \mathsf{Procname} \times \mathbb{N} \to \mathsf{Var} \to \mathsf{Var}$ is used, which generalizes the renaming function introduced in the previous section, by the Obj parameter. A typical use of this function is illustrated by $b\nu_{api(\alpha,p)}$ which renames the variables in b according to the triple $(\alpha, p, i(\alpha, p))$.

To describe the effect of evaluating goals on states we generalize the *effect function* given previously to $\mathsf{effect} : \mathsf{Obj} \to \mathsf{Goal} \to \Sigma \to \Sigma$ and we abbreviate $\mathsf{effect}(\alpha)(g)(\sigma)$ as $\sigma(g)(\alpha)$. We must however also extend the effect function to (matching) pairs of communication intentions, by a function of type $\mathsf{Obj} \times \mathsf{Obj} \to \mathsf{C} \times \mathsf{C} \to \Sigma \to \Sigma$, with $\mathsf{C} = \{\gamma!t, \gamma?t : \gamma \in \mathsf{Channel}, t \in \mathsf{Term}\}$, and write for this $\sigma(c, \bar{c})(\alpha, \beta)$, for $c, \bar{c} \in \mathsf{C}$. However, we are only interested in the special cases having the form $\sigma(\gamma!t_1, \gamma?t_2)(\alpha, \beta)$. As before the effect function is partial, and thereby influences the decision which axiom must be applied.

We use, just as before, success continuations from SuccCo given by $R ::= \surd \mid g; R$ and failure continuations from FailCo given by $F ::= \Delta \mid R : F$.

The transition rules below specify labeled transitions of the form $(\Gamma, \sigma) \xrightarrow{\eta} (\Gamma', \sigma')$, with η empty or an element of $\Lambda = \mathsf{Action} \cup \{\star\}$. In contrast to the previous section, Γ represents a collection of objects, so $\Gamma \in \mathcal{P}(\mathsf{Obj} \times \mathsf{FailCo})$, the set consisting of all subsets of pairs of object identifiers and failure continuations.

We assume to have a general rule of the form

$$(\Gamma, \sigma) \xrightarrow{\eta} (\Gamma', \sigma') \Longrightarrow (X \cup \Gamma, \sigma) \xrightarrow{\eta} (X \cup \Gamma', \sigma')$$

for $X \in \mathcal{P}(\mathsf{Obj} \times \mathsf{FailCo})$ disjoint from Γ and Γ'. Let $\sigma = (\theta, s, i)$.

$\mathcal{L}_1$

$$(\{< \alpha, a; R : F >\}, \sigma) \xrightarrow{a} (\{< \alpha, R : F >\}, \sigma^\star) \qquad \textit{Action}$$

$$\longrightarrow (\{< \alpha, F >\}, \sigma)$$

with for $a \in \{u(t), v := t, t_1 = t_2\}$ the interpretation

- $\sigma(u(t))(\alpha) = (\theta', s, i)$
  where $\theta' = \theta\{\alpha/X \leftarrow X\}_{X \in varsof(t)}$
- $\sigma(v := t)(\alpha) = (\theta, s', i)$
  where $s' = s\{\alpha/v \leftarrow s_\alpha(t)\theta_\alpha\}$
- $\sigma(t_1 = t_2)(\alpha) = (\theta', s, i)$
  where $\theta' = \theta\{\alpha/mgu(s_\alpha(t_1)\theta_\alpha, s_\alpha(t_2)\theta_\alpha) \circ \theta_\alpha\}$

$$(\{< \alpha, p(t); R : F >\}, \sigma) \xrightarrow{\star} (\{< \alpha, b\nu_{api(\alpha,p)}(t\theta_\alpha); R : F >\}, \sigma^\star) \qquad \textit{Rec}$$

for $p \leftarrow b$ in D_α, with the interpretation

- $\sigma(p(t))(\alpha) = (\theta, s, i\{(\alpha, p)/i(\alpha, p) + 1\})$

$$(\{< \alpha, (g_1; g_2); R : F >\}, \sigma) \longrightarrow (\{< \alpha, g_1; (g_2; R) : F >\}, \sigma) \qquad \textit{Seq}$$

$$(\{< \alpha, (g_1 \,\square\, g_2); R : F >\}, \sigma) \longrightarrow (\{< \alpha, (g_1; R) : (g_2; R) : F >\}, \sigma) \qquad \textit{Alt}$$

$$(\{< \alpha, C :: new(channel); R : F >\}, \sigma) \xrightarrow{\star} (\{< \alpha, R : F >\}, \sigma^\star) \qquad \textit{Chann}$$

with the interpretation

- $\sigma(C :: new(channel))(\alpha) = (\theta', s, i')$
  where $\theta' = \theta\{\alpha/C \leftarrow \gamma\}$, $\gamma = i'(ch)$
  and $i' = i\{ch/i(ch) + 1\}$

$$(\{< \alpha, new(c(t)); R : F >\}, \sigma) \xrightarrow{\star} (\{< \alpha, R : F >, < \beta, F_\beta >\}, \sigma^\star) \qquad \textit{New}$$

where $F_\beta \equiv c(s_\alpha(t)\theta_\alpha); \surd : \Delta$, with the interpretation

- $\sigma(new(c(t)))(\alpha) = (\theta, s, i')$
  where $\beta = (c, i'(c), 0)$ and $i' = i\{c/i(c) + 1\}$

$$(\{< \alpha, C_1!t_1; R_1 : F_1 >, < \beta, C_2?t_2; R_2 : F_2 >\}, \sigma) \qquad \textit{Comm}$$

$$\xrightarrow{\star} (\{< \alpha, R_1 : F_1 >, < \beta, R_2 : F_2 >\}, \sigma^\star)$$

$$\xrightarrow{\star} (\{< \alpha, C_1!t_1; R_1 : F_1 >, < \beta, F_2 >\}, \sigma)$$

if $C_1\theta_\alpha = \gamma = C_2\theta_\beta$, with the interpretation

- $\sigma(\gamma!t_1, \gamma?t_2)(\alpha, \beta) = (\theta', s, i)$
  where $\theta' = \theta\{\alpha/\zeta \circ \theta_\alpha, \beta/\zeta \circ \theta_\beta\}$ for $\zeta = mgu(t_1\theta_\alpha, t_2\theta_\beta)$

$$(\{< \alpha, \surd : F >\}, \sigma) \longrightarrow (\{< \alpha, F >\}, \sigma) \qquad \textit{Tick}$$

The transition rules for actions, recursive procedure call, sequential composition and alternative composition are to a large extent similar to the corresponding rules for $\mathcal{L}_0$, but for the generalization to collections of objects. New are the axioms for the creation of channels and objects, and the axioms for communication. Only the axioms for action and communication are conditional. Which alternative to choose depends on whether $\sigma^\star$ (which is the state that results from applying the effect function) is defined. Communication succeeds if the input term is unifiable with the output term, in this case both processes proceed; otherwise the input side fails and starts backtracking. The condition that both processes refer to the same channel is necessary, since if this condition is not satisfied neither transition may be taken.

Since the behavior of programs in $\mathcal{L}_1$ is indeterminate, because objects may be active concurrently, we take

$$\mathbb{R} = \Sigma \to \mathcal{P}_{nc}(\Lambda^\infty)$$

and define $\mathcal{O} : \mathcal{L}_1 \to \mathbb{R}$, giving the operational semantics of a program $< D \mid g >$.

Definition 8.2.1 $\mathcal{O}[\![< D \mid g >]\!] = \mathcal{T}[\![\{< \alpha_0, g; \surd : \Delta >\}]\!]$ *for an initial object* α_0, *where for* $\Psi : (\mathcal{P}(\mathsf{Obj} \times \mathsf{FailCo}) \to \mathbb{R}) \to (\mathcal{P}(\mathsf{Obj} \times \mathsf{FailCo}) \to \mathbb{R})$ *and*

$$\Psi(\phi)[\![\Gamma]\!](\sigma) = \begin{cases} \{\varepsilon\} & \text{if } (\Gamma, \sigma) \text{ blocks} \\ \bigcup\{\eta \circ \phi[\![\Gamma']\!](\sigma') : (\Gamma, \sigma) \xrightarrow{\eta} (\Gamma', \sigma')\} & \text{otherwise} \end{cases}$$

we define $\mathcal{T} = \mathsf{fix}\Psi$.

8.2.2 Denotational semantics

As a domain, reflecting the additional constructs for communication, we take a branching structure $\mathbb{P}$, with typical elements ρ, given by

$$\mathbb{P} \cong \{\delta\} \cup \Sigma \to \mathcal{P}_{co}(\Sigma \times \mathbb{P} \ \cup \ \mathsf{C} \times (T \to \mathbb{P}))$$

where $T = \{true, false\}$ and $\mathsf{C} = \{\gamma!t, \gamma?t : \gamma \in \mathsf{Channel}, t \in \mathsf{Term}\}$. The set $\rho(\sigma)$ for some σ, may contain apart from computation steps of the form $\sigma' \cdot \rho'$ also communication intentions of the form $c \cdot f$, with $c \in \mathsf{C}$. Functions $f \in T \to \mathbb{P}$ are of the form $\lambda\tau.\rho$. Such a function takes a truth value to deliver a resumption from $\mathbb{P}$. Our domain equation differs from the one given in section 6.4.2 in the component representing the communication intentions. Since we are now dealing with the non-uniform case, we are not able to use the more elegant syntactic solution provided there. We will denote typical elements of a set $\rho(\sigma)$ by ξ.

Semantically, we will model dynamic object creation and communication by means of a *parallel merge* operator and a communication operator.
The merge operator $\|$: $\mathbb{P} \times \mathbb{P} \to \mathbb{P}$ must satisfy $\delta \,\|\, \rho = \rho = \rho \,\|\, \delta$ and

$$\begin{aligned} \rho_1 \,\|\, \rho_2 = \lambda\sigma.(\ & \{\sigma' \cdot (\rho_i' \,\|\, \rho_j) : \sigma' \cdot \rho_i' \in \rho_i(\sigma) \quad \textit{for } i,j \in \{1,2\} \ \textit{and } i \neq j\} \ \cup \\ & \{c \cdot \lambda\tau.(f(\tau) \,\|\, \rho_j) : c \cdot f \in \rho_i(\sigma) \quad \textit{for } i,j \in \{1,2\} \ \textit{and } i \neq j\} \cup \\ & (\rho_1 \mid_\sigma \rho_2) \end{aligned}$$

with the communication operator $\mid_\sigma$ defined by

$$\rho_1 \mid_\sigma \rho_2 = \bigcup\{\xi_i \mid_\sigma \xi_j : \xi_i \in \rho_i(\sigma), \xi_j \in \rho_j(\sigma) \ \textit{ for } i,j \in \{1,2\} \ \textit{and } i \neq j\}$$

and

$$\xi_1 \mid_\sigma \xi_2 = \begin{cases} \{\sigma^\star \cdot (f_1(true) \,\|\, f_2(true))\} & \text{if } \xi_1 = \gamma!t_1 \cdot f_1 \text{ and } \xi_2 = \gamma?t_2 \cdot f_2 \\ & \text{and } \sigma^\star = \sigma(\gamma!t_1, \gamma?t_2)(\alpha, \beta) \text{ is defined} \\ \{\sigma \cdot (f_1(false) \,\|\, f_2(false))\} & \text{if } \xi_1 = \gamma!t_1 \cdot f_1 \text{ and } \xi_2 = \gamma?t_2 \cdot f_2 \\ & \text{and } \sigma(\gamma!t_1, \gamma?t_2)(\alpha, \beta) \text{ is not defined} \\ \emptyset & \text{otherwise} \end{cases}$$

Intuitively, by merging two processes all possible computation sequences are obtained, by interleaving. Moreover, an attempt is made to communicate when possible.

The function $\mathcal{D}[\![\cdot]\!] : \mathsf{Goal} \to \mathsf{Obj} \to \mathsf{Succ} \to \mathsf{Fail} \to \mathbb{IP}$ that defines the meaning of constructs from $\mathcal{L}_1$ extends the function $\mathcal{D}[\![\cdot]\!]$ given for $\mathcal{L}_0$ by taking an extra parameter from Obj, and also by having the more complex result domain $\mathbb{IP}$. We take $\mathsf{Succ} = \mathsf{Fail} \to \mathbb{IP}$ and $\mathsf{Fail} = \mathbb{IP}$. Again we use the effect function as defined in the transition system for $\mathcal{L}_1$. Notice that in equation (vi), when creating a new object, applying the effect function results in a new process identifier β that is given as an argument to the meaning representing the evaluation of the constructor.

Let $\sigma = (\theta, s, i)$, $R_0 = \lambda\rho.\rho$ and $F_0 = \delta$.

$\mathcal{L}_1$

(i) $\mathcal{D}[\![a]\!]\alpha RF = \lambda\sigma.\sigma(a)(\alpha)\mathit{defined} \to \{\sigma(a)(\alpha) \cdot RF\}, F\sigma$

(ii) $\mathcal{D}[\![p(t)]\!]\alpha RF = \lambda\sigma.\{\sigma(p(t))(\alpha) \cdot \mathcal{D}[\![b\nu_{\alpha pi(\alpha,p)}(t\theta_\alpha)]\!]\alpha RF\}$
for $p \leftarrow b$ in D_α

(iii) $\mathcal{D}[\![g_1; g_2]\!]\alpha RF = \mathcal{D}[\![g_1]\!]\alpha(\mathcal{D}[\![g_2]\!]\alpha R)F$

(iv) $\mathcal{D}[\![g_1 \,\square\, g_2]\!]\alpha RF = \mathcal{D}[\![g_1]\!]\alpha R(\mathcal{D}[\![g_2]\!]\alpha RF)$

(v) $\mathcal{D}[\![C :: \mathit{new}(\mathit{channel})]\!]\alpha RF = \lambda\sigma.\{\sigma^\star \cdot RF\}$
with $\sigma^\star = \sigma(C :: \mathit{new}(\mathit{channel}))(\alpha)$

(vi) $\mathcal{D}[\![\mathit{new}(c(t))]\!]\alpha RF = \lambda\sigma.\{\sigma^\star \cdot (RF \parallel \mathcal{D}[\![c(s_\alpha(t)\theta_\alpha)]\!]\beta R_0 F_0)\}$
with $\sigma^\star = \sigma(\mathit{new}(c(t)))(\alpha)$

(vii) $\mathcal{D}[\![C!t]\!]\alpha RF = \lambda\sigma.\{C\theta_\alpha!t \cdot f\}$ with $f = \lambda\tau \in T.\tau \to RF, \mathcal{D}[\![C!t]\!]\alpha RF$

(viii) $\mathcal{D}[\![C?t]\!]\alpha RF = \lambda\sigma.\{C\theta_\alpha?t \cdot f\}$ with $f = \lambda\tau \in T.\tau \to RF, F$

The first four equations are straightforward extensions of the equations given for $\mathcal{L}_0$. In the equation for $\mathit{new}(c(t))$ the meaning of the newly created object is merged with the meaning stored in the continuations. Communication statements result in a communication intention containing a function that decides what must be done when the communication succeeds and what must be done on failure. For the output intention the resumption that must be taken when communication fails, simply repeats the meaning of the output statement.

We remark that the technique used in section 6.2.2 also applies to the equations presented above.

Definition 8.2.2 $\mathcal{M}[\![< D \mid g >]\!] = \mathcal{D}[\![g]\!]\alpha_0 R_0 F_0$ *for some initial object* α_0.

8.2.3 Equivalence between operational and denotational semantics

Since, operationally, unresolved communication intentions result in blocking the computation, we need a projection π, mapping elements from $\mathbb{P}$ to elements of $\mathbb{R}$, that removes communication intentions for which an alternative is available; and moreover yields all computation sequences that may arise from a certain state. In other words, the projection operator eliminates the branching inherent in the process structure by computing all possible sequences.

Our equivalence result for $\mathcal{L}_1$ may be stated as

Theorem 8.2.3 $\mathcal{O}[\![< D \mid g >]\!] = \pi \circ \mathcal{M}[\![< D \mid g >]\!]$ *for a suitable projection* π.

The proof looks as follows.

$$\mathcal{O}[\![< D \mid g >]\!] = \mathcal{T}[\![\{< \alpha_0, g; \surd : \Delta >\}]\!] \text{ by definition 8.2.1}$$

$$= \pi \circ \mathcal{G}[\![\{< \alpha_0, g; \surd; \Delta >\}]\!] \text{ by corollary 8.2.7}$$

$$= \pi \circ \mathcal{F}[\![g; \surd : \Delta]\!]\alpha_0 \text{ by definition 8.2.5}$$

$$= \pi \circ \mathcal{D}[\![g]\!]\alpha_0 R_0 F_0 \text{ by definition 8.2.5}$$

$$= \pi \circ \mathcal{M}[\![< D \mid g >]\!] \text{ by definition 8.2.2}$$

As before we need an intermediary function $\mathcal{F} : \mathsf{Obj} \to \mathsf{FailCo} \to \mathsf{Fail}$ to take syntactic failure continuations to semantic failure continuations, and a function $\mathcal{R} : \mathsf{Obj} \to \mathsf{SuccCo} \to \mathsf{Succ}$, to take syntactic success continuations to semantic success continuations. Moreover, we need a function $\mathcal{G} : \mathcal{P}(\mathsf{Obj} \times \mathsf{FailCo}) \to \mathbb{P}$ that maps collections of objects to their semantic counterparts in $\mathbb{P}$.

Definition 8.2.4

a. $\mathcal{R}[\![\surd]\!]\alpha = \lambda\rho.\rho$

b. $\mathcal{R}[\![g; R]\!]\alpha = \mathcal{D}[\![g]\!]\alpha\mathcal{R}[\![R]\!]\alpha$

c. $\mathcal{F}[\![\Delta]\!]\alpha = \delta$

d. $\mathcal{F}[\![R : F]\!]\alpha = \mathcal{R}[\![R]\!]\alpha\mathcal{F}[\![F]\!]\alpha$

e. $\mathcal{G}[\![\{< \alpha_1, F_1 >, ..., < \alpha_n, F_n >\}]\!] = \mathcal{F}[\![F_1]\!]\alpha_1 \parallel ... \parallel \mathcal{F}[\![F_n]\!]\alpha_n$

Again we have the property that unlabeled transitions do not affect the meaning.

Lemma 8.2.5 *if* $(\Gamma, \sigma) \longrightarrow (\Gamma', \sigma)$ *then* $\mathcal{G}[\![\Gamma]\!]\sigma = \mathcal{G}[\![\Gamma']\!]\sigma$

The key step in the proof that $\mathcal{O} = \pi \circ \mathcal{M}$ consists of showing that $\pi \circ \mathcal{G}$ is a fixed point of the operator that characterizes the operational meaning of a program. (See definition 8.2.1)

We define a projection operator that reduces the unresolved communication intentions to $\lambda\sigma.\{\varepsilon\}$, the equivalent of the empty process δ.

We first define a function $rem : \mathbb{P} \to \mathbb{P}$ by

$$rem(\rho)(\sigma) = \rho(\sigma) \backslash \{c \cdot f : c \cdot f \in \rho(\sigma)\}$$

In other words $rem(\rho)(\sigma)$ is what remains after removing all unresolved communication intentions.

In order to lay hands on the behavior that gives rise to a state we define a labeling function $\dot{} : \Sigma \to \Lambda$ by $\dot{\sigma}(a)(\alpha) = a$ and $\dot{\sigma} = \star$ otherwise, as an inverse to the effect function.

We then define $\pi : \mathbb{IP} \to \mathbb{IR}$ by

$$\pi\ \rho\ = \begin{cases} \lambda\sigma.\{\varepsilon\} & \text{if } \rho = \delta \text{ or } rem(\rho)(\sigma) = \emptyset \\ \lambda\sigma.\bigcup\{\dot{\sigma}' \circ (\pi\ \rho')(\sigma') : \sigma' \cdot \rho' \in rem(\rho)(\sigma)\} & \text{otherwise} \end{cases}$$

Below we will use the property $\lambda\sigma.\bigcup\{\dot{\sigma}' \circ (\pi\ \rho')(\sigma')\} = \pi\ \lambda\sigma.\{\sigma' \cdot \rho'\}$, which can easily be verified.

Lemma 8.2.6 $\Psi(\pi \circ \mathcal{G}) = \pi \circ \mathcal{G}$

Proof: We use a complexity measure similar as the one introduced for $\mathcal{L}_0$, but extended in order to cope with configurations containing a collection of processes. We define $c(\Gamma) = \Sigma_{x \in \Gamma} c(x)$ and let $c(< \alpha, F >) = c(F)$. We also define

$$c(C :: new(channel)) = c(new(c(t))) = c(C!t) = c(C?t) = 1.$$

We must now prove that for all Γ and all σ we have $\Psi(\pi \circ \mathcal{G})[\![\Gamma]\!]\sigma = \pi \circ \mathcal{G}[\![\Gamma]\!]\sigma$. We will treat some selected cases. Let $\overline{R}_i = \mathcal{R}[\![R_i]\!]\alpha_i$ and $\overline{F}_i = \mathcal{F}[\![F_i]\!]\alpha_i$, with subscript i possibly empty.

- If $\Gamma = \{< \alpha, new(c(t)); R : F >\}$ then for $\sigma^\star = \sigma(new(c(t)))(\alpha)$
  with $\sigma = (\theta, s, i)$ $\Psi(\pi \circ \mathcal{G})[\![\Gamma]\!] =$
  $\lambda\sigma.\bigcup\{\dot{\sigma}^\star \circ (\pi \circ \mathcal{G})[\![\{< \alpha, R : F >, < \beta, c(s_\alpha(t)\theta_\alpha); \surd : \Delta >\}]\!](\sigma^\star)\} =$
  (by the property of π)
  $\pi\ \lambda\sigma.\{\sigma^\star \cdot \mathcal{G}[\![\{< \alpha, R : F >, < \beta, c(s_\alpha(t)\theta_\alpha); \surd : \Delta >\}]\!]\} =$
  (by the definition of $\mathcal{G}$)
  $\pi\ \lambda\sigma.\{\sigma^\star \cdot (\overline{R}\overline{F} \parallel \mathcal{D}[\![c(s_\alpha(t)\theta_\alpha)]\!]\beta\mathcal{R}[\![\surd]\!]\alpha\mathcal{F}[\![\Delta]\!]\alpha)\} =$
  (by the definition of $\mathcal{D}$)
  $\pi\ \mathcal{D}[\![new(c(t))]\!]\alpha\overline{R}\overline{F} =$ (by applying definition 8.2.4)
  $\pi \circ \mathcal{G}[\![\{< \alpha, new(c(t)); R : F >\}]\!]$ which proves the result.

- Let $\Gamma = \{< \alpha_1, C_1!t_1; R_1 : F_1 >, < \alpha_2, C_2?t_2; R_2 : F_2 >\}$
  and $\sigma^\star = \sigma(\gamma!t_1, \gamma?t_2)(\alpha_1, \alpha_2)$ for $\sigma = (\theta, s, i)$ and $C_1\theta_{\alpha_1} = \gamma = C_2\theta_{\alpha_2}$.
  If $\sigma^\star$ is defined then $\Psi(\pi \circ \mathcal{G})[\![\Gamma]\!]\sigma =$ (by applying *Comm*)
  $(\pi\ \lambda\sigma.\{\sigma^\star \cdot \mathcal{G}[\![\{< \alpha_1, R_1 : F_1 >, < \alpha_2, R_2 : F_2 >\}]\!]\})(\sigma) =$
  (by definition 8.2.4)
  $(\pi\ \lambda\sigma.\{\sigma^\star \cdot (\overline{R}_1\overline{F}_1 \parallel \overline{R}_2\overline{F}_2)\})(\sigma) =$ (by the definition of merging)
  $(\pi\ (\mathcal{D}[\![C_1!t_1]\!]\alpha_1\overline{R}_1\overline{F}_1 \parallel \mathcal{D}[\![C_2?t_2]\!]\alpha_2\overline{R}_2\overline{F}_2)\ )(\sigma) =$
  (by applying definition 8.2.4 again)
  $\pi \circ \mathcal{G}[\![\{< \alpha_1, C_1!t_1; R_1 : F_1 >, < \alpha_2, C_2?t_2; R_2 : F_2 >\}]\!]\sigma.$
  When $\sigma^\star$ is not defined then $\Psi(\pi \circ \mathcal{G})[\![\Gamma]\!]\sigma =$ (by applying *Comm*)
  $(\pi\ \lambda\sigma.\{\sigma \cdot \mathcal{G}[\![\{< \alpha_1, C_1!t_1; R_1 : F_1 >, < \alpha_2, F_2 >\}]\!]\ \})(\sigma) =$

(by definition 8.2.4)
$(\pi\ \lambda\sigma.\{\sigma \cdot (\mathcal{D}[\![C_1!t_1]\!]\alpha_1\overline{R}_1\overline{F}_1 \parallel \overline{F}_2)\})(\sigma) =$ (by the definition of merging)
$(\pi\ (\mathcal{D}[\![C_1!t_1]\!]\alpha_1\overline{R}_1\overline{F}_1 \parallel \mathcal{D}[\![C_2?t_2]\!]\alpha_2\overline{R}_2\overline{F}_2))(\sigma) =$
$\pi \circ \mathcal{G}[\![\Gamma]\!]\sigma$ as desired. □

As a consequence of the previous lemma we may state

Corollary 8.2.7 $\mathcal{T} = \pi \circ \mathcal{G}$

8.3 Dynamic object creation and method calls by rendez-vous

The language $\mathcal{L}_2$ extends $\mathcal{B}_2$ by providing the primitives for communication as actually used in DLP. $\mathcal{L}_2$ is to a certain extent similar to $\mathcal{L}_1$. Instead of communication over channels however, it provides the possibility for processes to communicate in a kind of rendez-vous. An object, or rather a process associated to an object, may call for a method to be evaluated by another object. Since backtracking may occur over the answers produced in evaluating the method, for each such rendez-vous a process is created. Instead of a single process, as for $\mathcal{L}_1$, now multiple processes may refer to a single object. We distinguish between the constructor process, executing the own activity of an object, and the processes created for evaluating a method. The process that called for the method is given a pointer to the process evaluating the call, and may through this pointer ask for all the answers that result from the evaluation. Backtracking over these answers is initiated by the invoking process. Since multiple processes are involved, we speak of distributed (or global) backtracking, as opposed to the backtracking occurring locally in a process.

Syntax Terms and actions are as for $\mathcal{L}_0$, however as additional *constants* we need object names $c \in$ Objname, and method names $m \in$ Method. We assume that Objname $\subset$ Method. We adopt the convention to use capital O for logical variables that may refer to an object and Q for variables that may become bound to pointers to evaluation processes.

As *primitives* $e \in$ Prim, by which $\mathcal{L}_2$ extends $\mathcal{L}_0$, we now have

$$e ::= O :: new(c(t)) \mid O!m(t) \mid accept(m_1, ..., m_n) \mid Q? \mid !t$$

And, we define the set goals $g \in$ Goal by

$$g ::= a \mid m(t) \mid g_1; g_2 \mid g_1 \,\square\, g_2 \mid e$$

The goal $O{::}new(c(t))$ can be used to create a new instance of the object c, for which the constructor $c(t)$ is evaluated. The variable O will become bound to this newly created object. The goal $O!m(t)$ initiates a rendez-vous, in that the object denoted by O is asked to evaluate the goal $m(t)$. The goal $accept(m_1, ..., m_n)$ may be used to state the willingness to wait for and accept any of the methods $m_1, ..., m_n$. As a restriction we require that accept goals may occur only in constructor processes.

The auxiliary request $Q?$ is used to ask for the answer substitutions resulting from the evaluation performed by the process that will become bound to Q. The goal $!t$ is an auxiliary goal used for sending back the answer substitutions. These auxiliary goals do not correspond to any goal in a method declaration resulting from the translation of clauses.

The clauses for a method m, are collected in a *method declaration* of the form $m \leftarrow b$ with $b \equiv \lambda\tau.g$, which corresponds to a procedure declaration in $\mathcal{L}_0$ and $\mathcal{L}_1$.

As for $\mathcal{L}_1$ we assume to have *object declarations* of the form

$$(c \leftarrow b_c, < v_i >_{i=1,...,r}, < m_i \leftarrow b_i >_{i=1,...,k>})$$

A *program* $< D \mid g >$ consists of a collection of object declarations D and a goal g.

8.3.1 Operational semantics

Our solution to the distributed backtracking that may arise in answering a method call forces us to distinguish between objects and processes.

The set of *objects*

$$\mathsf{Obj} = \{(c, n) : c \text{ an object name}, n \in \mathbb{N}\}$$

consists of all pairs containing an object name and an instance number. We need process identifiers $\alpha, \beta \in \mathsf{Proc}$, with Proc defined by

$$\mathsf{Proc} = \{(c, n, k) : (c, n) \in \mathsf{Obj}, k \in \mathbb{N}\}$$

to be able to refer to particular processes. The elements in the set Proc differ from those in Obj only in having a process number, indicating how many processes referring to the object (c, n) have been created. We have adopted the convention to identify the process $(c, n, 0)$, evaluating the constructor of c, with the object (c, n). We use a function *obj*, defined by $obj(c, n, k) = (c, n)$ to get hold of the object to which a process refers. For $\alpha = (c, n, k)$ we write D_α for the object declaration for c.

We extend the set of terms by redefining

$$t ::= \varepsilon \mid \delta \mid X \mid v \mid \alpha \mid f(t_1, ..., t_n) : f \text{ a function symbol}, n \geq 0$$

We have introduced the term δ to report the absence of any remaining answers for a method call.

We now use generalized substitutions *substitutions* $\theta \in \mathsf{Subst}^\star = \mathsf{Proc} \rightarrow \mathsf{Subst}$ with $\theta_\alpha \in \mathsf{Subst}$ the substitution belonging to some process α.

We also need a *store* $s \in \mathsf{Store}^+ = \mathsf{Obj} \rightarrow \mathsf{Store}$, and write $s_{obj(\alpha)} \in \mathsf{Store}$ for the store for object $obj(\alpha)$.

A *count* $i \in \mathsf{Count}^\star = (\mathsf{Objname} \cup \mathsf{Obj} \cup \mathsf{Proc} \times \mathsf{Method}) \rightarrow \mathbb{N}$ is used to administrate, respectively, the current instance number of a named object, the number of processes active for an object and the number of times a method has been called (locally) from within a process.

Our *states* σ come from the set

$$\Sigma = \{(\theta, s, i) : \theta \in \mathsf{Subst}^\star, s \in \mathsf{Store}^+, i \in \mathsf{Count}^\star\}$$

and we will use the abbreviations, as introduced for $\mathcal{L}_1$,

$$\theta\{\alpha/X \leftarrow t\} = \theta\{\alpha/\theta_\alpha\{X/t\}\}$$
$$s\{obj(\alpha)/v \leftarrow t\} = s\{obj(\alpha)/s_{obj(\alpha)}\{v/t\}\}.$$

For *renaming* we use a function $\nu \in \mathsf{Proc} \times \mathsf{Method} \times \mathbb{N} \rightarrow \mathsf{Var} \rightarrow \mathsf{Var}$.

For dealing with *communication* we need a set Comm of communication intentions, having elements of the form

$$Q :: \alpha!m(t),\ \overline{m},\ \alpha? \text{ and } !t.$$

We speak of (syntactically) matching communication intentions whenever we encounter one of the pairs $(Q :: \alpha!m(t), \overline{m})$ or $(\alpha?, !t)$.

We write $\sigma(c, \bar{c})(\alpha, \beta)$ to denote the application of the effect function to a pair of matching communication intentions $c, \bar{c} \in \mathsf{Comm}$.

As for $\mathcal{L}_0$ and $\mathcal{L}_1$ we use syntactic success continuations from SuccCo defined by $R ::= \surd \mid g; R$, and syntactic failure continuations from FailCo, defined by $F ::= \Delta \mid R : F$, to model the (local) backtracking behavior. In order to model the distributed backtracking (possibly) occurring in a rendez-vous we need syntactic process continuations from ProcCo defined by

$$C ::= \Xi \mid < \alpha, F > : C$$

The empty process continuation Ξ is the process that has nothing left to do. Intuitively, a non-empty process continuation of the form $< \alpha, F > : C$ represents a stack of processes from which the top must be popped if it communicates successfully with another process. When this happens the process continuation C starts to be executed in parallel with the left-overs from $< \alpha, F >$. We use configurations (Γ, σ) with $\Gamma \in \mathcal{P}(\mathsf{ProcCo})$.

Again, we assume to have a general rule of the form

$$(\Gamma, \sigma) \xrightarrow{\eta} (\Gamma', \sigma') \Longrightarrow (X \cup \Gamma, \sigma) \xrightarrow{\eta} (X \cup \Gamma', \sigma')$$

for $X \in \mathcal{P}(\mathsf{ProcCo})$ disjoint from Γ and Γ'. Let $\sigma = (\theta, s, i)$.[1]

[1] The axioms given here are identical to the axioms given for $\mathcal{L}$ in the previous chapter (section 7.4), except for the application of the effect function in *Method* that is slightly more complicated since it must be used in the denotational semantics as well.

$\mathcal{L}_2$

$$(\{< \alpha, a; R : F >: C\}, \sigma) \xrightarrow{a} (\{< \alpha, R : F >: C\}, \sigma^\star) \longrightarrow (\{< \alpha, F >: C\}, \sigma) \qquad \textit{Action}$$

with for $a \in \{u(t), v := t, t_1 = t_2\}$ the interpretation

- $\sigma(u(t))(\alpha) = (\theta', s, i)$
  where $\theta' = \theta\{\alpha/X \leftarrow X\}_{X \in varsof(t)}$
- $\sigma(v := t)(\alpha) = (\theta, s', i)$
  where $s' = s\{\alpha/v \leftarrow s_{obj(\alpha)}(t)\theta_\alpha\}$
- $\sigma(t_1 = t_2)(\alpha) = (\theta', s, i)$
  where $\theta' = \theta\{\alpha/mgu(t_1', t_2') \circ \theta_\alpha\}$
  for $t_i' = s_{obj(\alpha)}(t_i)\theta_\alpha$

$$(\{< \alpha, m(t); R : F >: C\}, \sigma) \xrightarrow{\star} (\{< \alpha, b\nu_{\alpha m i(\alpha,m)}(t\theta_\alpha); R : F >: C\}, \sigma^\star) \qquad \textit{Rec}$$

for $m \leftarrow b$ in D_α, with the interpretation

- $\sigma(m(t))(\alpha) = (\theta, s, i\{(\alpha, m)/i(\alpha, m) + 1\})$

$$(\{< \alpha, (g_1; g_2); R : F >: C\}, \sigma) \longrightarrow (\{< \alpha, g_1; (g_2; R) : F >: C\}, \sigma) \qquad \textit{Seq}$$

$$(\{< \alpha, (g_1 \,\square\, g_2); R : F >: C\}, \sigma) \longrightarrow (\{< \alpha, (g_1; R) : (g_2; R) : F >: C\}, \sigma) \qquad \textit{Alt}$$

$$(\{< \alpha, O :: new(c(t)); R : F >: C\}, \sigma) \xrightarrow{\star} (\{< \alpha, R : F >: C, < \beta, F_\beta >: \Xi\}, \sigma^\star) \qquad \textit{New}$$

where $F_\beta \equiv c(s_{obj(\alpha)}(t)\theta_\alpha; \surd : \Delta$, with the interpretation

- $\sigma(O :: new(c(t)))(\alpha) = (\theta', s, i')$
  where $\theta' = \theta\{\alpha/O \leftarrow \beta\}$ with $\beta = (c, i'(c), 0)$
  and $i' = i\{c/i(c) + 1\}$

$$(\{< \alpha\ , O!m(t); R_1 : F_1 >: C_1, \qquad \textit{Method}$$

$$< \beta, accept(..., m, ...); R_2 : F_2 >: C_2\}, \sigma)$$

$$\xrightarrow{\star} (\{< \alpha, Q?; R_1 : F_1 >: C_1,$$

$$< \beta', F_{\beta'} >:< \beta, R_2 : F_2 >: C_2\}, \sigma^\star)$$

if $O\theta_\alpha = \beta$,

where $F_{\beta'} \equiv (m(t'); !t' \,\square\, !\delta); \surd : \Delta$, for $t' = s_{obj(\alpha)}(t)\theta_\alpha$

with the interpretation

- $\sigma(Q :: \beta!m(t), \overline{m})(\alpha, \beta) = (\theta', s, i')$
  where for $\beta = (c, n, 0)$ and a fresh variable Q it holds that $\beta' = (c, n, i'(c, n))$, $i' = i\{(c, n)/i(c, n) + 1\}$ and $\theta' = \theta\{\alpha/Q \leftarrow \beta'\}$

$$(\{< \alpha, Q?; R_1 : F_1 >: C_1, < \beta, !t; R_2 : F_2 >: C_2\}, \sigma) \qquad \textit{Result}$$

$$\xrightarrow{\star} (\{< \alpha, F_1 >: C_1, < \beta, R_2 : F_2 >: \Xi, C_2\}, \sigma) \text{ if } t = \delta$$

$$\xrightarrow{\star} (\{< \alpha, (t\theta_\alpha = t\theta_\beta \,\square\, u(t\theta_\alpha); Q?); R_1 : F_1 >: C_1,$$

$$< \beta, R_2 : F_2 >: \Xi, C_2\}, \sigma)$$

provided that $Q\theta_\alpha = \beta$

$$(\{< \alpha, \surd : F >: C\}, \sigma) \longrightarrow (\{< \alpha, F >: C\}, \sigma) \qquad \textit{Tick}$$

The axioms given above are, apart from the axioms *Method* and *Result*, straightforwardly adapted from the ones given for $\mathcal{L}_1$.

To be able to apply *Method*, the condition $O\theta_\alpha = \beta$ must be fulfilled, otherwise no transition is possible. The rule operates on a pair $O!m(t)$ and $accept(..., m, ...)$ indicating both the call for a method m and the willingness to accept the call. As a result, a process β' is created and put on top of the process stack for β. The variable Q is introduced to be able to refer to β'. The process β' executes the goal $m(t'); !t' \,\square\, !\delta$, of which the first part represents the sequential composition of evaluating the call and sending the results. The second alternative states that no more answers are available.

These two parts correspond to the two possible ways in which the axiom *Result* may be applied. If the answered term is δ then the requesting process pops its failure stack, otherwise the answer substitution will be computed by unifying $t\theta_\alpha$ with $t\theta_\beta$. Since other answers may be produced an alternative request is made, preceded by the action $u(t\theta_\alpha)$ in order to undo the previous bindings of the argument t.

We take $\mathbb{R} = \Sigma \to \mathcal{P}_{nc}(\Lambda^\infty)$.

Definition 8.3.1 $\mathcal{O}[\![< D \mid g >]\!] = \mathcal{T}[\![\{< \alpha_0, g; \surd : \Delta >: \Xi\}]\!]$ *for some initial process* α_0, *where for* $\Psi : (\mathcal{P}(\mathsf{ProcCo}) \to \mathbb{R}) \to (\mathcal{P}(\mathsf{ProcCo}) \to \mathbb{R})$ *and*

$$\Psi(\phi)[\![\Gamma]\!](\sigma) = \begin{cases} \{\varepsilon\} & \text{if } (\Gamma, \sigma) \text{ blocks} \\ \bigcup\{\eta \circ \phi[\![\Gamma']\!](\sigma') \mid (\Gamma, \sigma) \xrightarrow{\eta} (\Gamma', \sigma')\} & \text{otherwise} \end{cases}$$

we define $\mathcal{T} = \mathsf{fix}\Psi$

8.3.2 Denotational semantics

As a domain we take $\mathbb{P}$, with typical elements ρ and q, given by

$$\mathbb{P} \cong \{\delta\} \cup \Sigma \to \mathcal{P}_{co}(\Sigma \times \mathbb{P} \cup \mathsf{C} \times \mathbb{P} \cup \mathsf{C} \times (\mathsf{Proc} \to \mathsf{Term} \to \mathbb{P}))$$

where $\mathsf{C} = \mathsf{Proc} \times \mathsf{Comm}$, with Comm the set of communication intentions as defined in section 8.3.1. We will denote elements of C by c and elements of $\mathsf{Proc} \to \mathsf{Term} \to \mathbb{P}$ by f. Take note that δ here denotes the empty process and not the special term. For non-empty $\rho \in \mathbb{P}$, the set $\rho(\sigma)$, for some σ, may contain ordinary computation steps of the form $\sigma' \cdot \rho'$, or communication intentions of the form $[\alpha, Q :: \beta!m(t)] \cdot \rho'$, $[\alpha, \overline{m}] \cdot f$, $[\alpha, \beta?] \cdot f$ or $[\alpha, !t] \cdot \rho'$. Other communication intentions will simply not arise. We use ξ to denote the elements of such a set $\rho(\sigma)$.

We have to adapt the *parallel merge* operator given for $\mathcal{L}_1$ to the present situation.

The operator $\|\colon \mathbb{P} \times \mathbb{P} \to \mathbb{P}$ must satisfy $\delta \parallel \rho = \rho = \rho \parallel \delta$ and

$$\begin{aligned} \rho_1 \parallel \rho_2 = \lambda\sigma.(& \{\sigma' \cdot (\rho_i' \parallel \rho_j) : \sigma' \cdot \rho_i' \in \rho_i(\sigma) \text{ for } i,j \in \{1,2\} \text{ and } i \neq j\} \cup \\ & \{c \cdot (\rho_i' \parallel \rho_j) : c \cdot \rho_i' \in \rho_i(\sigma) \text{ for } i,j \in \{1,2\} \text{ and } i \neq j\} \cup \\ & \{c \cdot \lambda\beta t.(f(\beta)(t) \parallel \rho_j) : c \cdot f \in \rho_i(\sigma) \text{ for } i,j \in \{1,2\}, i \neq j\} \cup \\ & (\rho_1 \mid_\sigma \rho_2) \end{aligned}$$

with $\rho_1 \mid_\sigma \rho_2 = \bigcup\{\xi_i \mid_\sigma \xi_j : \xi_i \in \rho_i(\sigma), \xi_j \in \rho_j(\sigma) \text{ for } i,j \in \{1,2\} \text{ and } i \neq j\}$

and

$$\xi_1 \mid_\sigma \xi_2 = \begin{cases} \{\sigma^\star \cdot (\rho \parallel f(\beta')(t))\} & \text{if } \xi_1 = [\alpha, Q :: \beta!m(t)] \cdot \rho \text{ and } \xi_2 = [\beta, \overline{m}] \cdot f \\ & \text{where } \sigma^\star = \sigma(Q :: \beta!m(t), \overline{m})(\alpha, \beta) \\ & \text{and } \beta' = (c, n, i(c,n)+1) \text{ for } \beta = (c, n, 0) \\ \{\sigma \cdot (\rho \parallel f(\beta)(t))\} & \text{if } \xi_1 = [\alpha, \beta?] \cdot f \text{ and } \xi_2 = [\beta, !t] \cdot \rho \\ \emptyset & \text{otherwise} \end{cases}$$

Notice that, when communicating an answer substitution the state σ is not changed, since the actual state change will be effected afterwards by the process asking for the solution.

In order to model our protocol of mutual exclusion (c.f. section 6.5.2) we define the *insertion* operator $\rhd_\alpha : \mathbb{IP} \times \mathbb{IP} \to \mathbb{IP}$ by $\delta \rhd_\alpha \delta = \delta, \delta \rhd_\alpha \rho = \rho$, $\rho \rhd_\alpha \delta = \delta$ and

$$\begin{aligned}\rho \rhd_\alpha q = \lambda\sigma.(\ &\{\sigma' \cdot \rho \rhd_\alpha q' : \sigma' \cdot q' \in q(\sigma)\} \cup \\ &\{[\alpha, !t] \cdot (\rho \parallel q') : [\alpha, !t] \cdot q' \in q(\sigma)\} \cup \\ &\{c \cdot \rho \rhd_\alpha q' : c \cdot q' \in q(\sigma) \ \ for \ \ c \neq [\alpha, !t]\} \cup \\ &\{c \cdot \lambda\beta\tau.\rho \rhd_\alpha f(\beta)(\tau) : c \cdot f \in q(\sigma)\}\ )\end{aligned}$$

The function $\mathcal{D}[\![\cdot]\!] : \mathsf{Goal} \to \mathsf{Proc} \to \mathsf{Succ} \to \mathsf{Fail} \to \mathbb{IP}$, giving a denotational meaning to the language $\mathcal{L}_2$, resembles the corresponding function for $\mathcal{L}_1$ to a large extent. We again take $\mathsf{Succ} = \mathsf{Fail} \to \mathbb{IP}$ and $\mathsf{Fail} = \mathbb{IP}$. Notice that we use the effect function as defined in the transition system for $\mathcal{L}_2$, and remember that in equation (v) the process identifier β results from applying the effect function.

Let $\sigma = (\theta, s, i)$, $R_0 = \lambda\rho.\rho$ and $F_0 = \delta$.

$\mathcal{L}_2$

(i) $\mathcal{D}[\![a]\!]\alpha RF = \lambda\sigma.\sigma(a)(\alpha) defined \to \{\sigma(a)(\alpha) \cdot RF\}, F\sigma$

(ii) $\mathcal{D}[\![m(t)]\!]\alpha RF = \lambda\sigma.\{\sigma(m(t))(\alpha) \cdot \mathcal{D}[\![b\nu_{\alpha m i(\alpha,m)}(t\theta_\alpha)]\!]\alpha RF\}$
for $m \leftarrow b$ in D_α

(iii) $\mathcal{D}[\![g_1; g_2]\!]\alpha RF = \mathcal{D}[\![g_1]\!]\alpha(\mathcal{D}[\![g_2]\!]\alpha R)F$

(iv) $\mathcal{D}[\![g_1 \,\square\, g_2]\!]\alpha RF = \mathcal{D}[\![g_1]\!]\alpha R(\mathcal{D}[\![g_2]\!]\alpha RF)$

(v) $\mathcal{D}[\![O :: new(c(t))]\!]\alpha RF = \lambda\sigma.\{\sigma^\star \cdot (RF \parallel \mathcal{D}[\![c(s_{obj(\alpha)}(t)\theta_\alpha)]\!]\beta R_0 F_0)\}$
with $\sigma^\star = \sigma(O :: new(c(t)))(\alpha)$

(vi) $\mathcal{D}[\![O!m(t)]\!]\alpha RF = \lambda\sigma.\{[\alpha, Q :: (O!m(s_{obj(\alpha)}(t)))\theta_\alpha] \cdot \mathcal{D}[\![Q?]\!]\alpha RF\}$
for a fresh variable Q

(vii) $\mathcal{D}[\![accept(m_1, ..., m_n)]\!]\alpha RF = \lambda\sigma.\{[\alpha, \overline{m_i}] \cdot f_i : 1 \leq i \leq n\}$
with $f_i = \lambda\beta t.RF \rhd_\beta \mathcal{D}[\![m_i(t); !t \,\square\, !\delta]\!]\beta R_0 F_0$

(viii) $\mathcal{D}[\![Q?]\!]\alpha RF = \lambda\sigma.\{[\alpha, Q\theta_\alpha?] \cdot f\}$
with $f = \lambda\beta t.t \neq \delta \to \mathcal{D}[\![(t\theta_\alpha = t\theta_\beta) \,\square\, (u(t\theta_\alpha); Q?)]\!]\alpha RF, F$

(ix) $\mathcal{D}[\![!t]\!]\alpha RF = \lambda\sigma.\{[\alpha, !t] \cdot RF\}$

The equation for the *accept* statement shows that a process for evaluating the method call and for returning the result is prepared, to become an actual process

when a process identifier and the actual parameters of the method call are provided. The meaning of the resumption statement $Q?$ is a process that effects the binding of the communicated solution while undoing it when it is forced into its failure continuation. When no more answers are available the failure continuation is taken directly.

Definition 8.3.2 $\mathcal{M}[\![< D \mid g >]\!] = \mathcal{D}[\![g]\!]\alpha_0 R_0 F_0$ *for some initial process* α_0.

8.3.3 Equivalence between operational and denotational semantics

We state our equivalence result.

Theorem 8.3.3 $\mathcal{O}[\![< D \mid g >]\!] = \pi \circ \mathcal{M}[\![< D \mid g >]\!]$ *for a suitable projection* π

The proof is along the same lines as the one for $\mathcal{L}_1$, but as an extra complexity we must now deal with (syntactic) process continuations. As an outline of the proof we put

$$\begin{aligned}
\mathcal{O}[\![< D \mid g >]\!] &= \mathcal{T}[\![\{< \alpha_0, g; \surd : \Delta >: \Xi\}]\!] \text{ by definition 8.3.1} \\
&= \pi \circ \mathcal{G}[\![\{< \alpha_0, g; \surd; \Delta >: \Xi\}]\!] \text{ by corollary 8.3.7} \\
&= \pi \circ \mathcal{F}[\![g; \surd : \Delta]\!]\alpha_0 \text{ by definition 8.3.4} \\
&= \pi \circ \mathcal{D}[\![g]\!]\alpha_0 R_0 F_0 \text{ by definition 8.3.4} \\
&= \pi \circ \mathcal{M}[\![< D \mid g >]\!] \text{ by definition 8.3.2}
\end{aligned}$$

We use a function $\mathcal{G} : \mathcal{P}(\mathsf{ProcCo}) \to \mathbb{IP}$, taking sets of process continuations to (semantic) processes, a function $\mathcal{C} : \mathsf{ProcCo} \to \mathbb{IP}$ that maps a process continuation into $\mathbb{IP}$, and the functions $\mathcal{F} : \mathsf{FailCo} \to \mathsf{Fail}$ and $\mathcal{R} : \mathsf{SuccCo} \to \mathsf{Succ}$.

Definition 8.3.4

a. $\mathcal{R}[\![\surd]\!]\alpha = \lambda\rho.\rho$

b. $\mathcal{R}[\![g; R]\!]\alpha = \mathcal{D}[\![g]\!]\alpha\mathcal{R}[\![R]\!]\alpha$

c. $\mathcal{F}[\![\Delta]\!]\alpha = \delta$

d. $\mathcal{F}[\![R : F]\!]\alpha = \mathcal{R}[\![R]\!]\alpha\mathcal{F}[\![F]\!]\alpha$

e. $\mathcal{C}[\![\Xi]\!] = \delta$

f. $\mathcal{C}[\![< \alpha, F >: C]\!] = \mathcal{C}[\![C]\!] \rhd_\alpha \mathcal{F}[\![F]\!]\alpha$

g. $\mathcal{G}[\![\{C_1, ..., C_n\}]\!] = \mathcal{C}[\![C_1]\!] \parallel ... \parallel \mathcal{C}[\![C_n]\!]$

Intuitively, the equation $\mathcal{C}[\![< \alpha, F >: C]\!] = \mathcal{C}[\![C]\!] \rhd_\alpha \mathcal{F}[\![F]\!]\alpha$ states that the process resulting from evaluating C is inserted in the process arising from evaluating F for α at the point that α is willing to communicate its first result.

Lemma 8.3.5 *if* $(\Gamma, \sigma) \longrightarrow (\Gamma', \sigma)$ *then* $\mathcal{G}[\![\Gamma]\!]\sigma = \mathcal{G}[\![\Gamma']\!]\sigma$

Not surprisingly by now, the key step in the proof $\mathcal{O} = \pi \circ \mathcal{M}$ consists of showing that $\pi \circ \mathcal{G}$ is a fixed point of the operator that characterizes the operational meaning of a program.

We use a projection π similar to the one we used for $\mathcal{L}_1$, but redefine for ρ non-empty

$$rem(\rho)(\sigma) = \rho(\sigma) \backslash (\{c \cdot \rho' : c \cdot \rho' \in \rho(\sigma)\} \cup \{c \cdot f : c \cdot f \in \rho(\sigma)\})$$

in order to remove all unresolved communication intentions.

Lemma 8.3.6 $\Psi(\pi \circ \mathcal{G}) = \pi \circ \mathcal{G}$

Proof: We extend the complexity measure given for $\mathcal{L}_1$ to syntactic process continuations by stating

$$c(< \alpha, F >: C) = c(F) + c(C) \text{ and } c(\Xi) = 0.$$

It is easy to establish that the required property of c stating that if $(\Gamma, \sigma) \rightarrow (\Gamma', \sigma)$ then $c(\Gamma) > c(\Gamma')$ holds. We treat the case corresponding to the axiom *Result*. Let $\overline{R}_i = \mathcal{R}[\![R_i]\!]\alpha_i$, $\overline{F}_i = \mathcal{F}[\![F_i]\!]\alpha_i$ and $\overline{C}_i = \mathcal{C}[\![C_i]\!]$, with subscript i possibly empty.

- If $\Gamma = \{< \alpha_1, Q?; R_1 : F_1 >: C, < \alpha_2, !t; R_2; F_2 >: C_2\}$ and $t = \delta$ then
  $\Psi(\pi \circ \mathcal{G})[\![\Gamma]\!]\sigma = \bigcup\{\dot{\sigma} \circ (\pi \circ \mathcal{G})[\![\{< \alpha_1, F_1 >: C_1, < \alpha_2, R_2 : F_2 >: \Xi, C_2\}]\!]\sigma\} =$
  (by the property of π)
  $(\pi\ \lambda\sigma.\{\sigma \cdot \mathcal{G}[\![\{< \alpha_1, F_1 >: C_1, < \alpha_2, R_2 : F_2 >: \Xi, C_2\}]\!]\})(\sigma) =$
  (by definition 8.3.4)
  $(\pi\ \lambda\sigma.\{\sigma \cdot (\mathcal{C}[\![< \alpha_1, F_1 >: C_1]\!] \parallel \mathcal{C}[\![< \alpha_2, R_2 : F_2 >: \Xi]\!] \parallel \mathcal{C}[\![C_2]\!])\})(\sigma) =$
  (taking $f = \lambda\beta t.t \neq \delta \rightarrow \overline{C}_1 \rhd_{\alpha_1} \rho, \overline{C}_1 \rhd_{\alpha_1} \overline{F}_1$
  for $\rho = \mathcal{D}[\![t\theta_{\alpha_1} = t\theta_\beta \,\square\, u(t\theta_{\alpha_2}); Q?]\!]\alpha_1\overline{R}_1\overline{F}_1$)
  $(\pi\ \lambda\sigma.\{\sigma \cdot (f(\alpha_2)(t) \parallel \overline{R}_2\overline{F}_2 \parallel \overline{C}_2)\})(\sigma) =$ (by the definition of merging)
  $(\pi\ (\lambda\sigma.\{[\alpha_1, Q\theta_\alpha?] \cdot f\} \parallel \lambda\sigma.\{[\alpha_2, !t] \cdot (\overline{R}_2\overline{F}_2 \parallel \overline{C}_2)\}))(\sigma) =$
  (by the definition of $\rhd_{\alpha_1}$ and $\rhd_{\alpha_2}$ for $f' = \lambda\beta\tau.\tau \neq \delta \rightarrow \rho, \overline{F}_1$)
  $(\pi\ (\ (\overline{C}_1 \rhd_{\alpha_1} \lambda\sigma.\{[\alpha_1, Q\theta_{\alpha_1}] \cdot f'\} \parallel (\overline{C}_2 \rhd_{\alpha_2} \lambda\sigma.\{[\alpha_2, !t] \cdot \overline{R}_2\overline{F}_2\})\ ))(\sigma) =$
  $(\pi\ ((\overline{C}_1 \rhd_{\alpha_1} \mathcal{D}[\![Q?]\!]\alpha_1\overline{R}_1\overline{F}_1) \parallel (\overline{C}_2 \rhd_{\alpha_2} \mathcal{D}[\![!t]\!]\alpha_2\overline{R}_2\overline{F}_2)))(\sigma) =$
  $(\pi\ (\mathcal{C}[\![< \alpha_1, Q?; R_1 : F_1 >: C_1]\!] \parallel \mathcal{C}[\![< \alpha_2, !t; R_2 : F_2 >: C_2]\!]))(\sigma) = \pi \circ \mathcal{G}[\![\Gamma]\!]\sigma.$

- If $\Gamma = \{< \alpha_1, Q?; R_1 : F_1 >: C, < \alpha_2, !t; R_2; F_2 >: C_2\}$ and $t \neq \delta$ then
  $\Psi(\pi \circ \mathcal{G})[\![\Gamma]\!]\sigma =$ (with $g' \equiv (t\theta_{\alpha_1} = t\theta_{\alpha_2} \,\square\, u(t\theta_{\alpha_1}); Q?)$)
  $\bigcup\{\dot{\sigma} \circ (\pi \circ \mathcal{G})[\![\{< \alpha_1, g'; R_1 : F_1 >: C_1, < \alpha_2, R_2 : F_2 >: \Xi, C_2\}]\!]\sigma\} =$
  (by the property of π)
  $(\pi\ \lambda\sigma.\{\sigma \cdot \mathcal{G}[\![\{< \alpha_1, g'; R_1 : F_1 >: C_1, < \alpha_2, R_2 : F_2 >: \Xi, C_2\}]\!]\})(\sigma) =$
  (by definition 8.3.4)
  $(\pi\lambda\sigma.\{\sigma \cdot (\mathcal{C}[\![< \alpha_1, g'; R_1 : F_1 >: C_1]\!] \parallel \mathcal{C}[\![< \alpha_2, R_2 : F_2 >: \Xi]\!] \parallel \mathcal{C}[\![C_2]\!])\})(\sigma) =$
  (taking $f = \lambda\beta t.t \neq \delta \rightarrow \overline{C}_1 \rhd_{\alpha_1} \mathcal{F}[\![g'; R_1 : F_1]\!]\alpha_1, \overline{C}_1 \rhd_{\alpha_1} \overline{F}_1$)
  $(\pi\ \lambda\sigma.\{\sigma \cdot (f(\alpha_2)(t) \parallel \overline{R}_2\overline{F}_2 \parallel \overline{C}_2)\})(\sigma) =$ (by the definition of merging)
  $(\pi\ (\lambda\sigma.\{[\alpha_1, Q\theta_\alpha?] \cdot f\} \parallel \lambda\sigma.\{[\alpha_2, !t] \cdot (\overline{R}_2\overline{F}_2 \parallel \overline{C}_2)\}))(\sigma) =$
  (by the definition of $\rhd_\alpha$ with $f' = \lambda\beta t.t \neq \delta \rightarrow \mathcal{F}[\![g'; R_1 : F_1]\!]\alpha_1, \overline{F}_1$)
  $(\pi\ (\ (\overline{C}_1 \rhd_{\alpha_1} \lambda\sigma.\{[\alpha_1, Q\theta_{\alpha_1}] \cdot f'\}) \parallel (\overline{C}_2 \rhd_{\alpha_2} \lambda\sigma.\{[\alpha_2, !t] \cdot \overline{R}_2\overline{F}_2\})\ ))(\sigma) =$
  $(\pi\ ((\overline{C}_1 \rhd_{\alpha_1} \mathcal{D}[\![Q?]\!]\alpha_1\overline{R}_1\overline{F}_1) \parallel (\overline{C}_2 \rhd_{\alpha_2} \mathcal{D}[\![!t]\!]\alpha_2\overline{R}_2\overline{F}_2)))(\sigma) =$
  $(\pi\ (\mathcal{C}[\![< \alpha_1, Q?; R_1 : F_1 >: C_1]\!] \parallel \mathcal{C}[\![< \alpha_2, !t; R_2 : F_2 >: C_2]\!]))(\sigma) = \pi \circ \mathcal{G}[\![\Gamma]\!]\sigma$
  as desired. □

Corollary 8.3.7 $\mathcal{T} = \pi \circ \mathcal{G}$

Part III

Implementation

Chapter 9

An implementation model for DLP

- But if what we desire is to increase our knowledge rather than cultivate our sensibility, we should do well to close all those delightful books; for we shall not find any instruction there upon the questions which most press upon us -
George Santayana, The Sense of Beauty

Having dealt with the design and semantics of DLP we are ready to look at the implementation of our language. In this chapter we will give an introductory account of the implementation model employed. The model that we will sketch covers all the constructs introduced in chapter 2. It is more general than the computation model presented in part II. A noticeable difference is that we allow nested accept statements, whereas in giving the semantics for DLP_2 we required accept statements to occur in the constructor process, which prevents such nesting. See section 3.3 for a more detailed account.

The basic notions that we deal with are, again, *objects*, encapsulating data; *processes*, created to evaluate goals; *communication* between processes; and *inference* by which the evaluation of goals takes place.

We will restrict ourselves here to a treatment of active objects, and communication by rendez-vous. We will not discuss inheritance, since inheritance, because of its static nature, does not influence the behavior of an object in any dynamic way. An overview of the prototype implementation is given in section 9.5.2. We have also included an introduction to the implementation language in section 9.5.1. In chapter 11 we will present a much more detailed description, including passive objects, inheritance and communication over channels.

For our prototype implementation we have chosen to use a Prolog interpreter

derived from a formal continuation semantics for Prolog, given in [Allison, 1986]. We describe this interpreter in chapter 10. We have not used the semantics given in the previous part for implementing DLP, simply because at the time we did not have any of these. The semantics given in part II have been developed afterwards to establish the soundness of the language. We must however remark that the semantics we did use only covers the Prolog inference part. It may be replaced by a more efficient Prolog interpreter without affecting the structure of the prototype in any significant way.

9.1 Objects

In DLP objects are given by a declaration such as

ctr

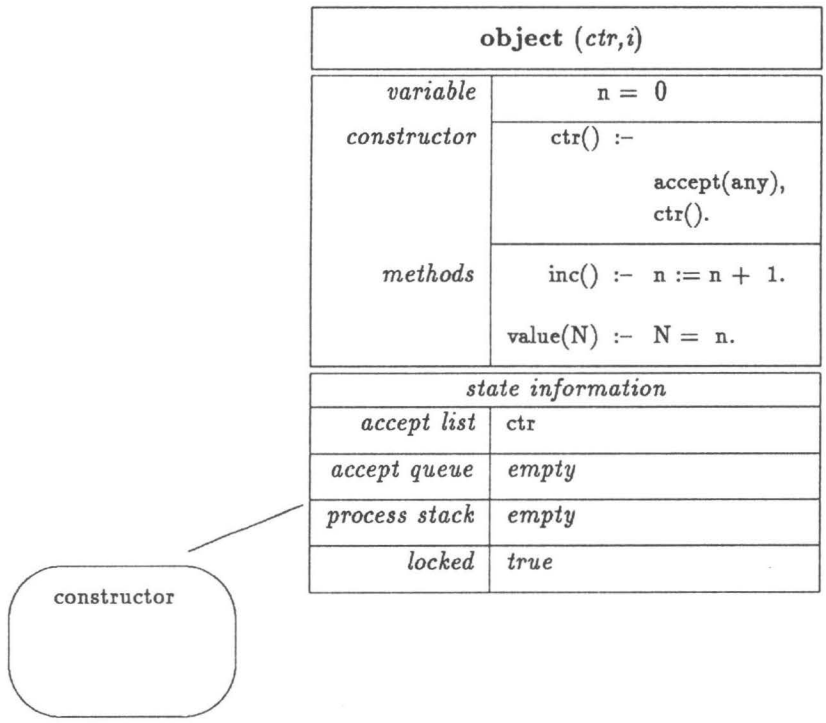
```
object ctr {
var n = 0.
ctr() :- accept(any), ctr().
inc() :- n := n + 1.
value(N) :- N = n.
}
```

Each instance of such an object is uniquely identified by an *instance number.* Each instance has a private copy of the non-logical variables and (conceptually) of the clauses declared for the object. In addition, each active object has a number of attributes, invisible to the programmer, that contain the *state* of the object determining the acceptance of method calls.

These attributes are

- the *accept list* - that contains the accept expressions by which it is decided whether a method call is acceptable,
- the *accept queue* - that contains the method calls for which the evaluation is suspended, and
- the *process stack* - that records the processes for which an accept statement is evaluated.

A stack of processes is maintained since accept statements may occur in a nested fashion. Apart from the attributes mentioned, a boolean variable *locked* is kept, to indicate whether an object is willing to receive any new requests for engaging in a rendez-vous.



In the figure above we have pictured the creation of a new active instance of the object declared above. The accept list is initially set to the constructor of the object. Both the accept queue and the process stack are initially empty. As soon as the constructor process is created the newly created object is locked and remains locked until it is notified of an accept statement.

The protocol by which method calls are handled will be illustrated by an example. We represent the state of an instance of the counter object by a tuple

(*n, accept list, accept queue, process stack, locked*)

where n is the non-logical variable of a counter. Suppose that we have as a goal

:- C = new(ctr()), C!inc(), C!value(X).

Evaluating the atom C = new(ctr()) results in an object (*ctr*, 1) for which we have

(1) (0, *ctr, empty, empty, true*)

as the state. On reaching the statement accept(any) the state becomes

(2) (0, *any, empty, constructor, false*)

in order to enable the object to accept a method call. The call C!inc() will now be accepted, and during the evaluation of inc() the state of the counter object will become

(3) (1, *any, empty, constructor, true*)

After the evaluation of inc() the state becomes

(4) (1, *any, empty, empty, true*)

to enable the constructor to continue. When the constructor then reaches the statement accept(any), after a recursive call to ctr(), the state becomes

(5) (1, *any, empty, constructor, false*)

enabling the acceptance of the call value(X). We have summarized these changes of state in the table below.

	(1)	(2)	(3)	(4)	(5)
n	0	0	1	1	1
accept list	*ctr*	*any*	*any*	*any*	*any*
accept queue	*empty*	*empty*	*empty*	*empty*	*empty*
process stack	*empty*	*constructor*	*constructor*	*empty*	*constructor*
locked	*true*	*false*	*true*	*true*	*false*

When an accept statement occurs in a process that is not the constructor process a similar procedure is followed. The process evaluating the accept statement is suspended and pushed on the process stack.

As the example above illustrates, if the accept queue contains no request satisfying an acceptance condition, the object is unlocked and waits for a method call. The object will consider requests for a method call only if it is not locked. If it then receives a request that does not satisfy the acceptance conditions this request is put in the accept queue. For an example of an object where this may happen look at the semaphore presented in section 2.3. When notified of an accept statement, the object first inspects the accept queue to see if there are any requests satisfying the new acceptance conditions. When no such request is available the object becomes unlocked, and waits for a method call.

Acceptance of method calls is determined by the accept expressions contained in the accept list. For a simple accept expression, just naming the method, it suffices to check the predicate name of the call. Conditional accept expressions require a more involved procedure to decide whether a call may be accepted. Recall that a conditional accept expression is of the form

method : *guard* → *goal* (see section 2.5)

To be accepted, a method call must unify with *method* and the *guard* must hold. The process created to establish this has to return a single result, recording the bindings of the variables in *method* and *guard*. If the test is successful, these bindings are taken over by the process created to evaluate the *goal*. The bindings

resulting from the test and the bindings resulting from the evaluation of the goal are communicated to the process in which the accept statement occurred. If the evaluation of the goal fails this process is sent only the bindings that result from unifying the call with *method* and evaluating the *guard*. The process that called the method receives only the answer substitution, that is the bindings of the variables uninstantiated at the time of the call. Only the bindings of these variables are undone when backtracking occurs. The binding of the variables in the accept statement are committed to when the test succeeds.

Method calls are accepted when they satisfy one of the expressions occurring in the accept list. If a method call does not satisfy any acceptance condition it is put in the accept queue to await a change of the acceptance conditions. The procedure sketched above guarantees a fair treatment of method calls, since incoming requests are either granted or stored in the accept queue in the order that they are received. No call has to wait indefinitely long under favorable acceptance conditions. C.f. [America, 1989b].

9.2 Processes

A process is created for each request to an object to evaluate a goal. The evaluation of a goal is done by a (more or less standard) Prolog inference engine. Each such inference is, in other words, accompanied by a so-called *evaluation process* taking care of communicating with the environment, if necessary. The evaluation process also serves to administrate where to cut off the search and to send tracing information to the user.

Each process refers to an object, from which it derives its functionality. When a process is created, it is given a pointer to the object for accessing non-logical variables and to delegate the evaluation of the accept statement.

To keep track of the whereabouts of the evaluation of the goal, a process keeps a *state* parameter that may have one of the following values:

BUSY - the inference is still going on,

PEND - an answer has been produced, but not necessarily the last answer,

WAIT - the last answer has been produced, and

STOP - the last answer has been sent to the caller.

Apart from the state parameter, we use two attributes to store the resulting answers.

- *result* - to store the most recent answer,
- *solutions* - which contains all the answers that have been produced.

When a method is called in DLP, a resumption request of the form $Q?$ is dynamically inserted to replace the call. The variable Q is bound to the evaluation process

handling the call. An evaluation process stores all solutions produced thus far to enable a process to backtrack repeatedly over these solutions. See also section 2.6. A *resumption* is an ordinary Prolog goal that must be executed by the caller of a method to effect the bindings produced by evaluating the call.

In order to deliver the proper resumption, the evaluation process has an attribute

- *goal* - for storing the initial goal

that is used to create a resumption when the process is asked for an answer. What resumption is returned depends on the *state* of the process and the contents of the *result* attribute. Assuming that we have an appropriate translation of the values of the attributes *goal*, *result* and *solutions* to Prolog terms, resumptions may take the following form

fail - the failing resumption, that is delivered when the state parameter indicates WAIT and the *result* is empty;

goal = *result* - the unifying resumption, that is delivered when the state indicates either PEND or WAIT and the *result* is not empty;

member(*goal*, *solutions*) - the backtracking resumption, that is deliver when the state indicates STOP.

The failing resumption, when executed by the process requesting an answer, results in failure, evidently. Both the unifying and the backtracking resumption bind the variables occurring in *goal* to their instantiations in respectively *result* and *solutions*. The latter resumption, moreover, backtracks over all solutions generated. Backtracking resumptions may be delivered when goals occur in between a call and a resumption request. C.f. section 2.6.

9.3 Communication

Communication between processes occurs on the occasion of calling a method and returning an answer. In the part dealing with the semantics of DLP we have described the synchronization that takes place between processes during such a rendez-vous. See sections 6.5.1 and 7.3.

Accepting a method call When a process encounters an accept statement, control is delegated to the object to which the process refers. The process itself is pushed on what we have called the process stack of the object. When a method call, satisfying the acceptance conditions, is accepted, a new process is created to evaluate the call. The process calling the method receives a pointer to this newly created evaluation process and states a resumption request to wait for an answer.

The evaluation process receives a pointer to the process in which the accept statement occurred. When the evaluation process comes in state PEND or WAIT the accepting process is told to continue. By then it has been popped from the process stack.

Returning an answer When an answer has been produced, the evaluation process waits for a request to return a resumption. The kind of resumption that is returned depends on the state of the evaluation process and the result that is produced. After sending the resumption, its changes its state into BUSY if it was in state PEND and to STOP if it was in a WAIT state. The process requesting the resumption checks whether the evaluation process was in state PEND when sending the resumption. If so, then the resumption executed by the requesting process will be of the form $(R; Q?)$, where R is the resumption received, to allow backtracking over alternative answers. In case the evaluation process was not in a PEND state, the resumption will simply be R.

9.4 Inference

Goals are evaluated using a sequential Prolog interpreter. The object, on behalf of which the evaluation occurs, creates an evaluation process to handle the inference. To deal with backtracking, the evaluation process (conceptually) possesses a copy of the clauses declared for the object to which it refers.

For our prototype implementation of DLP we have chosen to employ a technique for compiling clauses, derived from a formal continuation semantics originally given in [Allison, 1986]. Our compilation scheme has proven its usefulness in particular in the implementation of inheritance.

9.5 The prototype

In the next chapter we will show how we derived the code for the sequential Prolog interpreter from the continuation semantics given in [Allison, 1986]. In chapter 11 we will discuss the implementation of the prototype in detail. Here , however, we wish to give an overview of the structure of our prototype implementation of DLP. But first, we will devote some attention to the implementation language that we have used for developing the prototype.

9.5.1 The implementation language

For the implementation of the prototype we have used a syntactic variant of the parallel object oriented language POOL-X as defined in [America, 1989]. We will use the name POOL* for our dialect. We wish to remark that our variant conforms to the restrictions imposed in [Beemster, 1990].

Since our language DLP has partly been inspired by the language POOL, a forerunner of POOL-X, described in [America, 1987], the obvious similarities between these languages should not come as a surprise.

Those familiar with POOL-X should have no trouble reading the code fragments presented in this part and the appendix, when taking into account that we use lower case keywords and curly brackets { and } instead of BEGIN and END. Moreover, since we do not like to speak of routines, we have also replaced the keyword ROUTINE by the keyword *fn*, to indicate a function definition. Another

difference with POOL-X is that we have allowed ourselves the use of an adapted version of the C-preprocessor.

For those not familiar with POOL-X we explain the main features of our implementation language. The most interesting aspect of POOL* is that it allows to create dynamically an indefinite number of active objects that may communicate with each other by means of a synchronous method call, resembling the Ada rendez-vous.

Objects are taken to be instances of a class. A class declaration generically describes the behavior of each object that is an instance of that class. The language POOL* is strongly typed, in the sense that each entity, including integers and strings, must be an instance of some class.

Class declarations in POOL* have the following form, with keywords written in italic:

declaration

class name
var instance variables
newpar(new parameters)
init initialization part *tini*
functions
methods
body body part *ydob*
end name

Each class has a name that must start with a capital. A class declaration may specify a number of instance variables. In the sequel we will speak of instance variables as the attributes of an object, since instance variables contain the private data of an object, that may be accessed and modified by the methods defined for that class of objects. The new parameters are a special kind of instance variables. They may not be modified after being initialized. When creating a new instance of a class, the value for such a parameter must be given as an argument. Apart from the new parameters, an initialization part may be specified that will be executed, when creating a new object, before the body containing the own activity of the object will start to execute. Each class declaration may contain a number of function definitions. These functions cannot access the instance variables of objects belonging to the class. The class merely acts as the scope within which the function is known. In contrast, the methods defined for a class do have access to the local data of an object as stored in the instance variables. In effect methods are the exclusive way to access and modify the encapsulated data. Finally, we may encounter the specification of the own activity of an object in the body part of a class declaration. We remark that in an actual class declaration not all of the components listed above need to occur.

As an example of a class declaration in POOL* consider the definition of a semaphore. See section 2.3 for the declaration of a semaphore in DLP.

Sema

```
class Semaphore
var n:Integer
newpar(n1:Integer)
init n := n1 tini
method v():Sema { n := n + 1; result self }
method p():Sema { n := n - 1; result self }
body do
 if n = 0 then answer(v) else answer(p, v) fi
od ydob
end Semaphore
```

The declaration of class *Sema* shows all the aspects of a class declaration mentioned, except the definition of functions.

Statements in POOL* have a Pascal-like syntax. The keyword *result* indicates what to return as a result of applying a method. The expression *self* represents the object to which the method is applied. The statement *do...od* in the body part may be read as *while true do ... od.* The statements *answer*(v) and *answer*(p, v) are like the accept statements in DLP. When executing such a statement, the execution of own activity of the semaphore object will be interrupted, to await the call for a method allowed by the answer statement.

Creating a new instance of *Sema* is, having a variable x of type *Sema*, achieved by stating

$$x := Sema.new(1)$$

which calls the function *new* of *Sema*, as implicitly defined by the new parameters and the initialization part. We may then subsequently call the methods p and v of the newly created object by stating respectively $x!p()$ and $x!v()$.

Function definitions in POOL* are given as in the example below

```
fn max(i,j:Integer):Integer
{
if i > j then result i else result j fi
}
```

Such a function definition must occur within a class declaration, say of a class X. The function may then be called as, for example, $X.max(3, 7)$.

Another, in effect rather essential, feature of our implementation language is the possibility to define functions dynamically, as illustrated by the statement

```
inc := fn(i:Integer):Integer { result i + n }
```

where we assume that n is an integer variable declared in the environment surrounding the statement. The result of assigning the function definition to the variable inc is a function that increments an integer with the value n had at the moment of defining the function.

Temporary variables may be used in both function definitions and method definitions. As an example consider the function definition

```
inc := fn(i:Integer):Integer temp x:Integer { x := i; result x + n }
```

that stores its argument i first in the temporary variable x, declared by the keyword *temp*, before returning its result. Temporary variables may also be directly initialized to a value.

Special methods are provided when instance variables are annotated. For instance, the declaration

```
class F
var f put get : String
...
end F
```

implicitly declares the methods

```
method put_f(s:String):F { f := s; result self }
method get_f():String { result f }
```

that may be used to modify and access the value of the instance variable f.

As syntactic sugar POOL* provides the notation x@f with, for x a variable of type F, the meaning x!put_f(s) if x@f occurs on the left hand side of an assignment, as in x@f := s , and x!get_f() otherwise. These abbreviations may also be used when the corresponding methods are declared explicitly. As syntactic sugar POOL* also allows the form q[n], for objects q and n, to be used for the method call q!put1(n,v) when it occurs on the left hand side of an assignment q[n] := v and for the call q!get1(n) otherwise. The corresponding methods for q must be defined by the user. This notation may however also be used for arrays, in the usual way.

Alias definitions may be used in POOL* to abbreviate a complex type definition or a parametrized class name. For instance the definition

```
alias Intfun = fn (Integer): Integer
```

specifies Intfun to be a function type, with integers as its domain and range.

Union types allow the use of a kind of tagged variant records. They are conveniently defined by using an *alias* definition. For example

```
alias T = Union3(A,V,F)
```

defines the type T as the union of the types A, V and F. To create an object of type T, say from an object of type A, one must indicate the variant number of that object. As an example, having an object v of type V, the statement

```
t := T.new2(v)
```

creates an object of type T, consisting of v. To determine what kind of object t actually contains, one must use the method call

t@*switch*

which delivers the variant number 2 for the case above. To access the object contained in a union, one must use the variant number in a method call like

t@2

which delivers the object v in the case at hand.

Global objects and functions may be declared by using the keyword *global* as in

global knot := Knot.new()

that stores an instance of class Knot in the global variable *knot*.

Asynchronous method calls may replace synchronous method calls when there is no need to wait for an answer. For an object *io*, one may asynchronously call the method *print* by stating

io!!print("Hello world")

The method *print* may not have a result type, nor contain a result statement.

Units provide an additional scoping mechanism in POOL* . Each unit may contain a number of class declarations. Method definitions preceded by the keyword *internal* are known only within the unit. For POOL-X it is required to define both a specification unit, listing the classes, methods and functions visible to other units and an implementation unit, containing the actual declarations. We regard specification units as somewhat cumbersome and consequently have not included these in the listing of the code of the prototype.

Each unit must specify of what units it needs to know the specification. The declaration

use O Q

declares that the current unit uses, that is needs to know, the declarations contained in the units O and Q.

Pragmas of the form (* ALLOC at n *), for some integer n, are used to handle the allocation of instances of classes to processor nodes. The statement

O := O.new() (* ALLOC at n *)

where n is an integer variable defined in the environment, results in the creation of a new instance of class O allocated at processor node n.

Macro definitions may be used to define a number of additional abbreviations. Such definitions are preceded by the keyword *df*, as in

df m(X) = ...

Files may be included by commands of the form

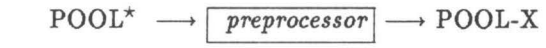
include N

that results in including the file N.h. Such inclusion commands are quite different from specifying the use of particular units, since file inclusion results in actually inserting the text of that file.

In addition to the language features discussed we use, in the next two chapters, conditional expressions of the form $b \longrightarrow e_1, e_2$ with the meaning *if b then* e_1 *else* e_2. This construct is not part of POOL* .

We wish to stress the fact that, apart from the commands for macro definitions and file inclusion, POOL* is a strict subset of POOL-X. We have not used exceptions, tuple-types or any of the esoteric syntactic sugar announced in [America, 1989a].

From a POOL* program we obtain a POOL-X program as indicated in the diagram below.



In its turn the POOL-X program is compiled to C by using the compiler described in [Beemster, 1990].

Concluding the description of our implementation language, we will note some of the differences between DLP and the implementation language POOL* . First of all DLP is untyped, whereas POOL* is strongly typed. Secondly, POOL* does not allow inheritance, while DLP supports inheritance by code-sharing. In the third place, the notion of an object and a process are identical for POOL* . For DLP, on the other hand, multiple processes may be active concurrently for a single object. And as a fourth, perhaps the major, difference between the two languages, DLP supports backtracking over the answers produced in a rendez-vous, whereas POOL* , because of its imperative character, naturally does not.

9.5.2 The implementation of the prototype

The prototype is subdivided in POOL* units, as listed below. In chapter 11 we will describe the main features of each of these units. Full detail is given in the appendix, containing the code.

- *Types and abbreviations*: We have collected the definition of the types used throughout the program in this unit. We have also put the most frequently occurring abbreviations in it. Some of the types are defined as abbreviations. See 11.1 and A.1.

- *Terms*: Since terms are the primary data structure on which the functions defined in the program operate we have devoted a separate unit to them. Both objects and processes may be represented by DLP terms. The unit contains definitions for *constructors*, to make terms, *selectors* to access the constituent parts of a term, and *tests* to be able to perform a case analysis on a term.

- *The sequential Prolog interpreter*: In the next chapter we will describe in detail how we have derived the sequential Prolog interpreter from a continuation semantics for a Prolog like language. There is no need to say that, in terms of efficiency, we could significantly improve on the execution speed by using a high performance Prolog implementation. C.f. [Warren, 1983].

- *Objects*: This unit defines the functionality of objects with respect to the acceptance of method calls. It also defines the attributes of DLP objects, such as the list of non-logical variables and the inheritance lists. In addition it contains the functions effecting inheritance and the initialization of DLP objects.

- *Processes*: The evaluation of goals requires some global administration and a facility to handle the declaration of objects and commands to consult a file. To the tasks of an evaluation process belong, apart from the communication with other objects and processes during a rendez-vous, the handling of cuts and to report trace information to the user.

- *Non-logical variables and channels*: Together with the definition of non-logical variables, this unit contains the protocol governing synchronous communication over channels.

- *The initial database*: The simplification function defined in this unit, is in particular important for the treatment of special forms used for the creation of objects and processes. This unit contains also our interpretation of the equality predicate and assignment. In addition the DLP system predicates are defined here. Part of these are included for compatibility with Prolog. Others are used as primitives for communication over channels and for engaging in a rendez-vous.

- *Utilities*: We have collected here a number of auxiliary definitions. Also this unit provides the low level term manipulation functions to support the handling of terms.

We have omitted a description of the units for reading in and parsing DLP programs.

Chapter 10

Deriving a Prolog interpreter

- The question is the story itself, and whether or not it means something is not for the story to tell. -
Paul Auster, The New York Trilogy

The use of formal semantics may vary from checking whether the constructs for a particular language are well-defined, in that they allow a consistent interpretation, to being a guideline for the correct implementation of the language. In developing DLP we have taken the semantics of a subset of what turned out to be the final language as a starting point for developing additional constructs, illustrating yet another usage of semantics.

To implement our Prolog interpreter we have developed a compilation scheme for converting clauses to functions by adapting a continuation semantics for Prolog (without cut) given in [Allison, 1986]. Using function composition to combine the functionality of separately compiled objects, we were able to implement inheritance in a rather elegant way. In this chapter we will present our adapted version of the semantics given in [Allison, 1986], and illustrate how to derive the code from the semantic equations.[1] Since our primary intention here is to elucidate the transition from semantic equations to code, we have only indicated how to provide

[1]The requirement imposed on the implementation language by the semantics we employed is that it must be possible to treat functions as first class objects. Although the use of such features definitely does not give the most efficient coding, in this stage of development conciseness is to be preferred to enable experimentation and to understand what has been done. See section 9.5.1 for a description of the implementation language used.

a mathematical justification, without going into details.

Terms lay at the basis of the implementation of DLP. We use an extended notion of terms to include, apart from constants, logical variables and applicative terms, also *atoms* (in a Prolog sense), *goals* and *clauses*. We define terms $t \in T$ by

$$t ::= \varepsilon \mid v \mid f(t_1, t_2)$$

where ε is called the empty term, v is an element of the set of logical variables V and f is an arbitrary function symbol from a set A. Constants may considered to be terms of the form $f(\varepsilon, \varepsilon)$. A term with one argument is written as $f(t_1, \varepsilon)$. An n-ary function term is represented as $f(t_1, (t_2, (..., t_n)))$, where the arguments $t_2, ..., t_n$ are represented as a compound term with a comma as a function symbol, written infix. Our definition of terms reflects the definition of terms used in the actual implementation. Algorithms on terms are simplified by this encoding, since recursion on subterms can be written out directly, without being mediated by lists of argument terms. We stress the fact that the set of function terms A we use for extended terms includes connectives such as the comma, predicate symbols that we may encounter in atoms and ordinary function symbols.

The definition of terms given above allows us to represent atoms a defined by

$$a ::= p(t_1, ..., t_n) : \ p \text{ a predicate symbol}, n \geq 0$$

as terms. Similarly, goals g defined by

$$g ::= \varepsilon \mid a, g$$

may be represented by terms, simply by taking the conjunctive comma as a special function symbol. Finally, (lists of) clauses cl defined by

$$cl ::= \varepsilon \mid (a :\!- g); cl$$

are also clearly representable as terms.

A program is a fixed list of clauses, represented by $\overline{cl}$.

Renaming and substitutions. We use *environments* $e \in E = V \rightarrow V$ to rename variables. Each logical variable is initially represented by a natural number indicating the position it first occurs in a clause. In effect, we take $V = N$ for N the set of natural numbers. Since logical variables are local to the clause in which they occur, it suffices to keep track of the number of variables in use and raise this count with the number of different variables occurring in the clause that is tried to solve a goal. Given a renaming function e, the renaming is effected by using a map of type $E \rightarrow T \rightarrow T$ to apply e to a term t, which we write as $\hat{e}(t)$. For determining the (maximum) number of variables in a term, we define a *norm* function that delivers the maximum of the variable numbers occurring in a term, be it a goal or a clause. The notation for applying the norm function is $\mid t \mid$.

A *substitution* is a function of type $S = V \rightarrow T$ that returns a term for a variable. Only for finitely many variables the term is different from the variable itself. Given a substitution $s \in S$, we write $\hat{s}(t)$ to denote the application of s to a term t.

10.1 Evaluation

Suppose that we have a goal, say $p(X)$, and that we have a program consisting of the following clauses.

```
p(1).
p(2).
```

Then, when stating $p(X)$ as a goal, we expect the answers $p(1)$ and $p(2)$, obviously. What we are actually interested in are the answer substitutions given for, in this case, the goal $p(X)$. The terms $p(1)$ and $p(2)$ represent the possible answer substitutions for $p(X)$, with respect to the program given above. Below, we will represent answer substitutions by instantiated terms.

Before giving a more detailed description of the semantics on which we based the implementation we wish to give an outline of the components involved, and a characterization of the type structure of the functions used.

As the result domain of the function $\mathcal{E}$, that we use for evaluating goals, we take $R = T^\omega \cup T^\infty$, the set of finite and infinite sequences over T. We will stick to the use of ε to denote the empty term, since we do not need an explicit notation for the empty sequence of terms.

As a metric on R we use the standard metric on sequences, defined by

$$d(w_1, w_2) = 2^{-sup\{n \mid w_1(n)=w_2(n)\}}$$

where $w(n)$ denotes the prefix of length n in case the length of w exceeds n and w otherwise. See section 6.1.1.

We introduce an operator $\cdot : R \times R \to R$ that concatenates sequences of terms. We remark that for contractivity to hold we cannot ignore the empty term ε occurring in a sequence, however informally we use the convention that $\varepsilon \cdot w = w = w \cdot \varepsilon$.

The function $\mathcal{E}[\![\cdot]\!]$ decomposes a non-empty goal t in an atom a and the remainder of the goal g. It then calls the database $\mathcal{C}[\![\overline{cl}]\!]k_0d_0$, compiled from the program $\overline{cl}$, to evaluate the atom a, after having prepared a suitable continuation to evaluate the remainder g. See section 10.2 for a characterization of $\mathcal{C}[\![\overline{cl}]\!]k_0d_0$.

We use continuations of type

$$C = N \times S \to R$$

to store the computation needed to evaluate the remainder of a goal. A typical continuation c is called as $c(n, s)$, where n is a natural administrating the highest variable number in use and s a substitution of type $V \to T$ that records the substitution computed while evaluating the goal.

The function $\mathcal{E}[\![\cdot]\!]$ has a type structure as expressed in

$$\mathcal{E}[\![\cdot]\!] : T \to C \to N \to S \to R$$

Apart from a goal t to evaluate we need to provide $\mathcal{E}[\![\cdot]\!]$ with a continuation $c \in C$ and parameters $n \in N$ and $s \in S$ to be used, possibly modified, for calling the continuation c.

Database functions are of type

$$D = T \times C \times N \times S \to R$$

analogous to the type of $\mathcal{E}[\![\cdot]\!]$. As will be seen in section 10.2, functions $d \in D$ are the result of compiling (a list of) clauses.

For a clause $a :- g$, such a function replaces a goal atom a' unifying with a by the goal g, in effect by recursively calling $\mathcal{E}[\![g]\!]$ with the appropriate continuation parameters, while modifying the substitution according to the unifier of a and a'. In section 10.3 we will discuss how we deal with unification in our continuation semantics.

To illustrate the use of the function $\mathcal{E}$ we may remark that, ignoring empty terms,

$$\mathcal{E}[\![p(X)]\!]cns_0 = p(1) \cdot p(2)$$

where we assume that c is the initial continuation $\lambda ns.\hat{s}(p(X))$, $n\ (= 1)$ the number of variables in $p(X)$ and s_0 the initial substitution, that acts as the identity on all variables.

The equations defining $\mathcal{E}[\![\cdot]\!]$ reflect the previous discussion.

$\mathcal{E}$

$\mathcal{E}[\![\cdot]\!] : T \to C \to N \to S \to R$ for goals, with

$\mathcal{E}[\![\varepsilon]\!] = \lambda cns.c(n, s)$

$\mathcal{E}[\![a, g]\!] = \lambda cns.\overline{d}(a, c', n, s)$ where $c' = \lambda ns.\mathcal{E}[\![g]\!]cns$ and $\overline{d} = \mathcal{C}[\![\overline{cl}]\!]k_0 d_0$

An empty goal simply results in calling the continuation. For non-empty goals the continuation argument c is modified to take care of the remainder of the goal.

The definition of the function *eval*, in its turn, reflects the equations for $\mathcal{E}$ in an obvious manner.

eval

```
fn eval(t:T,c:C,n:N,s:S):R
{
if t = ε then c(n,s)
elsif t = (a,g) then
    dotail := fn (n1:N, s1:S) { result eval(g,c,n1,s1) };
    result d̄(a,dotail,n,s)
}
```

The code straightforwardly results from converting the lambda terms occurring in the equations for $\mathcal{E}$ to the format imposed by our implementation language. We assume that the database corresponding to the program $\overline{cl}$ is stored in a global

function $\overline{d}$. We remark that if t does not conform to the syntactic conventions for goals we may consider the result undefined.

10.2 Compilation

We have now arrived at the point where we may look more closely at the procedure by which clauses are compiled to what we have called a database function. As a preparation, look at the empty database d_0, that we define as

$$d_0 = \lambda tcns.\varepsilon$$

which delivers the empty term, irrespective of its actual parameters.

Having a clause of the form $a :- g$, the database resulting from compiling this clause must deliver whatever evaluating g delivers, assuming that we have a goal atom a' that unifies with a. To effect this, we have defined a unification function

$$\mathcal{U}[\![\cdot]\!][\![\cdot]\!] : T \to T \to C \to N \to S \to R$$

that takes two terms, in the case sketched above the goal atom and the head of a clause, a continuation containing the computation for evaluating the body of the clause, a variable count and a substitution. Now suppose that, in compiling a list of clauses, we have arrived at a clause $a :- g$. In the general case we may, for a goal t, have tried clauses occurring before this particular clause; and likewise clauses may follow that contribute to the solutions for this (sub) goal. Just assume that we have stored the effect of the preceding clauses in a database function d, then the resulting database d' will be like

$$d' = \lambda tcns.(d(t, c, n, s) \cdot \mathcal{U}[\![t]\!][\![a]\!]c'n's)$$

for some continuation c' and variable count n'. Note that backtracking is taken care of by appending the result of using a clause to the result of the database encoding all previous clauses. When failure occurs the empty term ε is delivered, which consequently disappears. When trying an alternative clause the substitution need not be changed.

To deal with the clauses following the clause $a :- g$ we need an auxiliary continuation of type

$$K = D \to D$$

to compute the combined database function. When compiling a list of clauses $(a :- g); cl$, assuming that we already have compiled the clauses preceding $a :- g$ into a function d, we create a continuation $k \in K$ for compiling the clauses cl taking d', that is d augmented with the functionality of $a :- g$, as a parameter.

It is obvious that initially we must take as a continuation

$$k_0 = \lambda d.d$$

that acts as the identity on database functions. We list the equations for $\mathcal{C}$ below.

$\mathcal{C}$

$\mathcal{C}[\![\cdot]\!] : T \to D \to K \to D$ for clauses, with

$\mathcal{C}[\![\varepsilon]\!] = \lambda dk.k(d)$

$\mathcal{C}[\![(a :\!- g); cl]\!] = \lambda dk.\ k'(\lambda tcns.(d(t, c, n, s) \cdot \mathcal{U}[\![t]\!][\![\hat{e}(a)]\!]c'n's))$

where

$c' = \lambda ns.\mathcal{E}[\![\hat{e}(g)]\!]cns$

$n' = n + \mid a :\!- g \mid, e = \lambda v.v + n$

$k' = \lambda d'.\mathcal{C}[\![cl]\!]d'k$

Naturally, the renaming of variables must be taken care of, as has been done above.

The equations for $\mathcal{E}$, $\mathcal{C}$ and $\mathcal{U}$ are mutually recursive. To establish that these functions are mathematically well-defined, we should investigate whether a contractivity argument of the kind introduced in chapter 6 applies.

As we will illustrate below, compiling a program results in a series of attempts to unify the head of a clause with a particular goal, in the order the clauses occur in the program. Applying the unification function results in either the empty term or in the results of applying $\mathcal{E}$ to the body of the clause. Since we start with d_0, delivering the empty term ε for any goal, each application of $\mathcal{C}$ adds a term to the result, thus insuring contractivity.

To provide some intuition we will illustrate how this scheme works for the example with which we started section 10.1. As clauses, we have the unit clauses $p(1)$ and $p(2)$ that we write as

$p(1) :\!- \varepsilon.$
$p(2) :\!- \varepsilon.$

We start with the initial database $d_0 = \lambda tcns.\varepsilon$ and the initial continuation $k_0 = \lambda d.d$ to compute the database $\mathcal{C}[\![(p(1) :\!- \varepsilon); ((p(2) :\!- \varepsilon); \varepsilon)]\!]d_0k_0$.

Compiling $p(1) :\!- \varepsilon$ results in

$$d_1 = \lambda tcns.d_0(t, c, n, s) \cdot \mathcal{U}[\![t]\!][\![p(1)]\!](\mathcal{E}[\![\varepsilon]\!]c)ns$$

which, informally, we take to be equivalent to

$$\lambda tcns.\mathcal{U}[\![t]\!][\![p(1))]\!](\mathcal{E}[\![\varepsilon]\!]c)ns$$

since $d_0(t, c, n, s)$ evaluates to ε. The continuation arguments c, n and s remain unmodified because $p(1) :\!- \varepsilon$ does not contain any variables.

To compile the clauses $(p(2) :\!- \varepsilon); \varepsilon$ we have created the continuation

$$k' = \lambda d'.\mathcal{C}[\![(p(2) :\!- \varepsilon); \varepsilon]\!]d'k_0$$

which may be written as

$$k' = \lambda d'.\lambda tcns.(d'(t, c, n, s) \cdot \mathcal{U}[\![t]\!][\![p(2)]\!](\mathcal{E}[\![\varepsilon]\!]c)ns)$$

Hence, since $\mathcal{E}[\![\varepsilon]\!]c$ is equivalent to c, we may write the database function that is the result of compiling the program $(p(1) :- \varepsilon); ((p(2) :- \varepsilon); \varepsilon)$ as

$$\lambda tcns.\mathcal{U}[\![t]\!][\![p(1)]\!]cns \cdot \mathcal{U}[\![t]\!][\![p(2)]\!]cns$$

which, in other words, successively tries to unify a goal t with $p(1)$ and $p(2)$. Of course, the unification function is defined such that it delivers ε whenever t does not unify with either $p(1)$ or $p(2)$.

Again, the code for the compilation function may be derived from the equations given above in a rather straightforward way. In our presentation, we will first treat the part where we deal with a single clause $a :- g$, while assuming that we have a database function d containing the functionality of the clauses preceding the currently inspected clause. We dynamically define a function *newd* in the way pictured below.

newd

```
newd := fn ( t:T, c:C, n:N, s:S):R
            temp n1:N
            {
            n1 := n + | a :- g |;
            e := fn (v:V):V { result v + n };
            body := fn (n2:N, s2:S):T
                    {
                    result eval(q, ê(g), c, n2, s2)
                    };
            result append( d(t,c,n,s) , unify(t, ê(a), body,n1,s))
            }
```

First of all we need to take care of renaming variables according to the variable count n, given as a parameter. We then create a continuation for evaluating the (renamed) body of the clause. We append the result of evaluating the goal t by d and the result of unifying t with the (renamed) head of the clause, taking the evaluation of the body of the clause as a continuation. We can now define the function *compile*.

compile

```
fn compile(t:T, d:D, k:K):D
{
if t = ε then result k(d)
elsif t = (a :- g); cl then
                    dotail := fn (d1:D) { result compile(cl, d1, k) };
                    newd ;
                    result dotail( newd )
}
```

The picture must be completed by replacing *newd* by the definition of the function *newd* given above.

10.3 Unification

Finally, we arrive at defining the unification function

$$\mathcal{U}[\![\cdot]\!][\![\cdot]\!] : T \to T \to C \to N \to S \to R$$

that performs the unification of a goal atom and the head of a clause, to call the continuation evaluating the body of the clause if the unification succeeds.

If two terms are syntactically equal the unification results in calling the continuation.

Unifying two terms may result in a substitution, modifying the already given substitution by binding variables to terms. For a variable v that becomes bound to a term t, the modification of a substitution s is effected by creating a substitution s' given as

$$s' = \lambda v'.v = v' \to t, s(v')$$

that is a variant of s with respect to v. Binding a variable to a term is allowed only when the variable is unbound, otherwise the term must be unified with the substitution value of the variable. This condition is expressed in the definition of θ as

$$\theta = \lambda vt.s(v) = v \to c(n, s'), \mathcal{U}[\![s(v)]\!][\![t]\!]cns$$

where s' is the substitution s modified to bind v to t. The function $\mathcal{U}$ is given by the equations below.

$\mathcal{U}$

$\mathcal{U}[\![\cdot]\!] : T \to T \to C \to N \to S \to R$ for pairs of terms, with

$$\mathcal{U}[\![t_1]\!][\![t_2]\!] = \lambda cns.\begin{cases} c(n,s) & \text{if } t_1 = \varepsilon = t_2 \\ \theta(v,t_2) & \text{if } t_1 = v \\ \theta(v,t_1) & \text{if } t_2 = v \\ \mathcal{U}[\![t_{11}]\!][\![t_{21}]\!]c'ns & \text{if } t_1 = f(t_{11},t_{12}) \\ & \text{and } t_2 = f(t_{21},t_{22}) \\ \varepsilon & \text{otherwise} \end{cases}$$

where $\theta = \lambda vt.s(v) = v \to c(n,s'), \mathcal{U}[\![s(v)]\!][\![t]\!]cns$

and $s' = \lambda v'.v = v' \to t, s(v')$

and $c' = \lambda n's'.\mathcal{U}[\![t_{12}]\!][\![t_{22}]\!]cn's'$

For function terms with identical function symbols the unification is applied to their first arguments, taking as a continuation the unification of their second arguments. If the attempt at unification fails, the empty term ε is delivered. In agreement with other implementations of Prolog we have omitted the occur check.

We leave it to the reader to define the function $unify(t_1, t_2 : T, c : C, n : N, s : S) : R$ implementing the equations for $\mathcal{U}$. See section 11.3.3.

10.4 Initialization

Summarizing our description thus far, we have defined a function $\mathcal{E}$ to evaluate goals making use of database functions compiled from the program by the function $\mathcal{C}$. As an auxiliary function we have defined the unification function $\mathcal{U}$ to compute a most general unifier and to continue the computation accordingly. For a goal g we must create an initial continuation

$$c_g = \lambda ns.\hat{s}(g)$$

and we may then evaluate g by calling $\mathcal{E}[\![g]\!]c_g ns_0$, where n is the maximum variable number occurring in g.

Processes In defining the functions $\mathcal{E}$, $\mathcal{C}$ and $\mathcal{U}$ we have proceeded from the assumption that we have a fixed program $\overline{cl}$. In the context of DLP however, we may encounter a collection of objects, each having a list of clauses. Moreover, for each object, a multiple of processes may be active evaluating a goal.

In order to extend the functionality of the interpreter, we have introduced evaluation processes q of type Q that accompany the evaluation of a goal. Such a process is given as a parameter to the function *eval* and the database functions resulting from compiling clauses.

- Evaluation processes store the database function used in evaluating goals. The stored database function combines the functionality of an object with the functionality of the objects it inherits. In the function *eval*, the database

$\overline{d}$ must be replaced by the call $q@d$, that delivers the database stored in the attribute d of q.

- Another usage of evaluation processes, as will be explained in detail in the next chapter, is to keep record of the occurrence of cuts and to cut off the search accordingly.

- More importantly, however, evaluation processes play an intermediary role in communicating the answer substitutions that result from evaluating a goal to the process that requested the evaluation. The initial continuation for a goal g must then be of the form $\lambda ns.q!yes(\hat{s}(g))$, in order to notify the process q of the computed answer substitution. We have sketched the protocol by which this takes place in the previous chapter, when discussing the computation model underlying DLP. In the next chapter we will deal with the details of the implementation of this protocol.

10.5 Composition

Enlarging our scope by introducing objects, each containing a list of clauses, that may inherit each others functionality, the question arises how to combine the functionality of these objects.

The naive approach would be, for objects O_1 and O_2 with lists of clauses cl_1 and cl_2, to compile the concatenated list $cl_1; cl_2$ to arrive at the proper database function, in case O_1 inherits O_2.

However, composition $\circ : D \times D \to D$ is easily defined by

$$d_1 \circ d_2 = \lambda tcns.d_1(t, c, n, s) \cdot d_2(t, c, n, s)$$

which allows to compile the clauses of objects separately. We observe that composition is associative and $(D, \circ, d_0)$ a monoid, satisfying $d_0 \circ d = d$ and $d \circ d_0 = d$.

Alternatively, we may define a variant of composition where the functionality of one component may be overwritten as in

$$d_1 \hat{\circ} d_2 = \lambda tcns.d_1(t, c, n, s) = \varepsilon \to d_2(t, c, n, s), d_1(t, c, n, s)$$

With regard to the declarative semantics of DLP we have chosen to employ the first variant.

Chapter 11

The implementation of the prototype

- 'Then maybe you will understand this', she said, leading me over to the deep-freeze, and opening it. Inside were nothing but cats; stacks of frozen, perfectly preserved cats, dozens of them. -
Truman Capote, Music for chameleons

Having a global impression of the computation model underlying DLP, and an understanding of the interpreter supporting the Prolog like base language, we are prepared to inspect the prototype in more detail. In describing the prototype we follow the listing of the code contained in the appendix, of which an outline is given in section 9.5. Some aspects, such as the representation of terms, the interpreter and the way cuts are dealt with, are relevant only to the current implementation. Other aspects, such as the implementation of objects and processes, and our treatment of inheritance are of a more general concern, since they will be handled likewise in alternative implementations of DLP.

11.1 Types and abbreviations

As basic types we use strings, booleans and integers, which are abbreviated by respectively A, B and N. Logical variables are represented by integers, nevertheless we use type V to indicate explicitly that variables are expected as arguments. Several other types of entities play a role in the DLP system. We list them, together with the basic types just mentioned.

- A for strings,

- B for booleans,
- N for integers,
- V for logical variables ($= N$),
- O for objects (in the sense of DLP),
- Q for evaluation processes , and
- X for non-logical variables and channels.

In describing the interpreter we will use the type abbreviations

- $E = V \rightarrow V$ for environments or renaming functions,
- $S = V \rightarrow T$ for substitutions,
- $C = N \times N \times S \rightarrow T$ for continuations,
- $D = Q \times T \times C \times N \times N \times S \rightarrow T$ for database functions, and
- $K = D \rightarrow D$ for (clause) continuations,

where a function type of the form $N \times N \times S \rightarrow T$ must be read as $fn\ (N, N, S) : T$. The types introduced here are similar to the types employed in the previous chapter, when characterizing the interpreter. There are some differences, however. First of all, we take a sequence of terms to be a (compound) term itself; and so we have replaced the result type R by T. Secondly, we have introduced an additional integer parameter in the types C and D, to be able to handle cuts. See section 11.3. And in the third place, database functions of type D are enriched with a parameter of type Q in order to provide a pointer to the process accompanying an inference, as announced in section 10.4.

11.2 Terms

The language DLP is term-oriented. Terms may represent objects, processes and channels as well as goals and clauses. For this reason the term type T must include constants (that is strings) of type A, logical variables of type V ($= N$), function terms of type $A \times T \times T$, objects of type O, processes of type Q and non-logical variables and channels of type X. From entities of these types a term can be created by using one of the following constructors:

- $Con : A \rightarrow T$ for creating constants,
- $Var : V \rightarrow T$ for creating logical variables,
- $Fun : A \times T \times T \rightarrow T$ for creating function terms,
- $Obj : O \rightarrow T$ for converting an object into a term,
- $Prc : Q \rightarrow T$ for converting a process into a term, and

- $Chv : X \rightarrow T$ for converting a non-logical variable or channel into a term.

As an inverse to these constructors there are the selector functions

conof, varof, funof, objof, prcof, chvof

of obvious types.

With each term is associated a tag that indicates the type of a term. A tag is one of the following constants

CON, VAR, FUN, OBJ, PRC, CHV

that for arbitrary term t is obtained by calling *tagof*(t). These tags allow to inspect the type of a term in a case analysis on the structure of the term. For this purpose also the tests

iscon, isvar, isfun, isobj, isprc, ischv

all of type $T \rightarrow B$, can be used.

Compound terms A function term is represented as a function-symbol with two argument terms. For function terms with only one argument we make the second argument *nil*, which is an expression denoting undefined in our implementation language. This representation allows to write many algorithms with a direct recursion on the subterms of a term, but necessitates to code the argument list of a term as a term itself whenever the arity exceeds two.

The following functions are used for dealing with function terms. Let t be a term with base type $A \times T \times T$.

$fc(t)$ delivers the function symbol of t,

$a_1(t)$ delivers the first argument of t, and

$a_2(t)$ delivers the remaining arguments of t.

When the arity of a function term exceeds two, the second subterm will be a function term of the form (t_1, t_2). We call function terms with a comma as a function symbol *compound* terms. The test *isc*(t) decides whether a term is a compound term, the constructor *mkc*(t_1, t_2) makes a compound term of two terms. Function terms (and hence compound terms) are left justified. The first argument of such a term will be non-compound, while the second may be compound. This is reflected in the definition of the function *length*, given by

$$length(t) = \ \textit{if}\ isc(t)\ \textit{then}\ 1 + length(a_2(t))\ \textit{else}\ 1$$

which delivers the number of components of a term. The definition

$$argsof(t) = \ \textit{if}\ a_2(t) \equiv nil\ \textit{then}\ a_1(t)\ \textit{else}\ mkc(a_1(t), a_2(t))$$

allows then to define the arity of a term by

$$arity(t) = length(argsof(t)).$$

We will treat some auxiliary functions to manipulate terms in section 11.8. One of these functions is the function *maxvar* that gives us the maximum variable number occurring in a term.

Representing clauses Terms may be used to represent clauses. A clause of the form

```
p(X) :- q(X,Y), p(Y).
```

is, in our format, converted to a term by the expression

```
Fun(" :- ",Fun("p",Var(1),Nil), mkc(Fun("q",Var(1),Var(2)), Fun("p",Var(2),Nil)))
```

where *Nil* is the empty term, that is taken as an abbreviation for *Con*("[]"). We may access any part of the term by using the appropriate selectors. For example, the predicate name of the first atom of the body of the clause is gotten hold of by

$$fc(a_1(a_2(t)))$$

where t is the term representing the clause. We represent lists of clauses as clauses connected by a comma. When t_1 and t_2 represent clauses, the term $mkc(t_1, t_2)$ represents the list containing these clauses. In such a case we may use the test $isc(t)$ to determine whether t represents a list of clauses.

In the sequel we will use a function *append* that is like *mkc* except that it ignores empty terms. For searching whether a term occurs in a list of terms, we use a function *member*. The test $isempty(t)$ is used to decide whether t is the empty term.

11.3 The sequential Prolog interpreter

In chapter 10 we have outlined the structure of our interpreter by giving a continuation semantics from which we derived the functions *eval*, *compile* and *unify* that constitute our interpreter. We will repeat the description of these functions here, but with the inclusion of the code implementing the interaction with the accompanying inference process, needed for tracing, handling cuts, and the communication with other processes.

Tracing has been implemented according to the box-model described in [Clocksin and Mellish, 1981]. We notify the user of both the *call* entry for a goal atom, that is when the evaluation of the atom is started, and of the *exit* entry, when the evaluation resulted in a solution. We have omitted the *redo* entry, but indicate successful backtracking by reporting *exit* each time a solution is found. In addition we report *fail* in case of failure. To enable to distinguish between traces for separate evaluation processes, each entry is preceded by an identification of the evaluation process.

Cuts are dealt with by the process accompanying an inference. The idea is that on encountering a cut in *eval*, a counter that indicates the depth of the derivation with respect to the nesting of cuts is increased by one. The system database function (described in the section on the initial database) notifies the evaluation process of the depth at which the cut takes effect. The database function, delivered by *compile*, takes care that before a clause is tried a check is made whether a cut has taken effect, and if so results in failure.

Renamings and substitutions (C.f. section 10)

For renaming variables we use environments e of type E, that are typically defined as:

```
e := fn(v:V):V { result v + n }
```

where n is the number of variables already in use. Recall that V equals N. To effect a renaming we use a function *mape* and write $mape(e, t)$ to apply the environment e to the term t. See section 11.8.2 for the code of *mape*.

Substitutions, binding a variable v to a term t, may be created by statements of the form

```
s1 := fn(v1:V):T { result v = v1 → t, s(v1) }
```

that defines a substitution $s1$ that delivers t as a value if the actual parameter $v1$ equals the variable v and $s(v1)$ otherwise, for s the given substitution. We write $maps(s, t)$ to apply a substitution to a term. See also section 11.8.2 for the definition of *maps*.

11.3.1 Evaluation

In defining the *eval* function below, we assume that goals t are either empty, or consist of an atom a and a remainder g.

Empty goals result in the continuation. In case we have a compound goal, it is decomposed in an atom a and a remainder g. We apply the given substitution s to the atom a and create a continuation for g. Before evaluating a by the database function stored in the process q, we simplify a as described in section 11.7.1. Note that we have apart from q an additional parameter m, that we use for dealing with cuts.

eval

```
eval(q:Q,t:T,c:C,m,n:N,s:S):T
temp a,g:T
{
if isempty(t) then c(m,n,s)
else a := maps(s,a1(t)); g := a2(t);
    dotail := fn (m1,n1:N, s1:S)
                {
                q!trace("exit",maps(s1,a));
                if iscut(a) then mct := m1 + 1 else mct := m1;
                result eval(q,g,c,mct,n1,s1)
                };
    q!trace("call",a);
    result q@d(q,simplify(q,a),dotail,m,n,s)
}
```

When creating a continuation, a check is made whether the goal atom a is a cut by the test *iscut*(a). If this is the case then the depth-counter is increased by one, relative to the depth with which the continuation *dotail* is provided as an argument. When reporting *exit* of the evaluation of a, we apply the then current substitution $s1$, to show the binding that resulted from evaluating a.

11.3.2 Compilation

The compilation procedure given below differs from the one given in section 10.2 by the incorporation of statements for handling cuts. We refer to that section for the code of *compile*, which remains unaffected. Here we will treat the definition of the function *newd* that corresponds to compiling a single clause.

When compiling a clause of the form $a :- g$, assuming that we have already constructed a database d for the clauses preceding that clause, we insert a test for checking whether a cut is active:

```
r := d(q,t,c,m,n,s);
if checkcut(q,t,m) then result r
else ...
```

We store the result of evaluating a goal t by the database d in r. If a cut is active we return r as the result; otherwise we proceed by unifying the goal with the head of the clause and creating a continuation for evaluating its body. See section 11.5.3 for the definition of *checkcut*. The creation of *newd* is pictured below. We have used the notation $'a :- g'$ to represent the clause that is compiled; and write a for $a_1('a :- g')$ and g for $a_2('a :- g')$.

newd

```
newd := fn ( q:Q, t:T, c:C, m,n:N, s:S):T
            temp m1, n1:N
            {
            r := d(q,t,c,m,n,s);
            if checkcut(q,t,m) then result r
            else
                m1 := m + 1; n1 := n + maxvar('a :- g');
                e := fn (v:V):V { result v + n };
                body := fn (m2,n2:N, s2:S):T
                        {
                        result eval(q, mape(e,g), c, m2, n2, s2)
                        };
                result append( r , unify(t, mapcut(m,fc(a)), mape(e,a)),
                                            body,m1,n1,s))
                }
```

When creating the continuation for the body of the clause the cuts occurring therein are replaced by a special predicate $cuton(m, p)$, where m is the depth at which the clause is invocated and p the predicate name of the head of the clause. This is effected by the function *mapcut* that is defined as

mapcut

```
fn mapcut(m:N,p:A,t:T):T
{
if isc(t) then result mkc( mapcut(m,p,a1(t)), mapcut(m,p,a2(t)))
elsif iscon(t) & conof(t) = "!" then result Cuton(m,p)
else result t
}
```

where $Cuton(m, p)$ makes a predicate $cuton(m, p)$ that stores, when evaluated, the predicate name p and the depth m in the evaluation process q. See section 11.7.3. This information is used by the process q when $checkcut(q, t, m')$ is called to determine whether a cut is active.

11.3.3 Unification

Unification may result in modifying the given substitution, by binding variables to terms. The definition of the function updating a substitution s with respect to a variable v and a term t, as given below, refers to a continuation c and integers m and n, that come from the environment in which the function definition must be placed.

update

```
update := fn (v:V, t:T):T
          {
          news := fn(v1:V):T { result v = v1 → t , s(v1) };
          result unbound(v,s) → c(m,n,news), unify( s(v), t, c,m,n,s)
          }
```

In case the test $unbound(v, s)$ succeeds, the newly created substitution is handed as a parameter to the given continuation; otherwise the term t is unified with the substitution value of v. This definition is used in the function *unify* as depicted below.

unify

```
fn unify(t1,t2:T,c:C,m,n:N,s:S):T
{
update;
if isvar(t1) then result update(varof(t1), t2)
elsif isvar(t2) then result update(varof(t2), t1)
elsif isfun(t1) & isfun(t2) then
          if fc(t1) = fc(t2) then
                    doargs := fn (m1,n1:N, s1:S):T
                              {
                              result unify(a2(t1), a2(t2), c, m1, n1, s1)
                              };
                    result unify(a1(t1), a1(t2), doargs, m, n, s)
                    else result Nil
else result t1 = t2 → c(m,n,s) , Nil
}
```

For function terms, having identical function symbols, the first arguments are unified with as a continuation the unification of their remaining arguments.

11.3.4 Initialization

To enable an evaluation process to communicate the results of an inference to another process, we must take care to store the initial goal in the evaluation process and to report the instantiation of the goal (by the current substitution) each time a solution is found. This is effected by creating an initial continuation in the following manner.

yes

```
fn yes(q:Q,t:T):C
{
c := fn(m,n:N, s:S):T { q@result := maps(s,t); result maps(s,t) };
q@goal := t;
result c
}
```

Before returning the initial continuation c, that reports the instantiation of t with respect to the then current substitution s, the process q is called to store t in its *goal* attribute. C.f. sections 9.2 and 11.5. We may then start up an inference for a goal t by calling

$$eval(q,t,yes(q,t),0,maxvar(t),starts)$$

where *starts* is the initial substitution given by

$$fn\ starts(v{:}V){:}T\ \{\ result\ Var(v)\ \}$$

and $maxvar(t)$ represents the maximum variable number occurring in t.

For evaluating a goal t we may also spawn off an inference process by calling

$$infer(q,t,yes(q,t),0,maxvar(t),starts)$$

The function *infer* is similar to the function *eval*, except that a POOL* process is created to do the evaluation. The communication of the result is mediated by the accompanying process q.

11.3.5 Composition

As described in section 10.5, we deal with inheritance by composing database functions. The function *compose*, defined as

compose

```
fn compose(d1,d2:D):D
{
d := fn (q:Q,t:T,c:C,m,n:N,s:S):T
        {
        result append(d1(q,t,c,m,n,s), d2(q,t,c,m,n,s));
        }
result d
}
```

merely appends the results of the composed databases.

To be able to report *fail* on failure, we put the combined database of each DLP object in composition with the database *fail* defined as

fail

```
fn fail(q:Q,t:T,c:C,m,n:N,s:S):T { q!trace("fail",t); result Nil }
```

Apart from notifying q of failure, this database adds nothing to the result.

11.4 Objects

Objects, in the sense of DLP, are implemented by instances of the POOL* class *O*. In section 9.1 we have sketched the attributes and behavior of active objects. We will now also pay attention to the behavior of passive objects, that do not have own activity but instead allow multiple method calls to be active simultaneously.

We distinguish between four groups of attributes. The first three attributes are needed to identify (an instance of) an object and to keep account of the number of evaluation processes created for it.

- *name* - the name of the object,
- *instance* - the instance number of the object,
- *process number* - records the number of evaluation processes created for the object.

When a copy is made the instance number is raised by one and its process number is set to zero.

The second group of attributes stores the functionality of the object, the clauses and its inheritance relations.

- *use list* - the names of the object from which clauses are inherited,
- *isa list* - the names of the objects from which non-logical variables are inherited,

- *nlvars* - the list of non-logical variables,
- *clauses* - the clauses declared for the object,
- *d* - the database function compiled from the clauses.

Each instance receives a copy of these attributes. Alternatively, we could have stored this information in a data structure shared by all instances of an object.

The next two attributes are needed to distinguish between active and passive objects.

- *active* - that determines whether mutual exclusion between method calls is necessary, and
- *locked* - a boolean, that indicates whether the object is locked, i.e. unable to receive any requests for evaluating a method call.

The last group determines the dynamic behavior of active objects with respect to the acceptance of method calls:

- *accept list* - that contains the current acceptance conditions,
- *accept queue* - containing the requests that are waiting for a rendez-vous, and
- *process stack* - containing the processes for which an accept statement is evaluated,

11.4.1 The protocol

Passive objects display full internal concurrency, in that multiple method calls may be evaluated simultaneously. Active objects, on the other hand, refuse to grant a request as long as there are method calls waiting to be evaluated.

If the object is *locked* then only the methods safe with respect to the acceptance of method calls may be answered. These methods are collected in the macro *SAFE*. Compare section A.4.1. Otherwise any call may be answered, including the methods *accept* and *request* for respectively evaluating an accept statement and a (DLP) method call.

The body of *O* implements this behavior.

protocol

```
body do if locked then answer(SAFE) else answer(any) fi od ydob
```

11.4.2 Acceptance

When a process evaluating a goal encounters an accept statement, it calls the method *accept*, that we may characterize as below. Note that the method *accept* refers to the instance variables of O, written in italic. We have omitted the result parameter, not because the method is asynchronous, but because we are not interested in the result. Actually, the result of *accept* is used to encode the information needed for the process in which the accept statement occurred to continue its computation. We refer to the appendix, section A.4.2, for a more detailed description.

accept

```
method accept(q:Q, t:T)
{
suspend the process q and put it in the process stack
set the accept list to t,
and check the accept queue to see if it contains an
acceptable call, otherwise unlock
}
```

The accept queue contains pairs of the form (q, t), for q a process and t the call to be evaluated. Whether a call is acceptable depends on the accept expressions contained in the accept list. Simple accept expressions require merely to check the predicate name of the call.

For accept expressions of the form

$$method : guard \rightarrow goal$$

and a call mc, we create a test goal t by

$$t := mkc(Fun(" = ", mc, method), guard);$$

and, having created an evaluation process q referring to the object for which the test is made, we evaluate the test by stating

$$infer(q,t,yes(q,t),0,maxvar(t),starts)$$

We then ask for the resumption r by

$$r := q!resume();$$

and if r is not *fail* we modify *goal* by

$$goal := mkc(r,goal);$$

in order to effect the bindings that result from the evaluation of t before evaluating *goal*. The modified *goal* is then evaluated as an ordinary (accepted) method call. The bindings contained in the resumption r must further be communicated to the process for which the accept statement was evaluated.

11.4.3 Method calls

How a method call is treated depends on whether an object is active or not. For passive objects, a method call t results in immediately spawning off an inference processes by calling

$$infer(q,t,yes(q,t),0,maxvar(t),starts)$$

For an active object, it must first be determined whether the call is acceptable, as depicted in the definition of the method *request*. The result of calling *request* is an evaluation process created to accompany the evaluation of the goal.

request

```
method request(t:T):Q
{
q := "a new evaluation process for the object";
if ¬ active then
            infer(q,t,yes(q,t),0,maxvar(t),starts);
elsif acceptable(t) then
            lock();
            infer(q,t,yesQ(q,t),0,maxvar(t),starts);
else
            put (q,t) in the accept queue and suspend q
fi;
result q
}
```

If the goal is acceptable then the object becomes *locked* and an inference process will be created, otherwise the pair (q, t), containing the goal and the accompanying evaluation process, is put in the accept queue to await further treatment. In that case the process q is suspended.

11.4.4 Inheritance and compilation

An object may inherit non-logical variables and clauses from other objects. Multiple inheritance is allowed. Objects inherited by an inherited object are inherited. We have a simple algorithm to deal with possible cycles in the inheritance graph, illustrated in the definition below.

inherit

```
method inherit(x:T,had:T)
{
if isc(x) then inherit(a1(x), had); inherit(a2(x), had)
elsif ¬ member(x,had) then
      copy the non-logical variables of x and add x to had
fi
}
```

If x is compound, a recursive call is made to inherit from the components of x. Otherwise, copies of the non-logical variables of x are added to the list of non-logical variables of the object, and x is added to the *had* list.

For compiling the clauses of an object the function *compile* defined in section 11.3.2 is used. The compiled databases of the inherited objects are then put in composition with the own database of the object, following a similar procedure as outlined for the inheritance of non-logical variables. The combined database is stored in the attribute d, and given as a parameter to the evaluation processes created for the object.

11.5 Processes

A DLP program consists of a number of object declarations. To execute a program, the system needs to keep a record of the declared objects (in the sense of DLP) in order to create new instances.

A global *supervisor* is created to handle input from the user, and to create new instances of objects.

Each evaluation of a goal is accompanied by a so-called evaluation process. Its major tasks are to keep track of when a cut becomes active, to report the trace information to the user and to engage in communication with another process in order to deliver the answer substitutions resulting from the evaluation of a goal.

11.5.1 Global information

The global POOL* object *knot* maintains the

- *object list* - a list in which the declared objects (in the sense of DLP) are kept.

When reading in a program, each time an object declaration is encountered a new entry is created in the *object list*. Instances of objects do not give rise to a new entry.

11.5.2 Object and command management

User input is directed to the global POOL* object *supervisor*, that deals with the declaration of new objects.

For each top level goal that is evaluated, a copy of the supervisor is created, to handle the creation of instances of objects, using the information contained in the global *knot*.

11.5.3 Evaluation processes

Each inference process is accompanied by an evaluation process, referring to the object for which the goal is being evaluated. Evaluation processes are used for reporting trace information, for dealing with options, for keeping track of cuts, and more importantly for communicating the resumptions encoding the answer substitutions resulting from the evaluation of a goal to the invoking process.

Evaluation processes are implemented as instances of the POOL* class Q. Among the attributes of class Q we have:

- *name* - the name of the process to be used in tracing,
- *obj* - the object to which the process refers, and
- d - the database that is initially given to the process.

The name of the process is derived from the name, instance number and process number of the object to which the process refers. The object pointer is given to be able to access the non-logical variables of the object. The database stored in d is initially the database of the object. However, when clauses are dynamically asserted or retracted, the changes affect only the database stored in d, and not the database of the object.

Cuts are handled by the process accompanying an inference. We maintain a dynamic record of the positions in the derivation at which a cut occurs and check when evaluating a particular goal atom whether a cut is active, that is whether the derivation must be cut off.
For doing this we need the attributes

- *cut* - that stores the current cut level or cut.*depth*,
- *cms* - that stores the current (*depth*, *predicate*) pair, and
- *cutlist* - that is a stack of previous (*cut*, *cms*) values.

The cut level stored in *cut* is a number indicating the depth at which a cut is active. When a goal is evaluated at a depth below *cut* then necessarily this means that backtracking has occurred. In the Prolog context, encountering a cut means that on backtracking failure occurs, and moreover that no other clauses will be tried for the predicate that is the head of the clause in the body of which the cut occurs. For this reason, the *cms* attribute records the predicate name of the head of the clause that contains the cut as well as the depth at which the clause is tried. Whenever the depth of the evaluation becomes lower than the depth stored in the

cms attribute, backtracking may occur again, in this case the *cutlist* is popped to restore *cut* and *cms* to the values previously held.

The function *checkcut*, that is called in the database function produced by compiling a clause, summarizes this procedure. C.f. section 11.3.2.

checkcut

```
fn checkcut(q:Q,t:T,m:N):B
{
k := q@cut;
(k0,p) := q@cms;
result m > k → false,
       m ≤ k & m ≠ k, → true,
       m = k0 & fof(t) = p → true,
       checkcut(q!cutit(), t, m)
       }
```

When the last alternative is tried, that is when the actual depth of evaluation is lower than the depth of the current *cms* attribute the method *cutit*() is called, which amounts to popping the *cutlist* and resetting *cut* and *cms* to their appropriate value, as expressed by the definition

cutit

```
method cutit() { cut := m'; cms := (m,p); cutl := rest; }
```

where we assume that *cutlist* is of the form $(m', (m, p)) \cdot rest$.

Upon encountering a cut, that is when the system predicate $cuton(m, p)$, as introduced in the compilation, is evaluated by the system database, the method *cuton* is called, as in $q!cuton(m', m, p)$, to update the values of *cut*, *cms* and *cutlist*. The argument m' represents the depth at which the goal $cuton(m, p)$ is actually encountered, which is not necessarily equal to m.
The method *cuton* is defined as

cuton

```
method cuton(m',m:N,p:A)
{
cutlist := (cut, cms) · cutlist; cut := m'; cms := (m,p);
}
```

thus modifying the appropriate attributes in a straightforward way.

Returning answer substitutions As we have explained in chapter 9.2, answer substitutions requested by the process that invoked the method call are returned in the form of a resumption that must be executed by that process to effect the answer substitutions.

We use an attribute

- *state* - that may take the values BUSY, PEND, WAIT and STOP

to determine whether the process is willing to answer a request for a resumption, and if so what resumption will be returned.

The attribute

- *goal* - initialized when creating the initial continuation

is used to store the initial goal, and the attributes

- *result* - to store the most recent answer, and
- *solutions* - to store all answers that have been produced,

contain instantiations of (the variables of) the initial goal.

To be able to suspend a process a boolean *locked* is used, that may be changed by calling the methods *lock* and *unlock*. A process is locked, either when an accept statement is evaluated, or when it must await the acceptance of a method call.

The protocol governing the communication of resumptions is contained in the body of Q. Below we assume that *ME* represents all methods except *resume* and *unlock*.

protocol

```
body do
      if locked then answer(unlock)
      elsif state = BUSY then answer(ME)
      else answer(resume)
      fi
od ydob
```

The request for a resumption may obviously only be answered if the process is not BUSY evaluating a goal, that is when it has produced an answer, or cannot produce any further answers.

Dependent on the state, either a failing, a unifying or a backtracking resumption is created, as explained in section 9.2. The code for the method *resume* now looks as follows.

resume

```
method resume():T
{
if state = WAIT & isempty(result) then result 'fail'
elsif state = PEND then result 'goal = result'
else result 'member(goal,solutions)'
}
```

The quoted terms must be read as Prolog terms, constructed as explained in section 11.2.

11.6 Non-logical variables and channels

When executing a DLP program, two kinds of special objects may come into existence, non-logical variables and channels. Non-logical variables are typically created when an instance of a DLP object is created. Channels are created dynamically by asking for a new channel.

11.6.1 Non-logical variables

Non-logical variables have as attributes a *name* and a *value*. This value can be initialized when creating a non-logical variable. There are no restrictions on accessing the value of a non-logical variable other than those imposed by the behavior of the object to which it belongs.

11.6.2 Channels

Channels mediate the unification of terms that belong to different processes. Such a transfer takes place when both an input side and an output side are present, provided that the terms put on either side on the channel are unifiable. Communication over channels allows backtracking on the input side when the input term and output term do not unify.

The attributes of a channel are

- *free* - indicates whether an output term is present,
- *confirmed* - indicates whether new input terms are allowed,
- *store* - stores the most recent input term,
- *message* - stores the most recent output term.

These attributes, together with the protocol described below, enforce the desired behavior of the channel.

protocol

```
body do
        if ¬ confirmed & ¬ free then answer(read, confirm, force)
        elsif ¬ free & confirmed then answer(receive)
        else answer(any)
od ydob
```

Initially the channel is free, in that no output term is present. While the channel has not received an output term it may accept anything.

When it first receives an input term, by a call to *read*, it waits until the attribute *message* is given a value, by a call to *write*.

read

```
method read(t:T):T
{
store := t;
while isempty(message) do answer(write) od;
result message
}
```

When another call to *write* is still active, the channel must wait for a call to *read*.

write

```
method write(t:T)
{
while ¬ isempty(message) do answer(read) od;
message := t; free := false; confirmed := false;
}
```

When the message is unifiable with the input term the process evaluating the input statement calls for the method *confirm* that sets the attribute *confirmed* to true.

confirm

```
method confirm() { confirmed := true; }
```

The output side, by calling the method *receive*, then merely has to collect the input term as stored in the attribute *store*, to unify this with its message. C.f. section 11.7.3.

receive

```
method receive():T
{
free := true; confirmed := false;
message := nil; result store;
}
```

When the message is not unifiable with the input term then another input term must be provided. Since the attribute *message* already has a value the unification of the input term and output term is then tried immediately.

When the channel first receives an output term it must wait for a call to *read*, or a call to the method *force*, that has a similar effect as the method *confirm*, but for the fact that the value delivered results in a failure on the output side.

11.7 The initial database

For the evaluation of the special forms by which DLP extends Prolog, simplification is applied in order to replace symbolic terms referring to objects, processes, non-logical variables or channels by a pointer or a value. Simplification is also used to provide a special interpretation to equality and assignment. Apart from the system database, that contains primitives for dealing with cuts and communication, we will discuss also the DLP object *boot*, in which a number of standard Prolog connectives are defined.

11.7.1 Simplification

Simplification is performed by the function

fn simplify(q:Q,t:T):T

When a term is simplified, non-logical variables are replaced by their values, except when the non-logical variable occurs on the left hand side of assignment. In that case the name of the variable is replaced by a pointer to allow subsequent assignment of the value on the right hand side.

Simplification is also applied to terms with the function symbol *new*. For instance, a term of the form $new(c(t))$ is replaced by an active object, as shown in the code below.

new$(c(t))$

```
o := knot!obj('c')!cpy();
o@acceptlist := 'c'; o@active := true;
o!request('c(t)');
result o;
```

Somewhat informally, we use $'c'$ and $'c(t)'$ to represent the object name c and the constructor term $c(t)$. After creating a copy of the object c, as known to the global *knot*, the accept list of the newly created object o is set to c and the object is made active. Then o is requested to evaluate the constructor. Finally, the object o is returned to replace the term $new(c(t))$.

As another simplification, we wish to mention the replacement of arithmetic expressions like $'1 + 2'$ by their (standard) interpretation, $'3'$ in this case.

11.7.2 Equality and assignment

The interpretation of equality and assignment takes place by the function

fn assign(q:Q, t:T, c:C, m,n:N, s:S):T

Having a goal of the form $O = new(c(t))$, the term $new(c(t))$ is replaced by an object, as described above, and the logical variable O is bound to this object.

Goals of the form $Q :: O!m(t)$ are evaluated as depicted in the code below.

$Q::O!m(t)$

```
q := objof('O')!request('m(t)');
result unify('Q',Prc(q),c,m,n,s);
```

The notation $'O'$ and $'Q'$ is used to represent the terms containing the variables O and Q. We request the object to which O is bound to evaluate the goal $'m(t)'$ and unify the variable Q with the term $Prc(q)$, where q is the result of the request, with the continuation parameters c, m, n and s as originally given.

11.7.3 The system database

The evaluation of a number of system primitives is handled by the database function

fn system(*q:Q, t:T, c:C, m,n:N, s:S*):*T*

When a goal of the form $cuton(m', p)$, for a natural m and a predicate name p, is encountered the method *cuton* is called for the process q, given as a parameter to the system database, as in $q!cuton(m, m', p)$ for m the current cut level.

Accept statements of the form $accept(t_1, ..., t_n)$ are handed over to the object to which q refers. In case we have an accept statement containing conditional accept expressions, the result of processing the accept statement may be a non-empty goal that must be evaluated in order to effect the bindings resulting from the evaluation of the guard.

A synchronous call of the form $'O!m(t)'$ is rewritten into the sequence $'Q::O!m(t)'$, for a fresh variable Q, and evaluated by calling *assign*. In the code below it is shown how to create a fresh variable. C.f. section 11.2.

O!m(t)

```
g := mkc(Fun("::",Var(n + 1),t), Fun("?",Var(n + 1),Nil));
result assign(q,g,c,m,n + 1,s);
```

The evaluation of g by *assign* results in binding the variable $Var(n+1)$ to the process evaluating $m(t)$ to be able to ask for the resulting resumptions.

If the goal atom is of the form $Q?$ then a resumption is asked for and evaluated, as shown below.

Q?

```
r := prcof('Q')!resume();
if prcof('Q')@state ≠ BUSY then r := Fun(";",r,t) fi;
result eval(q,r,c,m,n,s);
```

The test if the process Q is *BUSY* is to determine whether backtracking must occur to ask for any alternative resumptions.

Goals of the form $C!t$ and $C?t$, with C bound to a channel are interpreted as respectively an output statement and an input statement.

An output statement is evaluated as follows, for C the channel and t the output term:

C!t

```
chvof('C')!write(t);
r := chvof('C')!receive();
result unify(t,r,c,m,n,s);
```

After the term t is written on the channel, an input term must be waited for. See 11.6.2.

An input statement, for channel C and input term t, is dealt with as depicted below:

C?t

```
r := chvof('C')!read(t);
g := mkc(Fun(" = ",t,r), Fun("confirm",'C',nil));
result eval(q,g,c,m,n,s);
```

Evaluating the goal '$t = r$, *confirm*(C)' amounts to unifying the input term t with the output term r, and to evaluate *confirm*(C) if the unification succeeds. In case the unification does not succeed, backtracking takes place over the alternatives

contained in the continuation c. The evaluation of *confirm*(C) results in calling the method *confirm*() for the channel C.

11.7.4 Booting

In the object *boot* we have defined, among others, the predicate *true*, negation by failure, the disjunctive connective of Prolog and the Prolog conditional. Also, we have defined the and-parallel connective of DLP treated in section 2.6, and the predicates *member* and *append*.

boot

```
object boot {

true.

not(A) :- A,!,fail.
not(A).

A;_ :- A.
_;B :- B.

A → B :- A, !, B.

A&B :- Q = self!A, B, Q?.

member(X,[ X | _ ]).
member(X,[ _ | T ]) :- member(X,T).

append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).
}
```

The object *boot* is by default inherited by all DLP objects.

11.8 Utilities

We have collected a variety of abbreviations and function definitions that support the definitions given in the previous sections in a separate unit.

11.8.1 Auxiliary definitions

This part defines a number of system parameters, such as the maximum number of objects that may be in existence, the amount of debugging and tracing information that must be given, and the symbols that are recognized as keywords of DLP.

11.8.2 Term manipulation functions

The representation of terms used enables to formulate the functions for manipulating terms by a direct recursion on the subterms. We will give three examples. The first example is the function *maxvar*, determining the maximum variable number occurring in a term.

maxvar

```
{
fn maxvar(t:T):N
{
if isvar(t) then result varof(t)
elsif isfun(t) then result max(maxvar(a1(t)),maxvar(a2(t)))
else result 0
}
```

The second is the function for applying a renaming function e of type E to a term.

mape

```
fn mape(e:E,t:T):T
{
case tagof(t)
VAR then result Var(e(varof(t))
FUN then result Fun( fc(t), mape(e, a1(t)), mape(e, a2(t)) )
otherwise result t
esac
}
```

And the last example, following a similar pattern, is the function for applying a substitution.

maps

```
fn maps(s:S,t:T):T
{
case tagof(t)
VAR then result isbot(s(varof(t))) → t, maps(s,s(varof(t)))
FUN then result Fun( fc(t), maps(s, a1(t)), maps(s, a2(t)) )
otherwise result t
esac
}
```

In all these cases, dealing with a function term gives rise to creating a function term with arguments appropriately modified.

Chapter 12

Conclusions and future research

- If someone is looking for something and perhaps roots around in a certain place, he shows that he believes that what he is looking for is there. -
Ludwig Wittgenstein, On certainty

Now that we have completed the description of the various aspects our language, we like to spend some time evaluating our efforts, and the route taken to arrive at our results. Also, we wish to indicate some lines of future research.

The research reported in this thesis covers the design, semantics and implementation of DLP, a language for distributed logic programming:

Design is a delicate issue. The, in our view, most distinguishing feature of our language DLP is that it offers a kind of *backtrackable rendez-vous*. Combining logic programming, object oriented features and parallelism, it allows to employ message based computation without sacrificing the opportunities of search by backtracking. Characterizing our language, we may say that DLP extends Prolog with features for parallel object oriented programming. The difference with other parallel object oriented extensions of logic programming languages is precisely that we did not wish to sacrifice the backtracking offered by Prolog. We have spent considerable effort to arrive at a notion of objects compatible with the demands of search based programming. We have defended our decisions with respect to the way (distributed) backtracking takes place, the mutual exclusion provided between method calls, and the protection of non-logical variables in chapter 5.

From the point of view of a declarative reading of DLP programs, non-logical variables, by which objects may be assigned a persistent state, form an impediment

to formulating a natural declarative semantics. The *conditional accept* statement, introduced in section 2.5, however, obviates the need for non-logical variables. We may remark that this feature has been developed relatively late, when studying the semantics of the language was in full progress. We think that it deserves a thorough semantic study. Such a study may well influence the design of the language. Nonetheless, eliminating non-logical variables altogether might make inheritance an infeasible feature, since basing inheritance on behavioral notions is far more difficult than basing it on code sharing.

Semantics may guide the design of a language. The formal semantics elaborated in part II encourages to believe that we have developed a sound language. Apart from being a rather entertaining enterprise, developing a formal semantic description is a good way of checking that the features included in the language are sensible, that is well-defined and non-trivial. For the designer of a language, to develop a formal semantics reduces the probability of errors. We disagree with the opinion that semantic amenability is the only criterion for the usefulness of a language construct. In our case, working on a semantic description helped in demarcating the core of our language. In proving our operational and denotational semantics equivalent we have restricted ourselves to the most important features, leaving out those that did not seem worth the trouble. Encouraging indeed was the fact that the operational description of the interplay between backtracking and message passing turned out to be rather concise and elegant, much more than we expected. Finding a suitable denotational, that is mathematical, compositional, characterization equivalent to our operational description strengthened our conviction to have hit on a solid (language) construct.

Implementing a prototype proved to be very fruitful in designing the language. Having a (modifiable) prototype available allowed us to experiment with different language features and to test our intuition by actual examples. Shortcomings could easily be corrected, giving full chance to a healthy exchange between practical experience and formal scrutiny.

The prototype, described in the previous chapters and the appendix, is written in POOL* , a variant of POOL-X, which is in its turn compiled to C by using the compiler described in [Beemster, 1990]. In developing the prototype we have taken benefit from the constructs for process creation and communication available in our implementation language. From the point of view of language design one could even view DLP as lifting a parallel object oriented (POOL-like) language to the level of logic programming. Nevertheless, in order to implement distributed backtracking we had to write code for object and process management, since it requires to make a distinction between (DLP) objects and processes.

We have few experimental figures that illustrate the adequacy of our implementation with respect to performance and the utilization of parallelism. We did however compare the performance of our system with the performance of C-Prolog, under Unix, on a 12 MB Sparc workstation, for a sequential version of the *quicksort* program described section 2.6. We varied the length of the lists to be sorted from 10 to 70. With the list of seventy elements our system ran into memory problems and gave up. Compared with the slight increase in execution time of

Prolog, when increasing the length of the lists, the increase in time for DLP was rather dramatic. Memory usage obviously is the problem.

Further, we did try to measure the speed-up obtained for the parallel quicksort program described in section 2.6. Unfortunately, the parallel interpreter we did had at our disposal also ran into memory problems.

We were able to check the adequacy of our allocation primitives. With a DLP program that merely creates processes, distributing them evenly over the available processors, we obtained a speed-up proportional to the number of processes created. We must remark that what is actually measured by using a parallel interpreter for POOL-X, is not the amount of parallelism in DLP but the parallelism displayed by its underlying implementation. We did not pursue this line of research, since efficiency and speed-up have simple not been our primary concern. Nevertheless, in this stage we begin to consider performance as an important issue, and we will make some recommendations aimed at improving the efficiency of the system.

Although we have repeatedly stated that our language is logic based, we have not paid any attention to the declarative semantics of our language. Concluding our thesis, we present an inventory of the problems posed by our language, and some of the possible directions of research.

12.1 Improving the efficiency of DLP

A breakthrough in the execution speed of sequential Prolog has been achieved with the introduction of the Warren Abstract Machine. See [Warren, 1977] and [Warren, 1983]. Since Prolog is the base language of DLP, an obvious improvement in performance would be obtained by replacing our Prolog interpreter, as described in chapter 10, by an interpreter based on the Warren Abstract Machine.

An extended version of the Warren Abstract Machine has also been the basis of a parallel implementation of Prolog. See [Hermenegildo, 1986] and [Hermenegildo and Nasr, 1986]. Although our needs are somewhat different, since we do not aim at realizing implicit parallelism, we may well profit from the primitives for process creation and communication developed in [Hermenegildo, 1986].

Performance would also definitely be improved by using compilation techniques for implementing the access to non-logical variables.

12.2 Declarative semantics for distributed logic programming

Although we did provide a mathematical denotational semantics for DLP in part II, we did not give a declarative semantics in the sense this phrase is commonly understood. C.f. [Apt et al, 1987]. Our denotational semantics may be understood as characterizing the flow of control of our language in a mathematical way, rather than providing a characterization of the logical meaning of the language.

Model theoretically, a Horn clause program may be given a meaning by assigning it a subset of the Herbrand base, the set of ground atoms over the signature of

the language. Such an interpretation, that is a subset of the Herbrand base, is usually obtained as the fixed point of a consequence operator, delivering the atoms that logically follow from a given interpretation. The least fixed point of such an operator is commonly called the success set, containing all atoms that are valid with regard to the program characterized by the operator.

For pure Horn clauses, a completeness result is know, stating that every atom in the success set may be derived as a goal using SLD resolution. [Lloyd, 1984].

Leaving the realm of pure Horn clause programs such a completeness result becomes more difficult to obtain. For instance, mechanisms encountered in Concurrent Prolog or Parlog, such as the commit operator that realizes *don't care* non-determinism controlled by guards, and input constraints, on which the synchronization of AND-processes is based, enforce a reduction of the success set. C.f. [de Boer et al, 1989].

More drastic repairs are necessary for most of the languages that we have studied in section 5.2. As a preliminary investigation into the declarative semantics of DLP, we will study the adaptations needed to deal with features such as objects with states, contexts in which a derivation takes place, and the synchronization due to explicit communication.

States The most characteristic property of an object, from a logical point of view, is that whatever functionality an object may provide, this functionality is dependent on its state. In [Chen and Warren, 1988] it is proposed to consider objects as intensions, that is as functions from states to values. It is observed that, operationally, each object has a closed past with respect to its current state, and an open future. An update to an object will determine the values delivered in the next state. The approach taken in [Chen and Warren, 1988] is to extend the semantics of first order logic programming by the notion of intensions, representing the behavior of objects that may change in time. Since there is a temporal dependence among the values of an object in consecutive states, we may adopt a frame assumption stating that the values of an object remain the same as in the last state, unless they have explicitly been updated.

Contexts Another way to view objects is as providing a context determining the meaning of a particular predicate. In [Monteiro and Porto, 1988] this aspect of objects is treated by introducing context dependent predicate definitions. A possible worlds declarative semantics is given for a language for, what they call, contextual logic programming. Method calls may be given meaning in this framework as a change of context for evaluating the call, that is as the transition to another world. A somewhat similar approach for dealing with modular features of logic programming has been sketched in [Miller, 1986].

Synchronization Finally, the question remains how to deal with communication in a declarative way. Work in this area has been reported in [Monteiro, 1984], that provided a semantic foundation for Delta Prolog. C.f. [Pereira and Nasr, 1984]. The idea is, roughly, that the validity of a goal is made dependent on the history of communication events.

A rather different approach is contained in [Falaschi et al, 1984], where an operational, model-theoretic and fixed point semantics is given for Horn clauses extended with a synchronization operator. The extended Horn clause logic presented there resembles the languages of Logical Objects and Communicating Prolog Units, described in section 5.2.

Our explorations thus far indicate that we need a Kripke style semantics to deal with the declarative meaning of DLP programs. No doubt DLP would benefit from such a declarative semantics.

Appendix A

The prototype

Developing and maintaining a medium sized program is definitely facilitated by adopting a style of programming that has been introduced by [Knuth, 1984] as *literate programming*. Apart from making a pun at the school of structured programming [Knuth, 1984] professes the advantage of writing a program like one writes an article. As not to be trapped in yet another ideological debate we merely wish to state that we took pleasure in structuring a program as a document and mixing in explaining text.

Before we start the description of our prototype implementation we wish to introduce some additional features of the DLP system.

Extensions to DLP

The DLP system contains a number of features not described in part I, II and III. For example options and defaults have not been treated, since they are not of importance from a conceptual point of view. Nevertheless, for actually using the system such features are of some importance.

A quite different consideration is that the language DLP is still evolving. We have included the code for these extensions, since it might provide a starting point for further experiments and research.

Options may be set for an object by a unit clause of the form

- set *name* = value.

which sets the option *name* to *value*. A value may be either *on* or *off*, or a number. When no value is specified, the value given is assumed to be *on*. Each process created for each instance of the object is initialized with the options as set for the object.

Options may also be set dynamically, by a goal of the form

- option(*name*) := *value*

which sets the option *name* to *value* for the process evaluating the goal.

If we do not explicitly mention what values an option may take, the value is either *on* or *off*. Unless explicitly set by the user, the value of an option is zero or *off*. We have the following options.

- *trace* - with possible values 0, 1, 2 or 3 to determine the amount of tracing.
- *nosimplify* - to enforce that no automatic simplification of arithmetic expressions and the like takes place.
- *eager* - to effect that all solutions are computed and communicated at the first resumption request.
- *sol* - to determine the number of solutions that must be produced. When *sol* has value 0 then all solutions will be generated. When *sol* is 1 then the behavior of the object will be deterministic. Otherwise, a call will result in no more solutions than indicated by the value of *sol*.
- *override* - if set the overriding mode of composition is used in combining the databases of the inherited objects; in other words, inherited predicates can only be redefined instead of being extended.
- *copy* - to initialize the process created for evaluating a method call to the options of the invoking process, instead of to the options of the object to which the call was addressed.
- *branch* - determines the branching degree of the (imaginary) tree of processors, as used in the allocation primitives described in section 2.7. The default is 2, when *branch* is zero.
- *width* - determines the width of the (imaginary) matrix of processors. See also section 2.7. The default is 4, when *width* is zero.
- *nproc* - determines the number of processors assumed to be available. The default is 10, when *nproc* is zero.

Further, we have three options aux_1, aux_2 and aux_3, that may be used for other purposes.

To inspect the value of an option the expression option(*name*) may be used, which will be replaced by the value of the option *name* unless the expression occurs on the left hand side of an assignment.

For debugging purposes the goals *debug* and *nodebug* may be used, that respectively turn on and switch off debugging globally.

Defaults may be used for the initialization of objects with respect to inheritance and the values of options. As an example, the command

- `:- default(use(boot,system),set(trace)).`

effects that all declared objects will inherit the objects *boot* and *system*, and will have their trace option set. See sections A.4 and A.5.

Context switches with respect to the functionality of a process may be effected by manipulating a stack of databases. For each evaluation process, clauses that are asserted or retracted take effect only for the evaluation performed by the process itself. Such modifications do not affect the functionality of the object to which the process refers. The state of a process, that is the clauses that are active, may be saved or restored in the following way. When clauses have been asserted, the goal

- push()

effects that the current database of the process is pushed onto a stack.

The goal

- pop()

restores the old situation by popping the stack.

Process splitting is allowed, in addition to creating a new process when asking an object to evaluate a goal. A copy of the current process can be made by using the expression new(this) or new(this@N) that simplify to a copy of the current process, in the latter case allocated at the processor node denoted by N.

Such a process may be asked to evaluate a goal. For instance, when we like to use the functionality of a particular process, we may state the goal

- new(this)!G

to have G evaluated by a copy of the current process.

Object to process conversion takes place when asking an active object for a resumption. The goal sequence

- O = new(c(t)), O?

results in creating an object for which the constructor $c(t)$ is evaluated, that is subsequently asked for the resumption resulting from evaluating $c(t)$. The request to the object O is simply redirected to the constructor process for O.

Accessing non-logical variables is facilitated by expressions of the form

- O@*name*

denoting the non-logical variable *name*. of the object O. C.f. chapter 4. We allow goals of the form

- O@name := t

to assign t as a value to the non-logical variable *name* of the object O.

Global non-logical variables may be declared by unit clauses of the form

- global *name*.

in the same way as ordinary non-logical variables. These variables may also be initialized when declared. A global non-logical variable may be accessed by using an expression of the form global(name). When such an expression occurs on the left hand side of an assignment, the (global) non-logical variable *name* is assigned the value on the right hand side.

Clashes between names of objects, non-logical variables, globals and options may be resolved by using the expressions

- obj(*name*)
- val(*name*)
- global(*name*)
- option(*name*)

indicating that respectively an object, a non-logical variable, a global non-logical variable or an option value must be delivered.

Hiding names local to an object may be done by indicating which names are private, with a declaration of the form

- private f_1, f_2.

which results in post-fixing the function symbols f_1 and f_2 with the name of the object in which the declaration occurs. The *private* list may contain arbitrary function symbols, even for instance the function symbol ';' provided that it is quoted.

Allocation of objects and processes is handled by *node expressions*. See section 2.7. We allow the following node expressions.

- () - allocation is left to the system,
- $\{N_1, ..., N_n\}$ - selects one of the processor nodes $N_1, ..., N_n$,
- $T : N$ - delivers the node on the N-th branch following the tree T (as explained in section 2.7),
- $N_1 \# N_2$ - delivers a process from the (imaginary) matrix of processors (as explained in section 2.7),
- $[N \mid T]$ - is just another notation for $T : N$, and finally
- *here* - allocates the object or process to the same processor.

The parameters of both the (imaginary) tree and the (imaginary) matrix of processors may be modified by setting the appropriate options, as discussed above.

Implementation units

A description of the implementation has been given in chapter 11. Below we repeat the outline of the prototype given in section 9.5, now with the actual unit names.

N In the (pseudo) unit N we have defined a number of frequently occurring abbreviations as macros. Apart from *type* abbreviations we define a conditional result statement and the possible values of the state variable of an evaluation process. Except for these definitions this unit is empty.

T The implementation of *terms* is contained in the unit T, together with a number of definitions to facilitate the use of terms. In order to handle terms conveniently we define constructors, tests and selectors for the major kinds of terms.

P The unit P contains the code for *the sequential Prolog interpreter*, including the evaluation and unification functions and the function for composing so-called *database* functions.

O In the unit O the implementation of *objects* in the sense of DLP, can be found. Objects contain part of the functionality necessary to handle a request for a rendez-vous. Also the code for dealing with inheritance is provided there.

Q The unit Q defines the global *knot* that maintains all global information, and the global *supervisor*, that manages the objects. This unit also contains the definition of the evaluation processes created for each inference to take care of tracing and communication with the environment.

X *Non-logical variables and channels* are defined in the unit X. For channels we define the read and write methods involved in synchronous communication.

D The unit D contains *the initial database*. Apart from a simplification function that is used in the interpretation of special forms, this unit contains also the definition of the primitives needed for process creation, communication over channels, and for engaging in a rendez-vous.

U Finally, the unit U contains a variety of utilities for manipulating terms. We have also collected there a number of auxiliary definitions.

As additional units, not contained in this listing, we have the unit

IO that defines some file handling primitives and the global object *io*, which handles all user input and output; and the units

L defining the lexical analyzer that reads in the input file, and

Y that converts DLP programs to instructions for the global *supervisor*, which functions as an object manager.

Note that the units Q and IO specify global objects; respectively the objects *knot*, *supervisor* and *io*, that interact when reading in user input. Object declarations in DLP are actually read in as a sequence consisting of goals and clauses. For example the declaration

```
object a {                    corresponds to    object(a).
var k, n = 1.                  the sequence     var(k, n = 1).
...                                             ...
}                                               end_object.
```

The clauses and goals are read in one by one, by the global object *io*, and sent to the global *supervisor*. For unit clauses, it is tested whether they must be interpreted as a command. C.f. section A.5.2. Each newly declared DLP object is then added

to the object list of the global *knot*. As a remark, the parser defined in Y allows brackets to be omitted in a number of cases. However, brackets may always be used.

When reading the code, keep in mind that variables occurring in methods, not declared as temporary variables, are usually instance variables of the object/class for which the method is defined. However, our use of macros, defined in the units N, U and T, somewhat overturns the scoping rules as used in POOL-X. To aid in understanding the program text, we have included a glossary of the names defined as macros, aliases, globals, classes, functions and methods.

A.1 Types and abbreviations

file: N.h 1.3

As types we have

- A for strings,
- B for booleans,
- N for integers,
- V for logical variables,
- O for objects in the sense of DLP,
- Q for evaluation processes, and
- X for non-logical variables and channels

that function as basic types for the type

- T for terms.

For function terms, we use the type

- $F = A \times T \times T$

which is a recursive type, since F is a subtype of T. C.f. section A.2,2.

As abbreviations, for implementing the Prolog interpreter (c.f. section 11.3), we employ

- $E = V \rightarrow V$ for environments or renaming functions,
- $S = V \rightarrow T$ for substitutions,
- $C = N \times N \times S \rightarrow T$ for continuations,
- $D = Q \times T \times C \times N \times N \times S \rightarrow T$ for database functions, and
- $K = D \rightarrow D$ for (clause) continuations.

Also we use the type

- P for sequential (Prolog) inference processes.

As actual macro definitions we have:

df A = String
df B = Bool
df N = Int
df V = N

In other words, the identifiers A, B, N and V are replaced, in the program text, by their respective definitions. As a remark, when A occurs in an expression like $n@A$ for an integer variable n then this must be read, according to POOL* conventions, as $n!get\_String()$, which delivers the string representation of the integer value of n.

Abbreviations To be able to use conditional result **statements** we have included the following macro-definitions

df return(B,X,Y) = *if* B *then result* X; *else result* Y *fi*;
df returnil(T,B,X,Y,Z) = *if* T ≡ *nil then result* X *else return*(B,Y,Z) *fi*;

The second variant first tests whether the argument T is *nil*. It is mainly used in the unit U, defining the utilities.

The most important active objects are those embodying an evaluation process. The communication protocol used there makes use of a **state** variable, that can take the values defined here.

df BUSY = 0
df PEND = 1
df WAIT = 2
df STOP = 3
df FREE = 4
df HALT = 5

We define the major **derivation functions** as global functions. The function *infer* is used to start up a new inference. The function *eval*, *unify*, *assign* and *bagof* are used respectively to evaluate a goal, to perform a unification, for interpreting a special form involving equality and assignment, and to execute a *bagof* goal.

df infer = P.new()!!inferQ
df eval = P.evalQ
df unify = P.unifyQ
df assign = System.eq_assign
df bagof = System.bag_of

The definitions of a number of **global functions** are collected below.

df mape(E,T) = U.m_e(E,T)
df maps(S,T) = U.m_s(S,T)
df mapc(M,S,T) = U.m_c(M,S,T)

The function *mapc* is used to rewrite cuts in a proper form. The functions *maps* and *mape* are used to effect respectively renamings and substitutions. These functions, as the functions below, are primarily used in the unit P.

df compile = P.compile_D
df compose = P.composeD
df system = System.start_d

df yes = P.yes_q
df starts = P.start_s
df nilD = P.nil_D
df nilK = P.nil_K
df failD = P.fail_D
df checkcut = Q.check_cut

To collect the variables of a term we use *varsin*.

df varsin(T) = U.v_in(T)

We use compound terms for lists of terms and define functions for appending terms, and for checking whether a term is contained in a list.

df append(T1,T2) = U.t_app(T1,T2)

df member(T,L) = (U.t_in(eq_t,T,L) > 0)
df inlist(T,L) = U.t_in(eq_t,T,L)
df atlist(T,L) = U.t_at(eq_t,T,L)

Both the *inlist* and the *atlist* function return the position of the term in the list, the latter however does not return zero if the term is not in the list but modifies the list by making it an element of the list.

Further we define a function to select a list of terms satisfying some criterion, a function to add objects defined by a term T to a list of objects L, and a function to simplify terms.

df select(F,L) = U.sel(F,L)
df addchv(T,L,A) = X.add_chv(T,L,A)
df simplify = System.simpl

For actually allocating POOL* objects on processors we use the definitions below. The class *Node* is defined in the standard unit *Nodes*.

df node(Q,T) = U.nd(Q,0,T)
df my_node = (Node.my_Node()@N)

Finally, we define some macros for debugging and error messages.

df dbx(P,X) = *if* knot@debug *then* io!!p("***P:" + X); *fi*;
df perror(X) = io!!p("###error " + X)

A.2 Terms

A.2.1 Definitions

file: T.h *1.3*

Tags are used to employ a form of dynamic typing for terms. With the help of these tags we can define the type of a term by means of a kind of tagged variant record. We provide *constructors*, *selectors* and *tests* for the major term types. We have adopted the convention to use capitals for the first letter of a constructor, to use the postfix *of* for selectors that give access to the contents of a term and to use the prefix *is* for tests.

df CON = 1
df VAR = 2
df FUN = 3
df OBJ = 4
df PRC = 5
df CHV = 6
df REC = 7

Constructors must be used to create terms from objects of some basic type, or to create a function term from terms. In order to be able to employ union types in a convenient way, we have used a name concatenation facility of our macro processor, which removes the ._. part after applying all expansions. See section 9.5.1.

df mkterm(C) T.new._.C
df Con(X) = mkterm(CON)(X)
df Var(X) = mkterm(VAR)(X)
df Fun(S,T1,T2) = mkterm(FUN)(F.mk(S,T1,T2))
df Obj(X) = mkterm(OBJ)(X)
df Prc(X) = mkterm(PRC)(X)
df Chv(X) = mkterm(CHV)(X)
df Rec(X) = mkterm(REC)(R.new()!X)

Selectors are used to access the constituting part of a term. The selectors each address one of the fields of the union type. For instance *conof*(*t*) expands to *t*@1, which accesses the first field of *t*, containing a string.

df conof(T) = ((T) @ CON)
df varof(T) = ((T) @ VAR)
df funof(T) = ((T) @ FUN)
df objof(T) = ((T) @ OBJ)
df prcof(T) = ((T) @ PRC)
df chvof(T) = ((T) @ CHV)
df recof(T) = ((T) @ REC)

We can further analyze a term by using the definitions below.

df fof(T) = U.func_of(T)
df fc(T) = T @ FUN @ f
df a1(T) = (T @ FUN @ arg_1)
df a2(T) = (T @ FUN @ arg_2)
df arg(N,T) = U.get_n(N,argsof(T))
df arity(T) = U.length(argsof(T))
df argsof(T) = U.args_of(T)

Tests are provided for determining the type of a term.

df tagof(T) = (T @ switch)
df symbolic(T) = (tagof(T) < OBJ)
df iscon(T) = (tagof(T) ≡ CON)
df isvar(T) = (tagof(T) ≡ VAR)
df isfun(T) = (tagof(T) ≡ FUN)
df isobj(T) = (tagof(T) ≡ OBJ)
df isprc(T) = (tagof(T) ≡ PRC)
df ischv(T) = (tagof(T) ≡ CHV)
df isrec(T) = (tagof(T) ≡ REC)
df isun(T) = (a2(T) ≡ *nil*)
df isbin(T) = ¬ isun(T)
df isconst(C,T) = U.is_con(C,T)
df isfc(F,T) = U.is_fun(F,T)
df isvarno(N,T) = U.is_var(N,T)

Compound terms, that are function terms having a *comma* as a function symbol, play a special role, and therefore deserve a special test and constructor.

```
df isc(T) = (isfun(T) & fof(T) = ",")
df mkc(A1,A2) = Fun(",",A1,A2)
```

Auxiliary constructors, selectors and tests are provided for convenience.

```
df unbound(V,S) = isvarno(V,S._.(V))
df isground(T) = U.is_ground(T)
df maxvar(T) = U.max_v(T)

df Int(N) = Con((N)@A)
df isint(T) = U.is_int(T)
df intof(T) = U.int_of(T)

df Error = Con("error")
df Clause(H,B) = Fun(" :- ",H,B)
df Fact(F) = Clause(F,Nil)
df Query(C) = Un("?-",append(C,Nil))
df Op(O,A1,A2) = Fun(O,A1,A2)
df Un(O,A) = Fun(O,A,nil)
df Dot(T1,T2) = Op(".",T1,T2)
df Nil = Con("[]")
df Fail = Con("fail")
df True = Con("true")
df Eq(T1,T2) = Op(" = ",T1,T2)
df Initial = mkc(Nil,nil)
df Cuton(M,S) = Fun("cuton",Int(M),Con(S))
df Cut(M,T) = Dot(Int(M),T)
df Yes(T) = Un("yes",T)
df Mbr(T,L) = Op("member",T,L)

df isnil(T) = isconst("[]",T)
df isfail(T) = isconst("fail",T)
df iscut(T) = ( isconst("!",T) | fof(T) ≡ "cuton" )
df isnull(T) = isvarno(0,T)
df isunit(T) = isconst("()",T)
df iscollect(T) = ( isfc("bagof",T) | isfc("setof",T) )
```

Recall that we have implemented logical variables as integers. We use the variable with number zero as the anonymous or null variable, that in Prolog is written as an underscore.

To make a copy of a term we use *cpyof*.

```
df cpyof(T) = U.t_cpy(T)
```

We use *isfuncar* to check if a term is a function term with function-symbol S and arity N.

```
df isfuncar(S,N,T) = U.is_funcar(S,N,T)
```

Further we may like to check if a term is a clause, and if so what kind of clause.

```
df isfact = U.is_fact
df isclause = U.is_clause
df headof = U.head_of
```

To extract a name from a term, containing possibly apart from a name also an initialization part or inheritance information, we use the function *nameof*.

df nameof = U.g_n

Partly the definitions are stated in terms of prior definitions and partly as an abbreviation for a function defined in the unit U. For a better understanding of the program text it is worth the trouble to study the definitions given above with some care and to inspect the implementation given in the unit defining the utilities. See section A.8.2.

For testing the **equality** of two terms we provide two functions, the latter of which allows a kind of matching.

df eq_t = U.t_eq
df match = U.t_mt

In some cases it is required to convert a compound term to a Prolog list, and vice versa.

df t_pl = U.t_pl
df pl_t = U.pl_t

For converting a term to string we provide two functions. The second differs from the first only in that it puts some space around the term.

df str(T) = U.t_str(T)
df qstr(T) = " " + str(T) + " "

A.2.2 Implementation

```
impl unit T                                              file: T.i    1.9

use Q O X
include N U T

alias T = Union7(A,V,F,O,Q,X,R)

class F
var f put get : A, arg_1 put get : T, arg_2 put get : T

fn mk(s:A,t1,t2:T):F
temp x := F.new()
{
if t2 ≡ nil & isc(t1) then x@arg_2 := a2(t1); x@arg_1 := a1(t1)
else
    x@arg_1 := t1; x@arg_2 := t2;
fi; x@f := s; result x;
} mk

end F
```

The code above shows that a term is defined as the union of a number of components, respectively a constant (string), a logical variable, a function term, an object, a process and a component for a non-logical variable or a channel. Notice that the function *mk* of the class *F* takes care of left justifying a function term, by putting the first term of its argument list in its first argument variable and the rest in the second, either as a *nil* term, a simple term or a compound term. This way of dealing with argument lists allows functions on terms to use a direct recursion on subterms, instead of a more expensive indirect recursion via a list of arguments.

An auxiliary class R is introduced for storing database functions and for possible future extensions.

```
class R var t put get : T, d put get : D end R
```

A.3 The sequential Prolog interpreter

```
impl unit P                                          file: P.i    1.3

use IO Q D T Text
include N U T
```

We need aliases for the types of environments (or renaming functions) E, substitutions S, continuations C, database functions D and clause continuations K.

```
alias E = fn(V):V
alias S = fn(V):T

alias C = fn(N,N,S):T
alias D = fn(Q,T,C,N,N,S):T
alias K = fn(D):D
```

The code for sequential (Prolog) inference processes is collected in the following class definition.

```
class P
```

For creating a new inference process we have defined a macro *infer* in section A.1, that expands to $P.new()!!inferQ(...)$, with the (asynchronous) method *inferQ* as defined below.

```
method inferQ(q:Q,t:T,c:C,m,n:N,s:S)   { eval(q,t,c,m,n,s) } inferQ   infer[1]
```

[1]For each method or function definition corresponding with a global macro definition, as defined in the unit N we have given the global name as a comment.

A.3.1 Evaluation

The function *eval* extends the evaluation function described in section 11.3.1. Among other things it tests for termination due to a halt statement.

```
fn evalQ(q:Q,t:T,c:C,m,n:N,s:S):T                                    eval
temp z,a,g:T, asktail:C
{
if t ≡ nil then result c(m,n,s)
elsif q@state = STOP then result nil
else
 if isc(t) then q!tr("compound",m,t); a := a1(t); g := a2(t) else a := t fi;
 asktail := fn (m1,n1:N, s1:S) : T
          temp mct := m1 , rx := maps(s1,a)
          {
          if iscut(a) then mct := mct + 1; fi;
          q!tr("exit",m,rx); result evalQ(q,g,c,mct,n1,s1)
          };
 a := maps(s,a); if fof(a) = "new" then a := Eq(Var(0),a) fi;
 if simplifiable(fof(a)) & q[NOSIMPLIFY] = 0 then
       a := simplify(q,0,n,a);
 fi;
 a := q!tr("call",m,a);
 if fof(z) = "halt" then result nil
 elsif isnil(a) then result asktail(m,n,s)
 else result q@d(q,a,asktail,m,n,s)
 fi;
fi
} evalQ
```

A.3.2 Compilation

To be able to trace which database function is used for evaluating a goal, a function is created that contains, apart from the compiled database, also a call to the evaluation process to give notice of the name of the object to which the database belongs.

```
fn compile_D(o,cl:T,d:D,k:K):D                                       compile
temp d1 := fn(q:Q,t:T,c:C,m,n:N,s:S):T
        { q!tr("D:" + str(o),m,t); result d(q,t,c,m,n,s); }
{ io!!p("***compile:" + qstr(o)); result compileD(cl,d1,k); } compile_D
```

The actual compilation is taken care of by the following function.

```
fn compileD(cl:T,d:D,k:K):D
temp
 newd:D, dotail:K
{
if cl ≡ nil then dbx(compile," empty clause "); result k(d)
elsif isnil(cl) then return(k ≡ nil,d,k(d))
elsif isc(cl) then
```

```
  dotail := fn (d1:D):D { result compileD(a2(cl) , d1, k) };
  result compileD(a1(cl),d,dotail);
 else
  dbx(co_C,str(cl));
  newd := fn(q:Q, t:T, c:C, m,n:N, s:S):T
        temp fs:A, b:B, m1,n1,nvars:N, e:E, s1:S, bdy:C, ru,rd:T
        {
        fs := fof(a1(cl)); rd := d(q,t,c,m,n,s);
        if fof(t) ≠ fs then result rd
        else b := checkcut(q,t,m,n);
          if b then q!tr("cut*",m,t); result rd
          else
            m1 := m + 1; nvars := maxvar(cl); n1 := n + nvars;
            e := fn (v:V):V { result v + n };
            s1 := fn(v:V):T { return(v > n,Var(v),s(v)) };
            bdy := fn (m2,n2:N,s2:S):T
            { result evalQ(q,mapc(m,fs,mape(e,a2(cl))),c,m2,n2,s2) };
            q!tr("try",m,cl);
            ru := unifyQ( mape(e,a1(cl)),t,bdy,m1,n1,s1);
            result append(rd,ru);
          fi;
        fi;
        }; %% newd
   return(k ≡ nil,newd,k(newd))
 fi
 } compileD
```

As an optimization, we check whether the predicate name of the goal is identical to the predicate name of the head of the clause.

A.3.3 Unification

We have employed a rather simple unification algorithm, using a direct recursion on subterms, when necessary. Notice that whenever we encounter two variables we update the older variable, the one with the lowest number, since the younger variable will become unbound first.

```
fn unifyQ( t1,t2: T, c:C, m,n:N, s:S ):T                                unify
temp
 doa:C
 upd := fn (v:V,t:T):T
      temp news := fn (v1:V):T { return(v ≡ v1,t,s(v1)) }
      { return(unbound(v,s),c(m,n,news),unifyQ(s(v),t,c,m,n,s)) }
{
if t1 ≡ nil | t2 ≡ nil then return(t1 ≡ t2,c(m,n,s),nil)
elsif isnull(t1) | isnull(t2) then result c(m,n,s)
else
case tagof(t1) of
CON then case tagof(t2) of
       VAR then result upd(varof(t2),t1) or
       CON then return(conof(t1) ≡ conof(t2),c(m,n,s),nil)
       else result nil esac or
```

```
VAR then if isvar(t2) then
          if varof(t1) = varof(t2) then result c(m,n,s)
        elsif varof(t1) < varof(t2) then result upd(varof(t1),t2)
        else result upd(varof(t2),t1)
        fi
      else result upd(varof(t1),t2) fi; or
FUN then case tagof(t2) of
      VAR then result upd(varof(t2),t1) or
      FUN then if fc(t1) = fc(t2) then
                 doa := fn(m1,n1:N,s1:S):T
                   { result unifyQ(a2(t1),a2(t2),c,m1,n1,s1) };
                 result unifyQ(a1(t1),a1(t2),doa,m,n,s)
                 else result nil fi
      else result nil esac
else result nil esac fi
} unifyQ
```

A.3.4 Initialization

The function *yes_q*, implementing *yes*, results in a continuation that notifies the evaluation process q of the answers that result from evaluating a goal.

```
fn yes_q(q:Q, m:N, t:T):C                                        yes
temp
 l := Yes(varsin(t))
 c := fn(m1,n:N,s:S):T
 temp r := maps(s,Dot(t,l)) { q!tr("yes",m,r);result a2(r) }
{ result c; q!tr("goal",m,Dot(t,l)) } yes_q
```

Next we give the definition for the initial substitution *starts*, the initial database *nilD* and the initial clause continuation *nilK*.

```
fn start_s(v:V):T { result Var(v) } start_s                      starts

fn nil_K(d:D):D { result d } nil_K                                nilK
fn nil_D(q:Q,p:T,c:C,m,n:N,s:S):T { result nil } nil_D            nilD
```

The always failing database function *failD* results, apart from notifying the evaluation process of failure, in *nil*.

```
fn fail_D(q:Q,p:T,c:C,m,n:N,s:S):T { q!tr("fail",m,p); result nil} fail_D   failD
```

A.3.5 Composition

In the composition function *compose*, the parameter b determines whether the overwriting variant of composition is used ($over = 1$), or the backtracking variant (which is the default, $over = 0$).

```
fn composeD(over:N,d1,d2:D):D                                    compose
temp d := fn(q:Q,t:T,c:C,m,n:N,s:S):T
        temp r := d1(q,t,c,m,n,s)
        {
        if over ≡ 0 then result append(r,d2(q,t,c,m,n,s))
        elsif r ≡ nil then result d2(q,t,c,m,n,s)
        else result r
        fi;
        }
{ return(d2 ≡ nilD,d1,d) } composeD
end P
```

A.4 Objects

```
impl unit O                                          file: O.i    1.3
use IO Q D P U T, Nodes
include N U T
```

The unit O implements objects in the sense of DLP. The standard unit Nodes of POOL* is used to be able to allocate such objects on a particular processor node.

As attributes a DLP object has among others an *instance* number (indicating the number of copies made), a process number (for keeping record of the evaluation processes created for the object), clauses, a use list and an isa list that contain the names of objects from which the object inherits clauses or non-logical variables. In addition a *private* list is used for making names local.

Further we need a number of attributes to govern the communication behavior of an object, first of all an accept list that names the methods for which a request may be accepted, an accept queue that stores requests that have not yet been accepted and a process stack that contains the processes for which an accept statement is executed. The attribute *constructor* contains the constructor process. For passive objects the value of the attribute is *nil*.

```
class O
newpar(nodenumber:N,name:T)
var
    instance put get := 0                          instance copy number
    procno put get := -2                    the number of active processes
    override := 0                                    composition mode
    clauses put get : T := Initial                        the clauses
    chvl put get := Initial                        non-logical variables
    chvar put get := Array(X).new(0,MAXCHV)
    usel put get := Initial                                use list
    isal put get := Initial                                isa list
    privl put get := Initial                            private list
    d put get : D := nilD                          the local database
    optar put get := Array(N).new(0,MAXOPT)             the options
```

```
ud put get : D := nilD                    the database of the used objects
fd put get : D                                          the final database

acceptlist put get := Con("any")                          the accept list
acceptqueue put get :T                                   the accept queue
qtop, qstack : T                                        the process stack
accno := 0
post := Dot(Int(0),Nil)              for the result of an accept statement

constructor put get : Q                                 the constructor process
active put get := false                                  mutual exclusion mode

compiled := false
locked := false

me := Obj(self)
```

When creating an object then first all options are set to zero, and thereafter the defaults contained in the global *knot* are installed by the method install. See section A.4.4.

```
init
 for i from 0 to MAXOPT do optar[i] := 0 od; optar[TRACE] := 0;
 if fof(name) ≠ "user" & fof(name) ≠ "system" then install(knot@default)
   fi;
 chvar[0] := X.new(Nil);
tini
```

The methods that may safely be applied when the object is locked are collected in the macro SAFE.

```
df SAFE = get_nd, get_id, get_name, cpy, get_cpy, \
        put_optar, get_optar, put_chvl, get_chvl, \
        put_privl, get_privl, put_acceptlist, get_acceptlist, \
        put_instance, get_instance, put_procno, get_procno, \
        put_usel, get_usel, put_ud, get_ud, \
        put_constructor, get_constructor, \
        put_active, get_active, set, install, \
        put_clauses, get_clauses, put_d, get_d, get_ini, \
        chv, accept, get_post
```

A.4.1 The protocol

The protocol simply states that anything is accepted unless the object is locked. An object is locked when it is busy evaluating a goal. It is unlocked when it needs a new goal to evaluate, that is when either the accept queue is empty, or no request in the accept queue satisfies the accept list. Notice that a lock affects only the method *request*.

```
body do if locked then answer( SAFE, unlock) else answer any fi od ydob
```

Locking involves the use of the (internal) methods *lock* and *unlock*.

```
method lock():T { if active then locked := true fi; result me } lock
method unlock():T { locked := false; result me } unlock
```

Objects have an eternal life, in principle. For eliminating objects no longer in use, we rely on the garbage collector of POOL* , that removes objects to which there exists no reference.

Administrative functions For access to the non-logical variables of the object the method *chv* must be used. Notice that we have used an array in combination with a compound term to store non-logical variables. We use such pairs, in this case (*chvl*, *chvar*), consisting of a compound term and an array, as a symbol-table. The compound term *chvl* contains the name of the variable and the array *chvar* the actual non-logical variable. The function *inlist* returns the position of the name in the list, which is zero if it does not occur. This position is then used to access the array containing the actual non-logical variables.

```
method chv(t:T):X { result chvar[inlist(t,a2(chvl))] } chv
```

To ask for the node at which the object is located, the identity of the object, or its name, one of the following methods must be used. Recall that the expression *instance*@*A* results in the string representation of the *instance* number.

```
method get_nd():N { result nodenumber } get_nd
method get_id():A { result str(name) + "." + instance@A + "." + procno@A
    } get_id
method get_name():T { result name } get_name
```

One can allocate a copy of an object on some specific node. By default an instance is allocated on the same processor node as the declared object. See section A.4.4 for the details of the initialization of an object.

```
method get_cpy():O { result cpy(nodenumber) } get_cpy

method cpy(n:N):O
temp o := O.new(n,name) (* ALLOC AT n *)
{
instance := instance + 1; o@instance := instance; o@active := active;
if ¬ compiled then get_ini() fi; o@fd := fd;
for i from 0 to MAXOPT do o@optar[i] := optar[i] od;
o@clauses := clauses; o@usel := usel; o@ud := ud; o@d := d;
for i from 0 to MAXCHV while chvar[i] ≠ nil do
            o@chvar[i] := X.new(chvar[i]@name)!put_val(chvar[i]@val)
            od;
o@chvl := chvl; o@privl := privl; result o
} cpy
```

The contents of an object can be printed.

```
method get_String(): A
{
result "\n" + " :- " + str(Un("object",name)) + "." + "\n"
        + " :- " + str(Un("var",chvl)) + "." + "\n"
        + " :- " + str(Un("use",usel)) + "." + "\n"
```

```
        + " :- " + str(Un("isa",isal)) + "." + "\n"
        + " :- " + str(Un("private",privl)) + "." + "\n"
        + str(clauses) + "\n" + " :- end_object." + "\n";
} get_String
```

Creating an **evaluation process** for an object requires that the object is compiled. Apart from information concerning the identity of the object, it is also given the compiled database function of the object, and the options set for the object, as an argument.

```
method process(n:N):Q
{
if ¬ compiled then get_ini() fi; procno := procno + 1;
result Q.new(n,get_id(),self,fd,optar)
} process
```

A.4.2 Acceptance

An **accept statement** results in setting the *accept* list. The result of the method call is determined by the result of checking the accept queue. If any of the requests stored in the accept queue satisfies the accept list, a process is created to evaluate that request.

```
method accept(q:Q,n:N,t:T):T
{
accno := n; active := true; q!lock(); push(q); acceptlist := t;
result checkqueue();
} accept
```

The process stack is used for temporarily storing the process invoking the method *accept*. The instance variable *qtop* points to the top of the stack, and *qstack* contains the remaining part.

```
method push(q:Q):T
{
if qstack ≡ nil & qtop ≡ nil then qtop := Prc(q)
elsif qstack ≡ nil then qstack := qtop; qtop := Prc(q);
else qstack := mkc(qtop,qstack); qtop := Prc(q);
fi; result me;
} push

method pop():T
temp r := qtop
{
if qstack ≡ nil then qtop := nil
elsif ¬ isc(qstack) then qtop := qstack; qstack := nil;
elsif isc(qstack) then qtop := a1(qstack); qstack := a2(qstack);
else perror("O-pop")
fi; result r
} pop
```

Processing the accept queue occurs when a new accept list has been installed. Each request stored in the accept queue is tried to see whether it satisfies the accept conditions, until one doing so is found. The request is then evaluated as an ordinary goal.

```
internal method checkqueue():T
temp r,t,h:T, p := acceptqueue, q:Q, m,n:N, g := Initial
{
acceptqueue := nil;
while p ≠ nil & r ≡ nil do
      if isc(p) then h := a1(p); p := a2(p) else h := p; p := nil fi;
      q := prcof(a1(h)); m := intof(a1(a1(a2(h))));
      n := intof(a1(a2(a2(h)))); t := a2(a2(h));
      if acceptable(t,n,g) then r := Nil; evaluate(q!unlock(),m,g) fi;
      if r ≡ nil then acceptqueue := append(acceptqueue,t)
      else acceptqueue := append(acceptqueue,p);
      fi;
od; if r ≡ nil then unlock(); r := Nil fi; result r;
} checkqueue
```

To determine whether a request is acceptable we use the method defined below. Some forms occurring in an accept statement may require a more extensive check than just for the occurrence of the name of the method.

For testing whether a method call *mc* satisfies a conditional accept expression of the form *method* : *guard* → *goal*, a process is created to check if *mc* unifies with *method* and the *guard* evaluates to true. The attribute *guard* of the evaluation process must be set, to prevent deadlock to occur when simplifying non-logical variables. See section A.7.1.

The substitutions resulting from the evaluation of the guard are given to the process evaluating the goal by modifying the goal, to execute the resumption containing these substitutions. In order to communicate the bindings resulting from the acceptance and evaluation of the method call, a non-logical variable *c* is used, that in a sense links the process evaluating the goal to the process for which the accept statement is evaluated. The attribute *post* of the object is used (as a stack, since accept statements may occur nested) to store the information needed for that process to continue. To cope with failure, when evaluating the goal, the double "##" is used as a function symbol. See also section A.5.3.

The parameter *g* of *acceptable* is supposed to be a compound term. It is used to store the resulting goal, in the first part. Note that the assignment $a_1(g) = mkc(...)$ is used to avoid the automatic left justification of compound terms, as would occur when using a term constructor. C.f. section A.2.2. The second part stores what must be communicated to the process calling the method. The variables occurring in the conditional accept expression remain hidden this way.

```
internal method acceptable(mc0:T,n:N,g:T):B
temp mc := mc0, fx,f:fn(T,T):B, e,e1,e2:fn(V):V, b := false, i:N,
   g1,g2, r,rt,rg,lt,mt,gt,st:T, q:Q , c := X.new(nil),
   pst := Dot(Int(accno),Nil)
{
```

```
if isconst("any",acceptlist) then a1(g) := mc; a2(g) := mc; b := true
else
fx := fn(t0,t1:T):B { return(iscon(t1),fof(t0) = fof(t1),match(t1,t0)) };
f := fn(t0,t1:T):B {
    if iscon(t1) then result fof(t0) ≡ fof(t1)
    elsif isfuncar(":",2,t1) then result fx(t0,a1(t1))
    elsif isfuncar(" → ",2,t1) then result fx(t0,a1(a1(t1)))
    elsif isfun(t1) then result fx(t0,t1)
    else result false
    fi
    };
i := U.t_in(f,mc,acceptlist);
if i > 0 then mt := U.get_n(i,acceptlist); lt := mt fi;
if mt ≡ nil then b := false
elsif iscon(mt) then a1(g) := mc; a2(g) := mc; b := true
else
    e := fn(v:V):V { result v + n };
    e1 := fn(v:V):V { return(v ≥ n,v-n,0) };
    e2 := fn(v:V):V { return(v < n,v,0) };
    if fc(mt) ≡ " → " then
      st := mape(e,a2(mt)); mt := a1(mt) else st := mc
    fi;
    if fc(mt) ≡ ":" then gt := mape(e,a2(mt)); mt := a1(mt) fi;
    if gt ≢ nil then rg := gt fi;
    if isfun(mt) then rg := mkc(Fun("unify",mc,mape(e,mt)),rg) fi;
    if rg ≡ nil then b := true
    else
        q := process(nodenumber); q@guard := true; q[NSOL] := 1;
    q[TRACE] := 1;
       infer(q,rg,yes(q,0,rg),0,maxvar(rg),starts); r := q@sols;
       if r ≢ nil then b := true;
            g1 := Op(" := ",Chv(c),mape(e,lt));
            g2 := Op(" = ",lt,Chv(c));
            rt := Fun("##",Chv(c),mape(e,lt));
         a1(g) := mkc(mkc(Fun("unify",r,Yes(varsin(rg))),rt),mkc(st,g1));
            a2(g) := mc;
            pst := Dot(Int(U.max(maxvar(g),accno)),g2);
       fi;
    fi;
fi; fi;
if b then post := mkc(pst,post) fi; result b
} acceptable
```

Conditional accept expressions require some processing after acceptance. We use an attribute *post* to store part of the information that is needed.

```
method get_post():T
{
if isc(post) then result a1(post); post := a2(post) else result post fi
} get_post
```

A.4.3 Method calls

The **handling of requests** is done by the method *request*. If the process given as an argument to *request* is *nil* then a new process is created. If no mutual exclusion is in effect then an inference process may be started without further ado. Otherwise it must be determined whether the request is acceptable. If so, the object is locked and a process is started to evaluate the goal, but not before a provision is made to unlock the process that belongs to the current accept list. This process may then be popped from the process stack. If the request is not acceptable, the process and the goal together with an indication of the depth of the inference is put in the accept queue for later treatment. Acceptability is tested by the (internal) method *acceptable*, that may, in case the accept list contains conditional accept expressions, start up an inference process to evaluate the guard. The call to *acceptable* is given an argument g, that initially contains the term ([], *nil*), which is filled with a proper goal that is given to the method *evaluate* when the call succeeds.

```
method request(q0:Q,m,n:N,t:T):T
temp q := q0, g := Initial ,r:T
{
if q ≡ nil then q := process(nodenumber) fi; r := Prc(q);
if ¬ active then infer(q,t,yes(q,m,t),m,maxvar(t),starts);
elsif acceptable(t,n,g) then lock(); evaluate(q,m,g)
else
 acceptqueue := append(acceptqueue,Dot(Prc(q),Dot(Dot(Int(m),Int(n)),t)));
  q!tr("wait",m,me); q!lock();
fi; result r
} request
```

The method *evaluate* expects a compound term of which the first component contains the goal that must actually be evaluated. The second component is supposed to contain the original method call. This part is expanded to a term of the form $'m(t)\#topq'$, for $m(t)$ the method call and *topq* the top of the process stack, if the process stack is not empty, in order to release the processes that imposed the acceptance conditions, as soon as the first result is communicated. See also section A.5.3.

```
internal method evaluate(q:Q, m:N, g:T):T
temp t := a2(g)
{
if qtop ≠ nil then t := Fun("#",a2(g), qtop); fi;
infer(q,a1(g),yes(q,m,t),m,maxvar(g),starts); pop(); result nil
} evaluate
```

A.4.4 Inheritance and compilation

Inheritance of non-logical variables is achieved by copying the non-logical variables of the objects occurring in the *isa* list, and recursively of the objects that occur in their *isa* lists. A check is made for cycles and double occurrences of objects. The term l is an object name or a list of object names and r is the list of

names of objects that have already been inherited. If the term l is compound then both components are treated recursively. Note that in the second call to *inhv*, the list r may have been changed by appending object names from the first component of l.

```
internal method inhv(l:T,r:T):O
temp m:O
{
if l ≡ nil then ;
elsif isnil(l) then ;
elsif ¬ isc(l) then dbx(inherit-check,qstr(l) + " :: " + qstr(r))
  if ¬ member(l,r) then
      m := supervisor[l]@ini; copyv(m@chvl,m@chvar); append(r,l);
  fi;
else inhv(a1(l),r); inhv(a2(l),r);
fi; result self
} inhv
```

Copying non-logical variables is straightforward.

```
internal method copyv(l:T,ar:Array(X)):O
temp i := 1,j := 1
{
while chvar[j] ≠ nil do j := j + 1 od;
while ar[i] ≠ nil do
      chvar[j] := X.new(ar[i]@name); i := i + 1; j := j + 1
      od;
chvl := append(chvl,a2(l)); result self;
} copyv
```

Compilation requires that the database functions of the objects inherited via the *use* lists are put in composition with the function that results from compiling the own clauses of the object. Notice that the mode of composition depends on whether the overriding option is set.

The method *inhd* has the same structure as the method *inhv* described above.

```
internal method inhd(l:T,r:T):D
temp d1,d2:D, cl:T, o:O, n:N
{
if l ≡ nil then result nilD
elsif isnil(l) then result nilD
elsif ¬ isc(l) then
      if ¬ member(l,r) then dbx(inhd,get_id() + qstr(l));
            o := supervisor[l]@ini;
            d1 := o@d; d2 := inhd(o@usel,append(r,l));
            return(d2 ≡ nilD,d1,compose(override,d1,d2));
      else
            result nilD
      fi;
else
      d1 := inhd(a1(l),r); d2 := inhd(a2(l),r);
      return(d1 ≡ nilD,d2,compose(override,d1,d2))
```

```
fi;
} inhd
```

Initialization is effected by the method *get_ini*. The combined database of the object, consisting of its own database *d* and the inherited database *ud*, is stored in the attribute *fd*. All non-logical variables, including those inherited, are initialized by applying the simplification function described in section A.7.1.

```
method get_ini():O
temp i := 1, q:Q
{
if ¬ compiled then procno := procno + 1; dbx(initialize,get_id());
q := Q.new(nodenumber,get_id(),self,fd,optar);
d := compile(name,clauses,nilD,nilK);
ud := inhd(usel,mkc(name,Nil)); inhv(isal,mkc(name,Nil));
fd := compose(OVERRIDE,
          compose(override,d,ud),compose(override,system,failD));
while chvar[i] ≠ nil do
      chvar[i]@val := simplify(q,0,0,chvar[i]@val); i := i + 1
      od
fi; compiled := true; result self;
} get_ini
```

Global default settings are installed by the method *install*.

```
method install(t:T):T
{
if t ≡ nil then ;
elsif isnil(t) then ;
elsif isc(t) then install(a1(t)); install(a2(t));
else add(t)
fi; result me
} install
```

Commands received from the *supervisor* are evaluated by the method *add*. See section A.5.2 for a description of the *supervisor*. When no keyword is recognized, the command is taken to be a request for evaluating a goal. Care is taken to initialize the object before evaluating a goal. See section A.8.1 for the list of keywords used.

```
method add(t0:T): T
temp s := fof(t0), l, r := me, t := t0, n:N
{
if isfact(t) then
  if keyword(fof(headof(t))) then s := fof(headof(t)); t := headof(t) fi;
fi;
if t ≡ nil then perror("O add: nil")
elsif s = "use" then append(usel,argsof(t))
elsif s = "isa" then append(isal,argsof(t))
elsif s = "inherit" then append(usel,argsof(t));
```

```
                append(isal,argsof(t))
elsif s = "var" then chvl := addchv(argsof(t),chvl,chvar)
elsif s = "private" then append(privl,argsof(t))
elsif s = "deterministic" then set(Un("set",t))
elsif s = "accept" then active := true; acceptlist := argsof(t)
elsif s = " :- " then append(clauses,rname(t))
elsif s = "lock" then active := true
elsif s = "override" then override := 1
elsif s = "set" then set(t)
elsif s = "clear" then n := inlist(a1(t),options); optar[n] := 0
elsif s = "option" then optar[intof(a1(t))] := intof(a2(t))
else get_ini(); r := request(nil,0,0,t)
fi; result r
} add
```

There are several ways of setting an option.

```
method set(t:T):T
temp l,r:T, n:N
{
if t ≡ nil then ;
elsif isnil(t) then ;
elsif isc(t) then set(a1(t)); set(a2(t));
else
 l := a1(t); r := a2(t);
 if isfuncar(" = ",2,l) then r := a2(l); l := a1(l) fi;
 n := inlist(l,options);
 if r ≡ nil then optar[n] := 1 else optar[n] := intof(r) fi;
fi; result me
} set
```

The renaming of function names, according to the *private* list is done simply by postfixing the object name. To understand the use of U.t_in it may help to recall that we have defined member(T,L) as (U.t_in(eq_t,T,L) > 0).

```
internal method rname(t:T):T
temp f: fn(T,T):B
{
if t ≡ nil then ;
elsif isfun(t) then
      f := fn(t0,t1:T):B { result fof(t0) = fof(t1) };
      rname(a1(t)); rname(a2(t));
      if U.t_in(f,t,a2(privl)) > 0 then fc(t) := fc(t) + "_" + fof(name) fi;
fi; result t
} rname
end O
```

A.5 Processes

impl unit Q *file: Q.i 1.3*

```
use IO D O P U T Nodes
include N U T
```

A.5.1 Global information

All global information is stored in the global POOL* object *knot*. We also declare a global *supervisor*, and a global list of option names.

```
global knot := Knot.new()
global supervisor := Supervisor.new()
global options := Options
```

The *knot* object contains the global information that is necessary for the creation of objects and processes.

```
class Knot
var
        unum := 0                          for generating fresh numbers
    cnd get := 0                           the current processor node
    state put get := 0                     the global state
```

We again use pairs of the form (*list*, *array*) as a symbol-table for storing the declared objects and the global non-logical variables. C.f. section A.4.1. The list of object names is initialized for the objects *system* and *user*.

```
objl get := mkc(Con("system"),Con("user"))
objar get := Array(O).new(0,MAXOBJ)
chvl put get := Initial
chvar put get := Array(X).new(0,MAXCHV)
```

To store the default by which each newly created object is initialized we use the attribute *default*.

```
default put get := Initial
```

Also the debugging mode is determined globally.

```
debug put get := false
```

Beginning the execution, the objects *system* and *user* are inserted in the object array.

```
init
  objar[0] := O.new(cnd,Con("system"));
  objar[1] := O.new(cnd,Con("user"));
  chvar[0] := X.new(Nil);
tini
```

To access the list of objects and the list of special objects we have defined the methods *obj* and *chv*.

```
method obj(t:T):O { result objar[inlist(t,a2(objl))] } obj
method chv(t:T):X { result chvar[inlist(t,a2(chvl))] } chv
```

Fresh numbers are used to create dynamically new channels.

```
method num():T { unum := unum + 1; result Int(unum) } num
end Knot
```

A.5.2 Object and command management

When the system is started an object and command manager *supervisor* is created that starts consulting the user, by reading in a file user.pl. Everything that is read in is assumed to be part of the object *user*, except for the parts that declare named objects. In that case a new object is created. Initializing the object is postponed until the object is actually used.

```
class Supervisor
var
    current put := 1                                  the current object
    cnd put get := knot@cnd                  the current processor node
    objl put get := knot@objl
    objar put get := knot@objar
```

The attribute *current* keeps track of the current object. The attribute *cnd* indicates the processor node number at which any new object must be created. Every supervisor moreover has pointers to the information concerning objects as maintained by the global *knot*.

To get an object with name t from a supervisor m the expression $m[t]$ may be used, which expands to $m!get1(t)$. The method *get1* gets the object bypassing the method defined for the knot.

```
method get1(t:T):O { result objar[inlist(t,a2(objl))] } get1
```

To get a copy of a command manager use *get_cpy*.

```
method get_cpy():Supervisor
temp tsuper := Supervisor.new()
{
tsuper@cnd := cnd; tsuper@current := current; result tsuper
} get_cpy
```

The method *consult* accepts a term that contains a list of file names. It creates a copy of the global object *io* and delegates the reading and parsing of the files to it, file by file. When reading in the contents of a file, this information is passed to the supervisor by calling the method *add*.

```
method consult(t:T):Supervisor
{
if t ≡ nil then ;
elsif isc(t) then consult(a1(t)); consult(a2(t))
else io@cpy!consult(str(t))
fi; result self
} consult
```

Every clause or goal that is read in is inspected by the supervisor in order to check whether any special action must be taken, as for example the creation of a new object.

```
method add(t0:T): T
temp s := fof(t0), r,a:T, n:N, t := t0
{
if isfact(t) then
  if keyword(fof(headof(t))) then s := fof(headof(t)); t := headof(t); fi;
fi;
if knot@state = HALT then dbx(super,"halt")        %% do nothing
elsif t ≡ nil then perror("super add : nil")
elsif s = "at" then cnd := intof(a1(t))
elsif s = "halt" then knot@state := HALT
elsifs = "global" then knot@chvl := addchv(argsof(t),knot@chvl,knot@chvar)
elsif s = "debug" then knot@debug := true
elsif s = "nodebug" then knot@debug := false
elsif s = "default" then knot@default := argsof(t)
elsifs = "end_object" then n := current; current := 1; dbx(object,objar[n]@A)
elsif s = "object" then current := atlist(a1(t),objl);
    if objar[current] ≡ nil then objar[current] := O.new(cnd,a1(t)) fi;
elsif s = "consult" then consult(argsof(t))
else r := objar[current]!add(t)
fi; result checkans(r);
} add
```

When no special command is recognized the term sent to the supervisor is delegated to the current object. C.f. section A.4.4. The result of this may possibly contain an instruction to wait for an answer coming from an evaluation process or an instruction to unlock a particular process. The check is performed by the function *checkans* described below.

```
fn checkans(t:T):T
temp a:T
{
if t ≡ nil then result nil
elsif isc(t) then checkans(a1(t)); result checkans(a2(t))
elsif isprc(t) then a := prcof(t)@sols; dbx(answer,qstr(a)); result a;
elsif fof(t) = "unlock" then prcof(a1(t))!unlock(); result nil
else result nil
fi;
} checkans

end Supervisor
```

A.5.3 Evaluation processes

A POOL* object of class *Q* implements an evaluation process created to accompany the evaluation of a goal. Its main function is to record a trace of the derivation, and to synchronize with the process that invoked the evaluation of a goal.

When created, ap process is given a processor node and its name as an argument, as well as the object to which it refers, and the database and the options of that object.

```
class Q
newpar(nd0:N,name0:A,obj0:O,d0:D,opt0:Array(N))
var
    invoc put get := 0                          the number of copies made
    state put get := BUSY                       the state variable
    guard put get := false                      is it a guard process?
```

To handle cuts we use the attributes

```
    cut  put get := -1
    cms put get : T := Cuton(-1001,"X")
    cutl put get : T := Cut(cut,cms)
```

To keep track of the answer substitutions generated we use

```
    goal := Nil                                 to store the initial goal
    resl : T                                    to record of the last solution(s)
    sols get : T                                to keep record of all the solutions
    bagl : T                                    to store solutions for bagof
    nsol := 0                                   to count the number of answers delivered
    unlck : T                                   to store the process that must be unlocked
```

The attributes storing the options and functionality of an evaluation process are at first directly taken from the object to which the process refers, and possibly changed dynamically during the evaluation of the goal.

```
    name get put := name0
    clauses put get:T := Initial                the local clauses
    d put get := d0                             the database
    optar := Array(N).new(0,MAXOPT)             the options
    optl := Options

    pd put get := Rec(put_d(d))                 stack of databases

    mysuper put get := supervisor@cpy           a supervisor
    stp := 0                                    the number of steps
    mno := 0                                    initial cut depth

    locked := false
    me := Prc(self)
df place = name + "(" + nd0@A + ")"
```

When a process is created all options are copied from the object to which it refers.

```
init for i from 0 to MAXOPT do optar[i] := opt0[i] od; tini
```

To get the object to which the process refers use *get_obj*.

```
method get_obj():O { result obj0 } get_obj
```

To inspect and change the value of an option the array notation, as in $q[i]$ may be used.

```
method get1(i:N):N { result optar[i] } get1
method put1(i,v:N):Q { optar[i] := v; result self } put1
```

To inspect and change the value of an option from a user program, using terms, we define the methods *getopt* and *putopt*.

```
method getopt(t:T):T { result Int( optar[inlist(t,optl)] ) } getopt
method putopt(t0,v:T):Q
temp i := 0, t := t0
{
if isnil(v) | isconst("true",v) | isconst("on",v) then i := 1
elsif isconst("false",v) | isconst("off",v) then i := 0
elsif isconst("fail",v) then i := 0
elsif isint(v) then i := intof(v)
fi; if isconst("deterministic",t) then t := Con("sol"); i := 1 fi;
optar[inlist(t,optl)] := i; result self
} putopt
```

For making copies we define the methods *get_cpy* and *cpy*, the latter of which takes a processor node number as an argument. The pragma (* ALLOC AT n *) takes care of the allocation.

```
method get_cpy():Q { result cpy(nd0) } get_cpy
method cpy(n:N):Q
temp q := Q.new(n,name0,obj0,d,optar) (* ALLOC AT n *)
{
invoc := invoc + 1; q@invoc := invoc;
q@name := name0 + "-" + invoc@A;
q@clauses := cpyof(clauses);
q@cut := cut; q@cms := cpyof(cms); q@cutl := cpyof(cutl);
q@pd := pd; result q!tell(BUSY)
} cpy
```

For changing the state of a process we use the method *tell*.

```
method tell(i:N):Q { state := i; result self } tell
```

For **handling cuts** the database function resulting from compiling a clause contains a call to the function *checkcut*.

```
fn check_cut(q:Q, t:T, m,n:N):B                                  checkcut
temp b := false, k,k0:N, s:S
{
k := q@cut; k0 := intof( a1(q@cms) );
if m > k then b := false
elsif m > k0 & m ≤ k then b := true
elsif m = k0 & fof(t) = conof(a2(q@cms)) then b := true
else q!cutit(); b := check_cut(q,t,m,n);
fi;
result b
} check_cut
```

The method *cuton* is used to record a new cut.

```
method cuton(i:N,t:T):Q
{
cutl := mkc(Cut(cut,cms),cutl); cut := i; cms := t; result self
} cuton
```

The method *cutit* is used to restore the previous cut.

```
method cutit():Q
{
cut := intof(a1(a1(cutl))); cms := a2(a1(cutl)); cutl := a2(cutl); result self
} cutit
```

Context switches can be effected by using a stack of databases. See section A.7.3 for the definition of the corresponding predicates *push*() and *pop*().

```
method push():Q { pd := mkc(Rec(put_d(d)),pd); result self } push
method pop():Q { d := recof(a1(pd))@d; pd := a2(pd); result self } pop
```

Locks require the use of the methods *lock* and *unlock*.

```
method lock():Q { dbx(setlock,name); locked := true; result self } lock
method unlock():Q { dbx(unsetlock,name); locked := false; result self } unlock
```

The **interaction with the supervisor** is mediated by the method *super*. This method is used to delegate commands, such as the command to consult a file, to the command manager, unless it concerns a command to dynamically assert or retract clauses. In this case the (private) database function of the process as stored in the attribute *d* must be adapted by recompiling the local *clauses*.

```
method get_super():Supervisor { result mysuper } get_super

method super(t:T): Q
temp f: fn(T):B, r:T, n := 0
{
if fof(t) = "assert" then r := a1(t);
      if ¬ isclause(r) then r := Fact(r) fi;
      clauses := mkc(r,clauses);
      d := compose(1,compile(Con("scratch"),clauses,nilD,nilK),d0);
elsif fof(t) = "retract" then r := a1(t);
      if ¬ isclause(r) then r := Fact(r) fi;
      clauses := mkc(r,clauses);
      f := fn(t0:T):B { return(match(t0,r),false,true) };
      clauses := select(f,clauses);
      d := compose(1,compile(Con("scratch"),clauses,nilD,nilK),d0);
else
      mysuper!add(t);
fi; result self
} super
```

Returning **answer substitutions** amounts to creating a resumption that, when executed as a goal in the environment of the process that has requested an answer, effects the bindings of the variables of the initial goal in that context.

The attribute *resl* is used to store the latest answer substitution(s). When the options indicate that the evaluation is *lazy* the resumption term is of the form $goal = a_1(resl)$, otherwise it is of the form *member*(*goal*, *resl*) with the compound term *resl* converted to a Prolog list.

Dependent on the state and the value of the options EAGER and NSOL the state is reset to either BUSY, WAIT or STOP.

```
method resume():T
temp r:T
{
if unlck ≠ nil then prcof(unlck)!unlock(); unlck := nil fi;
if state = STOP then r := Mbr(goal,t_pl(sols))
elsif resl ≡ nil then r := Fail
elsif ¬ isc(resl) then r := Eq(goal,resl); resl := nil
elsif optar[EAGER] = 0 & isc(resl) then
      r := Eq(goal,a1(resl)); resl := a2(resl);
else r := Mbr(goal,t_pl(resl)); resl := nil;
fi;
if optar[EAGER] = 0 & state = PEND then state := BUSY
elsif optar[NSOL] > 0 & optar[NSOL] ≥ nsol then state := STOP
elsif state = WAIT then state := STOP
fi; result r
} resume
```

The results of executing the *bagof* goal are collected in *bagl*.

```
method bag():T { result bagl } bag
```

The protocol below determines when an answer substitution may be returned. The methods indicated by ME are the methods that may always be answered, except in the case that the process is locked. The methods collected in SAFE on the other hand may all be answered even when the process is locked. The methods collected in RESUME may only be answered when the process is willing to communicate.

```
df ME = get_obj, wr, super, get_super, put1, get1, getopt, putopt, \
        get_state, put_state, tell, lock, unlock, \
        put_name, get_name, put_invoc, get_invoc, get_guard, put_guard, \
        get_cpy, cpy,\
        push, pop, \
        get_clauses, put_clauses, get_d, put_d, put_pd, \
        put_cut, put_cms, put_cutl, \
        get_cut, get_cms, cuton, cutit, get_cutl

df SAFE = get_name, get_guard, get_obj, tell, cpy, get_cpy, \
          get_super, get1, put1, getopt, putopt

df RESUME = resume, get_sols, bag
```

Notice that when the process is pending a call to method *tr* for tracing is not accepted, thus blocking the inference until a resumption is delivered.

```
body
while state ≠ FREE do
 if locked then answer(unlock, SAFE)
 elsif state = PEND & optar[EAGER] = 0 then answer(RESUME, ME)
 elsif state = WAIT then answer( tr, RESUME, ME )
 elsif state = STOP then answer( tr, RESUME, ME )
 else answer( tr, ME )
 fi
od; io!!p(place + " exit on: " + qstr(goal) + "\n");
ydob
undf ME SAFE RESUME
```

We have built in the possibility for a process to terminate, namely when its state is FREE. However, for being able to trace the behavior of a process we just let it exist and leave it to the POOL* garbage collector to remove it, when it is no longer known to exist by any other object.

Tracing is only one of the functions of the method *tr*. In effect, we have encoded part of the functionality of an evaluation process in *tr*.

First of all, the method *tr* plays an important role in the protocol described above. Since *tr* will not be answered when the process is pending or locked, it acts as an acknowledgement to continue the inference, and blocks the inference if such an acknowledgement is not warranted.

Secondly, it is used to store the results of an inference, the regular results as well as those coming from the evaluation of a *bagof* predicate.

In the third place, the method *tr* handles the release of the process that waits for the completion of an accept statement, by checking for a single #.

And, fourthly, it mediates the communication of the results of evaluating the goal between these processes, in case of failure, by checking for a double ##.

The first argument of *tr* is a string, that functions as a tag, taking the value of either *goal*, *yes*, *call*, *exit* or *fail*. The tags *goal* and *yes* are used for respectively setting the initial *goal* and for storing the instantiated results. When the tag is *goal*, the option NSOL is set to one for ground goals. For tracing the evaluation of atoms the tags *call*, *exit* and *fail* are used. A special provision has been made for unlocking processes that wait for the deliverance of the first result. When the tag is *yes* a check is made whether such unlocking must occur.

```
method tr( fs:A, m:N, t:T ): T
temp b:B, s1:A := fs, wm:A, rt:T := t
{
if state = HALT then rt := Un("halt",t)
elsif state = WAIT | state = STOP then %% do nothing
elsif s1 = "goal" then mno := m; goal := a2(t); rt := a1(t);
  if isground(rt) | iscollect(rt) then optar[NSOL] := 1 fi;
  if isfuncar("#",2,rt) then unlck := a2(rt) fi;
elsif s1 = "call" then stp := stp + 1;
  if fof(t) = "halt" then state := HALT; knot@state := HALT fi;
  if state = WAIT | state = STOP then rt := Con("halt") fi;
  if m = 0 & isnil(goal) then goal := t fi;
  if isfc("bag",t) then bagl := append(bagl,a1(t));
```

```
    elsif isfc("ans",t) then resl := append(resl,t); rt := Nil
    fi;
  elsif s1 = "yes" then
    sols := append(sols,a2(t)); nsol := nsol + 1;
    resl := append(resl,a2(t)); state := PEND; rt := a1(t);
    if unlck ≠ nil then prcof(unlck)!unlock(); unlck := nil fi;
    if optar[NSOL] = nsol then state := WAIT fi;
  elsif s1 = "fail" then
    if mno = m then state := WAIT fi;
  fi;
```

We need a hack to cope with failure in case of a conditional accept statement.

```
if s1 = "fail" & fof(t) = "##" then chvof(a1(t))@val := a2(t) fi;
```

The option TRACE determines the extent in which tracing information is given to the user.

```
case optar[TRACE] of
0 then b := s1 = "goal" | s1 = "yes" or
1 then b := ¬ syspred(fof(t)) & tr_port(s1) or
2 then b := ¬ syspred(fof(t)) & db_port(s1)
else b := true
esac;
if b then io!!p( place + " [" + stp@A + ":" + m@A + "]" + state@A
          + "*" + cut@A + "* " + s1 + ": " + str(rt) );
fi; result rt;
} tr
```

For writing terms the (asynchronous) method *wr* is used.

```
method wr( t:T ) { io!!p(place + " > " + str(t)); } wr
end Q
```

A.6 Non-logical variables and channels

```
impl unit X                                          file: X.i    1.9
use IO Q O D P U T, Nodes
include N U T
```

Objects of type X are used both to store (persistent) values, and to effect a synchronous communication between processes.

```
class X
newpar(name:T)
var
    val get : T
    channel put get := false
    confirmed := false
    free := true
```

```
store := Nil
message := Nil
me := Chv(self)
```

A.6.1 Non-logical variables

Non-logical variables may be initialized.

```
init if isfuncar(" = ",2,name) then val := a2(name) else val := Nil fi; tini
```

When assigning a value to a non-logical variable, we take some precautions when the name of the variable is nil, since these variables are used to communicate a substitution between the process evaluating a method call and the process in which the (conditional) accept statement occurred by which the method call got accepted. These variables may be assigned a value only once. Moreover, to avoid variable clashes between processes, all (unbound) logical variables are mapped to the anonymous logical variable, with number zero.

```
method put_val(t:T):X
temp e : fn(V):V
{
if name ≡ nil then
        if isnil(val) then
                e := fn(v:V):V { result 0 };
                val := mape(e,t);
        fi
else val := t
fi; result self
} put_val
```

For adding a list of non-logical variables t to a symbol-table pair (l, ar) the function *add_chv* is used. C.f. section A.4.1.

```
fn add_chv(t,l:T,ar:Array(X)):T
temp n:N
{
if t ≡ nil then ;
elsif isc(t) then add_chv(a1(t),l,ar); add_chv(a2(t),l,ar);
else n := atlist(nameof(t),l); if ar[n] ≡ nil then ar[n] := X.new(t) fi;
fi; result l
} add_chv
```

The name of a non-logical variable, possibly containing an initialization, is delivered by the method *get_name*.

```
method get_name():T { result name } get_name
```

A.6.2 Channels

Non-logical variables can be interpreted as channels that allow **synchronous communication**. See also the discussion in section 11.6.2.

The **protocol** governing synchronous communication over channels allows the input side to backtrack until a unification between the input term and the output term is possible.

When involved in a communication, only a limited number of methods may be answered. Part of these are collected in the macro ME.

```
df ME = get_name, get_val, put_val, get_channel, put_channel
body do
if ¬ confirmed & ¬ free then answer(ME, read, confirm, force )
elsif ¬ free & confirmed then answer(ME, receive);
else
      answer any
fi
od ydob
```

To engage in a communication the input side must call the method *read*.

```
method read(t:T):T
{
store := t;
while isnil(message) do dbx(read,qstr(t)); answer(ME,write) od;
result message
} read
```

The output side is supposed to call the method *write*.

```
method write(t:T):T
{
while ¬ isnil(message) do dbx(write,qstr(t)); answer(ME,read) od;
message := t; free := false; confirmed := false;
result nil
} write
```

For confirming the possibility of a unification and to collect the input term, the methods *confirm* and *receive* must be used.

```
method confirm():X { confirmed := true; result self } confirm
method receive():T {
free := true; confirmed := false; message := Nil; result store
} receive
```

The method defined below forces the output side to backtrack or to continue the computation, dependent on the value of the argument.

```
method force(t:T):X { store := t; confirmed := true; result self } force
end X
```

A.7 The initial database

For dealing with the special forms, that capture the functionality by which DLP extends Prolog, we need some simple term rewriting functions that convert symbolic terms to the entities to which these terms refer; moreover we need a function that embodies the special interpretation given to equality and assignment; and in addition we need a system database that evaluates the primitives involved in communication. Apart from the DLP specific functionality we have included arithmetic simplification and some primitives for compatibility with Prolog.

```
impl unit D                                              file: D.i    1.3
use Q O P T U Nodes
include N U T
class System
```

A.7.1 Simplification

The simplification function, defined below, is called for instance in the function *eval*, defined in section A.3.1, to simplify atoms. Simplifying a term means replacing that term by another term or itself, possibly with side-effects, such as the creation of an object or a process. As an example, the function *simplify* is used to replace non-logical variables by their values, if appropriate.

A call to the simplification function must have a parameter q referring to an evaluation process to have access to the object for which the inference takes place.

The integer k is used to indicate whether the left hand side of an equality is evaluated ($k = 1$), the right hand side ($k = 2$), or an ordinary expression ($k = 0$).

The parameter *nvars*, indicating the number of variables in use, is passed to the method *request* when creating a new active object.

```
fn simpl(q:Q,k:N,nvars:N,t0:T): T                                    simplify
temp n:N, r,t1,t2:T, o:O, z:A, t := t0
{
```

We first check whether t is a non-logical variable. If the process q is evaluating the guard of a conditional accept expression, then the non-logical value of the object may not be accessed, since otherwise a deadlock would occur.

```
if t ≡ nil then %% do nothing
elsifiscon(t) & ¬ isnil(t) & ¬ isint(t) & ¬ q@guard thenr := q@obj@chvl;
      if member(t,r) then
            if k = 1 then
                  t := Chv(q@obj!chv(t))
            else
                  t := q@obj!chv(t)@val
            fi;
      fi;
fi;
```

Then we continue with the simplification. Recall that only constants, logical variables and function terms are considered *symbolic*. C.f. section A.2.1.

```
if t ≡ nil then result nil
elsif ischv(t) then
      if chvof(t)@channel then result t else return(k = 1,t,chvof(t)@val) fi
elsif ¬ symbolic(t) then result t
```

The following constants may be used for allocating objects and processes. The expression my_node stands for the processor node of the object calling *simplify*. See further section A.8.2 for the definitions of the functions used.

```
elsif isconst("here",t) then result Int(my_node)
elsif isconst("branch",t) then result Int(U.branch(q))
elsif isconst("width",t) then result Int(U.width(q))
elsif isconst("nproc",t) then result Int(U.nproc(q))
```

The constant *scratch* refers to the local clauses of an evaluation process, to be modified by the use of the predicates *assert* and *retract*.

```
elsif isconst("scratch",t) then result t_pl(q@clauses)
```

If the term to be simplified is a function term, its arguments are simplified first except for certain special function symbols.

```
elsif isfuncar("quote",2,t) then result a1(t)
elsif ¬ isfun(t) then result t
elsif isfun(t) then z := fof(t); n := arity(t);
 if z ≠ "##" & z ≠ "@" & z ≠ "." & z ≠ "val" & z ≠ "global"
   then
 a1(t) := simpl(q,k,nvars,a1(t)); a2(t) := simpl(q,k,nvars,a2(t));
 fi;
 t1 := a1(t); t2 := a2(t);
 if z = "##" then t2 := simpl(q,k,nvars,t2) fi;
```

A *node expression* is converted to an integer. See section A.8.2 for the definition of *node*.

```
if z = "node" then result Int(node(q,a1(t)))
```

The tag of a term can be asked for.

```
elsif z = "tagof" then result Int(tagof(t1))
```

A term can be converted to an object, or interpreted as a local or global non-logical variable, or an option. These expressions may be used to avoid name clashes between objects, non-logical variables and options.

```
elsif z = "obj" then return(isconst("self",t1), Obj(q@obj), Obj(q@super[t1]))
elsif z = "val" then
    return(k = 1,Chv(q@obj!chv(t1)), simpl(q,k,nvars,q@obj!chv(t1)@val))
elsif z = "global" then
     return(k = 1,Chv(knot!chv(t1)), simpl(q,k,nvars,knot!chv(t1)@val))
elsif z = "option" then return(k = 1,t, q!getopt(t1) )
```

An *@-symbol* is meaningful only when its first argument is an object or a process. It is used to access the value of the non-logical variable of an object, or to inspect the value of an option for a particular process.

```
elsif z = "@" & k ≠ 1 then t1 := simpl(q,k,nvars,t1);
    if iscon(t1) then t1 := Obj(q@super[t1]) fi;
    if isobj(t1) then result simpl(q,k,nvars,objof(t1)!chv(t2)@val)
    elsif isprc(t1) then result prcof(t1)!getopt(t2)
    else result t
    fi;
```

A *new*-expression results in either a new object, a new process or a new channel. For an object also a new constructor process may be started. C.f. section 11.7.1.

```
elsif z = "new" then n := my_node;                                  new(t)
    if isfun(t1) & fof(t1) = "@" then
        n := node(q,a2(t1)); t1 := a1(t1);
    fi;
    if isconst("this",t1) then result Prc(q!cpy(n))
    elsif isconst("self",t1) then result Obj(q@obj!cpy(n))
    elsif fof(t1) = "channel" then
        if iscon(t1) then t2 := knot!num() fi;
        result Chv(X.new(t2)!put_channel(true))
    elsif isfun(t1) then
        o := q@super[Con(fof(t1))]@ini!cpy(n);
        o@acceptlist := Con(fof(t1)); o@active := true;
        r := o!request(nil,0,nvars,t1);
        o@constructor := prcof(r); result Obj(o)
    elsif isobj(t1) then result Obj(objof(t1)!cpy(n))
    elsif ischv(t1) then result Chv(X.new(chvof(t1)@name))
    elsif isprc(t1) then result Prc(prcof(t1)!cpy(n))
    elsif symbolic(t1) then result Obj(q@super[t1]@ini!cpy(n))
     else result t
    fi
```

A straightforward interpretation of (syntactic) equality and inequality is given.

```
elsif z = " ≡ " then return(eq_t(t1,t2),Nil,Fail)
elsif z = " ≠ " then return(eq_t(t1,t2),Fail,Nil)
```

We have defined some macros to deal respectively with arithmetical simplifications and the evaluation of comparison operators. When testing whether t was a function symbol, we have put the arity of t in n.

```
df a_simpl(O,T1,T2) = if n = 2 then ar_simpl(O,T1,T2) else result t fi;
df b_simpl(O,T1,T2) = if n = 2 then bl_simpl(O,T1,T2) else result t fi;

df ar_simpl(O,T1,T2) = if isint(T1) & isint(T2) \
    then result Int( intof(T1) O intof(T2) ) \
    else result Op("O",T1,T2) fi;
df bl_simpl(O,T1,T2) = if isint(T1) & isint(T2) \
    then if intof(T1) O intof(T2) then result Nil \
    else result Fail fi else result Op("O",T1,T2) fi;
```

Finally, we provide an interpretation for the arithmetic and comparison operators.

```
elsif z = " + "  then a_simpl( + ,t1,t2)                    arithmetic
elsif z = "-"  then a_simpl(-,t1,t2)
elsif z = "/"  then a_simpl(/,t1,t2)
elsif z = "*"  then a_simpl(*,t1,t2)
elsif z = "**" then a_simpl(**,t1,t2)
elsif z = "//" then a_simpl(//,t1,t2)
elsif z = " < "  then b_simpl( < ,t1,t2)
elsif z = " ≤ " then b_simpl( ≤ ,t1,t2)
elsif z = " > "  then b_simpl( > ,t1,t2)
elsif z = " ≥ " then b_simpl( ≥ ,t1,t2)
else result t
fi
else result t
fi
} simpl
```

A.7.2 Equality and assignment

The function *assign*, implemented by the function below, takes the same parameters as an ordinary database function. It is called by the *system* database to evaluate the binary 'predicates' for equality and assignment. C.f. section A.7.3. See also section 11.7.2.

```
fn eq_assign(q:Q, t:T, c:C, m,n:N, s:S) : T                 assign
temp k1 := 1,k2 := 2, t1,t2,ot:T,
     q1:Q, nd := my_node, z := fof(t)
{
```

We first simplify both the left hand side and right hand side of the equality and check whether both parts are syntactically identical, after simplification.

```
if z = " = " then k1 := 2 elsif z = "::" then k2 := 2 fi;
t1 := simplify(q,k1,n,a1(t)); t2 := simplify(q,k2,n,a2(t));
if eq_t(t1,t2) then result c(m,n,s)
```

When the (simplified) left hand side is not a variable, we interpret the equality statement as assigning a value to a non-logical variable or to an option of a process. Local options can also be given a value, directly. When no such form can be recognized both terms are simply unified.

```
elsif ¬ isvar(t1) & k1 = 1 then
  if isfuncar("@",2,t1) then
    if iscon(a1(t1)) then a1(t1) := Obj(q@super[a1(t1)]) fi;
    if isobj(a1(t1)) then objof(a1(t1))!chv(a2(t1))@val := t2; result c(m,n,s)
    elsif isprc(a1(t1)) then prcof(a1(t1))!putopt(a2(t1),t2); result c(m,n,s)
    else result unify(t1,t2,c,m,n,s)
    fi
  elsif ischv(t1) & ischv(t2) then
```

```
        chvof(t1)@val := chvof(t2)@val; result c(m,n,s)
    elsif ischv(t1) then chvof(t1)@val := t2; result c(m,n,s)
    elsif iscon(t1) then
        if member(t1,options) then q!putopt(t1,t2); result c(m,n,s)
        else result unify(t1,t2,c,m,n,s)
        fi
    elsif isfun(t1) & fof(t1) = "option" then q!putopt(t1,t2); result c(m,n,s)
    else result unify(t1,t2,c,m,n,s)
    fi
```

A new evaluation process is started when an expression of the form $Q = O!G$ is encountered. In case O refers to an evaluation process, a copy of the process is made to evaluate the goal G. Otherwise O is supposed to refer to an object, which is asked for a rendez-vous.

```
elsif isfuncar("!",2,t2) then ot := a1(t2); nd := my_node;        Q = O!G
  if symbolic(ot) then ot := Obj(q@super[ot]) fi;
  if isobj(ot) then
      q1 := objof(ot)!process(nd);
      if q[COPY] > 0 then for i from 1 to MAXOPT do q1[i] := q[i] od fi;
      objof(ot)!request(q1,m,n,a2(t2));
      result unify(t1,Prc(q1),c,m,n,s)
  elsif isprc(ot) then q1 := prcof(ot)!cpy(nd);
      infer(q1,a2(t2),yes(q1,m,a2(t2)),m,n,s);
      result unify(t1,Prc(q1),c,m,n,s)
  else perror("eq" + qstr(t)); result Error;
  fi
else
  result unify(t1,t2,c,m,n,s)
fi;
} eq_assign
```

A.7.3 The system database

We have defined a number of primitive predicates in the database function *system*. This function is put in composition with the database of each object, as described in section A.4.4.

```
fn start_d( q:Q, t:T, c:C, m,n:N, s:S) : T                        system
temp z := fof(t), t1,t2,g,r,mt:T,
      qa:Q,n1:N, e:E, b := false, f:fn(T):B
{
q!tr("D:system",m,t);
case tagof(t) of
CON then
  if z = "nl" then q!!wr(Con("\n"));result c(m,n,s)
  elsif z = "!" then q!cuton(m,Cuton(0,"X")); result c(m,n,s)
  elsif z = "push" then q!push(); result c(m,n,s)
  elsif z = "pop" then q!pop(); result c(m,n,s)
```

```
    else result nil
    fi; or
  VAR then result eval(q,maps(s,t),c,m,n,s) or
  FUN then t1 := a1(t); t2 := a2(t);
```

Some familiar **system predicates** are defined below.

```
if z = " :- " | z = "consult" | z = "assert" | z = "retract" then
                q!super(t); result c(m,n,s)
elsif z = "write" then q!!wr(argsof(t));result c(m,n,s)
elsif z = "unify" then result unify(t1,t2,c,m,n,s)
elsif z = "\+" then qa := q@cpy; r := q@d(qa,argsof(t),yes(qa,m,t1),m,n,s);
          if r ≡ nil then result c(m,n,s) else result nil fi;
elsif z = " = " | z = " := " | z = "::" then result assign(q,t,c,m,n,s)
elsif z = "," then result eval(q,t,c,m,n,s)
```

For our interpretation of the cut, to check whether a goal is acceptable, to evaluate an accept statement and to force an output process waiting for communication over a channel, we have defined the following predicates.

```
elsif z = "new" then result eval(q,Eq(Var(0),t),c,m,n,s)
elsif z = "acceptable" then
    return(isacceptable(t1,q@obj@acceptlist),c(m,n,s),nil)
elsif z = "accept" then r := q@obj!accept(q,n,argsof(t));
          if r ≡ nil then r := Nil else r := Un("?",r) fi;
          q!tr("accepted",m,t);
         mt := q@obj@post; n1 := intof(a1(mt)); r := mkc(r,a2(mt));
          result eval(q,r,c,m,n1,s)
elsif z = "cuton"  then q!cuton(m,t); result c(m,n,s)
elsif z = "free" | z = "fail" then
   if ischv(t1) then
     if z = "free" then r := Var(0) else r := Error fi;
     chvof(t1)!force(r); result c(m,n,s)
   else result nil fi
elsif z = "##" then result c(m,n,s) %% see section A.4.2
```

Additionally, we define the *univ* predicate for compatibility with Prolog.

```
elsif z = " =.." then
      if ¬ isvar(t1) then t1 := Con(fc(t1));
          r := Op(".",t1,t_pl(argsof(arg(1,t))));
         result unify(r,arg(2,t),c,m,n,s)
      else t1 := maps(s,arg(2,t));
         if isfc(".",t1) then
           r := Un(conof(arg(1,t1)),pl_t(arg(2,t1)));
            result unify(arg(1,t),r,c,m,n,s)
         else result nil
         fi
      fi;
```

Synchronous communication is dealt with by the following definitions.

```
elsif z = "!" then
        if ischv(t1) then                                              C!t
            chvof(t1)!write(t2); r := chvof(t1)!receive();
            n1 := maxvar(r); e := fn(nv:V):V { result n + nv };
            r := mape(e,r); result unify(t2,r,c,m,n + n1,s);
        elsif isconst("self",t1) then result eval(q,t2,c,m,n,s)
        else                                                           O!G
            r := Var(n + 1);
            result eval(q,mkc(Eq(r,t),Un("?",r)),c,m,n + 1,s);
        fi
elsif z = "?" then
        if isobj(t1) then t1 := Prc(objof(t1)@constructor); fi;
        if isnil(t1) then result c(m,n,s)
        elsif isprc(t1) then r := prcof(t1)!resume();                  Q?
            if t2 ≠ nil then r := Fun("unify",t2,r) fi;
            if prcof(t1)@state ≠ BUSY then r := r
            else r := Fun(";",r,t)
            fi; result eval(q,r,c,m,n,s)
        elsif ischv(t1) then mt := t2; g := Nil;                       C?t
            if isfuncar(":",2,mt) then
                g := a2(mt); mt := a1(mt)
            fi;
            r := chvof(t1)!read(mt);
            n1 := maxvar(r); e := fn(nv:V):V { result n + nv };
            r := mkc(Fun("unify",mt,mape(e,r)),
                     mkc(g,Un("confirm",t1)));
            result eval(q,r,c,m,n + n1,s);
        else result nil
        fi
```

When a successful communication (over a channel) is possible, the goal *confirm* is called from the input side.

```
elsif z = "confirm" then chvof(a1(t))!confirm(); result c(m,n,s)
```

Finally, we define the interpretation of the predicates *call* and *bagof*.

```
elsif z = "call" then result eval(q,argsof(t),c,m,n,s)
elsif z = "bagof" then result bagof(q,t,c,m,n,s)
else result nil
fi;
else result nil
esac;
} start_d
```

To evaluate the **bagof** predicate we need the function *bagof*.

```
fn bag_of( q:Q, t:T, c:C, m,n:N, s:S) : T                              bagof
temp qa:Q, g,r:T, a:T
{
if arity(t) ≠ 3 then perror(qstr(t)); result Error
else
  qa := q@cpy; g := Un("bag",arg(1,t)); g := mkc(arg(2,t),g);
```

```
    infer(qa,g,yes(qa,m,g),m,n,s); r := t_pl( qa!bag() );
    dbx(bag,qstr(Prc(qa)) + "::" + qstr(r));
    result unify(r,arg(3,t),c,m,n,s);
  fi
  } bag_of
  end System
```

A.7.4 Booting

A number of additional primitives are defined in the DLP object *boot*.

```
object boot {

true.

not(A) :- A,!,fail.
not(A).

A;_ :- A.
_;B :- B.

A → B :- A, !, B.

A&B :- Q = new(self)!A, B, Q?.

G@N :- O = new(self@N), O!G.

member(X,[ X | _ ]).
member(X,[ _ | T ]) :- member(X,T).

append([],L,L).
append([H|T],L,[H|R]) :- append(T,L,R).
}
```

A.8 Utilities

A.8.1 Auxiliary definitions

file: U.h 1.3

We need to know the maximum number of declared DLP objects the system can take, as well as the maximum number of non-logical variables each object can have.

```
df MAXOBJ = 24
df MAXCHV = 12
```

We define some system constants to be used as defaults in the allocation functions.

```
df P_NPROC = 10
df P_BRANCH = 2
df P_WIDTH = 4
```

To handle the **options** we give the definitions below.

```
df MAXOPT = 13
df Options = mkc(Con("trace"), mkc(Con("nosimplify"), \
        mkc(Con("eager"), mkc(Con("sol"), \
        mkc(Con("override"), mkc(Con("copy"), mkc(Con("branch"), \
        mkc(Con("width"), mkc(Con("nproc"), mkc(Con("aux1"), \
        mkc(Con("aux2"), Con("aux3") ))))))))))
df TRACE = 1
df NOSIMPLIFY = 2
df EAGER = 3
df NSOL = 4
df OVERRIDE = 5
df COPY = 6
df BRANCH = 7
df WIDTH = 8
df NPROC = 9
df AUX1 = 10
df AUX2 = 11
df AUX3 = 12
```

To determine of a method call if it is acceptable we define the macro *acceptable*.

```
df isacceptable(T,L) ( isconst("any",L) | member(Con(fof(T)),L) )
```

In order to be able to check for keywords and special names or operators in the DLP system the following definitions are provided.

```
df keyword(S) = ( S = "object" | S = "global" | S = "channel" | \
    S = "use" | S = "isa" | S = "inherit" | S = "default" | \
    S = "deterministic" | \
    S = "var" | S = "private" | S = "set" | S = "clear" | S = "accept")
df isnode(T) = (isint(T) | isconst("here",T) | isunit(T) | \
    isnil(T) | fof(T) = ":" | fof(T) = "." )
df isctr(T) = (fof(T) = "," | fof(T) = " → " | fof(T) = ";")
df simplifiable(S) = (S ≠ " = " & S ≠ " := " & S ≠ "::" & \
    S ≠ "accept")
df opsym(S) = (S = "," | S = "." | S = ";" | S = " → " | S = " :- " | \
    S = "::" | S = " := " | S = " = " | S = " + " | S = "-" | S = "*" | \
    S = "/" | S = "//" | S = "**" | S = "@" | S = "!" | S = "?" | \
    S = "#" | S = ":" | S = "&" | S = " | " | S = " ⇒ " | S = " < " | \
    S = " > " | S = " ≤ " | S = " ≥ " | S = " ≡ " | S = " ≠ ")
df syspred(S) = (S = "nl" | S = "write" | S = "[]")
```

We use some macros in determining the level of tracing.

```
df tr_port(S) = (S = "accepted" | S = "goal" | S = "call" | S = "exit" | \
    S = "fail" | S = "yes" | S = "cut*")
df db_port(S) = ( tr_port(S) | S[1]@N = 'D'@N )
```

A.8.2 Term manipulation functions

impl unit U *file: U.i 1.3*

use IO Q T Text Nodes
include N U T

class U

A term is converted to a string by means of one of the functions below.

```
df funstr(T,X) = fc(T) + "(" + t_str(a1(t)) + X + t_str(a2(t)) + ")"
df opstr(T) = "(" + str(a1(T)) + fc(T) + str(a2(T)) + ")"
df clstr(T) = " " + opstr(T) + "\n"
df objstr(T) = "*" + str(objof(T)@name) + "*"
df chvstr(T) = " + " + str(chvof(T)@name) + " + "
df prcstr(T) = "*" + prcof(T)@name + "*"

fn t_str(t:T):A                                                    str
{
if t ≡ nil then result ""
elsif isfuncar(" :- ",2,t) then result clstr(t)
elsif isc(t) then return(isun(t),t_str(a1(t)),opstr(t))
elsif opsym(fof(t)) then return(isfun(t),opstr(t),fof(t))
else
  case tagof(t) of
  CON then result conof(t) or
  VAR then return(isnull(t),"_","_" + varof(t)@A) or
  FUN then return(isun(t),funstr(t,""),funstr(t,",")) or
  OBJ then result objstr(t) or
  PRC then result prcstr(t) or
  CHV then result chvstr(t)
  else result "*" + tagof(t)@A + "*"
  esac;
fi
} t_str
```

The **map** functions for respectively environments, substitutions and for cuts are given below.

```
fn m_e(e:E,t:T):T                                                  mape
{
if t ≡ nil then result nil
elsif isvar(t) then return(isnull(t),t,Var(e(varof(t))))
elsif isfun(t) then result Fun(fc(t),m_e(e,a1(t)),m_e(e,a2(t)))
else result t fi
} m_e

fn m_s(s:S,t:T):T                                                  maps
{
if t ≡ nil then result nil
elsif isvar(t) then return(unbound(varof(t),s),t,m_s(s,s(varof(t))))
elsif isfun(t) then result Fun(fof(t),m_s(s,a1(t)),m_s(s,a2(t)))
else result t fi
} m_s
```

```
fn m_c(m:N,s:A,t:T):T                                          mapc
{
if t ≡ nil then result t
elsif isctr(t) then result Op(fof(t),m_c(m,s,a1(t)),m_c(m,s,a2(t)))
else return(t_eq(Con("!"),t),Cuton(m,s),t)
fi
} m_c
```

For allocating objects and processes, using node expressions, we define the function *nd* and a number of auxiliary functions.

The functions *branch*, *width* and *nproc* assign default values to the allocation parameters.

```
fn branch(q:Q):N { return(q[BRANCH] = 0,P_BRANCH,q[BRANCH]) }
   branch
fn width(q:Q):N { return(q[WIDTH] = 0,P_WIDTH,q[WIDTH]) } width
fn nproc(q:Q):N { return(q[NPROC] = 0,P_NPROC,q[NPROC]) } nproc
```

The functions *guess* and *ndsel* are used to implement, in a provisional way, the node expressions () and $\{N_1, ..., N_n\}$, denoting respectively an arbitrary node and any node from the set containing $N_1, ..., N_n$.

```
fn guess(q:Q):N { result my_node } guess
fn ndsel(q:Q,t:T):N { result nd(q,0,a1(t)) } ndsel
```

The functions *level* and *dim* implement functions used in the function *node*.

```
fn level(b,d:N):N { return(d ≤ 0,0,b**d + level(b,d-1)) } level
fn dim(s:A,t:T):N { returnil(t,isfuncar(s,2,t),0,dim(s,a1(t)) + dim(s,a2(t)),1)
   } dim
```

We leave the details of the function *node* to the reader.

```
fn nd(q:Q,n:N,t:T):N                                           node
temp k,m,r:N, np := nproc(q), pn,ln,nn,d,ms:N
{
if t ≡ nil then result 0
elsif isnil(t) then result 0
elsif isconst("{}",t) then result my_node
elsif isfun(t) & fof(t) = "{}" then result ndsel(q,argsof(t))
elsif isfuncar("#",2,t) then k := width(q)**(dim(fof(t),t)-1);
 m := nd(q,n,a1(t)) ; r := nd(q,n,a2(t)) ; result (k*m + r)//np
elsif isfuncar(":",2,t) then k := branch(q); r := nd(q,0,a2(t));
 if ¬ (n = 0) then d := n else d := dim(fof(t),t)-1 fi;
 m := nd(q,d-1,a1(t)); pn := level(k,d-2); ln := pn + (k**(d-1));
 if d = 1 then
      ln := 0; nn := 0
 else
      nn := m-1-pn
 fi; result (ln + nn*k + r)//np
elsif isfuncar(".",2,t) then k := branch(q); m := nd(q,0,a1(t));
 r := nd(q,n + 1,a2(t)); return(n = 0,(m + r)//np,(k**n)**m + r)
elsif isconst("here",t) then result my_node
```

```
elsif isunit(t) then result guess(q)
else result int_of(t)//np
fi
} nd
```

To support the tests, and selectors defined for terms we define the functions below.

```
fn is_con(s:A,t:T):B { return(iscon(t),conof(t) = s,false) } is_con        iscon
fn is_fun(s:A,t:T):B { return(isfun(t),fc(t) = s,false) } is_fun           isfun
fn is_var(n:V,t:T):B { return(isvar(t),varof(t) = n,false) } is_var        isvar
fn is_clause(t:T):B { return(isfun(t),fof(t) = " :- ",false) } is_clause   isclause
fn is_fact(t:T):B { return(is_clause(t),isnil(a2(t)),false) } is_fact      isfact
fn head_of(t:T):T { return(is_clause(t),a1(t),Error) } head_of             headof

fn is_funcar(s:A,n:N,t:T):B                                               isfuncar
{
if t ≡ nil then result false
elsif is_con(s,t) then result n ≡ 0
elsif is_fun(s,t) then result n ≡ arity(t)
else result false fi
} is_funcar

fn func_of(t:T):A                                                         fof
{
if t ≡ nil then result "error"
elsif iscon(t) then result conof(t)
elsif isfun(t) then result fc(t)
else result "error" fi
} func_of
```

For extracting the arguments, selecting an argument and for determining the arity of a function term we define the following functions.

```
fn args_of(t:T):T                                                         argsof
{
if t ≡ nil then result nil
elsif isfun(t) then return(a2(t) ≡ nil,a1(t),mkc(a1(t),a2(t)))
else result t
fi
} args_of

fn get_n(n:N,t:T):T                                                       arg
{
if n = 1 then return(isc(t), a1(t), t) else result get_n(n-1,a2(t)) fi
} get_n

fn length(t:T):N { returnil(t,isc(t),0,1 + length(a2(t)),1) } length      arity
```

We may check whether a term represents an integer, and also convert terms to integers.

```
df isdigit(c) = ( c ≥ '0') & (c ≤ '9')
fn is_int(t:T):B {return(iscon(t),isdigit(conof(t)[1]),false)} is_int     isint
fn int_of(t:T):N { return(isint(t),conof(t)@N@1,my_node) } int_of         intof
```

For extracting the list of variables occurring in a term we use the function *varsin*.

```
fn v_in(t:T):T                                                    varsin
temp l := mkc(Var(0),nil), v := v_of(t,l), r:T := a2(v)
{ return(r ≡ nil,Con("_"),r) } v_in

fn v_of(t,l:T):T
{
if t ≡ nil then result l
elsif isvar(t) then atlist(t,l); result l
elsif isfun(t) then v_of(a1(t),l); v_of(a2(t),l); result l
else result l
fi
} v_of
```

We may also have to determine the highest variable number occurring in term, or whether a term is ground.

```
fn max(i,j:N):N { return(i > j,i,j) } max
fn and(a,b:B):B { result a & b } and

fn max_v(t:T):N                                                   maxvar
{
if t ≡ nil then result 0
elsif isvar(t) then result varof(t)
elsif isfun(t) then result max(max_v(a1(t)),max_v(a2(t)))
else result 0 fi
} max_v

fn is_ground(t:T):B                                               isground
{
if t ≡ nil then result true
elsif isvar(t) then result false
elsif isfun(t) then result is_ground(a1(t)) & is_ground(a2(t))
else result true fi
} is_ground
```

For testing the (syntactic) equality of terms, or to determine whether they match, we need the following functions.

```
fn t_eq(t0,t:T):B                                                 equal
{
if t0 ≡ nil | t ≡ nil then result t0 ≡ t
elsif tagof(t0) ≠ tagof(t) then result false
else case tagof(t) of
    CON then result conof(t0) = conof(t) or
    VAR then result varof(t0) = varof(t) or
    FUN then
      return(fc(t0) = fc(t),t_eq(a1(t0),a1(t)) & t_eq(a2(t0),a2(t)),false)
    else result t0 ≡ t
    esac
fi
} t_eq

fn t_mt(t0,t:T):B                                                 match
{
if t0 ≡ nil | t ≡ nil then result t0 ≡ t
```

```
elsif isvar(t0) | isvar(t) then result true
elsif tagof(t) ≠ tagof(t0) then result false
else case tagof(t) of
    CON then result conof(t0) = conof(t) or
    FUN then
      return(fc(t0) = fc(t),t_mt(a1(t0),a1(t)) & t_mt(a2(t0),a2(t)),false)
    else result t0 ≡ t
    esac
fi
} t_mt
```

We have defined two membership functions. Both deliver the position where the term that is tested for membership occurs in the list. The result is zero if the term does not occur in the list.

Both functions are parametrized with a function of type $fn\ (T,T) : B$ that is used to perform the test whether the term equals or is similar to an element in the list. We remark that we have defined member(T,L) as (U.t_in(eq_t,T,L) > 0).

```
fn t_in(f:fn(T,T):B,t,l:T):N                                    member, inlist
temp n:N
{
if l ≡ nil then result 0
elsif isc(l) then if f(t,a1(l)) then result 1
                  else n := t_in(f,t,a2(l)); return(n > 0,n + 1,0);
                  fi
elsif f(t,l) then result 1
else result 0
fi
} t_in
```

The function *atlist* differs from the previous one in that when a similar term does not occur in the list, it is added as a last element.

```
fn t_at(c:fn(T,T):B,t0,t:T):N                                   atlist
{
if t ≡ nil then perror("t_at0:" + qstr(t0)); result -1001
elsif isc(t) then
  if c(t0,a1(t)) then result 0
  elsif a2(t) ≡ nil then a2(t) := mkc(t0,nil); result 1
  elsif ¬ isc(a2(t)) then
      if c(a2(t),t0) then result 1
      else
            a2(t) := mkc(a2(t),t0); result 2
      fi;
  else result 1 + t_at(c,t0,a2(t))
  fi
else perror("t_at:" + qstr(t0)); result -1001
fi
} t_at
```

We also need a function that appends two terms.

```
fn t_app(l,t:T):T                                               append
```

```
{
if l ≡ nil then result t
elsif t ≡ nil then result l
elsif isc(l) then
        if a2(l) ≡ nil then a2(l) := t
        elsif ¬ isc(a2(l)) then a2(l) := mkc(a2(l),t)
        else t_app(a2(l),t)
        fi; result l
else result mkc(l,t)
fi;
} t_app
```

We may have to copy a term.

```
fn t_cpy(t:T):T                                                    cpyof
{
if t ≡ nil then result nil
else case tagof(t) of
CON then result Con(conof(t)) or
VAR then result Var(varof(t)) or
FUN then result Fun(fc(t),t_cpy(a1(t)),t_cpy(a2(t))) or
OBJ then result Obj(objof(t)) or
PRC then result Prc(prcof(t)) or
CHV then result Chv(chvof(t))
else perror("t_cpy:" + str(t)); result t
esac; fi;
} t_cpy
```

For converting (compound) terms to Prolog lists and vice versa the functions *t_pl* and *pl_t* are used.

```
fn t__pl(t:T):T                                                    convert
{
returnil(t,isc(t),Nil,Dot(a1(t),t__pl(a2(t))),Dot(t,Nil))
} t__pl

fn pl__t(t:T):T
{
if t ≡ nil then result nil
elsif isconst("[]",t) then result nil
elsif isfc(".",t) then return(isnil(a2(t)),a1(t),mkc(a1(t),pl_t(a2(t))))
else result t fi
} pl__t
```

To extract the name of a non-logical variable, possibly containing an initialization we use the *nameof* function.

```
fn g_n(t:T):T                                                      nameof
temp s := fof(t)
{
return(s = ":" | s = "::" | s = " = ",g_n(a1(t)),t)
} g_n
```

To recognize the clauses that must be retracted the function *select* is used.

```
fn sel(c:fn(T):B,l:T):T                                          select
{
if l ≡ nil then result Nil
elsif ¬ isc(l) then return(c(l),l,Nil)
else return(c(a1(l)),mkc(a1(l),sel(c,a2(l))),sel(c,a2(l)))
fi
} sel
end U
```

Bibliography

[Aikins, 1980] J.S. AIKINS, *Prototypes and Production Rules: A knowledge representation for consultations,* Report STAN-CS-80-814 (1980) Stanford

[Ait-Kaci and Nasr, 1986] H. AIT-KACI AND R. NASR, *LOGIN: A logic programming language with built-in inheritance,* Journal of Logic Programming, 3 (1986) pp. 185-215

[Agha, 1986] G. AGHA, *Actors: A model of Concurrent Computation in Distributed Systems,* (MIT Press, 1986)

[Akama, 1986] K. AKAMA, *Inheritance hierarchy mechanism in Prolog,* LNCS 264 (1986) pp. 12-21

[Allison, 1986] L. ALLISON, *A practical introduction to denotational semantics,* Cambridge Computer Science Texts 23, 1986

[America, 1987] P. AMERICA, *POOL-T: a parallel object oriented language,* in: Yonezawa, A. and Tokoro, M., (eds.), Object oriented concurrent systems (MIT Press, 1987) pp. 199-220

[America, 1987a] P. AMERICA, *Inheritance and subtyping in a parallel object oriented language,* Proc. ECOOP 87, Paris, LNCS 276 (1987) pp. 234-242

[America, 1989] P. AMERICA, *Language definition of POOL-X,* Doc. Prisma 0350, Philips Research Laboratorium, Eindhoven

[America, 1989a] P. AMERICA, *Changes in POOL-X for Release 2.0,* Doc. Pooma 0104, Philips Research Laboratorium, Eindhoven

[America, 1989b] P. AMERICA, *Issues in the design of a parallel object oriented language,* Doc. DOOM 452, Philips Research Laboratorium, Eindhoven, also in [America and Rutten, 1989].

[America and de Bakker, 1988] P. AMERICA AND J.W. DE BAKKER, *Designing equivalent models for process creation,* TCS, Vol. 60, No. 2 (1988) pp. 109-176

[America et al, 1989] P. AMERICA, J.W. DE BAKKER, J.N. KOK AND J.J.M.M. RUTTEN, *Denotational semantics of a Parallel Object Oriented Language,* Information and Computation, Vol. 83, No. 2 (1989) pp. 152-205

[America and Rutten, 1989] P. AMERICA AND J.J.M.M. RUTTEN, *A parallel object-oriented language: design and foundations,* Ph.D. thesis, Free University Amsterdam (1989)

[America and Rutten, 1989a] P. AMERICA AND J.J.M.M. RUTTEN, *Solving reflexive domain equations in a category of complete metric spaces,* Journal of Computer and System Sciences

[Andrews and Schneider, 1983] G.R. ANDREWS AND F.B. SCHNEIDER, *Concepts and notations for concurrent programming,* ACM Computing Surveys, 15(1) (1983) pp.3-43

[Apt et al, 1987] K. APT , H. BLAIR AND A. WALKER, *Towards a theory of declarative knowledge,* In: Foundations of Deductive Databases and Logic Programming J. Minker (ed.) Morgan Kaufmann, Los Altos, 1987

[Armstrong et al, 1986] J.L. ARMSTRONG, N.H. ELSHIEWY AND R. VIRDING, *The Phoning Philosophers Problem or Logic for Telecommunications,* Proc. Symp. of Logic Programming, Salt Lake City (1986) pp. 28-35

[Bal et al, 1989] H. BAL, J. STEINER AND A. TANENBAUM, *Programming languages for distributed systems,* ACM Computing Surveys, Vol. 21, No. 3 (1989) pp. 262-322

[de Bakker et al, 1984] J.W. DE BAKKER, J.A. BERGSTRA, J.W. KLOP AND J.-J. CH. MEYER, *Linear time and branching time semantics for recursion with merge,* TCS 34 (1984) pp. 135-156

[de Bakker and Zucker, 1982] J.W. DE BAKKER AND J.I. ZUCKER, *Processes and the denotational semantics of concurrency,* Information and control 54 (1982) pp. 70-120

[de Bakker, 1988] J.W. DE BAKKER, *Comparative semantics for flow of control in logic programming without logic,* Report CS-R8840 (Centre for Mathematics and Computer Science, 1988)

[Beemster, 1990] M. BEEMSTER, *POOL±X,* doc. P0522, Philips Research Laboratorium, Eindhoven

[Black et al, 1987] A. BLACK, N. HUTCHINSON, E. JUL AND L. CARTER, *Distribution and abstract types in Emerald,* IEEE Trans. Softw. Eng. SE-13, 1 (Jan, 1987), pp. 65-76

[Bobrow, 1984] D.G. BOBROW, *If Prolog is the answer, what is the question?,* Proc. Conf. 5th Gen. Comp. Systems ICOT (1984) pp. 139-145

[de Boer et al, 1989] F.S. DE BOER, J.N. KOK, C. PALAMIDESSI, J.J.M.M. RUTTEN, *Semantic models for a version of Parlog,* In G. Levi and M. Martelli (eds.) Proc. of the sixth Conf. on Logic Programming Lisboa (1989) MIT Press, pp. 621-636 (extended version to appear in TCS)

[de Boer et al, 1989a] F.S. DE BOER, J.N. KOK, C. PALAMIDESSI, J.J.M.M. RUTTEN, *Control flow versus logic: a denotational and a declarative model for guarded Horn clauses,* Rep CS-R8952, Centre for Mathematics and Computer Science (1989)

[De Bosschere, 1989] K. DE BOSSCHERE, *Parallelism in Logic Programming,* Report DG 89-02 Lab. voor Electronica en Meettechniek, Gent, 1989

[Browne, 1986] J.C. BROWNE, *Framework for the formulation of parallel computation structures,* Parallel Computing 3 (1986) pp. 1-9

[de Bruin, 1986] A. DE BRUIN, *Experiments with continuation semantics: jumps, backtracking, dynamic networks,* Ph. D. thesis, Vrije Universiteit, Amsterdam (1986)

[Buchanan and Shortliffe, 1984] B.J. BUCHANAN, E.H. SHORTLIFFE (EDS.), *Rule Based Expert Systems, Addison-Wesley*

[Butler Cox, 1983] *Expert Systems,* Report no 37 Butler Cox Foundation Sept. 1983

[Butler and Karonis, 1988] R.M. BUTLER AND N.T. KARONIS, *Exploitation of parallelism in prototypical deduction problems,* CADE-9 LNCS 310 Springer (1988) pp. 333-343

[Butler et al, 1986] R. BUTLER, E. LUSK, W. MCCUNE AND R. OVERBEEK, *Parallel logic programming for numeric applications,* LNCS 225, 1986, pp. 375-388

[de Bruin and de Vink, 1988] A. DE BRUIN AND E.P. DE VINK, *Continuation semantics for Prolog with Cut,* Proc. Theory and Practice of Software Development '89 Vol. I, J. Diaz and F. Orejas (eds.), LNCS 351 (1989) pp. 178-192

[Campbell, 1984] J.A. CAMPBELL, *Implementations of Prolog,* Ellis Horwood 1984

[Chambers et al, 1984] F.B. CHAMBERS, D.A. DUCE, G.P. JONES (EDS.), *Distributed Computing,* (Academic Press, 1984)

[Chen and Warren, 1988] W. CHEN AND D.S. WARREN, *Objects as intensions,* Proc. 5th Int. Log. Prog. Conf., Seattle (1988) pp. 404-419

[Clark and Tarnlund, 1984] K.L. CLARK, S.A. TARNLUND (EDS.), *Logic Programming,* (Academic Press, 1982)

[Clark and Gregory, 1986] K.L. CLARK AND S. GREGORY, *PARLOG: Parallel programming in Logic,* ACM TOPLAS Vol. 8, No. 1 (1986) pp. 1-49

[Clocksin and Mellish, 1981] W.F. CLOCKSIN AND C.S. MELLISH, *Programming in Prolog,* Springer Verlag, New York

[Cohen, 1985] J. COHEN, *Describing Prolog by its interpretation and compilation,* CACM, Dec 1985, pp. 1311-1324

[Conery, 1987] J.S. CONERY, *Parallel execution of logic programs,* Kluwer, Boston 1987

[Conery, 1988] J.S. CONERY, *Logical Objects,* Int. Conf. on Logic Programming, Seatlle (1988) pp. 420-434

[Davis, 1980] R. DAVIS, *The Contract Net Protocol: High Level Communication and Control in a Distributed Problem Solver,* IEEE Trans. on Comp. Vol C-29 No 12 (1980) pp. 1104-1113

[Davison, 1989] A. DAVISON, *Polka: A Parlog Object Oriented Language,* Ph.D. thesis, Dept. of Computing, Imperial College London (1989)

[Davison, 1989a] A. DAVISON, *Design issues for logic programming-based object oriented languages,* Report Dep. of Computing, Imperial College London (1989), also in [Davison, 1989]

[DeGroot, 1984] D. DEGROOT, *Restricted and-parallelism,* Proc. FGCS (ICOT, 1984) pp. 471-478

[DoD, 1982] US DEPARTMENT OF DEFENSE, *Reference manual for the Ada programming language,* 1982

[Dijkstra, 1971] E.W. DIJKSTRA, *Hierachical oredering of sequential processes,* Acta Informatica 1, (1971) pp. 115-138

[Eliëns, 1989] A. ELIËNS, *Extending Prolog to a Parallel Object Oriented Language,* Proc. IFIP W.G. 10.3 Working Conference on Decentralized Systems (1989) Lyon

[van Emden and Kowalski, 1976] M.H. VAN EMDEN, R.A. KOWALSKI, *The semantics of predicate logic as a programming language,* JACM Vol. 21, No 4, Oct 1976, pp. 733-742

[van Emden and de Lucena Filho, 1982] M.H. VAN EMDEN, G.J. DE LUCENA FILHO, *Predicate logic as a language for parallel programming,* in: [Clark and Tarnlund, 1984] pp. 189-199

[Fahlman, 1985] S.F. FAHLMAN, *Parallel processing in Artificial Intelligence,* Parallel Computing 2 (1985) pp. 283-285

[Falaschi et al, 1984] M. FALASCHI, G. LEVI AND C. PALAMIDESSI, *A synchronization logic: Axiomatics and Formal Semantics of Generalized Horn Clauses,* Information and Control 60 (1984) pp. 36-69

[Fox, 1981] M. FOX, *An organizational view of distributed systems,* IEEE SMC-11, No. 1 (1981) pp. 70-80

[Fukunaga and Hirose, 1986] K. FUKUNAGA AND S. HIROSE, *An experience with a Prolog-based Object Oriented Language,* OOPSLA 86, SIGPLAN Notices Vol. 21, No. 11 (1986) pp. 224-231

[Gelernter et al, 1986] D. GELERNTER, S. AHUJA AND N. CARRIERO , *Linda and friends,* Computer 19, 8 (1986) pp. 26-34

[Goguen, 1984] J.A. GOGUEN, *Parametrized Programming,* IEEE Trans. on Software Engin. 10 (1984) pp. 528-543

[Goldberg and Robson, 1983] A. GOLDBERG AND D. ROBSON, *Smalltalk-80: The language and its implementation,* Addison Wesley (1983)

[Gomez and Chandrasekaran, 1981] F. GOMEZ AND B. CHANDRASEKARAN, *Knowledge Organization and Distribution for Medical Diagnosis,* IEEE Trans. on Systems, Man and Cyb. Vol. SMC-11, No. 1 (1981) pp. 34-42

[Hasegawa and Amamiya, 1984] R. HASEGAWA, M. AMAMIYA, *Parallel Execution of Logic Programs based on Dataflow Concept,* Proc 5th Gen. Comp. Syst. ICOT (1984) pp. 507-516

[Hayes-Roth, 1985] F. HAYES-ROTH, *Rule Based Systems,* CACM Vol. 28 No. 9 Sept. (1985) pp. 921-932

[van den Herik and Henseler, 1986] H.J. VAN DEN HERIK AND J. HENSELER, *Control mechanisms in the parallel knowledge based system HYDRA,* Report 86-36 University Delft

[Hermenegildo and Nasr, 1986] M.V. HERMENEGILDO AND R. NASR, *Efficient management of backtracking in and-parallelism,* LNCS 225 (1986) pp. 40-55

[Hermenegildo, 1986] M.V. HERMENEGILDO, *An abstract machine based execution model for computer architecture design and efficient implementation of logic programs in parallel,* thesis, TR-86-20, University of Texas at Austin (1986)

[Hewitt, 1977] C. HEWITT, *Viewing Control Structures as Patterns of Passing Messages,* Art. Int. (1977) 323-364

[Hoare, 1978] C.A.R. HOARE, *Communicating Sequential Processes,* CACM, Vol. 21, No. 8 (1978) pp. 666-677

[Houtsma and Balsters, 1988] M.A.W. HOUTSMA AND H. BALSTERS, *Formalizing the Data and Knowledge Model,* Memorandum INF-88-23 (University of Twente, 1988)

[Iline and Kanoui, 1987] H. ILINE AND H. KANOUI, *Extending Logic Programming to Object Programming: The system LAP,* Proc. 10th IJCAI, Vol. 1 (1987) pp. 34-39

[Inmos, 1984] INMOS LTD, *The Occam Programming Manual,* Prentice Hall International, London (1984)

[Ishikawa and Tokoro, 1986] Y. ISHIKAWA AND M. TOKORO, *A concurrent object oriented knowledge representation language Orient84/K: Its features and implementation,* OOPSLA 86, SIGPLAN Notices Vol. 21, No. 11 (1986) pp. 232-241

[Jones and Mycroft, 1984] N. JONES AND A. MYCROFT, *Stepwise development of operational and denotational semantics for Prolog,* 1984 Int. Symp. on Logic Programming, Atlantic City (1984) pp. 281-288

[Jones, 1985] G. JONES, *Programming in Occam,* Oxford TM PRG-43 (1985)

[Kahn et al, 1986] K. KAHN, E. TRIBBLE, M. MILLAR, D. BOBROW, *Objects in concurrent logic programming languages,* OOPSLA 86, SIGPLAN Notices Vol. 21, No. 11, 1986 pp. 242-257

[Karam, 1988] G.M. KARAM, *Prototyping Concurrent systems with Multilog,* Technical Report Dept. of Systems and Computer Eng. Carleton University (1988)

[Karam, 1988a] G.M. KARAM, C.M. STANCZYK AND G.W. BOND, *Critical races in ADA Programs,* TR SCE-88-15 Carleton University Ottawa (1988)

[Karam, 1989] G.M. KARAM, *Mlog: A language for prototyping Concurrent Systems,* electronic news

[Klint, 1985] P. KLINT, *A study in string processing languages,* LNCS 205, Springer-verlag (1985)

[Klint, 1986] P. KLINT, *Modularization and reusability in current programming languages,* Centre for Mathematics and Computer Science Report CS-R8635

[Kok and Rutten, 1988] J. KOK AND J. RUTTEN, *Contractions in comparing concurrency semantics,* LNCS 317 (1988) pp. 317-332

[Kornfield and Hewitt, 1981] W. KORNFELD AND C.E. HEWITT, *The scientific community metaphor* , IEEE SMC-11 No. 1 (1981)

[Kowalski, 1979] R. KOWALSKI, *Logic for problem solving,* (1979) North Holland

[Knuth, 1984] D. KNUTH, *Literate programming,* The Computer Journal, Vol. 27, No. 2, 1984, pp. 97-111

[Lassez and Maher, 1983] J.-L. LASSEZ, M.J. MAHER, *The denotational semantics of Horn clauses as production-systems,* in: Proc. Am. Nat. Conf. on A.I. Washington 1983

[Lipovski and Hermenegildo, 1985] G.J. LIPOVSKI M.V. HERMENEGILDO, *B-log: A branch and bound methodology for the parallel execution of logic programs,* IEEE Proc. Int. Conf. on Par. Processing (1985) pp.560-567

[Lloyd, 1984] J.W. LLOYD, *Foundations of Logic Programming,* Symbolic Computation, Springer-Verlag (1984)

[Lucas, 1986] P.J.F. LUCAS, *Knowledge representation and inference in rule based systems,* Report CS-R8613, Centre for Mathematics and Computer Science (1986)

[Lusk and Overbeek, 1980] E.L. LUSK, A.R. OVERBEEK, *Data-structures and Control-architectures for the implementation of theorem-proving programs,* Proc. 5th Conf. on Aut. Deduction, LNCS 87 (1980) pp. 232-249

[Lusk et al, 1982] E.L. LUSK, W. MCCUNE, A.R. OVERBEEK, *Logic Machine Architecture: inference mechanisms,* Proc. 6th Conf. on Aut. Deduction, LNCS 138 (1982) pp. 85-97

[Lusk and Overbeek, 1984] E.L. LUSK AND R.A. OVERBEEK, *Parallelism in automated reasoning systems,* Report Argonne National Laboratory (1984)

[Manthey and Bry, 1988] R. MANTHEY AND F. BRY, *SATCHMO: A theorem prover implemented in Prolog,* CADE-9, LNCS 310 Springer (1988) pp. 415-434

[McArthur et al, 1982] D. MCARTHUR, R. STEEB AND S. CAMMARATA, *A framework for distributed problem solving,* Proc. Conf. AAAI (1982) pp. 181-184

[Mello and Natali, 1986] P. MELLO AND A. NATALI, *Programs as collections of communicating Prolog units,* LNCS 213 (1986) pp. 274-288

[Meltzer, 1982] B. MELTZER, *Prolegomena to a theory of efficiency of proof procedures,* Machine Intelligence (1982) pp. 15-33

[Miller, 1986] D. MILLER, *A theory of modules for logic programming,* IEEE Symp. on Logic Prog. (1986) pp. 106-114

[Monteiro, 1981] L. MONTEIRO, *An extension to Horn clause logic allowing the definition of concurrent processes,* LNCS 107 (1981) pp. 401-407

[Monteiro, 1984] L. MONTEIRO, *A proposal for distributed programming in logic,* in: [Campbell, 1984] (1984), pp. 329-340

[Monteiro and Porto, 1988] L. MONTEIRO AND A. PORTO, *Contextual Logic Programming,* Report UNL-50/88 (University Lisboa, 1988)

[Mundie and Fisher, 1986] D.A. MUNDIE AND D.A. FISHER, *Parallel processing in Ada,* IEEE Computer Vol. 19 No. 8 (1986) pp. 20-25

[Nilsson, 1982] J.F. NILSSON, *Formal Vienna definition method models of Prolog,* in: [Campbell, 1984]

[Ohsuga and Yamauchi, 1985] S. OHSUGA, H. YAMAUCHI, *Multi-layer logic - A predicate logic including data-structure as knowledge representation language,* in: New Generation Computing 3 (1985) pp. 403-43

[Pelaez, 1989] E. PELAEZ, *Parallelism: Performance or Programming,* Computers & Society, Vol. 19, No. 4 (1989)

[Pereira and Nasr, 1984] L.M. PEREIRA AND R. NASR, *Delta Prolog: A distributed logic programming language,* Proc. FGCS (ICOT, 1984) pp. 283-231

[Pereira et al, 1986] L. PEREIRA, L. MONTEIRO, J. CUNHA, J. AND J. APARICO, *Delta Prolog: a distributed backtracking extension with events,* LNCS 225 (1986) pp. 69-83

[Plotkin, 1983] G.D. PLOTKIN, *An operational semantics for CSP,* in: Formal Description of Programming Concepts II, D.Bjorner (ed.), North Holland, Amsterdam (1983) pp. 199-223

[Rabin, 1974] M.O. RABIN, *Theoretical impediments to Artificial Intelligence,* IFIP 1974, pp. 615-619

[Ramakrishnan, 1986] R. RAMAKRISHNAN, *Annotations for Distributed Programming in Logic,* ACM POPL St. Petersburg (1986) pp. 255-262

[Ringwood, 1988] G.A. RINGWOOD, *Parlog86 and the dining logicians,* CACM, Vol. 31, No. 1 (1988) pp. 10-25

[Rizk et al, 1989] A. RIZK, J-M. FELLOUS AND M. TUENI, *An object oriented model in the concurrent logic programming language Parlog,* Report No 1067 (INRIA, 1989)

[Shapiro and Takeuchi, 1983] E. SHAPIRO AND A. TAKEUCHI, *Object-oriented programming in Concurrent Prolog,* New Generation Computing, Vol. 1, No. 2 (1983) pp. 5-48

[Shapiro, 1984] E. SHAPIRO, *Systolic programming: a paradigm of parallel processing,* Proc. FGCS (ICOT, 1984) pp. 458-470

[Shapiro, 1986] E. SHAPIRO, *Concurrent Prolog: Progress Report,* IEEE Computer Vol. 19, No. 8 (1986) pp. 44-59

[Shapiro, 1989] E. SHAPIRO, *The family of concurrent logic programming languages,* ACM Computing Surveys, Vol. 21, No. 3 (1989) pp. 414-510

[Smith and Davis, 1981] R.G. SMITH AND R. DAVIS, *Frameworks for Cooperation in Distributed Problem Solving,* IEEE Trans. on Systems, Man and Cyb. Vol. SMC-11 No 1 (1981) pp.61-69

[Stefik and Bobrow, 1986] M. STEFIK AND D.G. BOBROW, *Object Oriented Programming: Themes and Variations,* The AI Magazine, Vol.10, No.4 (1986) pp. 40-62

[Stroustrup, 1986] B. STROUSTRUP, *The C++ programming language,* Addison Wesly (1986)

[Subrahmanyam, 1985] P.A. SUBRAHMANYAM, *The Software Engineering of Expert Systems: Is Prolog appropriate?,* IEEE Trans. on Softw. Engin. Vol. SE-11 No. 11, 1985, pp. 1391-1400

[Talukdar, 1986] S.N. TALUKDAR, E. CARDOZO, L. LEAO, R. BANARES AND R. JOOBBANI, *A system for distributed problem solving,* in: Coupling symbolic and numerical computing in expert systems, J.S. Kowalik (ed.)

[Touretzky, 1986] S. TOURETSKY , *The mathematics of inheritance systems,* Pitman London (1986)

[Tsujino et al, 1984] Y. TSUJINO, M. ANDO, T. ARAKI AND N. TOKURA, *Concurrent C: A programming language for distributed systems,* Software Practice and Experience 14-11 (1984) pp. 1061-1078

[de Vink, 1989] E.P. DE VINK, *Comparative semantics for Prolog with cut,* Science of Computer Programming 13 (1989/1990) pp. 237-264

[Warren, 1977] D. WARREN, *Implementations of Prolog: Compiling Predicate Logic Programs,* Dep. of Art. Int. Univ. of Edinburgh

[Warren, 1983] D. WARREN, *An abstract Prolog instruction set,* Technical Note 309, SRI International (1983)

[Wegner, 1987] P. WEGNER, *Dimensions of Object-Based Language Design,* OOPSLA87, Orlando Florida (1987) pp. 168-182

[Wiener and Pinson, 1988] R.S. WIENER AND J.P. PINSON, *An Introduction to Object-Oriented Programming and C++,* Addison-Wesley (1988)

[Wos et al, 1984] L. WOS, R. OVERBEEK, E. LUSK AND J. BOYLE, *Automated Reasoning: Introduction and applications,* Prentice Hall (1984)

[Yokoi, 1986] S. YOKOI, *A Prolog based object oriented language SPOOL and its compiler,* Proc. Logic Programming 86, Tokyo, LNCS 264 (1986) pp. 116-125

[Zaniolo, 1984] C. ZANIOLO, *Object oriented programming in Prolog,* Proc. Int. Symp. on Logic Programming, Atlantic City (1984) pp. 265-270

Author Index

Samenvatting (in het nederlands):

DLP - Een taal voor gedistribueerd logisch programmeren

Ontwerp, semantiek en implementatie

Voorwoord (bij de samenvatting)

Dit proefschrift behandelt de taal DLP, een taal voor gedistribueerd logisch programmeren.

In deze samenvatting zal ik pogen een informele weergave van de inhoud van mijn proefschrift te geven, zonder afbreuk te doen aan het wetenschappelijk gehalte van mijn werk.

Ter introductie, om ook de lezers die de benodigde technische kennis missen een indruk te geven, schets ik aan de hand van een metafoor de achtergrond waartegen mijn werk geplaatst moet worden.

> De metafoor berust op de overeenkomst tussen een machine en een bediende. Een bediende kun je laten doen wat je wilt, bijvoorbeeld de afwas. Het probleem dat ik aan de orde wil stellen is hoe te communiceren met een bediende. Terwille van het verhaal neem ik aan dat in vroeger tijden een bediende precies verteld moest worden welke handelingen nodig zijn voor het uitvoeren van een opdracht. Het zal duidelijk zijn, voor een ieder die wel eens afwast, dat het opsommen van de reeks handelingen waaruit het afwassen bestaat eigenlijk zo uitputtend is dat de bediende vanuit het oogpunt van werkbesparing nauwelijks van enige waarde is.
>
> Stel dat er ook intelligente bediendes bestaan, die je slechts hoeft te vertellen *dat* de afwas gedaan moet worden en niet tot in detail *hoe*. Het instrueren van zo'n, laat ik maar zeggen moderne, bediende is aanmerkelijk eenvoudiger dan het bevel voeren over een bediende oude stijl.
>
> Een soortgelijke ontwikkeling heeft zich bij machines voorgedaan. Vroeger bestond het programmeren van een machine eruit stap voor stap aan te geven wat gedaan moest worden om een taak uit te voeren. Tegenwoordig hebben we de beschikking over programmeertalen waarmee we op een tamelijk eenvoudige wijze kunnen aangeven wat er moet gebeuren door op een logische wijze te beschrijven wat de aard van het probleem is. Het onderliggende systeem zoekt uit hoe de machine daarmee uit de voeten kan. Een voorbeeld van zo'n taal is de logische programmeertaal Prolog.
>
> Nogmaals met een beroep op uw voorstellingsvermogen, stel dat de taak die uitgevoerd moet worden dermate omvangrijk of ingewikkeld is dat één bediende, hoe intelligent ook, die niet alleen kan uitvoeren. Wat ligt meer voor de hand dan het inschakelen van meerdere bediendes, die samen aan het probleem gezet worden, zodat de inspanning over de groep gedistribueerd wordt.
>
> Inmiddels weten we hoe we één bediende moeten instrueren. Het beheren van een aantal bediendes is echter aanmerkelijk moeilijker. In de meeste gevallen moeten de bediendes ook onderling communiceren om de taak naar behoren te verrichten. Terugkerend naar de werkelijkheid die ten grondslag ligt aan deze metafoor, is ons probleem te karakteriseren als: welke taalconcepten zijn nodig om een taak gedistribueerd over een aantal machines te laten uitvoeren? Als oplossing

voor dit probleem wordt in dit proefschrift de taal DLP geïntroduceerd, een taal voor gedistribueerd logisch programmeren.

In het proefschrift wordt ingegaan op het *ontwerp*, de *semantiek* en de *implementatie* van DLP.

In het deel dat gaat over het *ontwerp* worden de taalconcepten geïntroduceerd die het mogelijk maken een parallelle machine door middel van een logische programmeertaal opdrachten te geven.

Het tweede deel gaat in op de semantiek van gedistribueerd logisch programmeren. Onder semantiek versta ik een wiskundige beschrijving van de functionaliteit van een taal. Met andere woorden, de semantiek geeft op preciese wijze weer wat een programma in de betreffende taal doet. Een dergelijke beschrijving kan dienen als leidraad bij een realisatie van de taal (op een machine) en, niet minder belangrijk, als toets of de gekozen realisatie voldoet aan de eisen die wij aan de taal opleggen.

In het *implementatie* deel bespreek ik de door mij ontwikkelde realisatie van DLP. Ik moet opmerken dat ik (slechts) een prototype geïmplementeerd heb, dat wil zeggen een systeem dat weliswaar de functionaliteit van de taal belichaamt maar, wat de verwerkingssnelheid betreft, in veel opzichten verbeterd kan worden.

Deel I: Ontwerp

Het ontwerp van een taal is een vrij gevoelsmatige kwestie. Een belangrijk motief bij het ontwerp van DLP was het ontwikkelen van een taal waarin het plezierig is te programmeren.

In deel I introduceer ik DLP. Ter illustratie van de mogelijkheden die de taal biedt beschrijf ik de implementatie van een gedistribueerd medisch expertsysteem. Tevens geef ik een schets van de achtergrond van DLP, en licht ik de belangrijkste ontwerpbeslissingen toe.

Hoofdstuk 1: Inleiding

Bij het ontwerp van DLP heb ik me laten leiden door een drietal programmeerstijlen.

Mijn belangrijkste inspiratiebron ligt bij de stijl van programmeren die met logisch programmeren wordt aangeduid. Programma's in een logische programmeertaal laten een logische interpretatie toe en zijn tevens uitvoerbaar door een machine. Een dergelijke taal is in het bijzonder geschikt voor het ontwikkelen van toepassingen in de sfeer van de kunstmatige intelligentie, zoals bijvoorbeeld expertsystemen. De taal Prolog is waarschijnlijk de belangrijkste representant van deze klasse van talen. In toenemende mate wordt Prolog ook in het buiten-universitaire leven gebruikt voor de ontwikkeling van ondermeer expertsystemen.

Een tweede inspiratiebron, voor de organisatie van programma's, is het object-georiënteerde programmeren. De leidende gedachte achter dit paradigma is dat problemen zich beter middels een computer laat modelleren indien de representatie van het probleem aansluit bij de werkelijkheid waarin het probleem speelt.

Een object is te zien als een entiteit die bepaalde diensten aanbiedt. De interne toestand van het object is onzichtbaar van buiten af. Zulk een afscherming dient ter beveiliging. De verantwoording voor het omgaan met de data die het object herbergt ligt bij het object zelf. Kenmerkend voor de object-georiënteerde stijl van programmeren is het gebruik van overerving, waardoor een klasse van objecten gebruik kan maken van de eigenschappen van de geërfde objecten.

De derde invalshoek, parallellisme, laat zich op twee manieren motiveren. Bepaalde problemen laten zich op een natuurlijke wijze modelleren door middel van parallelle processen. Een ander motief voor het gebruik van parallellisme komt voort uit de wens de verwerkingssnelheid van een programma te verhogen door gebruik te maken van een parallelle machine. Of een dergelijke snelheidswinst te realiseren is hangt in veel gevallen af van de aard van het probleem.

Bij het ontwerp van DLP stond het probleem van de expressiviteit voorop.

Logisch programmeren biedt de mogelijkheid tot een declaratieve beschrijving van een probleem, dat wil zeggen een formulering van de oplossing met behulp van een op de logica gebaseerd formalisme.

De object-georiënteerde stijl van programmeren bewerkstelligt vooral een natuurlijke opdeling van de oplossing in objecten, overeenstemmend met de conceptuele structuur van het probleemgebied.

De aandacht voor parallellisme is erop gericht de feitelijke uitvoering van de taak te zien als een collectie actieve coöpererende processen.

De vraag die ik in dit proefschrift heb gesteld is hoe deze stijlen van programmeren met elkaar verenigbaar zijn. Het antwoord daarop is weer te geven aan de hand van de pseudo-wiskundige formule

$$\text{DLP} = \text{LP} + \text{OO} + ||$$

die uitdrukt dat DLP de door mij beoogde combinatie van stijlen is: logisch programmeren (LP), object georiënteerd programmeren (OO) en parallellisme (||).

De taal DLP is bedoeld voor een parallelle machine bestaande uit, zeg tussen de tien en honderd, processorknopen die met elkaar verbonden zijn door een communicatienetwerk.

Hoofdstuk 2: De uitbreiding van Prolog tot een parallelle object-georiënteerde taal

Bij de ontwikkeling van DLP ben ik uitgegaan van de logische programmeertaal Prolog. Een programma in Prolog bestaat uit een aantal *clauses*.

Clauses van de vorm

$$A \leftarrow B_1, ..., B_n$$

zijn op te vatten als logische regels, die uitdrukken dat A afleidbaar is als aan de condities $B_1, ..., B_n$ voldaan is.

Een berekening in Prolog vindt plaats door te vragen of een doel A' afleidbaar is uit de gegeven regels. Het berekeningsmechanisme van Prolog draagt er zorg voor dat een afleiding van A' gezocht wordt.

Zowel de clauses als het doel kunnen variabelen bevatten. Een afleiding is in zo'n geval slechts mogelijk indien een geschikte toekenning van waarden aan die (logische) variabelen gevonden wordt. Zo'n toekenning heet een *substitutie.*

De zoekstrategie van Prolog maakt het mogelijk alle oplossingen voor een doel te vinden. Dit gebeurt door bij te houden welke alternatieve keuzes nog gemaakt kunnen worden in het vinden van een afleiding. De mogelijkheid terug te komen op eerder gemaakte keuzes wordt *backtracken* genoemd. Bij het backtracken over alternatieve oplossingen kunnen andere substituties gevonden worden. Indien een doel geen variabelen bevat is het in het algemeen niet zinvol na te gaan of er meer dan een oplossing is.

Om de combinatie van Prolog met de andere genoemde stijlen van programmeren tot stand te brengen heb ik gezocht naar een uitbreiding van Prolog met taalconstructies voor parallel object-georiënteerd programmeren.[2]

Objecten De objecten waarmee we Prolog uitbreiden bestaan uit *data*, opgeslagen in niet-logische variabelen, en *clauses* die de functionaliteit van het object bepalen.

Een objectdeclaratie heeft dan ook de vorm

object naam {
var variabelen.
clauses
}

Een object kan gevraagd worden de oplossing voor een doel te zoeken. Voor de afleiding daarvan heeft het object slechts de beschikking over de clauses die in de objectdeclaratie staan. Het object kan echter weer andere objecten vragen (sub) doelen af te leiden indien dat nodig is voor het vinden van een oplossing. Een object vragen een doel af te leiden noemen we een methode-aanroep aan het object.

Overerving Behalve door middel van een methode-aanroep kan een object een ander object ook gebruiken door de functionaliteit van dat object te erven. Door overerving krijgt het object de beschikking over zowel de niet-logische variabelen als de clauses van het geërfde object.

Een object kan van meer objecten erven. Bijvoorbeeld, bij een declaratie van de vorm

object a:(b,c) {
var variabelen.
clauses
}

[2]De feitelijke historie is overigens dat ik in eerste instantie begonnen ben met het zoeken naar uitbreidingen van Prolog die parallellisme mogelijk maken, en me pas later ben gaan richten op object-georiënteerde aspecten. Voor de uitleg ligt het echter meer voor de hand te beginnen met de introductie van objecten.

bevat het object a, behalve de eigen niet-logische variabelen en clauses, tevens de niet-logische variabelen en clauses van de objecten b en c.

Er is een belangrijk verschil tussen deze twee manieren van gebruik. Bij een methode-aanroep zijn de niet-logische variabelen afgeschermd, dat wil zeggen onzichtbaar voor de aanroeper. Bij overerving echter, worden de niet-logische variabelen eenvoudigweg toegevoegd aan de eigen niet-logische variabelen, zodat het ervende object er direct toegang tot heeft.

Transparant Methode-aanroepen zijn transparent, in de zin dat er voor de gebruiker geen verschil merkbaar is tussen een methode-aanroep en de gewone evaluatie van een doel, voor wat betreft het backtracken over alternatieve oplossingen. Dat hier sprake is van een ontwerpbeslissing wordt pas duidelijk zodra we parallellisme introduceren.

Parallellisme Parallellisme krijgen we door een aantal objecten tegelijkertijd actief te laten zijn. Een actief object is een object waarmee een proces is geassocieerd dat de eigen activiteit van het object uitvoert.

Actieve objecten moeten expliciet gecreëerd worden. Dit gebeurt door een copie te maken van het gedeclareerde object en een proces op te starten dat de eigen activiteit van het proces uitvoert. Dit proces noemen we het *constructor*-proces van het object.

Rendez-vous Zolang een object actief is kunnen methode-aanroepen niet beantwoord worden. Dit is pas mogelijk zodra het object de eigen activiteit expliciet onderbreekt en de bereidheid tot acceptatie van een methode-aanroep aangeeft. Het aanroepende object wacht hierop en vervolgt zijn berekening pas zodra het antwoord op de aanroep gegeven is. In de context van parallellisme spreken we hier van een *rendez-vous* tussen twee processen. Beide processen onderbreken hun activiteit om met elkaar in contact te treden, en vervolgen hun eigen activiteit na de beëindiging van het rendez-vous.

Wederzijdse uitsluiting Het rendez-vous biedt een natuurlijke bescherming tegen de interferentie van andere processen, tijdens de interactie tussen de beide processen.

Door de identificatie van objecten en processen voegt de object-georiënteerde programmeerstijl zich op een vanzelfsprekende wijze in dit communicatiemodel, althans in het geval dat de evaluatie van een methode een enkel antwoord oplevert. Wat te doen echter met eventuele alternatieve oplossingen die uit een methode-aanroep kunnen resulteren?

Bij het ontwerp van DLP stond ik hier voor de keuze tussen een aantal alternatieve ontwerpbeslissingen.

Een mogelijkheid is de beperking in te voeren dat een rendez-vous slechts één antwoord toelaat. Dit brengt echter met zich mee dat er een onderscheid is tussen passieve en actieve objecten, hetgeen conceptueel onwenselijk is.

Een andere mogelijkheid is het aangeroepen object zijn activiteit pas te laten hervatten nadat alle antwoorden op de methode-aanroep geretourneerd zijn. Het nadeel van deze oplossing is dat in het geval er oneindig veel oplossingen zijn,

het object zijn activiteit nooit zal hervatten en dus onbereikbaar is voor andere objecten.

De oplossing waarvoor ik uiteindelijk gekozen heb houdt in dat er niet langer sprake is van een identificatie van actieve objecten en processen. Behalve het constructor-proces wordt er voor iedere methode-aanroep aan een object een apart proces gecreëerd, zodat het backtracken over de resultaten van de aanroep parallel aan de eigen activiteit van het object plaats kan vinden. Het aangeroepen object hervat zijn eigen activiteit nadat het evaluatieproces het eerste antwoord geretourneerd heeft.

Met ieder object kunnen dus meer processen verbonden zijn, een constructor-proces en een willekeurig aantal processen gecreëerd voor de evaluatie van een methode-aanroep. Al deze processen delen de niet-logische variabelen van het object waar zij aan refereren.

Zodra een methode-aanroep actief wordt, vindt er geen interferentie door andere methode-aanroepen plaats totdat het eerste antwoord opgeleverd is. Ook de eigen activiteit van het aangeroepen object is voor zolang onderbroken. Wel kunnen andere processen nog bezig zijn met het backtracken over eerdere methode-aanroepen. Deze oplossing maakt dat voor actieve objecten een modificatie van de niet-logische variabelen veilig plaats kan vinden vóór de oplevering van het eerste antwoord.

Ook voor passieve objecten wordt bij iedere methode-aanroep een proces gecreëerd. Anders dan voor actieve objecten, kan voor passieve objecten een willekeurig aantal methode-aanroepen actief zijn.

Hoofdstuk 3: De taal DLP

Behalve de in het voorgaande geïntroduceerde constructies bevat de taal DLP nog een aantal andere mogelijkheden. De belangrijkste hiervan is de mogelijkheid objecten en processen te plaatsen op een specifieke processorknoop van de parallelle machine. Door een slimme plaatsing kan optimaal gebruik gemaakt worden van de mogelijkheden tot parallellisme.

Wellicht leeft bij de lezer al enige tijd de vraag waar niet-logische variabelen toe dienen. Gezien vanuit het perspectief van een logische programmeertaal vormen niet-logische variabelen immers een onzuiver element. De keuze voor niet-logische variabelen voor het opslaan van data, locaal voor het object, is gemaakt met het oog op toepassingen waarbij gebruik gemaakt wordt van mogelijk omvangrijke gegevensbestanden. Het gehanteerde protocol van wederzijdse uitsluiting van methode-aanroepen laat toe dat een wijziging van het gegevensbestand veilig kan geschieden vóór de oplevering van het eerste antwoord.

Hoofdstuk 4: Kennisrepresentatie en overerving

Als voorbeeld van een toepassing van DLP heb ik een gedistribueerd medisch expertsysteem ontwikkeld waarvan ik de opzet hier ruwweg zal schetsen.

Dit expertsysteem illustreert het gebruik van de mogelijkheden die de taal DLP biedt voor een object-georiënteerde implementatie van op kennis gebaseerde systemen, zoals het dynamisch creëeren van objecten, gedistribueerd backtracken,

overerving voor de representatie van kennis, en het delegeren van boodschappen naar andere objecten.

Expertsysteem Een expertsysteem is te karakteriseren als een systeem dat de vaardigheden van een menselijke expert bezit. Een typisch voorbeeld is een medisch expertsysteem dat in staat is diagnoses te opperen op grond van een aantal symptomen.

Een expertsysteem bestaat uit twee delen: een *bestand van gegevens* waar ondermeer de waargenomen symptomen in opgeslagen zijn, en een zogeheten *inferentie-machine* die uit deze gegevens nieuwe gegevens afleidt door gebruikmaking van kennisregels die de expertise bevatten.

Kennisregels van de vorm

$$condities \rightarrow conclusie$$

geven de voorwaarden aan waaronder nieuwe gegevens toegevoegd kunnen worden. Een nieuw gegeven kan toegevoegd worden aan het bestand van gegevens als de oorspronkelijke gegevens aan de condities van een regel voldoen.

De strategie die bij het afleiden van nieuwe gegevens wordt gebruikt is te karakteriseren als *achterwaarts redeneren*. Uitgegaan wordt van de conclusie, bijvoorbeeld een hypothese omtrent de te stellen diagnose, en vervolgens wordt gecontroleerd of aan de condities van de regel voldaan is. Aan een conditie is voldaan indien hetzij een bepaald gegeven voorkomt in het bestand, hetzij dat gegeven afgeleid kan worden door toepassing van weer andere regels.

Dokter Het stellen van een diagnose gebeurt in het werkelijke leven door een dokter. Om de functionaliteit van een dokter te modelleren definiëren we een object *dokter*.

Voorwaarde voor het stellen van een diagnose is dat de dokter beschikt over de kennis die hem in staat stelt een of meer hypotheses te vormen omtrent een mogelijk ziektebeeld. Voorts moet de dokter toegang hebben tot de gegevens van de patient. Een dokter, in ons systeem, is niet veel meer dan een inferentie-machine die successievelijk een aantal hypotheses toetst.

Ziekte Een object-georiënteerde modelleringstechniek is toegepast in de hiërarchische representatie van kennis omtrent ziektebeelden.

De structuur van onze kennis van ziektes heeft de vorm van een (omgekeerde) boom. Aan de wortel daarvan staat een object *ziekte*, dat de meest algemene kennis van ziektes vertegenwoordigt. Deze kennis is vastgelegd in regels die het mogelijk maken vast te stellen of de patient verschijnselen vertoont die wijzen op een ziekte. Tevens is er in dit object een lijst van mogelijke diagnoses opgenomen. Deze diagnoses functioneren als hypothese bij het zoeken naar een diagnose.

Andere ziektes zijn op te vatten als een bijzonder geval van het algemene ziektebeeld. De objecten die deze specifieke ziektes representeren delen door overerving in de meer algemene kennis. Deze meer algemene kennis is voor iedere ziekte afzonderlijk aangevuld met regels die de specifieke kennis met betrekking tot die ziekte belichamen.

Ook is voor iedere bijzondere ziekte een lijst van mogelijke diagnoses opgenomen, om het stellen van een diagnose gerichter te laten verlopen. Deze lijst overschrijft de lijst behorend bij het meer algemene geval.

Een voorbeeld van een meer specifiek ziektebeeld is de klasse van longziektes. De mogelijke diagnoses van deze klasse omvatten tuberculose en asthma. Tevens bevat het object *longziekte*, naast de overgeërfde algemene kennis omtrent ziekte, specifieke regels voor het vaststellen van een longziekte.

Specialist De hiërarchische structuur van medische kennis suggereert het zoeken naar mogelijke diagnoses, in het geval het een bijzondere ziekte betreft, uit te besteden aan een of meer specialisten. Zulk een distributie van taken is in overeenstemming met de wijze waarop een huisarts patienten doorverwijst naar een specialist.

Een *specialist* is een dokter die specifieke kennis heeft van een bepaalde klasse van ziektes en met het stellen van een diagnose tevens aanbevelingen doet voor eventuele verdere onderzoeken.

Deze definitie van een specialist maakt het mogelijk, aan de hand van de hiërarchie van mogelijke ziektes, op zoek te gaan naar de meest specifieke diagnose, beginnend bij het algemene ziektebeeld.

Om aanbevelingen voor verdere onderzoeken te kunnen doen is bij elke ziekte, behalve een lijst van mogelijke diagnoses, tevens een lijst van mogelijke oorzaken opgenomen die aangeeft welke ziektebeelden een bijzonder geval van de onderhavige ziekte zijn.

Kliniek Het gedistribueerde karakter van ons diagnostisch systeem komt pas ten volle aan het licht in de definitie van een *kliniek*, waarin de taakverdeling tussen specialisten geregeld wordt.

Een kliniek ontvangt patienten en wijst aan iedere patient een dokter toe. De dokter die een patient toegewezen krijgt is een *specialist* die alles weet van het meest algemene ziektebeeld.[3]

De specialist stelt een diagnose en geeft advies omtrent nader te verrichten onderzoeken. Aan de hand van deze adviezen vindt de verwijzing naar andere specialisten plaats, die kennis hebben van die meer specifieke ziektebeelden.

De aangewezen specialisten kunnen hun arbeid gelijktijdig verrichten. De gegevens van de patient blijven echter voor een ieder toegankelijk. Zo kan het voorkomen dat er een toevallige samenwerking tussen specialisten ontstaat, doordat een specialist gegevens afleidt die bruikbaar zijn voor andere specialisten.

Hoofdstuk 5: Ontwerpperspectieven

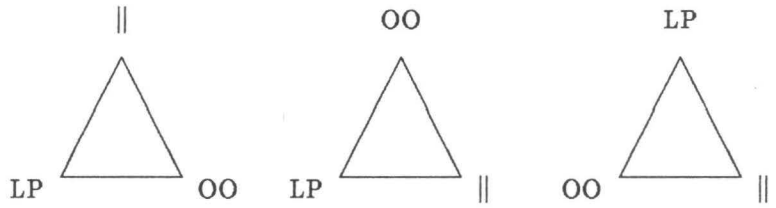
In mijn proefschrift zijn behalve de zojuist beschreven toepassing ook een aantal programmafragmenten opgenomen die de specifieke mogelijkheden van DLP voor het creëeren van objecten en de communicatie daartussen illustreren.

Kenmerkend voor de door mij voorgestelde taal is de mogelijkheid van backtracken over de resultaten van een methode-aanroep aan een actief object. Daarmee

[3]De definitie van een specialist laat toe dat wat we normaliter als een gewone dokter beschouwen tevens een specialist noemen.

onderscheid DLP zich van andere voorstellen die een integratie van logisch programmeren, object-georiënteerd programmeren en parallellisme beogen.

De verschillende voorstellen tot een combinatie van logisch programmeren, object-georiënteerd programmeren en parallellisme laten zich vergelijken aan de hand van de onderstaande diagrammen.



Het eerste diagram representeert de voorstellen die uitgaande van een combinatie van logisch programmeren en object-georiënteerd programmeren parallellisme proberen toe te voegen door de introductie van processen. Mijn opvatting is dat een notie van actieve objecten onontbeerlijk is om een werkelijke integratie met parallellisme te bewerkstelligen.

Het tweede diagram representeert de voorstellen die streven naar een object-georiënteerde uitbreiding van een parallelle logische programmeertaal. Mijn aanpak onderscheidt zich van deze voorstellen door de introductie van expliciete niet-logische variabelen die de toestand van een object representeren. Een meer wezenlijk verschil is dat geen van die voorstellen het backtracken over de antwoorden van een methode-aanroep ondersteunt. Mijn keuze is geweest gedistribueerd backtracken wel te ondersteunen.

Het derde diagram vertegenwoordigt voorstellen waarbij een parallelle object-georiënteerde taal als uitgangspunt is genomen voor de uitbreiding tot een logische programmeertaal. DLP onderscheidt zich van deze voorstellen alleen, wederom, door de mogelijkheid van backtracken over de resultaten van een rendez-vous.

Bij het ontwerp van DLP heb ik gestreefd naar een zo evenwichtig mogelijke combinatie van de verschillende programmeerstijlen, door de driehoeken keer op keer te kantelen. Conceptueel behoort DLP tot de categorie gerepresenteerd door het eerste diagram; wat de implementatie betreft is het derde diagram meer van toepassing; en, zoals ik al heb aangegeven, mijn oorspronkelijke intentie laat zich het best weergeven door het middelste diagram.

Deel II: Semantiek

Formele semantiek biedt een hulpmiddel ter controle van de keuzes gemaakt bij het ontwerp van een programmeertaal. Om erachter te komen of een ontwerp consistent is in de zin dat het geen onmogelijke voorstellen bevat, moet een beschrijving

gegeven worden die precies uitdrukt wat de betekenis van de verschillende constructies is. Formele semantiek is zo'n beschrijvingswijze.

Traditioneel worden twee soorten semantiek onderscheiden: de *operationele* semantiek, die een karakterizering levert van het gedrag van een programma, en de *denotationele* semantiek, die de betekenis van een programma veelal weergeeft als een wiskundige functie.

Aan een denotationele semantiek wordt de eis gesteld dat deze compositioneel is. Compositionaliteit houdt in dat de betekenis van een programma de som is van de betekenis van de delen van het programma. De eis van compositionaliteit wordt over het algemeen niet gesteld aan een operationele semantiek.

Onder *comparatieve* semantiek versta ik het formuleren van zowel een operationele als een denotationele semantiek, en het bewijs van hun equivalentie.

De vraag die ik mij in dit deel gesteld heb is te verwoorden als: hoe modelleer ik het gedrag van DLP programma's?

Hoofdstuk 6: Procescreatie en communicatie in de aanwezigheid van backtracken

De specifieke moeilijkheid waarvoor DLP ons stelt bij het vinden van een semantische beschrijving wordt veroorzaakt doordat fenomenen als dynamische procescreatie en communicatie door rendez-vous optreden in de aanwezigheid van backtracken.

Om deze moeilijkheden het hoofd te kunnen bieden heb ik drie deeltalen onderscheiden waarin het accent ligt op respectievelijk backtracken, procescreatie en communicatie. Voorts heb ik, om de aandacht te richten op de besturingsstructuur, in eerste instantie afgezien van een modellering van het toekennen van waarden aan variabelen.

Operationele semantiek Een operationele semantiek levert de meest natuurlijke beschrijvingswijze, omdat deze op een abstracte wijze de uitvoering van een programma door een machine het dichtst benadert.

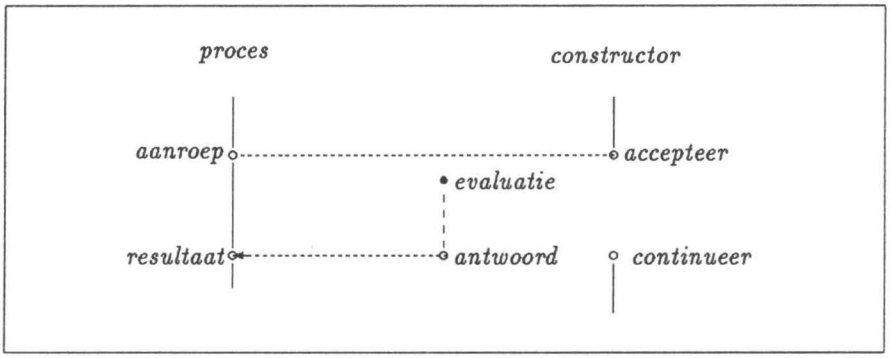
Het gedrag van taalconstructies wordt beschreven door middel van regels die vastleggen welke berekeningsstappen genomen worden bij de evaluatie van een opdracht. De betekenis van een programma is dan te karakteriseren als een opeenvolging van berekeningsstappen. In het geval mogelijke keuzes tot een ander gedrag kunnen leiden karakteriseren we de betekenis door een verzameling die alle mogelijke opeenvolgingen bevat.

Voor het modelleren van het backtrack-gedrag heb ik zogeheten syntactische continuaties gebruikt. Syntactische continuaties werken als een stapel waarin de verschillende te exploreren alternatieven opgeslagen zijn, in de volgorde waarin ze doorzocht moeten worden.

Procescreatie vereist het gebruik van verzamelingen van dergelijke continuaties, ter modellering van de simultane activiteit van actieve objecten.

Voor de modellering van het backtracken dat plaats kan vinden bij het beantwoorden van een methode-aanroep bleek het noodzakelijk *proces-continuaties* te

introduceren. Proces-continuaties zijn op te vatten als een stapel processen waarvan de eerste actief is. De rest van de stapel wordt actief door van het eerste element een zelfstandig proces te maken.



Het bovenstaande plaatje illustreert wat er zoal bij een methode-aanroep gebeurt. Zodra de communicatie tussen het aanroepende proces en het constructor-proces tot stand komt wordt een proces gecreëerd voor het evalueren van de aanroep. De activiteit van het constructor-proces wordt onderbroken en pas hervat zodra de evaluatie een resultaat oplevert. Semantisch wordt dit gemodelleeerd door het evaluatieproces samen met het constructor-proces op een stapel te zetten, met het evaluatieproces als het actieve deel. Zodra een antwoord opgeleverd is wordt het evaluatieproces zelfstandig en kan het constructor-proces zijn activiteit hervatten.

Denotationele semantiek Een denotationele semantiek beoogt niet zozeer het gedrag van een programma te beschrijven als wel de betekenis van een programma in wiskundige zin te karakteriseren.

De wiskundige objecten die een denotationele semantiek aflevert zijn, afhankelijk van de taal die bestudeerd wordt, meer of minder ingewikkelde structuren. Om er zeker van te zijn dat deze objecten wel-gedefiniëerd zijn vereisen we dat ze leven in zogeheten *metrische ruimtes*. Dat maakt dat we op een unieke wijze functies over deze objecten kunnen definiëren.

De functie die het operationele gedrag van een programma karakteriseert is een functie die wiskundige objecten met een vrij eenvoudige structuur oplevert, namelijk opeenvolgingen van berekeningsstappen.

De wiskundige objecten die een denotationele semantiek oplevert zijn over het algemeen een stuk ingewikkelder. Dat komt door de eis van compositionaliteit. Wij wensen dat, denotationeel, de betekenis van het geheel gelijk is aan de som van de betekenis van de delen.

De mogelijkheid van communicatie maakt het noodzakelijk, om aan de eis van compositionaliteit te kunnen voldoen, behalve het feitelijk gedrag ook eventuele alternatieven in het wiskundige object op te nemen, zodat de bereidheid tot communicatie later alsnog gerealiseerd kan worden.

Het grote probleem bij het ontwikkelen van de denotationele semantiek, ook met het oog op de vergelijking met de operationele semantiek, bleek het vinden van semantische tegenhangers voor de syntactische continuaties gehanteerd bij het ontwikkelen van de operationele berekeningsregels.

Voor de afbeelding van het backtracken heb ik gebruik gemaakt van *semantische* continuaties. Semantische continuaties zijn functies die het resultaat van een berekening weergeven afhankelijk van wat nog eventueel volgt. Deze techniek maakt het mogelijk de exploratie van alternatieve oplossingen als continuatieparameter mee te geven.

Procescreatie laat zich modelleren door middel van een operator die de parallelle compositie van twee denotaties definiëert. Bij de creatie van een actief object wordt de denotatie van het gecreëerde object parallel gezet aan de denotatie van het oorspronkelijke object.

Voor de beschrijving van het protocol van wederzijdse uitsluiting van methode-aanroepen heb ik een operator geïntroduceerd die de continuering van de eigen activiteit van het object voegt bij het backtracken over verdere antwoorden.

Comparatieve semantiek Voor de vergelijking met de door de operationele semantiek geleverde betekenis is het nodig te abstraheren van de extra informatie die vanwege de compositionaliteits-eis in de denotatie opgenomen is.

Indien we de operationele betekenis functie $\mathcal{O}$ noemen en de denotationele betekenis $\mathcal{D}$ dan moeten we bewijzen dat

$$\mathcal{O} = \pi \circ \mathcal{D}$$

waarbij π de abstractie is die op de denotaties verkregen door $\mathcal{D}$ toegepast moet worden.

De operationele functie $\mathcal{O}$ is gedefiniëerd als een dekpunt van een hogere orde afbeelding Ψ. Om nu de bovenstaande equivalentie te bewijzen volstaat het te bewijzen dat $\Psi(\pi \circ \mathcal{D}) = \pi \circ \mathcal{D}$, ofwel dat $\pi \circ \mathcal{D}$ een dekpunt is van deze Ψ. Deze bewijstechniek is toepasbaar omdat ook de betekenisfuncties elementen uit een metrische ruimte zijn en de hogere orde functie Ψ een uniek dekpunt heeft.

Hoofdstuk 7: De operationele semantiek van een abstracte versie van DLP

In het voorgaande hoofstuk heb ik de semantiek voor wat beschouwd kan worden als het skelet van de drie deeltalen van DLP gegeven. Als opstapje naar de vergelijkende semantieken voor DLP bespreek ik in hoofdstuk 7 wat er nodig is voor een semantische beschrijving van de andere aspecten van de taal, zoals de toekenning van waarden aan variabelen. De modellering van die aspecten vereist het bijhouden van een toestand, waarin de waarden van variabelen worden opgeslagen door middel van functies. Tevens dient de toestand voor de administratie die nodig is voor het dynamisch creëeren van objecten en processen.

Hoofdstuk 8: Vergelijkende semantieken voor DLP

Op een soortgelijke wijze als in hoofdstuk 6 heb ik de equivalentie bewezen voor operationele en denotationele semantiek van de aangeklede deeltalen van DLP.

Hiermee is aannemelijk gemaakt dat het voorstel voor DLP consistent is, en de taalconcepten van DLP een zinvolle interpretatie toelaten. De beknopte en elegante modellering van het gedistribueerde backtracken dat kan optreden bij een methode-aanroep versterkt het vertrouwen in de validiteit van dat concept.

Deel III: Implementatie

In het feitelijk ontstaan van DLP heeft de implementatie van een prototype een veel belangrijker rol gespeeld dan in het voorgaande tot uitdrukking is gekomen.

De intuïtie voor de constructies zoals die nu in de taal voorkomen is slechts langzaam gegroeid, door het experimenteren met kleine voorbeeldjes en, als gevolg daarvan, het aanpassen of uitbreiden van het prototype. Logisch gesproken is er een duidelijke hiërarchie te onderkennen in de overgang van het formuleren van de syntax voor DLP via de (wiskundige) betekenisgeving daarvan, naar het realiseren van de taal op de machine. In werkelijkheid echter is er tevens een beweging stroomopwaarts, waardoor de realisatie het feitelijk ontwerp van de taal beïnvloedt.

Hoofdstuk 9: Een implementatiemodel voor DLP

Ook in de realisatie van DLP spelen noties als objecten, processen en communicatie een belangrijke rol, maar nu op een implementatieniveau.

Objecten zijn entiteiten die data en clauses bevatten en, onzichtbaar van buitenaf, een interne toestand waarvan het communicatiegedrag afhangt.

Processen worden gecreëerd voor de evaluatie van een doel. Voor ieder object kunnen er een aantal processen actief zijn. Deze hebben alle de niet-logische variabelen en de interne toestand van het object waarmee zij geassocieerd zijn gemeenschappelijk.

Objecten, of precieser, processen communiceren met elkaar door het uitwisselen van boodschappen. De bereidheid van een proces een boodschap te accepteren hangt af van de interne toestand van het object waarmee het proces geassocieerd is.

Het probleem waar wij bij de implementatie van DLP voor staan, is te komen tot een integratie van de implementatiemodellen voor respectievelijk logische programmeertalen, object-georiënteerde programmeertalen en proces-georiënteerde talen die parallellisme ondersteunen.

De lezer zal zich herinneren dat ik DLP bij de beschrijving van het ontwerp heb gekarakteriseerd als een uitbreiding van Prolog met constructies voor parallel object-georiënteerd programmeren. Een dergelijke werkwijze is ook voor de implementatie denkbaar. Ik heb er echter de voorkeur aan gegeven uit te gaan van een bestaande parallelle object-georiënteerde taal en deze op te tillen naar het niveau van een logische programmeertaal. Deze werkwijze had als voordeel dat ik gebruik kon maken van de primitieven voor procescreatie en communicatie voorhanden in de implementatietaal.

Hoofdstuk 10: Het afleiden van een Prolog interpreter

Het bijzondere van het prototype is dat de implementatie van de Prolog interpreter zelf weer afgeleid is van een denotationele (continuatie) semantiek. Idealiter zou dit een van de semantieken zijn zoals behandeld in deel II. Feitelijk is dit niet het geval, eenvoudigweg omdat ik destijds nog niet de beschikking had over die semantieken.

Hoofdstuk 11: De implementatie van het prototype

Een belangrijk probleem dat bij het implementeren van een op de logica gebaseerde programmeertaal optreedt is de representatie van termen.

Termen Termen zijn de syntactische elementen die zowel variabelen en waarden die daaraan toegekend worden kunnen vertegenwoordigen, alsook delen van het programma zelf, zoals clauses en doelen. In het geval van DLP moet er rekening mee gehouden worden dat variabelen kunnen verwijzen naar dynamisch gecreëerde objecten en processen.

Objecten Hoewel uitgegaan is van een object-georiënteerde implementatie-taal bleek het nodig het management van DLP objecten zelf te implementeren. Een groot deel van de inspanning is gaan zitten in de implementatie van het protocol van acceptatie en wederzijdse uitsluiting van methode-aanroepen.

Processen Processen zijn geassocieerd met een object. Het object moet in staat zijn het gedrag van een proces te controleren. Een proces biedt een dergelijke synchronisatiemogelijkheid door middel van een zogeheten *lock* waarmee de activiteit van het proces stilgelegd kan worden.

Systeem Opdrachten in DLP die resulteren in de creatie van nieuwe objecten of het aangaan van een rendez-vous worden geïnterpreteerd door de systeemprimitieven van DLP. Deze systeemprimitieven zijn voor een DLP object beschikbaar door het te laten erven van het object *systeem*.

Hoofdstuk 12: Conclusies en toekomstig onderzoek

Met de taal DLP heb ik een bijdrage geleverd aan het bestand van programmeertalen dat in de loop der tijd ontwikkeld is.

Het onderscheidende kenmerk van DLP, de mogelijkheid van backtracken over de resultaten van een methode-aanroep, is vanuit verschillende perspectieven uitvoerig bestudeerd. Behalve aan mogelijke ontwerpalternatieven is ook aandacht besteed aan de semantische modellering van dit fenomeen.

Voor het maken van de uiteindelijke ontwerpkeuzes heb ik veel profijt gehad van het door mij ontwikkelde prototype.

Met mijn proefschrift beschouw ik de eerste fase van het onderzoek naar gedistribueerd logisch programmeren als voltooid.

Voor de toekomst zou ik graag zien dat er gewerkt wordt aan een efficiëntere implementatie van DLP. Voorts lijkt het me van belang dat er in aanvulling op de gedragssemantiek die ik hier heb gegeven, aandacht besteed wordt aan de logische betekenis van de taalconstructies van DLP.

Het zou jammer zijn als de afronding van mijn proefschrift tevens het einde van DLP zou betekenen. De toepassing van DLP voor het oplossen van problemen in de software engineering en kunstmatige intelligentie ligt voor de hand.

Appendix: Het prototype

Een bijna volledige listing van het prototype is opgenomen in de appendix. Bij het ontwikkelen van het prototype heeft het streven naar leesbare code een belangrijke rol gespeeld. Het gebruik van een formele semantiek als uitgangspunt voor een deel van de code was slechts één van de hulpmiddelen. De gemechaniseerde hulp van een tekstverwerker was daarbij naar mijn mening niet minder belangrijk.