A Framework for Incorporating Abstraction Mechanisms into the Logic Programming Paradigm

by

Joseph Lawrence Zachary

This research was supported in part by the National Science Foundation under grant DCR-8411639, and in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-83-K-0125.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts

A Framework for Incorporating Abstraction Mechanisms into the Logic Programming Paradigm

by

Joseph Lawrence Zachary

Abstract

To help make logic programming more suitable for writing large systems, we develop linguistic mechanisms that permit the organization of logic programs around abstractions. In particular, we present the design of Denali, an equational logic programming language that supports predicate and data abstraction.

The key issue in introducing predicate abstraction is dealing with the difference between the declarative and procedural interpretations of logic programs. We address this issue by introducing a two-dimensional type system to describe predicate interfaces. The two components are a sort system and a novel multi-valued mode system. Multi-valued modes constrain the ways in which arguments to predicates may be instantiated. A collection of such modes is defined by the programmer for each sort.

The key issue in introducing data abstraction is providing ways to obtain equational unification procedures. We develop a pragmatic approach that relies upon the programmer to implement these procedures. We facilitate this by supporting a variety of techniques that simplify the problem. Among these techniques are treating unification on a sort-by-sort basis, layering implementations so as to exploit built-in unification procedures, and using the mode system to constrain the unification problem.

Finally, we establish the basis for implementing Denali by developing procedures for performing moded equational resolution and for combining moded equational unification procedures.

Keywords: Logic Programming, Equational Logic Programming, Programming Methodology, Multi-valued Mode, Predicate Abstraction, Data Abstraction, Equational Theory, Moded Equational Resolution, Moded Equational Unification.

This thesis was supervised by John V. Guttag and was submitted to the Department of Electrical Engineering and Computer Science on July 20, 1987, in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science.

To Judy, Baxter, and Big Woofie.

Contents

Figures

Acknowledgments

John Guttag was my research advisor throughout my graduate career, which began long ago in a time when oral exams included questions about cantilevered piles of bricks. His technical insight contributed substantially to the content of the thesis, and his thorough editing improved its presentation. His encouragement and concern kept me on track during the times when my own enthusiasm flagged.

Bill Weihl and Paris Kanellakis were the other members of my thesis committee. They read multiple drafts, constructively criticized my ideas, and helped me discover the larger context of my work.

Kathy Yelick gave invaluable assistance. She collaborated with me on Chapter 6, and was usually the first person with whom I discussed formative ideas. She also reviewed a draft of the formal portions of the thesis.

Steve Garland reviewed drafts of several other parts of the thesis. His technical writing skills and independent viewpoint tightened and focused the presentation.

Judy Zachary sustained me with her love, nurture, and patience throughout the months during which I worked on this thesis, especially in the difficult weeks just prior to its completion.

1 Introduction

In this dissertation we investigate the problem of making logic programming more suitable for programming-in-the-large. We develop linguistic mechanisms that permit the organization of logic programs around abstractions. This makes it possible to apply the software engineering techniques that have been developed in the realm of conventional languages. We illustrate our approach by presenting the design of Denali, an equational logic programming language that supports predicate and data abstraction.

Our work synthesizes two major research directions that have evolved in the area of programming language design in the past fifteen years. Both start from the premise that large programs written in conventional languages are qualitatively more difficult to write than small ones. However, they adopt contrasting approaches to the problem of reliably constructing large software systems. The first approach grants that conventional languages are adequate for constructing small modules, and concentrates on the problem of composing these modules to form larger systems. The second approach emphasizes the need to develop more powerful programming languages and paradigms, and seeks to make it possible to implement large systems with smaller, more easily understood programs.

The first approach is exemplified by programming languages that encourage the organization of programs around independent abstractions, and by specification languages that can be used to describe these abstractions in an implementation independent fashion. Examples of such programming languages abound, and include Simula 67 [Dahl 70], CLU [Liskov 81], Smalltalk [Goldberg 84], and Ada [Barnes 80]. Examples of specification languages, which are hardly less numerous but perhaps less well known, include Larch [Guttag 85], Clear [Burstall 81], Iota [Nakajima 80], and Z [Abrial 80].

A programming method that exploits the distinction between the specification and implementation of an abstraction enjoys two advantages. The implementor of an abstraction needs to know nothing about the program in which the implementation will be embedded, because implementations that are correct relative to a given specification can be freely interchanged without compromising the correctness of the containing program. The client of an abstraction needs to know nothing about the idiosyncrasies of any particular implementation, since the relevant details of its behavior can be determined from its specification.

The second approach is exemplified by the growing class of programming languages whose semantics are based upon formal logics. Prolog [Kowalski 74], the canonical logic programming language, is grounded in first-order predicate logic. Eqlog [Goguen 86] and Tablog [Malachi 86] are both based upon first-order logic with equality.

The logic-based languages enjoy different kinds of advantages. Because their semantics have a direct mathematical basis, the meanings of programs are easily expressed. This means that programs are amenable to formal analysis, reducing the demand upon independent specifications. The problem of combining programs into larger systems is simplified because it is easier to extend these languages to higher-order domains. As a result, implementations are typically smaller than in a conventional language.

Relatively little attention has been paid to the potential benefits of combining these two approaches to programming. We believe that the two schools of thought can be combined to their mutual benefit.

Writing specifications for conventional programs is complicated by the semantics of the languages in which they are written. Though logic-based languages permit more succinct implementations, as programs become larger problems of scale rapidly become paramount. Programs written in logic-based languages would be easier to construct if a specification-based methodology could be adapted to their nature.

We will make the discussion in this dissertation concrete by presenting a design for an equational logic language, called Denali, that incorporates abstraction mechanisms into pure Prolog. Our research proceeds from the assumption that Denali programs will be designed, implemented, and understood using specifications. Our research goals can be cast as design goals for Denali.

First, Denali should be an extension of pure Prolog. Although Denali need not be a superset of Prolog, it should be possible to translate Prolog programs directly into Denali and to interpret them without excessive overhead.

Second, Denali should provide the kinds of abstractions that are appropriate for organizing logic programs. We believe that these abstractions are predicate and data abstractions. These are similar to, but not identical to, the procedural and data abstractions that are appropriate for conventional languages.

Finally, the design of Denali should be pragmatic. We should ensure that Denali can be implemented using existing technology, can be used easily by a programmer, and can be extended conveniently to adapt to future technological advances.

The remainder of this chapter is organized as follows. In Section 1.1 we give the background of our work. We show how the difficulty of organizing Prolog programs around abstractions limits the suitability of Prolog for programming in the large. We also make arguments about the form that predicate and data abstractions should take in a language based upon Prolog.

In Section 1.2 we give a brief overview of Denali and show how it addresses the problems of Prolog. We do this by discussing simple examples of predicate and data abstractions in Denali.

In Section 1.3 we survey related work, and in Section 1.4 we highlight the contributions of our research. We outline the balance of the dissertation in Section 1.5.

1.1 Prolog

Prolog exists in two substantially different forms. For our purposes, Prolog means definite clause programming. Often called pure Prolog, it has an abstract semantics based upon first-order logic [Emden 76], and has a straightforward interpreter based upon depth-first linear resolution. Because of its simplicity and purity, it is the starting point for all other logic programming languages.

In other contexts, Prolog is taken to be the more practical programming language that has evolved over the years. In response to the requirements imposed by applications, a number of extra-logical features have been grafted to the base language. These features encourage a number of programming paradigms that are foreign to the pure subset. [Sterling 86] gives a thorough grounding in the techniques of programming in this language, which we will call standard Prolog.

We use the pure subset of Prolog as the starting point of our language design for three reasons. First, the subset is easy to characterize formally. This is an advantage since we will deal with the formal basis of our extensions. Second, since we are interested only in defining the framework of our new language, we expect that further refinements will be necessary before Denali can be considered a practical language. By extending the pure subset, we can more easily avoid incorporating obstacles to future refinements of Denali. Finally, the specific drawback of pure Prolog that we address in our work—the absence of abstraction mechanisms—is shared by standard Prolog as well.

In Section 1.1.1 we give a summary of pure Prolog. In Sections 1.1.2 and 1.1.3 we explain what we mean by predicate and data abstraction in the context of Prolog. We argue that the addition of these abstraction mechanisms would complement and enhance the essential characteristics of Prolog.

1.1.1 Background

We assume that the reader is already somewhat familiar with Prolog; our goal is to establish a common basis for discussion. We will repeat most of the definitions given below in a more formal context when we discuss moded equational resolution in Chapter 5.

A *term* is either a variable symbol, e.g., N, or a function symbol followed by a sequence of subterms, e.g., $cons(X, nil)$. We distinguish variable symbols by capitalizing them. A *literal* is a predicate symbol that is followed by a sequence of terms, e.g., $size(cons(X, nil), N)$.

It is important to bear in mind the distinction between function and predicate symbols, and, correspondingly, between terms and literals. Function symbols are object constructors, while predicate symbols denote relations defined by the Prolog program in which they appear. Terms are the data objects manipulated by Prolog programs, while literals are invocations of relations.

A *substitution* is a mapping from variables to terms, e.g., $\langle X/0, N/1\rangle$. It is convenient to extend substitutions to mappings from terms to terms and from literals to literals. For example, $\sigma(size(cons(X, nil), N))$, where σ is the substitution given above, is the literal $size(cons(0, nil), 1)$. A substitution σ is called a *unifier* of two terms (or literals) t and r if $\sigma t = \sigma r$.

We say that ρ is an *instance* of σ, and write $\sigma \leq \rho$, whenever σ can be made identical to ρ by instantiating its variables. We say that σ is a most general unifier of a pair of terms if every other unifiers of that pair is an instance of σ.

Solutions to literals are expressed as substitutions. For example, suppose that *append* denotes the relation containing all list triples (L_1, L_2, L_3) such that L_1 and L_2, when appended, form L_3. A solution to the literal $append(cons(1, nil), cons(2, nil), L)$ is the substitution $\langle L/cons(1, cons(2, nil))\rangle$. In general, if P denotes a relation R, then σ is a solution to the literal $P(t_1, \ldots, t_n)$ whenever the tuple $(\sigma t_1, \ldots, \sigma t_n)$ is an element of R.

A query is any sequence of literals. A substitution σ is a solution to the query

$$\leftarrow \mathrm{L_1}, \ldots, \mathrm{L_n}$$

if σ is a solution of each of the literals L_i. For example, a solution to the query

$$\leftarrow \mathrm{append(L_1,\ L_2,\ cons(1, nil)), append(L_1,\ nil,\ nil)}$$

is the substitution $\langle L_1/nil,\ L_2/cons(1, nil)\rangle$.

Since a query can have more than one solution, we are generally interested in obtaining *complete* solutions. A complete solution of a query Q is a set of solutions Σ such that any solution ρ of Q is an instance of some $\sigma \in \Sigma$.

A *definite clause* is a sequence of literals, conventionally written as a reverse implication. The leftmost literal of a clause is its *head*, while the balance of the clause is its (possibly empty) *body*. The following two definite clauses constitute a program that defines the predicate symbol *append*.

$$\mathrm{append}(\mathrm{nil},\ \mathrm{L},\ \mathrm{L}).$$
$$\mathrm{append}(\mathrm{cons}(\mathrm{N}, \mathrm{L}_1),\ \mathrm{L}_2,\ \mathrm{cons}(\mathrm{N}, \mathrm{L}_3)) \leftarrow \mathrm{append}(\mathrm{L}_1,\ \mathrm{L}_2,\ \mathrm{L}_3).$$

The implication symbol is suppressed when the body is empty.

The relation that is associated by a Prolog program with each of its predicate symbols can be determined by regarding the program as a sentence of first-order logic [Emden 76]. Prolog interpreters employ a variant of Robinson's resolution procedure [Robinson 65] to solve queries relative to a program's abstract meaning.

The Prolog interpreter is based upon SLD resolution [Kowalski 71]. Let Q be the query

$$\leftarrow \mathrm{L}_1, \ldots, \mathrm{L}_\mathrm{n},$$

let C be the program clause

$$\mathrm{M} \leftarrow \mathrm{M}_1, \ldots, \mathrm{M}_\mathrm{m},$$

and let σ be a most general unifier of the leftmost literal of the query (L_1) and the head of the clause (M). The query

$$\leftarrow \sigma(\mathrm{L}_2), \ldots, \sigma(\mathrm{L}_\mathrm{n}), \sigma(\mathrm{M}_1), \ldots, \sigma(\mathrm{M}_\mathrm{m}).$$

is the *resolvent* of Q with C using σ.

Solving a query Q involves finding a chain of resolvents that begins at Q and ends with the empty query. The composition of the sequence of substitutions used to form this chain is a solution of Q. The set of all such substitutions is a complete solution of Q.

SLD resolution operates by constructing and searching a tree that contains all possible chains of resolvents. The root node contains the query to be solved; every other node contains a resolvent of its parent. Each branch from a given node corresponds to a program clause that is unifiable with the node's query. The order of the branches depends upon the relative order of the clauses in the program. Because some chains of resolvents are infinite, and because the tree is searched in a depth-first order, SLD resolution does not always find a complete solution.

1.1.2 Predicate abstraction

Conventional programs are organized, at least in part, around implementations of procedures. The values returned and the side-effects generated by the possible invocations of a procedure can be specified independently of any implementation. Such specifications are useful, of course, only when coupled with some means of judging the correctness of an implementation relative to its specification.

The separation of specification and implementation affords significant leverage when constructing and reasoning about programs. The design and implementation phases of program construction can be effectively decoupled. Reasoning about invocations of a procedure can proceed from the specification, which is designed to facilitate reasoning, rather than from the implementation, which is not. Correct implementations can be freely interchanged without compromising the correctness of the containing program.

Predicates in Prolog are analogous to procedures in conventional languages. A Prolog program defines a set of predicates which map input (a tuple of terms) to output (a sequence of substitutions). Literals can be regarded as predicate invocations, and thus are the analogs of procedure invocations in conventional programs. This is called the procedural interpretation of Prolog programs.

The analogy can be stretched no further. If Prolog consistently extended the ideas behind conventionally programming language design, logic programs would be organized around implementations of predicates. Instead, the only unit of encapsulation smaller than an entire program is an individual definite clause.

Predicates are defined in a flat name space that neither circumscribes the scope nor restricts the reuse of predicate names. Since several clauses are typically needed to define each predicate, there is no way to detect inadvertent overloading. The order in which clauses are loaded into the interpreter is significant, and is explicitly relied upon by programmers. As applications become larger, problems of name space management rapidly become critical.

By itself, this defect is easily remedied. It would not be difficult to package all of the clauses needed to define a predicate into a single syntactic unit. However, the absence of an encapsulation mechanism for Prolog predicates is just one manifestation of a larger problem. That problem is the absence of a discipline for treating predicates as independently specifiable components of a larger program.

There is a tendency to view this larger problem as unimportant for the following reasons. Because a Prolog program can be viewed as a sentence of logic, its abstract meaning can be obtained directly. Consequently, it is possible to view a Prolog program as both a specification and an implementation of the same abstraction, namely, some set of relations. This is called the declarative interpretation of Prolog programs.

There are two problems with this rationalization. First, when viewed as a sentence of logic, a Prolog program simultaneously specifies an entire set of predicates. It does not specify the predicates individually, and the relationships between them can be complex.

Second, the meaning of a program when viewed as a specification is different from its meaning when viewed as an implementation. The root of the problem is that the resolution strategy used to solve Prolog queries is incomplete. Given a Prolog program, a complete solution can generally be obtained only for some subset of the possible queries. For programs outside of this subset, the interpreter diverges.

The difference between the two interpretations would not be so serious if it were consistently predictable. In practice, though, two programs that have identical meanings as specifications can have different interpretations as implementations. That is, they can produce complete solutions for differing classes of queries.

An example of two such programs appears below. The predicates *reverse1* and *reverse2* each relate a list to its reverse. The predicate *suffix* relates its first argument, a list, and its second argument, an element, to its third argument, the list with the element appended.

$$\begin{array}{l}
\text{reverse1}(\text{cons}(X, L_1), L_2) \leftarrow \text{reverse1}(L_1, L_3), \text{suffix}(L_3, X, L_2).\\
\text{reverse1}(\text{nil}, \text{nil}).\\
\text{reverse2}(L_1, \text{cons}(X, L_2)) \leftarrow \text{reverse2}(L3, L2), \text{suffix}(L_3, X, L_1).\\
\text{reverse2}(\text{nil}, \text{nil}).\\
\text{suffix}(\text{nil}, X, \text{cons}(X, \text{nil})).\\
\text{suffix}(\text{cons}(Y, L_1), X, \text{cons}(Y, L_2)) \leftarrow \text{suffix}(L_1, X, L_2).
\end{array}$$

While the two reversal predicates possess identical meanings as specifications, they exhibit incompatible behavior for some queries. Unless the first argument to a *reverse1* literal is fully instantiated, an attempted solution of the literal diverges. Conversely, *reverse2* requires that its second argument be fully instantiated. The query

$$\leftarrow \text{reverse1}(\text{cons}(1, \text{cons}(2, \text{nil})), L)$$

can be solved, while the query

$$\leftarrow \text{reverse2}(\text{cons}(1, \text{cons}(2, \text{nil})), L)$$

cannot.

These two implementations, while they ostensibly satisfy the same specification, are not interchangeable. The problem is that not all of the necessary interface information, i.e., the degree to which arguments must be instantiated, is specified. In practice, Prolog programmers construct programs and queries so that implicit constraints such as the above are not violated. However, if a specification methodology such as the one that has grown up around procedural languages is to be followed, the limitations upon the use of predicates should be explicit in both implementation and specification.

1.1.3 Data abstraction

Conventional programming languages provide a variety of built-in scalar and composite data types, such as integers, records, and arrays. The differing data types are intended to provide expressive power and efficiency. This variety can be exploited by a programmer when designing the data structures for an application. Finding appropriate representations for data is a major part of the engineering effort required in writing a conventional program.

This form of engineering is not possible in Prolog. There is but a single built-in data type, uninterpreted terms, which are equivalent to immutable trees. Although all representations can be coded into this form, uninterpreted terms are deficient in many situations. For example, an efficient implementation of a hash table, which requires a data structure (such as an array) that provides random access in constant time, is not realizable in Prolog.

The absence of a variety of data types from Prolog is a serious drawback. The addition of new built-in data types to Prolog, however, is not straightforward. We must consider the generic characteristics exhibited by the uninterpreted term in its role as the sole Prolog data structure, and be careful to preserve these characteristics as we add new types.

There are three characteristics of Prolog terms that should be preserved by any other built-in type that is added to the language. First, terms can contain and can even be variables. Second, pairs of terms can be unified to produce substitutions. Third, terms can be denoted directly and can be written into programs. We will consider each of these points more closely.

The primary source of the expressive power of Prolog is its treatment of variables and variable-containing terms as first-class objects. This enables a number of programming paradigms that are unique to Prolog. The exploitation of these paradigms ought to be possible even when objects other than uninterpreted terms are added to the language. Any data types that are added must provide variable and variable-containing objects.

Unification is central to the interpretation of Prolog programs, much as parameter passing is central to conventional languages. The unification algorithm used in Prolog is a consequence of the fact that Prolog terms are uninterpreted. Syntactically distinct terms are always treated as being semantically distinct.

This form of unification is not appropriate for data types in which relationships exist among objects. These relationships must be taken into account by the unification algorithm. For example, assuming the normal interpretation given to natural numbers, the numbers $X+\mathit{2}$ and $\mathit{3}$ should be unifiable by the substitution $\langle X/\mathit{1}\rangle$.

If multiple data types are added to Prolog, the unification algorithm can no longer be viewed as a fixed component of the interpreter. Instead, each data type must provide an appropriate unification algorithm.

Not all objects in conventional languages possess denotations. For example, arrays in some languages cannot be written directly into the program text, but must instead be obtained indirectly through the evaluation of procedure invocations. In Prolog, however, every object can be denoted directly. This is a crucial aspect of the language's expressive power.

The means of denoting the objects provided by a data type should be regarded as part of that data type's interface, and should be independent of implementation details. For example, we denoted natural numbers earlier using term notation, but no reasonable implementation would represent them that way.

This completes our brief survey of the properties that might be expected of any data types added to Prolog. (We will pursue this topic further in Chapter 4.) The deficiency of Prolog, however, goes beyond the paucity of built-in data types. Most modern languages also support the definition, by the programmer, of abstract data types.

Experience in conventional languages has proven the value of organizing programs around data abstractions. Together with procedural abstractions, data abstractions provide a convenient means of specifying and reasoning about programs. If additional built-in data structures are to be added to Prolog, so should a mechanism for the creation of user-defined data types. These user-defined types should exhibit the properties outlined above.

1.2 Denali

Denali programs consist of implementations of predicate and data abstractions. In this section we will give some simple examples that illustrate the form such abstractions take in Denali. We will use these examples to illustrate some of the issues that we have raised and resolved in designing Denali.

1.2.1 Predicate abstraction

We begin with an example of an implementation of a predicate abstraction. The predicate *half* relates pairs of natural numbers in which the first is twice the second.

$$\begin{array}{l} \text{half} = \textbf{pred}\ (\text{nat}, \text{nat})\ \textbf{moding}\ (\text{gnd}, \text{any}), (\text{any}, \text{gnd}) \\ \quad \text{half}(0, 0). \\ \quad \text{half}(\text{s}(\text{s}(\text{N})), \text{s}(\text{M})) \leftarrow \text{half}(\text{N}, \text{M}). \end{array}$$

The body of this predicate is composed of two definite clauses. The header of the predicate, and the way that it is used, is more novel. In addition to the name of the predicate, the header gives sort and mode restrictions. These restrictions constrain the formation and solution of literals headed by *half*.

The sort symbol *nat* and the mode symbols *gnd* and *any* are not predefined. They are defined by an implementation of the data abstraction that we will examine below.

Sort restrictions are comparable to the type restrictions of conventional languages. They control the class of arguments that can be used to compose literals. In the example above, the sort restriction requires that both arguments to *half* be natural numbers. Sort restrictions are static, so they can be decided before runtime.

Mode restrictions augment the sort restrictions by further constraining the form of arguments. They control the degree to which the arguments must be instantiated. The mode restriction expressed above, for example, requires that at least one of the arguments to *half* be a variable-free natural number.

Some existing logic languages also have mode systems, but they distinguish only between variables and non-variables. We will call these bi-valued mode systems. The more expressive Denali modes are defined by the programmer on a sort-by-sort basis, with different modes used to constrain the objects of different sorts. Consequently, they can make finer sort-specific distinctions. We will term the Denali approach a multi-valued mode system.

The incorporation of multi-valued mode restrictions into the interfaces of predicates is designed to make it possible to realize predicate abstraction in Denali. Mode restrictions must be given in specifications as well as in implementations. They specify the portion of the well-sorted domain in which completeness is required of an implementation.

Unlike well-sortedness, which is a static property, well-modedness is dynamic and cannot be tested until runtime. This is because the degree to which the terms of a literal are instantiated can change as substitutions are applied. Mode restrictions are exploited by Denali to control the order in which the literals of queries are solved. The evaluation order is moded-first rather than leftmost-first.

1.2.2 Data abstraction

The interface of a data abstraction appears below. In this chapter, we give only the interface because we have not yet developed enough background to describe an implementation.

```
nat = cluster
    denoted by
        0 :→ nat
        s : nat → nat
        + : nat, nat → nat
    modes any > gnd
    unified by
        X+0 = X
        X+Y = Y+X
        s(X)+Y = s(X+Y)
    square = pred (nat, nat) moding (gnd, any)
    end
```

Each data abstraction introduces a sort name and provides a set of objects of that sort. The interface of a data abstraction has four components which, when implemented, provide different ways of manipulating the objects. These components are

- a grammar for denoting the objects as terms in programs,
- a set of modes suitable for constraining predicates that use the objects,
- a procedure for unifying pairs of the objects, and
- a set of predicates defined over the objects.

It is instructive to compare data abstraction in Denali with data abstraction in conventional languages. Data abstractions in conventional languages typically provide a set of procedures defined over a set of objects. If the abstraction is implemented as a built-in type, the abstraction will also usually provide a means of directly denoting objects. Such a facility is usually not available for user-implemented abstractions. Modes and unification procedures, since they are pertinent only in logic languages, are not provided.

An implementation of a data abstraction in Denali is called a *cluster*, and can be either built into the language or defined by the programmer. An implementation, as in a conventional language, fixes a concrete representation for the abstract objects it provides. It then implements the components of the interface in terms of this internal representation. In a well-constructed implementation, the representation details are not visible to users of the abstraction.

The placement of the denotation scheme in the interface, independent of the representation scheme, is a significant design decision. In the implementation of an abstraction, a mapping must be defined from denotations to representations. Before a program can be evaluated, this mapping must be used to translate all denotations into representations. This discipline is necessary to ensure that the users of an abstraction need not be concerned with its implementation details.

If the denotation and representation were not distinct, changes to the representation would require that modifications be made to each client of the abstraction. The design of a denotation and the choice of a representation must satisfy different criteria. This mandates that they be separable.

The choice of which modes and predicates to provide in the interface of a data abstraction is based upon knowledge of how the abstract objects will be used. This is analogous to how the procedures provided in a conventional data abstraction are chosen. Modes cannot be defined outside a cluster since their implementations must be privy to representation information. Predicates can be defined either inside or outside a cluster; however, only those predicates defined inside can exploit knowledge of the representation.

The equations specify the functional relationships between the symbols of the denotation. The unification procedure provided by the implementation must treat as equal pairs of terms that can be proven equal using the equations. Unification based upon equations in this fashion is called *equational unification*.

The unification procedure for each sort is written as a binary predicate over pairs of abstract objects. Simplifying the problem of implementing unification predicates for arbitrary abstract types is a critical facet of the design of Denali. A large body of research addresses the problem of equational unification. Equational unification algorithms are known for a variety of sets of equations that express general properties such as associativity and commutativity. For the most part, however, these algorithms are inaccessible to the non-specialist.

The body of existing equational unification algorithms is a useful source for the unification predicates of built-in abstractions. It is of little help, however, for the programmer implementing a cluster. Three aspects of Denali help simplify the problem faced by the programmer.

First, because cluster implementations are based upon concrete representations, unification predicates need not be constructed from first principles. Denali can provide a rich variety of built-in data abstractions that exhibit diverse behavior with respect to unification. With an appropriate choice of representation, a programmer-defined implementations of unification can be based upon an existing implementation that is already close to the required form. The possibility of building upon existing implementations of unification is a direct consequence of the separation of denotation and representation concerns.

Second, user-defined unification procedures are implemented on a sort-by-sort basis. The individual procedures are ultimately combined by the Denali interpreter to obtain an overall unification algorithm. This reduces the burden on the programmer by factoring the task into more manageable units.

Third, just as with ordinary predicates, modes can be used to restrict the interfaces of unification predicates. This can eliminate the necessity of defining the predicate over troublesome parts of its domain. As an extreme example, restricting unification to pairs of ground terms reduces the problem to an equality test modulo the set of equations. Of course, this restriction also sacrifices a large degree of the expressive power of unification. In general, the imposition of mode restrictions upon unification represents a tradeoff between ease of implementation and expressive power.

1.3 Related work

In this section we summarize some of the research that is related to the work reported in this dissertation. This related work falls into two broad categories: logic languages that exploit modes and logic languages that incorporate equality. Our intention here is to provide an outline; everything mentioned here is described in greater detail in later chapters.

We have briefly outlined the differences between Denali's multi-valued mode system and the simpler bi-valued mode systems of existing languages. At least four applications have been found for these bi-valued modes. In the Edinburgh Prolog compiler, modes are used to annotate predicate definitions to permit more efficient compilation [Warren 77]. Several sequential logic languages use modes to control the order of evaluation of literals, including Epilog [Porto 82] and Mu-Prolog [Naish 85]. The parallel logic languages Parlog [Clark 85] and Concurrent Prolog [Shapiro 83] use modes for concurrency control. Finally, modes are used to help plan the evaluation of database queries in NAIL! [Ullman 85]. A more detailed discussion of these uses of bi-valued modes appears in Chapter 2.

We have also discussed our pragmatic approach to incorporating equality. It involves providing a collection of linguistic mechanisms that permit the programmer to restrict and thus simplify the equational unification problem. Of the other logic languages that incorporate equality, only Kornfeld's extension to Prolog [Kornfeld 86] adopts a comparably pragmatic approach. His approach is to permit the implementation of equality procedures in Lisp. Eqlog [Goguen 86] allows the programmer to specify equality with a set of equations; the language implementation is responsible for synthesizing an equational unification procedure from these equations. Tablog [Malachi 86] is based upon a proof system, completely independent of resolution, that explicitly incorporates both definite clauses and equations. SLOG [Fribourg 84] is based upon definite clauses that define equality, and is implemented with a variant of resolution. We discuss these languages further in Chapter 3.

1.4 Contributions

In this section we summarize the primary contributions presented in this dissertation.

We identify the two forms of abstraction—predicate and data—around which we believe logic programs should be organized, and show how programs can be constructed using them. In most logic languages, programs are composed of individual definite clauses and, sometimes, equations. In Denali, they are composed of implementations of predicate and data abstractions. Two of our key contributions are the idea that the interfaces of predicate abstractions should express multi-valued mode restrictions upon arguments, and the idea that the language should make it possible for the programmer to implement equational unification procedures.

The mode system of Denali is more pervasive than that of other logic languages. In other languages, modes are used only as annotations that help control the order of interpretation. In Denali, modes are exploited in almost all aspects of an implementation. In addition to helping control the interpreter, modes document predicate interfaces, serve as guards of clauses, and help simplify the implementation of unification by restricting the formation of objects. The runtime checking of mode restrictions serves to catch programming errors that would otherwise be undetected.

Denali modes are also more expressive than those of other languages. Existing languages provide modes that distinguish only between variables and non-variables. These bi-valued modes are generic to all types of objects, and thus can be built directly into the language. Denali's multi-valued modes can express the finer-grained distinctions that are needed to fully document predicate interfaces. Because the distinctions that are required depend upon the application, Denali modes are defined by the programmer.

Denali is the first logic language that distinguishes between the way abstract objects are denoted and the way that they are represented. This separation makes it possible to build programs in layers of abstractions, as in conventional programming languages. It also makes it possible to engineer representations.

We adopt a novel approach to obtaining implementations of equational unification. Other languages attempt to handle unification automatically by synthesizing implementations from equations. The known approaches have limited applicability and almost always produce inefficient implementations. Furthermore, there are theoretical limitations upon how well any such approach can ever perform. In Denali, we place the burden of implementing unification procedures upon the programmer. To make this approach feasible, we place at the disposal of the programmer a number of techniques of restricting and thus simplifying unification. Unification procedures are defined on a sort-by-sort basis and then combined by the implementation. Because implementations of abstractions can be layered, unification procedures provided by built-in abstractions can be incorporated into user-defined implementations. Most importantly, modes can be used both to place interface restrictions upon unification procedures and to restrict the formation of objects.

Besides presenting a language design, we also establish the formal basis for Denali. We define a new form of resolution, moded equational resolution, and use it to define the semantics of Denali and provide the basis for constructing specifications and defining satisfaction for Denali programs. By extending an existing algorithm for combining unification algorithms, we establish the cornerstone of a Denali interpreter.

1.5 Roadmap

The remainder of this dissertation is composed of seven chapters. Chapters 2, 3, and 4 comprise an informal development of Denali, while Chapters 5 and 6 deal rigorously with moded equational resolution and moded equational unification. In Chapter 7 we draw the five preceding chapters together by developing a formal basis for Denali. We then conclude in Chapter 8.

In Chapter 2 we describe Denali's predicate abstraction mechanism, which provides a way to constrain the interfaces of predicate implementations. We develop our system of defining and using multi-valued modes, and contrast it with the bi-valued modes used in various existing logic languages.

In Chapter 3 we discuss how term equality is handled in Denali. As in other logic languages that permit the imposition of equality constraints, our method is centered around equational unification. Our mode system, however, figures prominently in constraining the problem of equational unification. The introduction of mode constraints in this context requires the development of moded equational unification and resolution.

In Chapter 4 we describe Denali's data abstraction mechanism. In addition to developing the composition of data abstractions, we give two ways of implementing them, called implicit and explicit implementations. In implicit implementations the denotation doubles as the representation, and the unification predicate is specified rather than implemented. This technique is applicable only when an algorithm for the specified unification predicate is known to the language implementation. Explicit implementations, on the other hand, are based upon concrete representations.

In Chapter 5 we develop a formal basis for moded equational resolution. Beginning with linear equational resolution, we consider a series of variations on the selection rule. This process culminates with a selection rule, based upon modes, that forms the basis of the semantics of Denali.

In Chapter 6 we consider the subsidiary problem of moded equational unification. We review the existing techniques for obtaining equational unification algorithms, and sketch how they can be extended to deal with modes. We then extend Yelick's algorithm [Yelick 85] for combining unification algorithms for disjoint sets of equations into an overall unification algorithm. The resulting extended combining procedure, which exploits sorts and modes, is suitable for use in the Denali abstract interpreter.

In Chapter 7 we present a more rigorous development of Denali than was possible in Chapters 2 through 4. It includes a formal semantics based upon the moded resolution developed in Chapter 5, and an abstract interpreter based upon the combining procedure of Chapter 6.

We conclude in Chapter 8 by highlighting the contributions of our work and suggesting avenues for further research. The most prominent of these is an implementation of Denali.

2 Predicate abstraction in Denali

In this chapter we informally describe Denali's predicate abstraction mechanism. We illustrate the changes that must be made to pure Prolog and its interpreter to support this form of abstraction. We develop the changes incrementally, and gradually evolve away from Prolog to a subset of Denali.

We begin in Section 2.1 by discussing the structure of the predicate abstraction mechanism. The addition of a two-dimensional type system for describing predicate interfaces is of primary interest. This system includes both sort restrictions, which are common in most type systems, and mode restrictions, which are meaningful only in logic languages.

We will assume that each predicate possesses an independent specification as a first-order relation. Mode restrictions can be used by a programmer to restrict the domain of a predicate to a subset for which it can be effectively computed. This makes it possible to precisely implement a given specification. We thus explicitly recognize and deal with a pervasive limitation of logic languages: only sufficiently instantiated queries can be solved by a practical interpreter.

Mode restrictions, unlike sort restrictions, can be checked only at runtime. In Section 2.2 we show how the Prolog interpreter can be modified to cope with, and even exploit to advantage, the mode restrictions associated with each predicate interface.

The mode restrictions that we study initially are expressed using bi-valued modes. Similar kinds of mode restrictions have appeared previously in a number of logic languages. Although they are useful for illustrating the structure of our type system, bi-valued modes are not expressive enough for our purposes. In Section 2.3 we show why, and replace them with multi-valued modes.

The most important contribution of this chapter is our development of Denali's multi-valued mode system and our description of how it is used. In Section 2.4 we summarize the mode-related definitions that are introduced throughout the chapter.

We show how multi-valued modes can be syntactically defined in Section 2.5. We conclude the chapter in Section 2.6 by discussing the body of related work that has developed and exploited bi-valued modes.

2.1 Predicate interfaces using bi-valued modes

Any two Denali predicate implementations that satisfy the same specification can be freely interchanged without affecting the remainder of the program in which they appear. We achieve this property by using the type system to express interface restrictions upon predicates. These restrictions can be used to limit predicates to domains over which complete implementations are realizable. By making the restrictions part of the specification, and by appropriately defining satisfaction, implementation transparency is obtained.

In this section we develop a preliminary version of the two-dimensional Denali type system. We describe the first dimension, a sort system, in Section 2.1.1. The sort system is comparable to the type systems of most typed languages. The terms manipulated by programs are divided into disjoint sets that are named by sorts. These sorts are used in turn to give signatures to predicates. Predicates need be defined only over well-sorted queries. Denali sort restrictions are static and can thus be verified before runtime.

We have seen that the Prolog evaluation strategy is incomplete for certain queries. In general, the less fully instantiated a query, the less likely it is that the query can be completely solved. The second component of the type system is a mechanism for imposing restrictions in the form of constraints upon the degree to which a literal must be instantiated before it can be evaluated.

Such restrictions are called mode restrictions. In Denali, a mode restriction is associated with each predicate symbol as part of its type interface. A literal cannot be evaluated until it satisfies the mode restriction associated with its head predicate symbol. Since the degree of instantiation of a term can change under the application of substitutions, mode restrictions must be checked at runtime.

In Section 2.1.2, we illustrate this second component of the Denali type system by describing how bi-valued modes can be used to express mode restrictions. Bi-valued modes are the simplest possible form of mode, and are the kind that have appeared previously in, among others, the Warren Prolog compiler [Warren 77], an extended version of Prolog called Epilog [Porto 82], and a concurrent logic language called Concurrent Prolog [Shapiro 83].

2.1.1 Sorts

A Denali program contains a set of encapsulated predicate implementations. An implementation consists of a sorted, moded header and a list of defining Horn clauses. We will ignore the moded portion of headers for now.

A *length* predicate, for example, is presented as:

$$\begin{array}{l} \mathrm{length} = \mathbf{pred}\ (\mathrm{list}, \mathrm{nat}) \\ \quad \mathrm{length}(\mathrm{nil}, 0). \\ \quad \mathrm{length}(\mathrm{cons}(\mathrm{X}, \mathrm{L}), \mathrm{s}(\mathrm{N})) \leftarrow \mathrm{length}(\mathrm{L}, \mathrm{N}). \end{array}$$

The header specifies that any literal headed by *length* must have two argument terms. The first must have sort *list* and the second must have sort *nat*.

Sorts are associated with terms by explicitly declaring the signatures of function symbols in the usual way, e.g.,

$$\begin{array}{ll} 0\!: \rightarrow \mathrm{nat} & \mathrm{nil}\!: \rightarrow \mathrm{list} \\ \mathrm{s}\!: \mathrm{nat} \rightarrow \mathrm{nat} & \mathrm{cons}\!: \mathrm{nat}, \mathrm{list} \rightarrow \mathrm{list}. \end{array}$$

Terms must be constructed consistently with the signatures. For simplicity, variables are not declared, though they must be used consistently within each clause.

So long as it is well-sorted, the meaning of a Denali program as described thus far is identical to the pure Prolog program obtained by deleting the function symbol declarations and predicate headers.

The Denali sort mechanism provides a syntactic check upon the structure of programs, makes possible the construction of a complementary mode system, and is the basis of data abstraction. However, it imposes restrictions not present in Prolog and is probably too restrictive for practical use. Modern polymorphic type systems such as that used in ML [Milner 78] would be more appropriate to Denali. As we are primarily interested in the more novel aspects of Denali, we have not incorporated these ideas here. There is no apparent reason, however, why the sort system could not be extended.

2.1.2 Bi-valued mode systems

Not all literals can be solved to yield a manageable sequence of substitutions within a reasonable amount of time. Some evaluations diverge by neither yielding a substitution nor terminating. Other evaluations are inefficient, yield a large or infinite number of substitutions, or both. Conscientious Prolog programmers must be careful to avoid the construction of programs or queries that permit no reasonable solution.

Consider the predicate *length* defined above. Using it, an interpreter can successfully solve any of the three queries

$\leftarrow \mathrm{length}(\mathrm{nil}, 0)$,
$\leftarrow \mathrm{length}(\mathrm{cons}(0, \mathrm{nil}), \mathrm{N})$, or
$\leftarrow \mathrm{length}(\mathrm{L}, \mathrm{s}(\mathrm{s}(0)))$,

producing a single substitution. However, if both of the arguments are uninstantiated, as in

$\leftarrow \mathrm{length}(\mathrm{L}, \mathrm{N})$,

an attempted evaluation will generate an infinite sequence of substitutions. Although it depends upon the circumstances, this behavior is usually unacceptable.

Some logic languages address this and related problems by providing bi-valued modes. These modes allow the programmer to specify restrictions, enforced by the interpreter, that prevent the evaluation of insufficiently instantiated literals. Mode restrictions can also be exploited by the interpreter to control the order of evaluation of the literals of a query.

A *mode* is a set of terms. Set membership depends upon structural properties, such as whether or not a term is a variable. The two modes of our bi-valued mode system are *all*, which contains all of the terms, and *nonVar*, which contains only non-variable terms. (This particular system is a composite example, but it is representative of existing bi-valued mode systems.) We will refer to the modes of a bi-valued system as *bi-valued modes*.

We say that a term t has mode M, and write $M(t)$, if $t \in M$. A term can have more than one mode. In our bi-valued system, for example, any non-variable term has both modes, *all* and *nonVar*.

One or more n-tuples of modes are associated with each n-ary predicate symbol. A literal $P(t_1, \ldots, t_n)$ is *well-moded* if P is associated with a mode tuple $(M_1, \ldots, M_n)$ such that $\forall i\, M_i(t_i)$. The set of all tuples of terms $(r_1, \ldots, r_n)$ such that $P(r_1, \ldots, r_n)$ is well-moded is called the *moding* of P.

Mode tuples, and thus modings, are associated with Denali predicates by annotating their headers. For example, the moding of *length* is given by writing:

$$\mathrm{length} = \mathbf{pred}\ (\mathrm{list}, \mathrm{nat})\ \mathbf{moding}\ (\mathrm{nonVar}, \mathrm{all}), (\mathrm{all}, \mathrm{nonVar}).$$

This specifies that a well-moded *length* literal must contain at least one non-variable term.

Only a single change need be made to the semantics of Denali to enforce mode restrictions. If an ill-moded literal is selected for reduction, the interpreter can fail immediately. Such a failure is termed a *mode failure*. For example, consider the query

$$\leftarrow \mathrm{length}(\mathrm{cons}(0, \mathrm{L}), \mathrm{N}).$$

The literal is well-moded, so one step of reduction leads to the query

$$\leftarrow \mathrm{length}(\mathrm{L}, \mathrm{M}).$$

The remaining literal satisfies neither of the mode tuples of *length*, so the interpreter reports a mode failure. This prevents the generation of the undesired infinite sequence of substitutions.

Mode failures are distinct from the overlap failures that occur in resolution-based interpreters. A mode failure is the run-time discovery that the original query was insufficiently instantiated to permit solution. It is a hard error, corresponding to a runtime type error, and cannot be recovered. An overlap failure, which occurs when a selected literal unifies with no clause, is a normal part of interpretation. It is recovered through backtracking.

2.2 Moded evaluation

As described so far, mode restrictions serve only to constrain the interpreter. In this section we describe two ways in which Denali and its interpreter can be made more robust by exploiting mode restrictions.

An attempt to evaluate an ill-moded literal leads to a runtime failure from which recovery is not possible. Fortunately, mode restrictions can be exploited by an interpreter to improve the order of evaluation of queries. We describe in Section 2.2.1 how the leftmost-first selection rule of Prolog can be replaced with a selection rule that dynamically alters the selection order so that the evaluation of ill-moded literals is deferred until they become sufficiently instantiated. This reduces the incidence of mode failures.

Modes can also be used to affect which of the multiple clauses of a predicate definition can be used to reduce a given literal. We describe mode guards, which make this possible, in Section 2.2.2.

2.2.1 Moded selection

The Prolog strategy for evaluating a query involves solving literals in a strict left-to-right order. We relax this strategy in Denali by always solving instead the leftmost well-moded literal. This has two effects. First, mode failures occur less frequently since they occur only when none of the literals of a query is well-moded. Second, some predicate implementations are rendered more robust, since they can tolerate a greater diversity in the modes of their arguments.

We illustrate these points with the following example.

$$\begin{array}{l} \text{double} = \textbf{pred}\ (\text{nat}, \text{nat})\ \textbf{moding}\ (\text{nonVar}, \text{all}), (\text{all}, \text{nonVar}) \\ \quad \text{double}(0, 0). \\ \quad \text{double}(\text{s}(\text{N}), \text{s}(\text{s}(\text{M}))) \leftarrow \text{double}(\text{N}, \text{M}). \end{array}$$

Suppose that we pose the following query, with the goal of finding the natural number K that can be quadrupled to obtain *0*:

$$\leftarrow \text{double}(\text{K}, \text{L}), \text{double}(\text{L}, 0).$$

Under the original leftmost-first selection rule, the evaluation would fail immediately since the first literal is ill-moded. Under the alternative semantics, however, the second literal can be selected for reduction, leading after one step to

$$\leftarrow \text{double}(\text{K}, 0).$$

Because the variable L has been instantiated, the remaining literal is now well-moded and can be reduced, obtaining the solution substitution $\langle K/0 \rangle$.

We could equally well specify that the selection rule select an arbitrary well-moded literal for reduction. We specify that the leftmost well-moded literal be chosen so that the reduction order for any particular query is deterministic.

Prolog's interpreter is based upon SLD resolution [Kowalski 71], which observes an invariant that must be respected under the new evaluation strategy. This invariant requires that a literal, once selected, be completely reduced before any other literal of the query is selected.

In Prolog, a literal that has been selected for reduction is replaced in the query by its resolvent. This fact, combined with left-to-right evaluation, guarantees that the invariant will hold. If this strategy of replacing a literal with its resolvent to obtain a new query were to be followed in Denali, however, the invariant would be breached, since the selection rule is different.

This observation can be illustrated with the following example. Using the *length* predicate defined above, and the *nonzero* predicate defined here,

$$\begin{array}{l} \text{nonzero} = \textbf{pred}\ (\text{nat}, \text{list})\ \textbf{moding}\ (\text{nonVar}, \text{all}) \\ \quad \text{nonzero}(\text{s}(\text{X}), \text{nil}). \end{array}$$

we can pose the following query:

$$\leftarrow \text{length}(\text{cons}(0, \text{L}), \text{N}), \text{nonzero}(\text{N}, \text{L}).$$

Since only the first literal is well-moded, one step of reduction can lead to

$\leftarrow \mathrm{length}(\mathrm{L}, \mathrm{M}), \mathrm{nonzero}(\mathrm{s}(\mathrm{M}), \mathrm{L}).$

Now, even though the reduction of the original literal is incomplete, the *nonzero* predicate has been sufficiently instantiated to be solved, leading to

$\leftarrow \mathrm{length}(\mathrm{nil}, \mathrm{M}).$

We can now complete the solution of the originally reduced literal.

It may seem at first that violating the invariant in this way is beneficial since it permits the solution of queries that would otherwise meet with mode failure. The query above, for example, could not be solved if the original *length* literal had to be completely solved before the *nonzero* literal could be solved. Unfortunately, violating the invariant compromises the integrity of predicate abstraction in Denali.

The problem is that predicates that satisfy the same specification can exhibit different behaviors with respect to the production of intermediate substitutions. If an interpreter is allowed to exploit these intermediate substitutions, then differences between two implementations can be detected.

Suppose, for example, that *length* were defined in the following way:

length = **pred** (list, nat) **moding** (nonVar, all), (all, nonVar)
  length(nil, 0).
  length(cons(X, L), N) ← length(L, M), increment(M, N).
increment = **pred** (nat, nat) **moding** (nonVar, all)
  increment(N, s(N)).

This definition of *length* behaves identically to the original implementation, except that no intermediate substitutions are generated. Hence, the query posed above, after one reduction, would stand at:

$\leftarrow \mathrm{length}(\mathrm{L}, \mathrm{M}), \mathrm{increment}(\mathrm{M}, \mathrm{N}), \mathrm{nonzero}(\mathrm{N}, \mathrm{L}).$

Since each of the literals is ill-moded, a mode failure would occur at this point.

Adopting the permissive evaluation strategy sketched above would require that the mode system document the possibilities for the partial reduction of a literal. Otherwise, implementation differences in predicates satisfying the same specification could be detected. Rather than complicate the mode system, we take a conservative approach and forbid the adoption of the permissive evaluation strategy. Instead, we require that the resolvent spawned by the reduction of a literal in a query be solved as a subquery before any of the other literals in the original query are reduced.

Notice that although we have avoided the necessity of specifying the modes of terms contained in intermediate substitutions, nowhere in a predicate interface do we specify the modes of terms appearing in result substitutions. It thus appears that differences in the implementations of predicates could be detected by observing the modes of the substitutions they produce. Fortunately, this is not the case. We will show in Chapter 5 that the modes of the substitutions produced by a correct implementation are a function of its inputs, and thus are indirectly fixed by its specification.

2.2.2 Mode guards

Denali's moded selection rule makes it easier to define predicates that operate over a wide range of modes. This can be illustrated with the predicate *twiceLength*:

$$\begin{array}{l} \mathrm{twiceLength} = \textbf{pred}\ (\mathrm{list}, \mathrm{nat})\ \textbf{moding}\ (\mathrm{nonVar}, \mathrm{all}), (\mathrm{all}, \mathrm{nonVar}) \\ \quad \mathrm{twiceLength}(\mathrm{L}, \mathrm{N}) \leftarrow \mathrm{length}(\mathrm{L}, \mathrm{M}), \mathrm{double}(\mathrm{M}, \mathrm{N}). \end{array}$$

This implementation is workable only because the order in which the literals of the defining clause are evaluated depends upon the modes of the arguments. Not all predicate definitions are readily invertible, so it is not always possible to successfully implement predicates in this way. It is sometimes necessary to use completely different approaches depending upon the modes of the arguments.

Mode guards, a control primitive provided by Denali, make this possible. The clauses that define a Denali predicate can compose either a *guarded* or an *unguarded* block. All of the predicates that we have presented thus far have been implemented with unguarded blocks. When a literal is reduced by an unguarded predicate implementation, each of the clauses in the block is considered in turn from top to bottom. Top to bottom evaluation is specified to make evaluation orders predictable, but is not crucial.

Each clause of a guarded block is prefixed by a mode tuple whose arity equals the arity of the containing predicate. These tuples, or mode guards, are used to arbitrate which clause is used to reduce a literal. Only a single clause is used: the topmost clause whose mode guard is satisfied by the arguments to the literal.

Consider, for example, the following predicate for squaring natural numbers.

square = **pred** (nat, nat) **moding** (nonVar, all), (all, nonVar)
(nonVar, all): square(N, M) $\leftarrow$ times(N, N, M).
(all, nonVar): square(N, M) $\leftarrow$ newtonMethod(M, N).

A literal formed from the *square* predicate is reduced with the upper clause if its first argument is ground, and by the lower clause otherwise.

The moded selection rule and mode guards possess complementary properties. Both support the definition of predicates that have flexible mode restrictions. The selection rule provides the flexibility needed when a single approach is sufficient for solving a problem for all possible modings. Mode guards provide the flexibility needed when different approaches must be taken for differently moded arguments.

2.3 Predicate interfaces using multi-valued modes

Under an appropriately expressive system of modes, it should be possible to find an acceptable solution to a well-moded literal with a reasonable amount of computation. It is up to the programmer to determine which solutions are acceptable, to decide how much computation is reasonable, and to place mode restrictions upon predicates accordingly. It is up to the language designer to ensure that the programmer can express sufficiently flexible restrictions.

Bi-valued modes are often not sufficiently powerful to specify completely the structure of terms that are acceptable to a predicate. Instead of specifying the circumstances under which an acceptable solution can be reasonably computed, they specify the circumstances under which one step of resolution reduction toward that end can be performed. For example, although the literal contained in the query

$\leftarrow$ length(cons(0, L), N)

is well-moded, we have seen that a mode failure occurs after one reduction step.

A more expressive mode system is needed that can characterize as ill-moded a broader class of literals whose reduction leads, immediately or ultimately, to a mode failure. Such a system contributes to efficiency by preventing fruitless resolution steps, and contributes to robustness by enabling the interpreter to be more selective about reduction orders.

Most importantly, as we will see in Chapter 5, it enables predicate abstraction by permitting precise interface descriptions.

Bi-valued modes would be appropriate if terms were always either variables or ground. The problem is that bi-valued modes cannot distinguish among the different classes of nonvariable terms. For example, the *length* predicate used in the example above would be more precisely moded with restrictions along the lines of

$$\text{length} = \textbf{pred } (\text{list}, \text{nat}) \textbf{ moding } (\text{has no list variable}, \text{all}), (\text{all}, \text{ground})$$

Thus, literals like

$$\text{length}(\text{cons}(X, \text{cons}(0, \text{nil})), N)$$

would be considered well-moded while literals like

$$\text{length}(\text{cons}(X, L), M)$$

would not.

Multi-valued modes are designed to express such finer-grained distinctions. In Section 2.3.1 we will introduce a provisional definition of multi-valued modes, and in Section 2.3.2 we will extend the definition.

2.3.1 Multi-valued mode systems

In multi-valued mode systems, the sets of terms constituting modes are all of the same sort and are closed under instantiation. We will refer to the modes of a multi-valued system as *multi-valued modes*. Since they are sets of terms, multi-valued modes fit directly into the framework of the Denali type system. The closure property ensures that a well-moded literal will remain well-moded regardless of what substitutions are applied to it. The semantics of Denali depends upon this monotonicity property.

Because of the monotonicity property, the order in which the well-moded literals of a query are evaluated is immaterial. An interpreter can choose an arbitrary well-moded literal for reduction without danger that some other well-moded literal will become ill-moded as a result. This makes it possible to specify the behavior of a predicate by specifying its behavior on individual literals, independent of the context in which the literal appears.

We did not highlight this point earlier because closure under instantiation, and hence the monotonicity property, accrues automatically to the two bi-valued modes. It is the monotonicity property, and not the closure property of which it is a consequence, that is fundamental to the semantics of Denali. In Section 2.3.2 we will show how to relax the closure requirement while retaining monotonicity.

We illustrate below some example multi-valued modes of sort *nat*. They are described here in prose. We will show in Section 2.5 how to define them syntactically.

all	all terms of sort *nat*,
nonVar	all nonvariable terms of sort *nat*, and
gnd	all variable-free terms of sort *nat*.

The modes *all* and *nonVar* are identical to their bi-valued namesakes, while *gnd* is distinct.

The modes defined above can be used to place more appropriate mode restrictions upon the *double* predicate than were possible in the previous section:

$$\mathrm{double} = \mathbf{pred}\ (\mathrm{nat}, \mathrm{nat})\ \mathbf{moding}\ (\mathrm{gnd}, \mathrm{all}), (\mathrm{all}, \mathrm{gnd})$$

Any well-moded *double* literal can now be evaluated completely with no possibility of mode failure. This is because some literals that were well-moded under the bi-valued system, such as $double(s(N), M)$, are now ill-moded. Recall that previously the evaluation of this literal led almost immediately to a mode failure.

As a second example, some example modes for *lists* are

all	all terms of sort *list*,
enum	all terms of sort *list* that contain no *list* variable, and
gnd	all ground terms of sort *list*.

These modes can be used to place the mode restrictions, expressed above in prose, upon *length*:

$$\mathrm{length} = \mathbf{pred}\ (\mathrm{list}, \mathrm{nat})\ \mathbf{moding}\ (\mathrm{enum}, \mathrm{any}), (\mathrm{any}, \mathrm{gnd}).$$

2.3.2 Moded bases

Consider the set *pure* of all *nat* terms that are either variables or ground terms, thus excluding terms such as $s(X)$. This set is not a mode under our current definition because it is not closed under instantiation. For example, applying the substitution $\langle X/s(Y)\rangle$ to the term X yields the term $s(Y)$, which is not in *pure*.

There are circumstances under which it is desirable to treat such a set as a mode. This requires altering our definition of what it means for a set to be a mode. The problem is to relax the requirement that modes be closed under instantiation without sacrificing the monotonicity property of well-moded literals.

The key observation is that a mode need be closed only under the application of the substitutions that can be generated by a program. We can exploit this observation by introducing a distinction between *moded* and *unmoded* terms. The moded terms, collectively called the *moded base* of a program, are all of the terms that belong to some mode of the program. A *moded substitution* is a substitution that maps all variables to moded terms.

We need no longer require that each mode be closed under instantiation. Instead, we require two less restrictive conditions. First, each mode must be closed under the application of moded substitutions. Second, every pair of moded terms must have a most general unifier that is a moded substitution.

These two conditions are sufficient to guarantee the monotonicity property of well-moded literals so long as we can guarantee that no unmoded terms are ever unified in the course of interpreting a Denali program. The easiest way to do this would be to forbid the appearance of unmoded terms in programs. In Chapter 3 we will devise a less restrictive means of making this guarantee.

Defining a set of modes is now a two-step process. The first step is to establish the moded base of terms. By convention, we will do this by defining, for each sort, a distinguished mode *any*. The union taken over each such mode *any* is the moded base. All other modes are then defined as subsets of the moded base.

For example, consider the terms of sort *nat*. If we let the moded base be the set of all ground terms and variables, then the following two sets are modes:

any	the set of all variables and ground terms of sort *nat*, and
gnd	the set of all ground terms of sort *nat*.

Notice that the mode *any* is the set *pure* defined above.

2.4 Summary of mode definitions

We summarize below the mode-related definitions that were developed incrementally in this chapter. We will draw upon these definitions throughout the balance of the dissertation. In Chapter 5 we will generalize the definitions of both moded bases and modes to account for equality. None of the other definitions will require further revision.

All of the definitions that follow are made with respect to some fixed moded base of terms. A *moded base* is any set of terms that contains all of the variable terms and is closed under moded instantiation and unification.

- (instantiation) $moded(\sigma) \wedge moded(t) \Rightarrow moded(\sigma t)$.
- (unification) $moded(r) \wedge moded(t) \wedge unifiable(r, t) \Rightarrow \exists\, \sigma$ s.t.
  σ is a most general unifier of r and t, and
  $moded(\sigma)$.

A term is *moded* if it belongs to the moded base; otherwise, it is *unmoded*. A substitution is moded if it maps every variable to a moded term; otherwise, it is unmoded.

A *mode* M of sort S is any set of moded terms of sort S that is closed under moded instantiation:

- $moded(\sigma) \wedge t \in M \Rightarrow \sigma t \in M$.

A *mode tuple* of signature $(S_1, \ldots, S_n)$ is an n-tuple $(M_1, \ldots, M_n)$ such that each M_i is a mode of sort S_i.

A literal $P(t_1, \ldots, t_n)$ is *well-moded* with respect to a mode tuple $(M_1, \ldots, M_n)$ if each t_i is an element of M_i. Otherwise the literal is *ill-moded*.

A set of mode tuples $\{\overline{M}_1 \ldots, \overline{M}_m\}$ is paired with each predicate symbol P. The *moding* of P is the set of all term tuples $(t_1, \ldots, t_n)$ such that $P(t_1, \ldots, t_n)$ is well-moded with respect to one of the $\overline{M}_i$.

Modes are related to their moded base as follows. Each mode is a subset of the moded base. Although they do not partition the moded base, the union of all of the modes is the moded base.

2.5 Defining multi-valued modes

In this section we describe a syntactic method for defining moded bases and modes. It is not expressively complete; that is, it cannot be used to define all possible combinations of moded bases and modes. When used within its limitations, however, it is an effective and easily implemented facility.

The technique involves giving, for each sort, a set of mode signatures. These signatures specify the relationship between the modes of the subterms of a term and the mode of the term. Any number of modes may be defined for each sort, but one of them must be *any*. By convention, variables and all other moded terms are always of mode *any*.

This convention is observed for the following reason. The mode signature technique indirectly defines the moded base since it is the union of the mode *any* of each sort. By definition, the moded base must contain all variable terms.

As a first example, consider the three modes for *nats* that were defined informally in Section 2.3.1. If we rename the mode *all* to be *any*, they can be specified as follows.

0: $\rightarrow$ gnd	s: gnd $\rightarrow$ gnd
0: $\rightarrow$ nonVar	s: nonVar $\rightarrow$ nonVar
0: $\rightarrow$ any	s: any $\rightarrow$ any

There is often an inclusion relationship between modes of the same sort. In the example above, for instance, *any* contains *nonVar*, which in turn contains *gnd*. By explicitly noting the inclusion relationships as part of a mode name presentation, shorter and clearer presentations can be constructed. For example, the mode presentation above can be recast as:

any > nonVar > gnd
0: $\rightarrow$ gnd
s: gnd $\rightarrow$ gnd
s: nonVar $\rightarrow$ nonVar
s: any $\rightarrow$ any

Since *0* is always of mode *gnd*, the inclusion relationships let us conclude that it is also always of modes *nonVar* and *any*.

Let P be a set of mode signatures, and let $>$ be an inclusion ordering. If t is a variable, then $M(t)$ if and only if

$M = any$.

If t is the non-variable term $f(t_1, \ldots, t_n)$, then $M(t)$ if and only if

$$\exists\, f{:}N_1, \ldots, N_n \rightarrow N \in P \;\text{ such that }\; \forall i\, N_i(t_i) \;\text{ and }\; M \geq N.$$

The mode *any* must include all others in an inclusion ordering. Under the definition above, this ensures that all moded terms are of mode any.

In the preceding example, the mode *any* contains all terms of sort *nat*. Recall that this was not the case with the alternate modings for *nat* that we constructed in Section 2.3.2. They can be specified by:

any > gnd
0: $\rightarrow$ gnd
s: gnd $\rightarrow$ gnd

Here $s(X)$ is unmoded because no mode signature applies to it.

Our final example is a presentation for the three modes of *list*.

any > enum > gnd
nil: $\rightarrow$ gnd
cons: gnd, gnd $\rightarrow$ gnd
cons: any, enum $\rightarrow$ enum
cons: any, any $\rightarrow$ any

Notice that these mode signatures incorporate the modes for *nats*, and that the mode names are overloaded.

A complete specification of the modes of a program includes a mode presentation for each sort. The set of terms associated by a presentation with a mode name is always a mode, and the union of each of the modes *any* is always a moded base. When we extend the definitions of moded bases and modes in Chapter 5 to account for equality constraints, this will no longer be the case. At that time we will define a syntactic check upon mode presentations that gives a partial guarantee of this property.

Our mode definition technique is not sufficiently powerful to describe arbitrary modes. For example, the mode that contains exactly the prime numbers cannot be defined using mode signatures. While it is not a complete technique, the advantages of the mode signature technique are its simplicity and ease of checking.

2.6 Related work

Bi-valued modes were first introduced by Warren as declarations for a Prolog compiler [Warren 77]. The interfaces of predicates can be annotated with bi-valued modes, permitting the Edinburgh Prolog compiler to optimize the compiled code. As long as they are correct, the annotations have no other effect. In fact, the annotations are ignored when uncompiled clauses are interpreted directly.

Bi-valued moding schemes have appeared in a number of languages designed as successors to Prolog, including Epilog [Porto 82] and Mu-Prolog [Naish 85]. In these languages, the modes are used to control the selection rule as described in this section.

Variants of bi-valued modes have also appeared in logic languages designed for concurrent applications. Modes are used in these languages to control the parallel solution of literals possessing shared variables. In such languages, the literals of a query are evaluated in parallel, with the evaluation of ill-moded literals suspended until they are sufficiently instantiated. This provides a simple form of concurrency control. The mode mechanism of Parlog [Clark 85] is of the bi-valued variety that we have described. In Concurrent Prolog [Shapiro 83], the mode restrictions are attached to the point of call rather than to the point of definition. The same effect is achieved, and some additional flexibility is obtained.

Modes have also been used to plan the evaluation of queries. [Dembinski 85] describes a scheme that exploits bi-valued mode declarations to perform intelligent backtracking in a Prolog-like interpreter. In the database language NAIL! [Ullman 85], bi-valued modes are combined with capture rules to preplan the evaluation of database queries. This approach is based on the assumption that the data values are either variable or ground, which makes its use of bi-valued modes entirely appropriate.

3 Equational unification in Denali

The function symbols used to construct terms in logic programs are sometimes implicitly interrelated so that syntactically distinct terms are semantically identical. Well-constructed logic programs should treat such terms as equal. In Prolog, and in Denali as defined so far, this can be done only by coding equality constraints indirectly into each predicate implementation.

We can illustrate this point with the sort *list*. Since *nil* and *cons* are the only constructors, all syntactically distinct *list* terms are also semantically distinct. Consider what happens if we add a function symbol *append* with the usual interpretation. We can construct syntactically distinct terms, e.g., $cons(0, nil)$ and $append(nil, cons(0, nil))$, that denote semantically identical *lists*.

Any predicate that manipulates *lists* should treat this pair, and others like it, as equals. We could redefine the *length* predicate, for example, to ensure this:

$$
\begin{array}{l}
\mathrm{length} = \mathbf{pred}\ (\mathrm{list}, \mathrm{nat})\ \mathbf{moding}\ (\mathrm{enum}, \mathrm{any}), (\mathrm{any}, \mathrm{gnd}) \\
\quad \mathrm{length}(\mathrm{nil}, 0). \\
\quad \mathrm{length}(\mathrm{cons}(\mathrm{N}, \mathrm{L}), \mathrm{s}(\mathrm{M})) \leftarrow \mathrm{length}(\mathrm{L}, \mathrm{M}). \\
\quad \mathrm{length}(\mathrm{append}(\mathrm{L}_1, \mathrm{L}_2), \mathrm{N}) \leftarrow \mathrm{length}(\mathrm{L}_1, \mathrm{N}_1), \mathrm{length}(\mathrm{L}_2, \mathrm{N}_2), \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathrm{plus}(\mathrm{N}_1, \mathrm{N}_2, \mathrm{N}).
\end{array}
$$

Under this definition, the twin queries

$\leftarrow$ length(cons(0, nil), N),
$\leftarrow$ length(append(nil, cons(0, nil)), N),

can both be reduced, albeit by different clauses, to produce the result $\langle N/s(0)\rangle$. The lengths of any two equal terms will be found to be the same.

The success of this *ad hoc* technique requires that the programmer consistently embed the appropriate notion of equality in all predicate definitions. A number of logic languages, some of which we will examine at the end of this chapter, make term equality explicit by providing a mechanism for directly defining equality constraints. There are several advantages to this approach. The imposition and maintenance of equality constraints is centralized. Predicate definitions are simpler when they are permitted to exploit equality instead of being required to define it. Equality constraints, such as commutativity, that could not be embedded into a convergent predicate definition can be handled.

In this chapter we show how *equational unification* [Plotkin 72] can be used to systematically impose equality constraints in Denali. We begin in Section 3.1 by giving an overview of equational unification and its role of Denali. This overview is informal; we reserve until Chapters 5 and 6 a more rigorous development. Besides illustrating the utility of equational unification, we point out the difficulties that it poses.

These difficulties pose obstacles to incorporating equational unification into a practical logic programming language. In Section 3.2 we show how the Denali mode system can be used to reduce these obstacles by simplifying the task of implementing equational unification procedures. We will exploit these simplifying properties in Chapter 4 when we describe how equational unification procedures can be written in Denali.

In Section 3.3 we modify the semantics of Denali to take advantage of moded equational unification. This change entails more than replacing classical unification with equational unification. It involves basing a semantics upon moded equational resolution, which we introduce.

We conclude the chapter in Section 3.4, where we discuss the ways in which existing logic languages have incorporated equality constraints.

3.1 Equational unification

In Section 3.1.1 we introduce equational unification. In Section 3.1.2 we outline the problems that we must solve before equational unification can be incorporated into Denali.

3.1.1 Definition

The three equations below express the notion of equality that we have been associating implicitly with the *list* constructors.

$$\mathrm{append}(\mathrm{nil}, \mathrm{L}) = \mathrm{L}$$
$$\mathrm{append}(\mathrm{L}, \mathrm{nil}) = \mathrm{L}$$
$$\mathrm{append}(\mathrm{cons}(\mathrm{N}, \mathrm{L}_1), \mathrm{L}_2) = \mathrm{cons}(\mathrm{N}, \mathrm{append}(\mathrm{L}_1, \mathrm{L}_2))$$

Two terms r and t are considered equal with respect to a set of equations E, written $r =_E t$, if r can be transformed into t (or t into r) using a series of rewriting steps. Rewriting a term r using an equation $e_1 = e_2$ involves finding a substitution σ such that $\sigma(e_1)$ is identical to a subterm of r, and then replacing that subterm with $\sigma(e_2)$. If $r =_E t$ we say that the two terms are E-equal.

The two terms $cons(0, nil)$ and $append(nil, cons(0, nil))$ can be proven E-equal by using a single rewriting step. The right-hand side of the equation

$$\mathrm{append}(\mathrm{nil}, \mathrm{L}) = \mathrm{L},$$

under the substitution $\langle L/cons(0, nil)\rangle$, is equal to $cons(0, nil)$, which is the first of the terms. Replacing this term with the instantiated left-hand side of the equation obtains $append(nil, cons(0, nil))$, which is the other term.

Equational unification is defined with respect to E-equality. A substitution σ is called an *E-unifier* of two terms r and t if and only if $\sigma r =_E \sigma t$. For example, an E-unifier of the terms $cons(0, L)$ and $append(L, cons(0, nil))$ is $\langle L/nil\rangle$. E-unification reduces to classical unification if the set E of equations is empty.

A query can be evaluated with respect to an equational theory E and a definite clause program by replacing classical unification with E-unification in the interpreter. The resulting proof procedure is called *equational resolution* [Plotkin 72]. Although we will see shortly that this replacement is not straightforward, we will assume that it is for the remainder of this section. We illustrate, using two examples, the power inherent in making the replacement.

In the first example, assume that we wish to adhere to the definition of *list* equality that we have established. If unification is done with respect to the *list* equations given above, we can revert to our original definition of the *length* predicate.

$$\begin{array}{l} \mathrm{length} = \mathbf{pred}\ (\mathrm{list}, \mathrm{nat})\ \mathbf{moding}\ (\mathrm{enum}, \mathrm{any}), (\mathrm{any}, \mathrm{gnd}) \\ \quad \mathrm{length}(\mathrm{nil}, 0). \\ \quad \mathrm{length}(\mathrm{cons}(\mathrm{N}, \mathrm{L}), \mathrm{s}(\mathrm{M})) \leftarrow \mathrm{length}(\mathrm{L}, \mathrm{M}). \end{array}$$

Consider, now, the solution of the query

$$\leftarrow \mathrm{length}(\mathrm{append}(\mathrm{cons}(0, \mathrm{nil}), \mathrm{cons}(0, \mathrm{nil})), \mathrm{S}).$$

Using the substitution $\langle N/0,\ L/cons(0, nil),\ S/s(M)\rangle$, the query literal can be E-unified with the head of the second clause to obtain the reduced query

$$\leftarrow \mathrm{length}(\mathrm{cons}(0, \mathrm{nil}), \mathrm{M}).$$

This can be solved in turn, and eventually the solution substitution $\langle S/s(s(0))\rangle$ is reported. Even though the query contains an *append* function symbol and the definition of *length* does not, the query can be solved under equational resolution.

In the second example, suppose that we augment the signature for natural numbers to incorporate a constructor +:

0: $\rightarrow$ nat	any > gnd
s: nat $\rightarrow$ nat	0: $\rightarrow$ gnd
+: nat, nat $\rightarrow$ nat	s: gnd $\rightarrow$ gnd
	s: any $\rightarrow$ any
	+: gnd, gnd $\rightarrow$ gnd
	+: any, any $\rightarrow$ any

Further assume that we perform unification with respect to the equations:

$$\begin{array}{l} 0+\mathrm{X} = \mathrm{X} \\ \mathrm{X}+\mathrm{Y} = \mathrm{Y}+\mathrm{X} \\ \mathrm{s}(\mathrm{X})+\mathrm{Y} = \mathrm{s}(\mathrm{X}+\mathrm{Y}). \end{array}$$

The *double* predicate, which previously required a recursive definition, can now be implemented as:

$$\begin{array}{l} \mathrm{double} = \mathbf{pred}\ (\mathrm{nat}, \mathrm{nat})\ \mathbf{moding}\ (\mathrm{gnd}, \mathrm{any}), (\mathrm{any}, \mathrm{gnd}) \\ \quad \mathrm{double}(\mathrm{X}, \mathrm{X}+\mathrm{X}). \end{array}$$

This implementation is more succinct than was possible before. Its interpretation can be illustrated with the query

$\leftarrow \mathrm{double}(\mathrm{N}, \mathrm{s}(\mathrm{s}(0)))$.

Using the E-unifier $\langle X/s(\mathit{0}),\ N/s(\mathit{0})\rangle$ the query can be reduced in one step.

3.1.2 Problems

We have been treating equational unification as a panacea for introducing equality constraints into Denali. This is misleading, however, since incorporating it into the language is more difficult than is at first apparent. Implementations of logic languages are based upon two properties of classical unification. First, classical unification yields at most a single most general unifier for any two pairs of terms. Second, efficient algorithms exist for finding this unifier. These properties are not, in general, shared by equational unification.

In the classical case, if two terms are unifiable, they possess exactly one most general unifier, unique up to variable renaming. Although the two terms may possess other unifiers, each of these is an instance of the most general one. The completeness of resolution depends upon most general unifiers being used at each step. If a non-general unifier is used during a resolution derivation, the solution derived may not be complete.

To illustrate this last point, we will show what happens when a non-general unifier is used. Consider the solution of the query

$\leftarrow \mathrm{length}(\mathrm{L}_1, \mathrm{s}(0))$.

The most general unifier of this literal and the head of the clause

$\mathrm{length}(\mathrm{cons}(\mathrm{N}, \mathrm{L}), \mathrm{s}(\mathrm{M})) \leftarrow \mathrm{length}(\mathrm{L}, \mathrm{M})$

is $\langle L_1/\mathit{cons}(N, L),\ M/0\rangle$. This leads to the solution $\langle L_1/\mathit{cons}(N, \mathit{nil})\rangle$. If, however, the less general unifier $\langle L_1/\mathit{cons}(\mathit{0}, L),\ M/\mathit{0}\rangle$ is used, the solution that is ultimately obtained is $\langle L_1/\mathit{cons}(\mathit{0}, \mathit{nil})\rangle$. This is a proper instance of the original solution, and as a result is incomplete.

Depending upon the set of equations, two E-unifiable terms may not possess a single most general E-unifier. For example, consider unifying the two terms $\mathit{append}(L_1, L_2)$ and $\mathit{cons}(\mathit{0}, \mathit{nil})$ in the *list* theory given earlier. These terms have the independent unifiers $\langle L_1/\mathit{nil},\ L_2/\mathit{cons}(\mathit{0}, \mathit{nil})\rangle$ and $\langle L_1/\mathit{cons}(\mathit{0}, \mathit{nil}),\ L_2/\mathit{nil}\rangle$.

Since we cannot depend upon an E-unification algorithm finding a single most general E-unifier, we must instead require that it find a complete set of E-unifiers. A complete set Σ of E-unifiers for two terms s and t has the property that any other E-unifier of s and t is an instance of some member of Σ. For example, the two unifiers given above form a complete set.

Equational resolution must account for the possibility of multiple unifiers. Each unifier of a clause and a literal must be used, in turn, to reduce the literal. This is a new source of backtracking and can lead to multiple solutions, much as considering multiple clauses can lead to multiple solutions. The key result in equational resolution is due to Plotkin [Plotkin 72], who showed that E-resolution is complete so long as a complete E-unification procedure is available.

The problem of obtaining complete E-unification procedures is not always easily solved. Although efficient algorithms exist for unifying terms in the classical case, the existence and performance of equational algorithms depends upon the theory in question. This is the key problem in implementing languages that incorporate equational unification.

3.2 Simplifying unification

If we are to incorporate equational unification into Denali, we must provide a way of coping with the complexities outlined above. Although the discovery of more powerful equational unification procedures would be helpful, we are not basing Denali upon any such development. Instead, we have designed Denali to support a methodology that relies upon the programmer's exploiting three separate approaches to the problem of containing the complexity of equational unification.

First, the programmer can exploit the sort hierarchy to stratify an otherwise monolithic equational theory so that the implementation of unification can be decomposed. The programmer is responsible for defining, for each sort, a unification procedure over the terms of that sort. The language implementation is responsible, in turn, for combining these separate procedures into an overall unification procedure. We will elaborate this strategy in Section 3.2.1.

Second, the programmer, by keeping in mind the limitations of unification in Denali, can strive to design realizable abstractions. It is possible to lessen the difficulty of unifying the terms of a given sort by converting some of the constructors of that sort into predicates. This approach is the topic of Section 3.2.2.

Third, the programmer can impose mode restrictions to constrain the domain of the unification procedure of each sort, consequently simplifying its implementation. In imposing the mode restrictions, the programmer must make an engineering tradeoff between the expressive power of the unification procedure and the difficulty of its implementation. We will describe this approach, which we call *moded equational unification*, in Section 3.2.3.

Although we discuss approaches to simplifying the demands upon implementations of unification, we do not describe how implementations are actually written. We will consider the implementation problem in detail in Chapter 4.

3.2.1 Sort stratification

The operational semantics of Denali under equational unification, which we will discuss in Section 3.3, requires that the programmer provide a unification procedure for each sort. The implementation of a unification procedure must respect an underlying equational theory. Thus, the *nat* unification procedure, given any two terms of sort *nat*, should enumerate a complete set of unifiers for the terms. For example, one complete set of unifiers of the two terms $X+Y$ and $s(0)$ is the sequence of two substitutions, $\langle X/0,\, Y/s(0)\rangle$ and $\langle X/s(0),\, Y/0\rangle$.

Because of sort restrictions, not all of the complexity of the equational theory that underlies a program need be built into any particular unification procedure. Only that part of the theory that applies to terms of the sort in question need be considered. For example, suppose that the theory underlying a program contains the union of the *nat* and *list* theories. The *list* constraints need not be taken into account when unifying *nats*, since no *nat* can contain a *list*. The converse is not true, however, since *lists* can contain *nats* as elements.

3.2.2 Simplifying abstractions

The programmer must be aware of the limitations of Denali, and avoid designing programs for which unification procedures cannot be constructed. The most important design decisions in this regard involve deciding what function symbols can be used to construct terms for a given sort. Adding function symbols, while increasing expressive power by making a greater variety of terms available, generally complicates the unification problem since the relationships between symbols can become more complicated.

An alternative to providing an n-ary function as a constructor is to provide it as an $n{+}1$-ary predicate instead. Providing it as a predicate lessens the expressive power of the terms of its sort, since a lesser variety of terms can be constructed. But it simplifies the equational theory since that function symbol's interaction with the others is eliminated.

The effects of this tradeoff can be seen with the *list* sort that we have used repeatedly as an example. When a binary *append* function is incorporated along with the *nil* and *cons* functions, a non-empty equational theory is required to express their relationships. If *append* is supplied as a ternary predicate, classical unification suffices for the remaining *list* terms.

3.2.3 Moded unification

The difficulty of obtaining a complete unification procedure lies not with the equational theory itself but in the set of terms with which the procedure must deal. The demands upon an equational unification procedure can be relaxed if the domain over which it must operate is restricted. Regardless of the theory, for example, it is always trivial to unify pairs of variables. In general, an implementation can be made more tractable by eliminating the instances in which its worst-case performance occurs.

Given an appropriate change to the semantics of Denali, to be explained in Section 3.3, modes can be used to place restrictions upon the interfaces of unification procedures. The resulting problem, moded equational unification, is often easier to solve than the corresponding unmoded equational unification problem.

The weakest possible restriction is to require that both arguments to a unification procedure be of mode *any*. We require that all unification procedures be at least this restrictive. This allows us to permit the appearance of unmoded terms in Denali programs while ruling out the possibility that pairs of unmoded terms will ever be unified.

The requirements of the unification procedure drive the design of the moded base. To illustrate this, consider the equational theory for *nat*. The smallest complete set of unifiers of the pair of terms $X+Y$ and $Z+W$ is infinite. If no moded *nat* term ever contains more than one variable, however, then every unifiable pair of moded *nats* will possess a singleton complete set of unifiers.

A programmer, having made this observation, might choose to define the modes for *nat* so that terms containing more than one variable are unmoded. This can be done by giving the following mode signature:

any $>$ gnd
0: $\rightarrow$ gnd
s: gnd $\rightarrow$ gnd
s: any $\rightarrow$ any
+: $\mathrm{gnd}, \mathrm{gnd} \rightarrow \mathrm{gnd}$
+: $\mathrm{gnd}, \mathrm{any} \rightarrow \mathrm{any}$
+: $\mathrm{any}, \mathrm{gnd} \rightarrow \mathrm{any}$

The key to this is the omission of the stipulation that

+: $\mathrm{any}, \mathrm{any} \rightarrow \mathrm{any}$.

Since terms containing more than one variable are unmoded, they never need to be considered by the unification procedure.

It is sometimes necessary to impose stricter mode restrictions upon unification. Consider, for example, the problem of unifying the two *list* terms $append(L_1, L_2)$ and $append(L_3, L_4)$. The smallest complete set of unifiers is infinite. On the other hand, unifying a pair of *list* terms such as $append(L_1, L_2)$ and $cons(X, cons(Y, cons(Z, nil)))$ is simpler, yielding only four unifiers. By imposing a mode restriction upon *list* unification that requires that one of the two terms be of mode *any* and the other of mode *enum*, better performance can be guaranteed since the problem of unifying pairs of unenumerated *lists* need not be considered by the implementation.

We will adopt the following general scheme for expressing the mode restriction that is to be placed upon the unification procedure for a sort S. A single mode M of sort

S encodes the restriction for the procedure. One argument to the unification procedure must be of mode M, while the other must be of mode *any*. In other words, if the mode restriction M is *any*, then the unification moding is (any, any); otherwise, the moding is $(M, any), (any, M)$. The moding for *nat* unification can be encoded as *any*, while the moding for *list* unification can be encoded as *enum*.

3.3 Moded equational resolution

We can now describe how moded equational unification is incorporated into the resolution procedure that underlies the semantics of Denali. We call the resulting procedure moded equational resolution. We will assume that a moded equational unification procedure is provided for each sort. In Chapter 4 we will see how these procedures can be implemented in Denali.

The sort-specific unification procedures can yield multiple substitutions. We deal with this problem by modeling each unification procedure as a Denali predicate. We can then treat the problem of unifying pairs of literals as one of solving a query derived from the literal. We explain this approach in Section 3.3.1.

Adopting this approach permits us to exploit the mode restrictions upon unification to control the order in which individual terms are unified, exactly as with ordinary Denali queries. Some problems stemming from this will lead us to intermix the currently distinct resolution steps of first unifying a literal with the head of a clause and then solving the clause's body. We discuss this revised approach in Section 3.3.2.

3.3.1 Multiple unifiers

The sort-specific unification procedures can be regarded as Denali predicates. A unification procedure maps pairs of terms to sets of substitutions. Viewed as a predicate, it maps pairs of terms to sequences of substitutions. We can exploit this similarity as we develop moded equational resolution.

We must create headers for the sort-specific unification predicates. For example, the headers for *nat* and *list* unification are, respectively,

$$\text{natUnify} = \textbf{pred}\ (\text{nat}, \text{nat})\ \textbf{moding}\ (\text{any}, \text{any}),$$
$$\text{listUnify} = \textbf{pred}\ (\text{list}, \text{list})\ \textbf{moding}\ (\text{any}, \text{enum}), (\text{enum}, \text{any}).$$

By convention, we will form the name of the predicate from the name of the sort over which it operates. This also fixes the sort signature of the predicate. The mode tuples are derived from the mode restriction upon unification as we described earlier. Thus, *natUnify* unifies pairs of moded *nats*, and *listUnify* unifies pairs of moded *lists*, at least one of which must be of mode *enum*.

Modeling the unification procedures as predicates in this manner affords us a convenient way of illustrating equational resolution. We can model the problem of unifying a pair of literals, e.g. $P(nil, N)$ and $P(L, 0)$, as the problem of solving the query,

$$\leftarrow \text{natUnify}(\text{nil}, \text{L}), \text{natUnify}(\text{N}, 0).$$

For example, consider the problem of solving the query

$$\leftarrow \text{less}(\text{s}(0)+\text{s}(0), \text{s}(0)).$$

with respect to the definition

$$\begin{array}{l} \text{less} = \textbf{pred}\ (\text{nat}, \text{nat})\ \textbf{moding}\ (\text{gnd}, \text{gnd}) \\ \quad \text{less}(\text{s}(\text{X}), 0). \\ \quad \text{less}(\text{s}(\text{X}), \text{s}(\text{Y})) \leftarrow \text{less}(\text{X}, \text{Y}). \end{array}$$

Reducing the query with respect to the second clause of the implementation would normally involve, as a first step, unifying two literals. We can model this problem as one of solving the query

$$\leftarrow \text{natUnify}(\text{s}(\text{X}), \text{s}(0)+\text{s}(0)), \text{natUnify}(\text{s}(\text{Y}), \text{s}(0)).$$

The solution obtained in this step, $\langle X/s(0),\ Y/0\rangle$, can then be used to instantiate the remainder of the query as usual.

There are two advantages to treating the unification step in this way. First, it provides a natural way of dealing with multiple unifiers. We are already accustomed to solving a query for more than one substitution.

Second, this way of doing unification gives us a way to enforce and exploit the mode restrictions upon unification. For example, consider solving the query

$$\leftarrow \text{double}(\text{s}(0), \text{N}).$$

Recall that our definition of *double* is

double = **pred** (nat, nat) **moding** (gnd, any), (any, gnd)
double(X, X+X).

This involves solving the unification query

$\leftarrow$ natUnify(s(0), X), natUnify(N, X+X).

The second literal of this query, because it contains the unmoded term $X+X$, is not well-moded. This means that we must solve the other literal first. Fortunately, in this step the unmoded terms becomes sufficiently instantiated and its literal can be subsequently solved.

3.3.2 Mode restriction anomaly

To this point we have maintained two separate phases of the resolution reduction of a query. First a unifier is obtained, and then that unifier is applied to the remainder of the query. This separation is not necessary and can actually be counterproductive. Eliminating the separation leads to a more robust evaluation strategy.

The mode restriction upon a unification predicate leads to an anomaly when it is stricter than *any*. Because variables are always of mode *any*, such a restriction explicitly forbids attempts to unify variables. For example, since the *list* unification restrictions require that one of the terms be of mode *enum*, the attempted unification of two list variables, as in

$\leftarrow$ listUnify(L_1, L_2)

is ill-moded and cannot be solved.

Since two variables can always be directly unified, regardless of the underlying equational theory, imposing mode restrictions upon unification appears to have a serious drawback. We will now evaluate possible solutions to the problem.

Sometimes the apparent need to unify a pair of variables can be avoided because of the ordering effects of modes. One of the variables can become sufficiently instantiated before the unification literal must be evaluated. This can be illustrated with the predicate *prefix*:

$$\begin{array}{l} \mathrm{prefix} = \textbf{pred}\ (\mathrm{list}, \mathrm{list})\ \textbf{moding}\ (\mathrm{enum}, \mathrm{any}) \\ \quad \mathrm{prefix}(\mathrm{append}(\mathrm{L}_1, \mathrm{L}_2), \mathrm{L}_1) \end{array}$$

This predicate relates the first argument to its prefixes. It can be invoked, for example, with the query query

$$\leftarrow \mathrm{prefix}(\mathrm{cons}(0, \mathrm{nil}), \mathrm{L}).$$

The first unification step in reducing this query is the query

$$\leftarrow \mathrm{listUnify}(\mathrm{cons}(0, \mathrm{nil}), \mathrm{append}(\mathrm{L}_1, \mathrm{L}_2)), \mathrm{listUnify}(\mathrm{L}_1, \mathrm{L}).$$

Even though the second literal is ill-moded, the first one is not and can be solved. It has multiple solutions, and in one case leads to

$$\leftarrow \mathrm{listUnify}(\mathrm{cons}(0, \mathrm{nil}), \mathrm{L}),$$

which is now well-moded and can be solved to yield a solution.

Unfortunately, not all attempts to unify variables can be avoided in this way. To illustrate this, we define another predicate,

$$\begin{array}{l} \mathrm{nPrefix} = \textbf{pred}\ (\mathrm{list}, \mathrm{nat}, \mathrm{list})\ \textbf{moding}\ (\mathrm{enum}, \mathrm{gnd}, \mathrm{any}) \\ \quad \mathrm{nPrefix}(\mathrm{L}_3, \mathrm{N}, \mathrm{L}_4) \leftarrow \mathrm{prefix}(\mathrm{L}_3, \mathrm{L}_4), \mathrm{length}(\mathrm{L}_4, \mathrm{N}). \end{array}$$

This predicate reports only prefixes of the specified length. As before, the first step in solving the query

$$\leftarrow \mathrm{nPrefix}(\mathrm{cons}(0, \mathrm{nil}), 1, \mathrm{L}),$$

involves solving the unification query

$$\leftarrow \mathrm{listUnify}(\mathrm{cons}(0, \mathrm{nil}), \mathrm{L}_3), \mathrm{natUnify}(1, \mathrm{N}), \mathrm{listUnify}(\mathrm{L}, \mathrm{L}_4).$$

Solving the first two well-moded unification literals does not further instantiate the ill-moded literal, and a mode failure results.

One potential solution to the problem is to treat variable unification as a special case, and always permit it regardless of mode restrictions. Although this solution is appealing in its simplicity, it is defective and cannot be adopted.

Recall that we require that well-moded literals remain well-moded under instantiation. If the variable exception were exploited in the solution of a unification query, it might be possible to instantiate one or both of the variables involved so that the exception no longer applied but the unification literal became ill-moded. For example, even if the variable exception were exploited to solve the query above, solving the query instance

$\leftarrow \mathrm{nPrefix}(\mathrm{cons}(0, \mathrm{nil}), 1, \mathrm{cons}(0, \mathrm{L}),$

would now lead to a mode failure. Because we want to ensure that the solvability of queries is invariant under instantiation, we reject the simplistic solution.

Fortunately, we can go far toward solving the problem by eliminating the arbitrary division between the unification and reduction steps that is present in our evaluation strategy. Instead, we permit the intermixing of the solution of unification and clause literals. This provides a greater opportunity for one term of a non-unifiable pair to become further instantiated.

This change permits the solution of the query above. The first step in the solution is to form the joint unification/body query:

$$\begin{aligned}\leftarrow\ & \mathrm{listUnify}(\mathrm{cons}(0, \mathrm{nil}), \mathrm{L}_3), \mathrm{natUnify}(1, \mathrm{N}),\\ & \quad \mathrm{listUnify}(\mathrm{L}, \mathrm{L}_4), \mathrm{prefix}(\mathrm{L}_3, \mathrm{L}_4), \mathrm{length}(\mathrm{L}_4, \mathrm{N}).\end{aligned}$$

The first two unification literals can be solved immediately, leading to

$\leftarrow \mathrm{listUnify}(\mathrm{L}, \mathrm{L}_4), \mathrm{prefix}(\mathrm{cons}(0, \mathrm{nil}), \mathrm{L}_4), \mathrm{length}(\mathrm{L}_4, 1).$

Now, even though the remaining unification literal is ill-moded, the *prefix* and then the *length* literals can be solved, leading to

$\leftarrow \mathrm{listUnify}(\mathrm{L}, \mathrm{cons}(0, \mathrm{nil})).$

The unification literal is now well-moded and can be solved, yielding a solution.

The modification to the evaluation strategy described above does not always succeed in preventing mode failure due to attempts to unify pairs of variables. It often does, however, because the inclusion of a variable in a literal is usually a technique for obtaining output from the evaluation of the literal. The modification strictly enlarges the class of queries that could be solved under the old strategy without changing the evaluation of any previously solvable queries. Finally, the change is not particular to pairs of variables. It helps prevent unification-related mode failures for all terms.

3.4 Related work

Kornfeld [Kornfeld 86] was one of the first to suggest the power of incorporating equality constraints into Prolog. His observations have inspired a number of other efforts in this direction. He incorporates equality by allowing the Prolog unification procedure to invoke

Lisp procedures whenever unification fails. This *ad hoc* approach has no sound formal basis.

Eqlog [Goguen 86] is closely related to our work, as it incorporates equational unification in much the same way as Denali. Its approach to obtaining implementations of equational unification, however, is completely different. Instead of relying upon a set of pragmatic techniques for simplifying the unification problem, Eqlog relies upon the narrowing procedure [Fay 79] to compile sets of equations into unification procedures. Unfortunately, the narrowing procedure is primarily of theoretical interest, as it is generally nonterminating and usually produces inefficient implementations. Furthermore, it is applicable only to sets of equations that can be converted into a canonical set of rewrite rules, and does not exploit modes. Under the narrowing procedure, the programmer has little control over or intuition about the efficiency characteristics of unification.

Tablog [Malachi 86] is a logic language based upon the Manna-Waldinger deductive-tableau proof system [Manna 80]. This proof system explicitly handles equations as well as clauses. The language was designed and implemented as a demonstration that a logic programming language can be based upon something other than the resolution procedure. Its treatment of equality is incomplete, however, as it makes an asymmetrical distinction between *primitive* and *defined* function symbols.

SLOG [Fribourg 84] is a logic language in which only one predicate may be defined by a programmer. This predicate is equality. Queries are solved with a variant of resolution called innermost superposition. The equality predicate is used to reduce not only entire literals but also inner terms.

4 Data abstraction in Denali

In this chapter we describe the Denali data abstraction mechanism, and show how user-defined abstractions can be implemented. As in non-logic programming languages that support user-defined data abstraction, this is done by first fixing a concrete representation for the abstract objects, and then realizing the interface in terms of this representation. This approach differs, however, from other languages that have extended Prolog, none of which distinguish between the means used to represent objects and the means used to denote them as terms.

The problem of implementing data abstractions in Denali is complicated by their intricacy. We begin in Section 4.1 by developing the requirements for these abstractions. In most languages, a data abstraction provides a set of abstract objects and operations. Denali abstractions also provide a denotation scheme, a set of modes, and a unification predicate.

A data abstraction can be implemented in three ways. The first way, which we mention for the sake of completeness, is by providing it as a built-in abstraction. This method is appropriate only for abstractions of broad applicability such as integers and lists. Fixing the set of built-in abstractions is a crucial part of a full language design. However, as we are concerned only with designing the framework of Denali, we do not address this aspect here.

The remaining two implementation techniques apply to user-defined abstractions, the focus of this chapter. We illustrate in Section 4.2 the technique of *implicit implementation*, in which denotations double as the concrete representations of objects. Abstract modes are defined in terms of denotations. The unification procedure is not explicitly implemented; instead, a set of equations is given that specifies the desired procedure. The language implementation, in turn, synthesizes the appropriate unification procedure.

In Section 4.3 we explain the technique of *explicit implementation*. With this technique, the programmer uses the objects of some other sort to stand as the concrete representation. Exploiting levels of abstraction in this way leads to more flexibility in program design than does implicit implementation. However, the implementation process is more complicated. The denotation scheme, the abstract modes, the unification procedure, and any exported predicates all must be explicitly implemented in terms of the concrete representation.

In Section 4.4 we discuss the differences between implicit and explicit implementation, and explain the circumstances under which each approach is applicable and appropriate.

We conclude our development of Denali in Section 4.5 by summarizing the role of abstraction in reasoning about Denali programs.

4.1 Requirements

In this section we develop requirements for data abstraction in Denali. Since a data abstraction encapsulates all of the information needed to characterize the objects of some sort, it must include the three forms of external information we assumed present when describing predicate abstraction in Chapter 2. This includes function signatures, mode definitions, and implementations of unification. A data abstraction also provides a set of predicates.

We begin in Section 4.1.1 by examining the role of data abstraction in conventional languages, and continue in Section 4.1.2 by considering the role of terms as abstract objects in Prolog. In Section 4.1.3 we synthesize our observations into a set of requirements for the Denali data abstraction mechanism.

4.1.1 Data abstraction in conventional languages

Data abstractions appear in one form or another in almost all programming languages. Built-in implementations of data abstractions that appear almost universally include scalar types such as integers and data structures such as arrays. Many modern languages also provide facilities that support the construction of user-defined data abstractions.

The role of a data abstraction is to provide a set of objects together with a set of operations to create, modify, and observe them. For example, languages that support integers generally provide constants, implementations of the standard arithmetic functions, and comparison relations. In some cases, an operation is logically related to a particular abstraction even though it is not presented as such. For example, array and record update operations are often described as special cases of assignment.

A trend in modern languages has been to support the construction of user-defined data abstractions that present the same kind of interfaces as their built-in counterparts. The programmer can design extensions to the set of data types provided by a language, tailoring the language's expressive power to the requirements of the application.

Almost inseparable from the concept of data abstraction is the idea of encapsulation. Ideally, the only way to manipulate the objects of an abstraction should be through the means provided by the abstraction's interface. If this principle is observed when implementing a data abstraction, several benefits accrue. The meaning of the abstraction can be specified by describing the behavior of the components of its interface. Because of this, two implementations can be considered equivalent if they present behaviorally identical interfaces. Equivalent implementations can be freely interchanged without fear of altering the behavior of clients of the abstraction. When the principle of encapsulation is observed, the interface need not be bound to any particular means of representing abstract objects.

This last property is intrinsic to implementing built-in types, and is one of the contributions of early programming languages. Thus, the implementation of integers in almost all programming languages is hidden from the programmer. The principle of encapsulation has been extended to user-defined abstractions. For example, two properly constructed implementations of a set abstraction, one based upon arrays and the other upon binary trees, can be transparently interchanged.

4.1.2 Data abstraction in Prolog

The requirements for data abstraction in the context of a logic language are more extensive than for the conventional case considered above. The source of the difference is the central role of unification and the need to denote directly the values of objects within the text of programs. Although Prolog provides no facilities for constructing user-defined data abstractions, it implicitly supplies a single built-in data type. We analyze below the role of this type in Prolog, in the process developing further insight into the requirements for Denali's data abstraction mechanism.

The only built-in type in pure Prolog is the uninterpreted term. Terms serve the same role in Prolog as objects do in imperative languages. They are similar, in some respects, to records. The most significant difference is that terms can contain, or can even be, variables. Anything from an individual variable to a ground term is an acceptable object in Prolog. This property is one source of Prolog's expressive power.

A data abstraction should provide all of the operations needed to manipulate its abstract objects. In pure Prolog, the only means of manipulating terms is by unifying and instantiating them. Because these two operations are central to the logic programming paradigm, and because there is only one built-in data type, it is convenient to regard them as built-in components of the Prolog interpreter. They could, however, be regarded as abstract operations supplied by the single built-in type. The importance of this distinction will become clear in Denali, in which more than one data abstraction can appear.

We have not been careful thus far to distinguish the terms provided by Prolog from the abstract objects they denote. It is important, however, to draw a distinction between an abstract object, its denotation, and its internal representation. It is difficult to motivate the difference in pure Prolog since uninterpreted terms can conveniently serve as both denotation and representation. Instead, we examine standard Prolog [Clocksin 81] to illustrate our point.

A specialized infix list notation is used extensively in Prolog programs. In this notation, a single list object can have two or more distinct denotations. In an implementation of Prolog, each list has a unique internal representation that is distinct from any denotation. List denotations must be translated into this representation before they can be manipulated with unification and instantiation.

The example above illustrates that a denotation is a syntactic artifact that need have no direct relationship to the representation of an object. This is especially true since denotations and representations are generally chosen to suit different criteria. The means used to denote the objects provided by a data abstraction should be part of the syntactic interface of the abstraction, and not a consequence of its implementation. Similar issues, as we have seen, arise in connection with conventional languages.

4.1.3 Denali interfaces

We have discussed the nature of data abstractions in both conventional and logic languages. The underlying idea is that an abstraction must provide a set of abstract objects in a way that separates the means of manipulating the objects from the means of implementing them. The mechanisms provided by an abstraction for manipulating its objects depends in part upon the nature of the language and in part upon the nature of the abstraction.

The interface of an abstraction must provide the syntactic information that clients need to use the abstract objects. We now describe the style of interface presented by data abstractions in Denali. Although we will eventually need to explain how implementations associate meaning with interfaces, we will concentrate for now upon their form.

The list below summarizes the five ways in which Denali objects can be derived and observed. It provides the starting point for the design of data abstraction in Denali:

nat = **cluster**
  denoted by
    0: $\to$ nat
    s: nat $\to$ nat
    +: nat, nat $\to$ nat
  modes any > gnd
  unify mode any
  add = **pred** (nat, nat, nat) **moding** (gnd, any, any),
            (any, gnd, any), (any, any, gnd)
  less = **pred** (nat, nat) **moding** (gnd, gnd)
  end

Figure 4.1: *Nat* interface

- Objects can be denoted directly by terms.
- Modes can be used to classify objects according to their degree of instantiation.
- Pairs of objects can be unified to obtain substitutions.
- Literals containing objects as arguments can be resolved to obtain substitutions.
- Substitutions can be applied to objects to derive new objects.

Data abstractions in Denali are implemented by *clusters*, and are defined on a sort-by-sort basis. Each cluster implements the objects of a particular sort, presenting in its interface the syntactic information needed to effect the manipulations listed above. The interface of a *nat* abstraction appears in Figure 4.1. We will refer to it below as we describe the components of cluster interfaces.

The header of the interface names the sort, in this case *nat*, to be defined by the cluster. We will refer to this sort generically as the *sort of interest.*

The first portion of the interface gives the sorted grammar that defines how objects of the sort of interest can be denoted. The range of every function symbol defined in this grammar must be the sort of interest. Independently of the grammar, variables are always valid denotations.

The expressive power of a data abstraction is related to the number of its objects that can be denoted. Not every abstract object need be denotable; in fact, no method of denotation need be given at all. In such a case, only variable objects can be written directly. This does not imply, of course, that non-variable abstract objects cannot exist.

It means instead that they must be constructed indirectly by solving literals. This is not an unusual condition, since objects can be obtained only indirectly in languages that do not support denotations.

The next portion of the interface identifies the names of the modes provided by the abstraction and indicates their inclusion relationship. Mode *any* is by convention the most general mode. It must always appear in the interface, and it must include all other modes. If it does not appear, we will assume it implicitly present. Modes are used, as in Chapters 2 and 3, to classify classify abstract objects. They can appear in the headers of predicates that manipulate arguments of the sort of interest.

A syntactic interface does not implement the mode names. However, the implementation is constrained by the interface to respect the indicated inclusion relationships. Although they are formally unrelated to the denotation scheme, modes can be described with respect to the denotation scheme, as in Chapter 2, whenever every abstract object is denotable.

The third component of the interface is the mode restriction for the unification procedure of the sort of interest. The procedure itself is, of course, provided by the implementation.

The remainder of the interface gives the headers of any predicates provided by the abstraction. Just as in other languages, it is often advantageous to define a predicate inside of a cluster so that it can take advantage of the internal representation of abstract objects. The set of predicates provided by the *nat* cluster, while by no means complete, is intended to be representative.

Neither interfaces nor implementations deal explicitly with the means of applying substitutions to abstract objects. This is because Denali treats the application of substitutions as a generic operation, analogous in some ways to assignment in imperative languages. Applying a substitution to an object always involves replacing the variables within the object with their images under the substitution. The generic treatment of substitutions is only possible if certain relationships hold between abstract objects, their denotations, and the means used to represent them. The most important relationship is that abstract variables must be represented by concrete variables.

bag = **cluster**
  denoted by
    nil: $\rightarrow$ bag
    cons: nat, bag $\rightarrow$ bag
  modes any > enum > gnd
  unify mode enum
  reduce = **pred** (bag, bag) **moding** (gnd, any)
  length = **pred** (bag, nat) **moding** (enum, any)
  end

Figure 4.2: *Bag* interface

4.2 Implicit implementation

An implicitly implemented cluster is, for the most part, a structured collection of the external pieces of information upon which we based the definition of predicates in Chapter 2. We will illustrate the technique of implicit implementation by showing how it can be used to implement the *bag* interface that appears in Figure 4.2.

We first give an informal specification of the *bag* abstraction. At the syntactic level, the abstraction provides exactly the same set of objects as does the familiar *list* abstraction. Semantically, however, *bags* that contain the same elements in the same numbers are treated as identical, regardless of the order in which they were originally formed. The unification procedure, of course, must respect this equality restriction. All *bag* objects are moded; further, any object that contains no *bag* variable is of mode *enum*, and any ground *bag* is of mode *gnd*.

An implicit implementation of the *bag* abstraction appears in Figure 4.3. We will draw upon it as an example as we discuss each of the aspects of implicit implementation in detail.

The denotation portion of an implicitly implemented cluster is carried over unadorned from the syntactic interface. Because the denotation scheme is the basis of the implementation, no mechanism is needed to relate it to the other facets of the abstraction. We will see that this is not the case under explicit implementation, in which the programmer must give a way of translating denotations into representations.

The abstract modes named by the interface are defined with a set of mode signatures.

```
bag = cluster
    denoted by
        nil: → bag
        cons: nat, bag → bag
    modes any > enum > gnd
    unify mode enum
    moded by
        nil: → gnd
        cons: gnd, gnd → gnd
        cons: any, enum → enum
        cons: any, any → any
    unified by
        cons(X, cons(Y, B)) = cons(Y, cons(X, B))
    reduce = pred (bag, bag) moding (gnd, any)
        reduce(nil, nil).
        reduce(cons(N, cons(N, B1), B2) ← reduce(cons(N, B1), B2).
    length = pred (bag, nat) moding (enum, any)
        length(nil, 0).
        length(cons(M, B), s(N)) ← length(B, N)
    end
```

Figure 4.3: Implicit implementation of *bag*

Taken together with the inclusion relations carried over from the interface, these equations specify the meanings of the modes as explained in Chapter 2.

The unification procedure is implemented by presenting an equational theory that appropriately constrains the function symbols introduced by the abstraction. In the implementation of *bag*, for example, only the single equation

$$\mathrm{cons}(X, \mathrm{cons}(Y, B)) = \mathrm{cons}(Y, \mathrm{cons}(X, B))$$

needs to be given. As this equation gives no constraints upon the equality of *nats*, it does not present the entire equational theory for *bag* objects. Given an implementation of *nat* unification, however, the Denali implementation is able to use this equation to synthesize a procedure for *bag* unification.

We are not yet prepared to describe in more than cursory fashion the means by which this is done. Our approach is a generalization of the algorithms developed by Yelick [Yelick 85] and Tiden [Tidèn 86] to combine unification algorithms. We will explain and justify this approach in Chapter 6. It is important to point out, however, that there are two important restrictions upon the synthesis approach.

First, the equational presentation given in an implicit implementation must describe a theory for which a unification procedure is known to the language implementation. The equation given in the *bag* cluster presents the theory of *left-commutativity*, for which a unification algorithm exists [Jeanrond 80]. In a real language, the name of the theory would probably appear instead of its specifying equations.

Second, the equational presentation must contain only function symbols whose range is the sort of interest. Such a presentation is said to be *homogeneous* with respect to the sort of interest. In the *bag* example, for instance, only the function symbol *cons* appears, so the presentation is homogeneous with respect to *bag*.

Because the denotation is used as the representation, it makes no difference whether or not a predicate is implemented inside or outside of a implicitly implemented cluster. Because their interfaces appear in the cluster interface, we have implemented the *bag* predicates *reduce* and *length* inside of the *bag* cluster. We will see in the next section that the location of a predicate's definition is material in the case of explicit implementations.

4.3 Explicit implementation

Each implicitly implemented cluster must be implemented independently of all others. Since the representation is tied to the denotation, there is no way to represent the objects of one abstraction in terms of the objects provided by another. The technique of constructing programs by building layers of abstractions is central to other languages that support user-defined abstract data types.

We now introduce a mechanism that complements implicit implementation by supporting the layering of abstractions. We will illustrate this mechanism by showing how to implement the *set* interface that appears in Figure 4.4. The key development is a technique for translating denotation terms into concrete representations.

Again, we give a brief specification. The *set* abstraction provides all of the *set* objects that can be formed from the functions *empty* and *insert*. Only individual variables, fully enumerated *sets*, and ground *sets* are moded. All other *set* objects are considered

set = **cluster**
    denoted by
        empty: $\rightarrow$ set
        insert: nat, set $\rightarrow$ set
    modes any $>$ enum $>$ gnd
    unify mode any
    size = **pred** (set, nat) **moding** (gnd, any)
    end

Figure 4.4: *Set* interface

unmoded. All *set* objects, including the unmoded ones, can be denoted.

An explicit implementation of the *set* abstraction appears in Figure 4.5. We will draw upon it as we discuss explicit implementation.

As with implicit implementations, the interface is carried over unchanged. The first line that follows the interface information indicates that objects of sort *bag* are used to represent abstract *set* objects. As in other languages that support user-defined data abstraction, fixing the representation is the first step in designing an implementation. The next step is to choose a *representation invariant* and an *abstraction function.* In the present case, the representation invariant specifies which *bag* objects are valid representations, and the abstraction function specifies a correspondence between valid *bag* objects and *set* objects.

Because of the way substitution application is treated, the abstraction function must satisfy the following condition. Abstract variables must always be represented by concrete variables. Hence, this aspect of the representation is established by default. In the *set* example, then, the programmer need only determine how non-variable sets are to be represented.

We have chosen a straightforward representation invariant and abstraction function for the *set* implementation. A *bag* is a valid representation unless it is a nonvariable that contains an embedded *bag* variable. A *set* is represented by any *bag* that contains exactly the elements of the *set*. The presence of duplicates is not forbidden, so more than one *bag* can represent a single *set*.

Having fixed a representation scheme, the remainder of the implementation task is to

```
set = cluster
    denoted by
        empty: → set
        insert: nat, set → set
    modes any > enum > gnd
    unify mode any
    represented by bag
    translated by setTrans
    moded by
        gnd from gnd
        enum from enum
        any from any
    unified by setUnify
    setTrans = pred (set_dnt, bag) moding (any, any)
        setTrans(empty, nil).
        setTrans(insert(N, L), cons(N, L)).
    setUnify = pred (bag, bag) moding (any, any)
        setUnify(nil, nil).
        setUnify(cons(X, B1), cons(X, B2)) ← setUnify(B1, B2).
        setUnify(cons(X, B1), cons(X, B2)) ← setUnify(cons(X, B1), B2).
        setUnify(cons(X, B1), cons(X, B2)) ← setUnify(B1, cons(X, B2)).
    size = pred (set, nat) moding (gnd, any)
        size(B1, N) ← reduce(B1, B2), length(B2, N).
    end
```

Figure 4.5: Explicit implementation of *set*

realize each of the four components of the interface in terms of the representation. We examine these four aspects of the implementation in the following four sections.

4.3.1 Denotations

The relationship between the denotation and representation schemes must be given explicitly by the programmer. This is done by defining, within every explicitly implemented cluster, a translation predicate that relates each denotation to its representation. In the *set* example, this translation predicate is *setTrans*.

At compile time, translation predicates are used to find a representation for every denotation in the program. The representations obtained this way are then used to replace the denotations in the program. After a query is evaluated, the translation process is reversed to obtain the denotation of the solution. To ensure that we translation process is reversible, we require that the translation predicate be defined over all well-sorted pairs of terms.

In the translation process, the denotations must be treated purely syntactically, i.e., as uninterpreted terms. For this reason the translation predicate is not defined over abstract objects but over denotation objects. This fact is apparent in the header of *setTrans*, which is reproduced below.

setTrans = **pred** (set_dnt, bag) **moding** (any, any)
  setTrans(empty, nil).
  setTrans(insert(N, L), cons(N, L)).

A translation predicate is defined over pairs of objects. The first argument must be of the denotation sort corresponding to the abstract sort (in this case *set_dnt*), and the second argument must be of the representation sort (in this case *bag*). The sort name *set_dnt* is a reserved symbol that can appear and is meaningful only within an explicitly implemented *set* cluster. It should be taken to be a sort that provides the same constructors as the *set* denotations, but with altered signatures as suggested below:

empty: $\rightarrow$ set_dnt
insert: nat, bag $\rightarrow$ set_dnt

We obtain the signatures for *set_dnt* by replacing each occurrence of the sort *set* in the domain of a function symbol with the representation sort, and by replacing each occurrence of *set* in the range of a function symbol with *set_dnt*.

The constructors of sort *set_dnt* are designed so that the translation process can be performed beginning with the innermost terms of a denotation and moving out. Consider, for example, the translation of the *set* denotation $insert(1, insert(N, empty))$. The innermost term is *empty*, which is a well-sorted *set_dnt* term. Translating *empty* involves posing and solving the query

$\leftarrow$ setTrans(empty, B),

where B is a fresh variable. Under the definition of *setTrans*, the substitution $\langle B/nil \rangle$ is obtained. We use the value of B to replace *empty* in the original term. The partially translated denotation is now $insert(1, insert(N, nil))$.

The innermost untranslated term, $insert(N, nil)$, is now a well-sorted *set_dnt* term. Consequently, we can translate it by posing and solving

$\leftarrow \mathrm{setTrans}(\mathrm{insert}(\mathrm{N}, \mathrm{nil}), \mathrm{B})$,

obtaining the substitution $\langle B/cons(N, nil)\rangle$.

Continuing in this fashion for one more step, we finally obtain the translated value of the original denotation, which is $cons(1, cons(N, nil))$.

We have only sketched the translation process. As we illustrated above, the task of translating an entire program must be ordered so that subterms are translated before outer terms. Furthermore, the translation predicates must be translated before they can be used. In general, constraints are needed upon the definition of denotations to ensure that a well-defined translation order always exists. We will discuss these constraints in Chapter 7 when we discuss the meaning of Denali programs.

4.3.2 Abstract modes

When explicit representations are used, the modes for the abstract objects must be defined in terms of the modes of the representation objects. This is done by explicitly associating concrete modes with the abstract modes they implement. In general, a single abstract mode M can be implemented by some number of concrete modes $N_1, \ldots, N_n$ by writing

M **from** $\mathrm{N_1} + \cdots + \mathrm{N_n}$.

Thus, a non-variable abstract object is of mode M only when it is represented by a concrete object of some mode N_i. Not every concrete object of mode N_i need represent an abstract object.

Abstract variables are always represented with concrete variables, and they are by default always of mode *any*. It is thus possible to define the abstract *any* in terms of an arbitrary concrete mode without excluding variables. In the *set* implementation, for example, the two abstract modes *enum* and *any* are both implemented in terms of the concrete *bag* modes *enum*. Only *any*, however, contains the abstract variables.

We illustrate the determination of abstract modes by using the *set* example. Bear in mind that we have overloaded the mode names between *sets* and *bags*. The *set* denotation $insert(1, insert(N, empty))$ is represented by the *bag* object $cons(1, cons(N, nil))$. The latter term belongs to the *bag* modes *any* and *enum*. The implementation of the *set* modes is reproduced below.

gnd **from** gnd
enum **from** enum
any **from** enum

The represented *set* term, then, has *set* modes *any* and *enum*. It is also easy to see that the *set* denoted by $insert(1, S)$ is unmoded, since its representation, $cons(1, S)$, is neither of mode *enum* nor of mode *gnd*.

4.3.3 Exported predicates

Predicates defined within an explicitly implemented cluster are written to deal with concrete arguments. Thus, for example, the *size* predicate is written to expect *bag* objects.

In the design of the *set* abstraction, we have have made a programming decision to require that the *set* argument to *size* be of mode *gnd*. This is because the size of a *set* cannot be known with certainty if it contains variables. We could have chosen instead to accept arguments of mode *enum*. Finding the size of a *set* containing variables, however, would require making assumptions for the values of the variables and reporting the size of the resulting *set* for each case. We have thus chosen a more restrictive but more efficient variant.

The size of a *set* is obtained by deleting duplicates from the representation and then taking the size of the resulting *bag*. If we could guarantee that no duplicate elements would ever appear in a representation, we could streamline the implementation and avoid the elimination of duplicates. Representation invariants such as this are often enforced when implementing data abstractions in conventional languages.

The problem with taking this approach in Denali is that objects can contain variables. The no-duplicate requirement expressed above cannot be enforced for non-ground objects. This does not mean that the maintenance of representation invariants in Denali is not a viable strategy. Rather, the class of invariants that can be enforced is different. For example, one invariant in force in the current example is that *bags* that are not of mode *enum* are not valid representations unless they are variables.

4.3.4 Unification predicates

The unification procedure for an explicitly implemented cluster is written as an ordinary predicate over pairs of objects of the representation sort. It is natural to implement a unification procedure, which maps pairs of objects to sets of substitutions, as a unification predicate, which maps pairs of objects to sequences of substitutions. The predicate that implements unification, in this case *setUnify*, is identified in the cluster implementation.

The language implementation is responsible for combining the unification predicate of each explicitly implemented cluster, and the synthesized unification procedure of each implicitly implemented cluster, into an overall unification procedure for the entire program. We will describe how this is done in Chapter 6.

This combining process treats variables as a special case. If one of the pair of terms to be unified is a variable, and the other argument does not contain that variable, then the trivial unification is done automatically. Consequently, the unification predicate need not be coded for this case. Since no moded *set* in the present example can contain a *set* variable, the unification of variables need not be treated explicitly at all in *setUnify*.

The implementation of *setUnify* given in the *set* cluster operates by pairing off elements of each representation. Since a single element can be paired with more than one element, elements are deleted nondeterministically.

To illustrate, consider the unification of the two sets $insert(1, insert(N, empty))$ and $insert(1, empty)$. These sets are mapped into *bag* terms and their unification is expressed as the query

$$\leftarrow \mathrm{setUnify}(\mathrm{cons}(1, \mathrm{cons}(N, \mathrm{nil})), \mathrm{cons}(1, \mathrm{nil})).$$

Of the four clauses in the definition of *setUnify*, only the last one leads to a successful evaluation in this case. It reduces the query above to

$$\leftarrow \mathrm{setUnify}(\mathrm{cons}(N, \mathrm{nil}), \mathrm{cons}(1, \mathrm{nil})),$$

which can eventually be solved by the substitution $\langle N/1 \rangle$.

We could not have implemented sets by implicit implementation, because the underlying equational theory of unification,

$$\mathrm{insert}(X, \mathrm{insert}(Y, S)) = \mathrm{insert}(Y, \mathrm{insert}(X, S))$$
$$\mathrm{insert}(X, \mathrm{insert}(X, S)) = \mathrm{insert}(X, S),$$

is not one for which a unification algorithm is known.

4.4 Comparison of implementation methods

Because of the way unification is treated, implicit implementation stands as a hybrid implementation approach. An implicitly implemented cluster shares characteristics with built-in abstractions and user-defined abstractions. It permits the presentation of a single equational theory in any number of ways by combining it with a variety of denotations, modes, and exported predicates. It would not be possible to provide every such variation as a built-in abstraction.

Implicit implementation is possible only when every abstract object has a denotation and the presented equational theory is one for which an algorithm is incorporated into the language. Its advantage is that the denotation and unification portions of an interface are realized by default. There are, nonetheless, two serious drawbacks, one methodological and one pragmatic.

First, because the implementation method is tied to the denotation scheme, implementations cannot be changed without affecting the denotations. Since the denotation scheme is part of the interface, this violates the principle of encapsulation.

Further, the programmer is limited to a single strategy for representing abstract objects. Experience with existing languages has shown that one of the most important aspects of implementing a data abstraction is selecting a suitable representation. This choice directly affects ease of implementation and efficiency of execution. Using the denotation as the representation is particularly restrictive since the function symbols used to build denotations are uninterpreted. The fact that denotation terms are unified within the empty theory limits their expressive power as concrete representations.

Even though a programmer may be willing to accept the limitations of the implicit implementation approach, it is not possible to use it for all abstractions. The set abstraction, for example, cannot be realized as a implicit implementation since its equational theory is one for which no unification algorithm is known. The programmer has no choice but to explicitly code a unification predicate. This is facilitated by using mode restrictions to simplify the task.

One distinctive aspect of the explicit implementation approach is that not all abstract objects need possess denotations. We do not require the programmer to define complete denotation schemes because it is not always necessary to denote arbitrary objects. There is probably no need to directly denote symbol tables, for example, as they are usually best constructed incrementally through the solution of literals. The programmer should be careful, however, that no non-denotable object is ever part of the solution of a query. If so, it will not be possible to translate that object back into a denotation for presentation to the user.

The drawback of explicit implementation is that it involves considerably more work than does implicit implementation. The problem of implementing unification predicates is particularly subtle. The ability to exploit built-in implementations of unification is a tremendous advantage, and should be exploited whenever feasible.

4.5 Summary of abstraction in Denali

We conclude our discussion of the design of Denali by summarizing the roles of predicate and data abstraction in reasoning about Denali programs.

We will adopt the point of view of the client of some set of abstractions. The interface of each abstraction contains the information needed by the client to construct syntactically correct applications. To construct semantically correct applications, the client must also know the meaning of each abstraction as given by its specification. To ensure that an application is modular, the client must rely only upon the specified meaning of each abstraction, and not upon the idiosyncrasies of a particular implementation.

Given a well-moded literal, the implementation of a predicate abstraction yields a sequence of substitutions. In a correct implementation, this sequence is a complete subset of some larger set of substitutions fixed by the specification. This specification is a relation; the set of solutions of a literal can be defined relative to the specifying relation. The identity of this relation, along with the moding of the predicate and the fact that a correct implementation will enumerate a complete subset, is all that a client need know. The composition and order of the particular sequence that is generated is immaterial.

The client of a data abstraction must know the set of objects provided by the abstraction. All other aspects of a data abstraction are defined in terms of these objects. The client must also know

- which abstract object is denoted by each term of the denotation scheme,
- what set of objects is associated with each mode name,
- the equational theory that specifies unification, and
- the meaning of each predicate provided by the abstraction.

The specific implementation that is chosen for unification is an irrelevant detail of the realization of a data abstraction. This choice will be visible in the sequence of unifiers yielded for pairs of terms. The client needs to know only that the implementation of unification is complete.

In conventional languages that support both data abstraction and the sharing of objects, it is possible to betray an important aspect of an implementation by exposing its representation. When the representation is exposed, the value of an abstract object can be altered by means other than the abstract operations. Even though Denali supports a form of sharing—through shared variables—it is not possible to expose the representation of a cluster. This is because objects can change only by becoming further instantiated and cannot be arbitrarily mutated.

If all of the implementations that compose a program are correct relative to their specifications, then the meaning of any query interpreted relative to this program is partially determined by the specifications. Assuming that it terminates, the evaluation of a query will either yield a complete solution or report a mode failure. The incidence of solution or mode failure is a property of the specifications and is independent of the implementations. Ensuring this property was an explicit goal of our research, and is a consequence of Theorem 5.15, which we prove in Chapter 5. Implementation decisions can show through only in the particular complete solution obtained for a query.

Suppose that a client program uses some set of abstractions. This program is modular if its proof of correctness can be based upon the specifications of the set of abstractions. This will be the case so long as the client makes no assumptions about the visible implementation details of the used abstractions. A modular program is correct so long as its used abstractions satisfy their specifications.

5 Moded equational resolution

In this chapter we give a rigorous development of moded equational resolution. This supplements and extends the informal treatment of Chapter 3, and provides us with the vocabulary needed to define the meaning of Denali programs in Chapter 7. The chapter is divided into five sections.

We begin in Section 5.1 by presenting a summary of the background that we expect of the reader. This background includes equational theories, the theory of equational definite clause programs, and related completeness results. This serves to introduce the central issue of the chapter: the development of efficient procedures for interpreting equational definite clause programs.

We continue in Section 5.2 by describing ESL resolution, a class of procedures for interpreting such programs. ESL procedures extend the SL procedures commonly used to interpret logic programs [Kowalski 71] in three ways. They deal with equational theories, they interleave resolution and overlap steps, and they afford a formal way of characterizing both completeness and incompleteness.

This final aspect of ESL procedures is important, because complete equational resolution procedures are difficult to implement. This is primarily because they require complete equational unification procedures. In Section 5.3 we introduce a subclass of the

ESL procedures called the moded ESL procedures. While incomplete, these procedures have the virtue of reducing the demands upon the underlying unification procedures, thus making them more easily realized.

In Section 5.4 we define and analyze a series of three moded ESL procedures. The goal is to develop a procedure whose degree of incompleteness can be precisely described and thus documented. The procedure that we ultimately develop, W-ESL resolution, forms the basis for the semantics of Denali.

We conclude in Section 5.5 by reviewing the chain of improvements that lead from ESL to W-ESL resolution.

5.1 Background

In this section we give a concise grounding in definite clause programs with equality. The discussion in the remainder of this chapter is predicated upon this background.

In Section 5.1.1, we describe equational theories and then define the theory of definite clause programs with equality. We continue in Section 5.1.2 by discussing the problem of finding equationally complete solutions to equations and queries. The terminology surrounding equational theories follows that of [Huet 80], while the terminology concerning the theory of definite clause programs follows that of [Emden 76] and [Jaffar 84].

5.1.1 Definite clause programs with equality

Let $\mathbf{V}$ be a countably infinite set of variables, $\mathbf{F}$ a finite set of function symbols, $\mathbf{P}$ a finite set of predicate symbols, and $\mathbf{S}$ a finite set of sorts.

Associated with every variable $v \in \mathbf{V}$ is a sort from $\mathbf{S}$. Let $\mathbf{V}_S$ be the set of all variables of sort S. The assignment of sorts to variables is such that for each sort S, $\mathbf{V}_S$ is infinite.

Associated with each function symbol $f \in \mathbf{F}$ is an arity n and a signature of $n+1$ sorts. A *term* of sort S is either a variable of sort S or is of the form $f(t_1, \ldots, t_n)$, where f has signature

$f\colon S_1, \ldots, S_n \to S$

and each of the t_i is a term of sort S_i. A term is *ground* if it contains no variables. The set of all terms is denoted $\mathbf{T}$, and the sort of a term $t \in \mathbf{T}$ is denoted by $sort(t)$. The set of all terms of sort S is denoted $\mathbf{T}_S$; to avoid problems with empty sorts we assume that each such set contains at least one ground element.

Associated with each predicate symbol $P \in \mathbf{P}$ is an arity n and a signature of n sorts. A *literal* is of the form $P(t_1, \ldots, t_n)$, where the signature of P is

$P\colon S_1, \ldots, S_n$

and each of the t_i is a term of sort S_i. The set of all literals is denoted by $\mathbf{L}$. Literals, which are headed by predicate symbols, are distinct from terms.

A *substitution* is a mapping from variables to terms denoted by a finite set of identically sorted variable/term pairs, e.g. $\langle X/2, L/nil\rangle$. All variables outside the set are implicitly mapped to themselves. Substitutions can be extended in the natural way to a homomorphism from terms to terms and from literals to literals. For a substitution, σ, the set $\{v \mid \sigma v \neq v\}$, denoted $\mathcal{D}(\sigma)$, is called its *domain*.

Two substitutions can be *composed* to form a new substitution, i.e., $(\sigma_1 \circ \sigma_2)(t) = \sigma_1(\sigma_2(t))$. The domain of a substitution can be *restricted* to a set V to obtain a new substitution, i.e., $\sigma\restriction_V(v)$ is $\sigma(v)$ if $v \in V$ and is v otherwise.

An *equation* is any pair of equally sorted terms, $r = t$. The *equational theory* presented by a set of equations E is the smallest equivalence relation E^* over $\mathbf{T}$ that contains E and is closed under the following two rules of inference. (We write $r =_E t$ whenever $r = t \in E^*$.)

- (substitution) $r =_E t \Rightarrow \sigma r =_E \sigma t$
- (equality) $r_1 =_E t_1 \cdots r_n =_E t_n \Rightarrow f(r_1, \ldots, r_n) =_E f(t_1, \ldots, t_n)$

Term equality can be extended to substitutions. We write $\sigma_1 =_E \sigma_2$ whenever for every variable v, $\sigma_1 v =_E \sigma_2 v$. Term equality can also be used to induce an ordering relation over substitutions. We say that σ_1 is *more general than* σ_2 *modulo* E, and write $\sigma_1 \leq_E \sigma_2$, whenever σ_2 is an instance of σ_1. This is the case whenever there exists a substitution τ such that $(\tau \circ \sigma_1)\restriction_V =_E \sigma_2\restriction_V$, where V is the domain of σ_1.

A clause is a sequence of literals of the form

$L \leftarrow L_1, \ldots, L_n$,
$\leftarrow L_1, \ldots, L_n$, or
L.

The literal L is the *head* and the sequence $L_1, \ldots, L_n$ is the *body*. Both are optional. If the head of a clause is not present, it is a *query*; otherwise, it is a *definite clause*. A definite clause with an empty body is a *fact*.

Let H be a set of definite clauses and let E be a set of equations. Then the *definite clause theory with equality* of H over E, denoted H_E^*, is the smallest set of literals that contains the facts of H and is closed under the following three rules of inference:

- (substitution) $L \in H_E^* \Rightarrow \sigma L \in H_E^*$
- (equality) $r_1 =_E t_1 \cdots r_n =_E t_n \wedge P(r_1, \ldots, r_n) \in H_E^* \Rightarrow P(t_1, \ldots, t_n) \in H_E^*$
- (inference) $L \leftarrow L_1, \ldots, L_n \in H \wedge \sigma L_1 \in H_E^* \cdots \sigma L_n \in H_E^* \Rightarrow \sigma L \in H_E^*$

The sort system we have adopted is particularly simple, and is an instance of the more general sort system described in [Schmidt-Schauss 86]. The *unsorted* case is, in turn, the special case of our system that occurs when the sort universe $\mathbf{S}$ contains exactly one element. Although most of the existing results concerning equational and first-order theories are based upon the unsorted case, they extend without difficulty to the many-sorted case described above.

5.1.2 Equational completeness

Let Σ_1 and Σ_2 be sets of substitutions and let E be a set of equations. Σ_1 is an *equationally complete subset* of Σ_2 with respect to E whenever

- $\Sigma_1 \subseteq \Sigma_2$
- $\forall \sigma_2 \in \Sigma_2 \, \exists \sigma_1 \in \Sigma_1$ such that $\sigma_1 \leq_E \sigma_2$

A substitution σ is called an *equational unifier* of terms r and t with respect to E, or E-unifier, whenever $\sigma r =_E \sigma t$. Let $U_E(r, t)$ denote the set of all E-unifiers of r and t. A set of substitutions is a *complete set of E-unifiers* of r and t if it is an equationally complete subset of $U_E(r, t)$.

A complete equational unification procedure is one that enumerates, for any two terms, a complete set of unifiers. Robinson's classical unification algorithm is a complete procedure for the special case in which the set E is empty [Robinson 65]. We will survey existing unification procedures for other theories in Chapter 6.

Let L be a literal. A substitution σ is called an *equational satisfier* of L with respect to a set of equations E and a set of definite clauses H whenever $\sigma L \in H_E^*$. Let $S_E^H(L)$ denote the set of all H_E-satisfiers of L. A set of substitutions is a *complete set of satisfiers* of L if it is an equationally complete subset of $S_E^H(L)$.

We can extend equational satisfaction to queries. Let Q be $\leftarrow L_1, \ldots, L_n$. $S_E^H(Q)$ denotes the set of all substitutions σ such that $\sigma \in S_E^H(L_1), \ldots, \sigma \in S_E^H(L_n)$. A set of substitutions is a complete set of satisfiers of Q if it is an equationally complete subset of $S_E^H(Q)$.

An *equational definite clause program* is a pair $(E; H)$, where E is a set of equations and H is a set of definite clauses. A complete equational satisfaction procedure is one that enumerates, given any such program and any query Q, an equationally complete subset of $S_E^H(Q)$. We will be examining one class of complete satisfaction procedures, the equational resolution procedures, in the remainder of this chapter.

5.2 ESL resolution

In this section we define a class of equational resolution procedures called the ESL resolution procedures. This definition forms the basis for our development of a spectrum of equational resolution procedures in subsequent sections.

ESL resolution procedures operate by constructing and searching ESL trees. Our ESL trees are based upon the kinds of trees constructed by the SL resolution procedure [Kowalski 71]. SL resolution is a complete satisfaction procedure defined with respect to the empty theory. It is a linear resolution strategy that imposes, at each step, characteristic constraints upon the selection of the next literal for reduction. Although ESL trees are structurally distinct from SL trees, many of the differences are pedagogically motivated. There are, however, three substantive differences.

First, ESL trees are defined with respect to arbitrary equational theories. We discussed the ramifications of incorporating non-empty equational theories into resolution in Chapter 3. This extension is straightforward, made so by the pioneering work of

Plotkin on the completeness of equational resolution [Plotkin 72].

Second, ESL trees are more flexible concerning the order in which unification must be performed. We illustrated the benefits of interleaving unification and overlap steps in Chapter 3. This additional flexibility does not compromise completeness, and we will be able to exploit it to considerable advantage.

Finally, we deal explicitly with incomplete ESL trees. This allows us to define incomplete ESL resolution procedures, while at the same time understanding and characterizing the limits of their completeness. Permitting controlled incompleteness is the centerpiece of our approach to equational resolution.

Our development of ESL resolution is organized as follows. We begin by giving the two reduction rules that together form the basis for the construction of ESL trees. In Section 5.2.1 we define the *overlap reduction rule*. This rule is a means of reducing the problem of solving a literal to that of solving a set of queries. Next, in Section 5.2.2, we define the *selection reduction rule*. This rule provides a way of reducing the problem of solving a query to that of solving a set of simpler queries. It assumes the existence of a means of solving literals.

These two reduction rules can be combined to obtain a comprehensive strategy for solving queries with respect to equational definite clause programs. This strategy involves the construction and search of ESL trees, in which each edge represents an application of one of the two reduction rules. In Section 5.2.3 we define ESL trees.

The ESL resolution procedures that we will eventually discuss operate by constructing ESL trees. They differ according to the kinds of trees that they construct. We conclude in Section 5.2.4 by describing how individual ESL procedures are defined. We specify one prototypical procedure, the N-ESL procedure, and prove its correctness.

Much of the discussion in this section is conducted relative to an arbitrary equational definite clause program $(E; H)$. However, a number of concrete examples are also given. In these cases, E should be taken to be the empty equational theory and H to be the set of clauses given below.

$$\begin{array}{l}\mathrm{ord}(\mathrm{W}\cdot\mathrm{nil}).\\ \mathrm{ord}(\mathrm{X}\cdot\mathrm{Y}\cdot\mathrm{L}) \leftarrow \mathrm{X}>\mathrm{Y}, \mathrm{ord}(\mathrm{Y}\cdot\mathrm{L}).\\ 3>2.\\ 3>1.\\ 2>1.\end{array}$$

These clauses define the two predicate symbols $>$ and *ord*. The symbol $>$ defines a total ordering over the elements 3, 2, and 1. The predicate *ord*, which is defined over lists of these elements, tests whether its argument is ordered with respect to $>$. Consequently, its meaning is the set of all substitutions that render its argument ordered.

5.2.1 Overlap reduction

The overlap reduction rule is a way of reducing the problem of solving a literal to the problem of solving a set of queries. There are two cases, which we will consider separately. The first deals with the reduction of conventional literals, and the second deals with the reduction of the distinguished unification literals that are produced by the first case.

Conventional literals

The first case handles literals of the form $P(t_1, \ldots, t_n)$. Let C be a clause that is headed by the predicate symbol P:

$$P(r_1, \ldots, r_n) \leftarrow L_1, \ldots, L_m.$$

(We assume that the literal and the clause contain disjoint sets of variables. The variables of the clause can be systematically renamed if necessary.) The query Q,

$$\leftarrow \mathit{unify}(t_1, r_1), \ldots, \mathit{unify}(t_n, r_n), L_1, \ldots, L_m,$$

is the overlap reduction of $P(t_1, \ldots, t_n)$ using C. Any satisfier of Q is also a solution of the original literal.

To illustrate, consider the literal *ord*($\mathit{3}\cdot Z\cdot \mathit{nil}$). One of the clauses that defines *ord* is

$$\mathrm{ord}(\mathrm{X}\cdot\mathrm{Y}\cdot\mathrm{L}) \leftarrow \mathrm{X}>\mathrm{Y}, \mathrm{ord}(\mathrm{Y}\cdot\mathrm{L}).$$

The *overlap reduction* of the literal above using this clause is the query

$$\leftarrow \mathrm{unify}(3\cdot\mathrm{Z}\cdot\mathrm{nil}, \mathrm{X}\cdot\mathrm{Y}\cdot\mathrm{L}), \mathrm{X}>\mathrm{Y}, \mathrm{ord}(\mathrm{Y}\cdot\mathrm{L}).$$

Overlap reduction has the following completeness property. Suppose that H contains exactly n clauses, $C_1, \ldots, C_n$, that define the head symbol of some literal L. This means

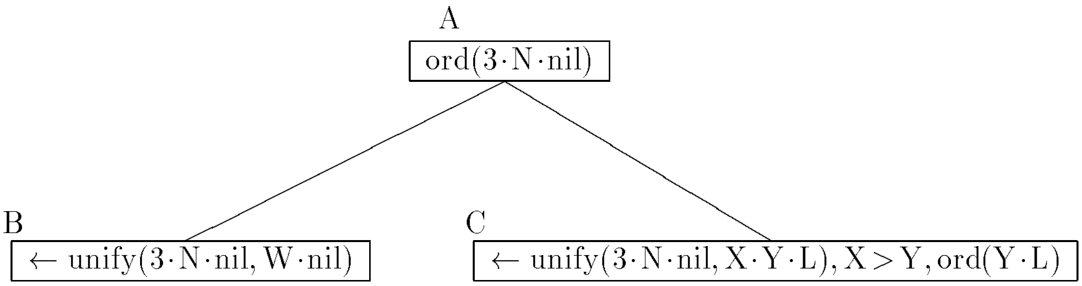


Figure 5.1: Overlap reduction

that it is possible to form n distinct overlap reductions of L, $Q_1, \ldots, Q_n$. Suppose that we obtain a complete set of satisfiers Σ_i for each query Q_i. Then the set

$$\bigcup_i \left\{\sigma\restriction_{\mathcal{V}(L)} \mid \sigma \in \Sigma_i\right\}$$

is a complete set of satisfiers for L.

Unification literals

The predicate symbol *unify* introduced in the definition above is a reserved symbol. It is introduced so that we can construct literals that encode, and thus defer, unification. The meaning of a *unify* literal is defined by

$$S_E^H(\mathit{unify}(t_1, t_2)) \equiv U_E(t_1, t_2).$$

Solving a *unify* literal is thus equivalent to unifying its arguments.

The reduction of *unify* literals is handled by the second case of overlap reduction. Here, no reduction to queries is possible or necessary. Any complete set of E-unifiers of t_1 and t_2 constitutes a complete set of satisfiers of $\mathit{unify}(t_1, t_2)$. Such a set can be determined by an appropriate E-unification procedure.

Solving literals

Assuming that we can obtain complete solutions for queries, the completeness properties described above suggest a procedure for obtaining complete solutions to literals. The operation of such a procedure is suggested in Figure 5.1, which expresses the complete solution to the literal $\mathit{ord}(3 \cdot N \cdot \mathit{nil})$.

There are two kinds of nodes in Figure 5.1. Node A contains a literal, while nodes B and C contain queries. The literal in node A is the literal to be solved, and each of the children contains an overlap reduction of this root literal. Node B contains the overlap reduction of the root with the first clause of the example program, and node C contains the overlap reduction of the root with the second clause.

Let $sols(Q)$ denote a complete set of satisfiers of a query Q. Then a complete set of satisfiers of the root literal L of a tree T, denoted $subs(T)$, can be obtained as follows. Let $Q_1, \ldots, Q_n$ be the children of the root literal. Then

$$subs(T) = \bigcup_i \left\{ \sigma\restriction_{\mathcal{V}(L)} \mid \sigma \in sols(Q_k) \right\}$$

A complete set of satisfiers of the root literal can be obtained by first taking the union of the complete sets of satisfier of the child queries, and then restricting the domains of the substitutions in this set to the variables of the root literal. In Figure 5.1, $sols(B)$ is empty, and $sols(C)$ is

$$[\langle X/3,\ N/2,\ Y/2,\ L/1 \cdot nil\rangle,\ \langle X/3,\ N/1,\ Y/1,\ L/nil\rangle].$$

Consequently, a complete solution to the root literal is

$$[\langle N/2\rangle,\ \langle N/1\rangle].$$

5.2.2 Selection reduction

The selection reduction rule is a way of reducing the problem of solving a query to the problem of solving a simpler set of queries. It assumes that a method exists for solving literals. Selection reduction has both a base case and a recursive case, which we will consider separately.

Base case

The base case applies when the input query consists of exactly one literal. Let Q be the query $\leftarrow L$, and let σ be any satisfier of the literal L. Then σ is also a satisfier of Q. Furthermore, if Σ is a complete set of satisfiers of L, then σ is also a complete set of satisfiers of Q.

Recursive case

The recursive case applies when the input query contains more than one literal. Let Q be the query $\leftarrow L_1, \ldots, L_n$, and let σ be a satisfier of one of its literals L_k. The query

$$\leftarrow \sigma L_1, \ldots, \sigma L_{k-1}, \sigma L_{k+1}, \ldots, \sigma L_n,$$

obtained by removing L_k and applying σ, is the *selection reduction* of Q at k using σ. Call this query Q'. If ρ is a satisfier of Q', then $\rho \circ \sigma$ is a satisfier of Q.

To illustrate, let P be the query

$$\leftarrow \mathrm{ord}(3 \cdot \mathrm{Z} \cdot \mathrm{nil}), \mathrm{ord}(\mathrm{Z} \cdot 1 \cdot \mathrm{nil}).$$

One satisfier of the first literal of P is $\langle Z/\mathit{2} \rangle$. The selection reduction of P at 1 using $\langle Z/\mathit{2} \rangle$ is the query

$$\leftarrow \mathrm{ord}(2 \cdot 1 \cdot \mathrm{nil}).$$

Since the empty substitution is a satisfier of this reduced query, we can conclude by selection reduction that $\langle\rangle \circ \langle Z/\mathit{2} \rangle$, or simply $\langle Z/\mathit{2} \rangle$, is a satisfier of the original query P.

The recursive case has the following completeness property. Returning to the abstract definition above, suppose that we obtain a complete set of satisfiers Σ of the literal L_k. Suppose also that, for each substitution $\sigma_i \in \Sigma$, we obtain a complete set of satisfiers R_i of the selection reduction of Q at k using σ_i. In this case, the set

$$\bigcup_i \left\{ (\rho \circ \sigma_i)\restriction_{\mathcal{V}(Q)} \mid \rho \in \mathrm{R}_i \right\},$$

is a complete set of satisfiers of the original query Q.

Solving queries

Assuming that we can solve literals, this property suggests a complete procedure for solving queries. The procedure involves constructing and searching trees of the form illustrated in Figure 5.2, which illustrates the solution of the query

$$\leftarrow \mathrm{ord}(3 \cdot \mathrm{N} \cdot \mathrm{nil}), \mathrm{ord}(3 \cdot \mathrm{M} \cdot \mathrm{nil}), \mathrm{N} > \mathrm{M}.$$

Each node (Q, k, Σ) in Figure 5.2 contains three components. The upper half contains a query Q, the lower left contains an integer k, and the lower right contains a sequence of substitutions Σ. The three components are related as follows. Σ is a complete set of

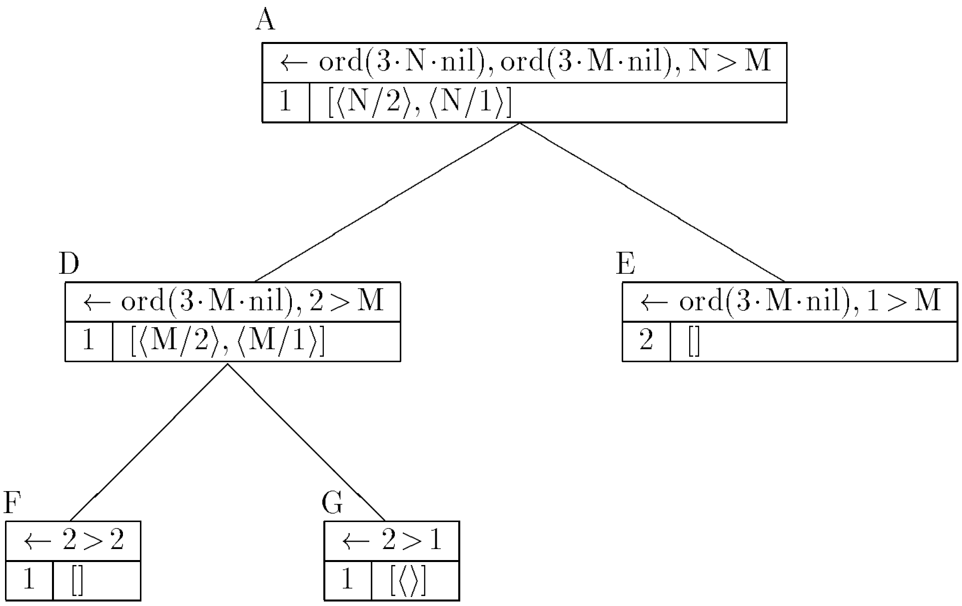


Figure 5.2: Selection reduction

satisfiers of the k^{th} literal of Q.

For example, in node A the sequence $[\langle N/2\rangle, \langle N/1\rangle]$ is a complete set of satisfiers of the literal $ord(3 \cdot N \cdot nil)$. In node E the empty sequence is a complete solution to $1 > M$, since the literal has no satisfiers. In node G the sequence containing the empty substitution is a complete solution to $2 > 1$.

Each child is a selection reduction of its parent. Each parent node (Q, k, Σ) has one child corresponding to each member of Σ. That is, the i^{th} child is the selection reduction of Q at k using the i^{th} element of Σ.

In both nodes A and D, the indexed literal has two satisfiers. Consequently, each node has two children. Literal 2 of node E has no satisfiers and thus no children. Nodes F and G have no children because their queries contain only one literal. They represent applications of the base case of selection reduction.

Let T be a tree with root node (Q, k, Σ). A complete set of satisfiers for Q, denoted $sols(T)$, can be obtained recursively as follows. If the root node has no children, then

$$sols(T) = \Sigma.$$

Otherwise the root child has n children, $T_1, \ldots, T_n$, and its sequence of substitutions contains n values, $\sigma_1, \ldots, \sigma_n$. In this case,

$$sols(T) = \bigcup_i \left\{(\rho \circ \sigma_i)\restriction_{\mathcal{V}(Q)} \mid \rho \in sols(T_i)\right\},$$

There is one solution in this set for every path from root node to nonempty leaf node.

In Figure 5.2, there is only one such path. The value of $sols(G)$ is $\{\langle\rangle\}$, so the value of $sols(D)$ is $\{\langle\rangle \circ \langle M/1\rangle\}$, or $\{\langle M/1\rangle\}$. Similarly, the value of $sols(A)$ is $\{\langle M/1,\ N/2\rangle\}$.

5.2.3 ESL trees

The two rules just discussed can be combined to form a single resolution strategy for finding complete solutions to queries. The selection reduction rule is predicated upon the ability to solve literals, while the overlap reduction rule is based upon the ability to solve queries. Furthermore, both have nonrecursive base cases. We can consequently combine the two in a mutually recursive fashion.

The combined approach to solving a query Q is as follows. First, select an arbitrary literal of Q and solve it using overlap reduction. Next, use the satisfiers obtained this way to form and solve the selection reductions of Q. Finally, combine the satisfiers of the selected literal with the satisfiers of the reduced queries to obtain a complete solution to Q.

It is convenient to organize the series of reductions required to solve a query into a tree. Such trees, which we call ESL trees, are a combination of the two kinds of trees we constructed to illustrate overlap and selection reduction.

As a first example, we return to the following query, whose solution under selection reduction we illustrated earlier:

$$\leftarrow \mathrm{ord}(3\cdot \mathrm{N}\cdot \mathrm{nil}), \mathrm{ord}(3\cdot \mathrm{M}\cdot \mathrm{nil}), \mathrm{N} > \mathrm{M}.$$

An entire ESL tree for this query is large. We give in Figure 5.3 the nodes in the first two layers.

The tree in Figure 5.3 can be decomposed into two subtrees that we have seen previously. The subtree consisting of nodes A, B, and C is the overlap tree of Figure 5.1 that expresses the solution of the literal $ord(3\cdot N\cdot nil)$. The subtree consisting of nodes A, D, and E is identical to the first two levels of the selection tree of Figure 5.2.

As with the selection tree examined earlier, each node in this tree is divided into three

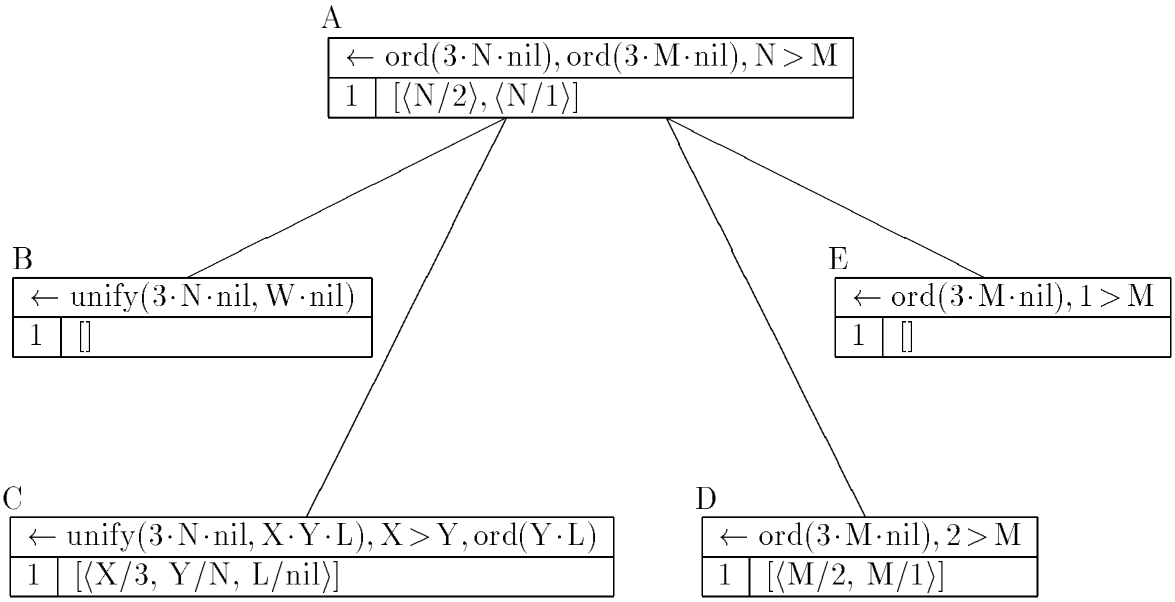


Figure 5.3: Upper two levels of an ESL tree

---

sections. The upper half contains a query Q. The lower left is an index k that identifies a literal L_k of Q. The lower right contains a set of substitutions Σ, which, as before, is a complete solution to the literal L_k. In this case, however, its value is derived from the remainder of the tree as described below.

The children that branch from the left side of a node T express the solution of the indexed literal. Accordingly, they are the overlap reductions of L_k and compute Σ. The value of Σ that they compute is exactly $subs(T)$ if we consider T to be just the node and its left children.

The children that branch from the right side are the selection reductions of Q at k using the members of Σ. The substitutions that they compute can be combined with Σ to obtain a complete solution to Q. For a tree T, this complete solution is exactly $sols(T)$ if we consider T to be just the node and its right children.

It is important to note the asymmetry between the left and right children of the root node. The left children are overlap reductions; the right children are selection reductions. A procedure for generating such a tree would have to compute at least some of the left children first, since the elements of Σ are needed to form the right children.

Despite their asymmetry, however, both left and right children are queries, and they

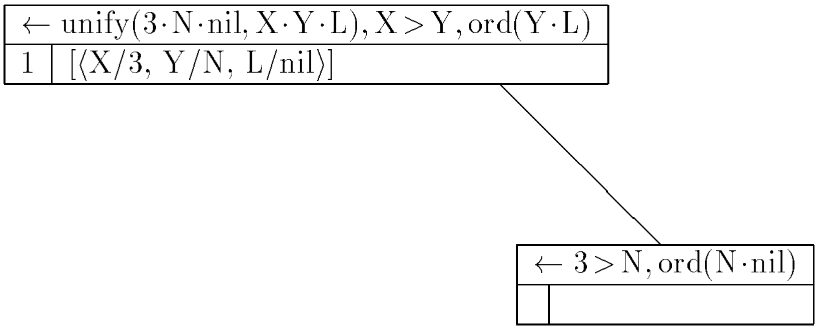


Figure 5.4: ESL tree with no left (overlap) children

can be recursively solved using the same techniques applied to the root. We will further illustrate the technique of ESL tree construction by giving three other nodes possessing differing combinations of left and right children. The branching structure of a node depends upon which case applies for each of the two reduction rules. The example we have just seen employs the recursive case for both rules.

The example in Figure 5.4 represents the solution of the query

$$\leftarrow \mathrm{unify}(3\cdot \mathrm{N}\cdot \mathrm{nil}, \mathrm{X}\cdot \mathrm{Y}\cdot \mathrm{L}), \mathrm{X} > \mathrm{Y}, \mathrm{ord}(\mathrm{Y}\cdot \mathrm{nil}).$$

This is the query that appears in node C of Figure 5.3.

Because the indexed literal is a *unify* literal, the base case of overlap reduction applies. This means that the root node need have no left children. The set Σ can be obtained directly by unifying the arguments of the literal. Since there is one unifier in the complete set, there is one right child.

The next example, in Figure 5.5, is part of the tree that expresses the solution of the singleton query $\leftarrow ord(2\cdot nil)$.

The selected literal in this instance is not a *unify* literal, so its solution must be computed recursively. There are two left children corresponding to the two possible overlap reductions. A single solution, the empty substitution, is obtained this way. Since the query contains a single literal, the base case of selection reduction applies and there are no right children.

In the final example, shown in Figure 5.6, the base case for both reduction rules applies. Here, the indexed literal is a *unify* literal, and the query contains but one literal.

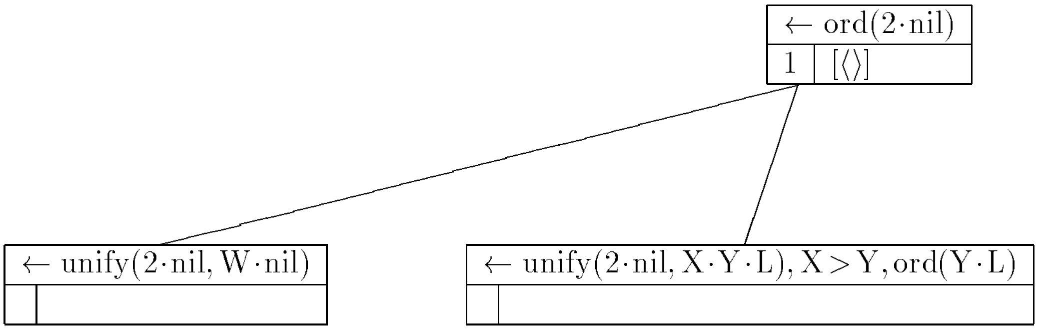


Figure 5.5: ESL tree with no right (selection) children

The result is a terminal node.

5.2.4 N-ESL resolution

An ESL resolution procedure is one that solves an input query Q by constructing an ESL tree T for Q and searching it to obtain the members of $sols(T)$. Because ESL trees have potentially infinite depth and breadth, the implementation of an ESL resolution procedure must interleave the construction and searching phases to ensure that all branches are eventually considered. We will abstract from this aspect of the problem, however, and regard ESL resolution as nothing more than the problem of constructing ESL trees for programs.

A given program can have an arbitrary number of ESL trees. This is because an ESL resolution procedure enjoys two degrees of freedom when constructing trees. First, it must select the indexed literal at each node. We will see that this is the most important determinant of the character of the procedure. Second, when the indexed literal is a *unify* literal, the procedure must choose a complete set of unifiers. This aspect of the resolution procedure's behavior is determined by the underlying unification procedure.

We will specify ESL resolution procedures by placing constraints upon these two de-

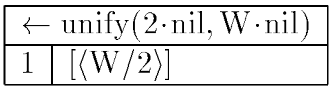


Figure 5.6: ESL tree with no children

grees of freedom.

Restrictions upon unification specify any properties, beyond completeness, required of the underlying unification procedure. We might, for example, require that a minimal number of unifiers always be produced. In practice, we will impose only the most general kinds of restrictions. Unless stated otherwise, we will assume only that the underlying unification procedure is complete.

Restrictions upon index selection are expressed with selection rules. A *selection rule* maps a query to a possibly empty set of literals. If R is a selection rule and Q a query, then $R(Q)$ must be a subset of the literals of Q. An ESL tree respects a selection rule R if the following two conditions hold at every node containing a query Q.

- If $R(Q)$ is nonempty, then the indexed literal must lie in $R(Q)$.
- If $R(Q)$ is empty, then the indexed literal must be 0.

We have not previously permitted the appearance of 0 as an index in a node. In this case we say that the node and the tree that contains it are *blocked.* If a node is blocked it must have no children and its set of substitutions must be empty.

An ESL tree expresses the solution of the query Q at which it is rooted. No guarantee can be made about the completeness of a blocked tree. However, because unblocked ESL trees are equivalent to SL trees whenever E is empty, the following lemma follows from the completeness result of [Kowalski 71].

Lemma 5.1 (Kowalski) *Let T be an unblocked* ESL *tree for the query Q with respect to a program $(E;H)$, where E is empty. Then sols(T) is a complete subset of $S_E^H(Q)$.*

Plotkin [Plotkin 72] treats the general problem of converting complete resolution strategies into complete strategies for equational theories. His results justify removing the condition upon E in Lemma 5.1.

Lemma 5.2 (Plotkin) *Let T be an unblocked* ESL *tree for Q with respect to the program $(E;H)$. Then sols(T) is a complete subset of $S_E^H(Q)$.*

Suppose that an ESL resolution procedure is based upon a selection rule R. The goal of that procedure, when given a query Q, is to produce an unblocked ESL tree for Q that respects R. If it always does this for any query, then the procedure is complete.

We will design a number of selection rules in the sections that follow. All of our selection rules will have both descriptive names and a single-letter abbreviation. Let R be a selection rule. We will refer to the subset of the ESL trees that respect R as R-ESL trees, and to the procedures based upon R as R-ESL procedures.

The most general possible selection rule is the nondeterministic selection rule (N), which permits the selection of any literal. Since all literals are selectable under this rule, there can be no blocked N-ESL trees. Thus, the N-ESL trees are exactly the unblocked ESL trees, and the following theorem follows directly from Lemma 5.2.

Theorem 5.3 N-ESL *resolution is complete for all queries and all equational definite clause programs.*

In the next section we will motivate the need for a more restrictive selection rule. In designing this selection rule, the dominant theme will be the tension between the twin goals of improving the performance of resolution and avoiding the possibility of blocked trees.

5.3 Moded resolution

Despite Theorem 5.3, N-ESL resolution is not a panacea. Its completeness depends upon the completeness of the underlying unification procedure. We will see in Chapter 6 that efficient and complete equational unification procedures are difficult, and sometimes impossible, to obtain. We show in this section that by restricting the domain of an equational resolution procedure, the demands upon the underlying unification procedure can be reduced.

For a given equational theory, the cost of performing a unification step depends upon the pair of terms being unified. Consequently, the complexity of a unification procedure, as well as the difficulty of implementing it, can be reduced by restricting its domain. We gave examples supporting this observation in Chapter 3.

Restricting unification procedures in this way sacrifices one dimension of their completeness. As a result, a resolution procedure based upon restricted unification will itself be incomplete. This incompleteness can be expressed as a restriction upon the domain over which an ESL resolution procedure is complete. Because the form of queries and programs can be circumscribed, however, such domain restrictions are tolerable. We can thus contemplate basing ESL resolution procedures upon restricted unification procedures.

We proceed in three steps. In Section 5.3.1 we extend the definitions of moded base and mode to account for equational theories. These definitions were originally given in Chapter 2 with respect to the empty theory. In Section 5.3.2 we describe how moded bases can be used to express domain restrictions upon both equational unification and resolution procedures. Finally, in Section 5.3.3, we show how unification procedures that are restricted by moded bases can be incorporated into ESL resolution procedures. These ESL procedures are based upon moded selection rules. The result is a more easily implemented class of resolution procedures with restricted, but documented, domains.

5.3.1 Equational modes and moded bases

The definitions of moded base and mode, originally presented in Chapter 2, must be altered slightly to account for equational theories. We discuss these changes below.

A *moded base* is any set of terms that is contains all of the variable terms and is closed under moded instantiation, equality, and unification.

- (instantiation) $moded(\sigma) \wedge moded(t) \Rightarrow moded(\sigma t)$.
- (equality) $moded(r) \wedge r =_E t \Rightarrow moded(t)$
- (unification) $moded(r) \wedge moded(t) \Rightarrow \exists\, \Sigma$ s.t.
  Σ is a complete set of E-unifiers of r and t, and
  $\sigma \in \Sigma \Rightarrow moded(\sigma)$.

Closure under moded instantiation is as before. Closure under equality is a new condition; it was not needed previously because distinct terms are always unequal in the empty theory. Closure under unification has been modified to deal with complete sets of E-unifiers rather than most general unifiers.

A *mode* M of sort S is any set of moded terms of sort S that is closed under moded instantiation and equality.

- (instantiation) $moded(\sigma) \wedge t \in M \Rightarrow \sigma t \in M$.
- (equality) $r \in M \wedge r =_E t \Rightarrow t \in M$.

Closure under instantiation is as before; closure under equality has been added.

We commented in Chapter 2 that, if the underlying equational theory is empty, any syntactic mode presentation is guaranteed to define a moded base and a set of modes. This guarantee cannot be made *a priori* in the equational case. It is possible for a mode to be not closed under equality, and it is possible for the moded base to be not closed under unification. A simple syntactic check that guarantees the absence of the first abnormality is outlined below. A check for the second problem, unfortunately, remains an open problem.

The check for closure under equality operates as follows. We must guarantee that the left- and right-hand sides of each equation in the presentation of the underlying equational theory belong to identical sets of modes. We must also guarantee that this property is invariant under the application of substitutions. This can be verified by evaluating the modes of both sides of each equation under all possible assumptions for the modes of terms that can be substituted for the variables.

5.3.2 Using moded bases

We must be precise about how the domain restrictions upon unification procedures are expressed. Recall that an unrestricted unification procedure is specified by giving an equational theory. We will specify restricted unification procedures by giving, in addition, a moded base of terms. The moded base forms the domain of the unification procedure.

We adopt the convention that resolution is carried out with respect to both an equational theory and a moded base. Recall that a term is *moded* if it belongs to the moded base. Similarly, a literal is moded if each of its terms is moded; a clause is moded if each of its literals is moded; and a program is moded if each of its clauses is moded. A substitution is moded if it maps each variable in its domain to a moded term.

A *modally complete* unification procedure has two characteristics. It is complete for all pairs of terms from the moded base, and it produces only moded substitutions. (If the moded base is the set of all terms, modal completeness reduces to equational completeness.) Modally complete unification procedures exist for all pairs of moded bases and equational theories possessing complete unification procedures. We can make this guarantee because moded bases are closed under unification.

An ESL tree T is modally complete if T is complete and $sols(T)$ contains only moded substitutions. A resolution procedure is modally complete for a given query if it produces only modally complete trees for that query.

We will assume from this point that all unification procedures are modally complete. Lemma 5.4 establishes a sufficient condition for the modal completeness of ESL trees in this context.

Lemma 5.4 *Let T be an* N-ESL *tree for a query Q. If every selected unify literal in T is moded, then T is modally complete for P.*

Proof. It follows from the modal completeness of the underlying unification procedure that T is complete and only moded unifiers are produced. The instantiation property of moded bases guarantees that composing moded substitutions yields moded substitutions. It follows that $sols(T)$ contains only moded substitutions, implying that T is modally complete. □

One way to guarantee the hypothesis of Lemma 5.4, thus obtaining a modal completeness theorem for N-ESL resolution, is to define a static test upon programs. This test should report whether or not a program has any unmoded N-ESL trees. Given such a test, we can proscribe programs with unmoded N-ESL trees as ill-formed. This strategy is analogous to compile-time type checking.

Theorem 5.5 establishes one such check by showing that it is sufficient to restrict our attention to programs containing only moded terms..

Theorem 5.5 N-ESL *resolution is modally complete for programs and queries that contain only moded terms.*

Proof. An N-ESL tree for such a program can contain only moded literals, meaning that every selected unification literal must be moded. By Lemma 5.4, then, each N-ESL tree for a moded program is modally complete. □

5.3.3 Moded selection rules

Theorem 5.5 applies only to programs and queries that contain no unmoded terms. While this restriction is sufficient to guarantee that the conditions of Lemma 5.4 are satisfied, it is not necessary. For some other programs it is possible to construct an ESL tree in which every selected literal is moded. This is because the selection and solution of a literal yields moded substitutions that serve to instantiate the remaining literals in a query. (We saw examples of this in Chapter 2.) In this section we begin to show how, by modifying the selection rules, the completeness guarantee of Theorem 5.5 can be extended to a broader class of programs.

We will investigate the effects of imposing a class of selection rules called *moded selection rules*. A moded selection rule is any selection rule R that satisfies the following three properties:

- (modedness) $L \in R(Q) \Rightarrow \mathit{moded}(L)$
- (instantiation) $L \in R(Q) \wedge \mathit{moded}(\sigma) \Rightarrow \sigma L \in R(\sigma L)$
- (equality) $L \in R(\leftarrow \ldots, L, \ldots) \wedge L =_E L' \Rightarrow L' \in R(\leftarrow \ldots, L', \ldots)$

Under a moded selection rule, a literal cannot be selected unless it is moded.

We will refer to resolution procedures based upon moded selection rules as moded resolution procedures. Moded resolution makes it possible to find a modally complete solution to some unmoded programs. Unfortunately, moded resolution can produce a blocked tree. This can happen if, for example, all of the literals at some node are unmoded.

In Section 5.4 we will investigate three different moded resolution procedures. Although the selection rules they are based upon are progressively more restrictive, they share with all moded rules the properties established by the following three lemmas.

Lemma 5.6 is the basis for proving the completeness of moded resolution procedures. It follows immediately from Lemma 5.4 and the fact that only moded literals can satisfy R.

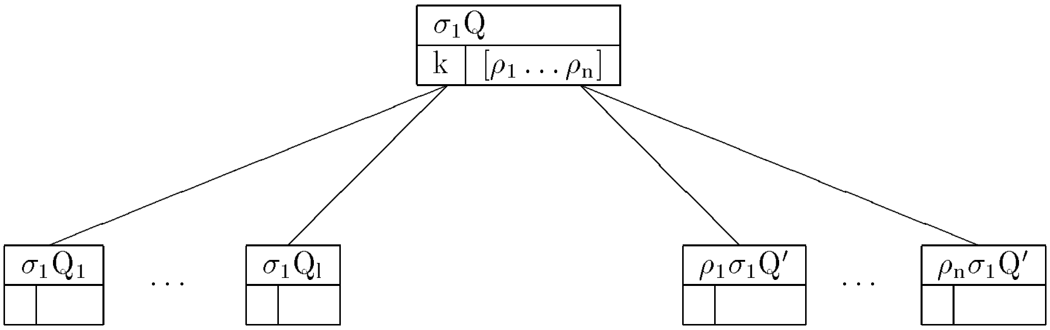
Lemma 5.6 *Let R be a moded selection rule. Any unblocked* R-ESL *tree is modally complete.*

Lemma 5.7 states that the existence of an unblocked tree for a program implies the existence of an unblocked tree for each of its moded instances.

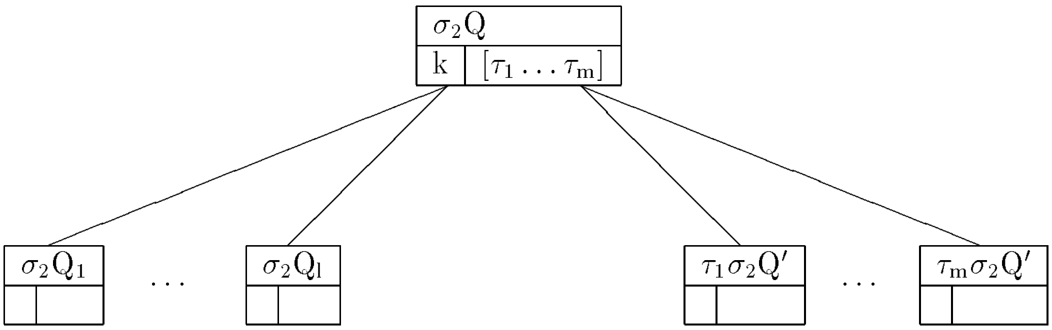
Lemma 5.7 *Let R be a moded selection rule, and let σ_1 and σ_2 be moded substitutions such that $\sigma_1 \leq_E \sigma_2$. If $\sigma_1 Q$ has an unblocked* R-ESL *tree, then $\sigma_2 Q$ has one as well.*

Proof. Proof is by induction. The induction hypothesis is that for any natural number d, if we are given an unblocked R-ESL tree for $\sigma_1 Q$ we can construct an R-ESL tree for $\sigma_2 Q$ that contains no blocked node within d nodes of the root. This is sufficient to establish the lemma because blocked nodes always terminate finite branches.

An unblocked R-ESL tree for $\sigma_1 Q$ will be of the form shown below.



Basis: The k^{th} literal of $\sigma_1 Q$ satisfies R. By the instantiation and equality properties of moded selection rules, the k^{th} literal of $\sigma_2 Q$ must also satisfy R. Hence, no R-ESL tree for $\sigma_2 Q$ can be blocked in the root node. This establishes the basis, and also means there is an R-ESL tree for $\sigma_2 Q$ of the form shown below.



Induction: We assume that the induction hypothesis holds to depth d, and show that it holds to depth $d+1$. The set $\{\rho_1, \dots, \rho_n\}$ is a complete solution for the k^{th} literal of $\sigma_1 Q$, and each τ_i is a solution for the k^{th} literal of $\sigma_2 Q$. This means that for every τ_i

there is a ρ_j such that $\rho_j \circ \sigma_1 \leq_E \tau_i \circ \sigma_2$. By the induction hypothesis, then, none of the children of the root node of the tree above contains a blocked node within d nodes of the root. Therefore, the entire tree contains no blocked node within $d+1$ nodes of the root.

□

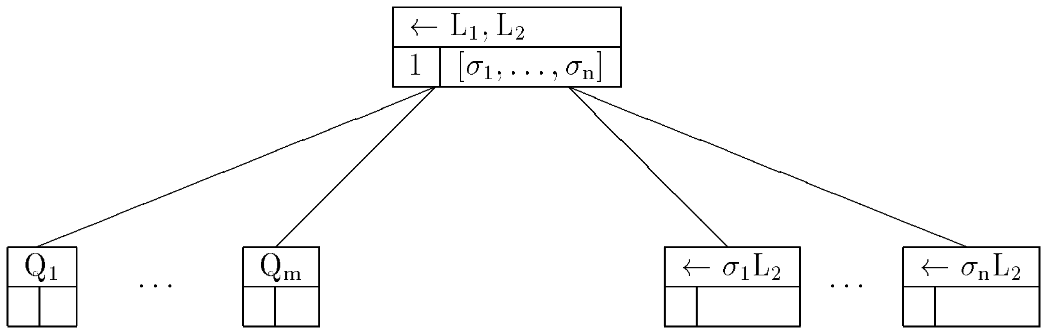
Lemma 5.8 states that if an unblocked moded tree exists for a literal, then there is no harm in selecting that literal for reduction at a node. This property will permit us to rearrange trees in subsequent proofs.

Lemma 5.8 *Let R be a moded selection rule, and let Q be a query that has an unblocked* R-ESL *tree. Let L be a literal of Q. Then $\leftarrow L$ has an unblocked* R-ESL *tree* $\iff$ *Q has an unblocked* R-ESL *tree in which L is selected in the root.*

Proof. Assume without loss of generality that Q is of the form $\leftarrow L_1, L_2$.

($\Leftarrow$) Immediate.

($\Rightarrow$) Suppose that $\leftarrow L_1$ has an unblocked R-ESL tree, and that Q has an unblocked R-ESL tree in which L_2 is selected in the root. This implies that $\leftarrow L_2$ has an unblocked R-ESL tree. Consider the following R-ESL tree for Q in which L_1 is selected in the root.



Because $\leftarrow L_1$ has an unblocked R-ESL tree, each of the left children has an unblocked R-ESL tree. Because $\leftarrow L_2$ has an unblocked R-ESL tree, each of the right children has an unblocked tree by Lemma 5.7. □

5.4 Moded resolution procedures

In this section we propose and evaluate a series of three moded selection rules. These rules give rise to three different ESL resolution procedures. The first two procedures contain instructive flaws, which we correct in the third procedure. This third procedure is the basis for the operational semantics of Denali.

We begin in Section 5.4.1 with the *nondeterministic moded selection rule* (M). This is the most permissive possible moded rule, as it always allows the selection of any moded literal. Because of its permissive nondeterminism, however, its behavior is unpredictable in one crucial respect. For a given program and query, it is possible for an M-ESL procedure to generate a blocked tree in one execution and an unblocked tree in another. This makes it impossible to characterize the cases for which M-ESL resolution is complete.

We correct this problem in Section 5.4.2 by introducing the *solvable selection rule* (S). If even a single unblocked M-ESL tree exists for a program and query, then every S-ESL tree for that combination is unblocked. S-ESL resolution is thus as robust as is possible for a moded procedure, and is in principle completely predictable with respect to completeness. Unfortunately, it is based upon the unrealistic assumption that perfect knowledge is available concerning which single-literal queries have unblocked M-ESL trees. Because of this assumption, S-ESL resolution is not always implementable.

In Section 5.4.3 we address this realizability problem by defining the *well-moded selection rule* (W). W-ESL resolution is neither as robust nor as predictable as S-ESL resolution. It is, however, always consistent in whether or not it generates a blocked tree for a given program and query. Because the well-moded selection rule is based upon an approximation to the perfect information assumed available to the solvable selection rule, W-ESL can sometimes block in cases in which S-ESL resolution does not. If the approximation is exact, however, W-ESL reduces to S-ESL.

5.4.1 Nondeterministic moded selection

The most general possible moded selection rule is the nondeterministic moded selection rule (M). It imposes no restrictions beyond those required of all moded selection rules; consequently, it selects all moded literals from a query.

By Lemma 5.6, any unblocked M-ESL tree is modally complete. M-ESL resolution is not modally complete, though, since it can produce blocked trees. Moreover, the approach of restricting the domain of M-ESL resolution by imposing static checking, as we did for N-ESL resolution, is not practical. In general, it is not possible to determine *a priori* whether the evaluation of a program under M-ESL resolution will lead to a blocked, and thus incomplete, tree. This fact becomes apparent only during evaluation.

An alternative is to adopt a dynamic approach. A language interpreter based upon M-ESL resolution could reject, as ill-formed, programs that lead to blocked trees. This strategy is analogous to run time type checking, in contrast to the static approach considered previously with N-ESL resolution.

The problem with this approach is that some programs possess both blocked and unblocked M-ESL trees. The kind of tree that is generated depends upon the selections made, within the constraints of the selection rule, at each node of the tree. As a result, an M-ESL interpreter as described above might reject in one execution a program that it was able to solve in another execution.

Despite its flaws, we will carry through with an analysis of M-ESL resolution in preparation for the introduction of the solvable selection rule in Section 5.4.2. This rule remedies the defects, without sacrificing the generality, of the moded selection rule.

Programs can be divided into three disjoint sets, according to whether blocked or unblocked M-ESL trees can be produced for them. The first set contains the programs that give rise only to unblocked trees. This set includes, but is not limited to, the set of programs containing only moded terms. M-ESL resolution is complete for these programs.

The second set contains the programs that give rise only to blocked trees. This set includes, but is not limited to, the set of programs in which each literal is unmoded. M-ESL resolution is not applicable to these programs.

The third set consists of the problematic programs we described above, those that can give rise to both blocked and unblocked trees. The kind of tree produced for a program in this set is determined by the selection that is made at each node.

For example, consider the program composed of the equational theory

$$\mathrm{X}+0 = \mathrm{X}$$
$$\mathrm{X}+\mathrm{Y} = \mathrm{Y}+\mathrm{X}$$
$$\mathrm{s(X)}+\mathrm{Y} = \mathrm{s(X+Y)}$$

and the definite clause

double(N, N+N).

We will examine the solution of the query

$\leftarrow$ double(s(0), X), double(X, Y)

with respect to the moded base that is composed of all *nat* terms containing no more than one variable. (We originally considered this moded base in Chapter 3.)

Both of the literals in the initial query are moded. Thus, either could be selected in the root node of an M-ESL tree for the query. The case in which the first literal is selected is shown in Figure 5.7. This tree is unblocked. The case in which the second literal is selected is shown in Figure 5.8. This tree is blocked because there is no moded literal in the bottommost node.

This example illustrates that whether or not M-ESL resolution is modally complete for a particular program is not always an intrinsic property of that program. Instead, it can be a function of what selections are made. In the next section we will revise M-ESL resolution so that it is complete for all programs for which at least one unblocked M-ESL tree exists.

5.4.2 Solvable selection

Our goal in this section is to devise a variant of ESL resolution that is complete for all queries that have at least one unblocked M-ESL tree. We do this by imposing the solvable selection rule (S). S-ESL resolution relates to M-ESL resolution as follows. If each of a query's M-ESL trees is blocked, then each of its S-ESL trees is blocked as well. If, on the other hand, a query has even one unblocked M-ESL tree, then all of its S-ESL trees are unblocked.

The definition of the new selection rule depends upon the notion of solution set. The *solution set* of an n-ary predicate symbol P with respect to a program $(E; H)$ is the set of all term tuples $(t_1, \ldots, t_n)$ such that the query $\leftarrow P(t_1, \ldots, t_n)$ has an unblocked M-ESL tree. Thus, a solution set embodies complete knowledge of which literals can be solved with M-ESL resolution.

For example, the solution set of the predicate symbol *double* from the previous section is the set of all pairs of natural numbers in which at least one element is ground. Thus,

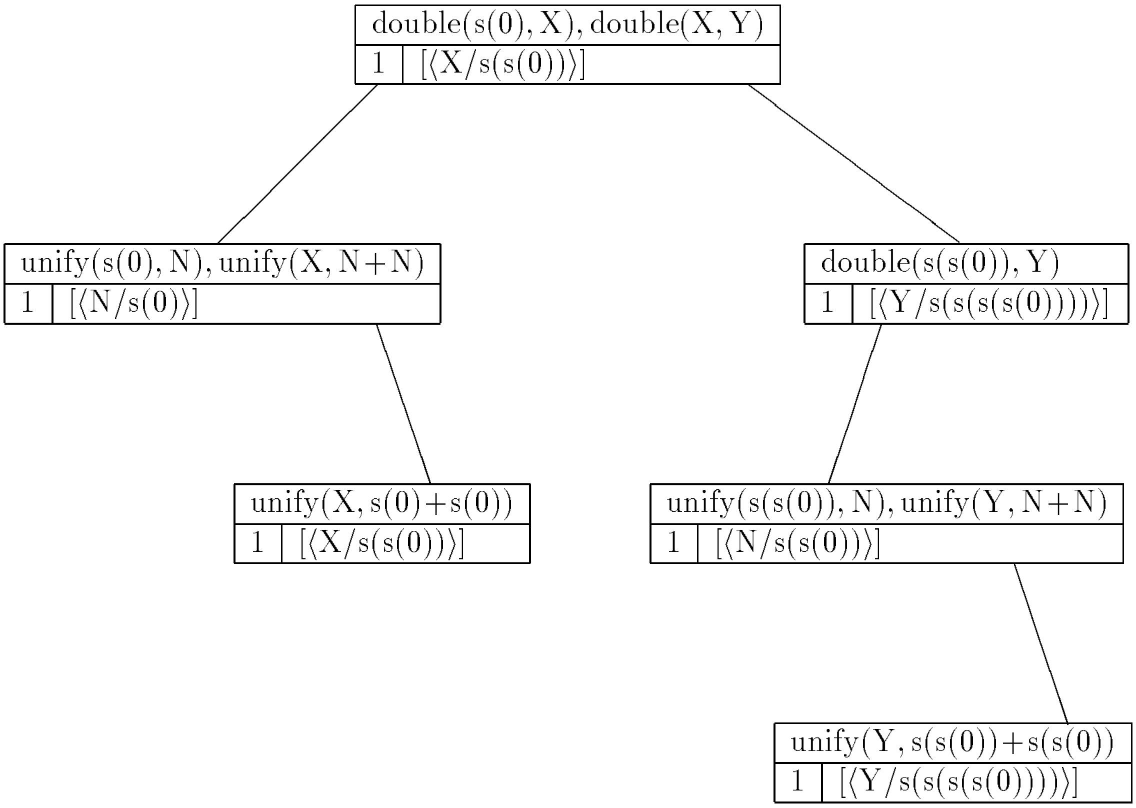


Figure 5.7: Unblocked M-ESL tree

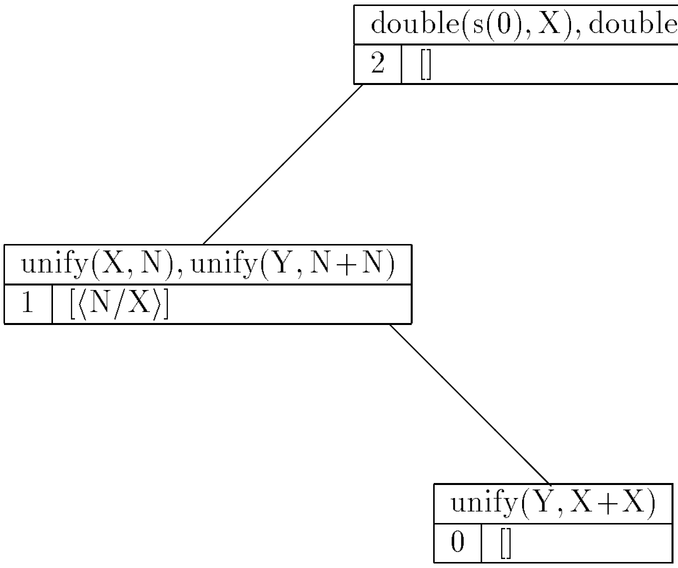


Figure 5.8: Blocked M-ESL tree

the solution set of *double* includes the pairs $(s(0), X)$ and $(s(Y), 0)$, but not $(s(Y), X)$.

Lemma 5.9 identifies a set of properties that solution sets possess.

Lemma 5.9 *Let SS be the solution set of a predicate symbol P whose signature is* $(S_1, \ldots, S_n)$. *Then*

- $(t_1, \ldots, t_n) \in M \Rightarrow \mathit{sort}(t_i) = S_i$,
- $\overline{t} \in M \Rightarrow \mathit{moded}(\overline{t})$, *and*
- $\sigma_1\overline{t} \in M \wedge \mathit{moded}(\sigma_1) \wedge \mathit{moded}(\sigma_2) \wedge \sigma_1 \leq_E \sigma_2 \Rightarrow \sigma_2\overline{t} \in M$.

Proof. We establish each of the three properties individually.

- By definition, the elements of *SS* share the same sort signature.
- Only moded literals can possess an unblocked M-ESL tree.
- Consequence of Lemma 5.7.

□

We are now prepared to define S-ESL resolution. A literal $\mathit{unify}(t_1, t_2)$ satisfies the solvable selection rule only if it is moded. This case is identical to the moded selection rule. A literal $P(t_1, \ldots, t_n)$ satisfies the solvable selection rule only if $(t_1, \ldots, t_n)$ is in the solution set of P. Since every tuple in a solution set must at least be moded, this case is a strict refinement of the moded selection rule. Consequently, every unblocked S-ESL tree is also an unblocked M-ESL tree.

Lemma 5.10 is the key to establishing the modal completeness of S-ESL resolution.

Lemma 5.10 *If a query has an unblocked* M-ESL *tree, then each of its* S-ESL *trees is unblocked.*

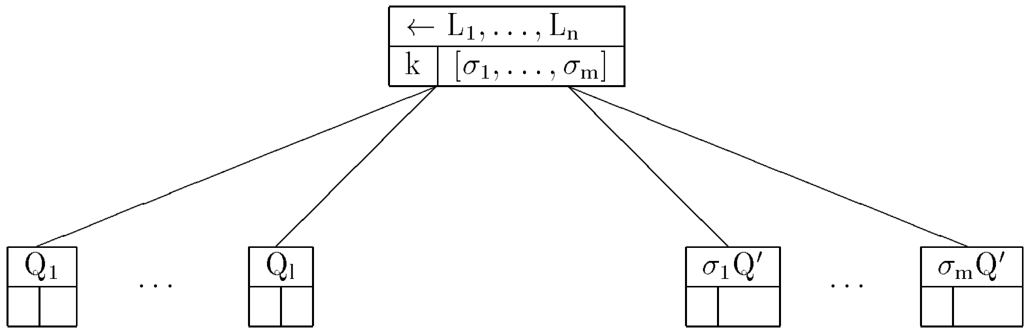
Rather than prove Lemma 5.10 directly, we prove the following more general lemma instead. Lemma 5.11 will prove useful in other contexts as well. It establishes a means of basing one selection rule upon another.

Lemma 5.11 *Let* R_1 *and* R_2 *be moded selection rules such that* $L \in R_1(Q)$ *holds only if* L *has an unblocked* R_2-ESL *tree. If a query has an unblocked* R_2-ESL *tree, then each of its* R_1-ESL *trees is unblocked.*

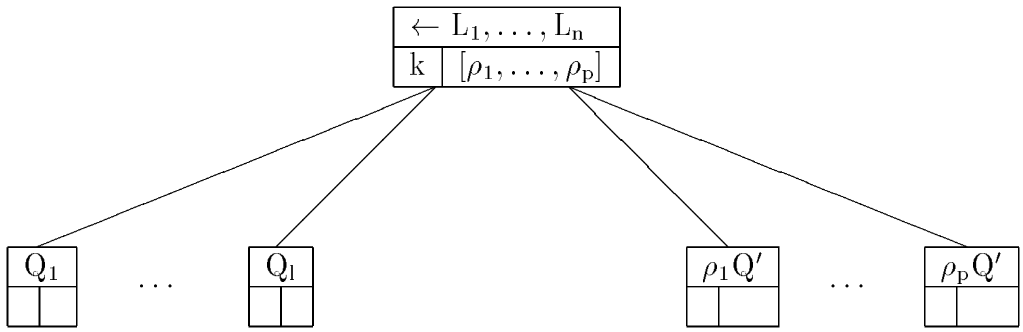
Proof. Proof is by induction. The induction hypothesis is that for any natural number d, if an unblocked R$_2$-ESL tree exists for some query Q, then no R$_1$-ESL tree for Q has a blocked node within d nodes of the root. This is sufficient to establish the lemma because blocked nodes always terminate finite branches.

Basis: Some literal L of Q is selected in the root of its unblocked R$_2$-ESL tree. Since L has an unblocked R$_2$-ESL tree, it follows that $R_1(L)$ holds. Hence, every R$_1$-ESL tree for Q must be unblocked at the root node.

Induction: We assume that the induction hypothesis holds to depth d, and show that it holds to depth $d+1$. Let k be the index of some literal of Q that satisfies the selection rule R_1. The following tree is a representative R$_1$-ESL tree for Q.



Since both Q and $\leftarrow L_k$ have unblocked R$_2$-ESL trees, if follows from Lemma 5.8 that Q has an unblocked R$_2$-ESL tree in which literal k is selected in the root node.



We can complete the proof by showing that every R$_1$-ESL tree for each child node in the first tree is unblocked to depth d. This is immediate for the left children by the induction hypothesis, since the left children are the same in the two trees. Because the substitutions $\{\ldots, \rho_i, \ldots\}$ are a complete solution to L_k, it follows that for each σ_i, there is a ρ_j such that $\rho_j \leq_E \sigma_i$. The induction hypothesis can then be applied along with

Lemma 5.7. □

Theorem 5.12 now follows immediately.

Theorem 5.12 S-ESL *resolution is modally complete for all queries that have at least one unblocked* M-ESL *tree.*

5.4.3 Well-moded selection

We have shown that S-ESL resolution is modally complete in the following sense. If a query has an ESL tree in which every selected literal and every substitution is moded, then S-ESL resolution is complete for that query. This means that S-ESL resolution is at least as broadly applicable as any other procedure based on a moded selection rule.

Unfortunately, S-ESL resolution is not a practical procedure. It is applicable only if it is possible to implement a test for membership in the solution set of each predicate symbol. Testing a literal for membership, however, would require enumerating all of the M-ESL trees for that literal.

We address this problem in this section, and in the process transform S-ESL resolution into a realizable procedure. Our approach is to devise a resolution procedure that is identical to S-ESL resolution over a restricted domain of queries. This new procedure does not require complete knowledge of the solution set of each predicate symbol. It instead uses modings to describe a subset of each solution set. This addresses the problem identified above because we have a way of defining and checking for membership in modings. The drawback of this approach is that fewer queries can be completely solved than with S-ESL resolution. The advantage is that the approach is easily implementable.

Recall that a moding is the set of all term tuples that are well-moded with respect to some set of mode tuples that share a common signature. Modings possess the closure properties of solution sets that we identified in Lemma 5.9. This is a consequence of the definition of modes given in Section 5.3.1.

We will work from this point with moded programs. A *moded program* is a triple $(E; H; M)$, where E is an equational theory, H is a set of definite clauses, and M is a moding function. A *moding function* maps predicate symbols to modings. This means two things. First, it maps every predicate symbol from **P** to a moding that matches that symbol's signature. Second, it maps every sort S to a moding of signature (S, S); this corresponds to giving a set of modings for *unify*.

$M(P)$ is intended to be an approximation to the solution set of P for which an efficient decision procedure exists. Our approximation of the solvable selection rule is based upon it. We say that a literal $P(t_1, \ldots, t_n)$ is *well-moded* with respect to a program $(E; H; M)$ if $(t_1, \ldots, t_n) \in M(P)$. The well-moded selection rule (W) is satisfied by any well-moded literal. We will refer to resolution procedures based upon this rule as W-ESL procedures.

We will only consider consistently moded programs. A moded program $(E; H; M)$ is *consistently moded* if, for every well-moded literal of the form $P(t_1, \ldots, t_n)$, the query $\leftarrow P(t_1, \ldots, t_n)$ possesses an unblocked W-ESL tree.

One example of a consistently moded program is one in which the moding function maps each predicate symbol to its solution set. In this case, W-ESL and S-ESL resolution coincide for all queries. Whenever the moding function maps some predicate symbol to a proper subset of its solution set, W-ESL resolution is complete over a strictly smaller domain.

We can now characterize the behavior of W-ESL resolution with respect to consistently moded programs. We can show that the W-ESL trees for a consistently moded program are either all blocked or all unblocked.

Lemma 5.13 *Let the underlying program be consistently moded. Then either all of the* W-ESL *trees for a query Q are blocked, or none are.*

Proof. In the statement of Lemma 5.11, let both R_1 and R_2 be W. The lemma follows immediately. □

W-ESL resolution is the form of resolution used in Denali. One desirable property is that programs satisfying the same specification not exhibit different behaviors in solving the same query. Lemma 5.14, which deals with operationally equivalent programs, establishes the basis of this property.

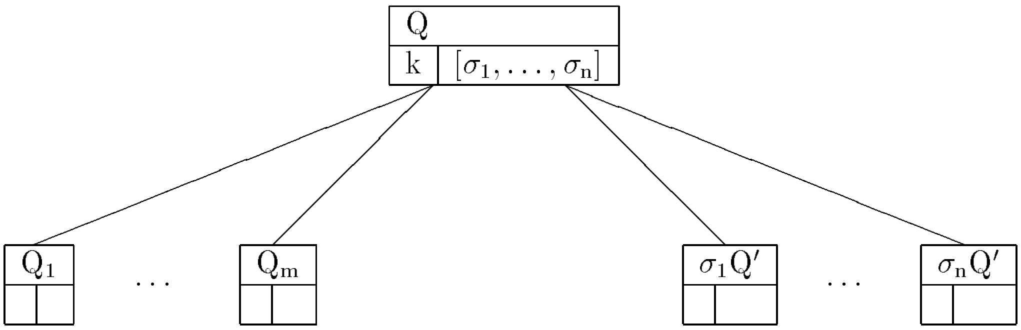
Let $(E; H_1; M_1)$ and $(E; H_2; M_2)$ be consistently moded programs. The two are *operationally equivalent* with respect to a predicate symbol P if the following two conditions hold. First, $M_1(P) = M_2(P)$. Second, for all well-moded literals $P(t_1, \ldots, t_n)$, the meanings of H_1 and H_2 are the same:

$$S_E^{H_1}(P(t_1, \ldots, t_n)) = S_E^{H_2}(P(t_1, \ldots, t_n)).$$

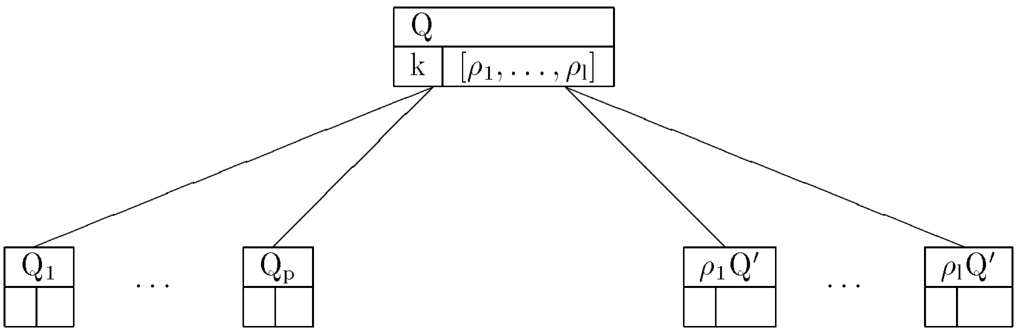
Lemma 5.14 *Let P_1 and P_2 be consistently moded programs that are operationally equivalent for each of the predicate symbols in some query Q. Then Q has an unblocked* W-ESL *tree with respect to P_1 if and only if Q has an unblocked* W-ESL *tree with respect to P_2.*

Proof. Proof is by induction. The induction hypothesis is that for any natural number d, the existence of an unblocked W-ESL tree for Q with respect to P_1 implies the existence of a W-ESL tree for Q with respect to P_2 that contains no blocked node within d nodes of the root.

Basis. An unblocked W-ESL tree for Q with respect to P_1 will be of the form:



Let L_k be the k^{th} literal of Q. If L_k is well-moded with respect to P_1, it is also well-moded with respect to P_2. This is because of the operational equivalence assumption. Hence, there must be a W-ESL tree for Q with respect to H_2 of the following form:



Induction: There exist unblocked left children for the tree above because L_k is well-moded. By operational equivalence, the sets $\{\sigma_1, \ldots, \sigma_n\}$ and $\{\rho_1, \ldots, \rho_l\}$ are subsets of the same set, and the former set is complete. Thus, for every ρ_j there is a σ_i such that $\sigma_i \leq_E \rho_j$. Thus, by Lemma 5.7 and the induction hypothesis, each of the right children of the tree above is unblocked to depth d. $\square$

The Operational Completeness Theorem follows directly from Lemmas 5.13 and 5.14.

Theorem 5.15 *Let P_1 and P_2 be consistently moded programs that are operationally equivalent for each of the predicate symbols in some query Q. The* W-ESL *trees for Q, with respect to both programs, are either all blocked or all unblocked.*

5.5 Summary

Our goal in this chapter has been to develop a pragmatic equational resolution procedure to serve as the basis of the semantics of Denali. We proposed four selection rules that gave rise to four different resolution procedures. In this section we summarize the properties of each procedure and highlight the relationships among them.

We began with the non-deterministic selection rule. It is the most general possible selection rule as it permits the selection of an arbitrary literal from any query. It gives rise to N-ESL resolution, which is complete (Theorem 5.3). Implementing N-ESL resolution, unfortunately, requires implementing a complete equational unification procedure for the theory contained in the program. Because complete equational unification procedures are inefficient, complicated, and sometimes impossible to obtain, N-ESL resolution is not a practical procedure.

We addressed the problem of obtaining equational unification procedures by exploiting moded bases, and in the process shifted our attention to moded equational resolution. We proposed the non-deterministic moded selection rule, which permits the selection of any of the moded literals of a query. It leads to M-ESL resolution, whose implementation requires only that a modally complete equational unification procedure for the theory contained in the program be obtained. Although it is practical, M-ESL resolution is not complete. Worse, there are programs for which M-ESL resolution sometimes returns complete solutions and sometimes does not, depending upon which selections are made in the course of constructing a tree (Figures 5.7 and 5.8.) This makes it unsuitable for use in defining Denali.

We then considered the solvable selection rule. It permits the selection of any literal that possesses an unblocked, and therefore complete, M-ESL tree. This rule gives rise to S-ESL resolution. S-ESL resolution is not complete—no resolution procedure based upon a moded selection rule can be—but its behavior is consistent. It is complete for all programs and queries that possess at least one unblocked M-ESL tree (Theorem 5.12). Because it is based upon semantic information, however, the solvable selection rule is undecidable. Determining whether a literal satisfies this selection rule requires enumerating its M-ESL trees.

We reached a pragmatic compromise with the well-moded selection rule. This rule is based upon the use of modings to describe the circumstances under which a literal may be selected. It gives rise to W-ESL resolution. Like M-ESL resolution, the selection rule upon which W-ESL resolution is based can be efficiently decided. Like S-ESL resolution, its behavior with respect to completeness is consistent (Lemma 5.13) so long as the moding restrictions are imposed consistently. These two properties make it suitable for use in defining Denali.

The Operational Equivalence Theorem (Theorem 5.15) is the most important result of the chapter. It states that W-ESL resolution does not differentiate between operationally equivalent programs. This provides the basis for using moded equational definite clause programs as specifications for Denali programs.

6 Moded equational unification

There are three general approaches to obtaining a unification procedure for a given equational theory. In this chapter we will consider these approaches, show how they can be extended to cope with and thus take advantage of moded equational theories, and comment upon their role in the Denali interpreter.

The most broadly applicable approach is to develop directly a procedure for the theory in question. This direct approach has been followed by a number of researchers.

The second method involves mechanically synthesizing a unification procedure from a presentation of the theory. Only a restricted class of theories are amenable to this synthesis approach, however, and inefficient non-terminating procedures generally result.

The third technique requires decomposing the theory into subtheories for which unification procedures are known. The subtheory procedures can then be combined to yield an overall unification procedure. Several algorithms for the combining problem have been developed; all impose restrictions upon the decomposition step.

In Section 6.1 we describe these three approaches in more detail. Direct implementation and procedure combination each have a role in implementing Denali. Consequently, we will also describe how these two approaches can be adapted to obtaining moded unification procedures.

A set of procedures based upon the combining algorithm described in Section 6.1 lies at the heart of the proposed Denali interpreter. The algorithm must be generalized, however, before it can be exploited for this purpose. In Section 6.2 we identify the limitations of the combining algorithm and then extend it appropriately.

6.1 Approaches to equational unification

In this section we describe the three approaches to obtaining equational unification algorithms that we outlined above.

6.1.1 Direct implementation

A number of algorithms have been developed for unification in the empty theory. The earliest was described in [Robinson 65] and runs in time that is exponential in the size of the input terms. Other algorithms have subsequently been developed that reduce worst-case running time to polynominal [Corbin 83], almost linear [Martelli 82, Baxter 73], and linear [Paterson 78].

Equational unification procedures, and in most cases terminating algorithms, are currently known for a number of equational theories. Some examples include procedures for the associative [Plotkin 72], commutative [Siekmann 79], associative-commutative [Stickel 81, Livesey 76], identity [Arnborg 85], and one-sided distributive [Arnborg 85] theories.

The unification problem in some theories, e.g. the associative-distributive theory, has been proven undecidable [Szabó 78]. In others, such as the associative theory, no terminating procedure exists [Makanin 77]. Terminating procedures, i.e. algorithms, exist for the remainder of the theories mentioned above. Typically, the execution time for existing algorithms is high. The worst-case complexity of a number of unification problems have been classified. Unification in the commutative theory is NP-complete [Garey 79] and in the associative-commutative theory is NP-hard [Benanav 85]. Identity unification is also NP-hard [Arnborg 85], while one-sided distributive unification is polynomial [Arnborg 85].

We have seen that the advantage of moded unification arises from the possibility, through the establishment of a moded base and the imposition moding constraints, of restricting the domain of a unification procedure to a more easily managed subset. The severity of the restrictions that can be imposed is limited only by the expressive power requirements of the application in which a given moded unification procedure is embedded.

We illustrated with the examples of Chapters 3 and 4 the utility of mode restrictions for facilitating the construction of unification predicates in Denali. We expect that the moded unification predicates constructed by Denali programmers will tend to be heavily restricted so that they might be more easily implemented. Mode restrictions might also be exploited in constructing more efficient moded versions of existing unification procedures as built-in predicates.

All existing unification procedures are automatically modally complete if the moded base is the set of all terms. For smaller moded bases it might be necessary to devise new procedures. In any event, the body of existing unification procedures provides a rich source for implementations of the built-in theories of Denali.

6.1.2 Narrowing

An approach based upon the narrowing operation of [Slagle 74] can be used to synthesize unification procedures for equational theories presented by a convergent term rewriting system [Fay 79]. The narrowing approach has been improved by [Hullot 80], who gives sufficient conditions for its termination; by [Jouannaud 83], who generalizes it to equational term rewriting systems; and by [Réty 85], who improves its efficiency.

Despite these improvements, the narrowing approach remains primarily of theoretical interest. Except for a small class of equational presentations, the procedures produced by narrowing are nonterminating. Furthermore, the procedures are in general too inefficient to be practical as part of a larger system.

The narrowing approach produces a modally complete unification procedure whenever the equational presentation upon which narrowing operates is composed entirely of moded equations. No more general condition guaranteeing modal completeness is known.

6.1.3 Combining algorithms

Three researchers [Yelick 85, Kirchner 85, Tidèn 86] have independently developed algorithms that address the problem of combining complete unification algorithms for disjoint theories to produce a complete unification algorithm for their union. Below, we characterize these approaches, which are designed to cope with unsorted theories and terminating algorithms. We do this in preparation for extending their approach, in Section 6.2, to account for sort restrictions, mode restrictions, and nonterminating procedures.

Suppose that we wish to obtain a unification algorithm for an equational theory E^*. Suppose further that E can be decomposed into a set of n equational presentations $E_1, \ldots, E_n$ such that $E = \bigcup E_i$, and that there exists a unification algorithm U_i corresponding to each E_i. It is possible to interconnect the U_i to obtain a complete unification algorithm for E if the E_i and the U_i meet the following four conditions.

First, it must be possible to partition the set of function symbols $\mathbf{F}$ into n pairwise disjoint sets $\mathbf{F}_i$ such that each of the function symbols in presentation E_i lies in $\mathbf{F}_i$. In other words, let $\mathbf{T}_i$ be the set of terms formable from the function symbols $\mathbf{F}_i$ and the variable symbols $\mathbf{V}$. Every term in E_i must lie in $\mathbf{T}_i$.

Second, each algorithm U_i must be complete with respect to the equational theory E_i^*. It need be defined only for terms contained in $\mathbf{T}_i$.

Third, each presentation E_i must contain only collapse-free equations. A collapse-free equation is one in which either both sides are variables or both sides are non-variables.

Fourth, each presentation E_i must contain only regular equations. A regular equation is one in which the set of variables appearing in the left-hand side is equal to the set of variables appearing in the right-hand side. Kirchner's version of this restriction is slightly stronger. Tidèn improves upon Yelick's algorithm by removing the regularity restriction altogether, but his improvement is inefficient. Although none of the extensions that we will describe precludes using Tidèn's approach to removing the regularity restriction, we will maintain it for reasons of efficiency and clarity.

If the presentations E_i and the algorithms U_i meet these four restrictions, then the combining algorithms of Yelick and Tidèn can link the U_i together to produce a complete unification algorithm for the theory presented by $\bigcup E_i$.

For example, suppose that we wish to obtain a unification algorithm for the theory presented by the following three equations E.

X+Y = Y+X
X+(Y+Z) = (X+Y)+Z
f(f(X)) = f(X)

The decomposition approach is applicable because each of these equations is both collapse-free and regular.

One three-way decomposition of E and F is as follows:

	E_i	$\mathbf{F}_i$	U_i
1	$x+y = y+x$ $x+(y+z) = (x+y)+z$	$+$	associative-commutative
2	$f(f(x)) = f(x)$	f	idempotent
3		$\mathbf{F} - \{+, f\}$	empty

If three unification algorithms can be obtained, then the combining algorithm can produce an overall unification algorithm for arbitrary terms with respect to E.

The combining algorithm is central to the implementation of Denali. We will modify it in Section 6.2 so that it can be used to combine the individual unification predicates provided for each sort in Denali into an overall unification algorithm.

In preparation for this step, we present below the basic combining algorithm, which is based upon that of Yelick. It has been rearranged somewhat to facilitate modification. It is composed of three procedures that we will discuss in turn. Our intention is to describe only enough of the algorithm to facilitate the pending changes. The reader should consult [Yelick 85] for a more detailed description and proof of correctness.

Effects: Returns a complete set of unifiers for t_1 and t_2.

CRunify = **proc** ($\mathrm{t_1}$: term, $\mathrm{t_2}$: term) **returns** (substSet)
  case
    $\mathrm{isVar}(t_1) \wedge \mathrm{isVar}(t_2) \Rightarrow$ **return**$(\{\langle t_1/t_2\rangle\})$
    $\mathrm{isVar}(t_1) \wedge t_1 \notin \mathcal{V}(t_2) \Rightarrow$ **return**$(\{\langle t_1/t_2\rangle\})$
    $\mathrm{isVar}(t_2) \wedge t_2 \notin \mathcal{V}(t_1) \Rightarrow$ **return**$(\{\langle t_2/t_1\rangle\})$
    $\neg\mathrm{isVar}(t_1) \wedge \neg\mathrm{isVar}(t_2) \wedge t_1.\mathrm{head} \in \mathbf{F}_i \wedge t_2.\mathrm{head} \in \mathbf{F}_j \wedge i \neq j \Rightarrow$
      return$(\{\})$
    else $\Rightarrow$ **return**$(\mathrm{doUnify}(t_1, t_2))$
    end
  end CRunify

CRunify treats the cases in which unification can be performed independent of knowledge about the equational theory, passing along the more complicated cases to *doUnify*. The first three cases check for situations involving variables. The fourth case checks whether the head symbols of non-variable arguments appear in different function symbol partitions. If so, the pair is assumed to be ununifiable because of the absence of collapse equations from the equational presentations.

Requires: Head symbols of t_1 and t_2 lie in same partition.
Effects: Returns a complete set of unifiers for t_1 and t_2.

doUnify = **proc** ($\mathrm{t_1}$: term, $\mathrm{t_2}$: term) **returns** (substSet)
  let $\theta_1(\hat{t}_1) = t_1$
  let $\theta_2(\hat{t}_2) = t_2$
  let $\theta = \theta_1 \cup \theta_2$
  case
    $\exists v$ **s.t.** $t_1 \leq \theta_2 v \Rightarrow$ **return**$(\{\})$
    $\exists v$ **s.t.** $t_2 \leq \theta_1 v \Rightarrow$ **return**$(\{\})$
    else $\Rightarrow$
      let $R = U_i(\hat{t}_1, \hat{t}_2)$
      let $\Sigma = \bigcup_{\rho \in R} \mathrm{mapUnify}(\rho,\ \theta,\ \mathcal{D}(\rho) \cup \mathcal{D}(\theta))$
      return(Σ)
    end
  end doUnify

The first phase of *doUnify* is to homogenize the input terms and find their preserving substitutions. A term is homogenized by replacing, with fresh variables, all subterms headed by function symbols that do not lie in the same partition as the term's head symbol. The homogenized version of a term t is denoted by $\hat{t}$. For example, suppose t is the term $X+(Y+f(Z))$. Using the partitions identified earlier, $\hat{t}$ is the term $X+(Y+W)$, where W is any fresh variable.

The substitution that maps a homogenized term to its original form is called its preserving substitution. The preserving substitution of t above is $\langle W/f(Z)\rangle$. In *doUnify*, the preserving substitution of t_1 is θ_1, and the preserving substitution of t_2 is θ_2. The union of these two preserving substitutions is θ.

If one argument term appears as a subterm ($\leq$) in the range of the other's preserving substitution, then the two terms are not unifiable. This is a consequence of the absence of non-regular equations from the underlying equational theory.

If neither subterm condition applies, the original terms are unified in two steps. First, the homogenized forms of the terms are unified by using the appropriate U_i unification algorithm. Next, each such unifier is unified with θ using *mapUnify*. The resulting set of substitutions is a complete set of unifiers of the input terms.

> Effects: Returns a complete set of unifiers of σ_1 and σ_2 with respect to the variables in *VS*.
>
> mapUnify = **proc** (σ_1: subst, σ_2: subst, VS: varSet) **returns** (substSet)
>   **case**
> $|\mathrm{VS}| = 0 \Rightarrow$ **return**($\{\langle\rangle\}$)
>   else $\Rightarrow$
>   **let** $\mathrm{v} \in \mathrm{VS}$
>   **let** $\mathrm{R} = \mathrm{CRunify}(\sigma_1\mathrm{v}, \sigma_2\mathrm{v})$
>   **let** $\mathrm{T}_\rho = \mathrm{mapUnify}(\rho \circ \sigma_1,\ \rho \circ \sigma_2,\ \mathrm{VS} - \mathrm{v}),\ \forall \rho \in \mathrm{R}$
>   **let** $\Theta = \bigcup_{\rho \in \mathrm{R}} \bigcup_{\tau \in \mathrm{T}_\rho} \tau \circ \rho$
>   **return**(Θ)
>   **end**
>   **end** mapUnify

MapUnify unifies pairs of substitutions by unifying, in turn, the images of each variable in their domains.

6.2 Combining moded unification procedures

A variant of the combining algorithm *CRunify* outlined in the previous section will form the heart of the Denali interpreter that we describe in Chapter 7. *CRunify* has four limitations, however, that we must remove to make this possible. In this section we describe these limitations and devise a modified version of *CRunify* that eliminates them.

In Sections 6.2.1–6.2.4 we describe how *CRunify* can be modified to cope with sorted theories, non-terminating procedures, external procedures, and moded theories. In Section 6.2.5 we accumulate these modifications and present a version of *CRunify* that is appropriate for Denali.

6.2.1 Sorted algorithms

Although the combining algorithm was originally defined only for unsorted theories, it extends immediately to the sorted case. If the input algorithms U_i are complete with respect to their respective sorted subtheories, the combined algorithm will be complete with respect to the sorted union of these subtheories. This holds because the combining algorithm produces new substitutions only by composing existing ones. This cannot introduce sort inconsistencies.

Suppose that the input procedures U_i, the equational presentations E_i upon which they are based, and the function symbol sets $\mathbf{F}_i$ are partitioned along sort boundaries. This means that each set $\mathbf{F}_i$ contains exactly the function symbols of some range sort S, and there is one partition per sort. If so, we can exploit the presence of sort information to relax two of the restrictions upon the combining algorithm.

From this point we will assume that partitions do in fact lie along sort boundaries. We will emphasize this assumption by subscripting the input procedures, equational presentations, and function symbol sets with sort names rather than with numbers. For every sort S, then, the set $\mathbf{F}_S$ contains all of the function symbols whose range is S. The meanings of U_S and E_S follow directly. Under this assumption on partitions we can eliminate the condition that the sets E_S be collapse-free and weaken the condition that they be regular.

In the examples below, assume that we have two sorts, *nat* and *list*, and thus two function symbol partitions, $\mathbf{F}_{nat}$ and $\mathbf{F}_{list}$:

0: $\rightarrow$ nat	nil: $\rightarrow$ list
1: $\rightarrow$ nat	cons: nat, list $\rightarrow$ list
$*$: nat, nat $\rightarrow$ nat	
inverse: nat $\rightarrow$ nat	

Collapse equations are a problem in the general case because they permit pairs of terms whose head symbols lie in distinct function symbol partitions to be equated. Since *CRunify* relies on the assumption that such pairs are not unifiable, collapse equations must be forbidden.

To illustrate, suppose that the collapse equation $X + 0 = X$ is in the equational presentation. One consequence of this presentation in the unsorted case is the equation $nil + 0 = nil$. Even though the two terms $nil + 0$ and nil are equal, *CRunify* will treat them as ununifiable because their head symbols ($+$ and nil respectively) lie in separate partitions. Consequently, collapse equations such as $X + 0 = X$ must be forbidden.

This construction is not possible in the sorted case if, as we assume, the partitions lie along sort boundaries. This is because any equation relating terms with different head symbols is necessarily ill-sorted. In the sorted case, the equation $nil{+}0 = nil$ is not well-sorted. It is proper for *CRunify* to treat the two terms in this equation as ununifiable, and collapse equations need not be forbidden.

Non-regular equations are a problem in the general case because they permit the occurrence of a term t_1 in the range of the preserving substitution of an equal term t_2. Since *doUnify* relies upon the assumption that pairs such as t_1 and t_2 are not unifiable, non-regular equations must be forbidden.

To illustrate, suppose that the non-regular equation $X * inverse(X) = 1$ is in the equational presentation. One consequence of this presentation in the unsorted case is the equation

$$\mathrm{cons}(1, \mathrm{nil}) * \mathrm{inverse}(\mathrm{cons}(1, \mathrm{nil})) = 1.$$

The homogenized form of the left-hand side of this equation is $W * W = 1$, and the preserving substitution is $\langle W / cons(1, nil) \rangle$. The preserving substitution contains the right-hand side of the equation above as subterm. Because of this, *CRunify* will treat the two terms in the equation as ununifiable. Consequently, collapse equations must be forbidden.

This example would not cause a problem in the sorted case, because the equation that leads to the problem is not well-sorted. Notice, however, that if we were to add a *nat* constructor size: list $\rightarrow$ nat

to $\mathbf{F}_{nat}$, then we would be able to infer the equation

$$\text{size}(\text{cons}(1, \text{nil})) * \text{inverse}(\text{size}(\text{cons}(1, \text{nil}))) = 1.$$

This equation is well-sorted, and the preserving substitution of its left-hand side contains the right-hand side as a subterm. It thus exhibits the characteristic problem of non-regular equations even in the sorted case.

A non-regular equation may or may not cause a problem, depending upon the signatures of the various function symbols. We will call an equation *sort-regular* if it cannot lead to an embedding problem of the form illustrated above. It is easy to construct a syntactic test for sort-regularity; it involves searching for sort cycles in function symbol signatures.

The consequence of all of this is that non-regular equations need not be forbidden altogether. *CRunify* can accommodate non-regular equations so long a they are sort-regular.

6.2.2 Non-terminating procedures

CRunify requires that the subsidiary unification procedures U_i be terminating algorithms. Before *CRunify* can be incorporated into Denali, we must extend it to cope with nonterminating procedures that enumerate potentially infinite complete sets of unifiers.

The restriction of the combining algorithm to terminating procedures is necessitated by the interface of *CRunify* rather than by any intrinsic characteristic of the problem. Both Yelick and Tidèn point out that the extension needed to accommodate nonterminating procedures is straightforward.

The interface of *CRunify* returns complete sets of substitutions. This is the same interface that is assumed of the subsidiary procedures U_i. In Section 6.2.5, we will cast *CRunify* and the subsidiary procedures as iterators that yield one substitution at a time. So long as it is implemented in a fair fashion, *CRunify* can be made to to yield a complete sequence of unifiers even when the subsidiary iterators are nonterminating.

6.2.3 Independent procedures

Our third extension involves the incorporation of self-contained unification procedures into the combining process. We have assumed to this point that each constituent unification procedure U_S is

- defined over terms that are homogeneous in $\mathbf{F}_S$, and
- is complete with respect to E_S.

In some instances, we might have available a more powerful unification procedure U_S that is

- defined over all terms of sort S, and
- is complete with respect to E.

We would like to be able to deal with such procedures in the combining process.

The need for this additional flexibility is motivated by the design of Denali. Recall that each sort is realized by either an implicit or an explicit implementation. In an implicit implementation, the unification predicate is provided by the implementation and is of the first form identified above. In an explicit implementation, the unification predicate is supplied by the programmer, and is of the second form identified above. Both kinds of procedures must be combined to obtain an overall unification procedure for a program.

If every unification procedure U_S were of the second form above, the combining process would involve only determining the sort of the terms to be unified and invoking the appropriate procedure. It should not be surprising, then, that such self-contained procedures can be incorporated into *CRunify* without disruption. Self-contained procedures can be called directly as subroutines whenever terms of sort S must be unified.

In the remainder of this section, we will assume that $\mathbf{S}$ is divided into two disjoint subsets containing the *implicit* and the *explicit* sorts. We will make different assumptions, detailed in Section 6.2.5, concerning the subsidiary unification procedures U_S in each case.

6.2.4 Moded procedures

To this point we have dealt with the problem of combining complete subsidiary procedures to produce a complete overall procedure. Our final extension is to treat the problem of combining modally complete subsidiary procedures to produce a modally complete overall procedure. We must treat two separable concerns here. First, we must deal with completeness relative to the underlying moded base of terms. Second, we must deal with any restrictions in the form of modings that are imposed upon the interfaces of the subsidiary procedures.

Treating the first concern is straightforward, because *CRunify* is transparent to moded bases. If the constituent unification procedures are modally complete, then the combined procedure is itself modally complete. As was the case with sorted completeness, this is because new substitutions are derived only through the composition of old ones. It is thus not possible for *CRunify* to generate any unmoded substitutions.

Dealing with interface restrictions is not as simple. Assume that associated with each sort S is a moding, denoted $moding(S)$. This moding, which in a Denali program would be imposed by the programmer, imposes an interface restriction upon the procedure U_S. Ideally, we would like to be able to guarantee that given any two terms t_1 and t_2 of sort S,

- if $(t_1, t_2) \notin moding(S)$, then $CRunify(t_1, t_2)$ reports a mode failure, and
- if $(t_1, t_2) \in moding(S)$, then $CRunify(t_1, t_2)$ is modally complete.

The first condition is easy to guarantee by checking whether arguments are properly moded and reporting a mode failure if they are not. The second condition cannot be guaranteed because an execution of *CRunify* can entail recursive calls to subsidiary procedures whose modings may not be compatible. Incompatibility is a possibility because the modings are imposed by programmers. Consequently, we modify *CRunify* to report inadvertent mode failures as they occur.

6.2.5 An extended combining procedure

We are now prepared to describe the combining procedure, which we have extended along the lines discussed in the preceding four sections. The most important difference is that the interfaces of the combining procedures are now cast as iterators. These iterators yield substitutions one at a time, and signal mode failures when they occur.

Before presenting the modified iterators, we will restate the revised restrictions that we have accumulated. Suppose that we wish to obtain a unification iterator for a sorted and moded equational theory E^*, and that the set of sorts $\mathbf{S}$ is divided into the implicit and the explicit sorts. For each sort S, let U_S be a unification iterator and let E_S be the set of all equations from E of sort S. It is possible to interconnect the U_S to obtain a moded unification iterator for E if the U_S and the E_S meet the following three conditions.

First, for each implicit sort S, U_S must be defined over the set of terms that are homogeneous in $\mathbf{F}_S$, and must be modally complete with respect to E_S.

Second, for each explicit sort S, U_S must be defined over all terms of sort S, and must be modally complete with respect to E.

Third, for each implicit sort S, E_S must contain only sort-regular equations.

If these conditions are satisfied, then the combining iterator *CRunify* can link the iterators U_S together to produce a modally complete unification procedure for the theory presented by E.

> Effects: Yields a complete sequence of unifiers of t_1 and t_2. Signals modeFailure if the arguments are ill-moded or if a mode failure occurs during execution.
>
> CRunify = **iter** ($\mathrm{t_1}$: term, $\mathrm{t_2}$: term) **yields** (subst) **signals** (modeFailure)
>   **let** S = sort($\mathrm{t_1}$)
>   **case**
> $(\mathrm{t_1}, \mathrm{t_2}) \notin \mathrm{moding}(\mathrm{U_S}) \Rightarrow$ **signal** modeFailure
> $\mathrm{isVar}(\mathrm{t_1}) \wedge \mathrm{isVar}(\mathrm{t_2}) \Rightarrow$ **yield**($\langle \mathrm{t_1}/\mathrm{t_2} \rangle$)
> $\mathrm{isVar}(\mathrm{t_1}) \wedge \mathrm{t_1} \notin \mathcal{V}(\mathrm{t_2}) \Rightarrow$ **yield**($\langle \mathrm{t_1}/\mathrm{t_2} \rangle$)
> $\mathrm{isVar}(\mathrm{t_2}) \wedge \mathrm{t_2} \notin \mathcal{V}(\mathrm{t_1}) \Rightarrow$ **yield**($\langle \mathrm{t_2}/\mathrm{t_1} \rangle$)
>   explicit(S) $\Rightarrow$ **reyield**($\mathrm{U_S}(\mathrm{t_1}, \mathrm{t_2})$)
>   implicit(S) $\Rightarrow$ **reyield**(doUnify($\mathrm{t_1}, \mathrm{t_2}$))
>   **end resignal** modeFailure
>   **end** unify

A check has been added to *CRunify* to ensure that the terms being unified meet the modings restrictions for their sort. The check that head symbols lie in the same partition is no longer necessary and has been removed; as we pointed out in Section 6.2.1, this is a consequence of sort consistency. The strategy used to unify a particular pair of terms now depends upon whether their sort is explicit or implicit. In the former case, the appropriate iterator is invoked directly. In the latter case, *doUnify* is invoked as before.

Requires: Head symbols of t_1 and t_2 lie in same partition.
Effects: Yields a complete sequence of unifiers of t_1 and t_2. Signals modeFailure if one occurs during execution.

```
doUnify = iter (t1: term, t2: term) yields (subst) signals (modeFailure)
    let S = sort(t1)
    let θ1(t̂1) = t1
    let θ2(t̂2) = t2
    let θ = θ1 ∪ θ2
    case
        ∃v s.t. t1 ≤ θ2v ⇒ return
        ∃v s.t. t2 ≤ θ1v ⇒ return
        else ⇒
            for ρ ∈ S − unify(t̂1, t̂2) do
                for σ ∈ mapUnify(ρ, θ, 𝒟(ρ ∪ 𝒟(θ)) do
                    yield(σ)
                    end
                end
        end resignal modeFailure
    end doUnify
```

The only important difference in the new version of *doUnify* is that the unification procedures are unrolled into loops to ensure fairness.

Effects: Yields a complete sequence of unifiers of σ_1 and σ_2 with respect to the variables in VS.

```
mapUnify = iter (σ1: subst, σ2: subst, VS: varSet) yields (subst)
                                     signals (modeFailure)
    case
        |VS| = 0 ⇒ yield(⟨⟩)
        else ⇒
            let v ∈ VS s.t. U_S(σ1(v), σ2(v)) is well moded
                    signal modeFailure if none exist
            for ρ1 ∈ CRunify(σ1(v), σ2(v)) do
                for ρ2 ∈ mapUnify(ρ1 ∘ σ1, ρ1 ∘ σ2, VS − v) do
                    yield(ρ2 ∘ ρ1)
                    end
                end
        end resignal modeFailure
    end mapUnify
```

MapUnify is also unrolled from its previous form. Notice also that at each step is uses mode information to choose which variable to treat next. If none is appropriate, it signals *modeFailure*. This ordering makes it more robust with respect to mode failures.

7 Semantics of Denali

In this chapter we draw upon the material of Chapters 5 and 6 to define the semantics of Denali.

Denali programs can appear in either denotation or representation form. All of the examples that we have seen thus far have been in denotation form, because this is the form in which programs must be written. We describe denotation form programs in Section 7.1 by giving their syntax and static semantics.

Representation form is a partially compiled version that is more convenient for describing the semantics. In Section 7.2 we define this form by showing how programs can be converted from denotation into representation form.

A key part of the translation between the two forms is a procedure that translates the denotation of an abstract object into its representation. This procedure is based upon the translation predicates that are included with each explicitly implemented cluster. We develop this translation procedure in Section 7.3.

In the final two sections we give the abstract and operational semantics of Denali programs. The abstract definition, given in Section 7.4, involves regarding a program as a moded equational definite clause program. It draws upon the material on moded resolution developed in Chapter 5. The operational definition, given in Section 7.5, is

length = **pred** (bag, nat) **moding** (enum, any)
 length(nil, 0).
 length(cons(N, B), s(X)) $\leftarrow$ length(B, X).

reduce = **pred** (bag, bag) **moding** (gnd, any)
 reduce(nil, nil).
 reduce(cons(X, cons(X, B_1), B_2) $\leftarrow$ reduce(cons(X, B_1), B_2).

sequential = **pred** (set, nat) **moding** (enum, any)
 sequential(insert(N, insert(s(N),)), s(N)).

Figure 7.1: Predicate implementations

based upon an interpreter whose implementation we sketch. It draws upon the *CRunify* procedure that we extended in Chapter 6.

The formulation of the semantics of Denali contained in this chapter does not take into account the mode guard construct of Chapter 2. Incorporating mode guards into the framework of our semantic definition remains an open problem.

7.1 Denotation form

In this section we give the abstract syntax and static semantics of Denali programs that are written in denotation form. Chapters 2 and 4 contain an informal development of the concrete syntax of Denali. We have duplicated several of the examples from Chapter 4 in Figures 7.1–7.3. We will refer to the implementations in these figures throughout this chapter.

We will develop the syntactic description incrementally, with each set of productions paired with the static semantic restrictions that apply to them. Along the way we will define a number of functions over the syntactic domain. Their names, e.g. SORTS, are written with small capitals. These functions are used to express static semantic constraints, and are also used later when defining the meaning of programs. Syntactic productions are highlighted by hollow bullets ($\circ$), and static semantic restrictions are highlighted by solid bullets ($\bullet$).

We begin by giving the top-level productions that characterize the module structure

```
nat = cluster
    denoted by
        0: → nat
        s: nat → nat
    modes any > gnd
    unify mode any
    moded by
        0: → gnd
        s: gnd → gnd
        s: any → any
    end

bag = cluster
    denoted by
        nil: → bag
        cons: nat, bag → bag
    modes any > enum > gnd
    unify mode enum
    moded by
        nil: → gnd
        cons: gnd, gnd → gnd
        cons: any, enum → enum
        cons: any, any → any
    unified by
        cons(X, cons(Y, B)) = cons(Y, cons(X, B))
    end
```

Figure 7.2: Implicitly implemented clusters

---

of programs. Non-terminals, e.g. ⟨program⟩, are delimited by angle brackets. Within a production, a vertical bar denotes an alternative and an asterisk denotes a sequence of zero or more elements.

- ⟨program⟩ ::= ⟨module⟩*
- ⟨module⟩ ::= ⟨cluster⟩ | ⟨predicate⟩
- ⟨cluster⟩ ::= ⟨implicitCluster⟩ | ⟨explicitCluster⟩
- ⟨implicitCluster⟩ ::= ⟨clusterHead⟩ ⟨implicitImpl⟩
- ⟨explicitCluster⟩ ::= ⟨clusterHead⟩ ⟨explicitImpl⟩ ⟨predicate⟩*
- ⟨predicate⟩ ::= ⟨predHead⟩ ⟨predImpl⟩

The examples in Figures 7.1–7.3 constitute a ⟨program⟩ composed of six ⟨modules⟩. Figure 7.1 contains three ⟨predicates⟩, Figure 7.2 contains two ⟨implicitClusters⟩, and Figure 7.3 contains one ⟨explicitCluster⟩.

```
set = cluster
    denoted by
        empty: → set
        insert: nat, set → set
    modes any > enum > gnd
    unify mode any
    represented by bag
    translated by setTrans
    moded by
        gnd from gnd
        enum from enum
        any from enum
    unified by setUnify
    setTrans = pred (set_dnt, bag) moding (any, any)
        setTrans(empty, nil).
        setTrans(insert(N, B), cons(N, B)).
    setUnify = pred (bag, bag) moding (enum, enum)
        setUnify(nil, nil).
        setUnify(cons(X, B1), cons(X, B2)) ← setUnify(B1, B2).
        setUnify(cons(X, B1), cons(X, B2)) ← setUnify(cons(X, B1), B2).
        setUnify(cons(X, B1), cons(X, B2)) ← setUnify(B1, cons(X, B2)).
    size = pred (set, nat) moding (gnd, any)
        size(B1, N) ← reduce(B1, B2), length(B2, N).
    end
```

Figure 7.3: Explicitly implemented cluster

The presentation of the remainder of the syntax is divided into four sections. Section 7.1.1 concerns the elaboration of ⟨clusterHead⟩ and ⟨predHead⟩, which present the interface information for clusters and predicates respectively. These module headers establish the name spaces with respect to which the static semantic constraints are defined.

In Section 7.1.2 we show how ⟨implicitImpl⟩, ⟨explicitImpl⟩, and ⟨predImpl⟩ are elaborated. These are the implementations of implicit clusters, explicit clusters, and predicates.

Corresponding to every explicitly implemented sort S is a reserved sort symbol S_dnt. These are the denotation sorts, which are used in the translation predicates where terms of sort S must be treated as uninterpreted syntactic objects. In Section 7.1.3 we discuss the role of these denotation sorts. Although they are not defined by the programmer, their implicit implementations can be derived mechanically from the program in which they appear.

In Section 7.1.4 we complete the syntactic description by giving the sort-checking rules.

7.1.1 Interfaces

The headers of clusters and predicates define the name spaces for sorts, predicates, functions, and modes. They also associate sort and mode signatures with predicates, and sort signatures with function names.

Cluster headers

The top-level production for ⟨clusterHead⟩ is

- ∘ ⟨clusterHead⟩ ::= ⟨sort⟩ ⟨funcDecl⟩* ⟨modeOrdering⟩* ⟨unifyMode⟩

The following restriction applies.

- Each ⟨clusterHead⟩ must introduce a globally unique ⟨sort⟩. Let SORTS be the set of all ⟨sorts⟩ introduced in a ⟨program⟩.

We say that a sort is *explicitly implemented* if it is introduced in a ⟨explicitCluster⟩, and that it is *implicitly implemented* otherwise. In addition to the sort symbols introduced in cluster headers, a collection of denotation sorts are introduced implicitly. For every explicitly implemented sort S, the set SORTS also contains the implicitly implemented sort S_dnt. In the example,

$$\textsc{sorts} = \{\mathrm{nat}, \mathrm{bag}, \mathrm{set}, \mathrm{set\_dnt}\}.$$

The implicitly implemented sorts are *nat*, *bag*, and *set_dnt*, while *set* is explicitly implemented.

We will consider the three remaining components of a ⟨clusterHead⟩ individually. First, the ⟨funcDecl⟩ section introduces a set of function signatures. These signatures are used to construct abstract objects of the cluster's sort.

- ∘ ⟨funcDecl⟩ ::= ⟨func⟩ ⟨sortDomain⟩ ⟨sortRange⟩
- ∘ ⟨sortDomain⟩ ::= ⟨sort⟩*
- ∘ ⟨sortRange⟩ ::= ⟨sort⟩

The following restrictions apply.

- Each ⟨func⟩ must be globally unique. Let FUNCS(S) be the set of all function symbols introduced in a cluster S.
- Each ⟨sortRange⟩ in a cluster S must be S.
- Each ⟨sort⟩ of a ⟨sortDomain⟩ must be in SORTS. Let SIG(f, S) denote the signature associated with function name f defined in cluster S.

In the example,

$$\begin{aligned}&\textsc{funcs}(\text{set}) = \{\text{empty}, \text{insert}\},\\ &\textsc{sig}(\text{empty}, \text{set}) = (\rightarrow \text{set}),\\ &\textsc{sig}(\text{insert}, \text{set}) = (\text{nat}, \text{set} \rightarrow \text{set}).\end{aligned}$$

The uniqueness requirement for function symbols is imposed to simplify our syntactic development. In practice, overloading can be permitted since sort information can be used to disambiguate overloaded symbols.

Second, the ⟨modeOrdering⟩ section of a ⟨clusterHead⟩ introduces a partially ordered set of mode names. We have included the ordering symbol, technically part of the concrete syntax, for clarity.

- ⟨modeOrdering⟩ ::= ⟨mode⟩ > ⟨mode⟩

Mode names must be unique within, but may be overloaded among, clusters.

- The mode name *any* must appear in each ⟨modeOrdering⟩ section. Let MODES(S) denote the set of mode names introduced in the ⟨modeOrdering⟩ section of a cluster S.
- There must be a partial ordering compatible with the ordering restrictions of a ⟨modeOrdering⟩ section. Let $\geq$ be the minimal such partial order.
- The partial order $\geq$ must have *any* as a greatest element.

In the example MODES(bag) $= \{any, enum, gnd\}$, and the modes are ordered as shown.

Third, the ⟨unifier⟩ section of a ⟨clusterHead⟩ gives the mode restriction for the unification predicate.

- ⟨unifier⟩ ::= ⟨unifyMode⟩
- The ⟨unifyMode⟩ of a cluster S must be a member of MODES(S). Denote this mode by UMODE(S).

In the example UMODE(set) $= \{any\}$.

Predicate headers

The top-level production for ⟨predHead⟩ is

- ○ ⟨predHead⟩ ::= ⟨pred⟩ ⟨sortSig⟩ ⟨modeSig⟩*

The following restriction applies

- Each ⟨predHead⟩ must introduce a globally unique ⟨pred⟩. Let PREDS denote the set of all ⟨preds⟩ introduced in a ⟨program⟩.

In the example,

$$\text{PREDS} = \{\text{length}, \text{reduce}, \text{sequential}, \text{setUnify}, \text{setTrans}, \text{size}\}.$$

Both sort and mode information are present in predicate headers.

- ○ ⟨sortSig⟩ ::= ⟨sort⟩*
- ○ ⟨modeSig⟩ ::= ⟨mode⟩*
- Each ⟨sort⟩ in a ⟨sortSig⟩ must be a member of SORTS. Let ABSTSIG(P) denote the ⟨sortSig⟩ of a predicate P.
- Each ⟨modeSig⟩ must have the same arity as its associated ⟨sortSig⟩. Let the set of ⟨modeSigs⟩ of a predicate P be denoted by MODINGS(P)
- Let $(S_1, \ldots, S_n)$ be a ⟨sortSig⟩ and $(M_1, \ldots, M_n)$ a ⟨modeSig⟩ of the same header. Then each M_i must be a member of MODES(S_i).

In the example,

$$\text{ABSTSIG}(\text{size}) = (\text{set}, \text{nat}).$$
$$\text{ABSTSIG}(\text{reduce}) = (\text{bag}, \text{bag}).$$

Inside of an explicitly implemented cluster, we sometimes require that the representation of an abstract object appear where the abstract object would be otherwise required. We make a provision for this possibility by defining the concrete signatures of predicates. Suppose that ABSTSIG(P) = $(S_1, \ldots, S_n)$, where P is a predicate defined within an explicit cluster S. Then CONCSIG(P) = $(R_1, \ldots, R_n)$, where $R_i = S_i$ unless $S_i = S$ in which case R_i = REP(S). (REP(S), which is defined below, is the representation sort of S.) If P is not defined within an explicit cluster, then ABSTSIG(P) and CONCSIG(P) are identical. In the example,

$$\text{CONCSIG}(\text{size}) = (\text{bag}, \text{nat}).$$
$$\text{CONCSIG}(\text{reduce}) = (\text{bag}, \text{bag}).$$

7.1.2 Implementations

Implicit cluster implementations

In an implicit implementation, only the modes and the equational theory for unification need be given.

- ○ $\langle$implicitImpl$\rangle$ ::= $\langle$dirModeImpl$\rangle$ $\langle$dirUnifImpl$\rangle$

First, the $\langle$dirModeImpl$\rangle$ section gives the meaning of the modes introduced in the $\langle$modeOrdering$\rangle$ section.

- ○ $\langle$dirModeImpl$\rangle$ ::= $\langle$modeDecl$\rangle^*$
- ○ $\langle$modeDecl$\rangle$ ::= $\langle$func$\rangle$ $\langle$modeDomain$\rangle$ $\langle$modeRange$\rangle$
- ○ $\langle$modeDomain$\rangle$ ::= $\langle$mode$\rangle^*$
- ○ $\langle$modeRange$\rangle$ ::= $\langle$mode$\rangle$
- • Let f: $M_1, \ldots, M_n \rightarrow M$ be a $\langle$modeDecl$\rangle$ of a cluster S, where $\textsc{sig}(f, S)$ is $(S_1, \ldots, S_n) \rightarrow S$. Then M must be in $\textsc{modes}(S)$ and each M_i must be in $\textsc{modes}(S_i)$.

Second, the $\langle$dirUnifImpl$\rangle$ section gives a set of equations that specify the equational theory of unification for the cluster.

- ○ $\langle$dirUnifImpl$\rangle$ ::= $\langle$equation$\rangle^*$
- ○ $\langle$equation$\rangle^*$ ::= $\langle$term$\rangle$ = $\langle$term$\rangle$

A sort restriction applies to each equation. The function TERMSORTED, which checks that terms are well-sorted, is defined later.

- • For each equation $r = t$ appearing in a cluster S, there must be a mapping V from the variables of r and t to SORTS such that $\textsc{termsorted}(r, S, V)$ and $\textsc{termsorted}(t, S, V)$ are both true. One consequence of this is that both r and t are of the same sort.
- • Only function symbols with range sort S can appear in an $\langle$equation$\rangle$ within a cluster S.
- • Every $\langle$equation$\rangle$ must be sort-regular as defined in Chapter 6.

Explicit cluster implementations

Explicit implementations of clusters are more involved than implicit implementations.

- ○ $\langle$explicitImpl$\rangle$ ::= $\langle$repSort$\rangle$ $\langle$expModeImpl$\rangle$ $\langle$expDntImpl$\rangle^*$ $\langle$expUnifImpl$\rangle$

We discuss the four non-terminals in the production above in turn.

First, the $\langle$repSort$\rangle$ of an $\langle$explicitCluster$\rangle$ is the sort upon which the implementation is based.

- ○ $\langle$repSort$\rangle$ ::= $\langle$sort$\rangle$
- • The $\langle$repSort$\rangle$ of a cluster S must be in SORTS. Denote this sort by $\textsc{rep}(S)$.

In the example, $\textsc{rep}(set) = bag$.

Second, the ⟨expModeImpl⟩ section gives the implementation of the modes introduced in the ⟨modeOrdering⟩ section.

- ∘ ⟨expModeImpl⟩ ::= ⟨modeMapping⟩*
- ∘ ⟨modeMapping⟩ ::= ⟨absMode⟩ ⟨repMode⟩*
- ∘ ⟨absMode⟩ ::= ⟨mode⟩
- ∘ ⟨repMode⟩ ::= ⟨mode⟩
- Each ⟨absMode⟩ appearing in a cluster S must be in MODES(S).
- Each ⟨repMode⟩ appearing in a cluster S must be in MODES(REP(S)).

Third, the ⟨expDntImpl⟩ section names a function that accomplishes the translation of denotation to representation.

- ∘ ⟨expDntImpl⟩ ::= ⟨func⟩ ⟨pred⟩
- The ⟨expDntImpl⟩ of a cluster S, denoted DPRED(S), must be in PREDS.
- The ABSTSIG of DPRED(S) must be (S_dnt, R), where $R =$ REP(S).

In the example, DPRED(set) = $setTrans$.

Fourth, the ⟨expUnifImpl⟩ section identifies a predicate that performs unification.

- ∘ ⟨expUnifImpl⟩ ::= ⟨pred⟩
- The ⟨expUnifPred⟩ of a cluster S, denoted UPRED(S), must be in PREDS.
- The ABSTSIG of UPRED(S) must be (R, R), where $R =$ REP(S).

In the example, UPRED(set) = $setUnify$.

Predicate implementations

We now consider the ⟨predImpl⟩ section.

- ∘ ⟨predImpl⟩ ::= ⟨clause⟩*
- ∘ ⟨clause⟩ ::= ⟨head⟩ ⟨query⟩
- ∘ ⟨head⟩ ::= ⟨literal⟩
- ∘ ⟨query⟩ ::= ⟨literal⟩*

The functions HEADSORTED and TAILSORTED check literals for sort correctness. They are defined later.

- The head symbol of each ⟨head⟩ appearing in a ⟨predImpl⟩ P must be P.
- For each ⟨clause⟩ $L \leftarrow L_1, \ldots, L_n$ there must be a mapping V from the variables of the clause to SORTS such that HEADSORTED(L, V) holds for the head literal and TAILSORTED(L_i, V) holds for each other literal.

set_dnt = **cluster**
  denoted by
    empty: $\rightarrow$ set_dnt
    insert: nat, bag $\rightarrow$ set_dnt
  modes any
  unify mode any
  moded by
    empty: $\rightarrow$ any
    insert: any, any $\rightarrow$ any
  end

Figure 7.4: Derived implementation of *set_dnt*

7.1.3 Denotation sorts

Although denotation sorts are used in programs (see *setTrans* in Figure 7.3), they are nowhere defined. This is because the implicit implementation of a denotation sort *S_dnt* can be derived mechanically from the interface of the explicitly implemented sort S. We show below how the denotation, modes, and unification sections of these implementations are derived. We will use *set_dnt*, whose implicit implementation appears in Figure 7.4, as an example. This implementation is derived from that of *set* in Figure 7.3.

Objects of a denotation sort *S_dnt* are denoted using constructors with the same name as, but with different signatures than, the constructors of the sort S. This is the only instance in which function name overloading is permitted. For example, the two constructors of sort *set* are

empty: $\rightarrow$ set.
insert: nat, set $\rightarrow$ set.

The two constructors of sort *set_dnt* are

empty: $\rightarrow$ set_dnt.
insert: nat, bag $\rightarrow$ set_dnt.

The transformed signatures are obtained by replacing each occurrence of the abstract sort (*set*) in the original domain with the representation sort (*bag*), and by replacing each occurrence of the abstract sort in the original range with the denotation sort (*set_dnt*). Thus, objects of a denotation sort are a kind of hybrid between abstract denotations and concrete representations.

Every denotation sort *S_dnt* has but one mode, which by default must be *any*. Consequently, the mode signatures are simple. There is one signature for each constructor; each signature contains only the mode *any*.

Since a sort *S_dnt* has only mode *any*, its unification mode must be *any*. Because denotation terms are always treated syntactically, the theory of unification of each sort *S_dnt* is empty. Consequently, no equations need be given in the implementation.

In Section 7.1.1 we defined several functions over the syntactic domain. When these functions are evaluated for a given program, they should defined over the implicitly implemented denotation sorts. We noted earlier that SORTS should contain *set_dnt* in addition to *nat*, *bag*, and *set*. Similarly, FUNCS and SIG should be augmented so that

$$\text{FUNCS}(\text{set\_dnt}) = \{\text{empty}, \text{insert}\},$$
$$\text{SIG}(\text{empty}, \text{set\_dnt}) = (\rightarrow \text{any}),$$
$$\text{SIG}(\text{insert}, \text{set\_dnt}) = (\text{nat}, \text{bag} \rightarrow \text{set\_dnt}).$$

In addition, MODES(*set_dnt*) should be $\{any\}$, and UMODE(*set_dnt*) should be *any*.

7.1.4 Sort checking

We can now define the sort-checking rules, which, with one exception, are straightforward. The three sort-checking functions are HEADSORTED and TAILSORTED for literals, and TERMSORTED for terms. Each of the three takes an argument that is a mapping from variables to sorts. This argument is necessary because variable symbols are unsorted in Denali. The only requirement is that they be used consistently within individual clauses and literals.

We begin with the syntactic productions for ⟨terms⟩ and ⟨literals⟩.

- ⟨literal⟩ ::= ⟨pred⟩ ⟨args⟩
- ⟨term⟩ ::= ⟨func⟩ ⟨args⟩ | ⟨var⟩
- ⟨args⟩ ::= ⟨term⟩*

Each of the literal functions takes two arguments: the literal to be checked and the variable mapping. HEADSORTED applies to the head literals of clauses. It checks the literal with respect to its concrete signature in case the literal lies within an explicitly defined cluster. (If not, the concrete and abstract signatures are identical.)

- HEADSORTED$(P(t_1,\ldots,t_n),V) \iff$
  CONCSIG$(P)=(S_1,\ldots,S_n) \wedge \forall i$ TERMSORTED(t_i,S_i,V).

TAILSORTED applies to all other literals. It checks the literal with respect to its abstract signature.

- TAILSORTED$(P(t_1,\ldots,t_n),V) \iff$
  ABSTSIG$(P)=(S_1,\ldots,S_n) \wedge \forall i$ TERMSORTED(t_i,S_i,V).

TERMSORTED takes three arguments: the literal to be checked, the sort required of the term, and the variable mapping.

- TERMSORTED$(v,S,V) \iff V(v)=S$.
- TERMSORTED$(f(t_1,\ldots,t_n),S,V) \iff$
  SIG$(f,S)=(S_1,\ldots,S_n \rightarrow S) \wedge$ TERMSORTED(t_i,S_i,V).

7.2 Representation form

Programs in representation form consist of a set of definite clauses, a set of equations with associated unification information, and a set of mode name presentations. Because they are close to equational definite clause programs, we base the definition of the semantics of Denali upon programs in this form.

In this section we describe how programs written in denotation form are converted to representation form. The overall conversion process is straightforward. The single complication is the translation of literals, which involves using the translation predicates that are included with every explicitly implemented cluster. To avoid cluttering this section we will assume the existence of a function, $T_{literal}$, for translating literals and defer its definition to Section 7.3.

We treat each step of the conversion separately. We begin in Section 7.2.1 by specifying the sets of symbols from which programs in representation form are constructed. In Section 7.2.2 we discuss clauses, in Section 7.2.3 we discuss unification, and in Section 7.2.4 we discuss modes.

7.2.1 Symbols

The first step in the conversion to representation form is fixing the sets of symbols to be used. These sets are the sort names **S**, the function symbols **F**, and the predicate

$\mathbf{S} = \{\mathrm{nat}, \mathrm{bag}, \mathrm{set}, \mathrm{set\_dnt}\}$
$\mathbf{F} = \{0, \mathrm{s}, \mathrm{nil}, \mathrm{cons}, \mathrm{empty}, \mathrm{insert}, \mathrm{A_{set}}\}$
$\mathbf{P} = \{\mathrm{length}, \mathrm{reduce}, \mathrm{setUnify}, \mathrm{setTrans}, \mathrm{size}, \overline{\textsc{size}}, \mathrm{sequential}\}$

0: $\rightarrow$ nat
s: nat $\rightarrow$ nat
nil: $\rightarrow$ bag
cons: nat, bag $\rightarrow$ bag
empty: $\rightarrow$ set_dnt
insert: nat, bag $\rightarrow$ set_dnt
$\mathrm{A_{set}}$: bag $\rightarrow$ set

length = **pred** (bag, nat) **moding** (enum, any)
reduce = **pred** (bag, bag) **moding** (gnd, any)
setUnify = **pred** (bag, bag) **moding** (enum, enum)
setTrans = **pred** (set_dnt, bag) **moding** (any, any)
size = **pred** (set, nat) **moding** (gnd, any)
$\overline{\textsc{size}}$ = **pred** (bag, nat) **moding** (any, any)
sequential = **pred** (set, nat) **moding** (enum, any)

Figure 7.5: Symbols of converted program

symbols**P**. We must also associate a sort signature with each function symbol, and a sort and mode signature with each predicate symbol. The symbols and signatures of the translated example program are given in Figure 7.5.

The sort universe **S** is exactly the set of sorts introduced by the program, i.e. SORTS. In our example, then, **S** is $\{nat, bag, set, set\_dnt\}$.

The definition of the function symbol universe **F** is almost as straightforward. For every implicitly implemented sort S, **F** contains FUNCS(S). For every explicitly implemented sort S, however, **F** contains only the single sort symbol A_S. This symbol is a syntactic version of the abstraction function. We introduce it in order to have an explicit constructor that is independent of the denotation symbols. In our example,

$$\mathbf{F} = \{0, \mathrm{s}, \mathrm{nil}, \mathrm{cons}, \mathrm{empty}, \mathrm{insert}, \mathrm{A_{set}}\}.$$

Note that *empty* and *insert* remain, but only as constructors of sort *set_dnt*.

We must also specify the sort signature associated with each function symbol in **F**. For a constructor f of an implicit sort, this signature is SIG(f). For each symbol A_S, the signature is (REP(S) $\rightarrow S$). Some representative signatures from our example are

0: $\rightarrow$ nat
$\mathrm{A_{set}}$: bag $\rightarrow$ set
insert: nat, bag $\rightarrow$ set_dnt

The predicate symbol universe **P** is a superset of PREDS. In addition to the symbols in PREDS, **P** also contains a symbol $\overline{P}$ corresponding to each predicate symbol P from PREDS whose abstract and concrete signatures differ. These additional predicate symbols are required to account for the dual interpretation that such predicates receive. In our example, **P** is $\{length, reduce, setUnify, setTrans, size, \overline{size}, sequential\}$.

The sort signature associated with each predicate symbol P is ABSTSIG(P), and the mode signature is carried over unaltered. The mode signature associated with each predicate symbol $\overline{P}$ is CONCSIG(P), and the mode signature is the tuple containing only *any*. In our example, the sort and mode signatures of *size* and $\overline{size}$ are as follows:

size = **pred** (set, nat) **moding** (enum, any)
$\overline{\text{size}}$ = **pred** (bag, nat) **moding** (any, any)

7.2.2 Clause conversion

The second step of conversion is to define the set of definite clauses. This set is derived from the clauses used to implement the predicates in the source program. The clauses cannot be carried over unaltered for two reasons. First, the terms that they contain must be translated from denotation to representation form. Second, we must account for the dual interpretations afforded to predicates that are defined inside of explicitly implemented clusters. The definite clauses of our converted example program appear in Figure 7.6.

We assume the existence of a translation function $T_{literal}$ that maps the denotations of objects to their representations. The implementation of this function exploits the translation predicates that are included in explicitly implemented clusters. We will discuss $T_{literal}$ in detail in Section 7.3.

We first consider predicate definitions whose abstract and concrete signatures are identical. Each such definition consists of a header and one or more definite clauses of the form

$\mathrm{length(nil, 0).}$
$\mathrm{length(cons(N, B), s(X)) \leftarrow length(B, X).}$
$\mathrm{reduce(nil, nil).}$
$\mathrm{reduce(cons(X, cons(X, B_1), B_2) \leftarrow reduce(cons(X, B_1), B_2).}$
$\mathrm{sequential(A_{set}(cons(N, cons(s(N), nil))), s(N)).}$
$\mathrm{setTrans(empty, nil).}$
$\mathrm{setTrans(insert(N, B), cons(N, B)).}$
$\mathrm{setUnify(nil, nil).}$
$\mathrm{setUnify(cons(X, B_1), cons(X, B_2)) \leftarrow setUnify(B_1, B_2).}$
$\mathrm{setUnify(cons(X, B_1), cons(X, B_2)) \leftarrow setUnify(cons(X, B_1), B_2).}$
$\mathrm{setUnify(cons(X, B_1), cons(X, B_2)) \leftarrow setUnify(B_1, cons(X, B_2)).}$
$\mathrm{size(A_{set}(B), N) \leftarrow \overline{size}(B, N)}$
$\mathrm{\overline{size}(B_1, N) \leftarrow reduce(B_1, B_2), length(B_2, N).}$

Figure 7.6: Clauses of converted program

$$\mathrm{L \leftarrow L_1, \ldots, L_n.}$$

For each such clause, the version that is added to the converted program is

$$\mathrm{T_{literal}(L) \leftarrow T_{literal}(L_1), \ldots, T_{literal}(L_n).}$$

Thus, nothing more is done than translate the literals. For example, converting the predicate definition

$$\mathrm{sequential} = \mathbf{pred}\ \mathrm{(set, nat)}\ \mathbf{moding}\ \mathrm{(enum, any)}$$
$$\quad \mathrm{sequential(insert(N, insert(s(N),)), s(N)).}$$

entails adding the clause

$$\mathrm{sequential(A_{set}(cons(N, cons(s(N), nil))), s(N)).}$$

to the converted program.

We now consider predicate definitions whose abstract and concrete signatures differ. Let P be such a predicate defined within a cluster S. Let $\textsc{abstsig}(P) = (S_1, \ldots, S_n)$ and $\textsc{concsig}(P) = (R_1, \ldots, R_n)$. The first step in the conversion of P is to add the clause

$$\mathrm{P(t_1, \ldots, t_n) \leftarrow \overline{P}(v_1, \ldots, v_n)}$$

to the converted program. The v_i are fresh variables; if $S_i = S$ then t_i is $A_S(v_i)$ and is v_i otherwise.

For example, consider the definition of *size*, which appears within the explicitly implemented *set* cluster.

$\mathrm{cons}(X, \mathrm{cons}(Y, B)) = \mathrm{cons}(Y, \mathrm{cons}(X, B))$

set **predicate** = setUnify

nat **moding** = (any, any)
bag **moding** = (any, enum), (enum, any)
set_dnt **moding** = (any, any)
set **moding** = (any, any)

Figure 7.7: Unification information of converted program

---

size = **pred** (set, nat) **moding** (gnd, any)
$\quad \mathrm{size}(B_1, N) \leftarrow \mathrm{reduce}(B_1, B_2), \mathrm{length}(B_2, N).$

Its abstract signature is (*set*, *nat*) while its concrete signature is (*bag*, *nat*). The converted clause is

$$\mathrm{size}(A_{\mathrm{set}}(B), N) \leftarrow \overline{\mathrm{size}}(B, N).$$

The effect of this clause is to transform the *set* argument in *size* into a *bag* argument in $\overline{size}$.

The second step in the conversion of a predicate such as P is to convert each of the clauses of its body. Suppose that one of these clauses is of the form

$$P(t_1, \ldots, t_n) \leftarrow L_1, \ldots, L_m.$$

The clause

$$T_{\mathrm{literal}}(\overline{P}(t_1, \ldots, t_n)) \leftarrow T_{\mathrm{literal}}(L_1), \ldots, T_{\mathrm{literal}}(L_m).$$

is added to the converted program. (There is no sort inconsistency between P and $\overline{P}$ because the concrete signature applies to P.) For example, the clause that constitutes the body of *size* is converted to

$$\overline{\mathrm{size}}(B_1, N) \leftarrow \mathrm{reduce}(B_1, B_2), \mathrm{length}(B_2, N).$$

7.2.3 Unification conversion

The third step of conversion is to carry over the information that pertains to unification. The classes of information that must be extracted are the equations from implicitly implemented clusters, the unification predicates from the explicitly implemented clusters, and the mode restrictions upon unification.

The unification informal is collected in three steps. First, each of the unification equations from the implicitly implemented clusters are accumulated. They form the equational theory that underlies the converted program. The single equation in our converted program appears in the first portion of Figure 7.7.

Second, the unification predicate for each explicitly implemented sort S is recorded. This predicate is UPRED(S). These predicates are used to augment the theory presented by the equations. The unification predicate for *set*, the only explicitly implemented sort in our example, appears in the second portion of Figure 7.7.

Third, the moding that restricts the unification of the terms of each sort S is derived from the implementation of S. Let M be UMODE(S). Then the mode tuples that restrict unification are (any, M) and (M, any). For example, UMODE(bag) $= enum$, so its unification mode tuples are $(any, enum)$ and $(enum, any)$. The unification mode tuples for our example program are recorded in the third portion of Figure 7.7.

7.2.4 Mode conversion

The final step in conversion is to obtain the mode signatures that define the mode names for each sort. There signatures are ultimately used to obtain the moded base of the program, as well as to determine the moding of each predicate symbol. The mode signatures for our converted example appears in Figure 7.8.

The mode signatures are carried over unaltered from the implicitly implemented clusters. The mode signatures for the explicitly implemented sorts are based upon the mode implementations. For each mode mapping of the form

M **from** $\mathrm{N_1, \ldots, N_n}$

in an explicitly implemented cluster S, we include the n mode signatures

$\mathrm{A_S}$: $\mathrm{N_i} \rightarrow \mathrm{M}$.

in the converted program.

For example, the mode implementation section of the *set* cluster is

(nat) any > gnd
    0: $\rightarrow$ gnd
    s: gnd $\rightarrow$ gnd
    s: any $\rightarrow$ any

(bag) any > enum > gnd
    nil: $\rightarrow$ gnd
    cons: gnd, gnd $\rightarrow$ gnd
    cons: any, enum $\rightarrow$ enum
    cons: any, any $\rightarrow$ any

(set) any > enum > gnd
    $\mathrm{A}_{\mathrm{set}}$: gnd $\rightarrow$ gnd
    $\mathrm{A}_{\mathrm{set}}$: enum $\rightarrow$ enum
    $\mathrm{A}_{\mathrm{set}}$: any $\rightarrow$ enum

(set_dnt) any
    empty: $\rightarrow$ any
    insert: any, any $\rightarrow$ any

Figure 7.8: Mode signatures of converted program

modes any > enum > gnd
moded by
    gnd **from** gnd
    enum **from** enum
    any **from** enum

Its converted form is

any > enum > gnd
    $\mathrm{A}_{\mathrm{set}}$: gnd $\rightarrow$ gnd
    $\mathrm{A}_{\mathrm{set}}$: enum $\rightarrow$ enum
    $\mathrm{A}_{\mathrm{set}}$: any $\rightarrow$ enum

7.3 Translating denotations to representations

The only portion of the conversion between denotation and representation form programs that we have not defined is the translation function $T_{literal}$. This function expects as an argument a literal whose terms are written as denotations, and returns a literal whose terms are written as representations. In the two forms the terms should stand for equivalent abstract objects, where equivalence is defined by the translation predicate that appears in each explicitly implemented cluster.

Before proceeding with the definition of $T_{literal}$, we establish the idea behind it by working through an example. Consider the abstract set object that contains the natural numbers 1 and 2. One of its denotations is the *set* term $insert(0, cons(1, empty))$, and one of its representations is the *bag* term $cons(0, cons(1, empty))$. We could, in principle, use this second term to stand for the *set* in a representation form program. Because of overloading, however, we would be unable to tell whether a term in representation form was a *set* or a *bag*.

We take a different approach instead. We use the term $A_{set}(cons(0, cons(1, empty)))$ to stand for the *set*discussed above. The abstracting function symbol A_{set} is a syntactic coercion that converts representations of abstract objects to the abstract sort. The translation process involves using the translation predicates to find the representations that correspond to denotations, and then applying abstracting functions as necessary to preserve sort correctness.

The translation is done beginning at the inside of the term and moving out. The first step in translating the term $insert(0, cons(1, empty))$, for example, is to translate the innermost term *empty*. To do this, it is necessary to regard *empty* not as a term of sort *set*, but as a term of sort *set_dnt*, because the translation function is written to expect arguments of sort *set_dnt*. This is possible because of the deliberate overloading of the constructor symbols between *set* and *set_dnt*.

The set translation predicate *setTrans* maps *empty* to the *bag* term *nil*. (The *nat* term *1*, which is also an innermost term, requires no translation as it is of an implicitly implemented sort.) We replace the innermost terms with their representations and consider the next term out, which is now $cons(1, nil)$. With its subterms having been replaced with representations, this term is now a well-sorted *set_dnt* term. Consequently, we can use *setTrans* to translate it to its representation, $insert(0, cons(1, nil))$.

Continuing in this fashion for one more step, we obtain the representation version of the original abstract object. Following the application of the abstracting function symbol, the translation is complete. It is possible for abstracting functions to be nested; this poses no problem.

Although we began by posing the problem of translating literals, we have proceeded by considering the problem of translating terms. This is because the function $T_{literal}$ does nothing more than use a subsidiary function to translate each of the terms in its argument literal.

$$\mathrm{T_{literal}(P(t_1, \ldots, t_n)) = P(\hat{t}_1, \ldots, \hat{t}_n)}$$
$$\text{where } \mathrm{\hat{t}_i = T_{term}(t_i, S_i)} \text{ and } \mathrm{SIG(P) = (S_i, \ldots, S_n)}$$

It is the function T_{term} that does the term translation.

In the remainder of this section we will describe more rigorously the translation process in general and T_{term} in particular. In Section 7.3.1 we show how T_{term} can be defined over ground terms, and then in Section 7.3.2 we show how T_{term} can be extended to cope with variable-containing terms. Finally, in Section 7.3.3, we describe how this translation process can be used to incrementally transform a program that itself contains the translation predicates.

7.3.1 Translating ground terms

The function T_{term} takes two arguments. The first is the denotation t of an abstract object, and the second is the sort S of that term. (This second argument is necessary because of overloading.) T_{term} translates t into a representation form term of the same sort.

$$\mathrm{T_{term}(t, S)} = \begin{cases} \mathrm{\hat{t}}, \text{ if implicit(S)} \\ \mathrm{A_S(\hat{t})}, \text{ if explicit(S)} \end{cases}$$
$$\text{where } \mathrm{\hat{t} = T_S(t, S)}$$

There are two cases, depending upon whether the S is implicitly or explicitly implemented. In both cases, the function T_S is invoked to perform the translation. In the latter case, the abstracting function symbol A_S is used to enclose the representation as discussed earlier.

There is one instance of the function T_S for every sort S. There are two explicit arguments. The first is the denotation t of an abstract object, and the second is the sort R of t. The implicit argument S is the sort of the term in which t is embedded. T_{term} translates t into a representation of t.

$$\mathrm{T_S(f(t_1,\ldots,t_n),R)} = \begin{cases} \mathrm{T_{term}(f(t_1,\ldots,t_n),R),} & \mathrm{if\ S \neq R} \\ \mathrm{translates_S(f(\hat{t}_1,\ldots,\hat{t}_n)),} & \mathrm{otherwise} \end{cases}$$
$$\mathrm{where\ \hat{t}_i = T_S(t_i,R_i)\ and\ \textsc{sig}(f,R) = (R_1,\ldots,R_n \to R)}$$

There are two cases. If R and S are not the same, then t is not the same sort as its parent term. Since the translation of t must be embedded in an abstracting function symbol, the function T_{term} is recursively invoked in this case. If R and S are the same, the subterms are recursively translated and the resulting term is then translated by the function $translate_S$.

If S is an implicit sort, then the function $translate_S$ is the identity function. Otherwise, the function application $translate_S(t)$ is reduced as follows. The query

$$\leftarrow \mathrm{translates_S(t,V)}$$

is formed and solved to obtain a substitution σ. Here, $translate_S$ is the translation predicate from the sort S and V is a fresh variable. The result of the function application is σV.

7.3.2 Translating general terms

The definitions of T_S and $translate_S$ are given above only for ground terms. In this section we show how to extend the translation functions to deal with variables and variable-containing terms. We first discuss why variables must be treated as a special case, and then give the extensions.

In principle, a *set* variable, such as S, should be translated to an abstracted variable, such as $A_{set}(B)$, where B is a *bag* variable. It is important to ensure that all occurrences of a variable in a program are translated to the same abstracted variable. We ensure this by reusing the variable names of an abstract sort in a denotation form program as the variable names of the representation sort in the converted program.

Thus, the denotation variable S is translated to the representation variable $A_{set}(S)$. There is no danger in reinterpreting S as a *bag* variable since all occurrences of S will be translated. We now see how this is accomplished.

The definition of T_S can be extended to account for variables as follows. If S is an implicit sort, then

$$\mathrm{T_S(v, R) = v}$$

because such a variable requires no translation. For explicitly implemented sorts S,

$$\mathrm{T_S(v, R) = T_{term}(v, \text{REP}(R))}$$

By invoking T_{term}, we ensure that the proper abstracting function is wrapped around the variable.

The difficulty encountered by $translate_S$ when applied to terms that contain variables is more subtle. Consider the solution of the query

$$\leftarrow \mathrm{translate_{set}(insert(N, nil)), V).}$$

The substitution that we would like to obtain is $\langle V/cons(N, nil\rangle$, which is the most general possible solution. However, the substitution $\langle V/cons(1, nil)\rangle$, with the variable N instantiated to 1, is also a solution.

When solving a translation query, we must require that the variables of the term being translated be treated as uninterpreted constants rather than as variables. With this restriction, we avoid the possibility of the program variables being accidentally, and unnecessarily, instantiated.

7.3.3 Translation paradigm

In our discussion of the translation process, we have assumed that the literals and terms being translated are separate from the program that defines the translation predicates. When translating a Denali program, this will not be the case.

Because of this, Denali programs must be translated incrementally. Before a term of an explicit sort S can be converted to representation form, the translation predicate for S must itself be translated. This will always be possible so long as terms of a sort S_dnt are not permitted to contain objects of sort S. This is in fact a restriction of Denali.

7.4 Abstract meaning

Because a Denali program in representation form is already close to an equational definite clause program, its semantics can be given directly. Recall that a moded equational definite clause program is a triple $(H; E; M)$ composed of a set of definite clauses H, an equational theory E^*; and a set of predicate and unification modings M. The triple is based upon a set of sorts $\mathbf{S}$, a set of function symbols $\mathbf{F}$, a set of predicate symbols $\mathbf{P}$, and a set of variable symbols $\mathbf{V}$.

In this section we will show how to extract a moded equational definite clause program from a Denali program that is in representation form. The meaning of the definite clause program, as developed in Chapter 5, is also the meaning of the Denali program.

We begin with the sets of symbols $\mathbf{S}$, $\mathbf{F}$, $\mathbf{P}$, and $\mathbf{V}$. The first three of these sets are already components of a representation form program. The choice of the variable name universe $\mathbf{V}$ is of technical interest only. We will assume, without loss of generality, that variable names are used consistently throughout Denali programs. This condition can be imposed, if necessary by a systematic renaming of variables. Under this assumption, we can choose $\mathbf{V}$ to be any consistently sorted superset of the program variables such that there are a countably infinite number of variables of each sort.

The set H of definite clauses is the set of clauses that appears in the representation form program.

A set R of equations appears in a representation form program. This set cannot be the sole basis of the equational theory E^* because it constrains only the implicitly implemented sorts. Instead, we must define E^* indirectly to be the smallest equational theory with the following two properties. First, it must contain R. Second, for all terms $A_S(t_1)$ and $A_S(t_2)$ of an explicitly implemented sort S whose unification predicate is P, it must be the case that $P(t_1, t_2) \in H_E^*$ if and only if $t_1 = t_2 \in E^*$.

Finally, the set M of predicate and unification modings can be obtained from the mode tuples of the representation form program by interpreting the mode name presentations as described in Chapter 2.

7.5 Operational Meaning

In this section we describe the meaning of Denali programs in an operational fashion by sketching an abstract interpreter. This interpreter is defined over Denali programs written in representation form, with the program treated as an implicit argument. In addition to objects of the abstract syntax, the interpreter manipulates and returns substitutions.

We will describe the interpreter in two stages. We begin in Section 7.5.1 by giving the giving the bulk of the interpreter. It mimics the incremental generation and traversal of an W-ESL tree for a query. Next, in Section 7.5.2 we consider unification, and show how the *CRunify* iterator of Chapter 6 can be harnessed to perform unification in Denali.

7.5.1 Interpreter

The iterator *interpret* is the top level of the interpreter. It takes a query as an argument and yields a possibly infinite sequence of substitutions as a result. If a mode failure occurs at any point during interpretation, it signals *modeFailure*.

interpret = **iter** (q: query) **yields** (subst) **signals** (modeFailure)
  for $\sigma \in$ solveQuery(q) **do**
    yield(σ)
    end resignal modeFailure
  end interpret

The mutually recursive iterators *solveQuery* and *solveLiteral* are used to simultaneously construct and search, in depth-first fashion, an W-ESL tree for a query.

The base case of *solveQuery* is the empty query, which is solved immediately by the empty substitution. If the query is nonempty, then a well-moded literal is selected and solved. This amounts to completely solving a node of an W-ESL tree. The outer loop corresponds to solving the left children, and the inner loop corresponds to solving the right children. If there is no well-moded literal, or if a mode failures occurs when solving one of the children, then *modeFailure* is signalled and the attempted solution terminates.

```
solveQuery = iter (q: query) yields (subst) signals (modeFailure)
    case
        |q| = 0 ⇒ yield(⟨⟩)
        else ⇒
            let l = select(q)
            for σ1 ∈ solveLiteral(l) do
                for σ2 ∈ solveQuery(σ1(q − l)) do
                    yield(σ2 ∘ σ1)
                    end
                end
        end resignal modeFailure
    end solveQuery
```

Because unification literals must be solved using different techniques from all other literals, *solveLiteral* maps literal solution to two separate cases.

```
solveLiteral = iter (l: literal) yields (subst) signals (modeFailure)
    case
        l = unify(t1, t2) ⇒ reyield(CRunify(t1, t2))
        else ⇒ reyield(solveRegular(l))
        end resignal modeFailure
    end solveLiteral
```

For each clause from the predicate that implements the head symbol of the literal, *solveRegular* forms and solves the overlap reduction.

```
solveRegular = iter (l: literal) yields (subst) signals (modeFailure)
    for q ∈ overlaps(l) do
        for σ ∈ solveQuery(q) do
            yield(σ)
            end
        end resignal modeFailure
    end solveRegular
```

The procedure *select* returns an arbitrary well-moded literal from its argument *query*, and signals *modeFailure* if there is none. Its implementation would make use of mode signatures as described in Chapter 2.

```
select = proc (q: query) returns (literal) signals (modeFailure)
```

The iterator *overlaps* yields each of the overlap reductions of its argument *literal*.

```
overlaps = iter (l: literal) yields (query)
```

That is, assuming l is of the form $P(t_1, \ldots, t_n)$, it yields the query

$$\mathrm{unify}(t_1, r_1), \ldots, \mathrm{unify}(t_n, r_n), L_1, \ldots, L_m$$

for each program clause of the form

$$P(r_1, \ldots, r_n) \leftarrow L_1, \ldots, L_m.$$

7.5.2 Unification

We rely upon the iterator *CRunify* to unify pairs of terms in the implementation of *solveLiteral*. Its definition in Chapter 6 assumes the existence of a subsidiary unification procedure U_S corresponding to each sort S. In this section we show how, by suitably defining these procedures U_S, *CRunify* can be harnessed for use in the abstract interpreter.

Recall that *CRunify* makes three requirements concerning the subsidiary procedures U_S. They are

- The equations constraining each implicit sort S must contain only sort-regular equations.
- For each implicit sort S, U_S must be defined over the set of terms that are homogeneous in $\mathbf{F}_S$.
- For each explicit sort S, U_S must be defined over all terms of sort S.

We now show how each of these requirements is satisfied.

The first requirement is guaranteed by a static semantic constraint upon implicitly implemented clusters.

The unification predicates U_S of the second requirement are provided by the language implementation. Consequently, only sets of equations for which unification procedures are known to the implementation can be incorporated into implicit implementations of clusters.

The unification predicates U_S of the third requirement are based upon the unification predicates supplied by the programmer within explicitly implemented clusters. The following iterator U_{set} illustrates the principle.

Requires: The arguments t_1 and t_2 are of sort set.

U_{set} = **iter** (t_1: term, t_2: term) **yields** (subst) **signals** (modeFailure)
assume $t_1 = A_{set}(r_1)$
assume $t_2 = A_{set}(r_2)$
for $\sigma \in$ solveQuery(setUnify(r_1, r_2)) **do**
yield(σ)
end resignal modeFailure
end U_{set}

The iterator forms and solves a query based upon the unification predicate for *set*, which is *setUnify*.

8 Conclusions

In this dissertation we have described the design of Denali, an equational logic programming language. Denali is based upon the premise that programs and pieces of programs written in logic languages should be separately specifiable and implementable, just as in conventional languages. The major design goal of Denali was to support such a programming methodology.

Achieving this goal required developing a coherent collection of new approaches to organizing, implementing, and describing equational logic programs. We summarize these research contributions in Section 8.1. In the course of our research we have identified a number of areas that merit further investigation; we outline these areas in Section 8.2.

8.1 Contributions

The primary contribution of this dissertation is the development of the framework that is the basis of Denali, a logic programming language designed to support programming-in-the-large. Elaborating this framework involved identifying the two forms of abstraction around which we believe logic programs should be organized and showing how programs could be constructed using them.

In most logic languages, programs are composed of individual definite clauses and, in some cases, equations. In Denali, they are composed of implementations of predicate and data abstractions. The cornerstone of predicate abstraction is the idea that multi-valued modes should be used, as part of a two-dimensional type system, to express structural restrictions upon arguments. The cornerstone of data abstraction is the idea that programmers should be helped to implement equational unification procedures.

The mode system of Denali is more pervasive than that of logic languages in which modes are used only as annotations that help control the order of interpretation. In Denali, modes are exploited in almost all aspects of an implementation. In addition to helping control the interpreter, modes document predicate interfaces, provide control flow by serving as guards of clauses, and help simplify the implementation of unification by restricting the formation of objects. The runtime checking of mode restrictions serves to catch programming errors that would otherwise be undetected.

Denali modes are also more expressive than those of other languages. Existing languages provide modes that distinguish only between variables and non-variables. These bi-valued modes are generic to all types of objects, and thus can be built directly into the language. Denali's multi-valued modes can express the finer-grained distinctions that are needed to fully document predicate interfaces. Because the distinctions that are required depend upon the application, Denali modes must be defined by the programmer. Consequently, we developed a mode signature technique for defining them.

Denali is the first logic language that distinguishes between the way abstract objects are denoted and the way that they are represented. This is one of the aspects of Denali that makes it possible to build programs in layers of abstractions, as is possible in conventional programming languages. It also makes it possible to introduce built-in types with which representations other than terms can be constructed.

We adopted a novel approach to obtaining implementations of equational unification. Other languages attempt to handle unification automatically by synthesizing implementations from equations. The known approaches have limited applicability and almost always produce inefficient implementations. Furthermore, there are theoretical limitations upon both the applicability and performance of such approaches. In Denali, we place the burden of implementing unification procedures upon the programmer. To make this approach feasible, we place at the disposal of the programmer a number of techniques for restricting and thus simplifying unification. Unification procedures are defined on a sort-by-sort basis and then combined by the implementation. Because implementations of abstractions can be layered, unification procedures provided by built-in abstractions can be incorporated into user-defined implementations. Most importantly, modes can be used both to place interface restrictions upon unification procedures and to restrict the formation of objects.

Besides presenting a language design, we have also established the formal basis for Denali. By devising W-ESL resolution, we provided the basis for the semantics of Denali and the basis for constructing specifications and defining satisfaction for Denali programs. By extending an existing algorithm to obtain a procedure for combining moded unification procedures, we established the cornerstone of a Denali interpreter.

8.2 Further work

In Chapter 5 we developed the background necessary for defining the correctness of implementations of abstractions relative to specifications expressed as moded equational definite clause programs. We have not yet completed the definition of satisfaction, so this represents an important avenue for further work. The key to the definition of satisfaction is Theorem 5.15, which establishes the fact that operationally equivalent programs are interchangeable.

We have omitted from our semantics of Denali the consideration of predicates that are implemented by the guarded blocks introduced in Chapter 2. Incorporating guarded blocks into the operational semantics would be a simple extension. Accounting for them in the abstract semantics is not as easy, since these semantics are based upon unguarded definite clause programs.

By fleshing out the design framework that we have presented, and then implementing Denali, it would be possible to obtain experimental evidence of how closely Denali comes to satisfying its design goals. The most critical remaining design issue is determining the set of built-in data abstractions and unification procedures. Similarly, the most challenging part of an implementation would be realizing these built-in components. The rest of the language implementation should be straightforward.

Not surprisingly, modes provide a number of directions for further research. For example, it should be possible to liberalize their definition. We currently require that every mode be closed under both instantiation and unification. By combining these two closure properties and requiring only that a mode be closed under instantiation by substitutions produced through unification, we could obtain a more expressive mode system. One drawback of this approach is that it would eliminate our ability to syntactically check that modes are closed under instantiation.

In a similar vein, a more powerful technique for implementing modes would also add expressive power. The mode signature technique, while simple and efficient, is far from complete. The evolution of modes along the lines that have been followed by polymorphic type systems in programming languages would also be valuable.

It might be possible to exploit modes in domains other than resolution. For example, narrowing and related unification synthesis techniques might be rendered more robust and efficient if they could exploit mode restrictions. This would in turn make the synthesis approach more appealing as a means of implementing an equational logic language.

Finally, there could be other applications for moded unification. Equational unification algorithms are used in a number of contexts other than logic languages, including term rewriting systems and theorem provers. All systems that depend upon equational unification are limited by the availability of efficient unification algorithms. It might be possible to exploit the potentially more efficient moded unification algorithms instead.

References

[Abrial 80] J. R. Abrial. The Specification Language Z: Syntax and "Semantics". Technical report, Programming Research Group, Oxford University, April 1980.

[Arnborg 85] S. Arnborg and E. Tidèn. Unification Problems with One-Sided Distributivity. In J.-P. Jouannaud, editor, *Proceedings of the First International Conference on Rewriting Techniques and Applications*, Dijon, France, pages 398–406, Berlin, May 1985. LNCS 202, Springer-Verlag.

[Barnes 80] J. G. P. Barnes. An Overview of Ada. *Software—Practice and Experience*, 10(11):851–887, November 1980.

[Baxter 73] L. D. Baxter. An Efficient Unification Algorithm. Technical Report CS-73-23, Department of Applied Analysis and Computer Science, University of Waterloo, Waterloo, Ontario, July 1973.

[Benanav 85] D. Benanav, D. Kapur, and P. Narendran. Complexity of Matching Problems. In J.-P. Jouannaud, editor, *Proceedings of the First International Conference on Rewriting Techniques and Applications*, Dijon, France, pages 417–429, Berlin, May 1985. LNCS 202, Springer-Verlag.

[Burstall 81] R. M. Burstall and J. A. Goguen. An Informal Introduction to Specifications using CLEAR. In R. S. Boyer and J. S. Moore, editors, *The Correctness Problem in Computer Science*, pages 185–213. Academic Press, London, 1981.

[Clark 85] K. L. Clark and S. Gregory. Notes on the Implementation of Parlog. *Journal of Logic Programming*, 2(1):17–42, May 1985.

[Clocksin 81] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, 1981.

[Corbin 83] J. Corbin and M. Bidoit. A Rehabilitation of Robinson's Unification Algorithm. In R. E. A. Mason, editor, *Information Processing 83. Proceedings of the IFIP Ninth World Computer Conference*, Paris, pages 909–914, Amsterdam, September 1983. North-Holland.

[Dahl 70] O. Dahl, B. Myhrhaug, and K. Nygaard. The SIMULA 67 Common Base Language. Publication S-22, Norwegian Computing Center, Oslo, 1970.

[Dembinski 85] P. Dembinski and J. Maluszynski. And-parallelism with Intelligent Backtracking for Annotated Logic Programs. In *Proceedings of the 1985 Symposium on Logic Programming*, Boston, pages 29–38, Los Angeles, July 1985. IEEE Computer Society.

[Emden 76] M. H. van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742, October 1976.

[Fay 79] M. Fay. First-order Unification in an Equational Theory. In *Proceedings of the Fourth Workshop on Automated Deduction*, Austin, TX, pages 161–167, February 1979.

[Fribourg 84] L. Fribourg. Oriented Equational Clauses as a Programming Language. *Journal of Logic Programming*, 1(2):165–177, August 1984.

[Garey 79] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., San Francisco, 1979.

[Goguen 86] J. A. Goguen and J. Meseguer. EQLOG: Equality, Types, and Generic Modules for Logic Programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming. Functions, Relations, and Equations*, pages 295–363. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Goldberg 84] A. Goldberg and D. Robson. *Smalltalk 80: The Language and Its Implementation.* Addison-Wesley, Reading, MA, 1984.

[Guttag 85] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in Five Easy Pieces. Technical Report 5, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, July 1985.

[Huet 80] G. Huet and D. C. Oppen. Equations and Rewrite Rules: A Survey. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, New York, 1980.

[Hullot 80] J.-M. Hullot. Canonical Forms and Unification. In W. Bibel and R. A. Kowalski, editors, *Proceedings of the Fifth Conference on Automated Deduction*, Les Arcs, France, pages 318–334, Berlin, July 1980. LNCS 87, Springer-Verlag.

[Jaffar 84] J. Jaffar, J.-L. Lassez, and M. J. Maher. A Theory of Complete Logic Programs with Equality. *Journal of Logic Programming*, 1(3):211–223, November 1984.

[Jeanrond 80] J. Jeanrond. Deciding Unique Termination of Permutative Rewrite Systems: Choose Your Term Algebra Carefully. In W. Bibel and R. A. Kowalski, editors, *Proceedings of the Fifth Conference on Automated Deduction*, Les Arcs, France, pages 335–355, Berlin, July 1980. LNCS 87, Springer-Verlag.

[Jouannaud 83] J.-P. Jouannaud, C. Kirchner, and H. Kirchner. Incremental Construction of Unification Algorithms in Equational Theories. In J. Diaz, editor, *Proceedings of the Tenth EATCS International Colloquium on Automata, Languages, and Programming*, Barcelona, pages 361–373, Berlin, July 1983. LNCS 154, Springer-Verlag.

[Kirchner 85] C. Kirchner. *Methodes et Outils de Conception Systematique d'Algorithms d'Unification dans les Théories Equationalles.* PhD thesis, Centre de Recherche en Informatique de Nancy, UER de Mathematiques, Université de Nancy I, Nancy, France, June 1985.

[Kornfeld 86] W. A. Kornfeld. Equality for Prolog. In D. DeGroot and G. Lindstrom, editors, *Logic Programming. Functions, Relations, and Equations*, pages 279–293. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Kowalski 71] R. A. Kowalski and D. Kuehner. Linear Resolution with Selection Function. *Artificial Intelligence*, 2(3/4):227–260, Winter 1971.

[Kowalski 74] R. A. Kowalski. Predicate Logic as Programming Language. In J. L. Rosenfeld, editor, *Information Processing 74. Proceedings of IFIP Congress 74*, Stockholm, pages 569–574, Amsterdam, August 1974. North-Holland.

[Liskov 81] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual.* LNCS 114, Spriger-Verlag, Berlin, 1981.

[Livesey 76] M. Livesey and J. H. Siekmann. Unification of A+C Terms (Bags) and A+C+I Terms (Sets). Interner Bericht 3/76, Institut für Informatik I, Universität Karlsruhe, 1976.

[Makanin 77] G. S. Makanin. The Problem of Solvability of Equations in a Free Semigroup. *Soviet Mathematics—Doklady*, 18(2):330–334, March-April 1977.

[Malachi 86] Y. Malachi, Z. Manna, and R. Waldinger. TABLOG: A New Approach to Logic Programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming. Functions, Relations, and Equations*, pages 365–394. Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Manna 80] Z. Manna and R. Waldinger. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, January 1980.

[Martelli 82] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[Milner 78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[Naish 85] L. Naish. Automatic Control for Logic Programs. *Journal of Logic Programming*, 2(3):167–183, November 1985.

[Nakajima 80] R. Nakajima, M. Honda, and H. Nakahara. Hierarchical Program Specification and Verification—a Many-sorted Logical Approach. *Acta Informatica*, 14(2):135–155, August 1980.

[Paterson 78] M. S. Paterson and M. N. Wegman. Linear Unification. *Journal of Computer and System Sciences*, 16(2):158–167, April 1978.

[Plotkin 72] G. D. Plotkin. Building-in Equational Theories. In B. Meltzer and D. Michie, editors, *Machine Intelligence 7*, pages 73–90. John Wiley & Sons, Toronto, 1972.

[Porto 82] A. Porto. A Language for Extended Programming in Logic. In M. V. Carneghem, editor, *Proceedings of the First International Logic Programming Conference*, Marseille, pages 31–37, September 1982.

[Réty 85] P. Réty, C. Kirchner, H. Kirchner, and P. Lescanne. NARROWER: a New Algorithm for Unification and its Application to Logic Programming. In J.-P. Jouannaud, editor, *Proceedings of the First International Conference on Rewriting Techniques and Applications*, Dijon, France, pages 141–156, Berlin, May 1985. LNCS 202, Springer-Verlag.

[Robinson 65] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[Schmidt-Schauss 86] M. Schmidt-Schauss. Unification in Many-Sorted Equational Theories. In J. H. Siekmann, editor, *Proceedings of the Eighth International Conference on Automated Deduction*, Oxford, England, pages 538–553, Berlin, July 1986. LNCS 230, Springer-Verlag.

[Shapiro 83] E. Shapiro. A Subset of Concurrent Prolog and its Interpreter. Technical Report TR-003, ICOT, Tokyo, February 1983.

[Siekmann 79] J. H. Siekmann. Unification of Commutative Terms. In E. W. Ng, editor, *EUROSAM '79, An International Symposium on Symbolic and Algebraic Manipulation*, Marseille, pages 530–545, Berlin, June 1979. LNCS 72, Springer-Verlag.

[Slagle 74] J. R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity and Associativity. *Journal of the ACM*, 21(4):622–642, October 1974.

[Sterling 86] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques*. MIT Press, Cambridge, MA, October 1986.

[Stickel 81] M. E. Stickel. A Unification Algorithm for Associative-Commutative Theories. *Journal of the ACM*, 28(3):423–434, July 1981.

[Szabó 78] P. Szabó. The Undecidability of the D+A-Unification Problem. Technical report, Institut für Informatik I, Universität Karlsruhe, 1978.

[Tidèn 86] E. Tidèn. Unification in Combinations of Collapse-Free Theories with Disjoint Function Symbols. In J. H. Siekmann, editor, *Proceedings of the Eighth International Conference on Automated Deduction*, Oxford, England, pages 431–449, Berlin, July 1986. LNCS 230, Springer-Verlag.

[Ullman 85] J. D. Ullman. Implementation of Logical Query Languages for Databases. *ACM Transactions on Database Systems*, 10(3):289–321, September 1985.

[Warren 77] D. Warren. Implementing Prolog—Compiling Predicate Logic Programs, Volume 1. DAI Research Report 39, University of Edinburgh, May 1977.

[Yelick 85] K. A. Yelick. A Generalized Approach to Equational Unification. Technical Report TR-344, MIT Laboratory for Computer Science, Cambridge, MA, August 1985.

Biography

Joseph L. Zachary was born in Taylorsville, North Carolina, where he attended the public school system, graduating from Alexander Central High School in May 1975. He subsequently enrolled at the Massachusetts Institute of Technology and graduated in June 1979 with the S.B. degree in Computer Science and Engineering. He then worked for a year in the Word Processing Group of the Digital Equipment Corporation as one of the last software engineers to program the PDP-8. This experience inspired him to return to MIT, where he entered graduate school in the fall of 1980. His tenure at MIT was interrupted in the summer of 1982, when he worked at the Xerox Palo Alto Research Center, and in the summer of 1983, when he visited the Université de Nancy. He received the S.M. degree in Computer Science in June 1983, and expects to receive the Ph.D. degree in Computer Science in September 1987. He will assume an appointment as Assistant Professor of Computer Science at the University of Utah at that time.