MODES AND TYPES IN LOGIC PROGRAMMING

A THESIS SUBMITTED TO
THE UNIVERSITY OF KENT AT CANTERBURY
IN THE SUBJECT OF COMPUTER SCIENCE
FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY.

By
Jan-Georg Smaus

December 1999

Abstract

This thesis deals with two themes: (1) construction of abstract domains for mode analysis of typed logic programs; (2) verification of logic programs using non-standard selection rules.

(1) Mode information is important mainly for compiler optimisations. The precision of a mode analysis depends partly on the expressiveness of the abstract domain. We show how specialised abstract domains may be constructed for each type in a typed logic program. These domains capture the degree of instantiation of a term very precisely. The domain construction procedure is implemented using the Gödel language and tested on some example programs to demonstrate the viability and high precision of the analysis.

(2) We provide verification methods for logic programs using selection rules other than the usual left-to-right selection rule. We consider five aspects of verification: termination; and freedom from (full) unification, occur-check, floundering, and errors related to built-ins. The methods are based on assigning a mode, input or output, to each argument position of each predicate. This mode is only fixed with respect to a particular execution. For termination, we first identify a class of predicates which terminate under the assumption that derivations are *input-consuming*, meaning that in each derivation step, the input arguments of the selected atom do not become instantiated. Input-consuming derivations can be realised using `block` declarations, which test that certain argument positions of the selected atom are non-variable. To show termination for a program where not all predicates terminate under the assumption that derivations are input-consuming, we make the stronger assumption that derivations are *left-based*. This formalises the "default left-to-right" selection rule of Prolog. To the best of our knowledge, this work is the first formal and comprehensive approach to this kind of termination problem. The results on the other four aspects are mainly generalisations of previous results assuming the left-to-right selection rule.

Acknowledgements

I am grateful to the many people who helped me write my PhD thesis.

The research of this thesis was partly carried out in the project ‘Detecting and Exploiting Determinacy in Logic Programs’ led by Andy King at the University of Kent at Canterbury and Pat Hill at the University of Leeds. Andy King was also my PhD supervisor. I would like to thank Andy and Pat for their critical guidance and friendly encouragement.

I was very fortunate to be a member of the logic programming group at the Universities of Leeds and Kent. I would like to thank Florence Benoy, Andrew Heaton, Jacob Howe and Jonathan Martin for the productive and enjoyable time we spent together.

I also thank Eerke Boiten, Naomi Lindenstrauss, Fred Mesnard and Erik Poll, who proofread parts of my thesis. Maarten Steen has given me much advice on organisational matters.

Many colleagues have inspired my work. I would like to mention Krzysztof Apt, Tony Bowers, Henning Christiansen, Michael Codish, Bart Demoen, Sandro Etalle, Fergus Henderson, Lee Naish and Salvatore Ruggieri.

I gratefully acknowledge the financial support from the EPSRC and the Computing Laboratory of the University of Kent at Canterbury. During the first two years of my PhD studies I was employed as a research associate under EPSRC Grant No. GR/K79635, and in the remaining time, I received an E. B. Spratt Bursary.

I made many friends during the time I spent in Canterbury, most of them through my ‘lunch group’ or through the orchestra of Kent University. They have made these three and a half years a wonderful time. When I leave Canterbury this summer, I will become another outpost of this circle of friends which already spans all over Europe, and even further.

I am grateful to my family for their continuing support. Last but not least, I thank Bénédicte, my muse.

Preface

Modes and types are two widely used concepts in analysis and verification of logic programs. On the analysis side, modes and types allow us to infer information about the program which is useful for compiler optimisations, helping to generate more efficient code. On the verification side, modes and types allow us to prove a number of desirable properties of the program, such as occur-check freedom and termination. Some logic programming languages even go as far as only admitting programs that meet certain mode and type requirements. This has great benefits in terms of efficiency and reliability of software.

The separations between the above areas are not clearcut. Moreover, the notions of mode and type have differing meanings depending on the context in which they are used. There is a whole spectrum of such meanings.

This thesis treats two substantially different themes. However, both are related to modes and types. The two themes are

- the construction of abstract domains for mode analysis of typed logic programs,
- the verification of logic programs for non-standard selection rules.

Modes and types have quite different, albeit certainly related, meanings for the two themes. Within each theme, our usage of these notions follows widespread conventions. To avoid confusion, it seems therefore reasonable to keep the two themes clearly separated.

This gives rise to the following structure of this thesis. The thesis has three parts: an introductory part and two parts corresponding to the two themes. Part I is divided into two chapters. Chapter 1 consists of two separate introductions for Parts II and III. Chapter 2 puts the two themes into context by giving an overview of the whole spectrum of mode and type concepts that are used in the literature, which encompasses the concepts used in this thesis.

The work presented in Part II has been accepted for presentation at the 9th International Workshop on Logic-Based Program Synthesis and Transformation (LOPSTR'99) [SHK99a]. The work presented in Part III is based on three conference papers [SHK98, SHK99b, Sma99], two of which the author has written together with Pat Hill and Andy King.

Contents

Part I

Introduction and Background

Chapter 1

Introduction

In this chapter, we will give two separate introductions for Parts II and III, respectively. As mentioned in the preface, both parts make use of notions of mode and type, but they use these notions in quite different ways.

In Part II, a *mode* is a characterisation of the degree of instantiation of a term. A *type* is a set of terms defined by means of a declaration, as provided in typed logic programming languages such as Gödel [HL94] or Mercury [SHC96].

In Part III, a *mode* is a specification of each argument position of each predicate in a program as either input or output. A *type* is any set of terms closed under instantiation.

In Chapter 2, we will consider the relationships between these notions in more detail.

1.1 Mode Analysis for Typed Logic Programs

In Part II we provide a generic method for constructing abstract domains for mode analysis of typed logic programs. A mode is a characterisation of the degree of instantiation of a term at a certain point in the execution of a program. Mode analysis is concerned with finding the modes of a program. We now present an introduction to mode analysis using abstract domains and then proceed to the actual contribution of Part II.

1.1.1 Previous Work

The following example illustrates the notions of *degree of instantiation* and *point in the execution.*

Example 1.1 Consider the following program[1] for the append predicate.

```
append(Xs,Ys,Zs) :-
  Xs = [],
  Ys = Zs.
append(Xs,Ys,Zs) :-
  Xs = [X|Xs1],
  Zs = [X|Zs1],
  append(Xs1,Ys,Zs1).
```

[1] The program is in so-called *normal form*, defined in Section 3.3.

```
append(Xs,Ys,Zs) :-                ground(ground).
  ground(Xs),
  iff(Ys,Zs).                      iff(X,X).
append(Xs,Ys,Zs) :-
  iff_and(Xs,X,Xs1),               iff_and(ground,ground,ground).
  iff_and(Zs,X,Zs1),               iff_and(any,ground,any).
  append(Xs1,Ys,Zs1).              iff_and(any,any,ground).
                                   iff_and(any,any,any).
```

Figure 1: An abstraction of `append`

When we assume an initial query `append`([1, 2], [3, 4], `Cs`) and the standard left-to-right selection rule, then we can say that at each point in the execution just before an atom `append`(s, t, u) is called, s and t have the following degree of instantiation: they are ground. Moreover, for every computed answer, `Cs` is instantiated to a ground term. ◁

Information as in the above example can be derived using abstract interpretation [CC77]. Here we will look at a particular technique of abstract interpretation called abstract *compilation* [CD94, CD95, DW86, HWD92], meaning that an abstract program is evaluated using a concrete semantics.

Example 1.2 Corresponding to the program in Example 1.1 is the abstract program shown in Figure 1. Note that the abstract program is obtained by replacing all unifications in the concrete program with calls to `ground`, `iff` and `iff_and`. These calls are called *abstract unifications.* The abstract unifications operate on abstract terms `any` and `ground`, where `ground` represents a term that is definitely ground and `any` represents any term. For example, `iff_and`(s, t, u) expresses that s is a ground term if and only if t and u are both ground terms. This reflects that on the concrete level, a list is ground if and only if its head and tail are ground.

When we assume an initial call `append(ground,ground,_)`, all calls to `append` in this abstract program will have the term `ground` in the first two arguments, and the only answer for `append` is `append`(`ground`, `ground`, `ground`). It has been shown by Codish and Demoen [CD95] that from this, it can be concluded that in the concrete program, all calls to `append` have ground terms in the first two arguments, and all answers to `append` have ground terms in all arguments — just as was observed in Example 1.1. ◁

The technique of the above example has been developed further [CD94] to derive groundness dependencies with more detail, using a more or less ad-hoc notion of type. This is shown in the following example. Note that we are still assuming untyped languages.

Example 1.3 Figure 2 shows an alternative abstraction of the program in Example 1.1. Without worrying about the details, observe that the abstract terms used in this abstraction would be terms such as `integer`, representing an integer,[2] `list`(`integer`), representing a nil-terminated list of integers, `list(any)`, representing a nil-terminated list whose elements could be arbitrary terms, and `any`, representing any term.

[2] Integers are just used as an example here.

```
append(Xs,Ys,Zs) :-
  nil_dep(Xs),
  iff(Ys,Zs).
append(Xs,Ys,Zs) :-
  cons_dep(Xs,X,Xs1),
  cons_dep(Zs,X,Zs1),
  append(Xs1,Ys,Zs1).
```

```
nil_dep(list(bot)).

iff(X,X).

cons_dep(list(A),B,list(C)) :-
  lub(A,B,C).
cons_dep(any,_,C) :-
  C \== list(_).

lub(A,A,A).
lub(A,A,bot).
lub(A,bot,A).
lub(any,A,B) :- A \== B.
```

Figure 2: An alternative abstraction of `append`

The concrete unification `Xs` = [`X`|`Xs1`] is abstracted as `cons_dep(Xs,X,Xs1)`, which relates an abstract term for the list `Xs` with the abstractions of its head `X` and its tail `Xs1`. For example, if `X` is `integer` and `Xs1` is `list`(`integer`), then `Xs` would be `list`(`integer`). If however `X` is `any` and `Xs1` is `list`(`integer`), then `Xs` would be `list`(`any`).

For example, assume an initial call `append(list(any),list(any),_)`, meaning that `append` is called with the first two arguments being instantiated to lists. Then all calls to `append` in this abstract program will have `list`(`any`) in the first two arguments, and the only answer for `append` is `append`(`list`(`any`), `list`(`any`), `list`(`any`)). For the concrete program, this implies: if `append` is called with the first two arguments being lists, then all subsequent calls to `append` also have lists in the first two arguments, and all answers to `append` have lists in all arguments. Similarly, we could infer: if `append` is called with the first two arguments being lists *of integers*, then all subsequent calls to `append` also have lists of integers in the first two arguments, and all answers to `append` have lists of integers in all arguments. ◁

Clearly, in order to abstract the `append` program as in the above example, one has to know what a *list* is. The definition of a list underlying the above example is the standard one: for any type τ, `nil` is of type `list`(τ); moreover, if h is of type τ and t is of type `list`(τ), then `cons`(h, t) is of type `list`(τ). Codish and Demoen [CD94] are not concerned with how such definitions could be derived in general, but only deal with a specific set of types including integers, lists, difference lists, and trees, and provide the definitions of the abstractions, such as the definition of `cons_dep` in the above example. Of course, this set includes the most frequently used types and therefore much useful information can already be inferred.

1.1.2 Exploiting Type Declarations

In typed logic programming languages, all types are defined by a type declaration. For example, in Gödel, the type of lists is defined as follows.

```
CONSTRUCTOR     List/1.
CONSTANT        Nil: List(u).
FUNCTION        Cons: u * List(u) -> List(u).
```

The first line defines a type constructor `List` with one type parameter. We say that `List(u)` is a polymorphic type, where `u` is a type parameter. In Sections 3.2 and 3.3, we explain the syntax of Gödel in more detail, but this example should be self-explanatory. Throughout the rest of this section, we will use Gödel type declarations to define types.

In Part II, we describe a method which takes a program, say the **append** program, including the type declarations, and generates an abstract program similar to the one in Example 1.3. In particular, the method generates the dependency predicates such as **cons_dep**, whose construction seems quite ad-hoc in the work of Codish and Demoen, since they are considering untyped languages.

To understand why this work is a proper generalisation of the work of Codish and Demoen [CD94] and also Codish and Lagoon [CL96], we must look at some more complex types. It is not surprising that when one introduces an ad-hoc notion of types into an untyped programming language, one is unlikely to deal with types that are more complex than, essentially, lists and trees. This is different when one considers typed languages, as we do in Part II.

First consider the following type declarations

```
BASE            IntegerList.
CONSTANT        Nil: IntegerList.
FUNCTION        Cons: Integer * IntegerList -> IntegerList.
```

These declarations define the type of integer lists, where we assume that `Integer` is the usual built-in type. Note that `IntegerList` contains exactly the same terms as `List(Integer)`, and therefore it is reasonable to expect that the abstract domain characterising the degree of instantiation of terms of type `IntegerList` should be the same as the one sketched in Example 1.3. In our formalism, this is indeed the case.

Our formalism is based on a relation on types called "is a subterm type of". `Integer` and `IntegerList` are both subterm types of `IntegerList`, meaning that a term of type `IntegerList` can have subterms of type `Integer` and subterms of type `IntegerList`. If σ is a subterm type of τ, and τ is *not* a subterm type of σ, then we say that σ is a *non-recursive* subterm type of τ. If σ is a subterm type of τ, and τ is a subterm type of σ, then we say that σ is a *recursive* type of τ. `Integer` is a non-recursive subterm type of `IntegerList`, and `IntegerList` is a recursive type of `IntegerList`.

The relation "is a non-recursive subterm type of" is a generalisation of the relation "is a parameter of" which underlies the domain construction of Example 1.3. One can argue that the type `IntegerList` has no raison d'être since it is better to use the instance `List(Integer)` of the polymorphic type `List(u)`. However, we shall see other examples of a non-recursive subterm type not being a parameter.

Example 1.4 As another example, consider a family tree.

```
CONSTRUCTOR  Family/1.
FUNCTION     Person: u * List(Family(u)) -> Family(u).
```

For a person, we may want to store the name, the age, or any other attribute. The first argument of `Person` is used for this purpose. Moreover, we want to store the list of children of this person, that is, a list of family trees, one for each child. As an example, consider `Family(String)`. Our formalism constructs an abstract domain for this type which can characterise that all the "names" in the term

```
Person("Lisa",[Person("Frank",[]),Person("Sara",[])])
```

are instantiated, whereas this is not true for the term

```
Person("Lisa",[Person(x,[]),Person(y,[])]).
```

◁

The methods of Codish and Demoen [CD94] and Codish and Lagoon [CL96] cannot deal with the above examples. We will see more examples in Part II.

The abstract domains used in our mode analysis are entirely in the spirit of previous work [CD94, CL96], and the inherent complexity of our mode analysis is therefore similar. In general, the complexity of a mode analysis depends on the complexity of the type declarations. We will argue that the formalism presented in Part II provides the highest degree of precision that a generic domain construction should provide. It also helps to understand other, more ad-hoc and pragmatic domain constructions as instances of a general theory. One could always simplify or prune down (widen) the abstract domains for the sake of efficiency.

Our method has been implemented in Gödel for Gödel programs. We show for some example programs that the analysis times compare well with a domain that only distinguishes ground and non-ground terms [CD95].

1.2 Non-Standard Derivations

Part III is concerned with verification methods for logic programs that use *non-standard* derivations, that is, they use a selection rule other than the usual left-to-right selection rule of Prolog. We consider five aspects of verification: termination, unification freedom[3], occur-check freedom, flounder freedom, and freedom from errors related to built-ins.

Non-standard derivations are useful for a variety of purposes: multiple modes, parallel execution [AL95], the test-and-generate paradigm [Nai92], and certain uses of accumulators [EG99].

[3] A program is called *unification free* if it only requires (double) matching instead of the full unification procedure.

For verification of logic programs [AE93, AL95, BC99, EBC99], in particular programs using non-standard derivations, it has been shown to be useful to assign a mode (input or output) to each argument position of each predicate, and require certain correctness properties concerning those modes. We will adopt some correctness properties that have previously occurred in the literature and also introduce some new ones.

Considering non-standard derivations does not imply that any atom in a query can be selected for resolution at any time. For some aspects of verification, such as termination or freedom from errors related to built-ins, it is necessary to ensure a certain degree of instantiation of an atom before that atom is selected [AL95]. We will argue that a reasonable minimal assumption is that derivations are *input-consuming*, that is, an atom is only selected once it is sufficiently instantiated in its input arguments, so that unification with a clause head does not instantiate these arguments any further.

Input-consuming derivations have not been defined in this form previously, although the concept is related to (F)GHC [Ued86] and non-destructive programs [ER98]. This is discussed in Section 11.1.

In existing implementations, input-consuming derivations can be ensured by *delay declarations* [HL94, SIC98, SHC96]. Using delay declarations, an atom in a query is selected only if its arguments are instantiated to a specified degree. In particular, we will consider `block` declarations. These are a simple kind of delay declaration where only tests for partial instantiation are possible, but not, for example, tests for groundness.

Hence Part III of this thesis is aimed at verifying programs with delay declarations, but we try to take the more abstract view and formulate results in terms of input-consuming derivations wherever possible. This view has not been taken by other authors previously [AL95, Lüt93, MT95, MK97, Nai92].

We now give an overview of Part III. Note first the following general points:

- Section 5.2 defines most of the notation and terminology. Sections 7.2 and 8.2 introduce some further terminology related to delay declarations. In any case, the index can be used to find the place where notation or terminology is introduced.

- Section 11.1 is devoted to the literature related to Part III. However, the related literature is also considered throughout the rest of Part III wherever useful for motivation or illustration.

1.2.1 Correctness Properties of Programs

In Chapter 5, we introduce a number of correctness properties concerning the modes of a program. The following example gives a flavour of these properties.

Example 1.5 Consider the usual append/3 program (it will be given in Figure 10 on page 57), where the first two arguments are input and the third is output. The query

$$\texttt{append}([1],[2],\texttt{Xs}),\ \texttt{append}([3],[4],\texttt{Ys}),\ \texttt{append}(\texttt{Xs},\texttt{Ys},\texttt{Zs})$$

is “well-behaved” in that it meets all correctness properties we introduce.

In particular, note that the third atom has variables `Xs` and `Ys` in input positions, and that these variables occur elsewhere in output positions. In other words, every variable has a *producer*. Moreover, note that Xs and Ys occur each only once in an output position. In other words, every variable has *at most* one producer. Finally, note that for each variable, the output occurrence precedes any input occurrence. If we assumed the left-to-right selection rule, this could be interpreted as follows: every piece of data is produced before it is consumed.

Having *at most* one producer is the main aspect of a well-known correctness property called *nicely-modedness*, and having *at least* one producer is the main aspect of an equally well-known correctness property called *well-modedness*. In contrast, the query

$$\texttt{append}([1],[2],\texttt{Xs}),\texttt{append}([3],[4],\texttt{Xs}),\texttt{append}(\texttt{Xs},\texttt{Ys},\texttt{Zs})$$

is not nicely moded because there are two output occurrences of Xs, and it is not well moded because there is no output occurrence of Ys. ◁

As can be seen in the above example, the correctness properties are traditionally defined assuming that there is a left-to-right data flow in a query (or clause body) [AE93, AL95, AM94, AP94b, BC99, EBC99, EG99]: every atom only uses as input data that was produced by other atoms occurring to the left. With such a restricted view, it is not possible to reason about programs where the textual order of atoms differs from the data flow. We will therefore generalise these properties by considering them up to permutation of a query. For example, a query is *permutation* nicely moded if some permutation of it is nicely moded.

1.2.2 Termination of Input-Consuming Derivations

Input-consuming derivations formalise the natural meaning of input. For most programs, assuming input-consuming derivations is necessary for termination. For example, it is easy to see that given the usual append program, an infinite derivation for the query

$$\texttt{append}([1],[],\texttt{As}),\texttt{append}(\texttt{As},[],\texttt{Bs})$$

is obtained by always selecting the rightmost atom (see Figure 10 on page 57).

This raises the question whether assuming input-consuming derivations is sufficient to ensure termination. In Chapter 6, we define a class of predicates for which this is indeed the case. We present a method for showing that a predicate is in this class. This method is based on level mappings, closely following the traditional approach for derivations using the standard left-to-right selection rule [EBC99].

Note however that the class of predicates for which all input-consuming derivations are finite is quite limited. Relying on this assumption alone cannot be a comprehensive method of showing termination for realistic programs. This is also the reason why we speak of a class of *predicates* and not of a class of programs. Within one program, some predicates may be in that class and some may not.

1.2.3 Ensuring Input-Consuming Derivations

In Chapter 7, we show how `block` declarations, which are a particularly simple and efficient kind of delay declaration, can be used to ensure that derivations are input-consuming. The `block` declarations declare that certain arguments of an atom must be *non-variable* before that atom can be selected for resolution.

Usually, one would have `block` declarations such that an atom is only selected when its input positions are non-variable. However, this is sometimes not sufficient. Suppose we have a predicate `p/1` whose argument is input, and "`p(f(1)).`" is a clause defining this predicate. The atom `p(f(X))` is non-variable in its input position. Nevertheless its selection would violate the requirement of an input-consuming derivation, since unification with the clause head instantiates `X`. This and similar problems give rise to the definition of two further correctness properties for programs. Despite these problems, `block` declarations are adequate for ensuring input-consuming derivations in existing implementations.

Previous literature on delay declarations has not recognised that the simplicity and efficiency of `block` declarations give them a special role. There has never been a systematic account of when `block` declarations are sufficient to ensure any desired properties such as termination, and when more complex constructs, say groundness checks, are needed.

1.2.4 Termination and `block` Declarations

In Chapter 8, we present two approaches to showing or ensuring termination for programs with `block` declarations. As suggested above, it is often necessary to make stronger assumptions about the selection rule rather than just to assume that derivations are input-consuming. We do so by assuming *left-based* derivations. This formalises the "default left-to-right" selection rule of most existing Prolog implementations.

The first approach is relatively simple and tries to eliminate the well-known problem of *speculative output bindings* [Nai92]. The approach consists of two complementary methods: one exploits the fact that a program does not use any speculative bindings; the other exploits the fact that a program does not make any speculative bindings.

The idea of the second approach is as follows: first, `block` declarations must be used to ensure that derivations are input-consuming. Some predicates are known, by Chapter 6, to terminate for all input-consuming derivations. For all other predicates, the textual position of atoms using those predicates must be taken into account. More precisely, the latter atoms must be placed sufficiently late, which ensures that they are only selected once their input is completely instantiated.

Example 1.6 The following clause is part of a program for the well-known n-queens problem. It is an example of the *test-and-generate* paradigm.

```
nqueens(N,Sol) :-
  sequence(N,Seq),
  safe(Sol),
  permute(Sol,Seq).
```

A solution to the n-queens problem is encoded as a permutation of the list $[1, \ldots, n]$, which represents the position of the queen in each row of the chess board. The predicate `sequence` generates the list $[1, \ldots, n]$. This list is then permuted using `permute` and the solution candidates are tested for being legal configurations by the predicate `safe`. The call to `safe` occurs before the call to `permute` to achieve coroutining of the two atoms `safe(Sol)` and `permute(Sol,Seq)`.

In the clause body, the call to `permute` is placed sufficiently late. Assuming left-based derivations, this means that when `permute` is called, the input `Seq` is ground. With less instantiated input, termination of `permute` could not be guaranteed. In contrast, the predicate `safe` will frequently be called with partially instantiated lists as input. However, this is not a problem because, as we will see, the assumption of input-consuming derivations is sufficient to ensure termination of `safe`. ◁

Chapter 8 formalises and extends heuristics that have previously been proposed to ensure termination of programs with `block` declarations under the assumption of a default left-to-right selection rule [Nai92]. In this informal work, even the selection rule itself is not formalised.

Most approaches to the termination problem for programs using non-standard derivations abstract from the relevance of the textual order of atoms for the selection rule. These approaches must either yield relatively weak results, or strengthen the assumptions about the selection rule in some other way rather than assuming the default left-to-right selection rule [AL95, Lüt93, MT95, MK97].

1.2.5 Further Aspects of Verification

In Chapter 9, we study some further aspects of verification of logic programs using non-standard derivations.

The first aspect is freedom from unification. This means that the unification procedure can be replaced with so-called *double matching*. The idea is that when a selected atom in a query is unified with the head of a clause, the input arguments of the clause head are first bound to the input arguments of the selected atom. This fits with the idea that derivations are input-consuming, since it means that the input arguments of the selected atom are not instantiated. Afterwards, the output arguments of the selected atom are bound to the output arguments of the clause. We will see that under certain conditions, programs are free from unification.

The second aspect is freedom from occur-check. It is well-known that the unification algorithm used in existing logic programming systems leaves out the occur-check for efficiency reasons. We show that for programs meeting certain correctness conditions, namely *permutation nicely moded, input-linear* programs, the occur-check can safely be omitted.

The third aspect is freedom from floundering. A derivation *flounders* if it ends with a non-empty query where no atom is sufficiently instantiated to be selected in accordance with the `block` declarations. Freedom from floundering is an important aspect of verification mainly because of its relationship to termination. In principle, termination and flounder freedom are conflicting aims. Clearly, termination could trivially be ensured by having `block` declarations such that no atom can ever be selected, which means that all derivations would flounder immediately. We show however that under reasonable assumptions, namely that programs are *permutation well typed*, no derivations flounder. This implies that our methods for showing termination in no way rely on trivial termination by floundering.

As the last aspect, we consider freedom from errors related to built-ins. These are type errors, arising from calls like `X is foo`, or instantiation errors, arising from calls like `X is V`. One previous proposal for preventing such errors uses well typed programs and delay declarations to ensure that built-ins are only called when their input arguments are ground [AL95]. Unfortunately, `block` declarations cannot test (directly) whether an argument is ground. The main contribution of Section 9.4 is to show that under certain conditions, `block` declarations are nevertheless sufficient. The method is based on *constant types*, that is types consisting only of constants. The most prominent examples would be *integer* or other numeric types. We exploit the fact that for a term of constant type, being non-variable implies being ground.

1.2.6 Weakening Some Conditions

In Chapter 10, we consider ways of simplifying the `block` declarations by omitting tests that can be proven at compile time to be always met at runtime. This is particularly useful for built-ins, since there is usually no direct way of having delay declarations for those. We will also consider ways of weakening a restriction imposed for many results in Part III, namely that the input arguments of each clause head contains no variable more than once. This restriction is quite severe in that it prevents two input arguments being tested for equality. Moreover, we consider a generalisation of the notion of a mode of a program, allowing for a predicate to be used in different modes even within a single execution of the program.

1.2.7 Related Work and Conclusion

Chapter 11 takes a look at the literature related to Part III. It then discusses some ideas and features that are distinctive of this work, as well as some open problems. Finally, it concludes the thesis with a summary of Part III.

Chapter 2

Notions of Modes and Types

This chapter gives an overview of mode and type concepts used in the literature, encompassing the uses made of these concepts in this thesis. In Section 2.1, we consider modes, in Section 2.2, we consider types, and in Section 2.3, we consider ways of combining the two concepts. Finally in Section 2.4, we recall very briefly the concepts of modes and types as used in this thesis.

2.1 Modes

One of the distinctive features of logic programming, as opposed to other programming paradigms, is that there is no a priori notion of input and output. *The same program can be used to compute answers to different problems* [Apt97]. The following example illustrates this.

Example 2.1 A program like the following is the standard example to introduce logic programming to novices [Apt97, SS86].

```
direct_flight(rome, london).
direct_flight(paris, london).
direct_flight(paris, rome).
direct_flight(london, bristol).

connection(X, Y) :-
  direct_flight(X, Y).
connection(X, Y) :-
  direct_flight(X, Z),
  connection(Z, Y).
```

This program can be used to answer questions of different kinds.

- Is there a flight connection from Rome to Bristol?

  ```
  | ?- connection(rome, bristol).
  yes
  ```

- To which cities are there flight connections from Rome?

```
| ?- connection(rome, City).
City = london ? ;
City = bristol ? ;
no
```

- From which cities are there flight connections to Rome?

```
| ?- connection(City, rome).
City = paris ? ;
no
```

- Where do I change planes flying from Paris to Bristol?

```
| ?- direct_flight(paris, City), direct_flight(City, bristol).
City = london ? ;
no
```

These different ways of using a logic program are usually referred to by saying that the program is used in different *modes*. For example, consider the second query above. The first solution to this query is computed by the following derivation:

$$\texttt{connection}(\texttt{rome},\texttt{City}) \leadsto \texttt{direct\_flight}(\texttt{rome},\texttt{City}) \leadsto \Box.$$

One way of characterising this derivation is by saying that the first argument positions of `connection` and `direct_flight`, respectively, are used as *input* positions, whereas the second positions are used as *output* positions.

Another way of characterising this is by saying that `connection`(`rome`, `City`) and `direct_flight`(`rome`, `City`) are *call* patterns in this derivation, whereas `connection`(`rome`, `london`) and `direct_flight`(`rome`, `london`) are *answer* patterns. Or more abstractly, `connection`(*ground*, *free*) and `direct_flight`(*ground*, *free*) are *call* patterns, whereas `connection`(*ground*, *ground*) and `direct_flight`(*ground*, *ground*) are *answer* patterns.

For the last query, assuming the standard left-to-right selection rule, we might also say that the first atom is a *producer* of `City` and the second atom is a *consumer* of `City`.

All those characterisations suggest that modes are inextricably linked to the procedural rather than the declarative view of logic programming. However, it is also possible to take a *declarative* view of modes [Nai96], as we will discuss in Subsection 2.3.2.

We will now shed some light on different notions of modes occurring in the literature by comparing them under two criteria. The first criterion is how prescriptive or descriptive the notion of modes is. The second criterion is the granularity with which modes are characterised.

2.1.1 Descriptive versus Prescriptive Modes

This criterion is closely linked to the question: In which context and for which purpose are modes used? Figure 3 shows a rough subdivision of the literature into three groups.

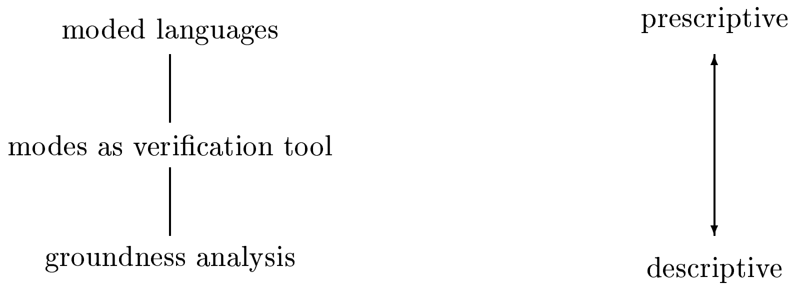

Figure 3: Descriptive versus prescriptive modes

Groundness analysis

Mode *analysis*, more often called *groundness analysis*, is concerned with the question "at a given program point, what is the degree of instantiation of variable x?", and in particular, "is x bound to a ground term[1]?". Such information is useful for compiler optimisations such as the specialisation of unification, but also because it improves the precision of other analyses [MS93]. It is also important for termination analysis [LS96, LS97]. Much research has been done on groundness analysis [AMSH94, AMSH98, BCHK97, Cod97, CBGH97, CDY94, CD94, CD95, CGBH94, CL96, GGS99, HHK97, HACK00, KSH99, MS93, TL97].

For the derivation on the facing page, it can be inferred that at the point just before `direct_flight` is called, the first argument of `direct_flight` is a ground term, and at the point after `direct_flight` is resolved, the second argument is also a ground term.

In this context, "mode" is a descriptive concept, that is, no assumptions are made about how programs are — or should be — written. The analysis takes an arbitrary program and *describes* the modes of this program. This is usually done using abstract interpretation [CC77]. Since groundness is an undecidable property, this description can only be approximate. For some program points an analysis might be able to infer that a variable is bound to a ground term, but it cannot decide the groundness of every variable for every program point.

One usually distinguishes goal-dependent and goal-independent groundness analyses [CBGH97, CDY94, CGBH94, MS93]. In the former, one assumes that the program is executed with an initial goal that is instantiated to a certain degree. This introduces a slight prescriptive aspect into groundness analysis, since it assumes that programs should be used in a certain way. Most of the literature on groundness analysis however is relevant for goal-dependent and goal-independent groundness analyses alike.

Part II is about the construction of abstract domains for groundness analysis. In the implementation, these domains are used for goal-dependent groundness analysis.

[1] A term is called *ground* if it does not contain variables.

Modes as verification tool

Modes have been used for a variety of verification purposes [AM94, EG99]. For example, they have been used to show that programs are occur-check free [AL95, AP94b], unification free [AE93], successful [BC99], and terminating [EBC99]. Here it is assumed that each argument position of each predicate is either input or output, and that the program and initial goal fulfill certain correctness properties such as being well moded or nicely moded. Usually, this approach is not concerned with how these modes are determined.

For the derivation on page 14, one would say that for both predicates, the first argument is input and the second is output, which can be denoted by writing the mode of the program as $\{\texttt{connection}(I, O), \texttt{direct\_flight}(I, O)\}$.

In this context, "mode" is a fairly prescriptive concept, since assumptions are made about how programs should be written and used. If a program does not adhere to the correctness property required for a certain verification purpose, the verification method is not applicable. Part III of this thesis uses modes to verify properties such as termination and occur-check freedom.

Moded languages

The most prescriptive approach to modes is to use a moded language, for example Mercury [Hen92, SHC96]. In Mercury, the user has to *declare* the mode of some predicates, while the mode of others is inferred. The program has to fulfill certain correctness properties concerning these modes. Otherwise it is not accepted by the compiler.

These correctness properties restrict the class of legal programs and hence to a certain extent limit the expressiveness of a language. On the other hand, as Mercury shows, they allow the compiler to generate very efficient machine code.

2.1.2 The Granularity

We now distinguish different mode concepts by another criterion: the granularity of the formalism to characterise the instantiation of a term, or in other words, the degree of precision with which the instantiation of a term can be characterised. Note that for this criterion, we cannot easily draw a picture like the one in Figure 3 on the preceding page, since there is no such obvious hierarchy. We distinguish between two-valued and more fine-grained characterisations.

Two-valued characterisations

The lowest granularity is given when we have a characterisation which can only take two possible values. Most groundness analyses only distinguish ground and possibly non-ground terms [AMSH94, AMSH98, BCHK97, CD95, HHK97, HACK00, KSH99, MS93]. Likewise, the works which use modes for verification purposes only distinguish input and output positions [AE93, AL95, AM94, AP94b, BC99, EBC99, EG99]. Part III of this thesis also falls into this category, since we assume that an argument position is either input or output.

More fine-grained characterisations

The mode analyses by Codish and others [CD94, CL96] characterise the degree of instantiation of the list, say, $[1, x, 5]$ by the abstract term $list(any)$, that is, a list whose elements cannot be characterised. Note that characterising this degree of instantiation is only meaningful with some notion of *type*. Similar approaches have been taken by Gallagher and de Waal [GW94] and Van Hentenryck et al. [VCL95], and in Part II of this thesis.

Other mode analyses that provide a relatively high degree of granularity but without using any notion of type have been developed by Janssens and Bruynooghe [JB92] and Tan and Lin [TL97].

The mode system of Mercury is based on instantiation states, which are a formalism for asserting how instantiated a term is. With instantiation states, one could express, say, that an argument position of a predicate is bound to a list of variables when the predicate is called and to a ground list when the predicate succeeds. This is a refinement of the notion of input and output.

2.2 Types

In logic programming, a *type* is usually a set of terms associated with an argument position, reflecting the programmer's understanding of what "kind" of term is expected in this argument position. For example, as arguments to the predicate `direct_flight` we might expect terms such as `rome` and `paris`, but not the number 3 or the list $[3, 5]$.

Types have been shown to be useful in all programming paradigms, since they can help detect logical errors in a program. However, types are not as widespread in logic programming as in imperative and functional programming.

As before, we discuss different notions of types occurring in the literature looking at them from various angles.

2.2.1 What *is* a Type?

First, we distinguish various approaches by how abstractly and generally the types are described. Figure 4 shows a rough subdivision of the literature into five groups. In this subsection, we ignore the existence of variables, that is, we only consider ground terms.

Built-in types in Prolog

Prolog is an untyped programming language. Nevertheless, in Prolog implementations, there are usually a few built-in types such as *integer* or *atom* [ISO95, SIC98]. These are only of any significance in connection with built-in predicates, for example `functor`/3. Any call to `functor` where the third argument is a ground term other than an integer results in a type error.

Ad-hoc types

Codish and Demoen [CD94] have shown how to derive type dependencies of logic programs using a specific set of types including integers, lists, difference lists, and trees.

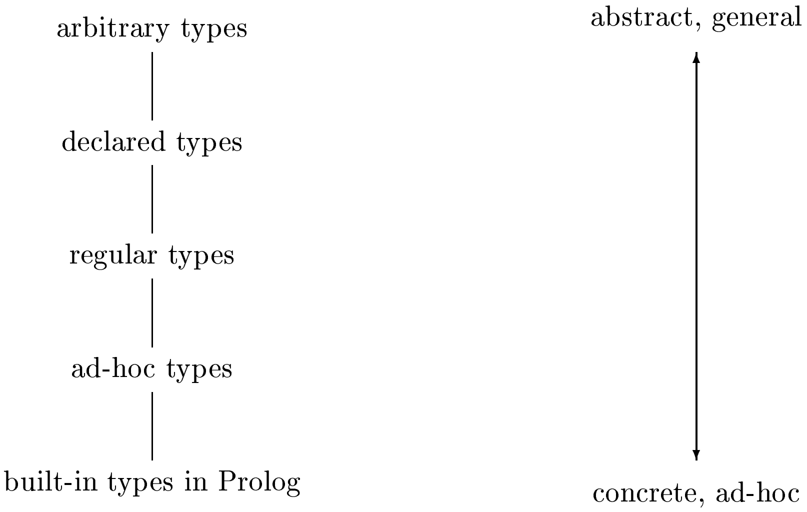

Figure 4: Expressiveness, generality of type formalisms

They suggest that this choice is for illustrative purposes and that it could easily be generalised, but as we will discuss in Section 4.5, the generalisation is by no means obvious.

Regular types

Many authors have developed formalisms to characterise types in a more general way, for example regular approximations [GW94, GL96, SG95a] or type graphs [VCL95]. The work of Codish and Demoen has also been developed further in this respect [CL96]. In all of these formalisms, an unlimited number of different types can be designed, but restrictions are imposed which ensure that these types are, in some sense, regular.

Declared types

Typed logic programming languages such as Mercury [SHC96] or Gödel [HL94] provide a syntax used to declare types. Each constant, function and predicate symbol used in a program must have its type declared. The type declarations have to meet a number of restrictions that can be syntactically checked. With these restrictions it is possible to type-check programs at compile time. Part II of this thesis uses this notion of types.

Arbitrary types

The literature that uses types for verification purposes [AE93, AL95, AM94, AP94b, BC99, BLR92] has the most general notion of type: any set of ground terms could be a type. On the level of the theory, there is no need to impose any restrictions. Part III of this thesis uses this notion of types.

2.2.2 Non-ground Types

In the previous subsection, we disregarded the possibility that a type might contain non-ground terms, or in other words, that a non-ground term might have a type. Considering non-ground terms adds another dimension to the classification of different approaches to types. Therefore this aspect should be studied separately.

In typed logic programming languages such as Mercury [SHC96] or Gödel [HL94], a variable has a type which is inferred from the declared types of the surrounding symbols. This ensures that the type of a term does not change via further instantiation. Hence the degree of instantiation and the type of a term are completely different issues. In contrast, Codish and others [CD94, CL96] would use, say, $list(any)$ to represent a list whose elements cannot be characterised, and they would refer to $list(any)$ as a *type*. In Part II, we also introduce objects such as $list(any)$, but we call them *abstract terms*, not types, since they only characterise the instantiation of a term, not its type.

Summarising, in typed logic programming languages, a non-ground term has a type which will not change via further instantiation. In the terminology used by some works on groundness analysis, a non-ground term also has a type, but this type represents the degree of instantiation of a term and hence may change via further instantiation.

The literature that uses types for verification purposes [AE93, AL95, AM94, AP94b, BC99, BLR92] defines a type as any set of terms closed under instantiation. Compared to requiring that a type must be a set of ground terms, this has the advantage that one can reason about programs that operate on non-ground data structures. For example, the predicate `append` can be used to append two lists whose elements are not instantiated. Part III also defines types in this way.

Defining a type as a set of terms closed under instantiation links the notion of *type* to that of *mode*. Therefore, we will consider non-ground types further in Section 2.3.

2.2.3 Polymorphism

Perhaps more important than the fact that the predicate `append` can be used to append two lists whose elements are not instantiated, is the fact that `append` can be used to append two lists regardless of the type of the list elements. Using a predicate for terms of different types in this way is called *(parametric) polymorphism.*

A polymorphic type is a type that is parametrised by another type. For example, the type $list(integer)$ is the type of integer lists and is composed of a type constructor $list$ and a type $integer$. For any type τ, there is a type $list(\tau)$. Note that allowing for type-checking at compile time, as practised in typed programming languages, is a much harder problem for polymorphic languages than for monomorphic ones [Hen93, Hil93, Mil78, MO84].

Part II of this thesis deals with groundness analysis of polymorphically typed programs. Previous works only allowed for very restricted forms of polymorphism. The works which use types for verification purposes [AE93, AL95, AM94, AP94b, BC99, BLR92], including Part III of this thesis, do not treat polymorphism explicitly.

There is another notion of polymorphism called *ad-hoc* polymorphism, but this is usually called *overloading* [Mil78, Str67]. For example, the constant nil may be used

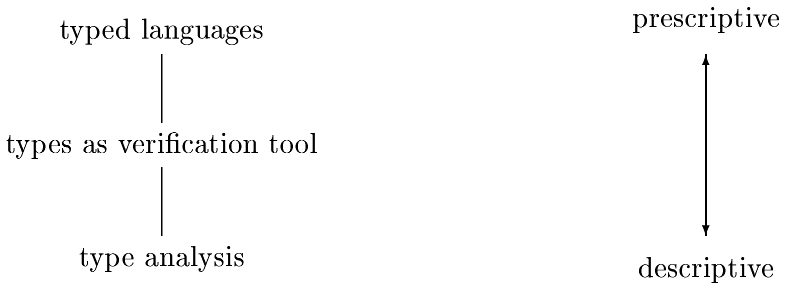

Figure 5: Descriptive versus prescriptive types

to denote the empty list as well as the empty tree. We are not concerned with ad-hoc polymorphism in this thesis.

2.2.4 Descriptive versus Prescriptive Types

As with modes, we can compare notions of types with respect to how descriptive and prescriptive they are. Figure 5 shows a subdivision of the literature into three groups. Note that this subdivision is very similar to the one we had for modes (Figure 3 on page 15).

Type analysis

Type analysis [CD94, CL96, GGS99, VCL95] is concerned with the question "what is the type of an argument or a variable?". This question can be qualified further by

- specifying the types of the arguments of the query with which the program is used,
- specifying program points of interest, such as the entry or exit point of a predicate.

In this context, "type" is a descriptive concept, and type analysis is inseparably linked to mode analysis. Saying that x is bound to a list can be viewed as a statement about the type of x as well as the degree of instantiation of x. Type analysis is a particularly precise kind of mode analysis, as described in Subsection 2.1.1, and further in the next section.

Type analysis is usually done using abstract interpretation [CC77]. The points made about abstract interpretation on page 15 apply here as well.

Types as verification tool

Just as type analysis is a particularly precise kind of mode analysis, types as verification tool [AE93, AL95, AM94, AP94b, BC99, BLR92] can be regarded as a refinement of modes as verification tool, and have been used for the same purposes. In addition to assuming that each argument position of a program is either input or output, a type is associated with each argument position. The program and initial goal have to be well typed, which is a property ensuring that all computed answers have terms of the correct type in each argument position. Usually, this approach is not concerned with how the type of each argument position is determined.

Just like modes as verification tool (page 16), "type" is a fairly prescriptive concept here, since assumptions are made about how programs should be written and used. Part III of this thesis uses this notion of type.

Typed languages

As with modes, the most prescriptive approach to types is having a typed language such as Mercury [Hen92, SHC96] or Gödel [HL94]. Part II deals with typed languages and hence uses this prescriptive notion of types. In typed languages, the user has to declare the types of each constant, function and predicate symbol used.[2] The type declarations have to meet a number of restrictions that can be syntactically checked. These ensure at compile time that no type errors can occur. That is, a predicate cannot be called with an argument not having the declared type.

2.3 Combining Modes and Types

We have seen on page 17 that fine-grained characterisations of the instantiation of a term often use some notion of type. On the other hand, we have seen in Subsection 2.2.2 that the degree of instantiation of a term plays a role in some concepts of types. Hence, modes and types are closely related. We now look at two ways of developing this relationship.

2.3.1 Directional Types

A natural way of joining modes and types is by the notion of *directional types* [BM95, BLR92, RNP92]. A directional type for an argument of a predicate has the form $\sigma \rightarrow \tau$. It is an assertion that if the argument is instantiated to a degree specified by σ at call time, then it will be instantiated to a degree specified by τ when the predicate returns. For example, the predicate `append` in forward mode could be specified by `append`$(list\!\rightarrow\!list,\ list\!\rightarrow\!list,\ free\!\rightarrow\!list)$ which should be read as: if `append` is called with the first and second arguments being lists, then for any answer, all arguments will be instantiated to lists.

Directional types have two aspects [BM95]. One is *input-output correctness*: if a call satisfies the input assertion, then the answer should specify the output assertion. It does not depend on the selection rule. The other is *call correctness*: If a call satisfies its input assertion, all triggered calls should also satisfy their input assertion. This aspect depends on the selection rule.

Both Part II and Part III of this thesis use formalisms that resemble directional types. The formalisms allow to express the intuition that, say, `append` is used in forward mode, although the precise meanings of the formalisms differ of course. To illustrate this point, we now show *how* this would be expressed. In Part II, simplifying the syntax

[2] This requirement could sometimes be relaxed since the types of some symbols can be reconstructed from the context.

somewhat, this intuition would be expressed by saying that append$(list, list, any)$ is a *call pattern* and append$(list, list, list)$ is an answer pattern. In Part III, it would be expressed by saying that the mode of append is append(I, I, O) and the type is append$(list, list, list)$.

2.3.2 A Declarative View of Modes

To understand Naish's declarative view of modes [Nai96], we must first understand his notion of *type*. It often happens that the success set of a program, that is, the set of ground atoms that are true in all its models, contains atoms that are not true according to the programmer's intentions. For example, the success set of the usual append program contains the atom append$([], 7, 7)$. A *type* is a set of atoms specified by the programmer which excludes such unintended atoms. For example, a natural type of append would be the set of all ground atoms append(s, t, u) where s, t, u are lists.

It is desirable that any call to a logic program can only give answers that are in the type. Calls that could result in answers not in the type should be considered unsafe. Suppose we are wondering whether a call to append(s, t, u) is safe. If we knew that all ground instances of append(s, t, u) that are in the success set of append are also in the type of append, then we would know that the call append(s, t, u) is safe. However, there is no way we could know the success set without actually executing the program.

Therefore, we have to approximate the success set. A *mode* of a program is any set of ground atoms which is a superset of the success set. One mode suggested for append is $\{$append$(s, t, u) \mid s \in list \wedge (t \in list \iff u \in list)\}$. Consider again the question whether a call is safe. If the call is append$([], \mathtt{X}, \mathtt{X})$, then it has a ground instance append$([], 7, 7)$ which is in the mode but not in the type, and it is therefore unsafe. If the call is append$([], \mathtt{X}, [])$, then for all instances in the mode, X must be bound to a list, and hence all instances in the mode are also in the type and the call is safe. In short, the mode together with the type encode the requirement that either the second or the third argument must be a list for a call to be safe, which means that either the second or the third argument must be input. This shows how procedural information can be derived from this declarative view.

2.4 Summary

In this chapter, we gave an overview of mode and type concepts used in the literature, by looking at these concepts from different angles. We now recall the most important properties of the mode and type concepts used in Parts II and III of this thesis.
In Part II, modes are

- *descriptive:* the modes of a program are *analysed*, not prescribed;
- *fine-grained:* the modes are characterised very precisely.

In Part II, types are

- *declared:* a syntax for this purpose is provided in typed programming languages;

- *prescriptive:* we consider typed programming languages, where a program must meet certain criteria concerning the types before it can be accepted by the compiler;
- *polymorphic:* a type can be parametrised by another type;
- *independent of instantiation:* the type of a term does not change via instantiation.

In Part III, modes are

- *(relatively) prescriptive:* the programs must meet certain criteria concerning the modes for our methods to be applicable;
- *coarse:* it is only possible to declare that arguments are input or output.

In Part III, types are

- *"arbitrary":* on the level of the theory, any set of terms (closed under instantiation) could be a type;
- *(relatively) prescriptive:* the programs must meet certain criteria concerning the types for our methods to be applicable;
- *closed under instantiation:* if a term has a type, then it continues to have that type even after it has been further instantiated.

Part II

Mode Analysis for Typed Logic Programs

Chapter 3

The Structure of Types and Terms

This part of the thesis describes a mode analysis for typed logic programs using abstract interpretation. It is divided into two chapters. This chapter is concerned with *concrete* terms, which are the data used in the programs we want to analyse. We define relations between the types in a program giving rise to certain structural properties of terms which the mode analysis is supposed to characterise.

In the next chapter, we will then define abstract terms to characterise these structural properties, as well as the actual mode analysis.

3.1 Introduction

Types are used in programming to restrict the underlying syntax so that only meaningful expressions are allowed. This enables most typographical errors and inconsistencies in the knowledge representation to be detected by the compiler. As a consequence, an increasing number of applications using typed logic programming languages such as Mercury [SHC96] or Gödel [HL94] are being developed.

Modes characterise the degree to which program variables are instantiated at certain program points. This information can be used to underpin optimisations such as the specialisation of unification and the removal of backtracking, and to support determinacy analysis [HK97]. When a mode analysis is formulated in terms of abstract interpretation, the program execution is traced using *descriptions* of data (the *abstract* domain) rather than *actual* data, and operations on these descriptions rather than operations on the actual data. A simple domain for mode analysis has two elements *ground* and *non-ground* to distinguish between ground and possibly non-ground terms. More complex domains can characterise partially instantiated data structures with more precision.

The main contribution of this part of the thesis is to describe a generic method of deriving precise abstract domains for mode analysis from the type declarations of a typed program. Each abstract domain is specialised for a particular type and characterises varying degrees of instantiation of terms of this type. In particular it characterises the property of *termination*. This property is well-known for lists as *nil*-termination and is here generalised to arbitrary types. Observe that termination of terms is closely related to the termination of programs that operate on these terms. For example, if the predicate `Append` is called with the first argument being a nil-terminated list, all invoked calls to `Append` also have the first argument being a nil-terminated list, and `Append` is guaranteed to terminate.

The procedure for constructing such domains is implemented (in Gödel) for Gödel programs. By incorporating the constructed domains into a mode analyser, we see that although the precision of the analysis is significantly improved, the analysis times (for the programs tested) compare well with a domain that only distinguishes ground and non-ground terms.

The abstract domains are used in an *abstract compilation* [CD95, DW86, HWD92] framework: a program is abstracted by replacing each unification with an abstract counterpart, and then the abstract program is evaluated by applying a standard operational semantics to it.

We believe that this work is the natural generalisation of work by Codish and others [CD94, CL96] and takes the idea presented there to its limits: our abstract domains provide the highest degree of precision that a generic domain construction should provide. It thus helps to understand other, more ad-hoc and pragmatic domain constructions as instances of a general theory.

This chapter is organised as follows. Section 3.2 introduces three examples. Section 3.3 defines some syntax. Section 3.4 defines relations between types. Section 3.5 defines termination of a term, as well as functions that extract certain subterms of a term.

3.2 Motivating and Illustrative Examples

We introduce three examples that are used throughout Part II. The syntax is that of the typed language Gödel [HL94]. Variables and (type) parameters begin with lower case letters; other alphabetic symbols begin with upper case letters. We use `Integer` (abbreviated as `Int`) to illustrate a type containing only constants $(1, 2, 3 \ldots)$.

Example 3.1 This is the usual list type. We give its declarations to illustrate the type description language of Gödel.

```
CONSTRUCTOR     List/1.
CONSTANT        Nil: List(u).
FUNCTION        Cons: u * List(u) -> List(u).
```

`List` is a (type) constructor; `u` is a type parameter that can be instantiated to any type such as `Int` or `List(Int)`; `Nil` is a constant of type `List(u)`; and `Cons` is the usual constructor for lists whose elements must all have the same type. We use the standard list notation $[\ldots|\ldots]$ where convenient. It is common to distinguish *nil-terminated* lists from *open* lists. For example, $[]$ and $[1, x, y]$ are nil-terminated, but $[1, 2|y]$ is open. ◁

Example 3.2 This example was invented to counter a common point of criticism that "list flattening" cannot be realised in Gödel, that is terms such as $[1, [2, 3]]$ cannot be defined, let alone flattened. The `Nests` module formalises nested lists by the type `Nest(v)`.

```
IMPORT           Lists, Integers.
CONSTRUCTOR      Nest/1.
FUNCTION         E: v -> Nest(v);
                 N: List(Nest(v)) -> Nest(v).
```

A trivial nest is constructed using function `E`, a complex nest by "nesting" a list of nests using function `N`. The notable property of the declaration for `N` is that the range type, `Nest(v)`, is a proper *sub"term"* (in the syntactic sense) of the argument type `List(Nest(v))`. We have seen a similar type declaration in Example 1.4. We use this example throughout to demonstrate that this work is a non-trivial generalisation of previous approaches to abstract domain construction [CD94, CL96, TL97]. The `Integers` module is imported since we frequently use `Nest(Int)` as an example. ◁

Example 3.3 A table is a data structure containing an ordered collection of nodes, each of which has two components, a key (of type `String`) and a value, of arbitrary type. We give part of the `Tables` module which is provided as a system module in Gödel.

```
IMPORT       Strings.
BASE         Balance.
CONSTRUCTOR  Table/1.
CONSTANT     Null: Table(u);
             LH, RH, EQ: Balance.
FUNCTION     Node:
                 Table(u) * String * u * Balance * Table(u) -> Table(u).
```

`Tables` is implemented in Gödel as an AVL-tree [Emd81]: A non-leaf node has a *key* argument, a *value* argument, arguments for the left and right subtrees, and an argument which represents balancing information. ◁

3.3 Notation and Terminology

The set of polymorphic types is given by the term structure $T(\Sigma_\tau, U)$ where Σ_τ is a finite alphabet of **constructor** symbols which includes at least one **base** (constructor of arity 0), and U is a countably infinite set of **parameters** (type variables). We define the order $\prec$ on types as the order induced by some (for example lexicographical) order on constructor and parameter symbols, where parameter symbols come before constructor symbols. Parameters are denoted by u, v. A tuple of *distinct* parameters ordered with respect to $\prec$ is denoted by $\bar{u}$. Types are denoted by $\sigma, \rho, \tau, \phi, \omega$ and tuples of types are denoted by $\bar{\sigma}, \bar{\tau}$.

Let Σ_f be an alphabet of **function** (term constructor) symbols which includes at least one **constant** (function of arity 0) and let Σ_p be an alphabet of predicate symbols. Each symbol in Σ_f (respectively Σ_p) has its *type* as subscript. If $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle} \in \Sigma_f$ (respectively $p_{\langle \tau_1 \ldots \tau_n \rangle} \in \Sigma_p$) then $\langle \tau_1, \ldots, \tau_n \rangle \in T(\Sigma_\tau, U)^\star$ and $\tau \in T(\Sigma_\tau, U) \setminus U$. If $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle} \in \Sigma_f$, then every parameter occurring in $\langle \tau_1, \ldots, \tau_n \rangle$ must also occur in τ. This condition is called **transparency condition**. We call τ the **range type** of

$f_{\langle\tau_1\ldots\tau_n,\tau\rangle}$. A symbol is often written without its type if it is clear from the context. Terms and atoms are defined in the usual way [HL94, HT92]. In this terminology, if a term *has* a type σ, it also *has* every *instance* of σ.[1] Thus in general, the type of a term is not unique. However the most general type of a term is unique up to parameter renaming. If V is a countably infinite set of variables, then the triple $L = \langle\Sigma_p, \Sigma_f, V\rangle$ defines a **polymorphic many-sorted first order language**. Variables are denoted by x, y; terms by t, r, s; tuples of *distinct* variables by $\bar{x}, \bar{y}$; and a tuple of terms by $\bar{t}$. The set of variables in a syntactic object o is denoted by $vars(o)$.

A **substitution** (denoted by θ) is a mapping from variables to terms which is the identity almost everywhere. The **domain** of a substitution θ is $dom(\theta) = \{x \mid x\theta \neq x\}$. The application of a substitution θ to a term t is denoted as $t\theta$. **Type substitutions** are defined analogously and denoted by ψ.

Programs are assumed to be in **normal form**. Thus a **literal**[2] is an equation of the form $x = y$ or $x = f(\bar{y})$, where $f \in \Sigma_f$, or an atom $Q(\bar{y})$, where $Q \in \Sigma_p$. A **query** G is a conjunction of literals. A clause is a formula of the form $Q(\bar{y}) \leftarrow G$. If S is a set of clauses, then the tuple $P = \langle L, S\rangle$ defines a **polymorphic many-sorted logic program**.

3.4 Relations between Types

An *abstract* term characterises the structure of a concrete term. It is a crucial choice in the design of abstract domains *which* aspects of the concrete structure should be characterised [TL97, VCL95]. In this part of the thesis we show how this choice can be based naturally on the information contained in the type declarations. This is formalised in this section. We describe how function declarations relate types to one another.

Definition 3.1 [subterm type] A type σ is a **direct subterm type of** ϕ (denoted as $\sigma \lhd \phi$) if there is $f_{\langle\tau_1\ldots\tau_n,\tau\rangle} \in \Sigma_f$ and a type substitution ψ such that $\tau\psi = \phi$ and $\tau_i\psi = \sigma$ for some $i \in \{1, \ldots, n\}$. The transitive, reflexive closure of $\lhd$ is denoted as $\lhd^*$. If $\sigma \lhd^* \phi$, then σ is a **subterm type of** ϕ. $\lhd$

Throughout Part II, we impose two restrictions on the language declarations we consider. We first need to define a *simple type*.

Definition 3.2 [simple type] A **simple type** is a type of the form $C(\bar{u})$, where $C \in \Sigma_\tau$. $\lhd$

The restrictions are as follows:

Simple Range Condition: For all $f_{\langle\tau_1\ldots\tau_n,\tau\rangle} \in \Sigma_f$, τ is a simple type.

Reflexive Condition: For all $C \in \Sigma_\tau$ and types $\sigma = C(\bar{\sigma}), \tau = C(\bar{\tau})$, if $\sigma \lhd^* \tau$, then σ is a sub"term" (in the syntactic sense) of τ.

[1] For example, the term `Nil` has type `List(u)`, `List(Int)`, `List(Nest(Int))` etc.

[2] We ignore negated literals here. In the implementation, negated literals may occur in the analysed program, but they are ignored in the analysis, which means that they do not contribute any information.

We do not know of any real programs that violate these conditions. In particular, they are met by all examples in Section 3.2. We now motivate the need for these restrictions.

The Simple Range Condition allows for the construction of an abstract domain for a type such as $\texttt{List}(\sigma)$ to be described independently of the type σ. An example of a violation of this condition would be to declare

```
FUNCTION        F: String -> List(Float).
```

in addition to the declarations in Example 3.1. Then we would have the pathological situation that a term of type `List(Float)` can have subterms of type `String`, `Float` and `List(Float)`, whereas for all $\sigma \neq \texttt{Float}$, $\texttt{List}(\sigma)$ can only have subterms of type σ and $\texttt{List}(\sigma)$. In Mercury [SHC96] and in typed *functional* languages such as ML or Haskell [Tho99], this condition is enforced by the syntax. For example, the list type would be declared in Haskell as

```
data List u = Nil | Cons u (List u)
```

and adding another declaration such as

```
data List Float = F String
```

would be illegal. Being able to violate the Simple Range Condition can be regarded as an artefact of the Gödel syntax.

An example of a violation of the Reflexive Condition would be to declare

```
FUNCTION        F: List(List(u)) -> List(u).
```

in addition to the declarations in Example 3.1. Then a term of type `List(Int)` could have subterms of type `List(Int)`, `List(List(Int))`, `List(List(List(Int)))` etc. The condition ensures that, for a program and a given query, there are only finitely many types and hence, the abstract program has only finitely many abstract domains.

Definition 3.3 [recursive type and non-recursive subterm type] A type σ is a **recursive type of** ϕ (denoted as $\sigma \bowtie \phi$) if $\sigma \lhd^* \phi$ and $\phi \lhd^* \sigma$.

A type σ is a **non-recursive subterm type of** ϕ (denoted as $\sigma \lhd\!\!\lhd \phi$) if $\phi \not\lhd^* \sigma$ and there is a type τ such that $\sigma \lhd \tau$ and $\tau \bowtie \phi$. We write $\mathcal{N}(\phi) = \{\sigma \mid \sigma \lhd\!\!\lhd \phi\}$. If $\mathcal{N}(\phi) = \{\sigma_1, \ldots, \sigma_m\}$ and $\sigma_j \prec \sigma_{j+1}$ for all $j \in \{1, \ldots, m-1\}$, we abuse notation and denote the *tuple* $\langle \sigma_1, \ldots, \sigma_m \rangle$ by $\mathcal{N}(\phi)$ as well. $\lhd$

Note that for example, $\texttt{Int} \bowtie \texttt{Int}$, although one might find it counterintuitive to think of `Int` as recursive type. Note moreover that in the above definition, $\tau \bowtie \phi$ includes the case that $\tau = \phi$. The definition has been designed to achieve uniformity of the presentation.

It follows immediately from the definition that if $\sigma \lhd\!\!\lhd \phi$, then $\sigma \not\bowtie \phi$. The relation $\lhd$ can be visualised as a *type graph* (similarly defined by Janssens and Bruynooghe [JB92], Somogyi [Som87] and Van Hentenryck et al. [VCL95]). The type graph for a type ϕ is a directed graph whose nodes are subterm types of ϕ. The node ϕ is called the *initial node*. There is an edge from σ_1 to σ_2 if and only if $\sigma_2 \lhd \sigma_1$. The recursive types of ϕ are all the types in the strongly connected component (SCC) of ϕ, and the non-recursive

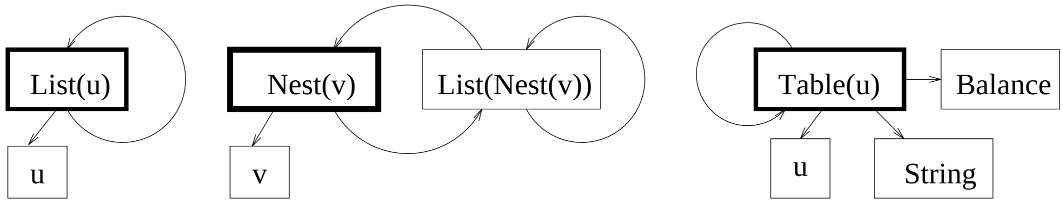

Figure 6: Some type graphs, with initial node highlighted

subterm types are all the types σ not in the SCC of ϕ but such that there is an edge from the SCC to σ. The finiteness of this graph is ensured by the Reflexive Condition. Our domain construction relies on the fact that $\mathcal{N}(\phi)$ is finite.

Example 3.4 In Figure 6 there is a type graph for each of the examples in Section 3.2. Trivially $\texttt{Int} \bowtie \texttt{Int}$. However, $\texttt{List(u)} \bowtie \texttt{List(u)}$ is non-trivial in that, in the type graph for `List(u)`, there is a path from `List(u)` to itself. Furthermore $\texttt{List(Nest(v))} \bowtie \texttt{Nest(v)}$. Non-recursive subterm types of simple types are often parameters, as in $\mathcal{N}(\texttt{List(u)}) = \langle \texttt{u} \rangle$ and $\mathcal{N}(\texttt{Nest(v)}) = \langle \texttt{v} \rangle$. However, this is not always the case, since $\mathcal{N}(\texttt{Table(u)}) = \langle \texttt{u}, \texttt{Balance}, \texttt{String} \rangle$. $\lhd$

It is important that the relation $\lhd$ is closed under instantiation of its arguments.

Lemma 3.1 Let σ, ϕ be types and ψ a type substitution. If $\sigma \lhd \phi$ then $\sigma\psi \lhd \phi\psi$. If $\sigma \lhd^* \phi$ then $\sigma\psi \lhd^* \phi\psi$.

PROOF. For the first statement, there is $f_{\langle \tau_1 \dots \tau_n, \tau \rangle} \in \Sigma_f$ and a type substitution ψ' such that for some $i \in \{1, \dots, n\}$, $\tau_i\psi' = \sigma$ and $\tau\psi' = \phi$. Consequently $\tau_i\psi'\psi = \sigma\psi$ and $\tau\psi'\psi = \phi\psi$, so $\sigma\psi \lhd \phi\psi$. The second statement follows from the first. □

The following lemma states another useful property of the relations $\lhd^*$ and $\bowtie$.

Lemma 3.2 Let ϕ, τ, σ be types so that $\sigma \lhd^* \tau \lhd^* \phi$ and $\sigma \bowtie \phi$. Then $\tau \bowtie \phi$.

PROOF. Since $\sigma \bowtie \phi$, it follows that $\phi \lhd^* \sigma$. Thus, since $\sigma \lhd^* \tau$, it follows that $\phi \lhd^* \tau$. Furthermore $\tau \lhd^* \phi$, and therefore $\tau \bowtie \phi$. □

The following lemma ensures that the abstract domains defined later are well-defined. It states that any sequence of non-recursive subterm types terminates.

Lemma 3.3 Let $\tau \in T(\Sigma_\tau, U) \setminus U$ and $\Gamma \subseteq \Sigma_\tau$. Let I be a non-empty index set (finite or infinite) starting at 1 and $\{(C_i(\bar{u}_i), \tau_i, \psi_i) \mid i \in I\}$ a sequence where $C_1 \in \Gamma$, $\tau_1 = C_1(\bar{u}_1)\psi_1 = \tau$, $dom(\psi_1) \subseteq \bar{u}_1$ and, for each $i \in I$ where $i > 1$:

- $C_i \in \Gamma$, $dom(\psi_i) \subseteq \bar{u}_i$ and $C_i(\bar{u}_i)\psi_i = \tau_i\psi_{i-1}$,
- $\tau_i \in T(\Gamma, U)$ and $\tau_i \lhd\!\!\lhd C_{i-1}(\bar{u}_{i-1})$.

$$\begin{tikzcd}
\cdots & & C_3(\bar{u}_3) \arrow[d, "\psi_3"'] & & C_2(\bar{u}_2) \arrow[d, "\psi_2"'] & & C_1(\bar{u}_1) \arrow[d, "\psi_1"'] \\
\cdots & \lll \arrow[ur] & = & \lll \arrow[ur] & = & \lll \arrow[ur] & = \\
\tau_4 \arrow[ur, dash] & & \tau_3 \arrow[u, "\psi_2"] \arrow[ur, dash] & & \tau_2 \arrow[u, "\psi_1"] \arrow[ur, dash] & & \tau_1 \arrow[u, "\psi_0"]
\end{tikzcd}$$

Figure 7: The sequence of non-recursive subterm types

Then I and hence $\{(C_i(\bar{u}_i), \tau_i, \psi_i) \mid i \in I\}$ is finite.

Proof. Let ψ_0 be the identity substitution. The sequence is illustrated in Figure 7. First note that, by Lemma 3.1 and Definition 3.3, for each $i \in I$ where $i \geq 2$, we have $\tau_i\psi_{i-1} \lhd^* \tau_{i-1}\psi_{i-2}$. Thus, for all $i, j \in I$ where $i > j$, $\tau_i\psi_{i-1} \lhd^* \tau_j\psi_{j-1}$.

Let $d(\rho)$ be the number of occurrences of constructors in a type ρ. If $\Gamma_0 \subseteq \Sigma_\tau$, define

$$D(\Gamma_0, \rho) = d(\rho) + \sum_{C \in \Gamma_0} \left(\sum_{\sigma \in \mathcal{N}(C(\bar{u}))} d(\sigma) \right).$$

The proof is by induction on $D(\Gamma, \tau)$. Since $\tau \notin U$, it follows that $D(\Gamma, \tau) \geq 1$. If $D(\Gamma, \tau) = 1$, then $\tau = C_1(\bar{u}_1)$, $\mathcal{N}(C_1(\bar{u}_1)) \subseteq U$ and $|I| \leq 2$.

Suppose that $D(\Gamma, \tau) = M > 1$. Assume that, for all types ρ and sets of constructors $\Gamma_0 \subseteq \Gamma$ such that $D(\Gamma_0, \rho) < M$, the result holds. Since the result obviously holds if $|I| \leq 2$, suppose $|I| > 2$ so that τ_2 is not a parameter. Consider the sequence $\{(C_i(\bar{u}_i), \tau_i, \psi'_i) \mid i \in I'\}$ where I' is an index set starting at 2, ψ'_1 is the identity substitution and, for each $i \in I'$, we have $C_i(\bar{u}_i)\psi'_i = \tau_i\psi'_{i-1}$. Since $\tau_i \lll C_{i-1}(\bar{u}_{i-1})$, $\psi'_i\psi_1 = \psi_i$ for each $i \in I'$. As in the first paragraph, for each $i \in I'$, $\tau_i\psi'_{i-1} \lhd^* \tau_2$. However, $\tau_2 \lll C_1(\bar{u}_1)$. Thus, by the Reflexive Condition and Lemma 3.2, for each $i \in I'$, we have $C_i \neq C_1$. Thus, for each $i \in I'$, we have $C_i \in \Gamma'$ where $\Gamma' = \Gamma \setminus \{C_1\}$. However,

$$D(\Gamma', \tau_2) = d(\tau_2) + D(\Gamma, \tau) - d(\tau) - \sum_{\sigma \in \mathcal{N}(C_1(\bar{u}))} d(\sigma).$$

Hence, as $d(\tau) > 0$ and $\tau_2 \in \mathcal{N}(C_1(\bar{u}))$, $D(\Gamma', \tau_2) < M$ and we can use the induction hypothesis. Hence I' is finite.

Assume now that I' is maximal with respect to the above conditions and that $|I'| = N'$ and suppose $K = N' + 1 \in I$. (If $K \notin I$, then, as I' is finite, I is finite.) Then $\tau_K\psi'_{K-1} = u$ where u is parameter since, if $\tau_K\psi'_{K-1} = C_K(\bar{u}_K)\psi'_K$, then K also satisfies the above conditions so that I' is not maximal. Thus ψ'_{K-1} is the identity substitution and $\tau_K = u$. By the transparency condition, since $\tau_K \lhd^* C_1(\bar{u}_1)$, $u \in \bar{u}_1$. As $\psi_{K-1} = \psi'_{K-1}\psi_1$, we have $\psi_{K-1} = \psi_1$ and $\tau_K\psi_{K-1} \in \bar{u}_1\psi_1$. Hence $d(\tau_K\psi_{K-1}) < d(\tau)$ so that

$$D(\Gamma, \tau_K\psi_{K-1}) < D(\Gamma, \tau).$$

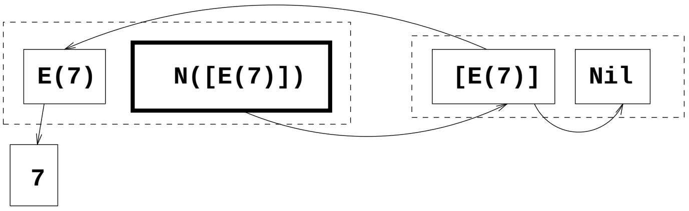

Figure 8: Term tree for $\texttt{N([E(7)])}^{\texttt{Nest(v)}}$

Hence, the inductive hypothesis can be applied to the remaining sequence starting at τ_K. Thus the subsequence starting at τ_K is finite and therefore the complete sequence starting at τ is finite. □

3.5 Traversing Concrete Terms

We now define *termination* of a term, as well as functions that extract certain subterms of a term.

From now on, we shall often annotate a term t with a type ϕ by writing t^ϕ. The use of this notation *always* implies that the type of t must be a (possibly trivial) *instance* of ϕ. The annotation ϕ gives the (type) context in which t is used. If S is a set of terms, then S^ϕ denotes the set of terms in S, each annotated with ϕ.

Definition 3.4 [subterm] Let t^ϕ be a term. Then t^ϕ is a **subterm of** t^ϕ at **depth** 0. If $s = f_{\langle\tau_1\ldots\tau_n,\tau\rangle}(s_1,\ldots,s_n)$ and for some type substitution ψ, $s^{\tau\psi}$ is a subterm of t^ϕ at depth d, then $s_i^{\tau_i\psi}$ is a **subterm of** t^ϕ at **depth** $d+1$ for $i \in \{1,\ldots,n\}$. We write $s^\sigma \lhd^* t^\phi$ if s^σ is a subterm of t^ϕ at some depth d ($s^\sigma \lhd t^\phi$ when $d = 1$). ◁

It can be seen that $s^\sigma \lhd^* t^\phi$ implies $\sigma \lhd^* \phi$. When the superscripts are ignored, the above is the usual definition of a subterm. The superscripts provide a uniform way of describing the "polymorphic type relationship" between a term and its subterms, which is independent of further instantiation.

Example 3.5 $\texttt{x}^{\texttt{v}}$ is a subterm of $\texttt{E(x)}^{\texttt{Nest(v)}}$, and $\texttt{7}^{\texttt{v}}$ is a subterm of $\texttt{E(7)}^{\texttt{Nest(v)}}$. ◁

Definition 3.5 [recursive subterm] Let s^σ and t^τ be terms such that $s^\sigma \lhd^* t^\tau$, and ϕ a type such that $\sigma \bowtie \phi$ and $\tau \lhd^* \phi$. Then s^σ is a ϕ-**recursive subterm of** t^τ. If furthermore $\tau = \phi$, then s^σ is a **recursive subterm of** t^τ. ◁

In particular, for every type ϕ, a variable is always a ϕ-recursive subterm of itself. The correspondence between subterms and subterm types can be illustrated by drawing the term as tree that resembles the corresponding type graph.

Example 3.6 The term tree for $t = \texttt{N([E(7)])}^{\texttt{Nest(v)}}$ is given in Figure 8 where the node for t is highlighted. Each box drawn with solid lines stands for a subterm. We can map this tree onto the type graph for `Nest(v)` in Figure 6 by replacing the subgraphs enclosed with dotted lines with corresponding nodes in the type graph. Thus the recursive subterms of t occur in the boxes corresponding to nodes in the SCC of `Nest(v)`. All subterms of t except $\mathtt{7}^{\mathtt{v}}$ are recursive subterms of t.

Note that $\mathtt{E(7)}^{\mathtt{Nest(v)}}$ is a `Nest(v)`-recursive subterm of $[\mathtt{E(7)}]^{\mathtt{List(Nest(v))}}$ (in Definition 3.5, take $\sigma = \phi = \mathtt{Nest(v)}$ and $\tau = \mathtt{List(Nest(v))}$). However, $\mathtt{E(7)}^{\mathtt{u}}$ is not a recursive subterm of $[\mathtt{E(7)}]^{\mathtt{List(u)}}$. Thus whether or not a member of a list should be regarded as a recursive subterm of that list depends on the context. $\triangleleft$

We now define *termination* of a term. Consider a term t^{ϕ}, where ϕ is simple. Termination of t means that no recursive subterm of t^{ϕ} is a variable. The formal definition is slightly more general.

Definition 3.6 [termination function $\mathcal{Z}$] Let t^{τ} be a term and ϕ be a type such that $\tau \bowtie \phi$. Define $\mathcal{Z}(t^{\tau}, \phi) = \mathit{false}$ if a ϕ-recursive subterm of t^{τ} is a variable, and *true* otherwise.

A term t is **terminated** if $t = f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}(t_1, \ldots, t_n)$ and $\mathcal{Z}(t^{\tau}, \tau) = \mathit{true}$.[3] A term is **open** if it is not terminated. For a set S^{τ} of terms define $\mathcal{Z}(S^{\tau}, \phi) = \bigwedge_{t \in S} \mathcal{Z}(t^{\tau}, \phi)$. We omit τ in the expression $\mathcal{Z}(t^{\tau}, \phi)$ whenever $\tau = \phi$. $\triangleleft$

Example 3.7 Any variable `x` is open. The term `7` has no variable subterm, therefore $\mathcal{Z}(\mathtt{7}, \mathtt{Int}) = \mathit{true}$ and `7` is terminated. The term $[\mathtt{x}]^{\mathtt{List(u)}}$ has itself and $\mathtt{Nil}^{\mathtt{List(u)}}$ as recursive subterms, therefore $\mathcal{Z}([\mathtt{x}], \mathtt{List(u)}) = \mathit{true}$ and $[\mathtt{x}]$ is terminated. However, $[\mathtt{x}]^{\mathtt{List(Nest(v))}}$ has $\mathtt{x}^{\mathtt{Nest(v)}}$ as a `Nest(v)`-recursive subterm, and so it follows that $\mathcal{Z}([\mathtt{x}]^{\mathtt{List(Nest(v))}}, \mathtt{Nest(v)}) = \mathit{false}$. Furthermore, $\mathtt{N}([\mathtt{x}])^{\mathtt{Nest(v)}}$ has $\mathtt{x}^{\mathtt{Nest(v)}}$ as a recursive subterm, so $\mathcal{Z}(\mathtt{N}([\mathtt{x}]), \mathtt{Nest(v)}) = \mathit{false}$ and $\mathtt{N}([\mathtt{x}])$ is open. $\triangleleft$

The abstract domain should not only characterise termination, but also the instantiation of subterms of a term. We define functions which extract sets of subterms from a term.

Definition 3.7 [extractor $\mathcal{E}^{\sigma}$ for σ] Let t^{τ} be a term and ϕ, σ be types such that $\tau \bowtie \phi$ and $\sigma \in \mathcal{N}(\phi)$. Let R be the set of ϕ-recursive subterms of t^{τ}. Define

$$\mathcal{E}^{\sigma}(t^{\tau}, \phi) = \mathit{vars}(R) \ \cup \ \{s \mid r^{\rho} \in R \text{ and } s^{\sigma} \lhd r^{\rho}\}.$$

For a set S^{τ} of terms define $\mathcal{E}^{\sigma}(S^{\tau}, \phi) = \bigcup_{t \in S} \mathcal{E}^{\sigma}(t^{\tau}, \phi)$. As with $\mathcal{Z}$, we write $\mathcal{E}^{\sigma}(t^{\tau}, \tau)$ simply as $\mathcal{E}^{\sigma}(t, \tau)$. $\triangleleft$

Example 3.8 For the term $\mathtt{N}([\mathtt{E(7)}])$ of type `Nest(Int)`, we have

$$\mathcal{E}^{\mathtt{v}}(\mathtt{N}([\mathtt{E(7)}]), \mathtt{Nest(v)}) = \{\mathtt{7}\}.$$

The type `Table(u)` has three non-recursive subterm types `u`, `Balance` and `String`, and so there are three extractor functions: $\mathcal{E}^{\mathtt{u}}$, which extracts all value subterms; $\mathcal{E}^{\mathtt{Balance}}$, which extracts all arguments containing balancing information; and $\mathcal{E}^{\mathtt{String}}$, which extracts all key subterms. In particular, this means that for a term t of type `Table(String)`, both $\mathcal{E}^{\mathtt{String}}(t)$ and $\mathcal{E}^{\mathtt{u}}(t)$ would contain terms of type `String`. $\triangleleft$

[3] Note that this includes the case that t is a constant.

Note that a priori, the extracted terms have no type annotation. This is because, in the proofs, we sometimes need to write an expression such as $\mathcal{E}^\sigma(\mathcal{E}^\rho(t,\tau)^{\rho\psi},\phi)$, which reads: first compute $\mathcal{E}^\rho(t,\tau)$, then annotate it with $\rho\psi$, then pass it to $\mathcal{E}^\sigma$.

Note also that if t has a ϕ-recursive subterm which is a variable, then this variable is always extracted. Intuitively this is because this variable might later be instantiated to a term which has variable subterms of type σ. Thus the property "$\mathcal{E}^\sigma(t^\tau,\phi)$ does not contain variables" is closed under instantiation of t.

The following lemma shows that $\mathcal{Z}$ and $\mathcal{E}^\sigma$ can be expressed in terms of the immediate subterms of a term. This provides the basis for defining the abstraction of a (normal form) equation in a concrete program, which naturally involves a term and its immediate subterms. Actually, we could have defined $\mathcal{Z}$ and $\mathcal{E}^\sigma$ by this property, but the definition using *subterms* is probably more intuitive.

Lemma 3.4 Let $t = f_{\langle\tau_1\ldots\tau_n,\tau\rangle}(t_1,\ldots,t_n)$ be a term and $\sigma \in \mathcal{N}(\tau)$. Then

$$\begin{aligned}\mathcal{Z}(t,\tau) &= \bigwedge_{\tau_i\bowtie\tau}\mathcal{Z}(t_i^{\tau_i},\tau)\\ \mathcal{E}^\sigma(t,\tau) &= \{t_i \mid \tau_i=\sigma\} \cup \bigcup_{\tau_i\bowtie\tau}\mathcal{E}^\sigma(t_i^{\tau_i},\tau).\end{aligned}$$

Proof. Let r^ρ be a τ-recursive subterm of $t_i^{\tau_i}$, for some $i \in \{1,\ldots,n\}$ where $\tau_i \bowtie \tau$. Then by Definitions 3.4 and 3.5, $\rho \bowtie \tau$ and $r^\rho \lhd^* t^\tau$, and hence r^ρ is a recursive subterm of t^τ.

Now let r^ρ be a recursive subterm of t^τ. Then either $r^\rho = t^\tau$ or, for some $i \in \{1,\ldots,n\}$, $r^\rho \lhd^* t_i^{\tau_i}$. In the latter case, by Definitions 3.4 and 3.5, $\rho \lhd^* \tau_i$, $\tau_i \lhd \tau$ and $\rho \bowtie \tau$. Hence, by Lemma 3.2, $\tau_i \bowtie \tau$ so that r^ρ is a τ-recursive subterm of $t_i^{\tau_i}$.

Thus the recursive subterms of t are t, together with the τ-recursive subterms of $t_i^{\tau_i}$, for all $\tau_i \bowtie \tau$. The result then follows from Definitions 3.6 and 3.7. □

The following lemmas are needed in the proof of Lemma 3.7, which is the key lemma used to prove Theorem 4.3.

Lemma 3.5 Let ϕ be a type, ψ a type substitution, and t a term having a type which is an instance of $\phi\psi$. If s^τ is a subterm of t^ϕ, then s has a type which is an instance of $\tau\psi$.

Proof. Induction on the depth of subterms. □

Lemma 3.6 Let $\sigma_1, \sigma_2, \sigma_3$ be types. If $\sigma_1 \bowtie \sigma_2$ and $\sigma_2\psi \bowtie \sigma_3$ for some type substitution ψ then $\sigma_1\psi \bowtie \sigma_3$.

Proof. By Lemma 3.1 it follows that $\sigma_1\psi \lhd^* \sigma_3$ and $\sigma_3 \lhd^* \sigma_1\psi$. □

Consider simple types ϕ and τ such that $\tau\psi \bowtie \phi$ for some type substitution ψ (for example $\phi = \mathtt{Nest(v)}$, $\tau = \mathtt{List(u)}$ and and $\psi = \{\mathtt{u/Nest(v)}\}$). The following key lemma relates ϕ with τ with respect to the termination and extractor functions.

Lemma 3.7 Let ϕ and τ be simple types such that $\tau\psi \bowtie \phi$ for some ψ, let t be a term having a type which is an instance of $\tau\psi$, and $\sigma \in \mathcal{N}(\phi)$. Then

$$\begin{aligned} \mathcal{Z}(t^{\tau\psi}, \phi) &= \mathcal{Z}(t, \tau) \wedge \bigwedge_{\substack{\rho \in \mathcal{N}(\tau) \\ \rho\psi \bowtie \phi}} \mathcal{Z}(\mathcal{E}^{\rho}(t, \tau)^{\rho\psi}, \phi) \quad (1) \\ \mathcal{E}^{\sigma}(t^{\tau\psi}, \phi) &= \bigcup_{\substack{\rho \in \mathcal{N}(\tau) \\ \rho\psi = \sigma}} \mathcal{E}^{\rho}(t, \tau) \cup \bigcup_{\substack{\rho \in \mathcal{N}(\tau) \\ \rho\psi \bowtie \phi}} \mathcal{E}^{\sigma}(\mathcal{E}^{\rho}(t, \tau)^{\rho\psi}, \phi) \quad (2) \end{aligned}$$

Proof. The proof consists of four parts. In Part 1, we define a number of sets of subterms of t. We then show six propositions which say that each expression occurring in (1) and (2) can be expressed in terms of these sets. In Part 2 we show how the left and right hand sides of both (1) and (2) can be related using these sets. This is then used in Part 3 to show (1), and in Part 4 to show (2).

Part 1: To avoid confusion between the many symbols occurring in the proof, keep in mind that ϕ, τ, σ and ψ occur in the statement and thus are *fixed*. We use f as an abbreviation for $f_{\langle \tau'_1 \ldots \tau'_n, \tau' \rangle}$ (*not* $f_{\langle \tau_1 \ldots \tau_n, \tau \rangle}$, as earlier in this chapter), and $\bar{r}$ to denote $(r_1, \ldots, r_n)$. Superscripts are omitted where irrelevant. Define

$$\begin{aligned} R &= \{r^{\omega} \mid r^{\omega} \text{ is a } \phi\text{–recursive subterm of } t^{\tau\psi}\} \\ S &= \{r_i \mid f(\bar{r})^{\tau'\psi'} \in R \text{ and } \tau'_i\psi' = \sigma\} \\ A &= \{r^{\omega} \mid r^{\omega} \text{ is a } \tau\text{–recursive subterm of } t^{\tau}\}. \end{aligned}$$

Note that, by Lemma 3.5, each $r^{\omega} \in A$ has a type which is an instance of $\omega\psi$. Furthermore for all $\rho \in \mathcal{N}(\tau)$ define

$$B^{\rho} = \{r_i \mid f(\bar{r})^{\tau'\psi'} \in A \text{ and } \tau'_i\psi' = \rho\}.$$

Note that, by Lemma 3.5, each $r_i \in B^{\rho}$ has a type which is an instance of $\tau'_i\psi'\psi$ $(= \rho\psi)$. For all $\rho \in \mathcal{N}(\tau)$ with $\rho\psi \bowtie \phi$ define

$$\begin{aligned} C^{\rho} &= \{r^{\omega} \mid r^{\omega} \text{ is a } \phi\text{-recursive subterm of some } s^{\rho\psi}, s \in B^{\rho}\} \\ D^{\rho} &= \{r_i \mid f(\bar{r})^{\tau'\psi'} \in C^{\rho} \text{ and } \tau'_i\psi' = \sigma\}. \end{aligned}$$

S1-S6 state how these sets relate to the computations of (1) and (2).

S1 $\mathcal{Z}(t^{\tau\psi}, \phi) = false$ if and only if $vars(R) \neq \emptyset$.

S2 $\mathcal{Z}(t, \tau) = false$ if and only if $vars(A) \neq \emptyset$.

S3 $\mathcal{E}^{\sigma}(t^{\tau\psi}, \phi) = vars(R) \ \cup \ S$.

S4 For each $\rho \in \mathcal{N}(\tau)$, $\mathcal{E}^{\rho}(t, \tau) = vars(A) \ \cup \ B^{\rho}$.

S5 For each $\rho \in \mathcal{N}(\tau)$ with $\rho\psi \bowtie \phi$, $\mathcal{Z}(\mathcal{E}^{\rho}(t, \tau)^{\rho\psi}, \phi) = false$ iff $vars(C^{\rho} \cup A) \neq \emptyset$.

S6 For each $\rho \in \mathcal{N}(\tau)$ with $\rho\psi \bowtie \phi$, $\mathcal{E}^{\sigma}(\mathcal{E}^{\rho}(t, \tau)^{\rho\psi}, \phi) = vars(A) \cup vars(C^{\rho}) \cup D^{\rho}$.

S1 and S2 follow from Definition 3.6 and the definitions of R and A. S3 and S4 follow from Definition 3.7 and the definitions of R, S, A and B^ρ. S5 and S6 are proved below. First we prove S5.

$$\begin{aligned}
&\mathcal{Z}(\mathcal{E}^\rho(t,\tau)^{\rho\psi},\phi) = \mathit{false} \iff \\
&\hspace{20em}\text{(by S4)}\\
&\mathcal{Z}((vars(A) \cup B^\rho)^{\rho\psi},\phi) = \mathit{false} \iff \\
&\hspace{20em}\text{(by Def. 3.6)}\\
&vars(\{r^\omega \mid r^\omega \text{ is a } \phi\text{−recursive subterm of } s^{\rho\psi}, s \in vars(A) \cup B^\rho\}) \neq \emptyset \iff \\
&\hspace{20em}\text{(by Def. 3.5)}\\
&vars(A) \cup vars(\{r^\omega \mid r^\omega \text{ is a } \phi\text{−recursive subterm of } s^{\rho\psi}, s \in B^\rho\}) \neq \emptyset \iff \\
&\hspace{20em}\text{(by Def. of } C^\rho)\\
&vars(A) \cup vars(C^\rho) \neq \emptyset.
\end{aligned}$$

We now prove S6.

$$\begin{aligned}
&\mathcal{E}^\sigma(\mathcal{E}^\rho(t,\tau)^{\rho\psi},\phi) = \\
&\hspace{20em}\text{(by S4)}\\
&\mathcal{E}^\sigma((vars(A) \cup B^\rho)^{\rho\psi},\phi) = \\
&\hspace{20em}\text{(by Def. 3.7)}\\
&vars(\{r^\omega \mid r^\omega \text{ is a } \phi\text{−recursive subterm of } s^{\rho\psi}, s \in vars(A) \cup B^\rho\}) \cup \\
&\qquad \{r_i \mid f(\bar{r})^{\tau'\psi'} \text{ is a } \phi\text{−recursive subterm of } s^{\rho\psi}, s \in B^\rho, \tau_i'\psi' = \sigma\} = \\
&\hspace{20em}\text{(by Def. 3.5)}\\
&vars(A) \cup vars(\{r^\omega \mid r^\omega \text{ is a } \phi\text{−recursive subterm of } s^{\rho\psi}, s \in B^\rho\}) \cup \\
&\qquad \{r_i \mid f(\bar{r})^{\tau'\psi'} \text{ is a } \phi\text{−recursive subterm of } s^{\rho\psi}, s \in B^\rho, \tau_i'\psi' = \sigma\} = \\
&\hspace{20em}\text{(by Def. of } C^\rho, D^\rho)\\
&vars(A) \cup vars(C^\rho) \cup D^\rho.
\end{aligned}$$

Part 2: Let r^ω be a subterm of t^τ at depth d. We show by induction on d that $r^{\omega\psi} \in R$ if and only if $r^\omega \in A$ or $r^{\omega\psi} \in C^\rho$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\psi \bowtie \phi$. For $d = 0$ this follows from the definitions of R and A.

Suppose now that r^ω is a subterm of t^τ at depth $d > 0$. Then there exists a subterm $f(\bar{r})^{\tau'\psi'}$ of t^τ at depth $d-1$ such that for some $i \in \{1,\dots,n\}$, $r = r_i$ and $\omega = \tau_i'\psi'$.

"$\Rightarrow$": Assume that $r^{\omega\psi} \in R$. Since $\omega\psi \bowtie \phi$, it follows from Lemma 3.2 that $\tau'\psi'\psi \bowtie \phi$ so that $f(\bar{r})^{\tau'\psi'\psi} \in R$. By the induction hypothesis there are two possibilities:

a) $f(\bar{r})^{\tau'\psi'} \in A$. Since $\tau'\psi' \bowtie \tau$, either $\omega \bowtie \tau$ or $\omega \lll \tau$. If $\omega \bowtie \tau$ then $r^\omega \in A$. If $\omega \lll \tau$, that is $\omega \in \mathcal{N}(\tau)$, then $r \in B^\omega$ and hence $r^{\omega\psi} \in C^\omega$, and therefore $r^{\omega\psi} \in C^\rho$ for some $\rho \in \mathcal{N}(\tau)$.

b) $f(\bar{r})^{\tau'\psi'\psi} \in C^\rho$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\psi \bowtie \phi$. Since $\omega\psi \bowtie \phi$ it follows that $r^{\omega\psi} \in C^\rho$.

"$\Leftarrow$": Again we break this up into cases:

a) $r^{\omega} \in A$. Since $\omega \bowtie \tau$, it follows by Lemma 3.2 that $\tau'\psi' \bowtie \tau$ so that $f(\bar{r})^{\tau'\psi'} \in A$. By the induction hypothesis $f(\bar{r})^{\tau'\psi'\psi} \in R$. Since $\omega \bowtie \tau$ and $\tau\psi \bowtie \phi$, it follows by Lemma 3.6 that $r^{\omega\psi} \in R$.

b) $r^{\omega\psi} \in C^{\rho}$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\psi \bowtie \phi$. By definition of C^{ρ} there are two possibilities: either $r \in B^{\rho}$, in which case $\omega = \rho$ and $f(\bar{r})^{\tau'\psi'} \in A$, or $\omega\psi \bowtie \phi$ and $f(\bar{r})^{\tau'\psi'\psi}$ is a subterm of an element of B^{ρ}. In the latter case, by Lemma 3.2, $\tau'\psi'\psi \bowtie \phi$ so that $f(\bar{r})^{\tau'\psi'\psi} \in C^{\rho}$.

   In both cases, by the induction hypothesis $f(\bar{r})^{\tau'\psi'\psi} \in R$. In the first case, since $\omega = \rho$ and $\rho\psi \bowtie \phi$, it follows that $r^{\omega\psi} \in R$. In the second case, since $\omega\psi \bowtie \phi$, $r^{\omega\psi} \in R$.

PART 3: We prove (1). By S1, $\mathcal{Z}(t^{\tau\psi}, \phi) = false$ if and only if $vars(R) \neq \emptyset$. By Part 2, $vars(R) \neq \emptyset$ if and only if $vars(A) \neq \emptyset$ or $vars(C^{\rho}) \neq \emptyset$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\psi \bowtie \phi$. Then, by S2 and S5, this holds if and only if

$$\mathcal{Z}(t,\tau) \wedge \bigwedge_{\substack{\rho \in \mathcal{N}(\phi) \\ \rho\psi \bowtie \phi}} \mathcal{Z}(\mathcal{E}^{\rho}(t,\tau)^{\rho\psi}, \phi) = false.$$

PART 4: We prove (2) by showing that:

$$vars(R) \cup S = \bigcup_{\rho\psi = \sigma} (vars(A) \cup B^{\rho}) \cup \bigcup_{\rho\psi \bowtie \phi} (vars(C^{\rho}) \cup D^{\rho}).$$

The result then follows from S3, S4, and S6.

"$\subseteq$": For a variable $x \in R$ it follows by Part 2 that $x \in A$, or $x \in C^{\rho}$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\psi \bowtie \phi$. For a term $r \in S$, there is $f(\bar{r})^{\tau'\psi'\psi} \in R$ such that $r = r_i$, and $\tau_i'\psi'\psi = \sigma$. By Part 2, either $f(\bar{r})^{\tau'\psi'} \in A$, or $f(\bar{r})^{\tau'\psi'\psi} \in C^{\rho}$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\psi \bowtie \phi$.

Assume first $f(\bar{r})^{\tau'\psi'} \in A$. We show that $r \in B^{\rho}$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\psi = \sigma$, namely $\rho = \tau_i'\psi'$. Since by construction of A, $\tau_i'\psi' \lhd^* \tau$, we only have to show that *not* $\tau_i'\psi' \bowtie \tau$. By Lemma 3.6, $\tau_i'\psi' \bowtie \tau$, together with $\tau\psi \bowtie \phi$, would imply $\tau_i'\psi'\psi \bowtie \phi$. This however is a contradiction, since it follows from $\tau_i'\psi'\psi = \sigma$ that $\tau_i'\psi'\psi \lhd\!\!\lhd \phi$.

Assume now $f(\bar{r})^{\tau'\psi'\psi} \in C^{\rho}$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\psi \bowtie \phi$. Since $\tau_i'\psi'\psi = \sigma$ it follows that $r \in D^{\rho}$.

"$\supseteq$": For a variable $x \in A$, or $x \in C^{\rho}$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\psi \bowtie \phi$, it follows by Part 2 that $x \in R$.

Secondly assume $r \in B^{\rho}$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\psi = \sigma$. By definition, there is $f(\bar{r})^{\tau'\psi'} \in A$ such that $r = r_i$ and $\tau_i'\psi' = \rho$. By Part 2, $f(\bar{r})^{\tau'\psi'\psi} \in R$, and since $\tau_i'\psi'\psi = \sigma$, it follows that $r \in S$.

Thirdly assume $r \in D^{\rho}$ for some $\rho \in \mathcal{N}(\tau)$ with $\rho\psi \bowtie \phi$. By definition, there is $f(\bar{r})^{\tau'\psi'\psi} \in C^{\rho}$ such that $r = r_i$ and $\tau_i'\psi'\psi = \sigma$. By Part 2, $f(\bar{r})^{\tau'\psi'\psi} \in R$, and since $\tau_i'\psi'\psi = \sigma$, it follows that $r \in S$. □

Example 3.9 First let $\phi = \tau = \texttt{List(u)}$ and ψ be the identity. Then by Definition 3.3 there is no ρ such that $\rho \in \mathcal{N}(\tau)$ and $\rho\psi \bowtie \phi$. Therefore in both equations of Lemma 3.7, the right half of the right hand side is empty. Furthermore there is obviously exactly one ρ such that $\rho\psi = \sigma$, namely $\rho = \sigma$. Thus the equations read

$$\begin{aligned} \mathcal{Z}(t,\tau) &= \mathcal{Z}(t,\tau) && (1) \\ \mathcal{E}^{\sigma}(t,\tau) &= \mathcal{E}^{\sigma}(t,\tau) && (2) \end{aligned}$$

In the same way, Lemma 3.7 reduces to a trivial statement for the `Tables` module (Example 3.3) and in fact for many types that are commonly used. However for Example 3.6, Lemma 3.7 says that

$$\mathcal{Z}([\texttt{E}(7)]^{\texttt{List(Nest(v))}}, \texttt{Nest(v)}) = \mathcal{Z}([\texttt{E}(7)], \texttt{List(u)}) \wedge \mathcal{Z}(\mathcal{E}^{\texttt{u}}([\texttt{E}(7)], \texttt{List(u)}), \texttt{Nest(v)}) \quad (1)$$

$$\mathcal{E}^{\texttt{v}}([\texttt{E}(7)]^{\texttt{List(Nest(v))}}, \texttt{Nest(v)}) = \emptyset \qquad \cup\, \mathcal{E}^{\texttt{v}}(\mathcal{E}^{\texttt{u}}([\texttt{E}(7)], \texttt{List(u)}), \texttt{Nest(v)}) \quad (2)$$

◁

In this chapter, we have defined the aspects of the structure of a (concrete) term which we want to characterise. First, we are interested in termination of a term. Secondly, we group the subterms of a term together according to their types. This is done using the extractor functions. In the next chapter, we will define abstract terms based on these concepts.

Chapter 4

Abstract Domains for Mode Analysis

In this chapter, we describe a mode analysis using abstract domains based on the termination and extractor functions introduced in the previous chapter.

This chapter is organised as follows. Section 4.1 defines the abstract domains and the abstraction function for terms. Section 4.2 defines termination and extractor functions for abstract terms, in analogy to the functions for concrete terms. Section 4.3 defines an abstract program and shows how its semantics approximates its concrete counterpart. Section 4.4 reports on experiments. Section 4.5 discusses the results and related work.

4.1 Abstraction of Terms

We first define an abstract domain for each type. Each abstract domain is a term structure, built using the constant symbols `Bot`, `Any`, `Ter`, `Open`, and the function symbols $C^{\mathcal{A}}$, for each $C \in \Sigma_\tau$. The meaning of these symbols will be explained shortly.

Definition 4.1 [abstract domain] If ϕ is a parameter, define

$$\mathcal{D}_\phi = \{\mathtt{Bot}, \mathtt{Any}\}.$$

If $C(\bar{u})$ is a simple type with $\mathcal{N}(C(\bar{u})) = \langle \sigma_1, \ldots, \sigma_m \rangle$ and $\phi = C(\bar{u})\psi$ where ψ is a type substitution, define

$$\mathcal{D}_\phi = \{C^{\mathcal{A}}(b_1, \ldots, b_m, \mathtt{Ter}) \mid b_j \in \mathcal{D}_{\sigma_j \psi}\} \cup \{C^{\mathcal{A}}(\underbrace{\mathtt{Any}, \ldots, \mathtt{Any}}_{m \text{ times}}, \mathtt{Open}),\ \mathtt{Bot},\ \mathtt{Any}\}.$$

$\mathcal{D}_\phi$ is the **abstract domain for** ϕ. If $b \in \mathcal{D}_\phi$, then b is an **abstract term for** ϕ. ◁

By Lemma 3.3, every abstract domain is well-defined. We shall see later that if an abstract term $C^{\mathcal{A}}(b_1, \ldots, b_m, \mathtt{Ter})$ abstracts a term t, then each b_j corresponds to a non-recursive subterm type σ_j of $C(\bar{u})$. The b_j characterises the degree of instantiation of the subterms extracted by $\mathcal{E}^{\sigma_j}$. In particular, the value `Any` for b_j corresponds to the case when a variable is extracted by $\mathcal{E}^{\sigma_j}$ from t. Thus, if t is a non-variable open term, each b_j must have the value `Any`.

The termination flags `Ter` and `Open` in the last argument position of an abstract term are not abstract terms but Boolean flags. The flag `Ter` abstracts the property of a term being terminated (and thus corresponds to *true*) and `Open` that of being open (and thus corresponds to $false$). Note that for some types, for example `Int`, a term can be open only if it is a variable. In these cases, the termination flag is omitted in the implementation (see Section 4.4). We keep it in the theory for the sake of a uniform presentation.

Example 4.1 Consider the examples in Section 3.2 (see also Figure 6 on page 31).

$$\mathcal{D}_{\texttt{Int}} = \{\texttt{Int}^{\mathcal{A}}(\texttt{Ter}),\ \texttt{Int}^{\mathcal{A}}(\texttt{Open}),\ \texttt{Bot},\ \texttt{Any}\}.$$

The following examples illustrate that Definition 4.1 is "parametric".

$$\begin{array}{rcll}
\mathcal{D}_{\texttt{List(Int)}} & = & \{\texttt{List}^{\mathcal{A}}(i,\texttt{Ter}) \mid i \in \mathcal{D}_{\texttt{Int}}\} & \cup\{\texttt{List}^{\mathcal{A}}(\texttt{Any},\texttt{Open}),\ \texttt{Bot},\ \texttt{Any}\}\\
\mathcal{D}_{\texttt{List(String)}} & = & \{\texttt{List}^{\mathcal{A}}(i,\texttt{Ter}) \mid i \in \mathcal{D}_{\texttt{String}}\} & \cup\{\texttt{List}^{\mathcal{A}}(\texttt{Any},\texttt{Open}),\ \texttt{Bot},\ \texttt{Any}\}\\
\mathcal{D}_{\texttt{List(u)}} & = & \{\texttt{List}^{\mathcal{A}}(i,\texttt{Ter}) \mid i \in \mathcal{D}_{\texttt{u}}\} & \cup\{\texttt{List}^{\mathcal{A}}(\texttt{Any},\texttt{Open}),\ \texttt{Bot},\ \texttt{Any}\}.
\end{array}$$

Some further examples are, assuming that $\texttt{u} \prec \texttt{Balance} \prec \texttt{String}$:

$$\begin{array}{rcl}
\mathcal{D}_{\texttt{Balance}} & = & \{\texttt{Balance}^{\mathcal{A}}(\texttt{Ter}),\ \texttt{Balance}^{\mathcal{A}}(\texttt{Open}),\ \texttt{Bot},\ \texttt{Any}\}\\
\mathcal{D}_{\texttt{String}} & = & \{\texttt{String}^{\mathcal{A}}(\texttt{Ter}),\ \texttt{String}^{\mathcal{A}}(\texttt{Open}),\ \texttt{Bot},\ \texttt{Any}\}\\
\mathcal{D}_{\texttt{Table(Int)}} & = & \{\texttt{Table}^{\mathcal{A}}(i,b,s,\texttt{Ter}) \mid i \in \mathcal{D}_{\texttt{Int}}, b \in \mathcal{D}_{\texttt{Balance}}, s \in \mathcal{D}_{\texttt{String}}\}\cup\\
 & & \{\texttt{Table}^{\mathcal{A}}(\texttt{Any},\texttt{Any},\texttt{Any},\texttt{Open}),\ \texttt{Bot},\ \texttt{Any}\}\\
\mathcal{D}_{\texttt{Nest(Int)}} & = & \{\texttt{Nest}^{\mathcal{A}}(i,\texttt{Ter}) \mid i \in \mathcal{D}_{\texttt{Int}}\} \cup \{\texttt{Nest}^{\mathcal{A}}(\texttt{Any},\texttt{Open}),\ \texttt{Bot},\ \texttt{Any}\}.
\end{array}$$

◁

We now define an order on abstract terms which has the usual interpretation that "smaller" stands for "more precise". Since the least upper and greatest lower bound of two abstract terms with respect to this order always exist, it follows that each abstract domain is a lattice.

Definition 4.2 [order $<$ on abstract terms] For the termination flags define $\texttt{Ter} < \texttt{Open}$. For abstract terms, $<$ is defined as follows:

$$\begin{array}{rcll}
\texttt{Bot} & < & b & \text{if } b \neq \texttt{Bot},\\
b & < & \texttt{Any} & \text{if } b \neq \texttt{Any},\\
C^{\mathcal{A}}(b_1,\ldots,b_m,c) & \leq & C^{\mathcal{A}}(b'_1,\ldots,b'_m,c') & \text{if } c \leq c' \text{ and } b_j \leq b'_j,\ j \in \{1,\ldots,m\}.
\end{array}$$

For a set S of abstract terms, let $\sqcup S$ denote the least upper bound of S with respect to the order $<$. ◁

We now define the abstraction function for terms. This definition needs an abstraction of *truth values* as an auxiliary construction. The abstraction function formalises the relationship between concrete and abstract terms, so that the results of a mode analysis can be interpreted. The abstraction function is never actually computed during the analysis.

Definition 4.3 [abstraction function α for terms] Let $\tau = C(\bar{u})$ and let $\mathcal{N}(\tau) = \langle \sigma_1, \dots, \sigma_m \rangle$. For the truth values define $\alpha(true) = \texttt{Ter}$ and $\alpha(false) = \texttt{Open}$. If S is a set of terms, define

$$\alpha(S) = \sqcup\{\alpha(t) \mid t \in S\},$$

where $\alpha(t)$ is defined as:

$$\begin{array}{ll} \texttt{Any} & \text{if } t \text{ is a variable,} \\ C^{\mathcal{A}}(\alpha(\mathcal{E}^{\sigma_1}(t,\tau)), \dots, \alpha(\mathcal{E}^{\sigma_m}(t,\tau)), \alpha(\mathcal{Z}(t,\tau))) & \text{if } t = f_{\langle \tau_1 \dots \tau_n, \tau \rangle}(t_1, \dots, t_n). \end{array}$$

◁

Note that this definition is based on the fact that $\alpha(\emptyset) = \texttt{Bot}$. From this it follows that the abstraction of a constant $t = f_{\langle \tau \rangle}$ is $C^{\mathcal{A}}(\texttt{Bot}, \dots, \texttt{Bot}, \texttt{Ter})$.

The least upper bound of a *set* of abstract terms gives a safe approximation for the instantiation of *all* corresponding concrete terms. *Safe* means that each concrete term is at least as instantiated as indicated by the least upper bound. As we will see in Section 4.3, our mode analysis can only give approximations of the instantiation of terms in this sense. It can never infer that a term is definitely free, that is, an uninstantiated variable. Inferring that a term is definitely free requires different techniques [BDB$^+$96].

Example 4.2 We illustrate Definition 4.3.

$$\begin{array}{ll} \alpha(7) = \texttt{Int}^{\mathcal{A}}(\texttt{Ter}) & (\tau = \texttt{Int}, m = 0, n = 0) \\ \alpha(\texttt{Nil}) & (\tau = \texttt{List(u)}, \mathcal{N}(\tau) = \langle \texttt{u} \rangle, n = 0) \\ \quad = \texttt{List}^{\mathcal{A}}(\alpha(\emptyset), \alpha(\mathcal{Z}(\texttt{Nil}, \tau))) & \\ \quad = \texttt{List}^{\mathcal{A}}(\texttt{Bot}, \texttt{Ter}) & \\ \alpha(\texttt{Cons(7,Nil)}) & (\tau = \texttt{List(u)}, \mathcal{N}(\tau) = \langle \texttt{u} \rangle, n = 2) \\ \quad = \texttt{List}^{\mathcal{A}}(\sqcup\{\alpha(\texttt{7})\}, \alpha(\mathcal{Z}(\texttt{Cons(7,Nil)}, \tau))) & \\ \quad = \texttt{List}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}(\texttt{Ter}), \texttt{Ter}). & \end{array}$$

Table 1 gives some further examples. Note that there is no term of type Int whose abstraction is $\texttt{Int}^{\mathcal{A}}(\texttt{Open})$. ◁

The following is an auxiliary lemma needed for the proof of Lemma 4.2.

Lemma 4.1 Let t^τ be a term. Every subterm of t^τ is either a recursive subterm of t^τ, or a subterm of a term in $\mathcal{E}^\sigma(t, \tau)$, for some $\sigma \in \mathcal{N}(\tau)$.

PROOF. The proof is by induction on the depth of subterms of t^τ. For the base case observe that t^τ is a recursive subterm of itself.

Now suppose the result holds for all subterms of t^τ up to depth i. Let r^ρ be a subterm of t^τ at depth i and $w^\omega \lhd r^\rho$. If r^ρ is not a recursive subterm of t^τ, then r^ρ is a subterm of a term in $\mathcal{E}^\sigma(t, \tau)$ for some $\sigma \in \mathcal{N}(\tau)$, and thus w^ω is also a subterm of a term in $\mathcal{E}^\sigma(t, \tau)$. If r^ρ is a recursive subterm of t^τ, then since $\rho \bowtie \tau$ and $\omega \lhd \rho$, by Definition 3.3 either $\omega \bowtie \tau$ or $\omega \lll \tau$. Thus either w^ω is a recursive subterm of t^τ or $w \in \mathcal{E}^\omega(t, \tau)$. □

The following lemma shows that the abstraction captures groundness.

Table 1: Some terms, their types, and abstractions

term	**type**	**abstraction**
$\mathtt{x}$	$\mathtt{u}$	$\mathtt{Any}$
$\mathtt{[7,x]}$	$\mathtt{List(Int)}$	$\mathtt{List}^{\mathcal{A}}(\mathtt{Any},\mathtt{Ter})$
$\mathtt{[7\vert x]}$	$\mathtt{List(Int)}$	$\mathtt{List}^{\mathcal{A}}(\mathtt{Any},\mathtt{Open})$
$\mathtt{E(7)}$	$\mathtt{Nest(Int)}$	$\mathtt{Nest}^{\mathcal{A}}(\mathtt{Int}^{\mathcal{A}}(\mathtt{Ter}),\mathtt{Ter})$
$\mathtt{[E(7)]}$	$\mathtt{List(Nest(Int))}$	$\mathtt{List}^{\mathcal{A}}(\mathtt{Nest}^{\mathcal{A}}(\mathtt{Int}^{\mathcal{A}}(\mathtt{Ter}),\mathtt{Ter}),\mathtt{Ter})$
$\mathtt{N([E(7)])}$	$\mathtt{Nest(Int)}$	$\mathtt{Nest}^{\mathcal{A}}(\mathtt{Int}^{\mathcal{A}}(\mathtt{Ter}),\mathtt{Ter})$
$\mathtt{N([E(7),x])}$	$\mathtt{Nest(Int)}$	$\mathtt{Nest}^{\mathcal{A}}(\mathtt{Any},\mathtt{Open})$
$\mathtt{N([E(7)\vert x])}$	$\mathtt{Nest(Int)}$	$\mathtt{Nest}^{\mathcal{A}}(\mathtt{Any},\mathtt{Open})$

Lemma 4.2 Let S be a set of terms having the same type. Then a variable occurs in an element of S (that is S is non-ground) if and only if $\mathtt{Any}$ or $\mathtt{Open}$ occurs in $\alpha(S)$.

PROOF. There are three cases depending on whether S is empty, contains a variable, or neither.

CASE 1: S is empty. Then $\alpha(S) = \mathtt{Bot}$.

CASE 2: $x \in S$ for some variable x. Then $\alpha(x) = \mathtt{Any}$ and thus $\alpha(S) = \mathtt{Any}$.

CASE 3: S contains no variables but contains a non-variable term. Then the type of terms in S is of the form $\tau\psi$ for some type substitution ψ and simple type $\tau = C(\bar{u})$. Suppose that $\mathcal{N}(\tau) = \langle\sigma_1, \dots, \sigma_m\rangle$ for some $m \geq 0$. Then there are abstract terms $b_1, \dots, b_m$ and a termination flag b such that

$$\alpha(S) = C^{\mathcal{A}}(b_1, \dots, b_m, b).$$

There are two subcases.

CASE 3a: For some $t \in S$ and variable x, x^{ρ} is a recursive subterm of t^{τ}. Then $\mathcal{Z}(t, \tau) = \mathtt{Open}$. Hence $b = \mathtt{Open}$ and

$$\alpha(S) = C^{\mathcal{A}}(b_1, \dots, b_m, \mathtt{Open}).$$

CASE 3b: No term in S has a recursive subterm that is a variable. Then $\mathcal{Z}(t, \tau) = \mathtt{Ter}$ for each $t \in S$. Hence, by Definition 4.2, $b = \mathtt{Ter}$. The proof for this case is by induction on the length of the longest $\lll$-sequence (see Lemma 3.3) for $\tau\psi$. The base case is when $m = 0$. Then by Lemma 4.1, every term in S is ground and $\alpha(S) = C^{\mathcal{A}}(\mathtt{Ter})$.

Now suppose $m > 0$. By Lemma 4.1, S contains a non-ground term if and only if $\mathcal{E}^{\sigma_j}(t, \tau)$ contains a non-ground term for some $t \in S$ and $j \in \{1, \dots, m\}$. By Definition 4.3

$$\alpha(S) = \sqcup\{C^{\mathcal{A}}(\alpha(\mathcal{E}^{\sigma_1}(t, \tau)), \dots, \alpha(\mathcal{E}^{\sigma_m}(t, \tau)), \mathtt{Ter}) \mid t^{\tau} \in S\}.$$

Thus, by Definitions 4.2 and 4.3, for each $j \in \{1, \dots, m\}$, we have $b_j = \alpha(\mathcal{E}^{\sigma_j}(S, \tau))$. Let $j \in \{1, \dots, m\}$. If $\mathcal{E}^{\sigma_j}(S, \tau)$ is empty, by Case 1 above, $\alpha(\mathcal{E}^{\sigma_j}(S, \tau)) = \mathtt{Bot}$. If $\mathcal{E}^{\sigma_j}(S, \tau)$

contains a variable, by Case 2 above, $\alpha(\mathcal{E}^{\sigma_j}(S,\tau)) = \texttt{Any}$. Otherwise, $\mathcal{E}^{\sigma_j}(S,\tau)$ contains a non-variable term and the terms in $\mathcal{E}^{\sigma_j}(S,\tau)$ have type $\sigma_j\psi$, for which, by induction hypothesis, the result holds. Hence b_j has an occurrence of `Any` or `Open` if and only if $\mathcal{E}^{\sigma_j}(S,\tau)$ contains a non-ground term. It follows that $\alpha(S)$ has an occurrence of `Any` or `Open` if and only if S contains a non-ground term. □

4.2 Traversing Abstract Terms

In order to define abstract unification and, in particular, the abstraction of an equation in a program, we require an abstract termination function and abstract extractors similar to those already defined for concrete terms. The type superscript annotation for concrete terms is also useful for abstract terms.

Definition 4.4 [abstract termination function and extractor for σ] Let ϕ and $\tau = C(\bar{u})$ be simple types such that $\tau\psi \bowtie \phi$ for some ψ, and $\mathcal{N}(\tau) = \langle\sigma_1,\dots,\sigma_m\rangle$. Let b be an abstract term for an instance of $\tau\psi$.

1. Abstract termination function.

$$\begin{array}{lll}
\mathcal{AZ}(b^{\tau\psi},\phi) = \texttt{Open} & & \text{if } b = \texttt{Any} \\
\mathcal{AZ}(b^{\tau\psi},\phi) = \texttt{Ter} & & \text{if } b = \texttt{Bot} \\
\mathcal{AZ}(b^{\tau\psi},\phi) = c \wedge \bigwedge\limits_{\sigma_j\psi\bowtie\phi} \mathcal{AZ}(b_j^{\sigma_j\psi},\phi) & & \text{if } b = C^{\mathcal{A}}(b_1,\dots,b_m,c).
\end{array}$$

2. Abstract extractor for σ. Let $\sigma \in \mathcal{N}(\phi)$.

$$\begin{array}{lll}
\mathcal{AE}^{\sigma}(b^{\tau\psi},\phi) = \texttt{Any} & & \text{if } b = \texttt{Any} \\
\mathcal{AE}^{\sigma}(b^{\tau\psi},\phi) = \texttt{Bot} & & \text{if } b = \texttt{Bot} \\
\mathcal{AE}^{\sigma}(b^{\tau\psi},\phi) = \sqcup(\{b_j \mid \sigma_j\psi = \sigma\}\cup & & \\
\qquad\qquad \{\mathcal{AE}^{\sigma}(b_j^{\sigma_j\psi},\phi) \mid \sigma_j\psi \bowtie \phi\}) & & \text{if } b = C^{\mathcal{A}}(b_1,\dots,b_m,c).
\end{array}$$

◁

As for the concrete termination functions and extractors, we omit the superscript $\tau\psi$ in the expressions $\mathcal{AZ}(b^{\tau\psi},\phi)$ and $\mathcal{AE}^{\sigma}(b^{\tau\psi},\phi)$ whenever $\tau = \phi$ and ψ is the identity. In this (very common) case, the abstract termination function is merely a *projection* onto the termination flag of an abstract term (or `Open` if the abstract term is `Any`). Similarly, the abstract extractor for σ is merely a projection onto the j^{th} argument of an abstract term, where $\sigma = \sigma_j$. Note the similarity between the above definition and Lemma 3.4.

Example 4.3

$$\mathcal{AZ}(\texttt{List}^{\mathcal{A}}(\texttt{Any},\texttt{Ter})^{\texttt{List(Nest(v))}},\ \texttt{Nest(v)}) \ = \ \texttt{Ter} \wedge \mathcal{AZ}(\texttt{Any},\texttt{Nest(v)}) = \texttt{Open}.$$

$$\mathcal{AE}^{\texttt{v}}(\texttt{List}^{\mathcal{A}}(\texttt{Any},\texttt{Ter})^{\texttt{List(Nest(v))}},\ \texttt{Nest(v)}) \ = \ \texttt{Any}.$$

$$\begin{array}{l}
\mathcal{AZ}(\texttt{List}^{\mathcal{A}}(\texttt{Nest}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}(\texttt{Ter}),\texttt{Ter}),\texttt{Ter})^{\texttt{List(Nest(v))}},\ \texttt{Nest(v)}) \ = \\
\quad \texttt{Ter} \wedge \mathcal{AZ}(\texttt{Nest}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}(\texttt{Ter}),\texttt{Ter}),\ \texttt{Nest(v)}) \ = \ \texttt{Ter}.
\end{array}$$

$$\begin{array}{l}
\mathcal{AE}^{\texttt{v}}(\texttt{List}^{\mathcal{A}}(\texttt{Nest}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}(\texttt{Ter}),\texttt{Ter}),\texttt{Ter})^{\texttt{List(Nest(v))}},\ \texttt{Nest(v)}) \ = \\
\qquad \mathcal{AE}^{\texttt{v}}(\texttt{Nest}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}(\texttt{Ter}),\texttt{Ter}),\ \texttt{Nest(v)}) \ = \ \texttt{Int}^{\mathcal{A}}(\texttt{Ter}).
\end{array}$$

$\triangleleft$

The following theorem states the fundamental relationship between concrete and abstract termination functions and extractors.

Theorem 4.3 Let ϕ and $\tau = C(\bar{u})$ be simple types such that $\tau\psi \bowtie \phi$ for some ψ, and $\sigma \in \mathcal{N}(\phi)$. Let $t^{\tau\psi}$ be a term. Then

$$\begin{aligned} \alpha(\mathcal{Z}(t^{\tau\psi}, \phi)) &= \mathcal{AZ}(\alpha(t)^{\tau\psi}, \phi) && (1)\\ \alpha(\mathcal{E}^{\sigma}(t^{\tau\psi}, \phi)) &= \mathcal{AE}^{\sigma}(\alpha(t)^{\tau\psi}, \phi) && (2) \end{aligned}$$

Proof. The proof is by induction on the structure of t. First assume t is a variable x or a constant d. Here we omit the type superscripts because they are irrelevant.

$$\begin{aligned} \alpha(\mathcal{Z}(x,\phi)) &= \alpha(false) = \texttt{Open} = \mathcal{AZ}(\texttt{Any}, \phi) = \mathcal{AZ}(\alpha(x), \phi).\\ \alpha(\mathcal{E}^{\sigma}(x,\phi)) &= \sqcup\{\alpha(x)\} = \texttt{Any} = \mathcal{AE}^{\sigma}(\texttt{Any}, \phi) = \mathcal{AE}^{\sigma}(\alpha(x), \phi). \end{aligned}$$

$$\begin{aligned} \alpha(\mathcal{Z}(d,\phi)) &= \alpha(true) = \texttt{Ter} = \mathcal{AZ}(C^{\mathcal{A}}(\texttt{Bot}, \ldots, \texttt{Bot}, \texttt{Ter}), \phi) = \mathcal{AZ}(\alpha(d), \phi).\\ \alpha(\mathcal{E}^{\sigma}(d,\phi)) &= \sqcup\,\emptyset = \texttt{Bot} = \mathcal{AE}^{\sigma}(C^{\mathcal{A}}(\texttt{Bot}, \ldots, \texttt{Bot}, \texttt{Ter}), \phi) = \mathcal{AE}^{\sigma}(\alpha(d), \phi). \end{aligned}$$

Now assume t is a compound term. Let $\mathcal{N}(\tau) = \langle \sigma_1, \ldots, \sigma_m \rangle$. In the following sequences of equations, $*$ marks steps which use straightforward manipulations such as rearranging least upper bounds or applications of α to sets. We show (1) working from right to left.

$$\begin{aligned} &\mathcal{AZ}(\alpha(t)^{\tau\psi}, \phi) = && \text{(Definition 4.3)}\\ &\mathcal{AZ}(C^{\mathcal{A}}(\alpha(\mathcal{E}^{\sigma_1}(t,\tau)), \ldots, \alpha(\mathcal{E}^{\sigma_m}(t,\tau)), \alpha(\mathcal{Z}(t,\tau)))^{\tau\psi},\ \phi) = && \text{(Definition 4.4)}\\ &\alpha(\mathcal{Z}(t,\tau)) \wedge \bigwedge_{\sigma_j\psi \bowtie \phi} \mathcal{AZ}(\alpha(\mathcal{E}^{\sigma_j}(t,\tau))^{\sigma_j\psi},\ \phi) = && (*\text{ and hypothesis})\\ &\alpha(\mathcal{Z}(t,\tau)) \wedge \bigwedge_{\sigma_j\psi \bowtie \phi} \alpha(\mathcal{Z}(\mathcal{E}^{\sigma_j}(t,\tau)^{\sigma_j\psi},\ \phi)) = && (*\text{ and Lemma 3.7})\\ &\alpha(\mathcal{Z}(t^{\tau\psi}, \phi)). \end{aligned}$$

We show (2), also working from right to left.

$$\begin{aligned} &\mathcal{AE}^{\sigma}(\alpha(t)^{\tau\psi}, \phi) = && \text{(Definition 4.3)}\\ &\mathcal{AE}^{\sigma}(C^{\mathcal{A}}(\alpha(\mathcal{E}^{\sigma_1}(t,\tau)), \ldots, \alpha(\mathcal{E}^{\sigma_m}(t,\tau)), \alpha(\mathcal{Z}(t,\tau)))^{\tau\psi},\ \phi) = && \text{(Definition 4.4)}\\ &\sqcup(\{\alpha(\mathcal{E}^{\sigma_j}(t,\tau)) \mid \sigma_j\psi = \sigma\} \cup \{\mathcal{AE}^{\sigma}(\alpha(\mathcal{E}^{\sigma_j}(t,\phi))^{\sigma_j\psi},\ \phi) \mid \sigma_j\psi \bowtie \phi\}) =\\ & && (*\text{ and hypothesis})\\ &\sqcup(\bigcup_{\sigma_j\psi=\sigma}\{\alpha(\mathcal{E}^{\sigma_j}(t,\tau))\} \quad \cup \bigcup_{\sigma_j\psi\bowtie\phi}\{\alpha(\mathcal{E}^{\sigma}(\mathcal{E}^{\sigma_j}(t,\tau)^{\sigma_j\psi},\ \phi))\}) =\\ & && (*\text{ and Lemma 3.7})\\ &\alpha(\mathcal{E}^{\sigma}(t^{\tau\psi}, \phi)). \end{aligned}$$

□

Example 4.4 This illustrates Theorem 4.3 for $\phi = \tau\psi = \texttt{List(u)}$ and $\sigma = \texttt{u}$.

$$\begin{aligned} \alpha(\mathcal{Z}([7], \texttt{List(u)})) &= \texttt{Ter} = \mathcal{AZ}(\texttt{List}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}(\texttt{Ter}), \texttt{Ter}), \texttt{List(u)})\\ \alpha(\mathcal{E}^{\texttt{u}}([7], \texttt{List(u)})) &= \texttt{Int}^{\mathcal{A}}(\texttt{Ter}) = \mathcal{AE}^{\texttt{u}}(\texttt{List}^{\mathcal{A}}(\texttt{Int}^{\mathcal{A}}(\texttt{Ter}), \texttt{Ter}), \texttt{List(u)}). \end{aligned}$$

◁

4.3 Abstract Compilation

We now show how the abstract domains can be used in the context of *abstract compilation*. We define an abstract program and show that it is a safe approximation of the concrete program with respect to the usual operational semantics.

In a (normal form) program, each unification is made explicit by an equation. We now define an abstraction of such an equation. Thus we define for each $f \in \Sigma_f$, a predicate which expresses the dependency between $\alpha(f(t_1, \dots, t_n))$ and $\alpha(t_1), \dots, \alpha(t_n)$.

Definition 4.5 [abstract dependency f_{dep}] Let $f_{\langle \tau_1 \dots \tau_n, \tau \rangle} \in \Sigma_f$ where $\tau = C(\bar{u})$ and $\mathcal{N}(\tau) = \langle \sigma_1, \dots, \sigma_m \rangle$. Then $f_{\mathsf{dep}}(C^{\mathcal{A}}(a_1, \dots, a_m, c), b_1, \dots, b_n)$ **holds** if

$$\begin{array}{rcll} a_j & = & \sqcup\left(\{b_i \mid \tau_i = \sigma_j\} \cup \{\mathcal{A}\mathcal{E}^{\sigma_j}(b_i^{\tau_i}, \tau) \mid \tau_i \bowtie \tau\}\right) & \text{for all } j \in \{1, \dots, m\} \quad (1) \\ c & = & \bigwedge_{\tau_i \bowtie \tau} \mathcal{A}\mathcal{Z}(b_i^{\tau_i}, \tau) & \quad (2) \end{array}$$

◁

Example 4.5 To give an idea of how Definition 4.5 translates into code, consider `Cons`. Assuming that $\mathtt{Lub}(a, b, c)$ holds if and only if $c = \sqcup\{a, b\}$, one clause for $\mathtt{Cons}_{\mathsf{dep}}$ might be:

```
Cons_dep(List_a(c,Ter),b,List_a(a,Ter)) <-
   Lub(a,b,c).
```

The first argument of $\mathtt{Cons}_{\mathsf{dep}}$ stands for a list, and the other arguments for the head and tail of this list. Note however that the code is slightly simplified. The reason is that unless the *type* of `a`, `b`, and `c` is specified, there are infinitely many answers for `Lub(a,b,c)`, which causes a termination problem. Therefore, in the implementation, this clause is parametrised with the type of `a`, `b`, and `c`. ◁

Lemma 4.4 If $t = f(t_1, \dots, t_n)$ then $f_{\mathsf{dep}}(\alpha(t), \alpha(t_1), \dots, \alpha(t_n))$ holds.

Proof. Suppose $\mathcal{N}(\tau) = \langle \sigma_1, \dots, \sigma_m \rangle$ and $\tau = C(\bar{u})$. By Definition 4.3

$$\alpha(t) = C^{\mathcal{A}}(\alpha(\mathcal{E}^{\sigma_1}(t, \tau)), \dots, \alpha(\mathcal{E}^{\sigma_m}(t, \tau)), \alpha(\mathcal{Z}(t, \tau))).$$

We must show (1) and (2) in Definition 4.5. First, we prove (1). For each $\sigma_j \in \mathcal{N}(\tau)$,

$$\begin{array}{ll} \alpha(\mathcal{E}^{\sigma_j}(t, \tau)) & \\ \quad = \alpha(\{t_i \mid \tau_i = \sigma_j\} \quad \cup \bigcup_{\tau_i \bowtie \tau} \mathcal{E}^{\sigma_j}(t_i^{\tau_i}, \tau)) & \text{(Lemma 3.4)} \\ \quad = \sqcup\left(\{\alpha(t_i) \mid \tau_i = \sigma_j\} \cup \{\alpha(\mathcal{E}^{\sigma_j}(t_i^{\tau_i}, \tau)) \mid \tau_i \bowtie \tau\}\right) & \text{(moving } \alpha \text{ inwards)} \\ \quad = \sqcup\left(\{\alpha(t_i) \mid \tau_i = \sigma_j\} \cup \{\mathcal{A}\mathcal{E}^{\sigma_j}(\alpha(t_i)^{\tau_i}, \tau) \mid \tau_i \bowtie \tau\}\right) & \text{(Theorem 4.3).} \end{array}$$

Equation (2) is proven in a similar way:

$$\begin{aligned}
&\alpha(\mathcal{Z}(t,\tau)) \\
&\quad = \alpha(\bigwedge_{\tau_i \bowtie \tau} \mathcal{Z}(t_i^{\tau_i},\tau)) && \text{(Lemma 3.4)} \\
&\quad = \bigwedge_{\tau_i \bowtie \tau} \alpha(\mathcal{Z}(t_i^{\tau_i},\tau)) && \text{(moving } \alpha \text{ inwards)} \\
&\quad = \bigwedge_{\tau_i \bowtie \tau} \mathcal{AZ}(\alpha(t_i)^{\tau_i},\tau) && \text{(Theorem 4.3).}
\end{aligned}$$

□

Definition 4.6 [abstraction $\aleph$ of a program] For a normal form equation e define

$$\aleph(e) = \begin{cases} e & \text{if } e \text{ is of the form } x = y \\ f_{\mathsf{dep}}(x, y_1, \ldots, y_n) & \text{if } e \text{ is of the form } x = f(y_1, \ldots, y_n). \end{cases}$$

For a normal form atom a and clause $K = h \leftarrow g_1 \wedge \ldots \wedge g_l$ define

$$\begin{aligned} \aleph(a) &= a \\ \aleph(K) &= \aleph(h) \leftarrow \aleph(g_1) \wedge \ldots \wedge \aleph(g_l). \end{aligned}$$

For a program $P = \langle L, S \rangle$ define

$$\aleph(P) = \{\aleph(K) \mid K \in S\} \cup \{f_{\mathsf{dep}}(a, a_1, \ldots, a_n) \mid f_{\mathsf{dep}}(a, a_1, \ldots, a_n) \text{ holds}\}.$$

◁

Example 4.6 In the following we give the usual recursive clause for `Append` in normal form and its abstraction.

```
%concrete clause                    %abstract clause
Append(xs,ys,zs) <-                 Append(xs,ys,zs) <-
  xs = [x|x1s] &                      Cons_dep(xs,x,x1s) &
  zs = [x|z1s] &                      Cons_dep(zs,x,z1s) &
  Append(x1s,ys,z1s).                 Append(x1s,ys,z1s).
```

◁

We now define the operational semantics of concrete and abstract programs. We assume a fixed language L and program $P = \langle L, S \rangle$, and a left-to-right computation rule. A *program state* is a tuple $\langle G, \theta \rangle$ where G is a query and θ a substitution. It is an initial state if θ is empty. We write $C \in_{\approx} S$ if C is a renamed variant of a clause in S.

Definition 4.7 [reduces to] The relation $\overset{P}{\leadsto}$ ("reduces to") between states is defined by the following rules:

$$\begin{aligned}
\langle h_1 : \ldots : h_l, \theta \rangle &\overset{P}{\leadsto} \langle h_2 : \ldots : h_l, \theta\theta' \rangle && \text{if } h_1 \text{ is `} x = t \text{' and } x\theta\theta' = t\theta\theta' && (1) \\
\langle h_1 : \ldots : h_l, \theta \rangle &\overset{P}{\leadsto} \langle G : h_2 : \ldots : h_l, \theta\theta' \rangle && \text{if } h \leftarrow G \in_{\approx} S \text{ and } h\theta\theta' = h_1\theta\theta' && (2)
\end{aligned}$$

Moreover, $\overset{P}{\leadsto}{}^{j}$ for $j \geq 0$ and $\overset{P}{\leadsto}{}^{*}$ are defined in the usual way. If for an initial query G,

$$\langle G, \emptyset \rangle \overset{P}{\leadsto}{}^{*} \langle p(x_1, \dots, x_n) : H, \theta \rangle \overset{P}{\leadsto}{}^{*} \langle H, \theta' \rangle,$$

we call $p(x_1, \dots, x_n)\theta$ a *call pattern* and $p(x_1, \dots, x_n)\theta'$ an *answer pattern* for p. $\lhd$

Note that it is common to require that θ' is the most general unifier, but nevertheless, our notion of "reduces" with arbitrary unifier has been considered by Lloyd [Llo87].

Theorem 4.5 Let H, H' be queries, θ a substitution and $j \geq 0$. If $\langle H, \emptyset \rangle \overset{P}{\leadsto}{}^{j} \langle H', \theta \rangle$, then $\langle \aleph(H), \emptyset \rangle \overset{\aleph(P)}{\leadsto}{}^{j} \langle \aleph(H'), \theta^{\alpha} \rangle$, where $\theta^{\alpha} = \{x/\alpha(x\theta) \mid x \in \mathit{dom}(\theta)\}$.

PROOF. By Definition 4.7, $\langle H, \emptyset \rangle \overset{P}{\leadsto}{}^{j} \langle H', \theta \rangle$ if and only if $\langle H, \theta \rangle \overset{P}{\leadsto}{}^{j} \langle H', \theta \rangle$, and likewise for $\aleph(P)$. Therefore it is enough to show that for all $j \geq 0$

$$\langle H, \theta \rangle \overset{P}{\leadsto}{}^{j} \langle H', \theta \rangle \qquad \text{implies} \qquad \langle \aleph(H), \theta^{\alpha} \rangle \overset{\aleph(P)}{\leadsto}{}^{j} \langle \aleph(H'), \theta^{\alpha} \rangle. \tag{3}$$

The proof is by induction on j. The base case $j = 0$ holds since

$$\langle \aleph(H), \theta^{\alpha} \rangle \overset{\aleph(P)}{\leadsto}{}^{0} \langle \aleph(H), \theta^{\alpha} \rangle.$$

For the induction step, assume (3) holds for some $j \geq 0$. We show that for every query H''

$$\langle H, \theta \rangle \overset{P}{\leadsto}{}^{j+1} \langle H'', \theta \rangle \qquad \text{implies} \qquad \langle \aleph(H), \theta^{\alpha} \rangle \overset{\aleph(P)}{\leadsto}{}^{j+1} \langle \aleph(H''), \theta^{\alpha} \rangle.$$

If $\langle H, \theta \rangle \overset{P}{\leadsto}{}^{j+1} \langle H'', \theta \rangle$ does *not* hold, the result is trivial. If $\langle H, \theta \rangle \overset{P}{\leadsto}{}^{j+1} \langle H'', \theta \rangle$, then

$$\begin{array}{lllllll} \langle H, \theta \rangle & \overset{P}{\leadsto}{}^{j} & \langle H', \theta \rangle & \overset{P}{\leadsto} & \langle H'', \theta \rangle & & \text{for some query } H'\text{, and} \\ \langle \aleph(H), \theta^{\alpha} \rangle & \overset{\aleph(P)}{\leadsto}{}^{j} & \langle \aleph(H'), \theta^{\alpha} \rangle & & & & \text{by hypothesis.} \end{array}$$

It only remains to be shown that $\langle \aleph(H'), \theta^{\alpha} \rangle \overset{\aleph(P)}{\leadsto} \langle \aleph(H''), \theta^{\alpha} \rangle$. We distinguish two cases depending on whether Rule (1) or (2) of Definition 4.7 was used for the step $\langle H', \theta \rangle \overset{P}{\leadsto} \langle H'', \theta \rangle$.

CASE 1: *Rule* (1) *was used.* $H' = h_1 : \dots : h_l$ where h_1 is '$x = t$', and $t = y$ or $t = f(x_1, \dots, x_n)$. In the first case $\aleph(h_1) = h_1$. Since $x\theta = y\theta$, it follows that $\{x/\alpha(x\theta), y/\alpha(x\theta)\} \subseteq \theta^{\alpha}$ and therefore $x\theta^{\alpha} = y\theta^{\alpha}$. Thus $\langle \aleph(H'), \theta^{\alpha} \rangle \overset{\aleph(P)}{\leadsto} \langle \aleph(H''), \theta^{\alpha} \rangle$ by Rule (1). In the second case $\aleph(h_1) = f_{\mathsf{dep}}(x, x_1, \dots, x_n)$. Since $x\theta = f(x_1\theta, \dots, x_n\theta)$,

$$\{x/\alpha(f(x_1\theta, \dots, x_n\theta)), x_1/\alpha(x_1\theta), \dots, x_n/\alpha(x_n\theta)\} \subseteq \theta^{\alpha}.$$

Hence, by Lemma 4.4, $f_{\mathsf{dep}}(x, x_1, \dots, x_n)\,\theta^{\alpha}$ holds so that $f_{\mathsf{dep}}(x, x_1, \dots, x_n)\,\theta^{\alpha} \in \aleph(P)$ by Definition 4.6. Thus $\langle \aleph(H'), \theta^{\alpha} \rangle \overset{\aleph(P)}{\leadsto} \langle \aleph(H''), \theta^{\alpha} \rangle$ by Rule (2).

CASE 2: *Rule* (2) *was used.* $H' = h_1 : \dots : h_l$ where $h \leftarrow G \in_{\approx} S$ and $h\theta = h_1\theta$. By Definition 4.6, $\aleph(h_1 \leftarrow G) \in_{\approx} \aleph(P)$. Furthermore $\aleph(h)$ has the form $Q(\bar{x})$, and $\aleph(h_1)$ has the form $Q(\bar{y})$. Since $\bar{x}\theta = \bar{y}\theta$ it follows that $Q(\bar{x})\,\theta^{\alpha} = Q(\bar{y})\,\theta^{\alpha}$. □

4.4 Implementation and Results

From now on we refer to the abstract domains defined in this chapter as *typed domains*. We have implemented the mode analysis for object programs in Gödel. This implementation naturally falls into two stages: in the first stage, the language declarations are analysed in order to construct the typed domains, and the program clauses are abstracted. In the second stage, the abstract program is evaluated using standard abstract compilation techniques.

We have implemented the first stage in Gödel, using the Gödel meta-programming facilities. The analysed program may consist of several (system or user-defined) modules, but its abstraction will always be a one-module program. Since virtually all Gödel programs use Gödel system modules[1], these are treated specially in our implementation in order to avoid analysing and abstracting them anew each time.

Gödel meta-programming is slow, but this first stage scales well, as the time for abstracting the clauses of a program is linear in their number. Analysing the type declarations is not a problem in practice. We have analysed contrived programs with extremely complex type declarations within a couple of seconds.

The second stage was implemented in Prolog, so that an existing analyser could be used. Abstract programs produced by the first stage were transformed into Prolog. All call and answer patterns, which may arise in a derivation of an abstract program for a given query, are computed by the analyser. By Theorem 4.5, these patterns correspond to patterns in the derivation of the concrete program. For example a call $\mathtt{p}(\mathtt{Any}, \mathtt{Int}^{\mathcal{A}}(\mathtt{Ter}))$ in a derivation of the abstract program indicates that there may be a call `p(x,7)` in a derivation of the concrete program.

In Table 2, the precision of the typed domain for `Table(Int)` (Example 4.1) is compared with a domain that can only distinguish between ground and non-ground terms. The latter domain has been shown by Codish and Demoen [CD95] to be equivalent to the well-known *Pos* domain [MS93]. The arguments of the predicate `Insert` represent: a table t, a key k, a value v, and a table obtained from t by inserting the node whose key is k and whose value is v. Table 2 shows some initial call patterns and the answer pattern that is inferred for each call pattern. For readability, we use some abbreviations and omit the termination flag for types `Integer`, `Balance` and `String`.

Clearly, inserting a ground node into a ground table gives a ground table. This could be inferred with the ground/non-ground domain (1) as well as the typed domains (3). Now consider the insertion of a node with an *uninstantiated* value into a ground table. With typed domains, it is inferred that the result is still a table but whose values may be uninstantiated (4). This cannot be inferred with a ground/non-ground domain (2). In fact, (2) only says that the answer pattern is no less instantiated than the call pattern, which is trivial.

We used a modified form of the analyser of Heaton et al. [HHK97] running on a Sun SPARC Ultra 170. The analysis times for `Tables` were: (1) 0.09 seconds, (2) 1.57 seconds, (3) 0.81 seconds, (4) 2.03 seconds. Apart from `Tables`, we also analysed some small programs, namely `Append`, `Reverse`, `Flatten` (from the `Nests` module),

[1] In Gödel, all built-ins except the equality predicate are provided via system modules.

Table 2: Some call and answer patterns for `Insert`

Ground/non-ground domain:	
Insert(ground, ground, ground, any) *leads to answer pattern* Insert(ground, ground, ground, ground).	(1)
Insert(ground, ground, any, any) *leads to answer pattern* Insert(ground, ground, any, any).	(2)
Typed domain:	
Insert(Tab$^{\mathcal{A}}$(Int$^{\mathcal{A}}$,Bal$^{\mathcal{A}}$,Str$^{\mathcal{A}}$,Ter),Str$^{\mathcal{A}}$,Int$^{\mathcal{A}}$,Any) *leads to answer pattern* Insert(Tab$^{\mathcal{A}}$(Int$^{\mathcal{A}}$,Bal$^{\mathcal{A}}$,Str$^{\mathcal{A}}$,Ter),Str$^{\mathcal{A}}$,Int$^{\mathcal{A}}$,Tab$^{\mathcal{A}}$(Int$^{\mathcal{A}}$,Bal$^{\mathcal{A}}$,Str$^{\mathcal{A}}$,Ter)).	(3)
Insert(Tab$^{\mathcal{A}}$(Int$^{\mathcal{A}}$,Bal$^{\mathcal{A}}$,Str$^{\mathcal{A}}$,Ter),Str$^{\mathcal{A}}$,Any,Any) *leads to answer pattern* Insert(Tab$^{\mathcal{A}}$(Int$^{\mathcal{A}}$,Bal$^{\mathcal{A}}$,Str$^{\mathcal{A}}$,Ter),Str$^{\mathcal{A}}$,Any,Tab$^{\mathcal{A}}$(Any,Bal$^{\mathcal{A}}$,Str$^{\mathcal{A}}$,Ter)).	(4)

`TreeToList`, `Quicksort`, and `Nqueens`. For these, all analysis times were below 0.03 seconds and thus too small to be very meaningful. For most of these, the typed domains resulted in more precise analyses, similarly as explained for `Tables`.

Our experience is that the domain operations, namely to compute the least upper bound of two abstract terms, are indeed the bottleneck of the analysis. Therefore it is crucial to avoid performing these computations unnecessarily. Also one might compromise some of the precision of the analysis by considering widenings [CC92] for the sake of efficiency. More work could be done on the embedding of the typed domains in the analysis. In order to conduct more experiments, one would need a suite of bigger typed logic programs. A formal comparison between analyses for typed logic programs and untyped ones is of course difficult.

4.5 Discussion and Related Work

We have presented a general domain construction for mode analysis of typed logic programs. For common examples (lists, binary trees), our formalism is simple and yields abstract domains that are comparable to the domains designed by Codish and Demoen [CD94]. In their formalism, however, an abstract domain for obtaining this degree of precision for, say, the types in the `Tables` module, would have to be handcrafted. In contrast, our work describes this construction for arbitrary types.

The fundamental concepts of this work are *recursive type* and *non-recursive subterm type*, which are generalisations of ideas presented previously for lists [CD94]. The resulting abstract domains are entirely in the spirit of previous work by Codish and others [CD94, CL96] and we believe that they provide the highest degree of precision that a generic domain construction should provide. Even if type declarations that require the full generality of our formalism are rare, this work is an important contribution because it helps to understand other, more ad-hoc and pragmatic domain constructions as instances of a general theory. One could always simplify or prune down our abstract domains for the sake of efficiency.

In its full generality the formalism is, admittedly, rather complex. This is primarily due to function declarations where the range type occurs again as a *proper* sub"term" of an argument type, such as the declaration of N in the Nests module (Example 3.2). If types were as widespread in logic programming as they are in functional programming, such declarations would probably not seem very unusual. They are used in the declarations for *rose trees*, that is, trees where the number of children of each node is not fixed [Mee88]. One should also note that while the theory which allows for a domain construction for, say, Nest(Int) is conceptually complex, the computational complexity of the actual domain operations for Nest(Int) is lower than for, say, List(List(List(Int))). In short, the complexity of the abstract domains depends on the inherent complexity of the type declarations, as illustrated by the type graphs (Figure 6).

We have built on ideas presented previously for untyped languages [CL96]. Notably the title of that work says that *type*, not *mode*, dependencies are derived. Even in an untyped language such as Prolog, one can define types as sets of terms given by some kind of "declaration", just as in a typed language [AL94]. In this case type analysis (inferring that an argument is instantiated to a term *of a certain type*) is inseparable from mode analysis. The analysis must account for "incorrectly" typed terms such as [3|17]. As it cannot be assumed that, say, [3|y] will eventually be bound to a list, it is abstracted as any, thus not capturing that it is at least *partially* instantiated. In typed languages, this problem does not arise. It seems that Codish and Lagoon [CL96] provide a straightforward domain construction for *arbitrary* types, but this is not the case. It is not specified what kind of "declarations" are implied, but the examples and theory suggest that all types are essentially lists and trees. The Tables and Nests examples given in Section 3.2 are not captured.

Recursive modes [TL97] characterise that the left spine, right spine, or both, of a term are instantiated. The authors admit that this may be considered an ad-hoc choice, but on the other hand, they present good experimental results. They do not assume a typed language and thus cannot exploit type declarations in order to provide a more generic concept of *recursive modes*, as we have done by the concept of *termination*. Also, the degree of instantiation that we would express by, say, $\mathtt{List}^{\mathcal{A}}(\mathtt{Table}^{\mathcal{A}}(\mathtt{Any},\mathtt{Ter}),\mathtt{Ter})$, cannot be characterised.

A complex system for type analysis of Prolog has been presented by Van Hentenryck et al. [VCL95]. As far as we can see, this system is not in a formal sense stronger or weaker than our mode analysis. The domain Pat(Type) used there is infinite, so that widenings have to be introduced to ensure finiteness, and "the design of widening operators is experimental in nature" [VCL95]. In contrast, we exploit the type declarations to construct domains that are inherently finite and whose size is dictated by the complexity of the type declarations. Similarly, in a paper by Janssens and Bruynooghe [JB92], the finiteness of abstract domains and terms is ensured by imposing an ad-hoc bound on the number of symbols.

Barbuti and Giacobazzi have presented a polymorphic type inference for (untyped) logic programs [BG92]. It is assumed that type declarations are given to define a language of "well-typed" terms, similarly as in typed logic programming languages. However, the types of the predicate symbols are not declared, but rather inferred. In particular, it might be inferred that some arguments of a predicate are not "well-typed". Such information can be useful for debugging programs.

Gallagher et al. have shown that the domain construction of any (static) program analysis can be cast in terms of *pre-interpretations* [GBS95]. Traditionally, pre-interpretations are used in predicate logic to assign a semantic value to a term, for example the number '2' to the term 1+1 or 2. However, they can also be used to specify a program analysis, by choosing an appropriate domain on which these pre-interpretations operate. The mode analysis we have presented here can without doubt also be expressed in these terms, by choosing as domains the abstract domains we propose here.

Mercury [SHC96] has a mode system based on *instantiation states*. These are assertions of how instantiated a term is. An instantiation state is similar to an abstract term. Indeed, given some type declarations, it is possible to define an instantiation state in Mercury syntax which, while not being exactly the same, is comparable in precision to an abstract term in our formalism. In Mercury, it is the *user* who has to specify a set of instantiation states by declaring the mode, and this mode is checked and enforced by the compiler. In contrast, we have described how the abstract terms and their values can be inferred automatically.

The Mercury compiler also does some mode inference. It is hard to assess whether or not the compiler can actually construct instantiation states without any help by mode declarations because the relevant literature [Hen92, Som87] only refers to simple examples and does not specify the mode inference precisely.

It has been noted by Henderson [Hen92] that instantiation states loosely correspond to abstract interpretation, used for mode analysis in a language such as Gödel, which does not *enforce* modes. In this part of the thesis we developed this argument. Our domain construction can be regarded as inferring automatically, from a set of type declarations, what the interesting instantiation states are.

The mode system in Mercury is based on work by Somogyi [Som87], where the Simple Range Condition and the Reflexive Condition that we impose are not explicitly required. However, Somogyi does not define the type system precisely, instead referring to Mycroft and O'Keefe [MO84], whose formal results have been shown to be incorrect, namely in ignoring the *transparency* condition [Hil93, HT92]. It is therefore difficult to assess whether that approach would work for programs which violate these conditions. We know of no real Gödel programs that violate either of the Simple Range or Reflexive Conditions. We have found that violating the Reflexive Condition raises fundamental questions about decidability in typed languages, which seem to be related to the concept of *polymorphic recursion* [Kah96, KTU93]. It would be interesting to investigate these questions further.

We believe that, since our abstract terms can characterise the instantiation of a term with what might be called a "reasonable" degree of precision, they could provide a good basis for two further applications: *declaring* modes and declaring conditions for delaying.

Concerning the first application, note that the present Mercury implementation does not support instantiation states in their full generality, and it is hard to imagine that this would ever be needed. Thus one might consider a language where modes are declared using our abstract terms.

In Gödel, the delay declarations which state that a predicate is delayed until an argument (or a subterm of the argument) is ground or non-variable, cannot describe the behaviour of the Gödel system predicates precisely. We have observed that, typically, the degree of instantiation for a Gödel system predicate to run safely without delaying could be specified by an abstract term in our typed domains. For example, the predicate `Append/3` will run safely if the first argument is a nil-terminated list.

Our approach may also be applicable to untyped languages, if we have information at hand that is similar to type declarations. Such information might be obtained by inferring declarations [Chr97] or from declarations as comments [SG95b]. Certainly our analysis would then regain aspects of *type* rather than *mode* inference, which it had lost by transferring the approach to typed languages.

Part III

Non-Standard Derivations

Chapter 5

Correctness Properties of Programs

In this chapter, the need for non-standard derivations is motivated. Then several correctness properties for programs concerning the modes and types are introduced. These properties will be used throughout Part III.

5.1 Why Non-Standard Derivations?

The paradigm of logic programming is based on giving a computational interpretation to a certain fragment of first order logic. Kowalski [Kow79] advocates the separation of the *logic* and *control* aspects of a logic program and has coined the famous formula

$$\text{Algorithm} = \text{Logic} + \text{Control}.$$

The programmer should be responsible for the logic part, and hence a logic program should be a (first order logic) specification. The control should be taken care of by the logic programming system.

In reality, logic programming is far from this ideal. Without the programmer being aware of the control and writing programs accordingly, logic programs would usually be hopelessly inefficient or even non-terminating.

One aspect of control in logic programs is the *selection rule.* This is a rule stating which atom in a query is selected in each derivation step. The standard selection rule is the *LD* selection rule: in each derivation step, the leftmost atom in a query is selected for resolution. This selection rule is based on the assumption that programs are written in such a way that the data flow within a query or clause body is from left to right.

Example 5.1 Consider the program in Figure 9 and the following derivation, where the selected atom is underlined in each query:[1]

$$\begin{array}{l}
\underline{\texttt{permute}([1],\texttt{As})} \rightsquigarrow \\
\underline{\texttt{permute}([],\texttt{Z}')},\ \texttt{delete}(1,\texttt{As},\texttt{Z}') \rightsquigarrow \\
\underline{\texttt{delete}(1,\texttt{As},[])} \rightsquigarrow \Box.
\end{array}$$

[1] In examples, we use $\rightsquigarrow$ to denote derivation steps.

```
permute([],[]).                    delete(X,[X|Z],Z).
permute([U|X],Y) :-                delete(X,[U|Y],[U|Z]) :-
  permute(X,Z),                      delete(X,Y,Z).
  delete(U,Y,Z).
```

Figure 9: The `permute` program

```
append([],Y,Y).
append([X|Xs],Ys,[X|Zs]) :-
  append(Xs,Ys,Zs).
```

Figure 10: The `append` program

In the second line, $\mathtt{Z}'$ is an output argument of $\mathtt{permute}([],\mathtt{Z}')$. The process of resolving this atom instantiates $\mathtt{Z}'$ to $[]$, which is used by the atom $\mathtt{delete}(1,\mathtt{As},\mathtt{Z}')$ as input. Hence the data flow is from left to right. ◁

Observe that the notion of data flow is based on the idea that some argument positions serve as *input* positions and others as *output* positions. In the above example, the first argument of `permute` is input and the second is output.

The LD selection rule ensures for this example that atoms are only selected when they have a certain degree of instantiation. The following example shows that this is crucial in order to ensure essential properties, in particular termination.

Example 5.2 Consider the usual `append` program given in Figure 10 and the following derivation where the rightmost atom is always selected:

$$\begin{array}{l}
\mathtt{append}([1],[],\mathtt{As}),\ \underline{\mathtt{append}(\mathtt{As},[],\mathtt{Bs})} \rightsquigarrow \\
\mathtt{append}([1],[],[\mathtt{X}'|\mathtt{As}']),\ \underline{\mathtt{append}(\mathtt{As}',[],\mathtt{Bs}')} \rightsquigarrow \\
\mathtt{append}([1],[],[\mathtt{X}',\mathtt{X}''|\mathtt{As}'']),\ \underline{\mathtt{append}(\mathtt{As}'',[],\mathtt{Bs}'')} \rightsquigarrow \ldots
\end{array}$$

The derivation is infinite although there are only finitely many answers to the query. For this example, the natural data flow would be from left to right. In fact, all derivations terminate if the LD selection rule is assumed. ◁

The LD selection rule is so established in logic programming that we have to justify why we consider other selection rules. There are at least four purposes for which other selection rules are useful: using predicates in multiple modes, parallel execution [AL95], the test-and-generate paradigm [Nai92], and some programs using accumulators [EG99]. For motivation, we give an example of the first purpose.

Example 5.3 Consider again the `permute` program (Figure 9). In the following derivation, the rightmost atom is selected in each step. The data flow is from right to left.

$$\begin{array}{l}
\underline{\texttt{permute(As,[1])}} \leadsto \\
\texttt{permute(X}',\texttt{Z}'\texttt{)},\ \underline{\texttt{delete(U}',\texttt{[1]},\texttt{Z}'\texttt{)}} \leadsto \\
\underline{\texttt{permute(X}',\texttt{[])}} \leadsto \Box.
\end{array}$$

In this example, the second argument of `permute` is input and the first is output. ◁

To allow for `permute` to be used in both modes, we need a selection rule which is more flexible than just stating that the leftmost or rightmost atom should be selected in each step. Several logic programming languages provide *delay declarations* for this purpose [HL94, SIC98, SHC96]. Using delay declarations, the user can specify a degree to which an atom must be instantiated in order to be selected.

Note that while delay declarations give the programmer some control, they do not specify precisely which atom is selected in each step, since there could be more than one atom which is sufficiently instantiated to be selected.

In the literature, the need for sufficient instantiation of the selected atom and hence the purpose of delay declarations is usually explained as "ensuring termination" and "preventing runtime errors related to built-in predicates" [AL95, Lüt93, MT95, MK97, Nai92]. Taking a more abstract viewpoint, one can characterise the minimal and most important purpose of delay declarations as follows:

> Delay declarations should ensure that in each derivation step, the input arguments of the selected atom cannot become instantiated.

In other words, an atom in a query can only be selected when it is sufficiently instantiated so that the most general unifier (MGU) with the clause head does not bind the input arguments of the atom. We call derivations which meet this requirement *input-consuming.*

Input-consuming derivations reflect the natural meaning of "input". The concept is useful because it abstracts from the technical details of particular delay constructs. Wherever possible we formulate results in terms of input-consuming derivations rather than in terms of delay declarations.

Note that for the query in Example 5.2, all derivations are input-consuming if the LD selection rule is assumed. In this and the following chapter, we do not worry about how input-consuming derivations can be achieved in existing implementations. In Chapter 7, we show how input-consuming derivations can be achieved using delay declarations.

This chapter is organised as follows. The next section defines some notation and terminology. Section 5.3 introduces a formalism consisting of a permutation for each clause in a program, which indicates the direction of data flow in this clause. Section 5.4 introduces permutation nicely moded programs. Section 5.5 introduces permutation well moded programs. Section 5.6 introduces permutation well typed programs. Section 5.7 defines a property called *type consistency.*

5.2 Notation and Terminology

We use standard notations of logic programming [Apt97, Llo87]. Our special notations related to modes and types follow Etalle et al. [EBC99] and Apt and Luitjes [AL95]. For the examples we use Prolog syntax. We recall some important notions.

The set of variables in a syntactic object o is denoted as $vars(o)$. A syntactic object is **linear** if every variable occurs in it at most once. A substitution is **idempotent** if $\sigma\sigma = \sigma$. Throughout Part III, we only consider idempotent substitutions. The **domain** of a substitution σ is $dom(\sigma) = \{x \mid x\sigma \neq x\}$. The **range** of a substitution σ is $ran(\sigma) = \{x\sigma \mid x \in dom(\sigma)\}$.

We say that a term u occurs **directly** in a vector of terms $\mathbf{t}$, or equivalently, u **fills a position in t**, if u is one of the terms of $\mathbf{t}$. (For example, a occurs directly in (a, b) but not in $(f(a), b)$.) A **flat** term is a variable or a term $f(x_1, \ldots, x_n)$, where $n \geq 0$ and the x_i are distinct variables.

For a predicate p/n, a **mode** is an atom $p(m_1, \ldots, m_n)$, where $m_i \in \{I, O\}$ for $i \in \{1, \ldots, n\}$. Positions with I are called **input positions**, and positions with O are called **output positions** of p. To simplify the notation, an atom written as $p(\mathbf{s}, \mathbf{t})$ means: $\mathbf{s}$ is the vector of terms filling the input positions, and $\mathbf{t}$ is the vector of terms filling the output positions. An atom $p(\mathbf{s}, \mathbf{t})$ is **input-linear** if $\mathbf{s}$ is linear. A **mode of a program** is a set of modes, one mode for each of its predicates.[2] A program can have several modes, so whenever we refer to the input and output positions, this is always with respect to one particular mode which is clear from the context.

A **type** is a set of terms closed under instantiation. A **non-variable type** is a type that does not contain variables. *The* **variable type** is the type that contains variables and hence, as it is instantiation closed, all terms. A **ground type** is a type that contains only ground terms. A **constant type** is a ground type that contains only (possibly infinitely many) constants. In the examples, we use the following types: *any* is the variable type, *all_ground* the type containing all ground terms, *list* the non-variable type of (nil-terminated) lists, *int* the constant type of integers, *il* the ground type of integer lists, *num* the constant type of numbers, *nl* the ground type of number lists, and finally, *tree* is the non-variable type defined by the context-free grammar $\{tree \rightarrow \texttt{leaf}; tree \rightarrow \texttt{node}(tree, any, tree)\}$. These types are also shown in Table 3.

We write $t : T$ for "t is in type T". We use $\mathbf{S}$, $\mathbf{T}$ to denote vectors of types, and write $\models \mathbf{s} : \mathbf{S} \Rightarrow \mathbf{t} : \mathbf{T}$ if for all substitutions σ, $\mathbf{s}\sigma : \mathbf{S}$ implies $\mathbf{t}\sigma : \mathbf{T}$. It is assumed that each argument position of each predicate p/n has a type associated with it. These types are indicated by writing the atom $p(T_1, \ldots, T_n)$ where $T_1, \ldots, T_n$ are types. The **type of a program** P is a set of such atoms, one for each predicate defined in P. An atom (query) is **correctly typed** if each argument position is filled with a term of the type of that position. A term t is **type-consistent** [DM98] **with respect to** T if there is a substitution θ such that $t\theta : T$. A term t occurring in an atom in some position is **type-consistent** if it is type-consistent with respect to the type of that position.

A **query** is a finite sequence of atoms. Atoms are denoted by a, b, h, queries by B, F, H, Q, R. We write $a \in B$ if a is an atom in B. Sometimes we say "atom"

[2] We discuss a more general notion of mode in Section 10.3.

Table 3: Some common types

Name	Description	Property
any	variable type	variable
all_ground	all ground terms	ground
list	(nil-terminated) lists	non-variable
int	integers	ground
il	integer lists	ground
num	numbers	ground
nl	number lists	ground
tree	$\{tree \rightarrow \texttt{leaf}; tree \rightarrow \texttt{node}(tree, any, tree)\}$	non-variable

instead of "query consisting of an atom". If $a_1, \ldots, a_n$ is a query, then $a_{i_1}, \ldots, a_{i_m}$, where $1 \leq i_1 < \ldots < i_m \leq n$, is a **subquery** of $a_1, \ldots, a_n$.

A **derivation step** for a program P is a pair $\langle Q, \theta\rangle; \langle R, \theta\sigma\rangle$, where $Q = Q_1, p(\mathbf{s}, \mathbf{t}), Q_2$ and $R = Q_1, B, Q_2$ are queries; θ is a substitution; $p(\mathbf{v}, \mathbf{u}) \leftarrow B$ a renamed variant of a clause in P; and σ the MGU[3] of $p(\mathbf{s}, \mathbf{t})\theta$ and $p(\mathbf{v}, \mathbf{u})$. We call $p(\mathbf{s}, \mathbf{t})\theta$ (or $p(\mathbf{s}, \mathbf{t})$)[4] the **selected atom** and $R\theta\sigma$ the **resolvent** of $Q\theta$ and $h \leftarrow B$. We call $R\theta\sigma$ an **LD**-resolvent if Q_1 is empty. A derivation step is **input-consuming** if $dom(\sigma) \cap vars(\mathbf{s}\theta) = \emptyset$.

A **derivation** ξ **for** a program P is a sequence $\langle Q_0, \theta_0\rangle; \langle Q_1, \theta_1\rangle; \ldots$ where each pair $\langle Q_i, \theta_i\rangle; \langle Q_{i+1}, \theta_{i+1}\rangle$ in ξ is a derivation step.[5] Alternatively, we also say that ξ is a **derivation of** $P \cup \{Q_0\theta_0\}$. We sometimes denote a derivation as $Q_0\theta_0; Q_1\theta_1; \ldots$. An **LD**-derivation is a derivation where the selected atom is always the leftmost atom in a query. An **input-consuming** derivation is a derivation consisting of input-consuming derivation steps.

A **selection rule** $\mathcal{R}$ is a set of derivations closed under prefixes, that is, if $\xi \in \mathcal{R}$, then for any prefix ξ' of ξ, we have $\xi' \in \mathcal{R}$. If $\xi \in \mathcal{R}$, we say that ξ is an $\mathcal{R}$**-derivation**.[6]

If (F, a, H); $(F, B, H)\theta$ is a step in a derivation, then each atom in $B\theta$ (or B)[4] is a **direct descendant** of a, and for all $b \in F, H$, we say $b\theta$ (or b)[4] is a **direct descendant** of b. We say b is a **descendant of** a if (b, a) is in the reflexive, transitive closure of the relation *is a direct descendant*. The descendants of a *set* of atoms are defined in the obvious way. Consider a derivation $Q_0; \ldots; Q_i; \ldots; Q_j; Q_{j+1}; \ldots$. We call $Q_j; Q_{j+1}$ an a**-step** if $a \in Q_i$ and the selected atom in $Q_j; Q_{j+1}$ is a descendant of a.

[3] The MGU is not unique. It is however unique up to renaming [Llo87], which is why we simply speak of *the* MGU. We assume that whenever possible, an MGU is chosen which does not bind $\mathbf{s}$.

[4] Whether or not the substitution has been applied is always clear from the context.

[5] This definition follows Lloyd [Llo87]. Apt requires that the sequence is *maximal* [Apt97].

[6] This definition is more general than the definitions by Lloyd [Llo87] and Apt [Apt97]. See also Subsection 11.1.13.

5.3 Modes and Permutations

Apt and Luitjes [AL95] consider four correctness properties for programs: *nicely moded*, *well moded*, *well typed*, and *simply moded*. Nicely-modedness is used to show that the occur-check can be safely omitted. Well-modedness and well-typedness are used to show that derivations do not flounder. Finally, simply-modedness is a special case of nicely-modedness and is used to show that a program is free from errors related to built-ins. Other authors have also used these or similar correctness properties, for example to show that programs are unification free [AE93], successful [BC99], and terminating [EBC99]. In Example 1.5, we have given a flavour of these correctness properties.

In this part of the thesis, we make extensive use of these correctness properties and also define two new ones. In Section 7.5, we will give an overview summarising the relationships between them.

In order to be useful for verification of programs assuming non-standard derivations, these properties must be generalised. We now discuss the basis of this generalisation.

5.3.1 The Order of the Atoms in a Query

In a query (clause body) one can consider three different orderings among the atoms.

First, there is the *textual* order. This does not need any explanation.

Secondly, there is the *producer-consumer relation* [KKS91] between atoms. A pair of atoms (a, b) is in the producer-consumer relation if a has a variable in an output position which b has in an input position. The correctness properties we define will ensure that the transitive closure of this relation is anti-symmetric. We shall refer to any order $<$ such that (a, b) is in the producer-consumer relation only if $a < b$ as *producer-consumer* order. Note that we neglect the fact that this order is not necessarily unique, since any producer-consumer order will do for our purposes.

Thirdly, there is the execution order, which depends on the selection rule.

In the case of LD-derivations, all of these orders are usually identical. The definitions of the above correctness properties as they are used in most works [AE93, BC99, EBC99] are based on this assumption. Otherwise, these orders may differ.

Example 5.4 Consider $\texttt{append}(I, I, O)$ (Figure 10 on page 57) and the following derivation, where we annotate the atoms with superscripts so that we can refer to them:

$$\begin{array}{l}
\texttt{append(As,[],Bs)}^{1.1},\ \underline{\texttt{append([1],[],As)}}^{2.1} \leadsto \\
\underline{\texttt{append([1|As'],[],Bs)}}^{1.1},\ \texttt{append([],[],As')}^{2.2} \leadsto \\
\texttt{append(As',[],Bs')}^{1.2},\ \underline{\texttt{append([],[],As')}}^{2.2} \leadsto \\
\underline{\texttt{append([],[],Bs')}}^{1.2} \leadsto \Box.
\end{array}$$

In each query, the producer-consumer order is the converse of the textual order. Concerning the execution order, note that atom 2.1 is selected for resolution before atom 1.1, but then atom 1.1 is selected, even before atom 2.1 is resolved away completely, that is, before all descendants of atom 2.1 are resolved. We say that the computations for the two atoms *interleave* or *coroutine*. ◁

To formalise the producer-consumer order, we associate, with each query and each clause in a program, a permutation π of the (body) atoms, which gives the producer-consumer order. That is, if (a_i, a_j) is in the producer-consumer relation, then $\pi(i) < \pi(j)$. This permutation depends on the mode. For different modes, the permutations are different.

This formalism has been proposed previously by Boye [Boy96]. Hoarau and Mesnard have developed a similar formalism for the purpose of reordering atoms in clause bodies automatically to ensure termination [HM99].

5.3.2 Are those Permutations Really Necessary?

The previous subsection raises two questions:

1. Could the textual order not be identical to the producer-consumer order?
2. Could we not *pretend* that the textual order is identical to the producer-consumer order, to simplify the notation?

Judging from the literature [AL95, Nai92] but also from personal communication we believe that it is not widely recognised that these question must be distinguished.

To answer the first question, compare the derivations for `permute` in Examples 5.1 and 5.3. Here we have a single program which can be used in two distinct modes. Depending on the mode, the producer-consumer order in each query (clause body) is different, whereas the textual order is always the same. Therefore, it is impossible that the textual order is always identical to the producer-consumer order. It has been proposed to solve this problem by generating a specialised version of a program for each mode, such that for each version, the textual order is always identical to the producer-consumer order [SHC96]. However, doing so implies a strict loss of generality, in the sense that we are not considering one single program running in several modes.

Although other authors [AL95, Nai92], in the context of delay declarations, have not explicitly assumed multiple modes, they mainly give examples where delay declarations are clearly used for that purpose (see page 133). Whether allowing multiple modes is a good approach or whether it is better to generate multiple versions of each predicate is an ongoing discussion [Hil98].

Even without assuming multiple modes, the textual order cannot always be identical to the producer-consumer order. For example, programs that use the test-and-generate paradigm rely on the atom which tests ("consumes") occurring to the left of the atom which generates ("produces"). We will see such a program in Figure 22 on page 106.

So the answer to the first question is: no, the textual order cannot always be identical to the producer-consumer order.

The answer to the second question is less clearcut. It depends on what kind of selection rule we consider.

Some authors have studied derivations where the textual order is irrelevant for the selection of an atom and hence for the execution order [AL95, Lüt93, MT95]. Therefore, one may assume for the sake of notational convenience that in fact the textual order is identical to the producer-consumer order. Although not explicitly stated, the definitions of the above correctness properties as they are used by Apt and Luitjes [AL95] are based on this assumption. More precisely, any result stated there can be generalised trivially to programs where the atoms in the clause bodies are permuted in an arbitrary way.

The same holds for many of the results presented in this thesis, and we will therefore also sometimes adopt this simplifying assumption, in particular in Chapter 6. Also in this chapter, we consider results for which the textual order of atoms is irrelevant. Nevertheless, we maintain the permutations to make the results easily applicable in other parts of the thesis.

Whenever we consider derivations where the textual order of atoms is irrelevant, we do not have to treat multiple modes explicitly. We can *pretend* that there is a renamed version of each predicate for each mode, such that in all clauses, the textual order is identical to the producer-consumer order. This is not a loss of generality, but merely a notational convenience. In the actual code, there is still only one version of each predicate.

Of course, when we consider input-consuming derivations, the selection rule must "know" what mode is assumed in a particular execution of the program, since otherwise it would not be defined what an input-consuming derivation is. This can be realised with delay declarations, as we will see in Chapter 7.

In Chapter 8, we will study *left-based* derivations, for which the textual order is relevant for the execution order. For left-based derivations, the textual order has to be taken into account as it is. It is not correct to make a simplifying assumption about it.

5.3.3 Uniqueness of Derived Permutations

As explained in Subsection 5.3.1, we associate, with each query and each clause in a program, a permutation of the (body) atoms, which gives the producer-consumer order. We will later define correctness properties which are parametrised by these permutations. However, some statements only depend on the permutations themselves and not on the correctness property considered. To avoid repeating virtually identical statements, we formulate these statements here in a general way.

In this subsection, we assume a program P where a permutation is associated with each clause, and an initial query Q that also has a permutation associated with it. We call Q or a clause in P **π-ordered** if the permutation associated with it is π. Later, *π-ordered* will be replaced with *π-nicely moded*, *π-well typed* etc. The π is omitted whenever π is the identity.

Let π be a permutation on $\{1, \dots, n\}$. For notational convenience we extend the domain of π by defining $\pi(i) = i$ whenever $i \notin \{1, \dots, n\}$. In examples, π is written as $\langle \pi(1), \dots, \pi(n) \rangle$. Also, we write $\pi(o_1, \dots, o_n)$ for the sequence obtained by applying π to the sequence $o_1, \dots, o_n$, that is $o_{\pi^{-1}(1)}, \dots, o_{\pi^{-1}(n)}$. For example, if $Q = a_1, a_2, a_3, a_4$ is a query and $\pi = \langle 4, 3, 1, 2 \rangle$, then $\pi(Q) = a_3, a_4, a_2, a_1$. Note that if $n \leq 1$, then a permutation on $\{1, \dots, n\}$ is necessarily the identity.

We now define the permutation associated with any query occurring in a derivation of $P \cup \{Q\}$. This is defined inductively. Given a π-ordered query and a ρ-ordered clause, the permutation associated with the resolvent is derived from π and ρ in a natural way.

Definition 5.1 [derived permutation] Let $Q' = a_1, \dots, a_n$ be a π-ordered query and $C = h \leftarrow b_1, \dots, b_m$ be a ρ-ordered clause. Suppose for some $k \in \{1, \dots, n\}$, h and

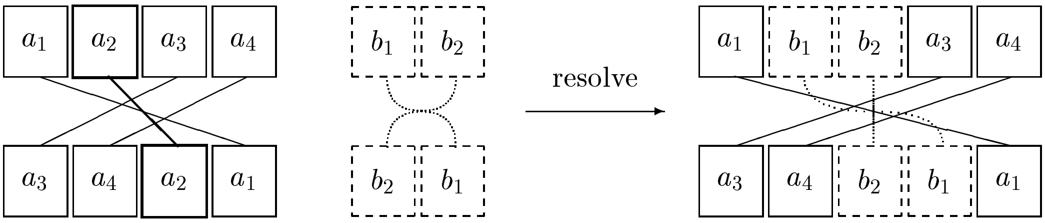

Figure 11: The derived permutation $Der(\pi, \rho, k)$

a_k are unifiable. Then we say that the resolvent of Q' and C with selected atom a_k is **ϱ-ordered**, where ϱ is a permutation on $\{1, \dots, n+m-1\}$ defined by

$$\varrho(i) = \begin{cases} \pi(i) & \text{if } i < k,\ \pi(i) < \pi(k) \\ \pi(i) + m - 1 & \text{if } i < k,\ \pi(i) > \pi(k) \\ \pi(k) + \rho(i - k + 1) - 1 & \text{if } k \le i < k + m \\ \pi(i - m + 1) & \text{if } k + m \le i < n + m,\ \pi(i - m + 1) < \pi(k) \\ \pi(i - m + 1) + m - 1 & \text{if } k + m \le i < n + m,\ \pi(i - m + 1) > \pi(k). \end{cases}$$

We call ϱ the **derived permutation** and write $Der(\pi, \rho, k) = \varrho$. ◁

Figure 11 illustrates the derived permutation when $n = 4$, $\pi = \langle 4, 3, 1, 2 \rangle$, $m = 2$, $\rho = \langle 2, 1 \rangle$, and $k = 2$. By Definition 5.1, we have $Der(\pi, \rho, k) = \langle 5, 4, 3, 1, 2 \rangle$, since

$$\begin{array}{rlll} Der(\pi,\rho,k)(1) = & \pi(1) + 2 - 1 = & 5 & \text{(2nd line)} \\ Der(\pi,\rho,k)(2) = & \pi(2) + \rho(2 - 2 + 1) - 1 = & 4 & \text{(3rd line)} \\ Der(\pi,\rho,k)(3) = & \pi(2) + \rho(3 - 2 + 1) - 1 = & 3 & \text{(3rd line)} \\ Der(\pi,\rho,k)(4) = & \pi(4 - 2 + 1) = & 1 & \text{(4th line)} \\ Der(\pi,\rho,k)(5) = & \pi(5 - 2 + 1) = & 2 & \text{(4th line).} \end{array}$$

Observe also that in the trivial case that π and ρ are the identity, $Der(\pi, \rho, k)$ is also the identity, for all $k \in \{1, \dots, n\}$.

Throughout Part III, we will frequently consider a derivation $Q_1; \dots; Q_n$ such that Q_1 is π_1-ordered and Q_n is π_n-ordered, where "ordered" is replaced with "nicely moded", "well typed", etc. Whenever we do this, we imply that π_n is uniquely determined. More precisely, we imply that there are indices $k_1, \dots, k_n$ and permutations $\pi_1, \dots, \pi_n$ and $\rho_1, \dots, \rho_{n-1}$ such that for each $i \in \{1, \dots, n-1\}$

- Q_i is π_i-ordered,
- the ${k_i}^{\text{th}}$ atom in Q_i is selected in the step $Q_i; Q_{i+1}$,
- the clause used in the step $Q_i; Q_{i+1}$ is ρ_i-ordered,
- $\pi_{i+1} = Der(\pi_i, \rho_i, k_i)$.

This is important to stress because the uniqueness of the permutation π_n will not necessarily follow from the definitions of the correctness properties. However, as stated in Subsection 5.3.1, it is no loss of generality to assume that the producer-consumer order is unique.

At each step of a derivation, the relative order of atoms given by the derived permutation is preserved. The following lemma formalises this.

Lemma 5.1 Let $Q; \ldots; R$ be a derivation for P, where $Q = a_1, \ldots, a_n$ is π-ordered and $R = b_1, \ldots, b_m$ is ρ-ordered.

a. Let $i, j \in \{1, \ldots, n\}$ such that $\pi(i) < \pi(j)$. Then for all $k, l \in \{1, \ldots, m\}$ such that b_k is a descendant of a_i and b_l is a descendant of a_j, we have $\rho(k) < \rho(l)$.

b. Let $k, l \in \{1, \ldots, m\}$ such that $\rho(k) < \rho(l)$, and let $i, j \in \{1, \ldots, n\}$ such that b_k is a descendant of a_i and b_l is a descendant of a_j (note that i and j exist and are unique). Then $\pi(i) \leq \pi(j)$.

Proof. Inspection of the derived permutation in Definition 5.1 shows that the result holds for derivations of length 1. The general result follows by a straightforward induction on the length. □

In the trivial case that all permutations are the identity, the above lemma merely states that resolution preserves the textual order of atoms in a query.

5.4 Permutation Nicely Moded Programs

Apt and Luitjes define *nicely moded* queries [AL95]. In a nicely moded query, a variable occurring in an input position does not occur later in an output position, and each variable in an output position occurs only once. We generalise this to *permutation nicely moded.*

Definition 5.2 [permutation nicely moded] Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a query and π a permutation on $\{1, \ldots, n\}$. Then Q is **π-nicely moded** if $\mathbf{t}_1, \ldots, \mathbf{t}_n$ is a linear vector of terms and for all $i \in \{1, \ldots, n\}$

$$vars(\mathbf{s}_i) \cap \bigcup_{\pi(i) \leq \pi(j) \leq n} vars(\mathbf{t}_j) = \emptyset.$$

The query $\pi(Q)$ is a **nicely moded query corresponding to** Q.

The clause $C = p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$ is **π-nicely moded** if Q is π-nicely moded and

$$vars(\mathbf{t}_0) \cap \bigcup_{j=1}^{n} vars(\mathbf{t}_j) = \emptyset.$$

The clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow \pi(Q)$ is a **nicely moded clause corresponding to** C.

A query (clause) is **permutation nicely moded** if it is π-nicely moded for some π. A program P is **permutation nicely moded** if all of its clauses are. A **nicely moded program corresponding to** P is a program obtained from P by replacing each clause C in P with a nicely moded clause corresponding to C. ◁

Note that in the clause head, the letter t is used for input and s is used for output, whereas in the body atoms it is vice versa. This convention is used throughout because it allows for a succinct notation, in particular in Definitions 5.4, 5.5 and 7.4.

Note also that a one-atom query $p(\mathbf{s}, \mathbf{t})$ is (permutation) nicely moded if and only if $vars(\mathbf{s}) \cap vars(\mathbf{t}) = \emptyset$ and $\mathbf{t}$ is linear.

For many results it is necessary to require that each clause head is input-linear.

Definition 5.3 [input-linear clause/program] A clause $C = p(\mathbf{t}, \mathbf{s}) \leftarrow Q$ is **input-linear** if $\mathbf{t}$ is input-linear. A program is **input-linear** if all of its clauses are input-linear and it contains no uses of `=`(I, I). ◁

Note that in the above definition, uses of the built-in equality predicate are taken into account. Conceptually, the equality predicate is defined as "`X = X.`". Therefore, an input-linear program must not use the equality predicate in mode `=`(I, I), since the clause "`X = X.`" is not input-linear for this mode. This is discussed further in Section 10.2.

Example 5.5 Consider the `permute` program (Figure 9 on page 57). For the mode $\{$`permute`(I, O), `delete`$(I, O, I)\}$, this program is nicely moded and input-linear.

In mode $\{$`permute`(O, I), `delete`$(O, I, O)\}$, it is permutation nicely moded and input-linear. The second clause for `permute` is $\langle 2, 1\rangle$-nicely moded, and the other clauses are nicely moded.

In "test mode", that is, $\{$`permute`(I, I), `delete`$(I, I, O)\}$, it is permutation nicely moded, but not input-linear, because the first clause for `delete` is not input-linear. The second clause for `permute` is $\langle 2, 1\rangle$-nicely moded, and the other clauses are nicely moded. ◁

The problem of *finding* a mode for a program so that it is nicely moded has been considered by Chadha and Plaisted [CP91].

We quote the following persistence property for nicely-modedness.

Lemma 5.2 [AL95, Lemma 11] Let Q be a nicely moded query and C be a nicely moded, input-linear clause where $vars(Q) \cap vars(C) = \emptyset$. Then every resolvent of Q and C is nicely moded.

We generalise this result to permutation nicely-modedness.

Lemma 5.3 Let $Q = a_1, \dots, a_n$ be a π-nicely moded query and $C = h \leftarrow b_1, \dots, b_m$ be a ρ-nicely moded, input-linear clause where $vars(Q) \cap vars(C) = \emptyset$. Suppose for some $k \in \{1, \dots, n\}$, h and a_k are unifiable. Then the resolvent of Q and C with selected atom a_k is $Der(\pi, \rho, k)$-nicely moded.

Proof. Let θ be the MGU of h and a_k. By Definition 5.2, $a_{\pi^{-1}(1)}, \dots, a_{\pi^{-1}(n)}$ is nicely moded and $h \leftarrow b_{\rho^{-1}(1)}, \dots, b_{\rho^{-1}(m)}$ is nicely moded and input-linear. Thus by Lemma 5.2,

$$(a_{\pi^{-1}(1)}, \dots, a_{\pi^{-1}(\pi(k)-1)}, b_{\rho^{-1}(1)}, \dots, b_{\rho^{-1}(m)}, a_{\pi^{-1}(\pi(k)+1)}, \dots, a_{\pi^{-1}(n)})\,\theta$$

is nicely moded, and so $(a_1, \ldots, a_{k-1}, b_1, \ldots, b_m, a_{k+1}, \ldots, a_n)\,\theta$ is $Der(\pi, \rho, k)$-nicely moded. □

The requirement that the clause must be input-linear can be dropped if the derivation step is input-consuming. It is assumed that the selected atom is sufficiently instantiated, so that a multiple occurrence of the same variable in the input arguments of the clause head cannot cause any bindings to the query.

Lemma 5.4 Let $Q = a_1, \ldots, a_n$ be a π-nicely moded query and $C = p(\mathbf{v}, \mathbf{u}) \leftarrow b_1, \ldots, b_m$ be a ρ-nicely moded clause where $vars(Q) \cap vars(C) = \emptyset$. Suppose for some $k \in \{1, \ldots, n\}$, $p(\mathbf{v}, \mathbf{u})$ and $a_k = p(\mathbf{s}, \mathbf{t})$ are unifiable with MGU θ, and $dom(\theta) \cap vars(\mathbf{s}) = \emptyset$. Then the resolvent of Q and C with selected atom a_k is $Der(\pi, \rho, k)$-nicely moded.

PROOF. Let $C' = p(\mathbf{v}', \mathbf{u}) \leftarrow b_1, \ldots, b_m$ be an input-linear clause such that

1. $vars(\mathbf{v}) \subseteq vars(\mathbf{v}')$ and $vars(\mathbf{v}') \cap vars(Q) = \emptyset$,
2. there exists a substitution σ such that $C'\sigma = C$ and $dom(\sigma) = vars(\mathbf{v}') \setminus vars(\mathbf{v})$.

Intuitively, $\mathbf{v}'$ is obtained from $\mathbf{v}$ by renaming, for each variable occurring several times, all but one occurrences apart using fresh variables.

Since $dom(\theta) \cap vars(\mathbf{s}) = \emptyset$, it follows that $\theta = \theta_1\theta_2$, where θ_1 is an MGU of $\mathbf{v}$ and $\mathbf{s}$, and $\mathbf{v}\theta_1 = \mathbf{s}$, and θ_2 is an MGU of $\mathbf{u}\theta_1$ and $\mathbf{t}\theta_1$.

By (2) and since $\mathbf{v}\theta_1 = \mathbf{s}$, we have $\mathbf{v}'\sigma\theta_1 = \mathbf{s}$. Moreover by (1), (2) and since $dom(\theta_1) \subseteq vars(\mathbf{v})$, we have $dom(\sigma\theta_1) \subseteq vars(\mathbf{v}')$, and hence $\sigma\theta_1$ is an MGU of $\mathbf{v}'$ and $\mathbf{s}$.

By (2), $\mathbf{u}\sigma = \mathbf{u}$ and $\mathbf{t}\sigma = \mathbf{t}$. Therefore θ_2 is an MGU of $\mathbf{u}\sigma\theta_1$ and $\mathbf{t}\sigma\theta_1$.

So we have that $\sigma\theta_1$ is an MGU of $\mathbf{v}'$ and $\mathbf{s}$, and θ_2 is an MGU of $\mathbf{u}\sigma\theta_1$ and $\mathbf{t}\sigma\theta_1$. Therefore $\sigma\theta_1\theta_2 = \sigma\theta$ is an MGU of $p(\mathbf{v}', \mathbf{u})$ and $p(\mathbf{s}, \mathbf{t})$ [Apt97, Lemma 2.24]. Hence by Lemma 5.3 and since C' is input-linear, $(a_1, \ldots, a_{k-1}, b_1, \ldots, b_m, a_{k+1}, \ldots, a_n)\sigma\theta$ is a $Der(\pi, \rho, k)$-nicely moded resolvent of C' and Q. However, by (1) and (2),

$$(a_1, \ldots, a_{k-1}, b_1, \ldots, b_m, a_{k+1}, \ldots, a_n)\theta = (a_1, \ldots, a_{k-1}, b_1, \ldots, b_m, a_{k+1}, \ldots, a_n)\sigma\theta,$$

and so $(a_1, \ldots, a_{k-1}, b_1, \ldots, b_m, a_{k+1}, \ldots, a_n)\theta$ is $Der(\pi, \rho, k)$-nicely moded. □

For a permutation nicely moded program and query, it is guaranteed that every input-consuming derivation step only instantiates other atoms in the query that occur “later” than the selected atom, according to the producer-consumer order.

Lemma 5.5 Make the same assumptions as in Lemma 5.4. Then for all i with $\pi(i) < \pi(k)$, $dom(\theta) \cap vars(a_i) = \emptyset$.

PROOF. Let $a_k = p(\mathbf{s}, \mathbf{t})$. Since the derivation step is input-consuming, $dom(\theta) \cap vars(Q) \subseteq vars(\mathbf{t})$. Thus since Q is π-nicely moded, $dom(\theta) \cap vars(a_i) = \emptyset$ for all i with $\pi(i) < \pi(k)$. □

The above lemma will be used in Chapter 6, where the permutation π is always the identity. For better readability, we restate the lemma for this case.

Lemma 5.6 Let $Q = Q_1, a, Q_2$ be a nicely moded query and $C = h \leftarrow B$ a nicely moded clause where $vars(Q) \cap vars(C) = \emptyset$. Let $\langle Q, \emptyset\rangle; \langle Q_1, B, Q_2,\ \theta\rangle$ be an input-consuming derivation step using C. Then $dom(\theta) \cap vars(Q_1) = \emptyset$.

5.5 Permutation Well Moded Programs

Well-modedness has been introduced by Dembinski and Małuszyński [DM85] and widely used for verification since [AL95, AP94b, EBC99]. When we assume LD-derivations, well-modedness ensures that the input arguments of an atom are ground when the atom is selected. In the programming language Mercury it is even mandatory that programs are well moded[8], which is one of the reasons for its remarkable performance [SHC96].

Definition 5.4 [permutation well moded] Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a query and π a permutation on $\{1, \ldots, n\}$. Then Q is **π-well moded** if for all $i \in \{1, \ldots, n\}$ and $L = 1$

$$vars(\mathbf{s}_i) \subseteq \bigcup_{L \leq \pi(j) < \pi(i)} vars(\mathbf{t}_j) \tag{1}$$

The clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$ is **π-well moded** if (1) holds for all $i \in \{1, \ldots, n+1\}$ and $L = 0$.

A **permutation well moded** query (clause, program) and a **well moded** query (clause, program) **corresponding to** a query (clause, program) are defined in analogy to Definition 5.2. ◁

Note that a one-atom query $p(\mathbf{s}, \mathbf{t})$ is (permutation) well moded if and only if $\mathbf{s}$ is ground.

Example 5.6 Consider the `permute` program (Figure 9 on page 57) It is well moded for mode $\{\texttt{permute}(I, O),\ \texttt{delete}(I, O, I)\}$, and permutation well moded for mode $\{\texttt{permute}(O, I),\ \texttt{delete}(O, I, O)\}$, with the same permutations as Example 5.5. ◁

We quote a persistence result for well-modedness which has been shown previously for LD-resolvents [AP94b] and arbitrary resolvents [AL95].

Lemma 5.7 [AL95, Lemma 16] Let Q be a well moded query and C be a well moded clause where $vars(Q) \cap vars(C) = \emptyset$. Then every resolvent of Q and C is well moded.

We generalise this result to permutation well-modedness.

Lemma 5.8 Let $Q = a_1, \ldots, a_n$ be a π-well moded query and $C = h \leftarrow b_1, \ldots, b_m$ be a ρ-well moded clause where $vars(Q) \cap vars(C) = \emptyset$. Suppose for some $k \in \{1, \ldots, n\}$, h and a_k are unifiable. Then the resolvent of Q and C with selected atom a_k is $Der(\pi, \rho, k)$-well moded.

PROOF. Analogous to Lemma 5.3, but using Lemma 5.7 instead of Lemma 5.2. □

[8] To be precise: *can be made* well moded by reordering of atoms.

5.6 Permutation Well Typed Programs

The disadvantage of (permutation) well-modedness is that it is not possible to reason about programs that operate on non-ground data structures. For example, the query $\mathtt{append([A,B],[C],Zs)}$ is not (permutation) well moded for mode $\mathtt{append}(I, I, O)$ since the input is not ground. Therefore well-modedness has been generalised to *well-typedness* [AL95, AP94b, BLR92].

In a *well typed* query, the first atom is correctly typed in its input positions. Furthermore, given a well typed query Q, a, Q' and assuming LD-derivations, if Q is resolved away, then a becomes correctly typed in its input positions. We generalise this to *permutation well typed.* As with the modes, we assume that the types of all argument positions are given. In the examples, they will be the obvious ones.

Definition 5.5 [permutation well typed] Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a query, where $p_i(\mathbf{S}_i, \mathbf{T}_i)$ is the type of p_i for each $i \in \{1, \ldots, n\}$. Let π be a permutation on $\{1, \ldots, n\}$. Then Q is **π-well typed** if for all $i \in \{1, \ldots, n\}$ and $L = 1$

$$\models (\bigwedge_{L \leq \pi(j) < \pi(i)} \mathbf{t}_j : \mathbf{T}_j) \Rightarrow \mathbf{s}_i : \mathbf{S}_i. \tag{2}$$

The clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$, where $p(\mathbf{T}_0, \mathbf{S}_{n+1})$ is the type of p, is **π-well typed** if (2) holds for all $i \in \{1, \ldots, n+1\}$ and $L = 0$.

A **permutation well typed** query (clause, program) and a **well typed** query (clause, program) **corresponding to** a query (clause, program) are defined in analogy to Definition 5.2. ◁

Note that a one-atom query $p(\mathbf{s}, \mathbf{t})$ is (permutation) well typed if and only if $\mathbf{s}$ is correctly typed.

Example 5.7 Consider the permute program (Figure 9 on page 57) where the type is $\{\mathtt{permute}(\mathit{list}, \mathit{list}), \mathtt{delete}(\mathit{any}, \mathit{list}, \mathit{list})\}$. It is well typed for mode $\{\mathtt{permute}(I, O)$, $\mathtt{delete}(I, O, I)\}$, and permutation well typed for $\{\mathtt{permute}(O, I)$, $\mathtt{delete}(O, I, O)\}$, with the same permutations as Example 5.5. The same holds when we assume type $\{\mathtt{permute}(\mathit{nl}, \mathit{nl})$, $\mathtt{delete}(\mathit{num}, \mathit{nl}, \mathit{nl})\}$. ◁

As before, we quote a persistence property for well-typedness.

Lemma 5.9 [AL95, Lemma 23] Let Q be a well typed query and C be a well typed clause where $\mathit{vars}(Q) \cap \mathit{vars}(C) = \emptyset$. Then every resolvent of Q and C is well typed.

We now generalise this result to permutation well-typedness.

Lemma 5.10 Let $Q = a_1, \ldots, a_n$ be a π-well typed query and $C = h \leftarrow b_1, \ldots, b_m$ be a ρ-well typed clause where $\mathit{vars}(Q) \cap \mathit{vars}(C) = \emptyset$. Suppose for some $k \in \{1, \ldots, n\}$, h and a_k are unifiable. Then the resolvent of Q and C with selected atom a_k is $\mathit{Der}(\pi, \rho, k)$-well typed.

Proof. Analogous to Lemma 5.3, but using Lemma 5.9 instead of Lemma 5.2. □

The following two statements are needed for the proof of Theorem 8.5. The first says that for a π-well typed query Q, every prefix of $\pi(Q)$ is well typed. It follows immediately from Definition 5.5.

Proposition 5.11 Let $Q = a_1, \dots, a_n$ be a π-well typed query. For all $i \in \{1, \dots, n\}$, the subquery of Q containing all a_j such that $\pi(j) \leq \pi(i)$ is permutation well typed.

The second statement says that if all atoms in $\pi(Q)$ before an atom a are resolved away, then a becomes correctly typed in its input positions.

Lemma 5.12 Let P be a permutation well typed program and $Q = a_1, \dots, a_n$ a π-well typed query. For all $j \in \{1, \dots, n\}$, if $Q; \dots; (F, a_j, H)\theta$ is a derivation of $P \cup \{Q\}$ and for all i with $\pi(i) < \pi(j)$, $(F, a_j, H)\theta$ contains no descendants of a_i, then $a_j\theta$ is correctly typed in its input positions.

PROOF. Suppose $(F, a_j, H)\theta$ consists of m atoms and is ρ-well typed, and a_j is the l^{th} atom in F, a_j, H. We show that $\rho(l) = 1$. Thus assume, for the purpose of deriving a contradiction, that there is a $k \in \{1, \dots, m\}$ such that $\rho(k) < \rho(l)$. Then by Lemma 5.1 (b), the k^{th} atom in $(F, a_j, H)\theta$ is either a descendant of a_j, or a descendant of some atom a_i such that $\pi(i) < \pi(j)$. The first case is impossible since a_j has not yet been resolved in $(F, a_j, H)\theta$ and thus the only descendant of a_j is $a_j\theta$. The second case is impossible by the assumption that for all i with $\pi(i) < \pi(j)$, $(F, a_j, H)\theta$ contains no descendants of a_i.

Thus there is no $k \in \{1, \dots, m\}$ such that $\rho(k) < \rho(l)$, and so $\rho(l) = 1$. Therefore it follows by Definition 5.5 that $a_j\theta$ is correctly typed in its input positions. □

It follows from the definitions that permutation well-typedness is a generalisation of permutation well-modedness. In the following proposition, recall that *all_ground* is *the* type containing all ground terms.

Proposition 5.13 Every permutation well moded program is permutation well typed, assuming all argument positions are of type *all_ground*.

Every permutation well typed program, where all argument positions have a ground type, is permutation well moded.

In Chapter 6, our formal results assume (permutation) well typed programs. These results are automatically applicable to all (permutation) well *moded* programs, since these are (permutation) well typed, assuming all argument positions are of type *all_ground*.

5.7 Type-Consistent Programs

Permutation well-typedness is closely linked to the modes of a program: the type correctness of certain output positions implies type correctness of certain input positions. This notion is quite different from the concept of well typed programs as it is used in typed logic programming languages such as Mercury [SHC96] or Gödel [HL94], and also in other contexts, as we have discussed in Section 2.2.

In typed logic programming languages, every argument position in a program has a type. The type-checking of the program allows to guarantee at compile time that no incorrectly typed term can ever occur in an argument position during a derivation for the program. This has been turned into the following slogan [Mil78, MO84]:

> Well-typed programs cannot go wrong.

This is clearly a desirable property, since the occurrence of an incorrectly typed term in an argument position nearly always reveals a programming error [HL94, page 5]. The property is also desirable for verification purposes, as we will see in the next chapter. Unfortunately, our notion of permutation well typed programs does not allow for such a guarantee.

Example 5.8 Consider $\mathtt{append}(I, I, O)$ (Figure 10 on page 57). The query

$$\mathtt{append}([], [], \mathtt{foo}),\ \mathtt{append}(\mathtt{foo}, [], \mathtt{Zs})$$

is well typed since trivially $\models \mathtt{foo} : \mathit{list} \Rightarrow \mathtt{foo} : \mathit{list}$. That is, since the output of the first atom is wrongly typed, we can say that correctly typed output of the first implies correctly typed input for the second atom. We will consider this problem again in Subsection 9.4.1. Boye has given a similar example and has argued that such queries (or programs) are pathological [Boy96]. $\triangleleft$

The question therefore is: given a permutation well typed program and a selection rule $\mathcal{R}$, do all $\mathcal{R}$-derivations for a permutation well typed query consist of queries that can be instantiated so that all arguments are correctly typed? We strongly suspect that this question is undecidable. Nevertheless, we will define classes of programs for which this question can be answered positively. We now give such programs a name.

Definition 5.6 [type-consistent] Let P be permutation well typed program and $\mathcal{R}$ a selection rule.

A query is **type-consistent** if it is permutation well typed and has a correctly typed instance. The program P is **type-consistent** with respect to $\mathcal{R}$ if for all all type-consistent queries Q, all $\mathcal{R}$-derivations of $P \cup \{Q\}$ consist of type-consistent queries. $\triangleleft$

In a slight abuse of terminology, we shall often say that a program is type-consistent with respect to LD-derivations, input-consuming derivations etc.

Obviously every query has a ground instance. This implies that for permutation well moded programs, we can immediately state the following proposition.

Proposition 5.14 Let P be a permutation well moded program, or equivalently (by Proposition 5.13), a permutation well typed program, where the type of all positions is $\mathit{all\_ground}$. Then P is type-consistent with respect to any selection rule.

Chapter 6

Termination of Input-Consuming Derivations

In this chapter, we identify a class of programs for which all input-consuming derivations terminate. To this end, we will make use of the correctness properties defined in Chapter 5.

6.1 Termination and the Selection Rule

Termination of logic programs has been widely studied for LD-derivations [Apt97, AP90, DD94, DVB92, DD93, DD98, EBC99, LS97]. All of these works are based on the following idea: at the time when an atom a in a query is selected, it is possible to *pin down the size* of a. The technical meaning of "pinning down the size" differs among different methods (see Subsection 11.1.1). What is important here is that this size cannot change via further instantiation. It is then shown that for the atoms introduced in this derivation step, it is again possible to pin down their size when eventually they are selected, and that these atoms are smaller than a.

This idea has also been applied to arbitrary derivations [Bez93]. Programs which terminate for arbitrary derivations are called *strongly terminating.* Since no restriction is imposed as to when an atom can be selected, it is required that for each query in a derivation, the size of each of its atoms is always bounded. The class of strongly terminating programs is very small: it contains hardly any "real" non-trivial programs.

For most programs, to ensure termination, it is necessary to require a certain degree of instantiation of an atom before it can be selected. This can be achieved using delay declarations [AL95, Lüt93, MT95, MK97, Nai92, SHK99b, SHK98]. The problem is that, depending on what kinds of delay declarations and selection rules are used, it may not be possible to pin down the size of the selected atom, since this size may depend on the resolution of other atoms in the query that are not yet resolved. Nevertheless, the approaches by Marchiori and Teusink [MT95] and Martin and King [MK97], and to a limited extent Lüttringhaus-Kappel [Lüt93] are based on the idea described above. Others avoid any explicit mention of "size" and instead try to reduce the problem to showing termination for LD-derivations [Nai92].

The approach taken in this chapter falls between the two extremes of making no assumptions about the selection rule on the one hand and making very specific assumptions on the other. We identify predicates for which all input-consuming derivations are finite. Other works in this area have usually made specific assumptions about the selection rule and the delay declarations, for example *local* selection rules [MT95], delay declarations that test arguments for groundness or rigidness [Lüt93, MK97], or the default left-to-right selection rule of most Prolog implementations [Nai92]. In contrast, we show how previous results about LD-derivations can be generalised, the only assumption about the selection rule being that derivations are input-consuming.

We exploit the fact that under certain conditions, it is enough to rely on a *relative* decrease in the size of the selected atom, even though this size cannot be pinned down.

Example 6.1 Consider $\texttt{append}(I, I, O)$ (Figure 10 on page 57) and the following input-consuming derivation. Note that the derivation is the same as in Example 5.4 except for the textual order of the atoms.

$$\begin{array}{l}
\underline{\texttt{append}([1],[],\texttt{As})},\ \texttt{append}(\texttt{As},[],\texttt{Bs}) \rightsquigarrow \\
\texttt{append}([],[],\texttt{As}'),\ \underline{\texttt{append}([1|\texttt{As}'],[],\texttt{Bs})} \rightsquigarrow \\
\underline{\texttt{append}([],[],\texttt{As}')},\ \texttt{append}(\texttt{As}',[],\texttt{Bs}') \rightsquigarrow \\
\underline{\texttt{append}([],[],\texttt{Bs}')} \rightsquigarrow \Box.
\end{array}$$

When $\texttt{append}([1|\texttt{As}'],[],\texttt{Bs})$ is selected, it is not possible to pin down its size in any meaningful way. In fact, nothing can be said about the length of the (input-consuming) derivation associated with $\texttt{append}([1|\texttt{As}'],[],\texttt{Bs})$ without knowing about other atoms that might instantiate $\texttt{As}'$. However, the derivation could be infinite only if the derivation associated with $\texttt{append}([],[],\texttt{As}')$ was infinite. Our method is based on such a dependency between the atoms of a query. ◁

The class of programs for which all input-consuming derivations are finite is obviously larger than the class of strongly terminating programs. Nevertheless, the class is still quite limited. We now give an example of a program which is not in the class.

Example 6.2 For the `permute` program (Figure 9 on page 57) in mode $\{\texttt{permute}(O, I), \texttt{delete}(O, I, O)\}$, we have the following infinite input-consuming derivation:

$$\begin{array}{l}
\underline{\texttt{permute}(\texttt{W},[1])} \rightsquigarrow \\
\texttt{permute}(\texttt{X}',\texttt{Z}'),\ \underline{\texttt{delete}(\texttt{U}',[1],\texttt{Z}')} \rightsquigarrow \\
\underline{\texttt{permute}(\texttt{X}',[1|\texttt{Z}''])},\ \texttt{delete}(\texttt{U}',[],\texttt{Z}'') \rightsquigarrow \\
\texttt{permute}(\texttt{X}'',\texttt{Z}'''),\ \underline{\texttt{delete}(\texttt{U}'',[1|\texttt{Z}''],\texttt{Z}''')},\ \texttt{delete}(\texttt{U}',[],\texttt{Z}'') \rightsquigarrow \\
\underline{\texttt{permute}(\texttt{X}'',[1|\texttt{Z}''''])},\ \texttt{delete}(\texttt{U}'',\texttt{Z}'',\texttt{Z}''''),\ \texttt{delete}(\texttt{U}',[],\texttt{Z}'') \rightsquigarrow \ldots
\end{array}$$

◁

To ensure termination even for programs like the one above, most authors have made stronger assumptions about the selection rule, thereby neglecting the important class for which assuming input-consuming derivations is sufficient. We will show in Chapter 8 that if we can identify predicates in this class, then this information can be embedded into a more comprehensive method for showing termination. We have attempted to formulate our results as generally as possible to make them widely applicable.

In this chapter, we consider derivations where the textual position of an atom within a query is irrelevant for its selection. As we have explained on page 63, we can therefore assume without loss of generality that the textual order of atoms within a query is identical to the producer-consumer order. That is, whenever we use one of the correctness properties introduced in Chapter 5, we can assume that the permutation is the identity and that each predicate has a fixed mode. This simplifies the notation.

This chapter is organised as follows. Section 6.2 explains why the order of clauses in a program is irrelevant for the termination problem we consider. Section 6.3 shows that for well typed and nicely moded programs, it is sufficient to prove termination for one-atom queries. Section 6.4 then shows how one-atom queries can be proven to terminate. In Section 6.5 we sketch how the method presented here could be applied. Section 6.6 discusses the results and some related work.

6.2 Existential vs. Universal Termination

Apart from the selection of an atom in each derivation step, there is also another aspect of control in logic programs: the choice of the *clause* used to resolve the atom. Different choices result in different derivations, some of which could be infinite. In most logic programming systems, the clauses are tried in order of textual occurrence. It is possible for a system first to compute one finite derivation but then on backtracking compute an infinite one, and hence not terminate. This situation is referred to as *existential* termination [DD94], since (at least) one finite derivation is computed. Whether or not a program existentially terminates for a query may depend on the textual order of clauses in the program.

As discussed by De Schreye and Decorte [DD94], most approaches to the termination problem are interested in *universal* termination, that is, finiteness of all derivations. This is also true for this thesis, and therefore, for the termination problems we consider, the clause order in a program is irrelevant. De Schreye and Decorte also remark that proving existential termination is a very hard problem, but nevertheless, it has been addressed by a few authors [Bau92, CT77, FGKP85, Mar96].

6.3 Controlled Coroutining

In this section we define *atom-terminating* predicates. A predicate p is atom-terminating if (under certain conditions) all input-consuming derivations of a query $p(\mathbf{s}, \mathbf{t})$ are finite. Like Etalle et al. [EBC99], we then show that termination for one-atom queries implies termination for arbitrary queries.

For LD-derivations, it is almost obvious that it is sufficient to show termination for one-atom queries, and it only requires that programs and queries are well moded, but not nicely moded [EBC99, Lemma 4.2]. Given an LD-derivation ξ for a query $a_1, \dots, a_n$, the sub-derivations for each a_i do not interleave, and therefore ξ can be regarded as a derivation for a_1 followed by a derivation for a_2 and so forth. The following example illustrates that in the context of interleaving sub-derivations (coroutining), this is not at all obvious.

Example 6.3 Consider $\texttt{append}(I, I, O)$ (Figure 10 on page 57) and the query

$$\texttt{append([],[],As),append([1|As],[],Bs),append(Bs,[],As)}.$$

This query is well moded but not nicely moded. Then we have the following infinite input-consuming derivation:

$$\begin{array}{l}\texttt{append([],[],As)},\ \underline{\texttt{append([1|As],[],Bs)}},\ \texttt{append(Bs,[],As)} \rightsquigarrow \\ \texttt{append([],[],As)},\ \texttt{append(As,[],Bs')},\ \underline{\texttt{append([1|Bs'],[],As)}} \rightsquigarrow \\ \texttt{append([],[],[1|As'])},\ \underline{\texttt{append([1|As'],[],Bs')}},\ \texttt{append(Bs',[],As')} \rightsquigarrow \dots\end{array}$$

This well-known termination problem of programs with coroutining has been identified as *circular modes* [Nai92]. ◁

To avoid the problem, we require programs and queries to be nicely moded. We do not require programs to be well moded. However, we require them to be well typed and type-consistent with respect to input-consuming derivations. By Proposition 5.14, well moded programs are one class of programs meeting this requirement.

Recall that a one-atom query $p(\mathbf{s}, \mathbf{t})$ is well typed and nicely moded if and only if $\mathbf{s}$ is correctly typed, $vars(\mathbf{s}) \cap vars(\mathbf{t}) = \emptyset$ and $\mathbf{t}$ is linear.

Definition 6.1 [atom-terminating predicate/atom] Let P be a well typed and nicely moded program which is type-consistent with respect to input-consuming derivations. A predicate p in P is **atom-terminating** if for each well typed, type-consistent and nicely moded query $p(\mathbf{s}, \mathbf{t})$, all input-consuming derivations of $P \cup \{p(\mathbf{s}, \mathbf{t})\}$ are finite. An atom is **atom-terminating** if its predicate is atom-terminating. ◁

We need the following simple auxiliary lemma to prove Lemma 6.2.

Lemma 6.1 Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a well typed, type-consistent and nicely moded query. Then there exists a substitution σ such that $dom(\sigma) = vars(\mathbf{t}_1, \dots, \mathbf{t}_{n-1})$, and $p_n(\mathbf{s}_n, \mathbf{t}_n)\sigma$ is well typed, type-consistent and nicely moded.

Proof. Since Q is type-consistent and types are closed under instantiation, there exists a (minimal) substitution σ such that $dom(\sigma) = vars(\mathbf{t}_1, \dots, \mathbf{t}_{n-1})$ and $(\mathbf{t}_1, \dots, \mathbf{t}_{n-1})\sigma$ is ground and correctly typed. Note that $vars(ran(\sigma)) = \emptyset$.

By Definition 5.5, $p_n(\mathbf{s}_n, \mathbf{t}_n)\sigma$ is well typed. Since Q is nicely moded, it follows that $dom(\sigma) \cap vars(\mathbf{t}_n) = \emptyset$ and hence $p_n(\mathbf{s}_n, \mathbf{t}_n)\sigma$ is type-consistent. Moreover, $vars(\mathbf{s}_n) \cap vars(\mathbf{t}_n) = \emptyset$ and $vars(ran(\sigma)) = \emptyset$, and hence $vars(\mathbf{s}_n\sigma) \cap vars(\mathbf{t}_n\sigma) = \emptyset$. Therefore by Definition 5.2, $p_n(\mathbf{s}_n, \mathbf{t}_n)\sigma$ is nicely moded. □

The following lemma says that an atom-terminating atom cannot proceed indefinitely unless it is repeatedly fed by some other atom.

Lemma 6.2 Let P be a well typed and nicely moded program which is type-consistent with respect to input-consuming derivations. Let F, b, H be a well typed, type-consistent and nicely moded query where b is an atom-terminating atom. An input-consuming derivation of $P \cup \{F, b, H\}$ can have infinitely many b-steps only if it has infinitely many a-steps, for some $a \in F$.

Proof. In this proof, by an F-**step** we mean an a-step, for some $a \in F$; likewise we define an H-**step**. By Lemma 5.6, no H-step can instantiate any descendant of b. Thus the H-steps can be disregarded, and without loss of generality, we assume that H is empty. Suppose ξ is an input-consuming derivation for $P \cup \{F, b\}$ containing finitely many F-steps. We can write

$$\xi = \langle F,b,\ \emptyset\rangle; \ldots ; \langle Q_0, \theta_0\rangle; \tilde{\xi}$$

where $\langle Q_0, \theta_0\rangle; \tilde{\xi}$ contains no F-steps. Since by Lemma 5.6, no b-step can instantiate any descendant of F, there exists an input-consuming derivation

$$\xi_2 = \langle F,b,\ \emptyset\rangle; \ldots ; \langle R, \rho\rangle; \ldots ; \langle Q_0, \theta_0\rangle; \tilde{\xi}$$

such that $\langle F,b,\ \emptyset\rangle; \ldots ; \langle R, \rho\rangle$ contains only F-steps and $\langle R, \rho\rangle; \ldots ; \langle Q_0, \theta_0\rangle; \tilde{\xi}$ contains only b-steps (that is, the F-steps are moved forward using the Switching Lemma [Llo87, Lemma 9.1]). Since $R = R', b$ for some R', there exists an input-consuming derivation

$$\xi_3 = \langle b, \rho\rangle; \ldots ; \langle I_0, \theta_0\rangle; \tilde{\xi}_3$$

obtained from $\langle R, \rho\rangle; \ldots ; \langle Q_0, \theta_0\rangle; \tilde{\xi}$ by removing the prefix R' in each query.

By Lemmas 5.10 and 5.4, $R\rho$ is well typed and nicely moded, and since P is type-consistent with respect to input-consuming derivations, $R\rho$ is type-consistent. Thus by Lemma 6.1, there is a substitution σ such that $b\rho\sigma$ is well typed, type-consistent and nicely moded. Moreover $dom(\sigma) = V$, where V is the set of variables in the output positions of $R'\rho$.

By Lemma 5.6, no b-step in ξ_2, and hence no derivation step in ξ_3, can instantiate a variable in V. Since $dom(\sigma) = V$, it thus follows that we can construct an input-consuming derivation

$$\xi_4 = \langle b, \rho\sigma\rangle; \ldots ; \langle I_0, \theta_0\sigma\rangle; \tilde{\xi}_3\sigma$$

by applying σ to each query in ξ_3.

Since $b\rho\sigma$ is a well typed, type-consistent and nicely moded query and b is atom-terminating, ξ_4 is finite. Therefore ξ_3, ξ_2, and finally ξ are finite. □

The following theorem is a consequence and states that atom-terminating atoms on their own cannot produce an infinite derivation.

Theorem 6.3 Let P be a well typed and nicely moded program which is type-consistent with respect to input-consuming derivations, and Q a well typed, type-consistent and nicely moded query. An input-consuming derivation of $P \cup \{Q\}$ can be infinite only if there are infinitely many steps where an atom is resolved that is not atom-terminating.

Proof. We first show:

(∗) For any well typed, type-consistent and nicely moded query Q', an input-consuming derivation of $P \cup \{Q'\}$ can be infinite only if it contains *at least one* step where an atom is resolved that is not atom-terminating.

So let ξ' be an infinite input-consuming derivation of $P \cup \{Q'\}$. Then it follows by Lemma 6.2 that ξ' contains infinitely many a-steps, for some $a \in Q'$ that is not atom-terminating. Hence the first a-step in ξ' is a step where an atom is resolved that is not atom-terminating. This implies (∗).

Now let ξ be an infinite input-consuming derivation of $P \cup \{Q\}$. Assume, for the purpose of deriving a contradiction, that ξ contains only finitely many steps where an atom is resolved that is not atom-terminating. Let $\tilde{\xi}$ be a suffix of ξ containing no steps where an atom is resolved that is not atom-terminating. By Lemmas 5.10 and 5.4, the first query of $\tilde{\xi}$ is well typed and nicely moded. Moreover, $\tilde{\xi}$ is infinite, and so we have a contradiction to (∗). Thus it follows that ξ contains infinitely many steps where an atom is resolved that is not atom-terminating, which completes the proof. □

Theorem 6.3 provides us with the formal justification for restricting our attention to one-atom queries.

6.4 Showing that a Predicate is Atom-Terminating

All approaches to termination mentioned earlier more or less explicitly rely on measuring the size of the *input* in a query [Apt97, AP90, DD94, DVB92, DD93, DD98, EBC99, LS97]. We agree with Etalle et al. [EBC99] that it is reasonable to make this dependency explicit. This gives rise to the notion of *moded level mapping*, which is an instance of *level mapping* introduced by Bezem [Bez93] and Cavedon [Cav89]. Since we use well typed programs instead of well moded ones, we have to generalise the concept further.

In the following definition, $\mathbf{B}_P$ denotes the set of ground atoms using predicates occurring in P.

Definition 6.2 [moded typed level mapping] Let P be a program. The function $|.|$ is a **moded typed level mapping** if

1. it is a level mapping, that is a function $|.| : \mathbf{B}_P \rightarrow \mathbb{N}$,

2. for any ground $\mathbf{s}$, $\mathbf{t}$ and $\mathbf{u}$, $|p(\mathbf{s}, \mathbf{t})| = |p(\mathbf{s}, \mathbf{u})|$.

3. if $p(\mathbf{s}, \mathbf{t})$ is correctly typed in its input positions, then $|p(\mathbf{s}, \mathbf{t})\theta_1| = |p(\mathbf{s}, \mathbf{t})\theta_2|$ for all substitutions θ_i such that $p(\mathbf{s}, \mathbf{t})\theta_i$ is ground $(i = 1, 2)$.

For $a \in \mathbf{B}_P$, $|a|$ is the **level** of a. ◁

Thus the level of an atom only depends on the terms in the input positions. Moreover, the level of an atom is fixed once its input arguments are correctly typed; this is where our concept differs from moded level mappings. As Proposition 5.13 shows, the concepts coincide if the only type is *all_ground*, that is, if we only consider well moded programs.

The following concept, adopted from Apt [Apt97], is useful for proving termination for a whole program incrementally, by proving it for one predicate at a time.

Definition 6.3 [depends on] Let p, q be predicates in a program P. We say that p **refers to** q if there is a clause in P with p in its head and q in its body, and p **depends on** q (written $p \sqsupseteq q$) if (p, q) is in the reflexive, transitive closure of *refers to*. We write $p \sqsupset q$ if $p \sqsupseteq q$ and $q \not\sqsupseteq p$, and $p \approx q$ if $p \sqsupseteq q$ and $q \sqsupseteq p$. ◁

Abusing notation, we shall also use the above symbols for *atoms*, where $p(\mathbf{s}, \mathbf{t}) \sqsupseteq q(\mathbf{u}, \mathbf{v})$ stands for $p \sqsupseteq q$, and likewise for $\sqsupset$ and $\approx$. Furthermore, we denote the equivalence class of a predicate p with respect to $\approx$ as $[p]_{\approx}$.

The following definition provides us with a criterion for proving that a predicate is atom-terminating.

Definition 6.4 [ICD-acceptable] Let P be a program and $|.|$ a moded typed level mapping. A clause $C = h \leftarrow B$ is **acceptable for input-consuming derivations (with respect to** $|.|$**)** if for every substitution θ such that $C\theta$ is ground, and for every a in B such that $a \approx h$, we have $|h\theta| > |a\theta|$. We abbreviate *acceptable for input-consuming derivations* by **ICD-acceptable**.

A program (set of clauses) is **ICD-acceptable with respect to** $|.|$ if each clause is ICD-acceptable with respect to $|.|$. ◁

Let us compare this concept to some similar concepts in the literature: *recurrent* [Bez93], *well-acceptable* [EBC99] and *acceptable* [AP94a, DD98] programs.

Like Decorte and De Schreye [DD98] and Etalle et al. [EBC99] but unlike Apt and Pedreschi [AP94a] and Bezem [Bez93], we require $|h\theta| > |a\theta|$ only for atoms a where $a \approx h$. This is consistent with the idea that termination should be proven incrementally: to show termination for a predicate p, it is assumed that all predicates q with $p \sqsupset q$ have already been shown to terminate. Therefore we can restrict our attention to the predicates q where $q \approx p$.

Like Bezem but unlike Apt and Pedreschi, Decorte and De Schreye and Etalle et al., our definition does not involve models or computed answer substitutions. Traditionally, the definition of acceptable programs is based on a model M of the program, and for a clause $h \leftarrow a_1, \ldots, a_n$, $|h\theta| > |a_i\theta|$ is only required if $M \models (a_1, \ldots, a_{i-1})\theta$. The reason is that for LD-derivations, $a_1, \ldots, a_{i-1}$ must be completely resolved before a_i is selected. By the correctness of LD-resolution [Llo87] and well-modedness, the accumulated answer substitution θ, just before a_i is selected, is such that $(a_1, \ldots, a_{i-1})\theta$ is ground and $M \models (a_1, \ldots, a_{i-1})\theta$.

Such considerations count for little when derivations are merely required to be input-consuming. This is illustrated in Example 6.2. In the third line of the derivation, $\texttt{permute}(\texttt{X}', [\texttt{1}|\texttt{Z}''])$ is selected, although there is no instance of $\texttt{delete}(\texttt{U}', [], \texttt{Z}'')$ in the model of the program. This problem has been described by saying that `delete` makes a *speculative output binding* [Nai92]. Programs that do not make speculative output bindings are considered in Subsection 8.3.2.

Theorem 6.4 Let P be a well typed and nicely moded program which is type-consistent with respect to input-consuming derivations, and let p be a predicate in P. Suppose all predicates q with $p \sqsupset q$ are atom-terminating, and all clauses defining predicates $q \in [p]_{\approx}$ are ICD-acceptable. Then p, and hence every predicate in $[p]_{\approx}$, is atom-terminating.

PROOF. Suppose the set of clauses defining the predicates $q \in [p]_{\approx}$ is ICD-acceptable with respect to the moded typed level mapping $|.|$. For an atom a using a predicate in $[p]_{\approx}$, we define $||a|| = sup(\{|a\theta| \mid a\theta \text{ is ground}\})$, if the set $\{|a\theta| \mid a\theta \text{ is ground}\}$ is bounded. Otherwise $||a||$ is undefined. Observe that

$$\text{if } ||a|| \text{ is defined for an atom } a, \text{ then } ||a\theta|| \le ||a|| \text{ for all } \theta. \qquad (*)$$

To measure the size of a query, we use the multiset containing the level of each atom whose predicate is in $[p]_{\approx}$. The multiset is formalised as a function *Size*, which takes as arguments a query and a natural number:

$$Size(Q)(n) = \#\{q(\mathbf{u}, \mathbf{v}) \mid q(\mathbf{u}, \mathbf{v}) \in Q,\ q \approx p \text{ and } ||q(\mathbf{u}, \mathbf{v})|| = n\}.$$

Note that if a query contains several identical atoms, each occurrence must be counted. We define $Size(Q) < Size(R)$ if and only if there is a number l such that $Size(Q)(l) < Size(R)(l)$ and $Size(Q)(l') = Size(R)(l')$ for all $l' > l$. Intuitively, a decrease with respect to $<$ is obtained when an atom in a query is replaced with a finite number of smaller atoms. By König's Lemma [Fit96] or Dershowitz [Der87], all descending chains with respect to $<$ are finite.

Let $Q_0 = p(\mathbf{s}, \mathbf{t})$ be a well typed, type-consistent and nicely moded query. Then $\mathbf{s}$ is correctly typed and thus $||Q_0||$ is defined. Let $\xi = Q_0; Q_1; Q_2 \ldots$ be an input-consuming derivation of $P \cup \{Q_0\}$.

Since all predicates q with $p \sqsupset q$ are atom-terminating, it follows by Theorem 6.3 that there cannot be an infinite suffix of ξ without any steps where an atom $q(\mathbf{u}, \mathbf{v})$ such that $q \approx p$ is resolved. We show that for all $i \ge 0$, if the selected atom in $Q_i; Q_{i+1}$ is $q(\mathbf{u}, \mathbf{v})$ and $q \approx p$, then $Size(Q_{i+1}) < Size(Q_i)$, and otherwise $Size(Q_{i+1}) \le Size(Q_i)$. This implies that ξ is finite, and, as the choice of the initial query $Q_0 = p(\mathbf{s}, \mathbf{t})$ was arbitrary, p is atom-terminating.

Consider $i \ge 0$ and let $C = q(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \ldots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ be the clause, $q(\mathbf{u}, \mathbf{v})$ the selected atom and θ the MGU used in $Q_i; Q_{i+1}$.

If $p \sqsupset q$, then $p \sqsupset q_j$ for all $j \in \{1, \ldots, m\}$ and hence by $(*)$ it follows that $Size(Q_{i+1}) \le Size(Q_i)$.

Now consider $q \approx p$. Since C is ICD-acceptable, it follows that $||q(\mathbf{v}_0, \mathbf{u}_{m+1})\theta|| > ||q_j(\mathbf{u}_j, \mathbf{v}_j)\theta||$ for all j with $q_j \approx p$. This together with $(*)$ implies $Size(Q_{i+1}) < Size(Q_i)$. □

Example 6.4 We now give a few examples of atom-terminating predicates. For all predicates, we assume that all argument positions have type *all_ground*. We denote the *term size* of a term t, that is the number of function and constant symbols that occur in t, as $TSize(t)$.

The program for $\texttt{append}(I, I, O)$ (Figure 10 on page 57) is ICD-acceptable, where $|\texttt{append}(s_1, s_2, t)| = TSize(s_1)$. Thus $\texttt{append}(I, I, O)$ is atom-terminating. The same holds for $\texttt{append}(O, O, I)$, defining $|\texttt{append}(t_1, t_2, s)| = TSize(s)$.

```
nqueens(N,Sol) :-               safe_aux([],_,_).
  sequence(N,Seq),              safe_aux([M|Ms],Dist,N) :-
  permute(Sol,Seq),               no_diag(N,M,Dist),
  safe(Sol).                      Dist2 is Dist+1,
                                  safe_aux(Ms,Dist2,N).
safe([]).
safe([N|Ns]) :-                 no_diag(N,M,Dist) :-
  safe_aux(Ns,1,N),               Dist =\= N-M,
  safe(Ns).                       Dist =\= M-N.
```

Figure 12: Fragment of a program for n-queens

The clauses defining $\texttt{delete}(O,I,O)$ (Figure 9 on page 57) are ICD-acceptable, where $|\texttt{delete}(t_1,s,t_2)| = TSize(s)$. Thus $\texttt{delete}(O,I,O)$ is atom-terminating. The same holds for $\texttt{delete}(I,O,I)$, defining $|\texttt{delete}(s_1,t,s_2)| = TSize(s_2)$.

In a similar way, we can show that $\texttt{permute}(I,O)$ is atom-terminating. However, $\texttt{permute}(O,I)$ is not atom-terminating, as seen in Example 6.2.

The book on the Gödel language [HL94, page 81] shows a program that contains a clause, which in Prolog would be written as

```
slowsort(X,Y) :-
  permute(Y,X),
  sorted(Y).
```

The mode is $\{\texttt{slowsort}(I,O), \texttt{permute}(O,I), \texttt{sorted}(I)\}$, and there are delay declarations to ensure that derivations are input-consuming. The predicate `slowsort` is *not* atom-terminating. However it can easily be made atom-terminating by replacing `permute(Y,X)` with `permute(X,Y)`, so that `permute` is used in the mode in which it is atom-terminating.[1]

Note that according to the Gödel specification, no guarantees are given about the selection rule that go beyond ensuring that derivations for the above program are input-consuming. Hence the program is not guaranteed to terminate even for a "well-behaved" query such as $\texttt{slowsort}([1,2],\texttt{Y})$. Even though Hill and Lloyd do not claim that the program terminates, one would still expect it to do so. In contrast, we can modify the program as stated above, and guarantee that the modified program terminates.

Figure 12 shows a fragment from a program for the n-queens problem. The mode is $\{\texttt{nqueens}(I,O)$, $\texttt{sequence}(I,O)$, $\texttt{safe}(I)$, $\texttt{permute}(O,I)$, $\texttt{is}(O,I)$, $\texttt{safe\_aux}(I,I,I)$, $\texttt{no\_diag}(I,I,I)$, $\texttt{=\textbackslash=}(I,I)\}$. Again using as level mapping the term size of one of the arguments, one can see that the clauses defining $\{\texttt{no\_diag}, \texttt{safe\_aux}, \texttt{safe}\}$ are ICD-acceptable and thus these predicates are atom-terminating. Note that for efficiency reasons, this program relies on input-consuming derivations where atoms using `safe` are selected as early as possible. This will be discussed in Chapter 8.

[1] This example had to be adapted because the argument order in the definition of `permute` given in the Gödel book is the reverse of the order in Figure 9. It is the case though that `slowsort`, as given in the Gödel book, is not atom-terminating.

```
plus_one(X) :-
  minus_two(succ(X)).

minus_two(succ(X)) :-
  minus_one(X).
minus_two(0).

minus_one(succ(X)) :-
  plus_one(X).
minus_one(0).
```

Figure 13: An example requiring a complex level mapping

As a more complex example, consider the program in Figure 13, whose mode is $\{\texttt{plus\_one}(I), \texttt{minus\_two}(I), \texttt{minus\_one}(I)\}$. Defining

$$\begin{aligned} |\texttt{plus\_one}(s)| &= 3 * TSize(s) + 4 \\ |\texttt{minus\_two}(s)| &= 3 * TSize(s) \\ |\texttt{minus\_one}(s)| &= 3 * TSize(s) + 2, \end{aligned}$$

the clauses are ICD-acceptable and thus the predicates are atom-terminating. ◁

We see from these examples that whenever in some argument position of a clause head, there is a compound term of some recursive data structure, such as [X|Xs], and all recursive calls in the body of the clause have a strict subterm of that term, such as Xs, in the same position — then the clause is ICD-acceptable using as level mapping the term size of that argument position. Since this situation occurs very often, it can be expected that an average program contains many atom-terminating predicates. However, it is unlikely that in any real program, *all* predicates are atom-terminating.

The example in Figure 13 shows that more complex scenarios than the one described above are possible, but we doubt that they would often occur in practice. Therefore level mappings such as the one used for this program will rarely be needed.

Consider again Definition 6.4. Given a clause $h \leftarrow a_1, \ldots, a_n$ and an atom $a_i \approx h$, we require $|h\theta| > |a_i\theta|$ for all grounding substitutions θ, rather than only for θ such that $(a_1, \ldots, a_{i-1})\theta$ is in a certain model of the program. This is of course a severe restriction. For example, if we consider $\texttt{permute}(O, I)$ (Figure 9 on page 57), there cannot be a moded typed level mapping such that $|\texttt{permute}([\texttt{U}|\texttt{X}], \texttt{Y})\theta| > |\texttt{permute}(\texttt{X}, \texttt{Z})\theta|$ for all θ. That however is not surprising since $\texttt{permute}(O, I)$ is not atom-terminating.

With a similar argument, we can show that there cannot be a moded typed level mapping such that the usual recursive clause for $\texttt{quicksort}(I, O)$ (a modified version of it is shown in Figure 14 on page 87) is ICD-acceptable, although we conjecture that $\texttt{quicksort}(I, O)$ is atom-terminating. This shows a limitation of the method presented here. It might be possible to relax Definition 6.4 to allow more programs, but the fact remains that many predicates are not atom-terminating.

Our method of showing that a predicate p is atom-terminating is based on assuming that all predicates q with $p \sqsupset q$ have already been shown to be atom-terminating. Thus if *p can be shown to be atom-terminating using Theorem 6.4*, then all predicates q with $p \sqsupset q$ are atom-terminating. This does not mean that if p *is* atom-terminating, then all predicates q with $p \sqsupset q$ are atom-terminating. This is demonstrated in the following example.

Example 6.5 Consider the following program with mode $\{\mathtt{p}(I), \mathtt{q}(I)\}$ and type $\{\mathtt{p}(int), \mathtt{q}(int)\}$.

```
p(0) :- q(0).                   q(0).
                                q(1) :- q(1).
```

The predicate `p` is atom-terminating, but our method fails to show this, since `q` is not atom-terminating. Of course this program is contrived, and we do not expect this problem to occur in "real" programs. $\triangleleft$

6.5 Applying the Method

The requirement of input-consuming derivations merely reflects the very meaning of *input*: an atom must only consume its own input, not produce it. Thus if one accepts that (appropriately chosen) modes are useful for verification and reflect the programmer's intentions, then one should also accept this requirement and regard any violation of it as pathological.[2]

The requirement of input-consuming derivations is trivially met for LD-derivations of a well moded query and program, since the leftmost atom in a well moded query is ground in its input positions. It can also be ensured by using delay declarations as in Gödel [HL94] that require the input arguments of an atom to be ground before this atom can be selected. In the next chapter we shall see how input-consuming derivations can be ensured using `block` declarations.

As we have said in the introduction of this chapter, the class of programs for which all input-consuming derivations terminate is quite limited. For the predicates that are not atom-terminating, stronger assumptions about the selection rule are necessary. In Chapter 8, we show one way of incorporating the method of this chapter into a more comprehensive method for proving termination. We now briefly sketch two other ways.

First, we could build on a technique developed by Martin and King [MK97]. They consider coroutining derivations, but impose a bound on the depth of each sub-derivation by introducing auxiliary predicates with an additional argument that serves as depth counter. Applying the results of this chapter, we only have to impose this depth bound for the predicates that are not atom-terminating. For the atom-terminating predicates, we can save the overheads involved in this technique.

Secondly, we could use delay declarations as they are provided for example in Gödel [HL94]. For the atom-terminating predicates, it is sufficient to ensure input-consuming derivations, by checking for partial instantiation of the input positions using a `DELAY`...`UNTIL NONVAR`... declaration. For the other predicates, it must be ensured that the input positions are ground using a `DELAY`...`UNTIL GROUND`... declaration. Note

[2] An exception, concerning programs we cannot verify using the methods of this thesis, is discussed in Subsection 11.1.9.

that according to its specification, Gödel does not guarantee a (default) left-to-right selection rule, and therefore delay declarations are crucial for termination. Note also that a groundness test is usually more expensive than a test for partial instantiation. To the best of our knowledge, there has never been a systematic treatment of the question of when `GROUND` declarations are needed, and when `NONVAR` declarations are sufficient.

6.6 Discussion

We have identified the class of programs for which all input-consuming derivations are finite. Predicates can be shown to be in that class using the notions of *level mapping* and *acceptable clause* in a very similar way to methods for LD-derivations [DD94, DD98, EBC99].

We have considered input-consuming derivations rather than, say, a particular kind of delay construct. This abstract view should make it possible to incorporate the results of this chapter into various more comprehensive methods for proving termination. One advantage is that in this chapter, we do not impose the restriction that programs must be input-linear. This restriction, as we will see in the next chapter, is necessary so that `block` declarations can ensure input-consuming derivations. Hence if input-consuming derivations can be ensured without imposing this restriction, say by using guards as in (F)GHC [Ued86], then the results of this chapter could be applied to show termination.

Note also that the method presented in this chapter can be used to show termination of parallel executions [CC94, Tic91]. In formalisations of parallel executions, one important question is which atoms should be allowed to be selected in parallel. This question has several aspects, one of which is termination. Concerning this aspect, we can state that performing input-consuming derivation steps in parallel, rather than consecutively, does not affect the termination behaviour of a program.

This chapter closely follows Etalle et al. [EBC99]. They have a statement analogous to Theorem 6.4, but they also show a converse statement. It says that if for a predicate p, all LD-derivations for a well moded query $p(\mathbf{s}, \mathbf{t})$ terminate, then there is a level mapping such that the clauses defining p are well-acceptable. It would be interesting to show a similar result for arbitrary input-consuming derivations, but presumably this must be difficult, since our definition of acceptability is much more restrictive.

Unlike most other approaches to termination [AP94a, Bez93, DVB92, DD98, EBC99, LS97, MK97], we do not rely on the idea that the size of an atom can be pinned down when the atom is selected. We show that under certain conditions, it is enough to rely on a *relative* decrease in the size of the selected atom, even though this size cannot be pinned down. More precisely, we exploit the fact that an atom in a query cannot proceed indefinitely unless it is repeatedly fed by some other atom occurring earlier in the query. This implies that every derivation for the query is finite.

Chapter 7

Ensuring Input-Consuming Derivations

In this chapter, we show how `block` declarations can be used to ensure that derivations are input-consuming. To this end, we must introduce further correctness properties in the style of the properties introduced in Chapter 5.

7.1 The Simplicity of `block` Declarations

The `block` declarations declare that certain arguments of an atom must be *non-variable* before that atom can be selected. Insufficiently instantiated atoms are delayed. As demonstrated in SICStus Prolog [SIC98], `block` declarations can be efficiently implemented: the test whether arguments are non-variable has a negligible impact on performance. Therefore `block` declarations or similar constructs are widely used.

It is a distinctive feature of this work that we consider `block` declarations, as opposed to delay declarations which can check for the instantiation of a *subterm* of an argument [HL94], or delay declarations that check for groundness.

One would expect that `block` declarations are sufficiently powerful to ensure that derivations are input-consuming. Consider the clause head `append([X|Xs],Ys,[X|Zs])` (Figure 10 on page 57) and assume that the mode is `append`(I, I, O). If we want to resolve an atom `append`(s, t, u) in a query, then we should check first whether s is non-variable, because otherwise the derivation step would not be input-consuming. However, we will see that the technical details are quite subtle.

As mentioned on page 58, we believe that the most important purpose of delay declarations is to ensure input-consuming derivations. Most works about delay declarations do not explicitly state what their purpose is [AL95, Lüt93, MT95, MK97, Nai92]. Moreover, at least Lüttringhaus-Kappel [Lüt93] considers delay declarations that are used for a purpose that goes far beyond ensuring that derivations are input-consuming. Namely, they are used to ensure that an atom is only selected when it is bounded with respect to some norm (this is done to ensure termination).

This chapter is organised as follows. The next section introduces some terminology related to `block` declarations. Section 7.3 introduces permutation simply typed programs, which are a class of programs for which `block` declarations can ensure input-consuming derivations. Section 7.4 introduces permutation robustly typed programs, which are an extension of the previous class for which `block` declarations can still ensure input-consuming derivations. Section 7.5 gives a summary and comparison of all the correctness properties for programs introduced in this thesis.

7.2 Terminology Related to `block` Declarations

A `block` declaration [SIC98] for a predicate p/n is a (possibly empty) set of atoms each of which has the form $p(b_1, \dots, b_n)$, where $b_i \in \{\texttt{?}, \texttt{-}\}$ for $i \in \{1, \dots, n\}$. A **program** consists of a set of clauses and a set of `block` declarations, one for each predicate defined by the clauses. If P is a program, then an atom $p(t_1, \dots, t_n)$ is **selectable in** P if for each atom $p(b_1, \dots, b_n)$ in the `block` declaration for p, there is some $i \in \{1, \dots, n\}$ such that t_i is non-variable and $b_i = \texttt{-}$.

A **delay-respecting derivation** for a program P is a derivation where the selected atom is always selectable in P. We say that it **flounders** if it ends with a non-empty query where no atom is selectable.

7.3 Permutation Simply Typed Programs

To ensure that derivations are input-consuming, one would expect that there should be `block` declarations such that an atom can only be selected when its input arguments are non-variable. The following example however shows that this is not sufficient.

Example 7.1 Consider the following version of `delete`(O, I, O).

```
:- block delete(?,-,-).
delete(X,[X|Z],Z).
delete(X,[U|[H|T]],[U|Z]) :- delete(X,[H|T],Z).
```

Then we have the following delay-respecting but not input-consuming derivation

$$\underline{\texttt{delete(A,[1|L],R)}} \leadsto \underline{\texttt{delete(A,[H'|L'],R')}} \leadsto \underline{\texttt{delete(A,[H''|L''],R'')}} \leadsto \dots$$

Note that although `delete(A,[1|L],R)` is not a well typed query, it may *occur* in a well typed query, say `delete(B,[2],L), delete(A,[1|L],R)`. This version of `delete` is part of the *most specific program* [MNL90] corresponding to the program in Figure 9 on page 57, proposed [Nai92] to prevent looping for `permute`(O, I). However, it does not work. The query `permute(A,[1])` indeed terminates, but `permute(A,[1,2])` still loops. ◁

Thus to ensure that derivations are input-consuming, we will require that each input argument in each clause head is *flat*. This condition is violated by the clause head `delete(X,[U|[H|T]],[U|Z])`, but it is met for the program in Figure 9 on page 57.

The next example shows however that requiring flat terms in clause heads is still not enough.

Example 7.2 Consider the following program in mode p(I, O).

```
:- block p(-,?).
p(g(Y),Y).
```

Then p(g(X), 3) $\leadsto \Box$ is a delay-respecting but not input-consuming derivation, since X becomes instantiated to 3. ◁

The easiest solution is to require that the output positions in a query are always filled with variables. In mode p(I, O), the query p(g(X),3) should not arise, since its output is already instantiated. We will now present this solution, although it has certain limitations. In Section 7.4, we will see how these limitations can partly be overcome.

We first define *permutation simply-modedness*, which is a generalisation of simply-modedness [AE93, AL95], just as for the other correctness properties. In a permutation simply moded query, the output positions are filled with variables.

Definition 7.1 [permutation simply moded] Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a query and π a permutation on $\{1, \dots, n\}$. Then Q is **π-simply moded** if it is π-nicely moded and $\mathbf{t}_1, \dots, \mathbf{t}_n$ is a vector of *variables.* The clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$ is **π-simply moded** if it is π-nicely moded and $\mathbf{t}_1, \dots, \mathbf{t}_n$ is a vector of *variables.*

A **permutation simply moded** query (clause, program) and a **simply moded** query (clause, program) **corresponding to** a query (clause, program) are defined in analogy to Definition 5.2. ◁

We quote the following persistence property for simply-modedness.

Lemma 7.1 [AE93, Lemma 27] Let Q be a simply moded query and C a simply moded clause where $vars(Q) \cap vars(C) = \emptyset$. Then every LD-resolvent of Q and C is simply moded.

We combine permutation simply-modedness with permutation well-typedness, adding an extra condition concerning the clause heads.

Definition 7.2 [permutation simply typed] A query is **π-simply typed** if it is π-simply moded and π-well typed. A clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ is **π-simply typed** if it is π-simply moded and π-well typed, and $\mathbf{t}_0$ has a variable in each position of variable type and a flat type-consistent term in each position of non-variable type.

A **permutation simply typed** query (clause, program) and a **simply typed** query (clause, program) **corresponding to** a query (clause, program) are defined in analogy to Definition 5.2. ◁

Note that since the vector of output arguments of a permutation simply typed query is a linear vector of variables, permutation simply typed queries are type-consistent.

Example 7.3 The permute program (Figure 9 on page 57), for any of the types in Example 5.7, is simply typed for mode $\{$permute(I, O), delete$(I, O, I)\}$, and permutation simply typed for mode $\{$permute(O, I), delete$(O, I, O)\}$. ◁

```
:- block quicksort(-,-).
quicksort([],[]).
quicksort([X|Xs],Ys) :-
  append(As2,[X|Bs2],Ys),
  part(Xs,X,As,Bs),
  quicksort(As,As2),
  quicksort(Bs,Bs2).

:- block append(-,?,-).
append([],Y,Y).
append([X|Xs],Ys,[X|Zs]) :-
  append(Xs,Ys,Zs).
```

```
:- block part(?,-,?,?),
         part(-,?,-,?),
         part(-,?,?,-).
part([],_,[],[]).
part([X|Xs],C,[X|As],Bs):-
  leq(X,C),
  part(Xs,C,As,Bs).
part([X|Xs],C,As,[X|Bs]):-
  grt(X,C),
  part(Xs,C,As,Bs).

:- block leq(?,-), leq(-,?).
leq(A,B) :- A =< B.

:- block grt(?,-), grt(-,?).
grt(A,B) :- A > B.
```

Figure 14: The `quicksort` program

Example 7.4 Figure 14 shows a version of `quicksort`. Assume the type $\{\texttt{quicksort}(nl,nl)$, $\texttt{append}(nl,nl,nl)$, $\texttt{leq}(num,num)$, $\texttt{grt}(num,num)$, $\texttt{part}(nl,num,nl,nl)\}$. The program is permutation simply typed for mode $\{\texttt{quicksort}(I,O)$, $\texttt{append}(I,I,O)$, $\texttt{leq}(I,I)$, $\texttt{grt}(I,I)$, $\texttt{part}(I,I,O,O)\}$. It is not permutation simply typed for mode $\{\texttt{quicksort}(O,I)$, $\texttt{append}(O,O,I)$, $\texttt{leq}(I,I)$, $\texttt{grt}(I,I)$, $\texttt{part}(O,I,I,I)\}$, because of the non-variable term `[X|Bs2]` in an output position.

As an aside, note that this program uses auxiliary predicates `leq` and `grt` to realise `block` declarations on the built-ins `=<` and `>`. Built-ins will be discussed in Sections 9.4 and 10.1. ◁

Example 7.5 Figure 15 shows a program that converts binary trees into lists or vice versa. The type of the program is $\{\texttt{treeList}(tree, list)$, $\texttt{append}(list, list, list)\}$. It is permutation simply typed for mode $\{\texttt{treeList}(I,O), \texttt{append}(I,I,O)\}$. However it is not permutation simply typed for mode $\{\texttt{treeList}(O,I), \texttt{append}(O,O,I)\}$, because of the non-variable term `[Label|RList]` in an output position. ◁

The persistence properties stated in Lemmas 5.3 and 5.10 are independent of the selection rule. We show a similar persistence property for permutation simply typed programs. However this property only holds if the derivation step is input-consuming, since otherwise output positions of the resolvent might become non-variable. In the following lemma, it is not actually assumed that the derivation step is input-consuming. It is only assumed that the input arguments of the selected atom are an instance of the input arguments of the clause head. While this is trivially *necessary* for a derivation step to be input-consuming, point (d) of the lemma states that it is also sufficient.

```
:- block treeList(-,-).
treeList(leaf,[]).
treeList(node(L,Label,R),List) :-
  append(LList,[Label|RList],List),
  treeList(L,LList),
  treeList(R,RList).
```

```
:- block append(-,?,-).
append([],Y,Y).
append([X|Xs],Ys,[X|Zs]) :-
  append(Xs,Ys,Zs).
```

Figure 15: Converting trees to lists or vice versa

Lemma 7.2 Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a π-simply typed query and $C = p_k(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \dots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ a ρ-simply typed, input-linear clause where $vars(C) \cap vars(Q) = \emptyset$. Suppose for some $k \in \{1, \dots, n\}$, $p_k(\mathbf{s}_k, \mathbf{t}_k)$ and $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})$ are unifiable and $\mathbf{s}_k$ is an instance of $\mathbf{v}_0$. Then there is an MGU $\theta = \theta_1\theta_2$ of $p_k(\mathbf{s}_k, \mathbf{t}_k)$ and $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})$ such that

a. $\mathbf{v}_0\theta_1 = \mathbf{s}_k$ and $dom(\theta_1) \subseteq vars(\mathbf{v}_0)$,

b. $\mathbf{t}_k\theta_2 = \mathbf{u}_{m+1}\theta_1$ and $dom(\theta_2) \subseteq vars(\mathbf{t}_k)$,

c. $dom(\theta) \subseteq vars(\mathbf{t}_k) \cup vars(\mathbf{v}_0)$,

d. $dom(\theta) \cap vars(\mathbf{s}_k) = \emptyset$, that is, the derivation step is input-consuming,

e. $dom(\theta) \cap vars(\mathbf{t}_1, \dots, \mathbf{t}_{k-1}, \mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_{k+1}, \dots, \mathbf{t}_n) = \emptyset$,

f. the resolvent of Q and C with selected atom $p_k(\mathbf{s}_k, \mathbf{t}_k)$ is $Der(\pi, \rho, k)$-simply typed.

Proof. Claim (a) follows from the assumption that $\mathbf{s}_k$ is an instance of $\mathbf{v}_0$.

Since $\mathbf{t}_k$ is a linear vector of variables, there is a substitution θ_2 such that $dom(\theta_2) \subseteq vars(\mathbf{t}_k)$ and $\mathbf{t}_k\theta_2 = \mathbf{u}_{m+1}\theta_1$, which shows (b).

Since Q is π-nicely moded, we have $vars(\mathbf{t}_k) \cap vars(\mathbf{s}_k) = \emptyset$, and therefore $vars(\mathbf{t}_k) \cap vars(\mathbf{v}_0\theta_1) = \emptyset$. Thus it follows by (b) that $\theta = \theta_1\theta_2$ is an MGU of $p_k(\mathbf{s}_k, \mathbf{t}_k)$ and $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})$. Claim (c) follows from (a) and (b). Claim (d) follows from (c) because $vars(\mathbf{t}_k) \cap vars(\mathbf{s}_k) = \emptyset$. Claim (e) follows from (c) because of the linearity of $(\mathbf{t}_1, \dots, \mathbf{t}_n, \mathbf{v}_0, \dots, \mathbf{v}_m)$.

By Lemmas 5.3 and 5.10, the resolvent is $Der(\pi, \rho, k)$-nicely moded and $Der(\pi, \rho, k)$-well typed. By (e), the vector of the output arguments of the resolvent is a linear vector of variables, and hence (f) follows. □

The following lemma states a persistence property similar to Lemma 7.2 (f) but for LD-resolvents only. Note that in this case, it is not necessary to require an input-linear clause. However, because of this weaker assumption, the lemma is not actually a corollary of Lemma 7.2.

Lemma 7.3 Every LD-resolvent of a simply typed query Q and a simply typed clause C, where $vars(C) \cap vars(Q) = \emptyset$, is simply typed.

Proof. By Lemma 7.1, the resolvent is simply moded. By Lemma 5.10, the resolvent is well typed. Therefore the resolvent is simply typed. □

The next lemma says that in an input-consuming derivation for a permutation simply typed program and query, it can be assumed without loss of generality that the output positions in each query are filled with variables that occur in the initial query or in some clause body used in the derivation. This is used to prove Theorem 8.3.

Lemma 7.4 Let P be a permutation simply typed, input-linear program, and Q_0 a permutation simply typed query. Let $\theta_0 = \emptyset$ and $\xi = \langle Q_0, \theta_0\rangle; \langle Q_1, \theta_1\rangle; \ldots$ be an input-consuming derivation of $P \cup \{Q_0\}$. Then for all $i \geq 0$, if x is a variable occurring in an output position in Q_i, then $x\theta_i = x$.

Proof. The proof is by induction on the position i in the derivation. The base case $i = 0$ is trivial since $\theta_0 = \emptyset$. Now suppose the result holds for some i and Q_{i+1} exists. By Lemma 7.2 (f), $Q_i\theta_i$ is permutation simply typed. Thus the result follows for $i + 1$ by Lemma 7.2 (e) and the inductive hypothesis. □

For permutation simply typed programs, `block` declarations can be used to ensure input-consuming derivations. However, before we show this, we first introduce a generalisation of permutation simply typed programs.

7.4 Permutation Robustly Typed Programs

Examples 7.4 and 7.5 suggest that Definition 7.2 is sometimes too restrictive. Both programs have an atom using `append` in a clause body where the second argument of that atom is non-variable. This means that these programs are not permutation simply typed when `append` is used in mode `append`(O, O, I).

It has been acknowledged previously by Apt and Etalle [AE93] that it is difficult to reason about queries where non-variable terms in output positions are allowed, but on the other hand, there are natural programs where this occurs. These authors assume that output positions in a query are always filled with variables, but consider allowing for non-variable terms as a direction for future work.

We define permutation *robustly-typedness*, which is a carefully crafted extension of permutation simply-typedness, allowing for non-variable but flat terms in certain output positions. The definition is more complicated than the definitions of previous correctness properties. The difficulty in designing such a concept is in ensuring that a persistence property analogous to Lemmas 5.3, 5.8, 5.10 and 7.2 holds. In particular, the definition is such that permutation robustly typed queries are type-consistent, which is important so that we can apply the results of Chapter 6.

In the sequel, we associate a label **free** or **bound** with each argument position of each predicate. The intuition behind these labels is as follows: an atom should be selectable only when it is non-variable in its bound input positions. Moreover, a query may contain a non-variable term in an output position only if the position is bound.

Definition 7.3 [free-bound-labelling] Let P be a permutation well typed program. A **free-bound-labelling** is a function assigning a label **free** or **bound** to each argument position of each predicate p, such that

- all positions of variable type are free,
- if there is a clause in P defining p whose head has a non-variable term in an input position, then this input position is bound.

We denote the projection of a vector of arguments $\mathbf{r}$ onto its free positions as $\mathbf{r}^{\mathsf{f}}$, and onto its bound positions as $\mathbf{r}^{\mathsf{b}}$. ◁

We assume that a free-bound-labelling is associated with each program, without making this explicit. As with assigning the mode and the type to a predicate, we do not propose a method of deciding which positions should be free or bound. In all our examples however, the choice is simple:

- an input position of p is bound if and only if there is some clause defining p whose head has a non-variable term in that position,
- an output position of p is bound if and only if there is some clause body containing an atom using p, which has a non-variable term in that position.

Note in particular that the conditions of the above definition can only be met if each clause head has a variable in each input position of variable type. By Definition 7.2, this requirement is clearly met by permutation simply typed programs.

Definition 7.4 [permutation robustly typed] Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a query and π a permutation on $\{1, \dots, n\}$. Then Q is **π-robustly typed** if it is π-nicely moded and π-well typed, $\mathbf{t}_1^{\mathsf{f}}, \dots, \mathbf{t}_n^{\mathsf{f}}$ is a vector of variables, and $\mathbf{t}_1^{\mathsf{b}}, \dots, \mathbf{t}_n^{\mathsf{b}}$ is a vector of flat type-consistent terms.

The clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$ is **π-robustly typed** if it is π-nicely moded and π-well typed, and

1. $\mathbf{t}_0^{\mathsf{f}}, \dots, \mathbf{t}_n^{\mathsf{f}}$ is a vector of variables, and $\mathbf{t}_0^{\mathsf{b}}, \dots, \mathbf{t}_n^{\mathsf{b}}$ is a vector of flat type-consistent terms,
2. if a position in $\mathbf{s}_{n+1}^{\mathsf{b}}$ of type τ is filled with a variable x, then x also fills a position of type τ in $\mathbf{t}_0^{\mathsf{b}}, \dots, \mathbf{t}_n^{\mathsf{b}}$.

A **permutation robustly typed** query (clause, program) and a **robustly typed** query (clause, program) **corresponding to** a query (clause, program) are defined in analogy to Definition 5.2. ◁

Permutation robustly typed programs are an extension of permutation simply typed programs. Consequently, Definition 7.2 coincides with Definition 7.4 in the case that all output positions are free, and all input positions of variable type are free. Note that a permutation simply typed program is also permutation robustly typed with respect to a free-bound-labelling where the input positions are labelled as explained just after Definition 7.3.

Example 7.6 Recall that we assume for all examples that an input position of a predicate p is bound if and only if there is some clause defining p whose head has a non-variable term in that position.

Consider again Example 7.3. The `permute` program (Figure 9 on page 57) is permutation simply typed in both modes and hence permutation robustly typed, assuming that all output positions are free.

Consider the `quicksort` program (Figure 14 on page 87) with the type given in Example 7.4. This program is permutation robustly typed in mode $\{\texttt{quicksort}(O, I)$, $\texttt{append}(O, O, I)$, $\texttt{leq}(I, I)$, $\texttt{grt}(I, I)$, $\texttt{part}(O, I, I, I)\}$, assuming the second position of `append` is the only bound output position. Note in particular that Condition 2 of Definition 7.4 is met for the recursive clause of `append`: the variable `Ys` fills an output position of the head and also an output position of the body. The program is also permutation robustly typed in mode $\{\texttt{quicksort}(I, O)$, $\texttt{append}(I, I, O)$, $\texttt{leq}(I, I)$, $\texttt{grt}(I, I)$, $\texttt{part}(I, I, O, O)\}$, assuming that all output positions are free.

Similarly, the `treeList` program (Figure 15 on page 88) is permutation robustly typed in mode $\{\texttt{treeList}(O, I)$, $\texttt{append}(O, O, I)\}$ assuming the second position of `append` is the only bound output position. It is also permutation robustly typed in mode $\{\texttt{treeList}(I, O)$, $\texttt{append}(I, I, O)\}$ assuming that all output positions are free. ◁

In Lemma 7.2, we showed a persistence property of permutation simply-typedness. There we did not actually assume that the derivation step is input-consuming, but only that the input arguments of the selected atom are an instance of the input arguments of the clause head. The following example shows that for permutation robustly-typedness, this is not sufficient.

Example 7.7 Consider $\texttt{append}(I, I, O)$ (Figure 10 on page 57) and assume that all positions are bound. Then the query

$$\texttt{append([], [], Bs), append([], Bs, [C|Cs])}$$

is (permutation) robustly typed. Suppose we want to resolve the second atom using the first clause for `append`. The vector $(\texttt{[]}, \texttt{Bs})$ is an instance of $(\texttt{[]}, \texttt{Y})$, and yet the MGU of $\texttt{append([], Bs, [C|Cs])}$ and $\texttt{append([], Y, Y)}$ binds `Bs` to `[C|Cs]`, and hence the derivation step would not be input-consuming. ◁

We now state a simple proposition which is illustrated in Figure 16. If we read $p(\mathbf{s}, \mathbf{t})$ as a selected atom and $p(\mathbf{v}, \mathbf{u})$ as a clause head, the proposition states a necessary condition for a derivation step to be input-consuming.

Proposition 7.5 Let $p(\mathbf{s}, \mathbf{t})$ and $p(\mathbf{v}, \mathbf{u})$ be two atoms that are unifiable with MGU θ, and suppose that $dom(\theta) \cap vars(\mathbf{s}) = \emptyset$. If in some position, $\mathbf{u}$ is filled with a variable x and $\mathbf{t}$ is filled with a non-variable term, and x also has a direct occurrence in $\mathbf{v}$ in position i, then $\mathbf{s}$ is non-variable in position i.

The following lemma shows a persistence property of permutation robustly-typedness.

Lemma 7.6 Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a π-robustly typed query, let $k \in \{1, \ldots, n\}$, and $C = p_k(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow q_1(\mathbf{u}_1, \mathbf{v}_1), \ldots, q_m(\mathbf{u}_m, \mathbf{v}_m)$ a ρ-robustly typed, input-linear clause where $vars(Q) \cap vars(C) = \emptyset$. Suppose that $p_k(\mathbf{s}_k, \mathbf{t}_k)$ and $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})$ are unifiable and

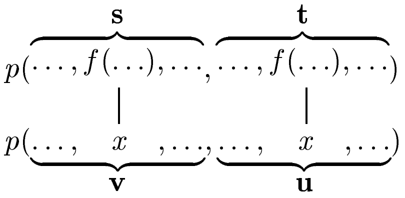
s
t
$p(\cdots,f(\ldots),\ldots,\ldots,f(\ldots),\ldots)$
$p(\ldots,\ x\ ,\ldots,\ldots,\ x\ ,\ldots)$
v
u

Figure 16: Illustrating Proposition 7.5

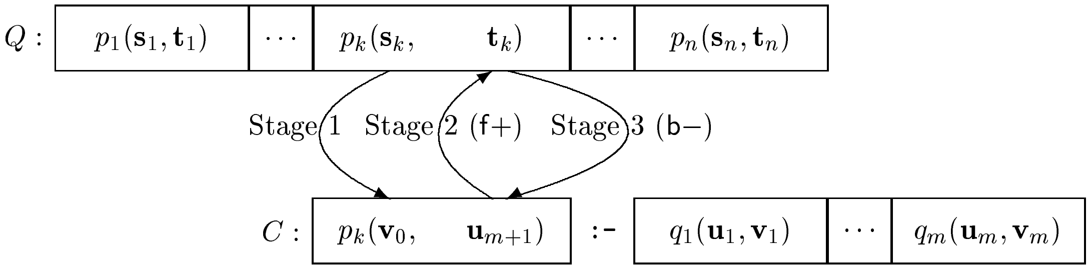
Q :
$p_1(\mathbf{s}_1,\mathbf{t}_1)$
$\cdots$
$p_k(\mathbf{s}_k,\quad \mathbf{t}_k)$
$\cdots$
$p_n(\mathbf{s}_n,\mathbf{t}_n)$
Stage 1
Stage 2 (f+)
Stage 3 (b−)
C :
$p_k(\mathbf{v}_0,\quad \mathbf{u}_{m+1})$
:-
$q_1(\mathbf{u}_1,\mathbf{v}_1)$
$\cdots$
$q_m(\mathbf{u}_m,\mathbf{v}_m)$

Figure 17: Data flow in the unification

1. $\mathbf{s}_k$ is an instance of $\mathbf{v}_0$, and
2. if a variable x fills positions i in $\mathbf{v}_0^{\mathsf{b}}$ and j in $\mathbf{u}_{m+1}^{\mathsf{b}}$, and position j in $\mathbf{t}_k^{\mathsf{b}}$ is non-variable, then position i in $\mathbf{s}_k^{\mathsf{b}}$ is also non-variable.

Then there is an MGU θ of $p_k(\mathbf{s}_k,\mathbf{t}_k)$ and $p_k(\mathbf{v}_0,\mathbf{u}_{m+1})$ such that

a. $\mathit{dom}(\theta)\cap \mathit{vars}(\mathbf{s}_k)=\emptyset$, that is, the derivation step is input-consuming,

b. the resolvent of Q and C with selected atom $p_k(\mathbf{s}_k,\mathbf{t}_k)$ is $\mathit{Der}(\pi,\rho,k)$-robustly typed.

Proof. We show how θ is computed, where we consider three stages. In the first, $\mathbf{s}_k$ and $\mathbf{v}_0$ are unified. In the second, the output positions are unified where the bindings go from C to Q. In the third, the output positions are unified where the bindings go from Q to C. Figure 17 illustrates which variables are bound in each stage. The first three parts of the proof correspond to the three stages of the unification.

Part 1: (unifying $\mathbf{s}_k$ and $\mathbf{v}_0$). Since by assumption 1, $\mathbf{s}_k$ is an instance of $\mathbf{v}_0$, there is a (minimal) substitution θ_1 such that $\mathbf{v}_0\theta_1=\mathbf{s}_k$. We show that the following statements hold:

S1a $\mathit{dom}(\theta_1)\cap \mathit{vars}(\mathbf{s}_k)=\emptyset$.

S1b $\mathit{dom}(\theta_1)\cap \mathit{vars}(\mathbf{v}_1,\ldots,\mathbf{v}_m,\mathbf{t}_1,\ldots,\mathbf{t}_n)=\emptyset$.

S1c Let x be a variable occurring directly in a position of type τ in $\mathbf{u}_{m+1}^{\mathsf{b}}\theta_1$, such that $\mathbf{t}_k^{\mathsf{b}}$ is non-variable in this position. Then $x\notin \mathit{vars}(\mathbf{s}_k)$. Moreover, x can only occur in $\mathbf{v}_1,\ldots,\mathbf{v}_m,\mathbf{t}_1,\ldots,\mathbf{t}_n$ in a bound position of type τ, and the occurrence must be direct.

S1d $vars(\mathbf{u}_{m+1}\theta_1) \cap vars(\mathbf{t}_k) = \emptyset$.

S1a holds by the construction of θ_1.

S1b holds since by Definition 7.4 and the assumption that C is input-linear, we have that $\mathbf{v}_0, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_n$ is linear.

Let x be a variable occurring directly in a position of type τ in $\mathbf{u}^{\mathsf{b}}_{m+1}\theta_1$, such that $\mathbf{t}^{\mathsf{b}}_k$ is non-variable in this position. Let y be the variable in the same position in $\mathbf{u}^{\mathsf{b}}_{m+1}$. Suppose that $y \in vars(\mathbf{v}_0)$. Then by Definition 7.4, y occurs *directly* in $\mathbf{v}^{\mathsf{b}}_0$, say in position i, and by assumption 2, $\mathbf{s}^{\mathsf{b}}_k$ is non-variable in position i. Thus $y\theta_1$ is not a variable, which is a contradiction. Therefore $y \notin vars(\mathbf{v}_0)$. Hence $y \notin dom(\theta_1)$ and thus $x = y$ and $x \notin vars(\mathbf{s}_k)$. Furthermore it follows by Definition 7.4 that x can only occur in $\mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_n$ in a bound position of type τ, and the occurrence must be direct. Thus S1c holds.

Since Q is permutation nicely moded, $vars(\mathbf{s}_k) \cap vars(\mathbf{t}_k) = \emptyset$ and hence $vars(ran(\theta_1)) \cap vars(\mathbf{t}_k) = \emptyset$. Thus S1d holds.

Part 2: (unifying $\mathbf{t}_k$ and $\mathbf{u}_{m+1}\theta_1$ in each position where either the argument in $\mathbf{t}_k$ is a variable, or the arguments in $\mathbf{t}_k$ and $\mathbf{u}_{m+1}\theta_1$ are both non-variable). Note that this includes all positions in $\mathbf{t}^{\mathsf{f}}_k$ and $\mathbf{u}^{\mathsf{f}}_{m+1}\theta_1$, but may also include positions in $\mathbf{t}^{\mathsf{b}}_k$ and $\mathbf{u}^{\mathsf{b}}_{m+1}\theta_1$. Since, by S1b, $\mathbf{t}_k\theta_1 = \mathbf{t}_k$, Part 2 covers precisely the output positions where the binding "goes from $\mathbf{u}_{m+1}\theta_1$ to $\mathbf{t}_k\theta_1$" (see Figure 17). We denote by $\mathbf{t}^{\mathsf{f}+}_k$ the projection of $\mathbf{t}_k$ onto the positions where the argument in $\mathbf{t}_k$ is a variable, or the arguments in $\mathbf{t}_k$ and $\mathbf{u}_{m+1}\theta_1$ are both non-variable, and by $\mathbf{t}^{\mathsf{b}-}_k$ the projection onto all other positions, and likewise for $\mathbf{u}_{m+1}\theta_1$.

By S1d, $vars(\mathbf{u}^{\mathsf{f}+}_{m+1}\theta_1) \cap vars(\mathbf{t}^{\mathsf{f}+}_k) = \emptyset$. Thus there is a minimal substitution θ' such that $\mathbf{t}^{\mathsf{f}+}_k\theta' = \mathbf{u}^{\mathsf{f}+}_{m+1}\theta_1$. Let $\theta_2 = \theta_1\theta'$. Then by S1b and S1d, $\mathbf{t}^{\mathsf{f}+}_k\theta_2 = \mathbf{u}^{\mathsf{f}+}_{m+1}\theta_2$. We show the following statements:

S2a $dom(\theta_2) \cap vars(\mathbf{s}_k) = \emptyset$.

S2b $dom(\theta_2) \cap vars(\mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_{k-1}, \mathbf{t}^{\mathsf{b}-}_k, \mathbf{t}_{k+1}, \dots, \mathbf{t}_n) = \emptyset$.

S2c Let x be a variable occurring directly in a position of type τ in $\mathbf{u}^{\mathsf{b}-}_{m+1}\theta_2$.[1] Then $x \notin vars(\mathbf{s}_k)$, and x can only occur in $\mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_{k-1}, \mathbf{t}^{\mathsf{b}-}_k, \mathbf{t}_{k+1}, \dots, \mathbf{t}_n$ in a bound position of type τ, and the occurrence must be direct.

S2d $vars(\mathbf{u}_{m+1}\theta_2) \cap vars(\mathbf{t}^{\mathsf{b}-}_k) = \emptyset$.

Since $vars(\mathbf{s}_k) \cap vars(\mathbf{t}_k) = \emptyset$, we have $dom(\theta') \cap vars(\mathbf{s}_k) = \emptyset$. This and S1a imply S2a.

S2b holds because S1b holds and $(\mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_n)$ is linear.

[1] By definition of the superscript notation $\mathsf{b}-$ we have that $\mathbf{t}^{\mathsf{b}-}_k$ is non-variable in this position.

By S1d, $dom(\theta') \cap vars(\mathbf{u}^{\mathsf{b}-}_{m+1}\theta_1) = \emptyset$. This together with S1c implies S2c. Furthermore, because of the linearity of $\mathbf{t}_k$, S2d follows.

PART 3: (unifying $\mathbf{t}^{\mathsf{b}-}_k$ and $\mathbf{u}^{\mathsf{b}-}_{m+1}\theta_2$). By S1d, $dom(\theta') \cap vars(\mathbf{u}^{\mathsf{b}-}_{m+1}\theta_1) = \emptyset$, and thus $\mathbf{u}^{\mathsf{b}-}_{m+1}\theta_2 = \mathbf{u}^{\mathsf{b}-}_{m+1}\theta_1$. Therefore, by the definition of the superscript $\mathsf{b}-$ in Part 2, $\mathbf{u}^{\mathsf{b}-}_{m+1}\theta_2$ is a vector of variables. By S2d, $vars(\mathbf{u}^{\mathsf{b}-}_{m+1}\theta_2) \cap vars(\mathbf{t}^{\mathsf{b}-}_k) = \emptyset$, so that there is a minimal substitution θ'' such that $\mathbf{u}^{\mathsf{b}-}_{m+1}\theta_2\theta'' = \mathbf{t}^{\mathsf{b}-}_k$. Let $\theta_3 = \theta_2\theta''$. Then, by S2b, we have $\mathbf{u}^{\mathsf{b}-}_{m+1}\theta_3 = \mathbf{t}^{\mathsf{b}-}_k\theta_3$. We show the following statements:

S3a $dom(\theta_3) \cap vars(\mathbf{s}_k) = \emptyset$.

S3b $(\mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_{k-1}, \mathbf{t}_{k+1}, \dots, \mathbf{t}_n)\theta_3$ is linear and has flat type-consistent terms in all bound positions and variables in all free positions.

By S2c, $dom(\theta'') \cap vars(\mathbf{s}_k) = \emptyset$. This and S2a imply S3a.

Suppose x is a variable in $\mathbf{u}^{\mathsf{b}-}_{m+1}\theta_2$ occurring in a position i of type τ, and x also occurs in $(\mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_{k-1}, \mathbf{t}_{k+1}, \dots, \mathbf{t}_n)$. By S2c, the latter occurrence of x is in a bound position of type τ, and the *only* occurrence of x in $(\mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_{k-1}, \mathbf{t}_{k+1}, \dots, \mathbf{t}_n)$. Let I be the set of positions where x occurs in $\mathbf{u}^{\mathsf{b}-}_{m+1}\theta_2$, and let T be the set of terms occurring in $\mathbf{t}^{\mathsf{b}-}_k$ in positions in I. Then T is a set of variable-disjoint, flat terms. Therefore their most general common instance $x\theta''$ is a flat term and $x\theta''$ is type-consistent with respect to τ. Moreover, since $(\mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_{k-1}, \mathbf{t}^{\mathsf{b}-}_k, \mathbf{t}_{k+1}, \dots, \mathbf{t}_n)$ is linear, $vars(x\theta'') \cap vars(\mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_{k-1}, \mathbf{t}_{k+1}, \dots, \mathbf{t}_n) = \emptyset$ and hence it follows that $(\mathbf{v}_1, \dots, \mathbf{v}_m, \mathbf{t}_1, \dots, \mathbf{t}_{k-1}, \mathbf{t}_{k+1}, \dots, \mathbf{t}_n)\theta''$ is linear and type-consistent. This and S2b imply S3b.

PART 4: Defining $\theta = \theta_3$ it follows that $p_k(\mathbf{s}_k, \mathbf{t}_k)\theta = p_k(\mathbf{v}_0, \mathbf{u}_{m+1})\theta$. By S3a, $\mathbf{s}_k\theta = \mathbf{s}_k$, which shows (a). By S3b and Lemmas 5.3 and 5.10, the resolvent of Q and C is $Der(\pi, \rho, k)$-robustly typed, which shows (b). □

From Lemma 7.6, we can conclude that permutation robustly typed programs are type-consistent with respect to input-consuming derivations. Of course, this holds in particular for permutation *simply* typed programs.

Lemma 7.7 Every permutation robustly typed program is type-consistent with respect to input-consuming derivations.

PROOF. Let P be a permutation robustly typed program and Q a permutation robustly typed query. Trivially, assumption 1 in Lemma 7.6 is necessary for a derivation step to be input-consuming. By Proposition 7.5, assumption 2 in Lemma 7.6 is also necessary for a derivation step to be input-consuming. Hence by Lemma 7.6 (b), any input-consuming derivation of $P \cup \{Q\}$ contains only permutation robustly typed queries. By Definition 7.4, every permutation robustly typed query is type-consistent, and hence P is type-consistent with respect to input-consuming derivations. □

We define *input selectability.* We will see that in a program with input selectability, an atom is selectable only if it meets assumptions 1 and 2 in Lemma 7.6.

```
:- block permute(-,-).              :- block delete(?,-,-).
permute([],[]).                     delete(X,[X|Z],Z).
permute([U|X],Y) :-                 delete(X,[U|Y],[U|Z]) :-
  permute(X,Z),                       delete(X,Y,Z).
  delete(U,Y,Z).
```

Figure 18: The `permute` program with `block` declarations

Definition 7.5 [input selectability] Let P be a permutation robustly typed program. P has **input selectability** if for every permutation robustly typed query Q, an atom in Q is selectable in P if and only if it is non-variable in all bound input positions. $\lhd$

Input selectability is similar to the condition that "the delay declarations imply matching" [AL95].

For a program to have input selectability, the `block` declarations must be such that an atom whose free output positions are all variable is selectable if and only if all bound input positions are non-variable.

Example 7.8 Figure 18 shows the `permute` program of Figure 9 on page 57, with `block` declarations added. Here we only consider `delete`. Let us first assume mode `delete`(I, O, I), with a free-bound-labelling `delete`$(free, free, bound)$ as explained on page 90. Then the `block` declarations ensure input selectability. Now assume mode `delete`(O, I, O) with a free-bound-labelling `delete`$(free, bound, free)$. For this mode, the `block` declarations also ensure input selectability. Hence the `block` declarations ensure input selectability with respect to two different modes. $\lhd$

The following proposition states that input selectability ensures that every selectable atom meets assumptions 1 and 2 in Lemma 7.6.

Proposition 7.8 Let P be a permutation robustly typed, input-linear program with input selectability, $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a π-robustly typed query, $k \in \{1, \dots, n\}$, and $C = p_k(\mathbf{v}_0, \mathbf{u}_{m+1}) \leftarrow B$ a clause in P. Suppose that $p_k(\mathbf{s}_k, \mathbf{t}_k)$ is selectable and $p_k(\mathbf{s}_k, \mathbf{t}_k)$ and $p_k(\mathbf{v}_0, \mathbf{u}_{m+1})$ are unifiable. Then assumptions 1 and 2 in Lemma 7.6 are fulfilled.

PROOF. Since $p(\mathbf{s}_k, \mathbf{t}_k)$ is selectable in P, it follows that $\mathbf{s}_k$ is non-variable in all bound positions. By Definition 7.4, $\mathbf{v}_0$ is a linear vector having flat terms in all bound positions, and variables in all other positions. Thus assumption 1 is fulfilled. Assumption 2 is fulfilled since $\mathbf{s}_k$ is non-variable in *all* bound positions. $\Box$

The following theorem is a consequence of Proposition 7.8 and Lemma 7.6.

Theorem 7.9 Let P be a permutation robustly typed, input-linear program with input selectability, and Q a permutation robustly typed query. Then every delay-respecting derivation of $P \cup \{Q\}$ is input-consuming.

Note that the converse is not true. There could be input-consuming derivation steps which are not delay-respecting.

The following example illustrates why it is an advantage that the selected atom only has to be non-variable in the *bound* input positions.

Example 7.9 Consider the `block` declaration for `append` in Figure 15 (page 88). Given that the usual modes for `append` are `append`(I, I, O) and `append`(O, O, I), one might expect a general theory to say that an atom using `append` should be selectable if either the first two arguments or the third argument are non-variable. This would correspond to the `block` declaration

```
:- block append(-,?,-), append(?,-,-).
```

However, the simpler `block` declaration is justified since by Definition 7.3, we may assume that for the mode `append`(I, I, O), the second position is free. The simpler `block` declaration is the one usually given [HL94, Lüt93, MT95], but to the best of our knowledge, its adequacy has never been explained on such an abstract level. ◁

The next example illustrates why in Definition 7.5, input selectability is defined with respect to atoms in permutation robustly typed queries.

Example 7.10 Consider `append`(O, O, I) where the second position is the only bound output position, as in `quicksort`(O,I) (Figure 14 on page 87) or `treeList`(O,I) (Figure 15 on page 88). The program for `append` has input selectability. $Q =$ `append(A,[B|Bs],[1])` is a permutation robustly typed query, and its atom is selectable. The atom `append([],[],C)` is also selectable, although its input position is variable. This does not contradict Definition 7.5, since the first position is free, and thus this atom cannot occur in a permutation robustly typed query with respect to mode `append`(O, O, I). ◁

Looking at Definition 7.4, one is tempted to think that it is best to associate the label *bound* with *all* output positions, because that would make the definition less restrictive. However, we require a program to have input selectability in each of its modes. Since input selectability is defined with respect to atoms in permutation robustly typed queries, and permutation robustly typed queries are defined with respect to given free and bound positions, it turns out that the choice of free and bound positions constrains the possible set of modes. This is illustrated in the following example.

Example 7.11 Consider `append`(O, O, I), where *both* output positions are bound, and the `block` declaration is as in Figure 15 (page 88). Note that this `block` declaration is intended to allow for the current mode `append`(O, O, I), but also alternatively for mode `append`(I, I, O). Now consider the query

$$\texttt{append(Cs,Ds,[1,2,3]), append([A|As],[B|Bs],Cs)}$$

This query is robustly typed with respect to the current mode `append`(O, O, I). The second atom is selectable although it is variable in its only bound input position. Therefore the program does not have input selectability. This could be rectified by replacing the `block` declaration with

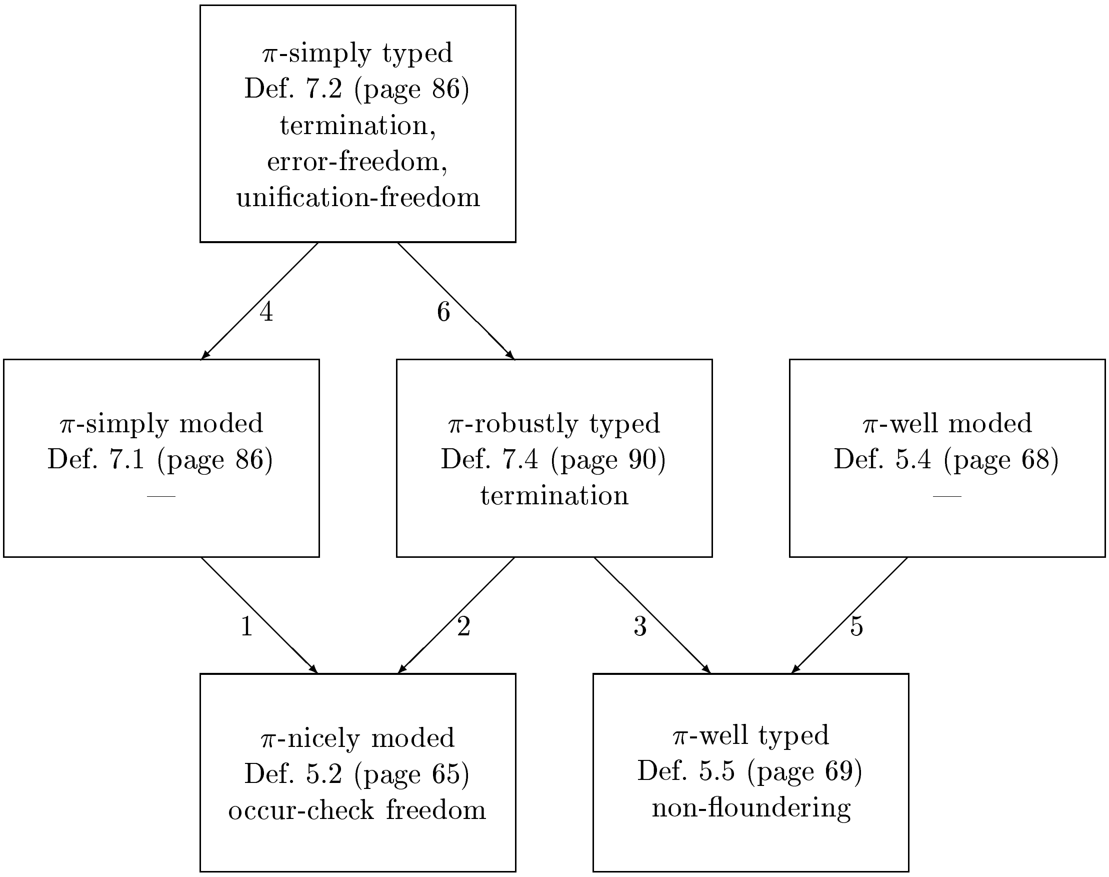

Figure 19: The correctness properties

```
:- block append(?,?,-).
```

but then the program could not be used in mode append(I, I, O) anymore. However, we have not encountered a case where a "natural" mode of a program was ruled out because of this problem. ◁

7.5 Summary of the Correctness Properties

We now give an overview of the correctness properties for programs and queries that are used in this thesis. Figure 19 shows all the properties. An arrow stands for implication. In each box, we quote the definition of the property and state the main purpose for which it is used, apart from the obvious purpose of defining other properties.

The arrows 1–4 correspond to implications by definition. As stated in Proposition 5.13, permutation well-modedness is permutation well-typedness for the special case that the only type is the type *all_ground*. Moreover, permutation simply-typedness is permutation robustly-typedness for the special case that all output positions, and exactly the input positions of variable type, are free.

Chapter 8

Termination and block Declarations

In this chapter, we consider termination of logic programs with `block` declarations. In Section 6.5, we said that often, assuming input-consuming derivations is not sufficient to ensure termination. We now make an additional assumption, namely that derivations are *left-based.* These are derivations where (allowing for some exceptions explained in the next section) the leftmost selectable atom is selected in each step. This is intended to model derivations in the common implementations of Prolog with `block` declarations [SIC98]. Since "leftmost" obviously refers to the textual order of atoms in a query, we cannot make the simplifying assumption in this chapter that the textual order is always identical to the producer-consumer order, as discussed in Subsection 5.3.2. That is, whenever we use one of the correctness properties such as permutation nicely-modedness, we cannot assume that the permutations are always the identity.

8.1 Two Approaches to the Termination Problem

Our first approach to the termination problem is focused on *speculative output bindings* [Nai92], that is, output bindings made before it is known that a solution exists. This is a well-known source of non-termination associated with delay declarations. We present two complementing methods for dealing with this problem and thus proving (or ensuring) termination. Which method must be applied depends on the program and on the mode being considered. The first method exploits the fact that a program does not *use* any speculative bindings, by ensuring that no atom ever delays for all left-based derivations. The second method exploits the fact that a program does not *make* any speculative bindings. This approach builds on previous heuristics [Nai85, Nai92] and relies on conditions which are easy to check. However, it is quite limited.

The second approach to the termination problem builds on Chapter 6 but assumes that derivations are not only input-consuming, but also *left-based.* The question is: what shall we do about predicates that are not atom-terminating? A good intuitive explanation for the problem these predicates pose is that they may loop when called with *insufficient* input. For example, consider the `permute` program as shown in Figure 20. For `permute`(O, I) the query `permute(A,[1|B])` has insufficient input and may loop.

```
:- block permute(-,-).             :- block delete(?,-,-).
permute([],[]).                    delete(X,[X|Z],Z).
permute([U|X],Y) :-                delete(X,[U|Y],[U|Z]) :-
  delete(U,Y,Z),                     delete(X,Y,Z).
  permute(X,Z).
```

Figure 20: Placing recursive calls last for `permute`

However, the query `permute(A,[1,2])` has sufficient input and terminates. The idea for proving termination is that, for such predicates, calls with insufficient input must never arise. This can be ensured by appropriate ordering of atoms in the clause bodies, as demonstrated in Figure 20 (in contrast to Figure 18 on page 95). This may actually work in several modes, provided not too many predicates have this undesirable property.

Both approaches implicitly rely on termination of LD-derivations, in that they translate the termination problem for a program with delay declarations to the same problem for a corresponding program executed left-to-right. It is assumed that, for the corresponding program, termination can be shown using some existing technique [Apt97, AP90, DD94, DVB92, DD93, DD98, EBC99, LS96, LS97]. For the example programs we give, except for the program in Figure 13 on page 81, Lindenstrauss has confirmed to us that the TermiLog system [LSS97] can automatically prove termination for the corresponding programs assuming LD-derivations.

This chapter is organised as follows. The next section defines left-based derivations. Section 8.3 presents the first approach. Section 8.4 presents the second approach. Section 8.5 discusses the results of this chapter and compares the two approaches.

8.2 Left-Based Derivations

We now attempt to formalise derivations in most existing Prolog implementations. Some authors have considered a selection rule stating that in each derivation step, the leftmost selectable atom is selected. Boye claims that several modern Prolog implementations and even Gödel [HL94] use this selection rule [Boy96, page 123]. Apt and Luitjes [AL95] have interpreted Naish's [Nai86, Nai92] notion of a "default left-to-right" selection rule in this way. Naish has not specified precisely what a default left-to-right selection rule is, but he is aware of the fact that the selection rule of most Prolog implementations does not state that the leftmost selectable atom is always selected.[1]

As an aside, Apt and Luitjes also claim that Lüttringhaus-Kappel [Lüt93] has considered this selection rule, but this is definitely not the case, since Lüttringhaus-Kappel considers arbitrary delay-respecting derivations.

Prolog implementations do not usually guarantee the order in which two simultaneously woken atoms are selected. In the following, we define *waiting* atoms, which are the atoms that were previously delayed, together with all their descendants. We specify that waiting atoms are always preferred over other atoms, but we do not specify the relative selection order of two waiting atoms.

[1] Personal communication.

Definition 8.1 [waiting atom, left-based derivation] Let P be a program and let $Q_0; \ldots; Q_i \ldots$ be a delay-respecting derivation, where $Q_i = R_1, R_2$, and R_1 contains no atom that is selectable in P. Then every descendant of every atom in R_1 is **waiting**. A delay-respecting derivation $Q_0; Q_1 \ldots$ is **left-based** if in each Q_i, an atom which is not waiting is selected *only* if there is no selectable atom to the left of it in Q_i. ◁

Example 8.1 Consider the following program:

```
:- block a(-).          :- block b(-)
a(1).                   b(X) :- b2(X).

c(1).                   b2(1).               d.
```

The following is a left-based derivation. Waiting atoms are overlined. The selected atom in each step is underlined, as in previous examples.

$$\overline{\mathtt{a(X)}}, \overline{\mathtt{b(X)}}, \underline{\mathtt{c(X)}}, \mathtt{d} \leadsto \overline{\mathtt{a(1)}}, \underline{\overline{\mathtt{b(1)}}}, \mathtt{d} \leadsto \overline{\mathtt{a(1)}}, \underline{\overline{\mathtt{b2(1)}}}, \mathtt{d} \leadsto \underline{\overline{\mathtt{a(1)}}}, \mathtt{d} \leadsto \underline{\mathtt{d}} \leadsto \Box.$$

Note that $\mathtt{b(1)}$ and $\mathtt{b2(1)}$ are waiting and selectable, and therefore they can be selected although there is the selectable atom $\mathtt{a(1)}$ to the left. In contrast, $\mathtt{d}$ is never waiting and can only be selected in the last step. The following is another left-based derivation. Here, the leftmost selectable atom is selected in each step.

$$\overline{\mathtt{a(X)}}, \overline{\mathtt{b(X)}}, \underline{\mathtt{c(X)}}, \mathtt{d} \leadsto \underline{\overline{\mathtt{a(1)}}}, \overline{\mathtt{b(1)}}, \mathtt{d} \leadsto \underline{\overline{\mathtt{b(1)}}}, \mathtt{d} \leadsto \underline{\overline{\mathtt{b2(1)}}}, \mathtt{d} \leadsto \underline{\mathtt{d}} \leadsto \Box.$$

◁

We do not believe that it would be useful or practical to try to specify the selection rule of existing Prolog implementations more precisely. Our experiments suggest that it depends on the order in which variables are bound when two terms are unified, which is clearly an artefact of the implementation. We are confident however that derivations in most Prolog implementations are left-based. To the best of our knowledge, this has not been formalised previously, although Naish has considered such derivations informally [Nai86, Nai92].

We can state the following simple lemma about left-based derivations.

Lemma 8.1 Let P be a program and ξ a left-based derivation such that in each query in ξ, the leftmost atom is selectable in P. Then ξ is an LD-derivation.

Proof. Let $\xi = Q_0; Q_1; \ldots$. We show by induction that for all $i \geq 0$, Q_i contains no waiting atom, and the leftmost atom in Q_i is selected in the step $Q_i; Q_{i+1}$.

In Q_0, no atom is waiting, and hence the leftmost atom is selected. Now suppose that for some $i > 0$, Q_i contains no waiting atom. Then, since the leftmost atom of Q_i is selectable, it is selected. Moreover, no atom in Q_{i+1} is waiting. □

8.3 Termination and Speculative Bindings

In this section, we present two complementing methods for showing termination. These are explained in the following example.

Example 8.2 Consider the `permute` program (Figure 18 on page 95). The derivation in Example 6.2 loops because `delete` produces a *speculative output binding* [Nai92]: The output variable `Z'` is bound before it is known that this binding will never have to be undone. Assuming left-based derivations, termination in both modes can be ensured by swapping the two body atoms of the recursive clause for `permute`. The modified program is shown in Figure 20 on page 99. This technique has been described as *placing recursive calls last* [Nai92]. To explain why the program terminates, we have to apply a different reasoning for the different modes.

In mode `permute`(O, I), the atom that produces the speculative output occurs *textually before* the atom that consumes it. This means that the consumer waits until the producer has completed, that is, undone the speculative binding. The program does not *use* speculative bindings. In mode `permute`(I, O), `delete` is used in mode `delete`(I, O, I), and in this mode it does not *make* speculative bindings.

Observe that in mode `permute`(O, I), termination for this example depends on derivations being left-based, and therefore any method which abstracts from the textual order must fail. ◁

The methods presented in this section can be used to prove termination for `permute` (Figure 20 on page 99), `treeList` (Figure 15 on page 88), `plus_one` (Figure 13 on page 81), and `delete` as defined in Example 7.1. However, they do not work for `quicksort` (Figure 14 on page 87) and `nqueens` (which will be shown in Figure 22 on page 106).

8.3.1 Termination by not Using Speculative Bindings

In LD-derivations, speculative bindings are never used [Nai92]. By Lemma 8.1, a left-based derivation *is* an LD-derivation, provided the leftmost atom in each query in the derivation is always selectable. Moreover, by Definition 5.5, the leftmost atom in a well typed query is always non-variable in its input positions of non-variable type. This implies the following theorem.

Theorem 8.2 Let Q be a well typed query and P a well typed program such that an atom is selectable in P whenever its input positions of non-variable type are non-variable. Then every left-based derivation of $P \cup \{Q\}$ is an LD-derivation.

We now give two examples of programs where by Theorem 8.2, we can use any method for LD-derivations [DD94] to show termination for any well typed query.

Example 8.3 Consider `permute`(O, I) (Figure 20 on page 99) with either of the types given in Example 5.7. This program is well typed. ◁

Example 8.4 Consider the `delete` program in Example 7.1. Assuming either of the types given in Example 5.7, this program is well typed. Moreover, this is a program for which Section 8.4 is not applicable, because the program is not permutation robustly typed. ◁

8.3.2 Termination by not Making Speculative Bindings

Some programs and queries have the property that there cannot be any failing derivations [PR99]. Bossi and Cocco [BC99] have defined a class of such programs called *noFD*, assuming LD-derivations. We define *non-speculative* programs, which is a similar concept. The definition is based on permutation *simply* typed programs.

Definition 8.2 [non-speculative] A program P is **non-speculative** if it is permutation simply typed, input-linear, and every simply typed atom using a predicate in P is unifiable with some clause head in P. $\lhd$

Note that unlike noFD programs, non-speculative programs must be input-linear. Thus in particular, they must not use the equality predicate in mode $\texttt{=}(I, I)$, that is, they must not use *equality tests*.

Example 8.5 We give some examples of non-speculative programs. Both versions of the `permute` program (Figure 18 on page 95 and Figure 20 on page 99), assuming either of the types given in Example 5.7, are non-speculative in mode $\{\texttt{permute}(I, O), \texttt{delete}(I, O, I)\}$. Every simply typed atom is unifiable with at least one clause head.

Both versions are *not* non-speculative in mode $\{\texttt{permute}(O, I), \texttt{delete}(O, I, O)\}$, because `delete(A,[],B)` is a simply typed atom which is not unifiable with any clause head.

The program `treeList` (Figure 15 on page 88) is non-speculative in the mode $\{\texttt{treeList}(I, O), \texttt{append}(I, I, O)\}$. It is not non-speculative in mode $\{\texttt{treeList}(O, I), \texttt{append}(O, O, I)\}$ because it is not permutation simply typed (see Example 7.5).

Now consider the `plus_one` program (Figure 13 on page 81) and suppose all arguments have type $\{\texttt{0}, \texttt{succ}(\texttt{0}), \texttt{succ}(\texttt{succ}(\texttt{0})), \ldots\}$. Then the program is non-speculative. We will see later that this gives us an alternative way of proving termination for this program. $\lhd$

A delay-respecting derivation for a non-speculative program P with input selectability and a permutation simply typed query cannot fail.[2] However it could still be infinite. The following theorem says that this can only happen if the simply typed program corresponding to P has an infinite *LD*-derivation for this query.

Theorem 8.3 Let P be a non-speculative program with input selectability and P' a simply typed program corresponding to P. Let Q be a permutation simply typed query and Q' a simply typed query corresponding to Q. If there is an infinite delay-respecting derivation of $P \cup \{Q\}$, then there is an infinite LD-derivation of $P' \cup \{Q'\}$.

PROOF. For simplicity assume that Q and each clause body in P do not contain two identical atoms. Let $Q_0 = Q$, $\theta_0 = \emptyset$ and

$$\xi = \langle Q_0, \theta_0 \rangle ; \langle Q_1, \theta_1 \rangle ; \ldots$$

[2] It can also not *flounder*, as we will see in Section 9.3.

be a delay-respecting derivation of $P \cup \{Q\}$. The idea is to construct an LD-derivation ξ' of $P' \cup \{Q'\}$ such that whenever ξ uses a clause C, then ξ' uses the corresponding clause C' in P'. It will then turn out that if ξ' is finite, ξ must also be finite.

We call an atom a **resolved in** ξ **at** i if a occurs in Q_i but not in Q_{i+1}. We call a **resolved in** ξ if for some i, a is resolved in ξ at i. Let $Q'_0 = Q'$ and $\theta'_0 = \emptyset$. We construct an LD-derivation

$$\xi' = \langle Q'_0, \theta'_0 \rangle; \langle Q'_1, \theta'_1 \rangle; \ldots$$

of $P' \cup \{Q'\}$ showing that for each $i \geq 0$ the following hold:

S1(i) If $q(\mathbf{u}, \mathbf{v})$ is an atom in Q'_i that is not resolved in ξ, then $vars(\mathbf{v}\theta'_i) \cap dom(\theta_j) = \emptyset$ for all $j \geq 0$.

S2(i) Let x be a variable such that, for some $j \geq 0$, $x\theta_j = f(\ldots)$. Then $x\theta'_i$ is either a variable or $x\theta'_i = f(\ldots)$.

We first show S1(0) and S2(0). Let $q(\mathbf{u}, \mathbf{v})$ be an atom in Q'_0 that is not resolved in ξ. Since $\theta'_0 = \emptyset$, it follows that $\mathbf{v}\theta'_0 = \mathbf{v}$. Furthermore, by Lemmas 7.3 and 7.4 and since $q(\mathbf{u}, \mathbf{v})$ is not resolved in ξ, we have $\mathbf{v}\theta_j = \mathbf{v}$ for all j. Thus S1(0) holds. S2(0) holds because $\theta'_0 = \emptyset$.

Now assume that for some i, $\langle Q'_i, \theta'_i \rangle$ is defined, Q'_i is not empty, and S1(i) and S2(i) hold. Let $p(\mathbf{s}, \mathbf{t})$ be the leftmost atom of Q'_i. We define a derivation step $\langle Q'_i, \theta'_i \rangle; \langle Q'_{i+1}, \theta'_{i+1} \rangle$ with $p(\mathbf{s}, \mathbf{t})$ as the selected atom, and show that S1($i+1$) and S2($i+1$) hold.

Case 1: $p(\mathbf{s}, \mathbf{t})$ is resolved in ξ at l for some l. Consider the simply typed clause $C' = h \leftarrow B'$ corresponding to the *uniquely renamed* clause (using the same renaming) used in ξ to resolve $p(\mathbf{s}, \mathbf{t})$. Since $p(\mathbf{s}, \mathbf{t})$ is resolved in ξ at l, and ξ is delay-respecting and P has input selectability, it follows that $p(\mathbf{s}, \mathbf{t})\theta_l$ is non-variable in all bound input positions. Thus each bound input position of $p(\mathbf{s}, \mathbf{t})$ must be filled by a non-variable term or a variable x such that $x\theta_l = f(\ldots)$ for some f. Moreover, $p(\mathbf{s}, \mathbf{t})\theta'_i$ must have non-variable terms in all bound input positions since $Q'_i\theta'_i$ is well typed. Thus it follows by S2(i) that in each bound input position, $p(\mathbf{s}, \mathbf{t})\theta'_i$ has the same top-level functor as $p(\mathbf{s}, \mathbf{t})\theta_l$, and since h has flat terms in the bound input positions, there is an MGU ϕ'_i of $p(\mathbf{s}, \mathbf{t})\theta'_i$ and h. We use C' for the step $\langle Q'_i, \theta'_i \rangle; \langle Q'_{i+1}, \theta'_{i+1} \rangle$.

We must show S1($i+1$) and S2($i+1$). Consider an atom $q(\mathbf{u}, \mathbf{v})$ in Q'_i other than $p(\mathbf{s}, \mathbf{t})$. By Lemma 7.2 (e), $vars(\mathbf{v}\theta'_i) \cap dom(\phi'_i) = \emptyset$. Thus for the atoms in Q'_{i+1} that occur already in Q'_i, S1 is maintained. Now consider an atom $q(\mathbf{u}, \mathbf{v})$ in B' which is not resolved in ξ. By Lemma 7.4, $\mathbf{v}\theta'_{i+1} = \mathbf{v}$. Since $q(\mathbf{u}, \mathbf{v})$ is not resolved in ξ, for all $j > l$ we have that $q(\mathbf{u}, \mathbf{v})$ occurs in Q_j and thus by Lemma 7.4, $\mathbf{v}\theta_j = \mathbf{v}$. Thus S1($i+1$) follows. S2($i+1$) holds because of S2($i$) and since $p(\mathbf{s}, \mathbf{t})$ is resolved using the same clause head as in ξ.

Case 2: $p(\mathbf{s}, \mathbf{t})$ is not resolved in ξ. Since P' is non-speculative, there is a (uniquely renamed) clause $C' = h \leftarrow B'$ in P' such that h and $p(\mathbf{s}, \mathbf{t})\theta'_i$ have an MGU ϕ'_i. We use C' for the step $\langle Q'_i, \theta'_i \rangle; \langle Q'_{i+1}, \theta'_{i+1} \rangle$.

We must show S1($i+1$) and S2($i+1$). Consider an atom $q(\mathbf{u},\mathbf{v})$ in Q'_i other than $p(\mathbf{s},\mathbf{t})$. By Lemma 7.2 (e), $vars(\mathbf{v}\theta'_i) \cap dom(\phi'_i) = \emptyset$. Thus for the atoms in Q'_{i+1} that occur already in Q'_i, S1 is maintained. Now consider an atom $q(\mathbf{u},\mathbf{v})$ in B'. Clearly $q(\mathbf{u},\mathbf{v})$ is not resolved in ξ, since it does not even *occur* in ξ. Since $vars(C') \cap vars(Q_j\theta_j) = \emptyset$ for all j and since by Lemma 7.4, we have $\mathbf{v}\theta'_{i+1} = \mathbf{v}$, S1($i+1$) follows.

By S1(i), we have $vars(\mathbf{t}\theta'_i) \cap dom(\theta_j) = \emptyset$ for all j. By Lemma 7.2 (c), we have $dom(\phi'_i) \subseteq vars(\mathbf{t}\theta'_i) \cup vars(C')$. Thus we have $dom(\phi'_i) \cap dom(\theta_j) = \emptyset$ for all j. Moreover, S2(i) holds, and so S2($i+1$) follows.

Since this construction can only terminate when the query is empty, either Q'_n is empty for some n, or ξ' is infinite.

Thus we show that if ξ' is finite, then every atom resolved in ξ is also resolved in ξ'. So let ξ' be finite of length n. Assume for the sake of deriving a contradiction that j is the smallest number such that the atom a selected in $\langle Q_j, \theta_j\rangle; \langle Q_{j+1}, \theta_{j+1}\rangle$ is never selected in ξ'. Then $j \neq 0$ since Q_0 and Q'_0 are permutations of each other and all atoms in Q'_0 are eventually selected in ξ'. Thus there must be a $k < j$ such that a does not occur in Q_k but does occur in Q_{k+1}. Consider the atom b selected in $\langle Q_k, \theta_k\rangle; \langle Q_{k+1}, \theta_{k+1}\rangle$. Then by the assumption that j was minimal, b must be the selected atom in $\langle Q'_i, \theta'_i\rangle; \langle Q'_{i+1}, \theta'_{i+1}\rangle$ for some $i \leq n$. Hence a must occur in Q'_{i+1}, since the clause used to resolve b in ξ' is a simply typed clause corresponding to the clause used to resolve b in ξ. Thus a must occur in Q'_n, contradicting that ξ' terminates with the empty query.

Thus if ξ' is finite, then ξ is also finite, or equivalently, if ξ is infinite, then ξ' is also infinite. □

As stated on page 99, for `permute`(I,O) (Figure 20 on page 99), `treeList`(I,O) (Figure 15 on page 88) and `plus_one`(I) (Figure 13 on page 81), the corresponding simply typed programs terminate for simply typed queries, assuming LD derivations. By Theorem 8.3 it follows that the former programs terminate for permutation simply typed queries, assuming delay-respecting derivations.[3]

All of these examples can also be shown to terminate using Chapter 6. We now give a program for which this is not the case.

Example 8.6 Consider the program in Figure 21, where the mode is $\{$`is_list`(I), `equal_list`$(I,O)\}$ and the type is $\{$`is_list`(list), `equal_list`$(\mathit{list},\mathit{list})\}$. The program is permutation simply typed (the second clause is $\langle 2,1\rangle$-simply typed) and non-speculative, and all LD-derivations for the corresponding simply typed program terminate. Hence it follows that all delay-respecting derivations of a permutation simply typed query and this program terminate. While we conjecture that `is_list` is also atom-terminating, the method of Chapter 6 cannot show this (compare this to the discussion about `quicksort`(I,O) on page 81).

This example is clearly a contrived one, which is partly because it has been designed to be as simple as possible. We are not aware of a more natural example, but this example suggests that the method presented in this subsection might be useful whenever the method of Chapter 6 fails to prove that a predicate is atom-terminating. ◁

[3] In the case of `plus_one`, we would have to add `block` declarations to ensure input selectability.

```
:- block is_list(-).               :- block equal_list(-,?).
is_list([]).                       equal_list([],[]).
is_list([X|Xs]):-                  equal_list([X|Xs],[X|Ys]):-
  is_list(Ys),                       equal_list(Xs,Ys).
  equal_list(Xs,Ys).
```

Figure 21: The `is_list` program

Note that any program that uses *tests* cannot be non-speculative. In the `quicksort` program (Figure 14 on page 87), the atoms `leq(X,C)` and `grt(X,C)` are tests. These tests are *exhaustive*, that is, at least one of them succeeds [BC99]. We are confident that the result of this subsection could be generalised to allow for such tests. We have not attempted this generalisation because on the whole, the method presented in the next section seems more useful. Pedreschi and Ruggieri however consider a more general notion of "non-failure", which allows for programs such as `quicksort` [PR99].

8.4 Termination and Atom-Terminating Predicates

We now present an alternative method for showing termination which overcomes some of the limitations of the methods presented in the previous section. In particular, the method can be used for `quicksort` (Figure 14 on page 87) and `nqueens` (Figure 22) as well as `permute` (Figure 20 on page 99) and `treeList` (Figure 15 on page 88). We expect the method presented here to be more useful, although, as Examples 8.4 and 8.6 show, it does not subsume the methods of the previous section.

In this section, two techniques are combined. On the one hand, we use Chapter 6 to show that certain predicates are atom-terminating. On the other hand, we reduce the problem of proving termination for a program *with* `block` declarations to the same problem for a corresponding program *without* `block` declarations, as in the previous section. It is assumed that termination for the corresponding program has been shown using some existing method for LD-derivations [DD94].

Let us now illustrate the limitations of the previous section. For `permute`(O, I) (Figure 20 on page 99), termination could be ensured by applying the heuristic of placing recursive calls last [Nai92]. The following example however shows that even this version of `permute`(O, I) can cause a loop depending on how it is called within some other program.

Example 8.7 Figure 22 shows a program for the n-queens problem. Here `block` declarations are used to implement the test-and-generate paradigm. We have already seen a fragment of this program in Figure 12 on page 80, however with a different order of atoms in the first clause.

Assuming mode {`nqueens`(I, O), `sequence`(I, O), `safe`(I), `permute`(O, I), $<(I, I)$, `is`(O, I), `safe_aux`(I, I, I), `no_diag`(I, I, I), `=\=`(I, I)} and type {`nqueens`(int, il), `sequence`(int, il), `safe`(il), `permute`(il, il), $<(int, int)$, `is`(int, int),

```
:- block nqueens(-,?).
nqueens(N,Sol) :-
  sequence(N,Seq),
  safe(Sol),
  permute(Sol,Seq).

:- block sequence(-,?).
sequence(0,[]).
sequence(N,[N|Seq]):-
  0 < N,
  N1 is N-1,
  sequence(N1,Seq).

:- block safe(-).
safe([]).
safe([N|Ns]) :-
  safe_aux(Ns,1,N),
  safe(Ns).
```

```
:- block safe_aux(-,?,?), safe_aux(?,-,?),
                          safe_aux(?,?,-).
safe_aux([],_,_).
safe_aux([M|Ms],Dist,N) :-
  no_diag(N,M,Dist),
  Dist2 is Dist+1,
  safe_aux(Ms,Dist2,N).

:- block no_diag(-,?,?), no_diag(?,-,?),
                         no_diag(?,?,-).
no_diag(N,M,Dist) :-
  Dist =\= N-M,
  Dist =\= M-N.

:- block permute(-,-).
permute([],[]).
permute([U|X],Y) :-
  delete(U,Y,Z),
  permute(X,Z).

:- block delete(?,-,-).
delete(X,[X|Z],Z).
delete(X,[U|Y],[U|Z]) :-
  delete(X,Y,Z).
```

Figure 22: A program for n-queens

safe_aux(*il*, *int*, *int*), no_diag(*int*, *int*, *int*), =\=(*int*, *int*)}, the first clause is $\langle 1, 3, 2\rangle$-robustly typed. Moreover, the query nqueens(4,Sol) terminates.

If however in the first clause, the atom order is changed by moving sequence(N,Seq) to the end, then nqueens(4,Sol) loops. This is because resolving sequence(4,Seq) with the second clause for sequence makes a binding (which is *not* speculative) which triggers the call permute(Sol,[4|T]). This call results in a loop since permute(O, I) is not atom-terminating. Note that [4|T], although non-variable, is insufficiently instantiated for permute(Sol,[4|T]) to be correctly typed in its input position: permute is called with *insufficient input.*

Note that in this example, unlike in the quicksort program (Figure 14 on page 87), there are no block declarations for the built-ins <, is and =/=. In Section 10.1, we will see why it is not necessary to have block declarations here. ◁

To ensure termination, atoms in a clause body that loop when called with insufficient input should be placed so that all atoms which produce the input for these atoms occur textually earlier. Note that this explains in particular why in the second clause for permute in the above example, the recursive call to permute must be placed last. In Chapter 6, we have seen that atom-terminating predicates do not loop for input-consuming derivations, which means in particular, they do not loop when called with insufficient input.

This section assumes permutation robustly typed programs. By Theorem 7.9, delay-respecting derivations for permutation robustly typed, input-linear programs with input selectability are input-consuming.

A query is called *well fed* if each atom is atom-terminating or occurs in such a position that all atoms which "feed" the atom occur earlier.

Definition 8.3 [well fed] Let P be a permutation robustly typed program. For a π-robustly typed query $p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$, an atom $p_i(\mathbf{s}_i, \mathbf{t}_i)$ is **well fed** if all predicates q with $p_i \sqsupseteq q$ are atom-terminating, or $\pi(j) < \pi(i)$ implies $j < i$ for all j. A π-robustly typed query (clause) is **well fed** if all of its (body) atoms are well fed. P is **well fed** if all of its clauses are well fed. ◁

Of course, since it is undecidable whether a predicate is atom-terminating, we must assume it to be not atom-terminating if it has not been shown to be atom-terminating. In Example 6.5, we have seen the situation that a predicate p is atom-terminating but some predicate q with $p \sqsupset q$ is not atom-terminating. To simplify the proof of Theorem 8.5, we want to exclude this pathological situation. This is reflected in the above definition by the requirement "all predicates q with $p_i \sqsupseteq q$ are atom-terminating", rather than just "p_i is atom-terminating".

Example 8.8 The programs mentioned in Example 7.6 are well fed in the given modes. The `nqueens` program (Figure 22 on page 106) is well fed in the mode given in Example 8.7. The program is not well fed in mode {`nqueens`(O, I), `sequence`(O, I), `safe`(I), `permute`(I, O), `<`(I, I), `is`(O, I), `safe_aux`(I, I, I), `no_diag`(I, I, I), `=\=`(I, I)}, because it is not permutation nicely moded in this mode: in the second clause for `sequence`, `N1` occurs twice in an output position. ◁

The property of being well fed is persistent under resolution.

Lemma 8.4 Every resolvent of a well fed query Q and an input-linear well fed clause C, where $vars(Q) \cap vars(C) = \emptyset$ and the derivation step is input-consuming, is well fed.

PROOF. By Lemma 7.6 (b), the resolvent is permutation robustly typed. The condition on the permutation in Definition 8.3 can be checked by inspecting Definition 5.1. □

The following theorem reduces the problem of showing termination of left-based derivations for a well fed program to showing termination of LD-derivations for a corresponding robustly typed program.

Theorem 8.5 Let P be an input-linear, well fed program with input selectability, and Q a well fed query. Let P' and Q' be a robustly typed program and query corresponding to P and Q, respectively. If every LD-derivation of $P' \cup \{Q'\}$ is finite, then every left-based derivation of $P \cup \{Q\}$ is finite.

PROOF. In this proof, call an atom $p(\mathbf{s}, \mathbf{t})$ **critical** if it is not the case that all predicates q with $p \sqsupseteq q$ are atom-terminating. Let $Q_0 = Q$, $\theta_0 = \emptyset$ and

$$\xi = \langle Q_0, \theta_0 \rangle; \ldots; \langle R_1, \sigma_1 \rangle; \langle Q_1, \theta_1 \rangle; \ldots; \langle R_2, \sigma_2 \rangle; \langle Q_2, \theta_2 \rangle \ldots$$

be a left-based derivation, where $R_1, R_2, \ldots$ are the queries in ξ where a critical atom is selected.

PART 1: We show for each $i \geq 0$: If R_i exists, then in each query in $\langle Q_0, \theta_0 \rangle; \ldots; \langle R_i, \sigma_i \rangle$, the critical atoms are not waiting, and for each $l \leq i$, the leftmost critical atom in R_l is selected in the step $\langle R_l, \sigma_l \rangle; \langle Q_l, \theta_l \rangle$. The proof is by induction on i.

CASE 1: *Base case.* The case $i = 0$ is trivial since R_0 does not exist.

CASE 2: *Inductive step.* Suppose the statement holds for some $i \geq 0$.

CASE 2a: If R_{i+1} does not exist, the statement follows trivially for $i + 1$.

CASE 2b: Now suppose that R_{i+1} exists. Let $Q_i = a_1, \ldots, a_n$ and suppose $Q_i\theta_i$ is π-robustly typed, and k is the smallest number such that a_k is critical.

Let (F, a_k) be the subquery of Q_i containing all a_j with $\pi(j) \leq \pi(k)$. By Lemma 8.4, $Q_i\theta_i$ is well fed, and thus $j \leq k$ for all a_j in (F, a_k). By Proposition 5.11

$$(F, a_k)\theta_i \text{ is permutation well typed.} \tag{1}$$

Consider an arbitrary $\langle \tilde{Q}, \tilde{\theta} \rangle$ in $\langle Q_i, \theta_i \rangle; \ldots; \langle R_{i+1}, \sigma_{i+1} \rangle$ and assume that no critical atom in the query preceding $\langle \tilde{Q}, \tilde{\theta} \rangle$ in ξ is waiting. Note that since $\tilde{Q}$ contains a_k, it follows that $\tilde{Q}$ contains at least one descendant of (F, a_k). By (1) and Lemma 5.10, $\tilde{Q}$ contains, in particular, at least one descendant a of (F, a_k) such that $a\tilde{\theta}$ is *selectable*, and moreover, either $a = a_k$ or a occurs to the left of a_k in $\tilde{Q}$. Therefore no critical atom in $\langle \tilde{Q}, \tilde{\theta} \rangle$ is waiting.

Suppose that $R_{i+1}\sigma_{i+1}$ contains a descendant a of (F, a_k) such that $a\sigma_{i+1}$ is selectable, and $a \neq a_k$. Then, since by the previous paragraph, a_k is not waiting in R_{i+1}, it follows that a_k cannot be selected in $\langle R_{i+1}, \sigma_{i+1} \rangle; \langle Q_{i+1}, \theta_{i+1} \rangle$, which contradicts the definition of R_{i+1}. Thus it follows that

$$R_{i+1} \text{ contains no descendant of } F, \tag{2}$$

and so $a_k\sigma_{i+1}$ is selectable. Moreover, no critical atom in $R_{i+1}\sigma_{i+1}$ is waiting, and so the selected atom in $\langle R_{i+1}, \sigma_{i+1} \rangle; \langle Q_{i+1}, \theta_{i+1} \rangle$ is a_k.

PART 2: For all $i > 0$ such that R_i exists, let C_i be the uniquely renamed clause used in the step $\langle R_i, \sigma_i \rangle; \langle Q_i, \theta_i \rangle$, and let C'_i be a robustly typed clause corresponding to C_i (using the same renaming). Let $Q'_0 = Q'$ and $\theta'_0 = \emptyset$. We construct an LD-derivation

$$\xi' = \langle Q'_0, \theta'_0 \rangle; \ldots; \langle R'_1, \sigma'_1 \rangle; \langle Q'_1, \theta'_1 \rangle; \ldots; \langle R'_2, \sigma'_2 \rangle; \langle Q'_2, \theta'_2 \rangle \ldots,$$

where $R'_1, R'_2, \ldots$ are the queries in ξ' where a critical atom is selected, such that for all $i > 1$, C'_i is the clause used in $\langle R'_i, \sigma'_i \rangle; \langle Q'_i, \theta'_i \rangle$. Since ξ' is finite by assumption, this implies that ξ is finite. We show the following statements for all $i \geq 0$ such that Q_i exists:

S1(i) The critical atoms of Q_i and Q'_i are identical and occur in the same order.

S2(i) $\theta_i = \theta'_i\rho_i$ for some substitution ρ_i.

S3(i) Let $Q_i = a_1, \dots, a_n$ and assume that $Q_i\theta_i$ is π-robustly typed, and let a_k be a critical atom ($k \in \{1, \dots, n\}$). By S1(i) we can write $Q'_i = (F', a_k, I')$ for some F' and I'. For every a in F', for every a_j ($j \in \{1, \dots, n\}$) that is a descendant of a in ξ, we have $\pi(j) < \pi(k)$.

The proof is by induction on i.

Case 1: *Base case.* S1(0) follows from Definition 8.3. S2(0) holds since $\theta_0 = \theta'_0 = \emptyset$. For S3(0), note that $Q'_0 = \pi(Q_0)$ and hence F' contains exactly the atoms a_j with $\pi(j) < \pi(k)$.

Case 2: We now assume that S1(i)–S3(i) hold for some $i \geq 0$ and that Q_{i+1} exists, and construct

$$\langle Q'_i, \theta'_i\rangle; \dots; \langle R'_{i+1}, \sigma'_{i+1}\rangle; \langle Q'_{i+1}, \theta'_{i+1}\rangle$$

so that S1($i+1$)–S3($i+1$) hold.

As in Part 1, let $Q_i = a_1, \dots, a_n$, suppose that $Q_i\theta_i$ is π-robustly typed, let k be the smallest number such that a_k is critical, and (F, a_k) be the subquery of Q_i containing all a_j with $\pi(j) \leq \pi(k)$. By S1(i), $Q'_i = (F', a_k, I')$ for some F' and I', where F' contains only atom-terminating atoms. By S3(i), for every a in F', for every a_j ($j \in \{1, \dots, n\}$) that is a descendant of a in ξ, we have $\pi(j) < \pi(k)$, and therefore a_j is in F. Thus it follows by (2) in Part 1 that

$$R_{i+1} \text{ contains no descendants of } F'. \tag{3}$$

Let $R'_{i+1} = a_k, I'$. By (3) and since by S2(i), θ'_i is more general than θ_i, it is possible to construct an LD-derivation $\langle Q'_i, \theta'_i\rangle; \dots; \langle R'_{i+1}, \sigma'_{i+1}\rangle$, such that if $\tilde{C}$ is the uniquely renamed clause used to resolve an atom in ξ, then a robustly typed clause $\tilde{C}'$ corresponding to $\tilde{C}$ (using the same renaming) is used in $\langle Q'_i, \theta'_i\rangle; \dots; \langle R'_{i+1}, \sigma'_{i+1}\rangle$. Furthermore σ'_{i+1} is more general than σ_{i+1}. Hence C'_{i+1} can be used in $\langle R'_{i+1}, \sigma'_{i+1}\rangle; \langle Q'_{i+1}, \theta'_{i+1}\rangle$.

Since in the clause body of C'_{i+1}, the critical atoms occur in the same order as in C_{i+1}, S1($i+1$) holds. Since σ'_{i+1} is more general than σ_{i+1}, it follows that θ'_{i+1} is more general than θ_{i+1}, so S2($i+1$) holds. For the critical atoms in Q_{i+1} which occur in the clause body of C'_{i+1}, S3($i+1$) follows from Definition 8.3. For the critical atoms in Q_{i+1} which occur already in R_{i+1}, S3($i+1$) follows from S3(i).

By Definition 7.4, Q is permutation well typed, type-consistent and permutation nicely moded. By Lemma 7.7, P is type-consistent with respect to input-consuming derivations. By Theorem 7.9, ξ is input-consuming. Hence by Theorem 6.3, ξ could be infinite only if there are infinitely many steps where a critical atom is resolved.[4] Since ξ' is finite, ξ cannot have infinitely many steps where a critical atom is resolved, and thus ξ is finite. □

[4] Recall that as discussed on page 63, Theorem 6.3 generalises to *permutation* well typed and *permutation* nicely moded programs and queries.

Example 8.9 Consider the `quicksort` program (Figure 14 on page 87) with the type given in Example 7.4. As stated in Example 8.8, this program is well fed in mode {`quicksort`(I, O), `append`(I, I, O), `leq`(I, I), `grt`(I, I), `part`(I, I, O, O)}. In particular, the `append` atom in the body of the recursive clause for `quicksort` is well fed since it is atom-terminating (see Example 6.4). All other body atoms in the program are well fed because of their textual position.

As stated on page 99, the robustly typed program corresponding to this program terminates for all robustly typed queries, assuming LD-derivations. By Theorem 8.5 it follows that the `quicksort` program of Figure 14 terminates for all well fed queries, assuming left-based derivations.

Now consider the mode {`quicksort`(O, I), `append`(O, O, I), `leq`(I, I), `grt`(I, I), `part`(O, I, I, I)}. The `quicksort` program is also well fed with respect to this mode. The two recursive calls in the second clause for `quicksort` are well fed because of their textual position. All other atoms are well fed because they are atom-terminating. For `part`, this can be shown using Theorem 6.4, where the level mapping of an atom `part`(l, c, s, b) is defined as the sum of the list lengths of s and b. As for the first mode, we can conclude that the program terminates for all well fed queries, assuming left-based derivations. ◁

Example 8.10 Consider the `nqueens` program (Figure 22 on page 106). We have seen in Example 6.4 that `no_diag`, `safe_aux` and `safe` are atom-terminating.

The clause defining `nqueens` is $\langle 1, 3, 2\rangle$-robustly typed. The second atom is well fed since it is atom-terminating. The first atom is well fed since for $\pi = \langle 1, 3, 2\rangle$, $\pi(j) < \pi(1)$ implies $j < 1$ for all j. The third atom is well fed since $\pi(j) < \pi(3)$ implies $j < 3$ for all j.

As stated on page 99, the robustly typed program corresponding to this program terminates for all robustly typed queries, assuming LD-derivations. By Theorem 8.5 it follows that the `nqueens` program of Figure 22 terminates for all well fed queries, assuming left-based derivations.

According to the producer-consumer order, `safe(Sol)` occurs textually too early. However, this is the idea of the test-and-generate paradigm: the test `safe(Sol)` comes before the generator `permute(Sol,Seq)`. This way, `safe(Sol)` is always selected as early as possible and therefore “non-solutions” to the n-queens problem are detected early.

Our method can only show termination for the mode given in Example 8.7, but not for the mode `nqueens`(O, I), although the program actually terminates for that mode (provided the `block` declarations are modified to allow for both modes). The reason that our method fails is not some insignificant detail of our definitions that could easily be rectified. One can definitely say that the modes in this program “go wrong”: every call to `sequence`(O, I) triggers calls to `sequence`(I, O). The consequence is that `nqueens`(O, I) runs in exponential time although it could run in quadratic time.

To the best of our knowledge, no method previously proposed can prove termination for this program, which is a classical example of a program using coroutining. ◁

Similarly, we can show termination for `permute` (Figure 20 on page 99) and `treeList` (Figure 15 on page 88). We are assuming here that all built-ins have input selectability.

Built-ins will be discussed in Section 9.4. In Section 10.1, we will see why in some cases, it is not necessary to have `block` declarations for the built-ins.

8.5 Discussion

In this chapter, we have presented two approaches to proving termination for programs with `block` declarations.

The first approach is focused on *speculative output bindings*, which have long been recognised as a source of non-termination in programs with delay declarations [Nai92]. The approach consists of two complementing methods based on not *using* and not *making* speculative bindings, respectively. For `permute` (Figure 20 on page 99) and `treeList` (Figure 15 on page 88), it turns out that in one mode, the first method applies, and in the other mode, the second method applies. This approach is simple to understand and to apply, and it represents the first work on termination we have published [SHK99b].

The second approach builds on Chapter 6. We require programs to be *permutation robustly typed*, a property which ensures that derivations are input-consuming. In the next step, we identify predicates that are *atom-terminating*. Atom-terminating atoms can be placed in clause bodies anywhere. The other atoms must be placed sufficiently late, so that their input is sufficiently instantiated when they are called. Provided that the corresponding robustly typed program terminates for all LD-derivations, this then implies that the original program terminates for all left-based derivations.

On the whole, the second approach is more useful. It can be used to show termination for `quicksort` (Figure 14 on page 87) and `nqueens` (Figure 22 on page 106), where the first approach fails. In the original paper where this approach was first presented [SHK98], it was not yet based on the results of Chapter 6 in their present general form. In this thesis, the approach follows the idea that one should abstract from the details of particular delay constructs wherever possible, and instead consider input-consuming derivations.

On the other hand, as Examples 8.4 and 8.6 show, the second approach does not formally subsume the first. Example 8.6 suggests in particular that the method of Subsection 8.3.2 might be useful whenever the method of Chapter 6 fails to prove that a predicate is atom-terminating, although it actually is. Of course, it would ultimately be desirable to have a more powerful method for proving that a predicate is atom-terminating, but we consider this to be a difficult problem.

Chapter 9

Further Aspects of Verification

So far, we have studied *termination* of non-standard derivations. Following work by Apt and others [AE93, AL95], we now investigate four other aspects of verification: programs should only require matching instead of the full unification procedure wherever possible; the omission of the occur-check should be safe; programs should not flounder; and there should be no type or instantiation errors with the use of built-ins.

Our results on unification freedom, occur-check freedom and flounder freedom are generalisations of previous work [AE93, AL95]. Our work on built-ins is aimed mainly at *arithmetic* built-ins. We exploit the fact that for numbers, being non-variable implies being ground, and show how to prevent *instantiation* and *type* errors.

This chapter is organised as follows. Section 9.1 shows when programs are unification free. Section 9.2 shows when the occur-check can safely be omitted. Section 9.3 shows when programs do not flounder. Section 9.4 is about errors related to built-ins. Section 9.5 concludes.

9.1 Unification Free Programs

A program is *unification free* if unification can be replaced by matching. Knowing that a program has this property can improve the efficiency of the compiled code. Apt and Etalle [AE93] show unification freedom for LD-derivations. They assume simply moded and well typed programs and rely on the selected atom always being correctly typed in its input positions.

When we generalise these results to arbitrary input-consuming derivations, we must take into account that the selected atom may not be sufficiently instantiated to be correctly typed in its input positions. Nevertheless, we will now see that permutation simply typed programs are unification free. We first recall some definitions [AE93].

Definition 9.1 [match, left-right disjoint] Given two vectors of terms $\mathbf{s} = s_1, \ldots, s_n$ and $\mathbf{t} = t_1, \ldots, t_n$ we use $\{\mathbf{s} = \mathbf{t}\}$ as abbreviation for the set of equations $\{s_1 = t_1, \ldots, s_n = t_n\}$. Consider a set of equations $E = \{\mathbf{s} = \mathbf{t}\}$. A substitution θ such that $dom(\theta) \subseteq vars(\mathbf{s})$ and $\mathbf{s}\theta = \mathbf{t}$, or $dom(\theta) \subseteq vars(\mathbf{t})$ and $\mathbf{t}\theta = \mathbf{s}$, is a **match** for E. Furthermore, E is **left-right disjoint** if $vars(\mathbf{s}) \cap vars(\mathbf{t}) = \emptyset$. $\triangleleft$

The following is a special case of *iterated matching* [AE93].

Definition 9.2 [double matching] Let E be a left-right disjoint set of equations. E is **solvable by double matching** if the following holds: if E is unifiable, then there are sets of equations E_1 and E_2 and substitutions θ_1 and θ_2 such that

- $E = E_1 \cup E_2$,
- $E_2\theta_1$ is left-right disjoint, and
- θ_1 is a match for E_1 and θ_2 is a match for $E_2\theta_1$.

$\triangleleft$

We now define programs that are unification free *for input-consuming derivations*, as opposed to LD-derivations as assumed by Apt and Etalle [AE93].

Definition 9.3 [unification free for input-consuming derivations] Let ξ be a derivation. Let $p(\mathbf{s})$[1] be a selected atom in ξ and $p(\mathbf{t})$ the head of the clause used to resolve $p(\mathbf{s})$. Then the set of equations $\mathbf{s} = \mathbf{t}$ is **successfully considered in** ξ.

Let P be a program and Q a query. Suppose that all sets of equations successfully considered in all input-consuming derivations of $P \cup \{Q\}$ are solvable by double matching. Then $P \cup \{Q\}$ is **unification free for input-consuming derivations**. $\triangleleft$

Note that unlike Apt and Etalle, we say that a set of equations is *successfully* considered, rather than just *considered.* This is because an atom can only be resolved if the unification with the clause head is successful. In our notion of derivation, there is no such thing as "trying" to unify an atom with a clause head unsuccessfully.

In the sequel, since we only consider input-consuming derivations, we will simply say "unification free" instead of "unification free for input-consuming derivations".

Apt and Etalle [AE93] exploit the fact that many programs have *generic expressions* in their input positions. A generic expression for a type T is a term t such that if s is a term of type T and s is unifiable with t, then s is an instance of t. In a permutation simply typed program, the input positions of each clause head are filled with generic expressions, since they are filled with variables in positions of variable type and flat type-consistent terms in positions of non-variable type.

[1] Note that $\mathbf{s}$ is a vector of terms. We do not care about input or output positions at this point.

Theorem 9.1 Let P be a permutation simply typed, input-linear program and Q a permutation simply typed query. Then $P \cup \{Q\}$ is unification free.

Proof. Consider a derivation step $R; R'$ in an input-consuming derivation of $P \cup \{Q\}$, where $p(\mathbf{s}, \mathbf{t})$ is the selected atom, $p(\mathbf{v}, \mathbf{u})$ is the head of the clause used in this step and θ is the MGU. By Lemma 7.2 (f), R is permutation simply typed. Let $E_1 = \{\mathbf{s} = \mathbf{v}\}$ and $E_2 = \{\mathbf{t} = \mathbf{u}\}$ so that $E_1 \cup E_2$ is the set of equations successfully considered at this step. By Lemma 7.2 (a, b), $\theta = \theta_1\theta_2$ where θ_1 is a match for E_1, $dom(\theta_1) \subseteq vars(\mathbf{v})$, $vars(ran(\theta_1)) \subseteq vars(\mathbf{s})$ and θ_2 is a match for $\{\mathbf{t} = \mathbf{u}\theta_1\}$. Since $dom(\theta_1) \subseteq vars(\mathbf{v})$ and $vars(\mathbf{v}) \cap vars(\mathbf{t}) = \emptyset$, we have $E_2\theta_1 = \{\mathbf{t} = \mathbf{u}\theta_1\}$. Therefore θ_2 is a match for $E_2\theta_1$. Since R is permutation simply typed, $vars(\mathbf{s}) \cap vars(\mathbf{t}) = \emptyset$ so that $E_2\theta_1$ is left-right disjoint. Therefore $E_1 \cup E_2$ is solvable by double matching and hence $P \cup \{Q\}$ is unification free. □

Most programs we have seen are permutation simply typed and input-linear, and hence unification free. However, quicksort(O, I) (Figure 14 on page 87) and treeList(O, I) (Figure 15 on page 88) are not permutation simply typed. The following example illustrates why the reasoning of the above theorem does not work for those programs, even though they may well be unification free. This difficulty has been acknowledged previously by Apt and Etalle [AE93].

Example 9.1 Consider the following two derivations for treeList(O, I) (Figure 15 on page 88). Here the first clause for append is used:

$$\begin{array}{l}
\underline{\texttt{treeList(A,[1])}} \leadsto \\
\underline{\texttt{append(LList,[Label|RList],[1])}},\ \texttt{treeList(L,LList)},\ \texttt{treeList(R,RList)} \leadsto \\
\texttt{treeList(L,[])},\ \texttt{treeList(R,[])}
\end{array}$$

and here the second clause is used:

$$\begin{array}{l}
\underline{\texttt{treeList(A,[1])}} \leadsto \\
\underline{\texttt{append(LList,[Label|RList],[1])}},\ \texttt{treeList(L,LList)},\ \texttt{treeList(R,RList)} \leadsto \\
\texttt{append(Xs,[Label|RList],[])},\ \texttt{treeList(L,[1|Xs])},\ \texttt{treeList(R,RList)}.
\end{array}$$

In both derivations, the last step is solvable by double matching. In the first case, the partitioning of the set of equations is

$$E_1 = \{[\texttt{1}] = \texttt{Y}\}, \quad \mathtt{E}_2 = \{[\texttt{Label|RList}] = \texttt{Y},\ \texttt{LList} = []\}.$$

In the second case, it is

$$E_1 = \{[\texttt{1}] = [\texttt{X|Zs}],\ [\texttt{Label|RList}] = \texttt{Ys}\}, \quad \mathtt{E}_2 = \{\texttt{LList} = [\texttt{X|Xs}]\}.$$

Note that the second argument position of append is in a different set of the partition depending on the clause which is used. It is not possible to fix a partitioning into the input and output positions, which is the idea underlying Theorem 9.1. ◁

9.2 Occur-Check Freedom

A derivation is *occur-check free* if for every set of equations considered in this derivation, the occur-check can safely be omitted. We must first define what it means for a set of equations to be *considered.* This builds on Definition 9.3. The concept has been previously defined by Apt and Luitjes [AL95]. However, their definition is imprecise in that it depends on a concept of a derivation which may end with a failed attempt to unify a selected atom with a clause, without actually defining this concept formally.

Definition 9.4 [considered] Let P be a program and ξ a derivation. A set of equations $\mathbf{s} = \mathbf{t}$ is **considered in** ξ if it is either successfully considered in ξ, or there is an atom $p(\mathbf{s})$ in the last query of ξ and a clause in P whose head is $p(\mathbf{t})$. $\triangleleft$

In the above definition, no assumptions are made about the degree of instantiation of the "selected atom" $p(\mathbf{s})$. This is because our result on occur-check freedom holds for arbitrary derivations. It would of course be possible to take into account that ξ is say, delay-respecting or left-based, and impose a restriction such as "$p(\mathbf{s})$ must be selectable". It would however not be meaningful to take into account that ξ is *input-consuming.* We illustrate this with an example.

Example 9.2 Consider the program

```
p(A,B).
p(A,A).
```

where the mode is $\mathtt{p}(I, I)$, and consider the query $\mathtt{p}(\mathtt{X}, \mathtt{f}(\mathtt{X}))$. Suppose we require that derivations are input-consuming. Then we can perform a derivation step using the first clause. We cannot perform a derivation step using the second clause, because $\mathtt{p}(\mathtt{X}, \mathtt{f}(\mathtt{X}))$ and $\mathtt{p}(\mathtt{A}, \mathtt{A})$ are not unifiable. It is therefore meaningless to reason about whether this derivation step would have been input-consuming. The notion of *input-consuming* is only meaningful for *actual* derivation steps, not for attempted ones. $\triangleleft$

Definition 9.5 [occur-check free] A derivation is **occur-check free** [AL95, AP94b] if no execution of the Martelli-Montanari unification algorithm [MM82] for a set of equations considered in this derivation ends with a set of equations including an equation $x = t$, where x is not t, but x occurs in t. $\triangleleft$

We quote the following theorem.

Theorem 9.2 [AL95, Theorem 13] Let P be a nicely moded, input-linear program and Q a nicely moded query. Then all derivations of $P \cup \{Q\}$ are occur-check free.

The next theorem is a trivial consequence of this and Lemma 5.3.

Theorem 9.3 [occur check] Let P be a permutation nicely moded, input-linear program and Q a permutation nicely moded query. Then all derivations of $P \cup \{Q\}$ are occur-check free.

Most programs considered in this thesis are permutation nicely moded and input-linear, and hence occur-check free.

9.3 Floundering

Freedom from floundering is an important aspect of verification mainly because of its relationship to termination. As Apt and Luitjes [AL95] put it

> [...] the "stronger" the delay declarations are the bigger the chance that a deadlock arises, but the smaller the chance that divergence [non-termination] can result. So deadlock freedom and termination seem to form two boundaries within which lie the "correct" delay declarations.

In other words, one can always trivially ensure termination by having delay declarations such that no atom is ever selectable. That way, every derivation immediately flounders and hence terminates. Likewise, one can trivially ensure non-floundering by declaring that every atom is always selectable.[2] That way, no derivation can ever flounder but possibly at the cost of non-termination.

Therefore, for every approach to the termination problem of programs with delay declarations, one must ask critically: Does the method "buy" termination with floundering? For the automatically generated delay declarations of Lüttringhaus-Kappel [Lüt93], the answer could sometimes be "yes". This is discussed in Subsection 11.1.5.

Compared to termination however, non-floundering is an easy problem. Under the reasonable assumption that programs and queries are permutation well typed, it can be shown that no derivation flounders. The assumption is reasonable because most programs are permutation well typed.[3] On the other hand, it is usually unreasonable to expect non-floundering for a query that is not instantiated enough to be permutation well typed. We have argued in Subsection 1.2.2 that ensuring input-consuming derivations is paramount. Usually, floundering is the only way to ensure this for insufficiently instantiated queries. As an example, consider the query $\texttt{append}([\texttt{1}|\texttt{Xs}], [], \texttt{Zs})$ (see Figure 10 on page 57).

The following theorem generalises [AL95, Theorem 26] to *permutation* well typed programs. Note that permutation robustly typed programs with input selectability (Definition 7.5) fulfill the condition that an atom is selectable if it is non-variable in all input positions of non-variable type.

Theorem 9.4 Let P be a permutation well typed program and Q be a permutation well typed query. Assume that an atom is selectable if it is non-variable in all input positions of non-variable type. Then no delay-respecting derivation of $P \cup \{Q\}$ flounders.

Proof. Let $Q_0 = Q$ and $\xi = Q_0; Q_1; \ldots$ be a delay-respecting derivation of $P \cup \{Q\}$. Consider an arbitrary $Q_i = a_1, \ldots, a_n$ where $n \geq 1$. By Lemma 5.10, Q_i is π-well typed for some π. By Definition 5.5, the atom $a_{\pi^{-1}(1)}$ is correctly typed in its input positions, and thus non-variable in its input positions of non-variable type. Therefore $a_{\pi^{-1}(1)}$ is selectable. Thus every non-empty query in ξ contains a selectable atom, and so ξ does not flounder. □

[2] Technically, this is achieved simply by having no delay declarations at all.

[3] Etalle and others [AE93, EBC99] even claim that most programs are well typed and simply moded.

The above theorem can be used to show freedom from floundering for all programs with `block` declarations we have introduced.

9.4 Errors Related to Built-ins

Built-in predicates (*built-ins*) can be a source of execution errors. Some built-ins produce an error if certain arguments have a wrong type or are insufficiently instantiated. For example, `X is foo` results in a type error and `X is V` results in an instantiation error.

Not surprisingly, delay declarations are useful to prevent instantiation errors, since they test for sufficient instantiation. The relationship between delay declarations and *type* errors will be explained in the next subsection.

One problem with built-ins is that their implementation may not be written in Prolog, or whatever logic programming language we consider. Thus we assume that each built-in is conceptually defined by possibly infinitely many (fact) clauses. The ISO standard for Prolog [ISO95] does not define the built-in predicates as conceptual clauses, but it is nevertheless so precise that it should generally be possible to verify whether such a definition is correct.

To prove that a program is free from errors related to built-ins, we require it to meet certain correctness properties (see Section 7.5). These properties have to be satisfied by the conceptual clauses for the built-ins as well as by the user-defined clauses.

For example, there could be facts "`0 is 0+0.`", "`1 is 0+1.`", and so forth. A particularly interesting example is "`X = X.`" which is the definition of the built-in `=`. This is why in an input-linear program, the mode `=`(I, I) is forbidden, since the clause is not input-linear for that mode.

In this section, we first explain why type errors are related to delay declarations. We then present two approaches to ensuring freedom from instantiation and type errors for programs with delay declarations. For different programs and built-ins, different approaches may be applicable.

9.4.1 The Connection between Delay Declarations and Type Errors

At first sight, it seems that delay declarations, or more generally, non-standard selection rules, do not affect the problem of type errors, be it positively or negatively. Delay declarations cannot enforce arguments to be correctly typed. Also, one would not expect that a non-standard selection rule could be the *cause* of wrongly typed arguments.

This is probably true in practice, but in theory, there is the problem of *type consistency*, which is particularly relevant for non-standard derivations (see Section 5.7). Consider the program consisting of the fact clause "`two(2).`" and the built-in `is`, with type $\{\texttt{two}(int), \texttt{is}(int, int)\}$ and mode $\{\texttt{two}(O), \texttt{is}(O, I)\}$. Suppose an atom using `is` is selectable only when its input is non-variable. The query

$$\texttt{X is foo, two(foo)}$$

is $\langle 2, 1\rangle$-well typed since trivially $\models \texttt{foo} : int \Rightarrow \texttt{foo} : int$. It results in a type error.

For LD-derivations this problem does not arise. The well typed query corresponding to the above query is `two(foo), X is foo`. Since the type of `two` is *int* and the program is well typed, the atom `two(foo)` can never be resolved, and therefore the derivation fails without ever reaching `X is foo`.

9.4.2 Exploiting Constant Types

The approach described in this subsection aims at preventing instantiation and type errors for built-ins, for example arithmetic built-ins, that require arguments to be ground. It has been proposed by Apt and Luitjes [AL95] to equip these predicates with delay declarations so that they are only executed when the input is ground. This has the advantage that one can reason about arbitrary arithmetic expressions, as in, say, `quicksort([1+1,3-8],M)`. The disadvantage is that `block` declarations cannot be used. In contrast, we assume that the type of arithmetic built-ins is the constant type *num*. Then we show that `block` declarations are sufficient. The following lemma is similar to and based on [AL95, Lemma 27].

Lemma 9.5 Let $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be a π-well typed query, where $p_i(\mathbf{S}_i, \mathbf{T}_i)$ is the type of p_i for each $i \in \{1, \dots, n\}$. Suppose, for some $k \in \{1, \dots, n\}$, that $\mathbf{s}_k$ has a non-variable term s occurring directly in a position of constant type S, and there is a substitution θ such that $\mathbf{t}_j\theta : \mathbf{T}_j$ for all j with $\pi(j) < \pi(k)$. Then $s : S$ (and thus s is ground).

Proof. By Definition 5.5, $\mathbf{s}_k\theta : \mathbf{S}_k$, and thus $s\theta : S$ and so $s\theta$ is a constant. Since s is already non-variable, it follows that s is a constant and thus $s\theta = s$. Therefore $s : S$. □

By Definition 7.2, for every permutation simply typed query Q, there is a θ such that $Q\theta$ is correctly typed in its output positions. Thus by Lemma 9.5, if the arithmetic built-ins have type *num* in all input positions, then it is enough to have `block` declarations such that these built-ins are only selected when the input positions are non-variable.

Note that in the following theorem, we do not mention instantiation or type errors, as we have not defined formally what an error “is”. From a formal point of view, all that matters is that an atom selected when its input arguments are correctly typed does not produce an error.

Theorem 9.6 Let P be a permutation simply typed, input-linear program with input selectability and Q be a permutation simply typed query. Then in any delay-respecting derivation ξ of $P \cup \{Q\}$, an atom will be selected only when it is correctly typed in its input positions of constant type.

Proof. By Lemma 7.2 (f) and Theorem 7.9, ξ consists of permutation simply typed queries. The result thus follows from Lemma 9.5. □

Example 9.3 Consider `quicksort`(I, O) (Figure 14 on page 87) with the type given in Example 7.4. No delay-respecting derivation for a permutation simply typed query and this program can result in an instantiation or type error related to the arithmetic built-ins. ◁

```
:- block length(-,-).           :- block len_aux(?,-,?), len_aux(-,?,-).
length(L,N) :-                  len_aux([],N,N).
  len_aux(L,0,N).               len_aux([_|Xs],M,N) :-
                                  less(M,N),
:- block less(?,-), less(-,?).    M2 is M + 1,
less(A,B) :-                      len_aux(Xs,M2,N).
  A < B.
```

Figure 23: The `length` program

9.4.3 Atomic Positions

Sometimes, when the above method does not work because a program is not permutation simply typed, it is still possible to show absence of instantiation errors for arithmetic built-ins. We observe that these built-ins have argument positions of type *num* or *int* which are constant types. Thus, the idea is to declare certain argument positions in a predicate, including the above argument positions of the built-ins, to be *atomic*. This means that they can only be ground or free but not partially instantiated. Then there need to be `block` declarations such that an atom is only selected when the arguments in these positions are non-variable, and hence ground. Just as with types and modes, we assume that the positions which are atomic are already known.

Definition 9.6 [respects atomic positions] A query (clause) **respects atomic positions** if each term in an atomic position is ground or a variable which *only* occurs in atomic positions. A program respects atomic positions if each of its clauses does. $\lhd$

A program need not be permutation nicely moded or permutation well typed in order to respect atomic positions.

Example 9.4 The program in Figure 23 computes the length of a list. In this example, we are regarding the atom `M2 is M + 1` as an atom with three arguments `M2`, `M`, and `1`. The program then respects atomic positions, assuming that all argument positions are atomic, except the first argument position of `length` and `len_aux`, respectively. The `block` declaration on the built-in `<` is realised with an auxiliary predicate `less`. $\lhd$

The property of respecting atomic positions is persistent under resolution.

Lemma 9.7 Let C be a clause and Q a query which respect atomic positions, where $vars(C) \cap vars(Q) = \emptyset$. Then a resolvent of C and Q also respects atomic positions.

Proof. Let $Q = a_1, \ldots, a_n$ be the query and $C = h \leftarrow b_1, \ldots, b_m$ be the clause. Let a_k be the selected atom and assume it is unifiable with h using MGU θ. We must show that

$$Q' = (a_1, \ldots, a_{k-1}, b_1, \ldots, b_m, a_{k+1}, \ldots, a_n)\theta$$

respects atomic positions.

Let x be a variable which fills an atomic position in a_k or h. Since Q and C respect atomic positions, $x\theta$ is either a variable which only occurs in atomic positions in Q', or a ground term.

Consider a term s filling an atomic position in $a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n$ or $b_1, \dots, b_m$. If s is a ground term, then $s\theta$ is also a ground term. Suppose that s is a variable. If $s \notin dom(\theta)$, then $s\theta$ is also a variable. If $s \in dom(\theta)$ then s must fill an atomic position in a_k or h. By the previous paragraph, $s\theta$ is either a variable which only occurs in atomic positions in Q', or a ground term. □

By the following theorem, instantiation errors can be prevented by having `block` declarations such that an atom using a built-in is only called when it is non-variable in its atomic positions. The theorem is a consequence of the above lemma.

Theorem 9.8 Let P be a program and Q be a query which respect atomic positions. Let p be a predicate such that an atom using p is selectable in P only if it is non-variable in its atomic positions. Then in any delay-respecting derivation of $P \cup \{Q\}$, an atom using p is selected only when it is ground in its atomic positions.

Using Theorem 9.8 we can show freedom from instantiation errors for programs where the arithmetic arguments are variable-disjoint from any other arguments, such as the program in Figure 23. Note that *type* errors cannot be ruled out using the theorem.

Note also that for this example, we can only rule out instantiation errors caused by `<`, since the auxiliary predicate `less` realises a `block` declaration for `<`. We cannot rule out instantiation errors caused by `is`. In Section 10.1, it will be justified that there is no `block` declaration for `is`.

9.5 Discussion

In this chapter, we have presented verification methods concerning four aspects of verification: freedom from unification, occur-check, floundering, and errors related to built-ins. These methods build on and improve previous work in this area [AE93, AL95].

We have shown that permutation simply typed programs are unification free for arbitrary input-consuming derivations. This result is more general than the corresponding one by Apt and Etalle [AE93] since they only consider (input-consuming) *LD*-derivations. However, we require that all clause heads are input-linear and have flat terms in their input positions.

Our results on occur-check freedom and non-floundering are straightforward variations of previous results [AL95]. They are based on the observation that when we consider derivations where the textual order of atoms in a query is irrelevant for the selection of an atom, any result for nicely moded or well typed programs trivially generalises to *permutation* nicely moded or *permutation* well typed programs. Note that our result on occur-check freedom holds for *all* derivations.

We have shown that for (arithmetic) built-ins, `block` declarations are often sufficient to ensure freedom from instantiation and type errors. This improves previous results [AL95] in that those assume delay declarations that test for groundness. In the next chapter, we will show that sometimes, no delay declarations are needed at all.

Chapter 10

Weakening Some Conditions

In this chapter, we consider ways of weakening some conditions imposed on the programs for verification purposes. We have postponed these considerations so far to avoid making the main arguments of Part III unnecessarily complicated.

In Section 10.1, we give conditions so that certain `block` declarations can be omitted without affecting the runtime behaviour. In Section 10.2, we study ways of weakening the requirement that clause heads must be input-linear. Section 10.3 shows that we can easily generalise the notion of *mode of a program*. Section 10.4 is a discussion.

10.1 Simplifying the `block` Declarations

Even for programs containing `block` declarations, it is rare that *all* predicates have `block` declarations. In particular, `block` declarations for built-ins are awkward because they can only be realised (at least in SICStus [SIC98]) by introducing an auxiliary predicate (see Figure 14 on page 87). This makes previous methods for verification [AL95] but also the methods we introduced in Chapter 9 somewhat impractical. The `nqueens` program (Figure 22 on page 106), which is a standard example of a program using `block` declarations, does not have any `block` declarations for the built-ins.

Even for user-defined predicates, it is desirable to omit the `block` declarations if possible, since runtime testing for instantiation has an overhead, albeit small.

In this section, we show how, using information about the initial query, it can be ensured that some of the instantiation tests always succeed so that they actually become redundant. This justifies the omission of `block` declarations.

An additional benefit is that in some cases, we can even ensure that arguments are ground, rather than just non-variable. We will see in Section 10.2 that this is useful in order to weaken the restriction that every clause head must be input-linear.

10.1.1 Permutation Simply Typed Programs Using Constant Types

In the program in Figure 22 on page 106, there are no `block` declarations and hence no auxiliary predicates for `<`, `is` and `=\=`. This is justified because the input for those predicates is always provided by the clause heads. For example, it is not necessary to have a `block` declaration for `<` because when an atom using `sequence` is called, the first argument of this atom is already ground.

We show here how this intuition can be formalised for permutation simply typed programs. In the following definition, we consider a set $\mathcal{B}$ containing the predicates for which we want to omit the `block` declarations.

Definition 10.1 [$\mathcal{B}$-ground] Let P be a permutation simply typed program and $\mathcal{B}$ a set of predicates whose input positions are all of constant type.

A query is **$\mathcal{B}$-ground** if it is permutation simply typed and each atom using a predicate in $\mathcal{B}$ has ground terms in its input positions.

An argument position k of a predicate p in P is a **$\mathcal{B}$-position** if there is a clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ in P such that for some i where $p_i \in \mathcal{B}$, some variable in $\mathbf{s}_i$ also occurs in position k in $p(\mathbf{t}_0, \mathbf{s}_{n+1})$.

The program P is **$\mathcal{B}$-ground** if every $\mathcal{B}$-position of every predicate in P is an input position of constant type, and an atom $p(\mathbf{s}, \mathbf{t})$, where $p \notin \mathcal{B}$, is selectable only if it is non-variable in the $\mathcal{B}$-positions of p. ◁

As the following example shows, the requirement on selectability in the above definition is not automatically met by programs with input selectability.

Example 10.1 The `nqueens` program (Figure 22 on page 106) is $\mathcal{B}$-ground, where $\mathcal{B} = \{$`<`, `is`, `=\=`$\}$. The first position of `sequence`, the second position of `safe_aux`, and all positions of `no_diag` are $\mathcal{B}$-positions.

Does input selectability guarantee for this example that an atom $p(\mathbf{s}, \mathbf{t})$, where $p \notin \mathcal{B}$, is selectable only if it is non-variable in the $\mathcal{B}$-positions of p? According to Definition 7.3, the second position of `safe_aux` and all positions of `no_diag` might be free positions. Therefore the answer is no. However, the `block` declarations given in Figure 22 do guarantee this requirement. ◁

The following theorem says that for $\mathcal{B}$-ground programs, the input of all atoms using predicates in $\mathcal{B}$ is always ground.

Theorem 10.1 Let P be a $\mathcal{B}$-ground, input-linear program and Q a $\mathcal{B}$-ground query, and ξ an input-consuming, delay-respecting derivation of $P \cup \{Q\}$. Then each query in ξ is $\mathcal{B}$-ground.

Proof. The proof is by induction on the length of ξ. Let $Q_0 = Q$ and $\xi = Q_0; Q_1; \dots$. The base case holds by the assumption that Q_0 is $\mathcal{B}$-ground.

Now consider some Q_j where $j \geq 0$ and Q_{j+1} exists. By Lemmas 7.2 (f) and 7.7, Q_j and Q_{j+1} are permutation simply typed and type-consistent. The induction hypothesis is that Q_j is $\mathcal{B}$-ground.

Let $p(\mathbf{u}, \mathbf{v})$ be the selected atom, $C = p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \dots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be the clause and θ the MGU used in the step $Q_j; Q_{j+1}$. Consider an arbitrary $i \in \{1, \dots, n\}$ such that $p_i \in \mathcal{B}$.

If $p \notin \mathcal{B}$, then by the condition on selectability in Definition 10.1, $p(\mathbf{u}, \mathbf{v})$ is non-variable in the $\mathcal{B}$-positions of p, and hence, since the $\mathcal{B}$-positions are of constant type, $p(\mathbf{u}, \mathbf{v})$ is *ground* in the $\mathcal{B}$-positions of p. If $p \in \mathcal{B}$, then $p(\mathbf{u}, \mathbf{v})$ is ground in all input positions by the induction hypothesis, and hence $p(\mathbf{u}, \mathbf{v})$ is a fortiori ground in all $\mathcal{B}$-positions of p.

Thus it follows that $\mathbf{s}_i\theta$ is ground. Since the choice of i was arbitrary and because of the induction hypothesis, it follows that Q_{j+1} is $\mathcal{B}$-ground. □

In Section 7.4, we have seen that input-consuming derivations can be ensured with `block` declarations so that programs have input selectability (Theorem 7.9). Now by the above theorem, we can drop the requirement of input selectability for the predicates in $\mathcal{B}$. Regardless of selectability, atoms using predicates in $\mathcal{B}$ are only selected when their input is ground, simply because their input is ground at all times during the execution. Theorems 7.9 and 9.6 are applicable for programs where only the predicates not in $\mathcal{B}$ meet the requirement of input selectability. On the other hand, for those predicates, the requirements on the `block` declarations may actually go beyond input selectability.

Example 10.2 In the `nqueens` program (Figure 22 on page 106), there are no `block` declarations, and hence no auxiliaries, for the occurrences of `is`, `<` and `=\=`, but there are `block` declarations on `safe_aux` and `no_diag` that ensure the condition on selectability in Definition 10.1. Theorems 7.9 and 9.6 are applicable for the `nqueens` program. ◁

10.1.2 Programs that Respect Atomic Positions

The idea used in the previous subsection can also be applied to programs which are not permutation simply typed but which respect atomic positions. However there are some small technical differences. The example we use for illustration here is the program in Figure 23 on page 119.

Note that in the following definition, we associate a mode (or possibly several alternative modes) with a program, although Definition 9.6 is independent of modes.

Definition 10.2 [$\mathcal{B}$-ground*] Let P be a program which respects atomic positions and $\mathcal{B}$ a set of predicates whose input positions are all atomic.

A query is **$\mathcal{B}$-ground*** if it respects atomic positions and each atom using a predicate in $\mathcal{B}$ has ground terms in its input positions.

An argument position k of a predicate p in P is a **$\mathcal{B}$-position*** if there is a clause $p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ in P such that for some i where $p_i \in \mathcal{B}$, some variable in $\mathbf{s}_i$ also occurs in position k in $p(\mathbf{t}_0, \mathbf{s}_{n+1})$.

The program P is **$\mathcal{B}$-ground*** if every $\mathcal{B}$-position* of every predicate in P is an atomic input position, and an atom $p(\mathbf{s}, \mathbf{t})$, where $p \notin \mathcal{B}$, is selectable only if it is non-variable in the $\mathcal{B}$-positions* of p. ◁

Example 10.3 Consider the program in Figure 23 on page 119 with atomic positions defined as in Example 9.4. This program is $\{$`is`$\}$-ground*, and the second position of `len_aux` is an $\{$`is`$\}$-position*. ◁

The following theorem is analogous to Theorem 10.1.

Theorem 10.2 Let P and Q be a $\mathcal{B}$-ground* program and query, and ξ be a delay-respecting derivation of $P \cup \{Q\}$. Then each query in ξ is $\mathcal{B}$-ground*.

PROOF. The proof is by induction on the length of ξ. Let $Q_0 = Q$ and $\xi = Q_0; Q_1; \ldots$. The base case holds by the assumption that Q_0 is $\mathcal{B}$-ground*.

Now consider some Q_j where $j \geq 0$ and Q_{j+1} exists. By Lemma 9.7, Q_j and Q_{j+1} respect atomic positions. The induction hypothesis is that Q_j is $\mathcal{B}$-ground*.

Let $p(\mathbf{u}, \mathbf{v})$ be the selected atom, $C = p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ be the clause and θ the MGU used in the step $Q_j; Q_{j+1}$. Consider an arbitrary $i \in \{1, \ldots, n\}$ such that $p_i \in \mathcal{B}$.

If $p \notin \mathcal{B}$, then by the condition on selectability in Definition 10.2, $p(\mathbf{u}, \mathbf{v})$ is non-variable in the $\mathcal{B}$-positions* of p, and hence, since Q_j respects atomic positions, $p(\mathbf{u}, \mathbf{v})$ is *ground* in the $\mathcal{B}$-positions* of p. If $p \in \mathcal{B}$, then $p(\mathbf{u}, \mathbf{v})$ is ground in all input positions by the induction hypothesis, and hence $p(\mathbf{u}, \mathbf{v})$ is a fortiori ground in all $\mathcal{B}$-positions of p.

Thus it follows that $\mathbf{s}_i\theta$ is ground. Since the choice of i was arbitrary and because of the induction hypothesis, it follows that Q_{j+1} is $\mathcal{B}$-ground*. □

By Theorem 10.2, it is justified that there is no `block` declaration for `is` in the program in Figure 23 on page 119. More precisely, any delay-respecting derivation for this program and an {`is`}-ground* query is also a derivation for the same program except that `is` is only selectable when its input is non-variable. Therefore by Theorem 9.8, there are no instantiation errors.

10.1.3 Exploiting the Fact that Derivations Are Left-Based

We now show that if derivations are left-based, the `block` declarations can be omitted in even more cases.

Definition 10.3 [well placed] Let P be a permutation well typed program and $Q = p_1(\mathbf{s}_1, \mathbf{t}_1), \ldots, p_n(\mathbf{s}_n, \mathbf{t}_n)$ a π-well typed query. An atom $p_i(\mathbf{s}_i, \mathbf{t}_i)$ is **well placed in** Q if for all $j \in \{1, \ldots, n\}$, $\pi(j) < \pi(i)$ implies $j < i$. For the clause $C = p(\mathbf{t}_0, \mathbf{s}_{n+1}) \leftarrow Q$, an atom is **well placed in** C if it is well placed in Q. ◁

Not surprisingly, well placed atoms stay well placed throughout a derivation. This proposition can be verified by inspecting Definition 5.1.

Proposition 10.3 Let C and Q be a permutation well typed clause and query and let Q' be a resolvent of C and Q. Then each well placed atom in Q, other than the selected atom, is also well placed in Q'. Moreover, if the selected atom is well placed in Q, then each well placed atom in C is also well placed in Q'.

The following theorem says that in a left-based derivation, a well placed atom is not selected before it is correctly typed in its input positions, since the atoms that "feed" it will always be preferred. Therefore, if it can be ensured that atoms using a predicate p are always well placed, then it is not necessary to check the input positions of atoms using p with `block` declarations.

Theorem 10.4 Let P be a permutation well typed program where an atom is selectable in P if all input positions of non-variable type are non-variable, and let Q be a permutation well typed query. Let p be a predicate and $\mathcal{W} = \{q \mid q \sqsupseteq p\}$, and suppose in

Q and all clauses of P, all atoms using predicates in $\mathcal{W}$ are well placed. Then in any left-based derivation of $P \cup \{Q\}$, all atoms using predicates in $\mathcal{W}$ are selected only when they are correctly typed in their input positions.

PROOF. Let ξ be a left-based derivation of $P \cup \{Q\}$. We show that atoms using predicates in $\mathcal{W}$ which are eventually selected never become waiting (see Definition 8.1) in ξ.[1] In particular, we look at one arbitrary but fixed atom using a predicate in $\mathcal{W}$ which is eventually selected in ξ. We show that if it is not waiting at some query in ξ, then it will never become waiting. When it is eventually selected, then any direct descendants of this atom that use a predicate in $\mathcal{W}$ are not waiting either. Since in the initial query, no atoms are waiting, it follows by an obvious inductive argument that atoms using predicates in $\mathcal{W}$ which are eventually selected never become waiting in ξ.

Let $Q' = a_1, \ldots, a_n$ be a π-well typed query in ξ where for some $i \in \{1, \ldots, n\}$, a_i is an atom using a predicate in $\mathcal{W}$ which is eventually selected in ξ. Assume that a_i is not waiting. By Proposition 10.3, a_i is well placed in Q'. Since a_i is eventually selected, we can write ξ as

$$\xi = Q; \ldots; Q'; \ldots; (F, a_i, H)\theta;\ (F, B, H)\sigma \ldots$$

Consider an arbitrary query $(\tilde{F}, a_i, \tilde{H})\tilde{\theta}$ in $Q'; \ldots; (F, a_i, H,)\theta$ (that is, a query in ξ before a_i is selected).

If $(\tilde{F}, a_i, \tilde{H})\tilde{\theta}$ contains any descendents of atoms a_j such that $\pi(j) < \pi(i)$, then since a_i is well placed in Q', it follows that these descendants occur in $\tilde{F}\tilde{\theta}$. Since $(\tilde{F}, a_i, \tilde{H})\tilde{\theta}$ is permutation well typed, it follows by Lemma 5.1 (a) that there is at least one selectable atom in $\tilde{F}\tilde{\theta}$, and therefore a_i does not become waiting in the derivation step following $(\tilde{F}, a_i, \tilde{H})\tilde{\theta}$.

If however, $(\tilde{F}, a_i, \tilde{H})\tilde{\theta}$ contains no descendant of an atom a_j such that $\pi(j) < \pi(i)$, then by Lemma 5.12, $a_i\tilde{\theta}$ is correctly typed in its input positions and hence selectable. Therefore a_i does not become waiting in the derivation step following $(\tilde{F}, a_i, \tilde{H})\tilde{\theta}$.

Since a_i is never waiting, it follows by the definition of left-based derivations that a_i can only be selected if there is no selectable atom to the left of a_i. That is, $F\theta$ contains no selectable atom. Therefore, since a_i is well-placed in Q', it follows that $(F, a_i, H)\theta$ contains no descendant of an atom a_j such that $\pi(j) < \pi(i)$, and thus by Lemma 5.12, $a_i\theta$ is correctly typed in its input positions.

Moreover, since a_i is not waiting when it is selected, it follows that the direct descendants of a_i that use a predicate in $\mathcal{W}$ are not waiting either. □

Note that permutation robustly typed programs with input selectability (Definition 7.5) fulfill the condition that an atom is selectable if it is non-variable in all input positions of non-variable type.

In a similar way as in Subsection 10.1.1, the above theorem justifies dropping the requirement of input selectability for the predicates in $\mathcal{W}$. Theorem 7.9 is applicable for programs where only the predicates not in $\mathcal{W}$ meet the requirement of input selectability.

[1] They also never become waiting if they are never selected, but we are not interested in such atoms.

Example 10.4 In the `nqueens` program (Figure 22 on page 106 and Figure 20 on page 99), the `block` declaration for `permute` can be omitted. Note however that this requires that any call to `nqueens` is well placed in the query where it occurs. Moreover, the version of `permute` without `block` declarations can only be used in mode `permute`(O, I). ◁

10.2 Weakening Input-Linearity of Clause Heads

For most of our results, it is assumed that programs are input-linear. Building on the previous section, we now discuss ways of weakening this rather severe restriction.

The requirement that clause heads are input-linear is needed to show the persistence of permutation nicely-modedness (Lemma 5.3). This is analogous to the same statement restricted to nicely-modedness (Lemma 5.2, [AL95, Lemma 11]). However, the clause head does not have to be input-linear when the statement is further restricted to *LD*-resolvents [AP94b, Lemma 5.3]. The following example by Apt (personal communication) demonstrates this difference.

Example 10.5 Consider the program

```
eq(A,A).
q(A).
r(1).
```

where the mode is $\{$`q`(I), `r`(O), `eq`$(I, I)\}$. The query

$$\texttt{q(X), r(Y), eq(X,Y)}$$

is nicely moded. The query `q(X), r(X)` is a resolvent of the above query, and it is *not* nicely moded. Since `eq`/2 is equivalent to the built-in =/2, the example illustrates why input-linear programs must not contain uses of =(I, I). ◁

Requiring input-linear clause heads is undoubtedly a severe restriction. It means that it is not possible to test two input arguments for equality. However, this also indicates why in the above example, resolving `eq(X,Y)` is harmful: `eq` is intended to be a test, clearly indicated by its mode `eq`(I, I), but in the given derivation step, it is actually not a test, since it binds variables.

By Lemma 5.4, the requirement of input-linear heads can be dropped if derivation steps are input-consuming. This means that an atom using =(I, I) must be only selected when both arguments are ground.

The mode =(I, I) could be realised with an equality test, say `eq_test`(s, t), whose operational semantics is as follows: if s and t are identical, it succeeds; if s and t are not unifiable, it fails; otherwise, the test is delayed until s or t become further instantiated. Such a test is used in the *guards* of clauses in concurrent (constraint) logic languages such as (F)GHC [Ued86], but in ordinary logic programming languages, it is usually not provided.

Alternatively, the mode =(I, I) can be realised with a delay declaration such that an atom s=t is selected only when s and t are ground. In SICStus, this can be done using the built-in `when` [SIC98]. However we do not follow this line because we focus on `block` declarations, and because it would commit a particular occurrence of s=t to be a test in all modes in which the program is used.

Nevertheless, even using `block` declarations, there are situations when clause heads that are not input-linear can be allowed. Effectively, we have to show that each derivation step using a non input-linear clause could be replaced with a derivation step using an input-linear clause.

We first need to define formally what it means for an atom to have a subterm "in a certain place", and what a non-linear place is.

Definition 10.4 [to have in a place] Let $a = p(t_1, \dots, t_n)$ be an atom. Then for each $i \in \{1, \dots, n\}$, a **has** t_i **in place** p^i. Moreover, if a has a term $f(s_1, \dots, s_m)$ in place ζ, then for each $i \in \{1, \dots, m\}$, a **has** s_i **in place** $\zeta.f^i$. A place ζ is an **input place** of a if $\zeta = p^i.\zeta'$ and i is an input position of p. $\lhd$

Example 10.6 The atom $\mathtt{p(f(g(X)),h(Y))}$ has $\mathtt{X}$ in place $\mathtt{p}^1.\mathtt{f}^1.\mathtt{g}^1$ and $\mathtt{Y}$ in place $\mathtt{p}^2.\mathtt{h}^1$. The atom $\mathtt{p(f(g(Z)),h(7))}$ has $\mathtt{7}$ in the same place where $\mathtt{p(f(g(X)),h(Y))}$ has $\mathtt{Y}$. $\lhd$

Definition 10.5 [non-linear place] Let $p(\mathbf{v}, \mathbf{u}) \leftarrow B$ be clause. A place ζ is a **non-linear place** of $p(\mathbf{v}, \mathbf{u})$ if it is an input place and $p(\mathbf{v}, \mathbf{u})$ has a variable in ζ which occurs more than once in $\mathbf{v}$. $\lhd$

Example 10.7 Let $\mathtt{p(f(g(X)),h(X))} \leftarrow \dots$ be a clause where the mode is $\mathtt{p}(I, I)$. Then $\mathtt{p}^1.\mathtt{f}^1.\mathtt{g}^1$ and $\mathtt{p}^2.\mathtt{h}^1$ are non-linear places of $\mathtt{p(f(g(X)),h(X))}$. Moreover, $\mathtt{p(f(g(Z)),h(7))}$ has the terms $\mathtt{Z}$ and $\mathtt{7}$ in the non-linear places of $\mathtt{p(f(g(X)),h(X))}$. $\lhd$

The following lemma states that if a selected atom is ground in all non-linear places of the clause head, and the selected atom is unifiable with the clause head, then this clause can be replaced by a certain input-linear clause without affecting the resolvent. Note the similarity between the following lemma and Lemma 5.4.

Lemma 10.5 Let $Q = a_1, \dots, a_n$ be a query and $C = p(\mathbf{v}, \mathbf{u}) \leftarrow b_1, \dots, b_m$ a clause where $vars(Q) \cap vars(C) = \emptyset$. Suppose that for some $k \in \{1, \dots, n\}$, $p(\mathbf{v}, \mathbf{u})$ and $a_k = p(\mathbf{s}, \mathbf{t})$ are unifiable with MGU θ, and $p(\mathbf{s}, \mathbf{t})$ is ground in all non-linear places of $p(\mathbf{v}, \mathbf{u})$.

Let $C' = p(\mathbf{v}', \mathbf{u}) \leftarrow b_1, \ldots, b_m$ be an input-linear clause such that

1. $vars(\mathbf{v}) \subseteq vars(\mathbf{v}')$ and $vars(\mathbf{v}') \cap vars(Q) = \emptyset$,
2. there exists a substitution σ such that $C'\sigma = C$ and $dom(\sigma) = vars(\mathbf{v}') \setminus vars(\mathbf{v})$.

Then $(a_1, \ldots, a_{k-1}, b_1, \ldots, b_m, a_{k+1}, \ldots, a_n)\theta$ is also a resolvent of Q and C'.

PROOF. Consider an arbitrary variable x that occurs more than once in $\mathbf{v}$, and let $X \subseteq dom(\sigma)$ be the set of variables x' such that $x'\sigma = x$. Since $p(\mathbf{s}, \mathbf{t})$ is ground in all places where $p(\mathbf{v}, \mathbf{u})$ has x, it follows that $\mathbf{s}$ has the same ground term, say t, in all places where $\mathbf{v}$ has x. Therefore it follows that any unifier of $p(\mathbf{v}', \mathbf{u})$ and $p(\mathbf{s}, \mathbf{t})$ binds each variable in X to t. This means that $\sigma\theta$ is an MGU of $p(\mathbf{v}', \mathbf{u})$ and $p(\mathbf{s}, \mathbf{t})$. Moreover, by assumptions 1 and 2,

$$(a_1, \ldots, a_{k-1}, b_1, \ldots, b_m, a_{k+1}, \ldots, a_n)\theta = (a_1, \ldots, a_{k-1}, b_1, \ldots, b_m, a_{k+1}, \ldots, a_n)\sigma\theta,$$

and so $(a_1, \ldots, a_{k-1}, b_1, \ldots, b_m, a_{k+1}, \ldots, a_n)\theta$ is not only a resolvent of C and Q, but also a resolvent of C' and Q. □

Lemma 10.5 should not be interpreted as suggesting a program transformation, namely to replace clauses with corresponding input-linear clauses. Such a transformation might make a clause head unifiable with a potential selected atom where it was not unifiable before, which would affect the semantics of the program. It is only in the case that $p(\mathbf{s}, \mathbf{t})$ and $p(\mathbf{v}, \mathbf{u})$ are already unifiable that we can, conceptually, replace C with C'.

Lemma 10.5 is applicable whenever we can guarantee that the selected atom is always ground in the non-linear places of the clause head. We now outline two ways in which this can be ensured.

First, one can exploit the fact that atoms are well placed. Consider a permutation well typed program where an atom is selectable in P if all input positions of non-variable type are non-variable. We can weaken Definition 5.3 by allowing for clause heads $p(\mathbf{t}, \mathbf{s})$ where a variable x occurs in several input positions, provided that

- all occurrences of x in $\mathbf{t}$ are in positions of ground type, and
- in each clause of the program and in any initial query for the program, each atom using a predicate $q \sqsupseteq p$ is well placed.

By Theorem 10.4, it is then ensured that multiple occurrences of a variable in the input of a clause head implement an equality *test* between input arguments. Therefore, Lemmas 5.3, 7.2 and 7.6 hold assuming this weaker definition of "input-linear".

Example 10.8 Consider the `append` program (Figure 10 on page 57) in "test mode", that is `append`(I, I, I). This program is permutation nicely moded but not input-linear. Nevertheless, the program can be used in this mode provided that all arguments are of ground type and calls to `append` are always well placed. ◁

Secondly, one can exploit the fact that the arguments being tested for equality are of constant type. This time we have to weaken Definition 5.3 by allowing for clause heads $p(\mathbf{t}, \mathbf{s})$ where a variable x occurs in several input positions, provided that

- all occurrences of x in $\mathbf{t}$ are direct and in positions of constant type, and
- an atom using p is selectable only if these positions are non-variable.

By Theorem 9.6, it is then ensured that when an atom $p(\mathbf{u}, \mathbf{v})$ is selected, $\mathbf{u}$ has constants in each position where $\mathbf{t}$ has x.

Example 10.9 The `length` program in Figure 23 on page 119 can be used in mode $\{\texttt{length}(O, I), \texttt{len\_aux}(O, I, I)\}$ in spite of that fact that `len_aux([],N,N)` is not input-linear, using either of the two explanations above. The first explanation relies on all atoms using predicates $q \sqsupseteq \texttt{len\_aux}$ being well placed. This is somewhat unsatisfactory since imposing such a restriction impedes modularity. Therefore, the second explanation is preferable. $\triangleleft$

10.3 Generalising Modes

In Section 5.2, we have defined a mode of a program as a set containing one mode for each of its predicates. This means that we have allowed for a program to be used in different modes at different executions, but within each execution, the mode of each predicate was fixed. For example, the query

$$\texttt{append([1,2],[3,4],Zs)},\ \texttt{append(As,Bs,Zs)},$$

where the first atom has mode $\texttt{append}(I, I, O)$ and the second atom has mode $\texttt{append}(O, O, I)$, uses the same predicate in different modes and hence would not be, say, permutation nicely moded. This is a disadvantage as one can easily imagine that a program might use $\texttt{append}(I, I, O)$ is one place and $\texttt{append}(O, O, I)$ in another. We have defined modes in this way to avoid unnecessary confusion.

It is easy to see however that the definition of a mode of a program could be generalised. Define a mode M of a program as a set of modes containing *at least one* mode for each of its predicates. Define that a clause $C = p(\mathbf{t}, \mathbf{s}) \leftarrow B$ is, say, permutation nicely moded *with respect to a mode* $p(m_1, \dots, m_n) \in M$ if it is permutation nicely moded, assuming that the mode $p(m_1, \dots, m_n)$ is assigned to the clause head and some mode in M is assigned to each body atom in C. Define that a program is permutation nicely moded if for each predicate p and each mode $p(m_1, \dots, m_n) \in M$, all clauses defining p are permutation nicely moded with respect to $p(m_1, \dots, m_n)$.

10.4 Discussion

In this chapter, we presented some methods that can be used to improve the results of the earlier chapters in two ways: omitting the `block` declarations for some predicates, and allowing for multiple occurrences of variables in the input of clause heads.

Omitting the `block` declarations is particularly useful for (arithmetic) built-ins. It aims at the way arithmetic built-ins are used in practice: it is awkward having to introduce auxiliary predicates to implement delay declarations for built-ins. The `nqueens` program is a standard example of a program which contains `block` declarations, but not for the built-ins. We give a formal justification for this.

The requirement that clauses must be input-linear is quite common [AL95, AE93, ER98]. However it is a rather severe restriction, in that it usually rules out predicates running in "test mode" (see Example 10.8). We have shown how this restriction can sometimes be weakened.

Finally, we have outlined a generalisation of modes allowing for predicates to be used in different modes in different places in a program, even within a single execution.

Chapter 11

Related Work and Conclusion

In Chapter 2, we gave an overview of the literature using modes and types. In this chapter, we look, more specifically, at the literature related to Part III of this thesis. We then conclude the thesis by highlighting the main contributions and novel ideas, and mentioning some open problems.

11.1 Related Work

This section has several subsections, each of which is devoted to a paper or a group of closely related papers. Subsection 11.1.1 is an exception. It discusses the significance of "pinning down the size" of an atom throughout the termination literature. We discuss the papers more or less in chronological order.

In Subsection 6.1, we observed that a distinguishing aspect between works on termination is the assumptions they make about the selection rule. This includes assumptions about delay declarations, as one usually thinks of the selection rule as being parametrised by the delay declarations, if there are any. Figure 24 illustrates a variety of assumptions about the selection rule that have been made in the literature. We will refer to this figure as we discuss the different approaches.

11.1.1 The Significance of "Pinning Down the Size" of an Atom

As explained in Section 6.1, most approaches to termination rely on the idea that the size of an atom can be pinned down when the atom is selected. Depending on this size, it is then possible to give an upper bound for the number of descendants of this atom.

Technically, "pinning down the size" usually means that the atom is *bounded* with respect to some level mapping [AP94a, Bez93, EBC99, LS97, MK97]. However, there are exceptions [DVB92, DD98]. In those works, termination can be shown for the query, say, $\mathtt{append([X],[],Zs)}$ using as level mapping the term size of the first argument, even though the term size of $\mathtt{[X]}$ is not bounded. However, the method only works for LD-derivations and relies on the fact that any future instantiation of $\mathtt{X}$ cannot affect the descendants of $\mathtt{append([X],[],Zs)}$. Therefore it is effectively possible to pin down the size of $\mathtt{append([X],[],Zs)}$.

On the whole, there seems to be a strong reluctance to give up this idea, although

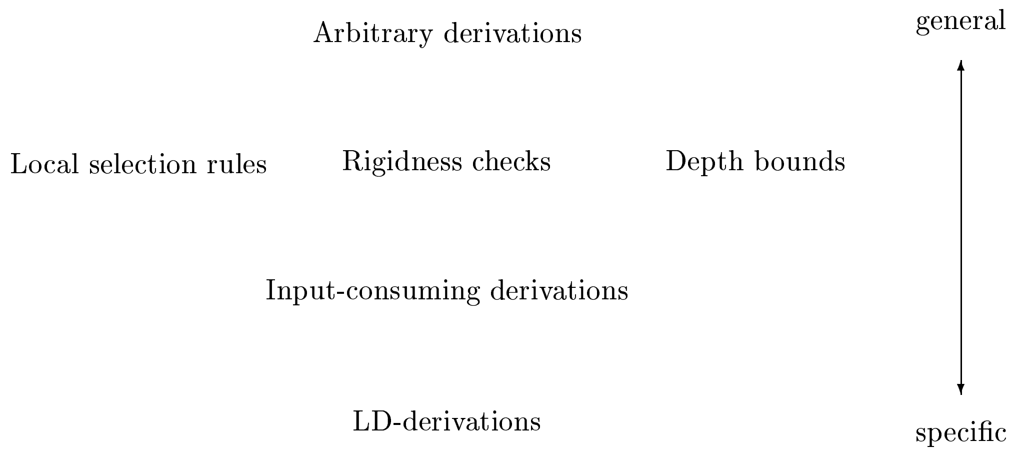

Figure 24: Assumptions about the selection rule

it is recognised that it must fail on some standard examples of programs using coroutining [Nai92]. This is illustrated in Example 6.1. Therefore, some authors attempt to simplify the actual problem by proposing program transformations or introducing additional assumptions about the selection rule [Bez93, MT95, MK97]. It seems that these modifications mainly serve the purpose of making it easier to *reason* about termination, and not of making programs terminate that would not terminate otherwise. We will discuss this point further below when we look at the various approaches.

11.1.2 Guarded Horn Clauses

The definition of input-consuming derivations has a certain resemblance with derivations in the language of *(Flat) Guarded Horn Clauses* [Ued86, Ued88]. In (F)GHC, a clause has the form $h \leftarrow G \,|\, B$, where G is called a *guard*. There is no backtracking, that is, the choice of a clause to resolve an atom cannot be undone later if the derivation fails. It is therefore crucial that the "correct" clause is used in each step. To this end, an atom a can be resolved using a clause $h \leftarrow G \,|\, B$ only when a is an instance of h and $G\theta$ is entailed, where θ is an MGU of a and h. The atom a can become instantiated only later via explicit unifications (using the built-in equality predicate) occurring in the body B.

Thus whether or not an atom a is selectable in (F)GHC depends not only on a itself but, at least in theory, on the clause used to resolve a. This is similar to the concept of input-consuming derivations, where whether or not a derivation step is input-consuming may depend on the clause used to resolve an atom.

When we consider *Moded* FGHC [CU96, UM93, UM94], this resemblance becomes even clearer. Intuitively, arguments of the selected atom that affect the choice of the clause are input arguments, whereas arguments that become instantiated by the body B are output arguments. In Moded FGHC, a number of correctness conditions are imposed that formalise, among other things, this intuition.

11.1.3 Coroutining and Terminating Logic Programs

Naish studies the problem of termination of programs with coroutining [Nai92]. He considers the `when` declarations of NU-Prolog [TZ86], which are essentially the same as `block` declarations. These declarations effectively ensure input-consuming derivations, although Naish does not use this concept. The default left-to-right selection rule of Prolog is assumed. This work gives good intuitive explanations why programs loop and heuristics to ensure termination. However, the work is not formal. It is not even formalised what the default left-to-right selection rule is.

Predicates are assumed to have a single mode. As mentioned on page 62, Naish suggests that alternative modes should be achieved by multiple versions of a predicate. This approach is quite common and is also taken in Mercury [SHC96], where these versions are generated by the compiler. While it is possible to take that approach, some authors give the impression that assuming single modes does not imply any loss of generality [AE93, AL95, EBC99]. However, generating multiple versions implies code duplication and hence a loss of generality (see Subsection 5.3.2).

Naish uses examples where under the assumption of single modes, there is no reason for using delay declarations in the first place. For example, if we only consider `permute`(O, I), then the program in Figure 20 (page 99) does not loop for the plain reason that no atom ever delays, and thus the program is executed using LD-derivations. In this case, the elaborate interpretation that one should "place recursive calls last" is misleading. On the other hand, if we only consider `permute`(I, O), then the version of Figure 20 would hardly be used, on the grounds that it is much less efficient than the version of Figure 18 (page 95). In short, Naish's discussion on delay declarations lacks motivation when only one mode is assumed.

11.1.4 Strong Termination

Bezem [Bez93] has identified the class of strongly terminating programs, which are programs that universally terminate under *any* selection rule (see Figure 24 on the facing page). While it is shown that every total recursive function can be computed by a strongly terminating program, this does not change the fact that few existing programs are strongly terminating. Transformations are proposed for three example programs to make them strongly terminating, but no general procedure for transforming programs is given.

11.1.5 Generating Delay Declarations Automatically

Lüttringhaus-Kappel [Lüt93] proposes a method for generating control (delay declarations) automatically, and has applied it successfully to many programs. However, rather than pursuing a formalisation of some intuitive understanding of why programs loop, and imposing appropriate restrictions on programs, he attempts a high degree of generality. This has certain disadvantages.

First, the method only finds *acceptable* delay declarations, ensuring that the most general selectable atoms have finite SLD-trees. What is required however are *safe* delay declarations, ensuring that *instances* of most general selectable atoms have finite SLD-trees. A *safe* program is a program for which every acceptable delay declaration is safe. Lüttringhaus-Kappel states that all programs he has considered are safe, but gives no hint as to how this might be shown in general. This is a missing link.

Secondly, the delay declarations for some programs such as `quicksort` require an argument to be a nil-terminated list before an atom can be selected. Such a list is sometimes called *rigid* [MK97, MKS97], since its length cannot change via further instantiation (see Figure 24 on page 132). As Lüttringhaus-Kappel points out, "in NU-Prolog [*or SICStus*] it is not possible to express such conditions"[1] [TZ86]. Note that such uses of delay declarations go far beyond ensuring that derivations are input-consuming. In fact, they ensure that the size of the selected atom can be pinned down.

In a way, the need for such strong delay declarations arises because Lüttringhaus-Kappel assumes arbitrary delay-respecting derivations, rather than left-based derivations. Obviously, his method cannot show termination when termination depends on derivations being left-based.

Thirdly, floundering cannot be ruled out systematically, but only avoided on a heuristic basis. Thus in principle, the method sometimes enforces termination by floundering. This lies in the nature of the weak assumptions made, and thus is sometimes unavoidable, but there is no notion that would allow to reason about whether for a particular program, it was avoidable or not. In contrast, the notions of permutation well-typedness and input-consuming derivations allow to reason about whether floundering is avoidable or not (see Section 9.3).

11.1.6 Verification Using Modes and Types

Apt, **Etalle**, **Luitjes** and **Pellegrini** are among the authors who use correctness properties related to modes and types to verify logic programs [AE93, AL95, AP94b]. These correctness properties have been adopted and extended in this thesis (see Section 7.5).

Apt and Luitjes [AL95] present some methods for verification of logic programs with delay declarations. They consider four aspects of verification: occur-check freedom, non-floundering, freedom from errors related to built-ins, and termination.

The results on occur-check freedom are a generalisation of work by Apt and Pellegrini [AP94b] from LD-derivations to arbitrary derivations. Occur-check freedom is shown based on nicely-modedness. As discussed in Section 10.2, showing the persistence of nicely-modedness, and hence occur-check freedom, for arbitrary derivations requires that clause heads are input-linear.

For arithmetic built-ins, Apt and Luitjes require delay declarations such that an atom is delayed until the arguments are ground. Such declarations are usually implemented less efficiently than `block` declarations.

Little attention is devoted to termination. Apt and Luitjes propose a method for showing termination which is limited to deterministic programs, that is programs where for each selected atom, there is at most one clause head unifiable with it. Moreover,

[1] This statement should probably be weakened. It *is* possible to express such conditions, but only by introducing auxiliary predicates [MK97].

Apt and Luitjes give conditions for the termination of `append`, but these are ad-hoc and do not address the general problem.

The results on unification freedom of Section 9.1 are based on work by Apt and Etalle [AE93]. These authors assume well typed programs and LD-derivations.

11.1.7 Termination of LD-Derivations

The methods for proving termination presented in Chapter 8 implicitly rely on previous work on termination for LD-derivations [Apt97, AP90, DVB92, DD93, DD98, EBC99]. **De Schreye** and **Decorte** give a survey of the termination literature [DD94]. The TermiLog system is a tool for proving termination automatically [LS96, LS97, LSS97].

11.1.8 Termination for Local Selection Rules

For proving termination, **Marchiori** and **Teusink** [MT95] rely on norms and the *covering* relation between subqueries of a query. This is loosely related to well-typedness. However, their results are not comparable to ours because they assume a *local selection rule*, that is a rule which always selects an atom which was introduced in the most recent step. No existing language using a local selection rule is mentioned. Assuming local selection rules, it can be ensured that the size of the selected atom can always be pinned down.

The authors state that programs that do not use *speculative bindings* deserve further investigation, and that they expect any method for proving termination with *full* coroutining either to be very complex, or very restrictive in its applications.

11.1.9 Directional Types

Boye [Boy96] defines *generally well typed* programs, of which the *permutation* well typed programs considered here are a special case. The generalisation lies in considering not just a producer-consumer relation between atoms in a query, but rather between the individual argument positions. This allows to reason about certain programs which operate on open data structures.

The standard example is a program which takes as input a binary tree whose labels are numbers, and returns a tree with the same structure but where all labels are replaced by the maximum label of the original tree. Although this is conceptually a two-pass problem, the program does only one pass over the original tree. This works by first constructing the output tree such that all labels are aliased to the same variable. Only after the original tree has been passed completely, and thus the maximum label is known, will this variable be instantiated.

The maximum label of the original tree is a passed as an *input* argument to the main predicate of this program, and nevertheless, by the very nature of the algorithm, it cannot be instantiated at the time when an atom using this predicate is selected. Therefore programs using this technique cannot work assuming input-consuming derivations. At present, we can only state that such programs are an exception to the principle that derivations must be input-consuming. It would certainly be desirable to generalise the principle so that such programs would also be included.

11.1.10 Termination by Imposing Depth Bounds

Martin and **King** [MK97] ensure termination by imposing a depth bound on the SLD-tree (see Figure 24 on page 132). This is realised by a program transformation introducing additional argument positions for each predicate which are counters for the depth of the computation. As with other approaches, the size of the selected atom can always be pinned down: it is simply the value of the depth bound. The difficulty is of course to find an appropriate depth bound that does not compromise completeness.

11.1.11 Beyond Success and Failure

Etalle and **van Raamsdonk** [ER98] study generalisations of the notions of *successful* and *failing* derivations, which are traditionally regarded as the cornerstones of control in logic programming. They define *non-destructive programs.* This concept is similar to input-consuming derivations, although they take a different viewpoint: they define a program property rather than a property of the selection rule. A non-destructive program is a program for which all delay-respecting derivations are input-consuming. In Chapter 7, we have seen several (syntactically defined) classes of non-destructive programs.

11.1.12 Termination of Well-Moded Programs

Chapter 6 closely follows **Etalle** et al. [EBC99], who study *well-terminating* programs, that is programs for which all LD-derivations for all well moded queries terminate. Proving that a program has this property is based on *moded level mappings* and *well-acceptable* clauses. These concepts are similar to moded typed level mapping (Definition 6.2) and ICD-acceptable clause (Definition 6.4). For simply moded programs, the paper even gives a *characterisation* of well-termination. That is, it shows that if a program is well-terminating, then its clauses are well-acceptable. This is not a contradiction to the undecidability of termination, as the existence of a level mapping with respect to which a program is well-acceptable is undecidable.

11.1.13 $\exists$-Universal Termination

Bezem [Bez93] has defined *strong* termination, which is universal termination for all selection rules. **Ruggieri** [Rug99] has defined a complementary concept called $\exists$-*universal* termination. A program P and query Q $\exists$-universally terminate if there exists a selection rule $\mathcal{S}$ such that all $\mathcal{S}$-derivations of $P \cup \{Q\}$ are finite. This concept is important with regards to the separation of the logic and control aspects of a program as advocated by Kowalski [Kow79]. If a program $\exists$-universally terminates, then it is, at least in principle, possible to associate control with the program so that it actually terminates. If the program does not $\exists$-universally terminate, then it does not terminate for *any* selection rule.

In this context, *fair* selection rules play a special role. A selection rule is *fair* if each atom in a query is eventually selected. Ruggieri shows that a program $\exists$-universally terminates if and only if it terminates for all fair selection rules [Rug99, Theorem 2.4.3]. Thus from the point of view of proving termination, assuming fair selection rules is the strongest assumption one can make about the selection rule. If a program does not terminate for a fair selection rule, it does not terminate for any selection rule.

Note that Ruggieri follows Apt [Apt97] in defining a selection rule as a function that takes a derivation and returns an atom in the last query (the selected atom). However, this definition is too restrictive for our purposes. For example, it is not possible to define a selection rule that exactly corresponds to input-consuming derivations. A selection rule as defined by Apt cannot be used to model the situation that no atom can be selected, or that more than one atom can be selected (so that it is left open which atom is actually selected). Moreover, it cannot be used to model that whether or not an atom can be selected may depend on the clause used to resolve this atom. This latter aspect cannot even be modelled using *sets* of selection rules as defined by Apt. Lloyd [Llo87] has a definition of selection rule which is even more restrictive than that of Apt, in that whether or not an atom is selectable may only depend on the present query, and not on the whole derivation.

11.1.14 Assertion-Based Debugging of (Constraint) Logic Programs

Puebla et al. have developed an assertion-based debugging system for constraint logic programs [PBH99]. This has aspects of program analysis as well as verification. Unlike the verification methods we have presented here, no restrictions (such as well-typedness) are imposed on the program. The system incorporates various techniques involving abstract interpretation and runtime checking. One could imagine that the verification techniques of this thesis could also be incorporated into this system.

11.2 Conclusion

The main contribution of Part III is to provide a method for showing termination of programs with `block` declarations assuming left-based derivations. That is, we are proposing a solution to the termination problem for programs with delay declarations as the problem was originally stated, albeit informally, by Naish [Nai92]. This problem is a "realistic" one, since the assumptions of `block` declarations and left-based derivations reflect the most commonly used implementations.

To the best of our knowledge, this is the first formal and comprehensive approach to this problem. Other authors have either been informal [Nai92], or made other (usually stronger) assumptions and hence studied another problem [MT95, MK97], or dealt with the problem under very restricted circumstances [AL95].

We now highlight some original, distinctive ideas and concepts of Part III. We then mention some open problems. Finally we recall the main results.

11.2.1 Some Distinctive Novel Ideas

Formalising Selection Rules

It is commonly assumed that selected atoms in a derivation should be instantiated to a certain degree in order to ensure termination and other desirable properties [AL95].

In Chapter 5, we presented the concept of *input-consuming derivation*, providing a characterisation of "a certain degree" which is both abstract and intuitive.

Without assuming input-consuming derivations, even predicates for which termination should be trivial do not terminate (see page 9). On the other hand, we have shown that for many predicates, this assumption about the selection rule, together with some correctness conditions satisfied by the program, is sufficient to ensure termination.

However, there are also many predicates for which this assumption is not sufficient. One way to strengthen the assumptions about the selection rule is to assume the default left-to-right selection rule of Prolog. Owing to subtleties involving simultaneously woken atoms, neither software manuals nor theoretical works have attempted to formalise this rule precisely. The notion of *left-based* derivation introduced in this thesis (based on previously published work [SHK98]) is a formalisation of default left-to-right selection rules. It is relatively simple and unrestrictive, so that we can claim with reasonable confidence that derivations in existing Prolog systems are left-based.

Termination without Pinning down the Selected Atom

Most methods for proving termination of logic programs are based on the following idea: when an atom a in a query is selected, it is possible to pin down the size of a, and the new atoms introduced in this derivation step are smaller than a. These methods are bound to fail on most programs using coroutining, such as the coroutining derivation of `append` in Example 6.1 [Bez93, Lüt93, MT95, MK97]. In contrast, we show that under certain conditions, it is sufficient to rely on a relative decrease in the size of the selected atom, even though this size cannot be pinned down. This is the key to proving termination for programs with coroutining.

Three Orderings on Atoms

In this thesis, three different orderings between the atoms of a query (or clause body) are elaborated: the textual order, the producer-consumer order and the execution order. It is shown that for LD-derivations, all of these orders are identical. Moreover, for selection rules where the textual position is irrelevant for the selection of an atom, the textual order and the producer-consumer order can be *assumed* to be identical, as a matter of simplification. For selection rules where the textual position of atoms matters, the producer-consumer order can be made explicit using a permutation of the atoms.

(Permutation) Robustly Typed Programs

Many verification methods for logic programs, including some in this thesis, rely on the assumption that programs are simply moded, so that a query always has variables in the output positions [AE93, EBC99]. In Section 7.4, we define (permutation) robustly-typedness, a correctness property allowing for non-variable terms in certain output positions. This property is persistent under resolution and type-consistent with respect to input-consuming derivations.

We have used this property for showing termination, but it may well have other uses, for example to show unification freedom for a larger class of programs [AE93].

Multiple Modes

Throughout Part III, it is assumed that predicates may be used in multiple modes, although this assumption is not always made explicit. We have argued that in the context of programs using non-standard derivations, one should at least *allow* for multiple modes, although only few predicates can reasonably be used in multiple modes. In previous literature, there is sometimes a lack of motivation: for the examples given, there is no reason for using delay declarations in the first place, if not to enable multiple modes.

`block` Declarations

We have argued that among the various kinds of delay declarations, `block` declarations, which can only test for partial instantiation of arguments of an atom, play a special role. They can be more efficiently implemented than more complex constructs such as delay declarations testing for groundness. Moreover, they are well suited to realise input-consuming derivations while allowing for coroutining.

11.2.2 Open Problems

We now discuss some open problems and possible extensions of this work.

Weakening the Correctness Properties

The verification methods introduced in this thesis are based on a number of correctness properties that the verified programs must have (see Section 7.5). Etalle and Gabbrielli [EG99] have identified programs using *layered modes*, which are a small but interesting class of programs for which none of the above correctness properties holds, since it is not possible to establish a producer-consumer relation (see Subsection 5.3.1) between the atoms of each query. Therefore, Etalle and Gabbrielli refine the concept of producer-consumer relation by considering the individual argument positions rather than entire atoms, similarly to Boye [Boy96]. It would be interesting to extend some results of this thesis to such programs.

Termination for Input-Consuming Derivations

As stated previously (page 81), we cannot show that all input-consuming derivations of `quicksort`(I, O) are finite, although we conjecture that they are. Ideally, one would like to find a *characterisation* of the programs for which all input-consuming derivations are finite (see Section 6.6 and Subsection 11.1.12).

A Uniform Verification Method for Built-ins

For showing that a program is free from errors related to built-ins (Section 9.4), we have introduced two methods. Whether one of these methods or even both are applicable depends on the program. It would be desirable to find *one* uniform approach which would work for a larger class of programs.

Weakening the `block` Declarations

We have discussed that `block` declarations can be omitted or simplified when sufficient instantiation can be guaranteed at compile time. This issue is related to another problem, namely the rather severe restriction that clause heads must be input-linear. It would be interesting to study this relationship further and come up with results that are more general than the ones in Chapter 10.

11.2.3 Summary of Part III

In Part III of this thesis, we have presented verification methods for logic programs using non-standard derivations, that is programs not using the LD selection rule.

In Chapter 5, we motivated the usefulness of non-standard derivations. We then introduced a number of correctness properties concerning the modes of a program. Many verification methods can be based on these properties.

In Chapter 6, we introduced input-consuming derivations as a minimal assumption needed to prove termination. We used level mappings to provide a method for proving that a program (fragment) terminates for all input-consuming derivations.

In Chapter 7, we showed how `block` declarations can be used to ensure that derivations are input-consuming. Examples were used to illustrate that this is a non-trivial problem. We introduced the class of *permutation robustly typed* programs, which is carefully crafted so that `block` declarations can in fact ensure input-consuming derivations, without being too restrictive.

In Chapter 8, we presented a comprehensive method for showing termination for programs with `block` declarations. It is based on the insight that for some atoms, the textual position in a query is irrelevant, whereas other atoms must be placed sufficiently late in a query to ensure that they are always called with sufficient input. This assumes left-based derivations.

In Chapter 9, we presented verification methods concerning some further aspects of verification. These were freedom from unification, occur-check, floundering, and errors related to built-ins.

In Chapter 10, we considered ways of omitting the `block` declarations for some predicates, and allowing for multiple occurrences of variables in the input of clause heads.

Bibliography

[AE93] K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science*, LNCS, pages 1–19, Berlin, 1993. Springer-Verlag.

[AL94] A. Aiken and T. K. Lakshman. Directional type checking of logic programs. In B. Le Charlier, editor, *Proceedings of the 1st Static Analysis Symposium*, LNCS, pages 43–60. Springer-Verlag, 1994.

[AL95] K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In V. S. Alagar and M. Nivat, editors, *Proceedings of AMAST'95*, LNCS, Berlin, 1995. Springer-Verlag. Invited Lecture.

[AM94] K. R. Apt and E. Marchiori. Reasoning about Prolog programs: From modes through types to assertions. *Formal Aspects of Computing*, 6(6A):743–765, 1994.

[AMSH94] T. Armstrong, K. Marriott, P. Schachte, and H.Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In B. Le Charlier, editor, *Proceedings of the 1st Static Analysis Symposium*, LNCS, pages 266–280. Springer-Verlag, 1994.

[AMSH98] T. Armstrong, K. Marriott, P. Schachte, and H.Søndergaard. Two classes of Boolean functions for dependency analysis. *Science of Computer Programming*, 31(1):3–45, 1998.

[AP90] K. R. Apt and D. Pedreschi. Studies in pure Prolog: Termination. In J. W. Lloyd, editor, *Proceedings of the Symposium in Computational Logic*, LNCS, pages 150–176. Springer-Verlag, 1990.

[AP94a] K. R. Apt and D. Pedreschi. Modular termination proofs for logic and pure Prolog programs. In G. Levi, editor, *Advances in Logic Programming Theory*, pages 183–229. Oxford University Press, 1994.

[AP94b] K. R. Apt and A. Pellegrini. On the occur-check free Prolog programs. *ACM Transactions on Programming Languages and Systems*, 16(3):687–726, 1994.

[Apt97] K. R. Apt. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[Bau92] M. Baudinet. Proving termination properties of Prolog programs: A semantic approach. *Journal of Logic Programming*, 14:1–29, 1992.

[BC99] A. Bossi and N. Cocco. Successes in logic programs. In P. Flener, editor, *Proceedings of the 8th International Workshop on Logic Program Synthesis and Transformation*, LNCS, pages 219–239. Springer-Verlag, 1999.

[BCHK97] F. Benoy, M. Codish, A. Heaton, and A. M. King. Widening *Pos* for Efficient and Scalable Groundness Analysis. Technical Report 515, University of Kent at Canterbury, 1997. Available at `http://www.cs.ukc.ac.uk/pubs/1997/515/index.html`.

[BDB$^+$96] M. Bruynooghe, B. Demoen, D. Boulanger, M. Denecker, and A. Mulkers. A freeness and sharing analysis of logic programs based on a pre-interpretation. In R. Cousot and D. A. Schmidt, editors, *Proceedings of the 3rd Static Analysis Symposium*, LNCS, pages 128–142. Springer-Verlag, 1996.

[Bez93] M. Bezem. Strong termination of logic programs. *Journal of Logic Programming*, 15(1 & 2):79–97, 1993.

[BG92] R. Barbuti and R. Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Science of Computer Programming*, 19:281–313, 1992.

[BLR92] F. Bronsard, T. K. Lakshman, and U. S. Reddy. A framework of directionality for proving termination of logic programs. In K. R. Apt, editor, *Proceedings of the 9th Joint International Conference and Symposium on Logic Programming*, pages 321–335. MIT Press, 1992.

[BM95] J. Boye and J. Małuszyński. Two aspects of directional types. In L. Sterling, editor, *Proceedings of the 12th International Conference on Logic Programming*, pages 747–761. MIT Press, 1995.

[Boy96] J. Boye. *Directional Types in Logic Programming.* PhD thesis, Linköpings Universitet, 1996.

[Cav89] L. Cavedon. Continuity, consistency and completeness properties for logic programs. In G. Levi and M. Martelli, editors, *Proceedings of the 6th International Conference on Logic Programming*, pages 571–584. MIT Press, 1989.

[CBGH97] M. Codish, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Exploiting goal independence in the analysis of logic programs. *Journal of Logic Programming*, 32(3):247–261, 1997.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[CC92] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the 4th Symposium on Programming Language Implementations and Logic Programming*, LNCS, pages 269–295. Springer-Verlag, 1992.

[CC94] J. Chassin de Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing Surveys*, 26(3):295–336, 1994.

[CD94] M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of *Prop*. In B. Le Charlier, editor, *Proceedings of the 1st Static Analysis Symposium*, LNCS, pages 281–296. Springer-Verlag, 1994.

[CD95] M. Codish and B. Demoen. Analyzing logic programs using "PROP"-ositional logic programs and a Magic Wand. *Journal of Logic Programming*, 25(3):249–274, 1995.

[CDY94] M. Codish, D. Dams, and E. Yardeni. Bottom-up abstract interpretation of logic programs. *Theoretical Computer Science*, 124(1):93–125, 1994.

[CGBH94] M. Codish, M. García de la Banda, M. Bruynooghe, and M. Hermenegildo. Goal dependent versus goal independent analysis of logic programs. In F. Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, LNCS, pages 305–319. Springer-Verlag, 1994.

[Chr97] H. Christiansen. Deriving declarations from programs. Technical report, Roskilde University, P.O.Box 260, DK-4000 Roskilde, 1997.

[CL96] M. Codish and V. Lagoon. Type dependencies for logic programs using ACI-unification. In *Proceedings of the Israeli Symposium on Theory of Computing and Systems*, pages 136–145. IEEE Press, 1996. To appear in *Theoretical Computer Science.*

[Cod97] M. Codish. Efficient goal directed bottom-up evaluation of logic programs. In L. Naish, editor, *Proceedings of the 14th Joint International Conference and Symposium on Logic Programming.* MIT Press, 1997. Presented as poster.

[CP91] R. Chadha and D.A. Plaisted. Correctness of unification without occur check in Prolog. Technical report, University of North Carolina, 1991.

[CT77] K. L. Clark and S.-Å. Tärnlund. A first order theory of data and programs. In B. Gilchrist, editor, *Information Processing, Proceedings of the IFIP Congress 77, Toronto*, pages 939–944, 1977.

[CU96] K. Cho and K. Ueda. Diagnosing non-well-moded concurrent logic programs. In M. Maher, editor, *Proceedings of the 13th Joint International Conference and Symposium on Logic Programming*, pages 215–229. MIT Press, 1996.

[DD93] S. Decorte and D. De Schreye. Automatic inference of norms: A missing link in automatic termination analysis. In *Proceedings of the 10th International Logic Programming Symposium*, pages 420–436. MIT Press, 1993.

[DD94] D. De Schreye and S. Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19/20:199–260, 1994.

[DD98] S. Decorte and D. De Schreye. Termination analysis: Some practical properties of the norm and level mapping space. In J. Jaffar, editor, *Proceedings of the 15th Joint International Conference and Symposium on Logic Programming*, pages 235–249. MIT Press, 1998.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3(1 & 2):69–115, 1987. Corrigendum 4(3), 409–410.

[DM85] P. Dembinski and J. Małuszyński. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proceedings of the 2nd International Logic Programming Symposium*, pages 29–38. MIT Press, 1985.

[DM98] P. Deransart and J. Małuszyński. Towards soft typing for CLP. In François Fages, editor, *JICSLP'98 Post-Conference Workshop on Types for Constraint Logic Programming.* École Normale Supérieure, 1998. Available at `http://discipl.inria.fr/TCLP98/`.

[DVB92] D. De Schreye, K. Verschaetse, and M. Bruynooghe. A framework for analysing the termination of definite logic programs with respect to call patterns. In *Proceedings of FGCS*, pages 481–488. ICOT Tokyo, 1992.

[DW86] S. K. Debray and D. S. Warren. Detection and optimization of functional computations in Prolog. In E. Shapiro, editor, *Proceedings of the 3rd International Conference on Logic Programming*, LNCS, pages 490–504. Springer-Verlag, 1986.

[EBC99] S. Etalle, A. Bossi, and N. Cocco. Termination of well-moded programs. *Journal of Logic Programming*, 38(2):243–257, 1999.

[EG99] S. Etalle and M. Gabbrielli. Layered modes. *Journal of Logic Programming*, 39:225–244, 1999.

[Emd81] M. van Emden. AVL tree insertion: A benchmark program biased towards Prolog. *Logic Programming Newsletter 2*, 1981.

[ER98] S. Etalle and F. van Raamsdonk. Beyond success and failure. In J. Jaffar, editor, *Proceedings of the 15th Joint International Conference and Symposium on Logic Programming*, pages 190–204. MIT Press, 1998.

[FGKP85] N. Franchez, O. Grumberg, S. Katz, and A. Pnueli. Proving termination of Prolog programs. In R. Parikh, editor, *Logics of Programs*, pages 89–105. Springer-Verlag, 1985.

[Fit96] M. Fitting. *First-order Logic and Automated Theorem Proving.* Springer-Verlag, 1996.

[GBS95] J. Gallagher, D. Boulanger, and H. Sağlam. Practical model-based static analysis for definite logic programs. In J. W. Lloyd, editor, *Proceedings of the 12th International Logic Programming Symposium*, pages 351–365. MIT Press, 1995.

[GGS99] T. Gabrić, K. Glynn, and H. Søndergaard. Strictness analysis as finite-domain constraint solving. In P. Flener, editor, *Proceedings of the 8th International Workshop on Logic-based Program Synthesis and Transformation*, LNCS, pages 255–270. Springer-Verlag, 1999.

[GL96] J. P. Gallagher and L. Lafave. Regular approximation of computation paths in logic and functional languages. In O. Danvy, R. Glück, and P. Thiemann, editors, *Proceedings of the Dagstuhl Seminar on Partial Evaluation*, LNCS, pages 115–136. Springer-Verlag, 1996.

[GW94] J. P. Gallagher and A. de Waal. Fast and precise regular approximations of logic programs. In P. Van Hentenryck, editor, *Proceedings of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[HACK00] A. Heaton, M. Abo-Zaed, M. Codish, and A. M. King. A simple polynomial groundness analysis for logic programs. Submitted to the Journal of Logic Programming, 2000.

[Hen92] F. Henderson. Strong modes can change the world! Honours report, Department of Computer Science, University of Melbourne, Australia, 1992.

[Hen93] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, 1993.

[HHK97] A. J. Heaton, P. M. Hill, and A. M. King. Analysing logic programs with delay for downward-closed properties. In N.E. Fuchs, editor, *Proceedings of the 7th International Workshop on Logic Program Synthesis and Transformation*, LNCS. Springer-Verlag, 1997.

[Hil93] P. M. Hill. The completion of typed logic programs and SLDNF-resolution. In A. Voronkov, editor, *Proceedings of the Fourth International Conference on Logic Programming and Automated Reasoning*, LNCS, pages 182–193. Springer-Verlag, 1993.

[Hil98] P. M. Hill, editor. *ALP Newsletter*, `http://www-lp.doc.ic.ac.uk/alp/`, February 1998. Pages 17,18.

[HK97] P. M. Hill and A. M. King. Determinacy and determinacy analysis. *Journal of Programming Languages*, 5(1):135–171, 1997.

[HL94] P. M. Hill and J. W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.

[HM99] S. Hoarau and F. Mesnard. Inferring and compiling termination for constraint logic programs. In P. Flener, editor, *Proceedings of the 8th International Workshop on Logic-based Program Synthesis and Transformation*, LNCS, pages 240–254. Springer-Verlag, 1999.

[HT92] P. M. Hill and R. W. Topor. *Types in Logic Programming*, chapter 1, pages 1–61. MIT Press, 1992.

[HWD92] M. Hermenegildo, R. Warren, and S. K. Debray. Global flow analysis as a practical compilation tool. *Journal of Logic Programming*, 13(1-4):349–366, 1992.

[ISO95] International Organization for Standardization. *The ISO Prolog Standard*, 1995. `http://www.logic-programming.org/prolog_std.html`.

[JB92] G. Janssens and M. Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2 & 3):205–258, 1992. First author name erroneously spelt "Janssen".

[Kah96] S. Kahrs. Limits of ML-definability. In H. Kuchen and S. D. Swierstra, editors, *Proceedings of the 8th Symposium on Programming Language Implementations and Logic Programming*, LNCS, pages 17–31. Springer-Verlag, 1996.

[KKS91] M. R. K. Krishna Rao, D. Kapur, and R. K. Shyamasundar. A transformational methodology for proving termination of logic programs. In *Proceedings of the 5th Conference for Computer Science Logic*, LNCS, pages 213–226. Springer-Verlag, 1991.

[Kow79] R. A. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

[KSH99] A. M. King, J.-G. Smaus, and P. M. Hill. Quotienting share for dependency analysis. In D. Swierstra, editor, *Proceedings of the European Symposium on Programming*, 1999.

[KTU93] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, 1993. Title wrongly given in table of contents: Type *recursion* in the presence of polymorphic recursion.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[LS96] N. Lindenstrauss and Y. Sagiv. Checking termination of queries to logic programs. Technical report, Hebrew University of Jerusalem, 1996. Available at `http://www.cs.huji.ac.il/~naomil`.

[LS97] N. Lindenstrauss and Y. Sagiv. Automatic termination analysis of logic programs. In L. Naish, editor, *Proceedings of the 14th Joint International Conference and Symposium on Logic Programming*, pages 63–77. MIT Press, 1997.

[LSS97] N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. TermiLog: A system for checking termination of queries to logic programs. In O. Grumberg, editor, *Proceedings of Computer Aided Verification*, LNCS, pages 444–447. Springer-Verlag, 1997.

[Lüt93] S. Lüttringhaus-Kappel. Control generation for logic programs. In D. S. Warren, editor, *Proceedings of the 10th International Conference on Logic Programming*, pages 478–495. MIT Press, 1993.

[Mar96] M. Marchiori. Proving existential termination of normal logic programs. In M. Wirsing and M. Nivat, editors, *Proceedings of AMAST'96*, LNCS, pages 375–390. Springer-Verlag, 1996.

[Mee88] L. Meertens. First steps towards the theory of rose trees. CWI, Amsterdam; IFIP Working Group 2.1 working paper 592 ROM-25, 1988.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[MK97] J. C. Martin and A. M. King. Generating efficient, terminating logic programs. In M. Bidoit and M. Dauchet, editors, *Proceedings of TAPSOFT'97*, LNCS, pages 273–284. Springer-Verlag, 1997.

[MKS97] J. C. Martin, A. M. King, and P. Soper. Typed norms for typed logic programs. In J. P. Gallagher, editor, *Proceedings of the 6th International Workshop on Logic Program Synthesis and Transformation*, LNCS, pages 224–238. Springer-Verlag, 1997.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[MNL90] K. Marriott, L. Naish, and J. L. Lassez. Most specific logic programs. *Annals of mathematics and artificial intelligence*, 1(2), 1990. Also in proceedings of the 5th Joint International Conference and Symposium on Logic Programming.

[MO84] A. Mycroft and R. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[MS93] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.

[MT95] E. Marchiori and F. Teusink. Proving termination of logic programs with delay declarations. In J. W. Lloyd, editor, *Proceedings of the 12th International Logic Programming Symposium*, pages 447–461. MIT Press, 1995.

[Nai85] L. Naish. Automatic control of logic programs. *Journal of Logic Programming*, 2(3):167–183, 1985.

[Nai86] L. Naish. *Negation and Control in Prolog.* Number 238 in LNCS. Springer-Verlag, 1986.

[Nai92] L. Naish. Coroutining and the construction of terminating logic programs. Technical Report 92/5, University of Melbourne, 1992.

[Nai96] L. Naish. A declarative view of modes. In M. Maher, editor, *Proceedings of the 13th Joint International Conference and Symposium on Logic Programming*, pages 185–199. MIT Press, 1996.

[PBH99] G. Puebla, F. Bueno, and M. Hermenegildo. A framework for assertion-based debugging in constraint logic programming. In A. Bossi, editor, *Pre-Proceedings of the 9th International Workshop on Logic-based Program Synthesis and Transformation*, pages 31–38. Università Cà Foscari di Venezia, 1999. Extended abstract.

[PR99] D. Pedreschi and S. Ruggieri. On logic programs that do not fail. In S. Etalle and J.-G. Smaus, editors, *Proceedings of the Workshop on Verification, organised within ICLP'99*, volume 30 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1999.

[RNP92] Y. Rouzaud and L. Nguyen-Phoung. Integrating modes and subtypes into a Prolog type checker. In K. R. Apt, editor, *Proceedings of the 9th Joint International Conference and Symposium on Logic Programming*, pages 85–97. MIT Press, 1992.

[Rug99] S. Ruggieri. *Verification and Validation of Logic Programs.* PhD thesis, Dipartimento di Informatica, Università di Pisa, 1999.

[SG95a] H. Sağlam and J. P. Gallagher. Approximating constraint logic programs using polymorphic types and regular descriptions. Technical Report CSTR-95-017, University of Bristol, 1995. Presented as a poster at the 7th Symposium on Programming Language Implementations and Logic Programming.

[SG95b] K. Stroetmann and T. Glaß. A semantics for types in Prolog: The type system of PAN version 2.0. Technical report, Siemens AG, ZFE T SE 1, 81730 München, Germany, 1995.

[SHC96] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1–3), 1996.

[SHK98] J.-G. Smaus, P. M. Hill, and A. M. King. Termination of logic programs with `block` declarations running in several modes. In C. Palamidessi, editor, *Proceedings of the 10th Symposium on Programming Language Implementations and Logic Programming*, LNCS. Springer-Verlag, 1998.

[SHK99a] J.-G. Smaus, P. M. Hill, and A. M. King. Mode analysis domains for typed logic programs. In A. Bossi, editor, *Pre-Proceedings of the 9th International Workshop on Logic-based Program Synthesis and Transformation*, pages 163–170. Università Cà Foscari di Venezia, 1999. Extended abstract.

[SHK99b] J.-G. Smaus, P. M. Hill, and A. M. King. Preventing instantiation errors and loops for logic programs with multiple modes using `block` declarations. In P. Flener, editor, *Proceedings of the 8th International Workshop on Logic-based Program Synthesis and Transformation*, LNCS, pages 289–307. Springer-Verlag, 1999.

[SIC98] Intelligent Systems Laboratory, Swedish Institute of Computer Science, PO Box 1263, S-164 29 Kista, Sweden. *SICStus Prolog User's Manual*, 1998. `http://www.sics.se/isl/sicstus/sicstus_toc.html`.

[Sma99] J.-G. Smaus. Proving termination of input-consuming logic programs. In D. De Schreye, editor, *Proceedings of the 16th International Conference on Logic Programming*. MIT Press, 1999.

[Som87] Z. Somogyi. A system of precise modes for logic programs. In J.-L. Lassez, editor, *Proceedings of the 4th International Conference on Logic Programming*, pages 769–787. MIT Press, 1987.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, 1986.

[Str67] C. Strachey. Fundamental concepts in programming languages. Notes for the International Summer School in Computer Programming, Copenhagen, 1967.

[Tho99] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999. Second Edition.

[Tic91] E. Tick. *Parallel Logic Programming*. MIT Press, 1991.

[TL97] J. Tan and I. Lin. Recursive modes for precise analysis of logic programs. In J. Małuszyński, editor, *Proceedings of the 14th International Logic Programming Symposium*, pages 277–290. MIT Press, 1997.

[TZ86] J. Thom and J. Zobel. *NU-Prolog Reference Manual, version 1.0*. Department of Computer Science, University of Melbourne, Australia, 1986. Technical Report 86/10.

[Ued86] K. Ueda. Guarded Horn clauses. In E. Wada, editor, *Proceedings of the 4th Japanese Conference on Logic Programming*, LNCS, pages 168–179. Springer-Verlag, 1986.

[Ued88] K. Ueda. Guarded Horn Clauses, a parallel logic programming language with the concept of a guard. In M. Nivat and K. Fuchi, editors, *Programming of Future Generation Computers*, pages 441–456. North Holland, Amsterdam, 1988.

[UM93] K. Ueda and M. Morita. Message-oriented parallel implementation of Moded Flat GHC. *New Generation Computing*, 11(3):323–341, 1993.

[UM94] K. Ueda and M. Morita. Moded Flat GHC and its message-oriented implementation technique. *New Generation Computing*, 13(1):3–43, 1994.

[VCL95] P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of Prolog using type graphs. *Journal of Logic Programming*, 22(3):179–209, 1995.

Index