A HIGHER-ORDER LOGIC AS THE BASIS FOR LOGIC PROGRAMMING

Gopalan Nadathur

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

1987

Supervisor of Dissertation

Graduate Group Chairperson

Acknowledgements

Many people have contributed towards making my experience at the University of Pennsylvania a valuable one, and I am grateful to them.

First of all, I would like to thank Dale Miller for being a constant source of encouragement and intellectual input, a patient and tolerant listener and, most importantly, a good friend. The many discussions I have had with my professors and colleagues on matters both pertaining to and unrelated to my thesis research have been valuable to my intellectual development, and for this I thank, especially, Hassan Aït-Kaci, Peter Buneman, Will Dowling, Jean Gallier, Aravind Joshi and Lokendra Shastri. I am indebted to Scott Weinstein for having taught the two courses that I consider the most stimulating in my graduate career. Also, his confidence in me was greatly uplifting during my bouts of melancholic introspection. Thanks are also due Ramesh, Sitaram, Hendrik, Ravi, Bhagi and Chacha, among many others, who have been responsible, in one way or the other, for making my stay in Philadelphia an enjoyable one. These acknowledgements would not be complete without a mention of Ameeta, who has always been there when I needed her. Her editorial comments and careful proof-reading of this thesis are but one manifestation of this fact.

This research has been supported in part by the NSF grants MCS-78-08401, MCS-81-07190 and MCS-82-19196.

ABSTRACT

A Higher-Order Logic as the Basis for Logic Programming

Gopalan Nadathur

Supervisor: Dale A. Miller

The objective of this thesis is to provide a formal basis for higher-order features in the paradigm of logic programming. Towards this end, a non-extensional form of higher-order logic that is based on Church's simple theory of types is used to provide a generalisation to the definite clauses of first-order logic. Specifically, a class of formulas that are called higher-order definite sentences is described. These formulas extend definite clauses by replacing first-order terms by the terms of a typed λ-calculus and by providing for quantification over predicate and function variables. It is shown that these formulas, together with the notion of a proof in the higher-order logic, provide an abstract description of computation that is akin to the one in the first-order case. While the construction of a proof in a higher-order logic is often complicated by the task of finding appropriate substitutions for predicate variables, it is shown that the necessary substitutions for predicate variables can be tightly constrained in the context of higher-order definite sentences. This observation enables the description of a complete theorem-proving procedure for these formulas. The procedure constructs proofs essentially by interweaving higher-order unification with backchaining on implication, and constitutes a generalisation, to the higher-order context, of the well-known SLD-resolution procedure for definite clauses. The results of these investigations are used to describe a logic programming language called λProlog. This language contains all the features of a language such as Prolog, and, in addition, possesses certain higher-order features. The nature of these additional features is illustrated, and it is shown how the use of the terms of a (typed) λ-calculus as data structures provides a source of richness to the logic programming paradigm.

Table of Contents

Chapter 1
Introduction

In a logic programming language such as Prolog the data structures, *i.e.* the devices used to represent objects that a program "computes" on or reasons about, are first-order terms. It is easy to imagine an extension of these data structures to higher-order terms or, more specifically, to the terms of a λ-calculus in which predicate and function variables can appear free. Let us, then, imagine such an extension and examine some of its benefits.

There are several kinds of objects whose representation, in a logically correct manner, requires a term language that incorporates higher-order notions. Examples of these kinds of objects are provided by programs and formulas. The task of describing the denotations of programs, for instance, requires an allusion to the operations of abstraction and application. In order to represent programs in a fashion closely related to their meanings requires the data structures provided, for instance, by λ-terms. In the same vein, an adequate characterisation of the operation of quantification in first-order formulas requires the use of data structures in which the notions of bound variables and the scopes of variable bindings can be represented. These notions cannot be captured easily in data structures that are based only on first-order terms. They can, however, be represented in a rather direct manner using data structures that provide the notion of variable abstraction in conjunction with first-order terms.

The provision of λ-terms in a logic programming language would, thus, provide us with a facility in representing certain kinds of objects that cannot be adequately represented by first-order terms. If function variables are also permitted in our λ-terms, we could use terms in which such variables appear free as schemata that represent classes of objects whose meaning have a common "compositional structure." Clauses in our hypothetical logic programming language could then be used to specify logically meaningful relationships between classes of objects thus represented. For specific examples where such an ability might be useful, consider the following.

(i) Rules of logical inference can be described as relationships between formula schemata. Given that such schemata can be represented by the data structures of our hypothetical logic programming language, the process of deduction in a particular logical system can easily be specified by clauses in our language. (Examples illustrating such a possibility appear in [27] and [34].)

(ii) Certain kinds of program transformations [12] can be thought of as relationships between program schemes. Program schemes can be represented by λ-terms in which function variables appear free [23]. Clauses in our hypothetical language may therefore be used to encode, and thus specify, such program transformations.

If "specifications" in our language could also be executed, we would in effect have a lan-

guage that supports, amongst other things, the implementation of program transformation systems and of inference systems in a clean and easily justifiable manner.

The provision of predicate variables in addition to other function variables would, on the other hand, provide a different kind of facility in our language. In the context of a logic programming language, predicates correspond to names of procedures. If predicate variables are permitted to appear as the arguments of predicates and also as the names of predicates, our language would provide an ability to write procedures that abstracted over other procedures. If our language also permits λ-terms to contain logical connectives, then such expressions could be used, in the course of an evaluation, to instantiate predicate variables that appear as arguments. Since the predicate variables they instantiate may themselves appear as the names of procedures, these expressions may later be "evaluated." We would thus obtain in our language some of the facilities provided by a functional language such as Lisp for constructing λ-expressions that may be passed as parameters and, later, invoked as programs.

The above observations reveal a potentially rich realm of applications for a higher-order logic programming language. While the addition of higher-order features to a language like Prolog has been previously considered, past work has been restricted to introducing some of these features through *ad hoc* mechanisms and has not understood the true potential of the others. The value of predicate variables has, for instance, been realised by analogy with Lisp-like languages. However, attempts to obtain their usefulness have either been restricted to describing techniques for encoding them within the first-order language [43] or have lead to the introduction of "extra-logical" predicates (like `call` and `univ`) into logic programming languages. The value of function variables in conjunction with λ-terms, on the other hand, does not seem to have been recognised. It is common to dismiss their addition with the observation that the unification problem for higher-order terms is undecidable, without a proper analysis of why this is undesirable in a theoretical sense and without considering whether its use in practice might actually lead to conceptually elegant solutions to difficult problems.

The above arguments provide the major motivation for the work undertaken in this thesis. It is our goal to describe a logic programming language that realises the full complement of higher-order features discussed above that is, at the same time, based on sound theoretical principles, and to expose, in a more tangible fashion, the applications that exist for such a language.

Section 1: Outline of Thesis

Description of a Higher-Order Logic. Our primary focus in this thesis is to describe a logical basis for the introduction into a logic programming language of the higher-order features discussed above. The proper theoretical framework in which to seek such a basis is obviously that of a higher-order logic. The term "higher-order logic," as it is often understood, pertains to a logic whose language admits function and predicate variables, and in which such variables are interpreted as ranging over arbitrary functions and relations on any given domain. By virtue of Gödel's incompleteness theorem, it is known that a logic of this kind is not recursively axiomatizable and that its set of valid sentences is not effectively enumerable. Such a logic is not very interesting from our viewpoint, since our purpose is to use theorem-proving as the method of computation. There is, however, a higher-order logic that involves a weaker notion of quantification that can be recursively axiomatized. The Simple Theory of Types, presented by Church in [10], is a typed λ-calculus formulation of this logic, and it is a non-extensional version of this system that we use as the basis for our analysis. There are several reasons to believe that this logic is particularly apt for our purposes: The language of this logic provides the mechanism of the typed λ-calculus for constructing function and predicate terms and permits variables that range over such constructions; this was the main reason for our quest for a higher-order logic. The proof-theory for this logic bears a close resemblance to that of first-order logic; there is, for instance a generalisation to Herbrand's Theorem [26, 4] that holds for (a variant of) this logic. Finally, the problem of unification for terms in this logic has been studied, and a unification procedure has been described for these terms [22]; this property is of obvious importance when one considers the computational uses of a logic.

We describe this logic in Chapter 2, and present several of the logical notions that we need in this thesis. Since we find it more convenient to cast our subsequent discussions in the framework of a sequent calculus, this chapter also presents a Gentzen-style formulation of the logical system. The chapter ends with a discussion that highlights the similarities between theorem-proving in first-order logic and higher-order logic and points out the manner in which predicate variables make it difficult to naively extend theorem-proving techniques developed in the context of first-order logic to higher-order logic.

Generalising Definite Clauses. The basis of (first-order) logic programming languages is in first-order formulas that may be called *goal formulas* and *definite clauses**. A goal formula is defined (recursively) as being either an atomic formula or the conjunction of two goal formulas. A definite clause is the universal closure of an atomic formula or of a formula of the form $G \supset A$, where A is an atomic formula and G is a goal formula. Let $\mathcal{P}$

* It is more usual to describe these formulas jointly as Horn clauses [41] and to view logic programming in a refutational setting. For the purposes of this thesis, we find it more convenient to use the form of presentation adopted above.

be a set of definite clauses and let G be a syntactic variable for a goal formula. Then, the programming use of these formulas is dependent on the following property of provability in first-order logic:

(i) $\mathcal{P} \vdash \exists x_1 \ldots \exists x_n G$ if and only if there is a substitution instance G' of G such that $\mathcal{P} \vdash G'$.

This property allows us to identify a collection of definite clauses as a program and a goal formula as a query and to use the notion of provability to provide an abstract description of a computation; the computation is to be the construction of a proof for the existential closure of the query from the program, and the result of such a computation is to be a set of substitutions for the free variables in the query. Within this paradigm, a definite clause is interpreted as a procedure declaration and a goal formula is interpreted as a sequence of procedure calls. The suitability of such an interpretation is justified by the following properties of provability:

(ii) $\mathcal{P} \vdash G_1 \wedge G_2$ if and only if $\mathcal{P} \vdash G_1$ and $\mathcal{P} \vdash G_2$.

(iii) If A is an atomic formula, then $\mathcal{P} \vdash A$ if and only if

  (a) A is a substitution instance of a definite clause in $\mathcal{P}$, or

  (b) there is a G such that $G \supset A$ is a substitution instance of a definite clause in $\mathcal{P}$ and $\mathcal{P} \vdash G$.

Thus, an atomic formula corresponds to a procedure, the name of which is the head of the formula. In this context, a definite clause $\forall \bar{x}(G \supset A)$ may be thought of as a (partial) declaration of a procedure whose name is the head of A; it may be invoked nondeterministically and would lead, in turn, to a sequence of procedure calls.

Our objective is to provide a basis for the introduction of higher-order features into the paradigm of logic programming. Such a basis is obtained by using the formulas of our higher-order logic to describe a generalisation of the first-order notions of definite clauses and goal formulas. In very rough terms, the generalisation may be described as the one obtained by permitting predicate and function variables and typed λ-terms into these formulas. Some restrictions are placed on the appearances of predicate variables in definite clauses and of logical connectives in terms, but these restrictions are well motivated in the programming context. The restrictions may, in fact, be understood in the following fashion. First, the name of a procedure defined by a definite clause, *i.e.* the head of A in a (generalised) definite clause $\forall \bar{x}(G \supset A)$, cannot be a variable. Second, only those logical connectives that may appear in the top-level logical structure of a goal formula are permitted in terms*. The intuitive picture here is that a term that appears as the

* In the context of the present discussion, this means that the only logical connective permitted in terms is $\wedge$. It is, however, possible to generalise the syntax of goal formulas to include the logical connective $\vee$ and existential quantification, and we do this in

argument of a procedure call may be used to instantiate a predicate variable in the body of a procedure declaration; in such a case we expect the result of the instantiation to be a query. With these restrictions, we shall observe that our generalisations, together with the notion of provability in our higher-order logic, provide an abstract description of computation in much the same way as in the first-order case.

The actual realisation of the computational paradigm of logic programming depends on the description of a procedure for constructing proofs. In the first-order case such a procedure, called SLD-resolution [5], is obtained directly from the properties (ii) and (iii) above and the use of unification. More care must be taken in the higher-order case, however. There are two potentially complicating factors:

(1) Unification alone does not suffice as a means for finding substitutions for variables in a higher-order logic. Predicate variables may appear as the heads of formulas and, in such cases, the terms that need to be substituted may include logical connectives and quantifiers. Unification does not encompass this richer notion of substitution.

(2) Unification for first-order terms is a simple operation. However, for higher-order terms this is a more complex and, in fact, an undecidable operation.

Fortunately, the first problem is easily resolved in the context of our restricted sets of formulas. For these formulas we find that the necessary substitutions for predicate variables are, in most cases, provided through unification. When unification does not suffice, the appropriate substitution is rather easily determined. With regard to the second problem, we note that the search space for a unifier may be described by a finitely branching tree, the MATCHING tree of [22], and shares several characteristics with the search space for a proof in the case of first-order definite clause logic which is described by an SLD-tree in [5]. With this observation it is possible to describe a theorem-prover in the higher-order case whose search space may be seen as the amalgamation of an SLD-tree and a MATCHING tree. The notion of a $\mathcal{P}$-derivation that we shall describe (in Chapter 3) corresponds to a branch through such a tree.

A precise description of the higher-order generalisation to definite clauses and an investigation of the various properties of this generalisation is the subject of Chapter 3.

A Higher-Order Logic Programming Language. In Chapter 4 of this thesis we describe an experimental logic programming system called λProlog that, in its current incarnation, is based largely on the generalisation to definite clauses discussed in this thesis. This system may, in an informal sense, be described as a *typed*, *higher-order* version of Prolog. The purpose of types in λProlog, however, is mainly to distinguish between objects of different functional types. It is possible for a user of the system to treat all first-order objects as objects of the same type, and, in this sense, the language of this system actually

Chapter 3.

contains that of Prolog.

In a manner very similar to the first-order case, the task of designing an interpreter for λProlog involves making trade-offs between completeness and practicality. The choices in the context of a higher-order language are, however, somewhat more complex than in the first-order case, and we discuss some aspects of these choices in Chapter 4. The current interpreter for λProlog performs a depth-first unification-first search with backtracking when attempting to solve a goal. This is very much like the search a standard Prolog interpreter performs, the main difference being that unification is a more difficult operation in the higher-order case, and may involve a branching search. The interpreter may, therefore, need to backtrack not only over choice of clauses but also over unifiers, and so must record such choices in the course of its search.

Our experiences with λProlog have revealed that it is useful to have the facility for writing "polymorphic" procedures. We have provided such a facility by permitting type variables to appear in the types of symbols in a procedure declaration. One of the virtues of this provision is that a user may omit type declarations if she so wishes since such declarations may generally be inferred by using the techniques of [29]. However, type variables constitute a strictly metalinguistic facility since the underlying theory requires that all such variables in a procedure declaration be instantiated before the procedure is used. In our implementation of an interpreter for λProlog we have adopted some techniques that permit the instantiations of type variables to be delayed in the hope that they may be uniquely determined at a later stage. We discuss these techniques in Chapter 4.

Our main interest in λProlog is in that it constitutes a logic programming language that it incorporates a set of data structures that are based on higher-order terms, and we illustrate some of the uses for these in Chapter 4. We show here the use of predicate variables to write procedures that may take complex queries as arguments. The more novel and, consequently, more interesting additions, however, are the λ-terms and function variables. λ-terms together with λ-conversion provide us with a notion of substitution that is useful in certain contexts. We illustrate this by considering the task of generating logical forms from English sentences. Function variables, together with unification, provide us with a sophisticated mechanism for pattern matching, and we illustrate the use of this mechanism by considering two different tasks. One of these involves the task of writing an interpreter for a logic programming language and, therefore, of performing manipulations on logical formulas. The other involves the task of effecting transformations between programs. These discussions also reveal the suitability of λ-terms as data structures for representing the kinds of objects that are being reasoned about.

Chapter 2
A Higher-Order Logic

The higher-order logic, $\mathcal{T}^*$, that we shall use in this thesis is related to Church's formulation of the simple theory of types [10] and to the system $\mathcal{T}$ in [1]. The language of $\mathcal{T}^*$ is essentially the one Church uses in his formulation of a higher-order logic. This language is of interest to us because it incorporates the rather elegant mechanism of the λ-calculus for providing an understanding of higher-order terms. The language, in addition, uses the notion of types to provide explicit syntactic distinctions between expressions that denote different kinds of intuitive objects; we believe that these distinctions are useful when we consider the programming applications of the logic. The formal system of $\mathcal{T}^*$ corresponds closely to that of [1], which is itself derived from Church's system principally by the exclusion of the axioms concerning infinity, choice, extensionality and description. Our interest in $\mathcal{T}$ is motivated by a desire for a logic that generalises first-order logic by providing a stronger notion of a variable, but at the same time encompasses only the most primitive logical notions that are relevant in this context; again, it is our belief that only these notions are of consequence in the computational applications that we envisage for a higher-order logic. $\mathcal{T}^*$ differs from the system in [1] primarily in that it incorporates an understanding of a larger set of propositional connectives and in that existential quantification is used as the primitive notion in its formulation. These choices are motivated by the considerations in Chapter 3. Despite these differences, the two systems are equivalent in a sense that may be made precise, and several of the properties of $\mathcal{T}$ carry over to $\mathcal{T}^*$.

The purpose of this chapter is primarily to present the system $\mathcal{T}^*$ and to review some logical notions that are relevant either to its formulation or to an understanding of the other parts of this thesis. In sections 2.1 and 2.2 below we present the language of $\mathcal{T}^*$ and review some of the aspects of λ-conversion. Following this, we present a Hilbert-style axiomatisation of the logic. We find it more convenient to cast our subsequent discussions of $\mathcal{T}^*$ in the framework of a Gentzen-style sequent system. To facilitate such a discussion we describe a sequent calculus in Section 2.4 and in Section 2.5 we relate this calculus to $\mathcal{T}^*$.

Section 1: The Language of $\mathcal{T}^*$

As mentioned earlier, the language of $\mathcal{T}^*$ is derived from that of Church's simple theory of types. This language is *typed*, in the sense that every well-formed expression in the language has a type associated with it. The purpose of this type is to determine the position of the expression in a functional hierarchy. The primary mechanisms for constructing higher-order terms are those of function abstraction and application. We describe these notions below and introduce some of the notations that we find useful in other parts of this thesis.

2.1: The Language of $\mathcal{T}^*$

Types. We are initially supplied with a set $\mathcal{S}$ of *sorts*, containing the distinguished sort o and at least one other sort. We are also provided with a set $\mathcal{C}$ of *type constructors*, each specified with a unique positive arity; formally, $\mathcal{C}$ is a set of pairs $\langle c, n\rangle$ where n is a positive integer and for each c at most one such pair belongs to $\mathcal{C}$. The class of *types* relative to $\mathcal{C}$ and $\mathcal{S}$ is then the smallest collection that satisfies the following properties:

(i) Each sort is a type.

(ii) If $\langle c, n\rangle \in \mathcal{C}$ and $\alpha_1, \ldots, \alpha_n$ are types then $(c\,\alpha_1\ \ldots\ \alpha_n)$ is a type.

(iii) If α and β are types, then $(\alpha \to \beta)$ is a type.

In the rest of this thesis, we shall use the letters α and β, perhaps with subscripts, as syntactic variables for types. In writing types, we often omit the surrounding parentheses; in restoring such an expression to a uniquely readable form, we assume that the parentheses around those types formed by virtue of (ii) are to be inserted first and that $\to$ is right associative. We refer to the types obtained by virtue of (i) and (ii) as *atomic types* and to those obtained by virtue of (iii) as *function* types. Evidently every type may be written in the form $(\alpha_1 \to \cdots \alpha_n \to \beta)$ where β is an atomic type. Given such a representation of a type, we refer to $\alpha_1, \ldots, \alpha_n$ as its *argument* types and to β as its *target* type. In an informal sense, each type may be construed as corresponding to a set of objects. Under such a construal, the type $\alpha_1 \to \alpha_2$ corresponds to the collection of functions each of whose domain and range is determined by α_1 and α_2, respectively.

2.1.1. Example. Let us assume that $int \in \mathcal{S}$ and $\langle list, 1\rangle \in \mathcal{C}$. The following are then legitimate types: int, $list\,int$ and $list\,int \to int \to int$. The last of these in an unambiguous form is the type $((list\,int) \to (int \to int))$. The argument types of this expression are $list\,int$ and int, and its target type is int. We may construe the listed types as the types of integers, of lists of integers, and of functions from lists of integers to functions on integers, respectively. Such a reading is a little misleading, though, since no *a priori* interpretation is accorded to the type constructors, and the only sort, as we shall see presently, that has an initial interpretation is o.

Well-Formed Formulas. We now assume that for each type α we are provided with a denumerable set, $\mathcal{V}ar_\alpha$, of *variables* of type α. We also assume that we are supplied with a collection of constants of arbitrary given types, such that the subcollection at each type α is denumerable and disjoint from $\mathcal{V}ar_\alpha$. This collection contains at least one constant of each atomic type, and also includes the following infinite list of symbols called the *logical* constants: $\top$ of type o, $\sim$ of type $o \to o$, $\wedge$ and $\vee$ of type $o \to o \to o$, and, for each α, Σ of type $(\alpha \to o) \to o$. The remaining constants are called the *parameters* of $\mathcal{T}^*$. The class of *well-formed formulas* (*wffs*) or *terms*, relative to the given collections, is now defined inductively by the following rules:

(i) A variable or a constant of type α is a wff of type α.

(ii) If x is a variable of type α_1 and F is a wff of type α_2 then $[\lambda x.F]$ is a formula of type $\alpha_1 \to \alpha_2$.

(iii) If F^1 is a wff of type $\alpha_1 \to \alpha_2$ and F^2 is a wff of type α_1 then $[F^1\ F^2]$ is a wff of type α_2.

In the interpretation intended for the language, λ is to be the abstraction operator, and juxtaposition is to be the operation of function application. In keeping with this intention we refer to the wff in (ii) as the *abstraction* of F by x, and to the wff in (iii) as the *application* of F^1 to F^2. In the former case we also say that the abstraction *binds* x and that its *scope* is F.

There are certain conventions concerning wffs that we shall find useful in this thesis. First, we shall occasionally find it necessary to distinguish between variables and other wffs. For this purpose we employ the convention that wffs indicated by lower-case letters correspond to variables, unless accompanied by an explicit statement to the contrary. Second, we observe that every wff is defined as being of a unique type. This type may be indicated by employing it as a subscript or by an explicit statement; in the latter case we may write "F is a wff_α (variable_α)" as an abbreviation for "F is a wff (variable) of type α". Often the type of the wff is either discernable from the context in which it appears or is inessential to the discussion at hand and in such cases we shall omit its mention. Finally, in displaying wffs, we shall often omit the brackets that surround the expressions formed by virtue of (ii) and (iii) above; such an expression may be restored to a uniquely readable form using the conventions that it is well-formed and that application is left associative.

Based on the rules of formation, we may identify the well-formed subparts of each wff. Specifically, let F be a wff. Then G *occurs* in, or is a *subformula* of, F if (a) G is F, or (b) F is $\lambda x.F^1$ and G occurs in F^1, or (c) F is $F^1\ F^2$ and G occurs in either F^1 or F^2; note that G may have several distinct occurrences in F. An occurrence of a variable x is considered to be either *bound* or *free* depending on whether it is or is not an occurrence in the scope of an abstraction that binds x in F. x is then considered a bound (free) variable of F if it has at least one bound (free) occurrence in F. F is said to be *closed* just in case no free variables occur in it. We use the expression $\mathcal{F}(F)$ to denote the set of free variables of a wff F. This notation is generalised to sets of wffs and sets of pairs of wffs in the following way: If $\mathcal{D}$ is a set of wffs then $\mathcal{F}(\mathcal{D}) = \bigcup\{\mathcal{F}(F) \mid F \in \mathcal{D}\}$ and if $\mathcal{D}$ is a set of pairs of wffs then $\mathcal{F}(\mathcal{D}) = \bigcup\{\mathcal{F}(F^1) \cup \mathcal{F}(F^2) \mid \langle F^1, F^2\rangle \in \mathcal{D}\}$.

2.1.2. Example. Let `cons` be a parameter of type $int \to (list\ int) \to (list\ int)$, and let `1` and `2` be parameters, where *list* and *int* are as in Example 2.1.1. Then

```
λl.[cons 1 [cons 2 l]]
```

is an abbreviated representation of the wff

$$\texttt{[}\lambda\texttt{l.[[cons 1] [[cons 2] l]]].}$$

The type of this wff is evidently $(list\, int) \rightarrow (list\, int)$. The types of `1` and `2` are clearly int — we shall henceforth implicitly assume that the symbols representing the natural numbers are parameters of type int. The above wff has one bound variable, `l`, of type $(list\, int)$. There are no free variables in this wff, although `l` has a free occurrence in the subformula `[cons 1 [cons 2 l]]`.

The type o has a special significance in $\mathcal{T}^*$. Wffs that are of this type are *propositions*, and a wff that has the type $\alpha_1 \rightarrow \cdots \rightarrow \alpha_n \rightarrow o$ is a *predicate* of n arguments whose i^{th} argument is of type α_i. In keeping with our construal of types, predicates of single arguments may informally be thought of as representing sets and predicates of multiple arguments may be thought of as representing relations. In Section 2.3, we describe a formal system that is based on an intention to interpret the logical constants in the following manner: $\top$ corresponds to the tautologous proposition, the *(propositional) connectives* $\sim$, $\vee$ and $\wedge$ correspond respectively to the operations of negation, disjunction and conjunction on propositions, and the family of constants Σ are existential quantifiers, viewed as propositional functions of propositional functions. In writing wffs that contain the logical constants, we shall find the following abbreviations useful:

$$\begin{array}{ccc} [F \vee G] & \text{for} & [\vee\, F\, G] \\ [F \wedge G] & \text{for} & [\wedge\, F\, G] \\ [F \supset G] & \text{for} & [[\sim F] \vee G] \\ [\exists x.F] & \text{for} & [\Sigma\, \lambda x.F] \\ [\forall x.F] & \text{for} & [\sim\Sigma\, \lambda x.\sim F] \end{array}$$

The first two of these abbreviations correspond to the more common practice of writing conjunction and disjunction as infix operations, and the last two illustrate the use of Σ along with an abstraction to create the operations of existential and universal quantification that are familiar in the context of first-order languages. The brackets present in these abbreviations may again be omitted. In restoring an expression so written into an unambiguous form, use is made of the conventions that the brackets that surround $\sim F$ have the smallest scope, followed by those around $F \vee G$ and $F \wedge G$, and, finally, by those around $\exists x.F$. We shall sometimes use $\bar{x}$ as an abbreviation for a sequence of variables $x^1, \ldots, x^n$. In such cases, the expression $\exists \bar{x}.F$ is to be construed as a shorthand for $\exists x^1.\ \ldots\ \exists x^n.F$, and a similar interpretation is to be bestowed upon $\forall \bar{x}.F$.

Section 2: λ-Conversion and Substitution

Based on the intended interpretation of λ and of juxtaposition, there are certain logically meaningful operations defined on wffs. These operations are given by the rules of λ-conversion. The primary purpose of these, as rules of inference, is to specify circumstances under which two different wffs may be considered equal. Since λ-conversion plays an intrinsic role in the formulation of $\mathcal{T}^*$, we review these rules, and the associated notions of equality, in this section. λ-conversion also provides us with the means to describe a generalised notion of substitution, and we do this towards the end of this section.

The Calculus of λ-Conversion. We begin by defining a notion of substitution pertaining to wffs. Let x be a variable$_\alpha$, let G be a wff$_\alpha$ and let F be an arbitrary wff. We then write $S^x_G F$ to denote the result of replacing all the free occurrences of x in F by G. This operation may be made explicit by the following recursive definition:

(i) F is a variable or a constant. If F is x then $S^x_G F = G$. Otherwise $S^x_G F = F$.

(ii) F is of the form $\lambda y.C$. If y is x then $S^x_G F = F$. Otherwise $S^x_G F = \lambda y.S^x_G F$.

(iii) F is of the form $[C\,D]$. Then $S^x_G F = [(S^x_G C)\,(S^x_G D)]$.

In performing this operation of replacement, there is the danger that the free variables of G may become inadvertently bound. We use the term "G is *free for* x *in* F" to describe the situations in which the operation is logically correct; G is free for x in F just in case x does not occur free in a subformula $\lambda y.C$ of F where y is a free variable of G.

The rules of λ-conversion may now be described. These comprise the following operations on wffs; these operations are referred to individually as the rules of *α-conversion*, *β-reduction*, *β-expansion*, *η-reduction*, and *η-expansion*, respectively.

(1) Replacing a subformula $\lambda x.F$ by $\lambda y.S^x_y F$ provided y is free for x in F and $y \notin \mathcal{F}(F)$.

(2) Replacing a subformula $[\lambda x.F]\,G$ by $S^x_G F$ provided G is free for x in F.

(3) The converse of (2), *i.e.* if G results from a subformula F of A by (2), then replacing F by G in A.

(4) Replacing a subformula $\lambda x.[F\,x]$ by F provided x does not occur free in F.

(5) The converse of (4), *i.e.* replacing a subformula F of type $\alpha \to \beta$ by the subformula $\lambda x.[F\,x]$ where x is a variable$_\alpha$ that is not free in F.

The two β rules above are often collectively referred to as the β-conversion rules, and the two η rules are, likewise, referred to as the η-conversion rules. The rules of λ-conversion may be informally understood in the following manner. There is a certain sense in which the choice of name for the variable bound by an abstraction is unimportant, and the α-conversion rule makes this precise. Given the interpretation intended for the mechanisms for constructing wffs, we may consider a notion of function evaluation, and β-conversion provides a syntactic correlate for this notion; in this context, α-conversion enables bound

variables to be renamed to permit β-conversions in certain situations. Finally, η-conversion corresponds to a weak notion of extensionality for wffs*.

Based on the rules above, we define the following three relations between wffs.

2.2.1. Definition. *F λ-conv (β-conv, $\equiv$) G* just in case there is a sequence of applications of the λ-conversion (respectively α- and β-conversion, α-conversion) rules that transforms F into G.

It is apparent that these three relations hold only between wffs that have the same type. Furthermore, each of these are equivalence relations; each application of α-conversion is clearly invertible, and the β- and η-conversion rules each constitute invertible pairs. Thus, these relations define three notions of equality between wffs. In this thesis we shall use the strongest of these notions, *i.e.* we consider F and G equal just in case F λ-conv G. There are certain distinctions to be made between wffs by omitting the rules of η-conversion, but we feel that these distinctions are not important in our context. We note, however, that most of the later discussions go through with minor changes even if we choose this weaker notion of equality.

In our discussions concerning wffs, we shall find the notion of a normal form useful. This notion may be made precise as follows. Given a wff F, we refer to an occurrence of a wff of the form $[\lambda x.A]\,B$ in F as a *β-redex* of F; a β-redex is thus a subformula to which a β-reduction step may be applied. Similarly, we refer to an occurrence of a wff of the form $\lambda x.[A\,x]$ as an *η-redex* of F just in case x does not occur free in A. A wff is then a *λ-normal* (*β-normal*) *formula* in the case that there are no β- or η-redexes (respv. no β-redexes) in it. Evidently a β-normal formula is a wff that has the following form:

$$\lambda x^1.\ \ldots\ \lambda x^n.[A\,F^1\ \ldots\ F^m]$$

where A is a constant or variable, and, for $1 \leq i \leq m$, F^i also has the same form. We refer to the sequence $x^1, \ldots, x^n$ as the *binder*, to A as the *head* and to $F^1, \ldots, F^m$ as the *arguments* of the wff; in particular instances, the binder may be empty, and the wff may also have no arguments. We shall say that such a wff is *rigid* if its head, A, is either a constant or a variable that appears in the binder, and that it is *flexible* otherwise†. If $\bar{x}$ denotes the sequence $x^1, \ldots, x^n$, we shall sometimes find it convenient to use the abbreviation $\lambda\bar{x}.[A\,F^1\ \ldots\ F^2]$ for the above wff. We note, finally, that a wff of the above form is also a

* The notion of extensionality introduced by this rule is weaker than that provided by the axiom of extensionality in [10]: $[\forall x.[f\,x = g\,x]] \supset [f = g]$, where $F = G$ is defined as $\forall p.p\,F \supset p\,G$. This rule also has weaker connotations than the corresponding rule in the untyped version of the language. In the latter case the rule has the effect of asserting that everything is a function. In our context there are distinctions between various kinds of objects that are provided by types, and this distinction is unchangeable.

† This terminology is motivated by the fact that applying a substitution to a rigid formula in a certain sense leaves its head unchanged.

λ-normal formula if F^m is not identical to x^n and, in addition, each of the F^is are also in this form.

The following proposition assures us that there is a β-normal formula corresponding to each wff.

2.2.2 (Normal-Form Theorem). *For each wff there is a β-normal formula that may be obtained from it by a sequence of β-reductions and α-conversions.*

Proofs of this proposition may be found in [1] and in [13]. With regard to η-reductions we observe that they reduce the number of symbols in a wff and they do not introduce any new β-redexes or η-redexes into the wff. Any wff may therefore be transformed into a λ-normal formula by first converting it into a β-normal formula and then applying a sequence of η-reductions to this formula.

We define a λ-normal form of a wff F to be a λ-normal formula G such that F λ-conv G. Our earlier discussion assures us that such forms exist for every wff. While such forms are not unique, they are nevertheless closely related as the following proposition states. This proposition was originally proved for a system of λ-conversion pertaining to a language without type symbols. A proof for that system of conversion may be found in [6, pp 59-67]. The results apply to the system under consideration as well, as can be verified by an examination of the mentioned proof.

2.2.3 (Church-Rosser Theorem). *If F, G are λ-normal formulas such that F λ-conv G, then $F \equiv G$. In other words, a λ-normal form of a wff is unique upto a renaming of bound variables.*

For the most part we shall be satisfied with any one of these normal forms corresponding to a wff F, and we shall write $\lambda norm(F)$ to denote such a form. In certain situations we shall need to talk about a unique normal form and, in such cases, we shall use $\rho(F)$ to designate what we shall call the *principal normal form* of F; *i.e.* ρ is a function from wffs to λ-normal formulas. There are several schemes that may be used to pick a representative of the ≡-equivalence classes of λ-normal formulas and the one implicitly assumed here is that of [1]. (Under this scheme we first assume an ordering of the variables of each type. We then define a ρ-wff to be a λ-normal formula in which, in each subformula of the form $\lambda x_\alpha.G$, x_α is the first variable that is distinct from all the other free variables of type α in G. It is an easy observation that there is a unique ρ-wff corresponding to each wff F and it is this that we denote by $\rho(F)$). For our purposes the particular choice of scheme is unimportant, and the only requirement that we place on ρ is that if F λ-conv G then $\rho(F) = \rho(G)$.

2.2.4. Example. Let `nil` be a parameter (of type $(list\ int)$) and let the other symbols below be as in Example 2.1.2. Then, the wff

```
[λl.[cons 1 [cons 2 l]]] [cons 3 nil]
```

has the λ-normal form

```
[cons 1 [cons 2 [cons 3 nil]]].
```

The binder of the last wff is empty, its head is `cons` and its arguments are `1` and `[cons 2 [cons 3 nil]]`. This wff is also a rigid formula.

The notion of normal forms is useful for two reasons. First, the existence of a normal form and of a mechanism to convert any wff into a normal form provides a means for determining whether two wffs are equal. Second, normal forms provide a means for discussing the properties of wffs in terms of a representative for each of the equivalence classes that has a convenient structure. As a particular instance, we shall have use for the structure of λ-normal formulas of type o that is described below.

2.2.5. Definition. A wff of type o is an *atom* (is *atomic*) if its leftmost symbol that is not a bracket is either a variable or a parameter.

A λ-normal formula of type o, then, has one of the following forms:

(i) it is $\top$,

(ii) it is an atom,

(iii) it is $\sim F$, where F is a λ-normal formula of type o,

(iv) it is $F \vee G$ or $F \wedge G$, where F and G are λ-normal formulas of type o, or

(v) it is ΣP, where P is a λ-normal formula.

Substitutions. λ-conversion provides us with the facility for defining one other notion pertaining to wffs, namely that of substituting wffs for some of the free variables in a given wff. This notion is made precise as follows. First, we define a *substitution* to be a finite set of pairs of the form

$$\{\langle x^i, F^i\rangle \mid 1 \leq i \leq n\},$$

where, for $1 \leq i \leq n$, each x^i is a distinct variable and F^i is a wff in principal normal form* of the same type as x^i; we refer to such a substitution as a *substitution for* x^i, and we say that its *range* is $\{F^1, \ldots, F^n\}$. Now, we view a substitution as a type preserving mapping on variables that is the identity almost everywhere. Thus, if θ is a substitution and y is a variable, the result of *applying* θ *to* y is defined by the following equation:

$$\theta(y) = \begin{cases} F, & \text{if } \langle y, F\rangle \in \theta; \\ y, & \text{otherwise.} \end{cases}$$

Finally, we extend this mapping to the class of all wffs in a manner that is consistent with the above view: If

* While it is not really necessary to place this restriction on F^i, we do it mainly to maintain a uniformity between the application of a substitution to a variable and to a wff.

$$\theta = \{\langle x^i, F^i\rangle \mid 1 \leq i \leq n\}$$

and G is an arbitrary wff, then

$$\theta(G) = \rho([\lambda x^1. \ \ldots \ \lambda x^n.G]\, F^1 \ \ldots \ F^n).$$

It is easily seen that this definition is independent of the order in which we take the pairs from θ. Furthermore, given our notion of equality between wffs, the application of a substitution to a wff G is evidently a formalisation of the idea of replacing the free occurrences of $x^1, \ldots, x^n$ in G simultaneously by the wffs $F^1, \ldots, F^n$.

2.2.6. Example.

(i) Let F be the wff `[cons x [cons 2 l]]`, where `x` and `l` are variables and the other symbols are as in Example 2.1.2. If θ is the substitution $\{\langle \mathtt{x}, \mathtt{1}\rangle, \langle \mathtt{l}, [\mathtt{cons\ x\ nil}]\rangle\}$, then $\theta(F) =$`[cons 1 [cons 2 [cons x nil]]]`. Note that the order in which we take the pairs from θ is immaterial. Also note that the head of F remains unchanged under the application of θ.

(ii) Given a wff$_o$ F let us say that a predicate variable x occurs *extensionally* in F if

(a) x is the leftmost non-bracket symbol in F, or

(b) F is $\sim F^1$ and x occurs extensionally in F^1, or

(c) F is $F^1 \vee F^2$ or $F^1 \wedge F^2$ and x occurs extensionally in either F^1 or F^2.

Assume that P is a parameter of type $\iota \to \iota \to o$, that A and B are parameters of type $\iota \to o$, and that y is a variable of type $\iota \to \iota \to o$. Then y occurs extensionally in the wff $F = [P\ A] \supset [y\ A]$. Let θ be the substitution

$$\{\langle y, \lambda z.[[P\ z] \wedge \forall x.[[z\ x] \supset [B\ x]]]\rangle\}.$$

Then

$$\theta(F) \equiv [P\ A] \supset [[P\ A] \wedge \forall x.[[A\ x] \supset [B\ x]]].$$

The point to note in this example is that applying a substitution to a wff$_o$ in which a predicate variable occurs extensionally has the potential of producing a wff$_o$ that has a different propositional and quantificational structure.

There are certain notational conventions and notions pertaining to substitutions that we shall find useful in our later discussions, and we describe these below.

(i) Given a set of variables $\mathcal{V}$, we shall often need to consider the *restriction* of a substitution θ to $\mathcal{V}$. This notion is denoted by the notation $\theta \uparrow \mathcal{V}$, and is defined in the usual manner, *i.e.*

$$\theta \uparrow \mathcal{V} = \{\langle x, F\rangle \mid \langle x, F\rangle \in \theta \text{ and } x \in \mathcal{V}\}.$$

From this definition, it is easy to show that $\theta(G) = (\theta \uparrow \mathcal{F}(G))(G)$.

(ii) We shall find use for the notion of the *composition* of two arbitrary substitutions θ_1 and θ_2. This is a substitution that is denoted by $\theta_1 \circ \theta_2$ and is precisely the composition of θ_1 and θ_2 when these are viewed as mappings: $\theta_1 \circ \theta_2(G) = \theta_1(\theta_2(G))$. Alternatively, this may be described as a finite set of pairs given in the following manner:

$$\theta_1 \circ \theta_2 = \{\langle x, \theta_1(\theta_2(x))\rangle \mid x \text{ is a variable and } \theta_1(\theta_2(x)) \neq x\}.$$

(iii) We shall also need to compare substitutions. For this purpose, we define first a notion of equality between substitutions relative to a set of variables as follows:

$$\theta_1 =_\nu \theta_2 \text{ if and only if } \theta_1 \uparrow \mathcal{V} = \theta_2 \uparrow \mathcal{V}.$$

We then say that a substitution θ_1 is *less general than* a substitution θ_2 relative to $\mathcal{V}$ just in case there is a substitution σ such that $\theta_1 =_\nu \sigma \circ \theta_2$. We denote this relation by writing $\theta_1 \preceq_\nu \theta_2$.

Finally, we shall sometimes talk of the result of applying a substitution to sets of wffs and to sets of pairs of wffs. In the first case, we mean the set that results from applying the substitution to each wff in the set, and, in the latter case, we mean the set of pairs that results from the application of the substitution to each element in each pair.

Section 3: The Formal System

Let $p, q, r \in \mathcal{V}ar_o$, $x \in \mathcal{V}ar_\alpha$ and for each α let $f \in \mathcal{V}ar_{\alpha\to o}$. The wffs$_o$ in the following infinite list then constitute the formal axioms of the system*.

1. $\top$
2. $p \wedge q \supset p$
3. $p \wedge q \supset q \wedge p$
4. $p \supset (q \supset (p \wedge q))$
5. $p \vee p \supset p$
6. $p \supset p \vee q$
7. $p \vee q \supset q \vee p$
8. $p \supset q \supset [r \vee p \supset r \vee q]$

9^α. $f\, x \supset \exists x.f\, x$

* 9^α is to be interpreted as a schema; it determines wffs$_o$ depending on the particular choice for α.

The following operations now comprise the rules of inference of the system. In these statements, upper-case letters constitute syntactic variables for wffs of the indicated types.

(I) (λ-*Conversion*) To infer G from F if G results from F by a single application of one of the rules of α-, β-, or η-conversion to F.

(II) (*Substitution*) To infer $[F\,A]$ from $[F\,x]$, if x is a variable$_\alpha$ that does not occur free in F and A is a wff$_\alpha$.

(III) (*Modus Ponens*) To infer G from F and $F \supset G$.

(IV) (*Existential Rule*) To infer $[\exists x.F\,x] \supset G$ from $[F\,x] \supset G$, provided x does not occur free in F or G.

The *theorems* of the system are those wffs that may be obtained from the formal axioms by a succession of applications of the rules of inference; we write $\vdash_{\mathcal{T}^*} F$ to indicate that F is a theorem of the system. A *proof* of a theorem, F, is a list, $F^1, \ldots, F^n$, of wffs$_o$ such that F_n is F and for $1 \leq i \leq n$, F_i is either an axiom or derived from one or two of the previous wffs in the list by a rule of inference.

The system described by the above axioms and rules of inference is closely related to the system in [1]. The relationship may be explicated by the statement that we have formulated the logic using existential quantification as the primitive notion, we have included η-conversion as a rule of inference, and we have, in addition, chosen as primitives certain symbols that may be introduced via definitions in the system of [1]. Indeed, the replacement of each occurrence of Σ by $\lambda b.{\sim}\Pi\,\lambda x.{\sim}[b\,x]$, of each occurrence of $\top$ by $p \supset p$ and of each occurrence of $\wedge$ by $\lambda p.\lambda q.{\sim}[{\sim}p \vee {\sim}q]$ in a theorem of this system produces a theorem of the latter system augmented with the rule of η-conversion. Conversely, a theorem of the latter system may be rendered into a theorem of this system by the replacement of each occurrence of Π by $\lambda b.{\sim}\Sigma\,\lambda x.{\sim}[b\,x]$.

In subsequent discussions, we shall find useful the notion of a proof of a wff$_o$ G on the assumption of the wffs$_o$ $F^1, \ldots, F^n$. This notion corresponds to a list of wffs$_o$, $G^1, \ldots, G^m$, such that G^m is G and, for $1 \leq i \leq m$, each G^i is either an axiom or is in the list $F^1, \ldots, F^n$ or is obtainable from preceding wffs by an application of a rule of inference subject to the condition that no variable that occurs free in any of the assumption wffs$_o$ is acted upon by the rule of substitution or the existential rule. The fact that there is a proof for G on the assumption of $F^1, \ldots, F^n$ is expressed by the use of the notation

$$F^1, \ldots, F^n \vdash_{\mathcal{T}^*} G.$$

The proof of the following proposition may be obtained by an adaptation of its proof in [10] for the related system.

2.3.1 (Deduction Theorem). *For $n \geq 1$ let $F^1, \ldots, F^n$, and G be wffs$_o$. If $F^1, \ldots, F^n \vdash_{\mathcal{T}^*} G$ then $F^1, \ldots, F^{n-1} \vdash_{\mathcal{T}^*} F^n \supset G$.*

2.3: The Formal System

Our presentation of a Hilbert-style axiomatisation is motivated by a desire to place our endeavour within the framework of a well-studied higher-order logic. In discussing certain proof-theoretic properties of this system, however, we find it more convenient to employ a Gentzen-style formulation of the same system. We describe this formulation in the next section. In establishing a correspondence between the two formulations of the logic, we shall make use of the fact, already implicit, that the tautologous wffs are theorems of $\mathcal{T}^*$; a verification of this fact may be supplied by the technique used, for example, in ([24] pp 45 – 48). We shall also find useful the following notion of an *abstract derivability property* that is based on the notion in [26] for the related system.

2.3.2. Definition. A property Λ of finite sets of wffs_o is an abstract derivability property if, for all finite sets $\mathcal{S}$ of wffs_o, the following holds:

ADP1 If $\top \in \mathcal{S}$, then $\Lambda(\mathcal{S})$.

ADP2 If there is an atomic formula F such that $F \in \mathcal{S}$ and $\sim F \in \mathcal{S}$, then $\Lambda(\mathcal{S})$.

ADP3 If $\Lambda(\mathcal{S} \cup \{\rho(F)\})$, then $\Lambda(\mathcal{S} \cup \{F\})$.

ADP4 $\Lambda(\mathcal{S} \cup \{F\})$ if and only if $\Lambda(\mathcal{S} \cup \{\sim\sim F\})$.

ADP5 If $\Lambda(\mathcal{S} \cup \{F, G\})$, then $\Lambda(\mathcal{S} \cup \{F \vee G\})$.

ADP6 If $\Lambda(\mathcal{S} \cup \{\sim F\})$ and $\Lambda(\mathcal{S} \cup \{\sim G\})$, then $\Lambda(\mathcal{S} \cup \{\sim[F \vee G]\})$.

ADP7 If $\Lambda(\mathcal{S} \cup \{F\})$ and $\Lambda(\mathcal{S} \cup \{G\})$, then $\Lambda(\mathcal{S} \cup \{F \wedge G)\}$.

ADP8 If $\Lambda(\mathcal{S} \cup \{\sim F, \sim G\})$, then $\Lambda(\mathcal{S} \cup \{\sim[F \wedge G]\})$.

ADP9 If $\Lambda(\mathcal{S} \cup \{\sim P\, y\})$ where y is a parameter (variable) that does not occur (occur free) in P or in any wff in $\mathcal{S}$, then $\Lambda(\mathcal{S} \cup \{\sim \Sigma\, P\})$.

ADP10 If $\Lambda(\mathcal{S} \cup \{\Sigma\, P, P\, C\})$ for some wff C, then $\Lambda(\mathcal{S} \cup \{\Sigma\, P\})$.

Let $\vee\mathcal{S}$ denote the disjunction of the members of any finite set $\mathcal{S}$ of wffs_o taken in an arbitrary order. Then the use that we shall find for the above notion is based on the following proposition.

2.3.3 (Relative Completeness of Abstract Derivability Properties). *Let $\mathcal{S}$ be a finite set of wffs_o and let Λ be an abstract derivability property. If $\vdash_{\mathcal{T}^*} \vee\mathcal{S}$ then $\Lambda(\mathcal{S})$.*

The proof of this proposition may be obtained from a modified version of the abstract consistency property of [1], which is a generalisation to $\mathcal{T}$ of the analytic consistency property described in [39]. For the sake of the sceptical reader, we present this property and then outline the changes necessary to the arguments in [1] in Appendix A.

Before concluding this section, we note the similarity between well known formulations of the first-order functional calculus and the higher-order logic that we study in this thesis — as is manifest in our presentation of the formal system. From one perspective the logic that we study here is apparently a many-sorted version of first-order logic. The key

differences, though, are the richer syntax of wffs, and the presence of λ-conversion as a rule of inference.

Section 4: A Gentzen-Style System

In this section we present a sequential calculus, $LKH\Sigma$, for our higher-order logic. $LKH\Sigma$ constitutes a higher-order extension to a system derived from the logistic classical calculus LK of [15]; in this regard the main modification is an incorporation of the result pertaining to cut-elimination for LK in the formulation of the calculus. We use $LKH\Sigma$ in our arguments in Chapter 3, and our formulation of the calculus is designed so as to provide a facility in the discussions therein. For the sake of completeness, in our presentation of $LKH\Sigma$ we summarise some of the notions in [15] that are relevant to sequent systems. In the next section we shall show that the sequent calculus that we present below is actually equivalent to $\mathcal{T}^*$.

A central notion in a Gentzen-style system is that of a *sequent*. A sequent is an expression of the form

$$F^1, \ldots, F^n \ \longrightarrow \ G^1, \ldots, G^m$$

where $n \geq 0$, $m \geq 0$, and, for $1 \leq i \leq n$ and $1 \leq j \leq m$, F^i and G^j are terms from some predefined language. The terms $F^1, \ldots, F^n$ form what is known as the *antecedent* of the sequent, and the terms $G^1, \ldots, G^m$ form its *succedent*. From the description above, it is clear that both the antecedent and the succedent may be empty in any given sequent.

A sequent calculus is characterised by its *inference figures*, which are arrangements of sequents of the following form:

$$\frac{\Gamma_1 \ \longrightarrow \ \Delta_1 \quad \ldots \quad \Gamma_n \ \longrightarrow \ \Delta_n}{\Gamma \ \longrightarrow \ \Delta} \qquad n \geq 1$$

where the letters Γ_i and Δ_i denote finite sequences of terms. Given an inference figure of the above sort, the sequents $\Gamma_i \ \longrightarrow \ \Delta_i$, for $1 \leq i \leq n$, are called the *upper* sequents and the sequent $\Gamma \ \longrightarrow \ \Delta$ is called the *lower* sequent of the figure.

We are interested in "tree-like" arrangements of sequents which combine to form inference figures in the following way:

(i) Each sequent, with the exception of exactly one that is called the *end* sequent, is the upper sequent of exactly one inference figure.

(ii) Each sequent is the lower sequent of at most one inference figure; those sequents that are not the lower sequents of any inference figure are called the *leaf* sequents of the arrangement.

(iii) There are no cycles in the arrangement in that there is no sequence of sequents, where each member is an upper sequent of an inference figure whose lower sequent is the preceding member, whose last member is again succeeded by the first member.

Each calculus is also characterised by a set of sequents designated as *initial* sequents. A *proof figure* in a given calculus is then an arrangement of the above sort in which each leaf sequent is also an initial sequent. If the end sequent of such a figure is $\Gamma \longrightarrow \Delta$, then this figure is also referred to as a proof figure for $\Gamma \longrightarrow \Delta$. A *path* in a proof figure is a sequence of sequents, the first member being the end sequent, and every other member being the upper sequent of an inference figure whose lower sequent is the preceding member*. Each sequent in a proof figure is evidently on a unique path, and we refer to the number of elements that precede it on this path as the *distance* of the sequent from the end sequent. The *height* of the proof figure is the length of the longest path in the figure.

With these preliminary remarks, we turn to the specific characterisation of the calculus $LKH\Sigma$. The terms in the sequents of $LKH\Sigma$ are the wffs$_o$ of $\mathcal{T}^*$. The initial sequents of this calculus are the sequents that result from the schemata

$$\Gamma_1, A, \Gamma_2 \longrightarrow \Delta_1, A, \Delta_2 \qquad\qquad \Gamma \longrightarrow \Delta_1, \top, \Delta_2$$

by replacing Γ and, for $i = 1, 2$, Γ_i and Δ_i by finite sequences of wffs$_o$ and A by an atomic wff$_o$. The inference figures are those that result from the schemata in Figure 2.4.1 by a substitution of the following kind: Replace the Γs and Δs by finite sequences of wffs$_o$. Replace F and G by wffs$_o$. In the schemata designated by λ, replace F' by a wff$_o$ resulting from the wff that replaces F in the lower sequent of the schema by a sequence of λ-conversions. Finally, in the schemata designated by $\Sigma-$IA and $\Sigma-$IS, replace P by a wff of type $\alpha \to o$, C by a wff$_\alpha$ and y by a parameter or variable of type α for some choice of α; in the last case, we also stipulate that the parameter or variable that replaces y should not occur free in any of the wffs that are substituted in the lower sequent.

We distinguish the figures that result from the first two schemata as the structural figures, and those that result from the remaining as operational figures. In each of these figures, we designate a wff in the lower sequent as the *principal* formula of the figure: In the structural figures, this is the wff that replaces the symbol F in the corresponding schema. In the operational figures, this is the wff substituted for the expression containing the logical constant in the corresponding schema. We also note that two upper sequents appear in some of the operational figures, and we distinguish between these by referring to them as as the left and right upper sequents of the figure.

---

* Since there may be distinct occurrences of the same sequent in a given proof figure, we need to make the qualification that what we actually refer to is the sequent as characterised by its *occurrence* in the figure. We implicitly assume this qualification wherever it is necessary.

Contraction in the antecedent

$$\frac{\Gamma_1, F, F, \Gamma_2 \longrightarrow \Delta}{\Gamma_1, F, \Gamma_2 \longrightarrow \Delta}$$

Contraction in the succedent

$$\frac{\Gamma \longrightarrow \Delta_1, F, F, \Delta_2}{\Gamma \longrightarrow \Delta_1, F, \Delta_2}$$

λ in the antecedent

$$\frac{\Gamma_1, F', \Gamma_2 \longrightarrow \Delta}{\Gamma_1, F, \Gamma_2 \longrightarrow \Delta}$$

λ in the succedent

$$\frac{\Gamma \longrightarrow \Delta_1, F', \Delta_2}{\Gamma \longrightarrow \Delta_1, F, \Delta_2}$$

$$\frac{\Gamma_1, \Gamma_2 \longrightarrow \Delta_1, F, \Delta_2}{\Gamma_1, \sim F, \Gamma_2 \longrightarrow \Delta_1, \Delta_2} \sim\text{−IA} \qquad \frac{\Gamma_1, F, \Gamma_2 \longrightarrow \Delta_1, \Delta_2}{\Gamma_1, \Gamma_2 \longrightarrow \Delta_1, \sim F, \Delta_2} \sim\text{−IS}$$

$$\frac{\Gamma_1, F, \Gamma_2 \longrightarrow \Delta \qquad \Gamma_1, G, \Gamma_2 \longrightarrow \Delta}{\Gamma_1, F \vee G, \Gamma_2 \longrightarrow \Delta} \vee\text{−IA}$$

$$\frac{\Gamma_1, F, G, \Gamma_2 \longrightarrow \Delta}{\Gamma_1, F \wedge G, \Gamma_2 \longrightarrow \Delta} \wedge\text{−IA} \qquad \frac{\Gamma \longrightarrow \Delta_1, F, G, \Delta_2}{\Gamma \longrightarrow \Delta_1, F \vee G, \Delta_2} \vee\text{−IS}$$

$$\frac{\Gamma \longrightarrow \Delta_1, F, \Delta_2 \qquad \Gamma \longrightarrow \Delta_1, G, \Delta_2}{\Gamma \longrightarrow \Delta_1, F \wedge G, \Delta_2} \wedge\text{−IS}$$

$$\frac{\Gamma_1, \rho(P\, y), \Gamma_2 \longrightarrow \Delta}{\Gamma_1, \Sigma P, \Gamma_2 \longrightarrow \Delta} \Sigma\text{−IA} \qquad \frac{\Gamma \longrightarrow \Delta_1, \rho(P\, C), \Delta_2}{\Gamma \longrightarrow \Delta_1, \Sigma\, P, \Delta_2} \Sigma\text{−IS}$$

Figure 2.4.1: The $LKH\Sigma$ Inference Figure Schemata

In the remainder of this section we observe some simple properties of $LKH\Sigma$ proof figures that we shall find useful in later discussions.

2.4.1 Lemma. *Let $\Gamma_1, \Gamma_2 = \Gamma_1', \Gamma_2'$. Then, there is a proof figure of height h for a sequent $\Gamma_1, F^1, \ldots, F^n, \Gamma_2 \longrightarrow \Delta$ if and only if there is a proof figure of height h for the sequent $\Gamma_1', F^1, \ldots, F^n, \Gamma_2' \longrightarrow \Delta$. A corresponding relationship holds for the succedent.*

Proof. By an induction on the height of the purported proof figure for either sequent. ∎

2.4: A Gentzen-Style System

We shall often find it convenient to consider proof figures in which the appearances of the inference figure λ are prior to the other inference figures. This is the reason for our incorporation of λ-conversion into the schemata $\Sigma-$IA and $\Sigma-$IS. The following lemma guarantees that we may restrict our attention to such proof figures.

2.4.2 Lemma. *Let $\Gamma' \longrightarrow \Delta'$ result from $\Gamma \longrightarrow \Delta$ by replacing each wff F in the latter sequent by its principal normal form. Then, the second sequent has a proof figure of height h only if the first has one of height $\leq h$ in which the inference figure λ does not appear.*

Proof. By an easy induction on the height of the proof figure for the second sequent. ∎

2.4.3 Lemma. *A sequent of the form $\Gamma_1, \sim F, \Gamma_2 \longrightarrow \Delta_1, \Delta_2$ has a proof figure of height h only if the sequent $\Gamma_1, \Gamma_2 \longrightarrow \Delta_1, F, \Delta_2$ has a proof figure of height $\leq h$. A similar relationship holds between the sequents $\Gamma_1, \Gamma_2 \longrightarrow \Delta_1, \sim F, \Delta_2$ and $\Gamma_1, F, \Gamma_2 \longrightarrow \Delta_1, \Delta_2$.*

Proof. We only show the first part of the lemma; the proof of the second part is similar. For this part, we claim that if F^1 λ-conv $\sim F$ and if there is a proof figure of height h for the sequent $\Gamma_1, F^1, \Gamma_2 \longrightarrow \Delta_1, \Delta_2$, then there is a proof figure of height $\leq h$ for the sequent $\Gamma_1, \Gamma_2 \longrightarrow \Delta_1, F, \Delta_2$. It is apparent that the lemma follows from this claim.

We show this claim by an induction on the height of the proof figure for the first sequent. In the case that this height is 1, the first sequent is evidently an initial sequent. But then, so also is the second sequent. Let us, therefore, assume that the height is $h+1$. Consider now the inference figure for which the lower sequent is the end sequent. Evidently the upper sequent(s) of this inference figure have proof figures of height $\leq h$. In the cases when the principal formula of the inference figure under consideration is not F^1, it is easy to see how a proof figure of height $\leq h$ for the second sequent may be obtained by a use of the inductive hypothesis and the same inference figure schemata.

Thus, the only cases that require further consideration are those in which the inference figure is a *Contraction in the antecedent*, a λ *in the antecedent* or a $\sim-$IA, the principal formula in each case being F^1. In the first case, we observe that $\Gamma_1, F^1, F^1, \Gamma_2 \longrightarrow \Delta_1, \Delta_2$ has a proof figure of height h. From the hypothesis we see that $\Gamma_1, F^1, \Gamma_2 \longrightarrow \Delta_1, F, \Delta_2$ has a proof figure of height $\leq h$. But then, the claim follows from another use of the hypothesis. In the second case we see that there is a proof figure of height h for $\Gamma_1, F^2, \Gamma_2 \longrightarrow \Delta_1, \Delta_2$ where F^2 λ-conv $\sim F$, and so the claim follows directly from the hypothesis. In the last case, we see that the upper sequent is $\Gamma_1, \Gamma_2 \longrightarrow \Delta_1', F', \Delta_2'$, where $\Delta_1', \Delta_2' = \Delta_1, \Delta_2$ and F' λ-conv F. But then by Lemma 2.4.1 we see that $\Gamma_1, \Gamma_2 \longrightarrow \Delta_1, F', \Delta_2$ has a proof figure of height h. By using an inference figure λ *in the succedent* in conjunction with this proof figure, we obtain a proof figure of height $h+1$ for the sequent $\Gamma_1, \Gamma_2 \longrightarrow \Delta_1, F, \Delta_2$.
∎

We now introduce into $LKH\Sigma$ two new inference figure schemata that we shall find useful in Chapter 3.

$$\frac{\Gamma_1, \Gamma_2 \longrightarrow \Delta_1, F, \Delta_2 \qquad \Gamma_1, G, \Gamma_2 \longrightarrow \Delta_1, \Delta_2}{\Gamma_1, \sim F \vee G, \Gamma_2 \longrightarrow \Delta_1, \Delta_2} \text{ imply} - \text{IA}$$

$$\frac{\Gamma_1, \rho([\lambda x.P]\, C), \Gamma_2 \longrightarrow \Delta}{\Gamma_1, \sim\Sigma\, \lambda x.\sim P, \Gamma_2 \longrightarrow \Delta} \text{ all} - \text{IA}$$

These schemata are "derived" in the following sense: their effect may be obtained by the use of the other schemata. (The effect of the first may be obtained by a use of $\vee-$IA whose left upper sequent is then the lower sequent of a $\sim-$IA. The effect of the second may be obtained by a combination of figures $\sim-$IA, $\Sigma-$IS and $\sim-$IS). The primary use that we make of these figures is in describing proof figures that have a "normal-form." The description of this normal form is contained in the following lemma.

2.4.4 Lemma. *Let each wff in Γ and Δ be in principal normal form. Then there is a proof figure of height h for $\Gamma \longrightarrow \Delta$ only if there is a proof figure of height $\leq h$ for the same sequent in which*

*(i) the principal formula of a $\vee-$*IA *inference figure is not a formula of the form $\sim F \vee G$,*

*(ii) the principal formula of a $\sim-$*IA *inference figure is not a formula of the form $\sim\Sigma\, \lambda x.\sim F$, and*

(iii) the inference figures λ in the antecedent and λ in the succedent do not appear.

Proof. By an induction on the height of the given proof figure. The only cases we need to consider are those in which the end sequent is the lower sequent of an inference figure that violates (i), (ii) or (iii). In the first case we have

$$\frac{\Gamma_1, \sim F, \Gamma_2 \longrightarrow \Delta \qquad \Gamma_1, G, \Gamma_2 \longrightarrow \Delta}{\Gamma_1, \sim F \vee G, \Gamma_2 \longrightarrow \Delta}$$

and the sequents $\Gamma_1, \sim F, \Gamma_2 \longrightarrow \Delta$ and $\Gamma_1, G, \Gamma_2 \longrightarrow \Delta$ each have proof figures of height $< h$. From Lemma 2.4.3 it follows that $\Gamma_1, \Gamma_2 \longrightarrow \Delta, F$ has a proof figure of height $< h$. We observe, now, that each wff H in the sequents $\Gamma_1, \Gamma_2 \longrightarrow \Delta, F$ and $\Gamma_1, G, \Gamma_2 \longrightarrow \Delta$ is in principal normal form. From our hypothesis it follows, then, that these sequents each have proof figures of height $< h$ that satisfy (i), (ii), and (iii). But then we use these and an inference figure imply $-$ IA to obtain a proof figure of the requisite sort for $\Gamma_1, \sim F \vee G, \Gamma_2 \longrightarrow \Delta$. A similar argument may be provided for the second case. In the third case we use Lemma 2.4.2. ∎

Section 5: The Equivalence of $LKH\Sigma$ and $\mathcal{T}^*$

We now consider the relationship between the calculus $LKH\Sigma$ and $\mathcal{T}^*$. Our objective in this section is to show that these two systems are equivalent in the sense obtained from [15]. Readers familiar with this notion and to whom the equivalence of the two systems is already apparent may desire to proceed directly to the next section.

Let Θ be a sequence of wffs$_o$ $F^1, \ldots, F^{n-1}, F^n$. We shall, then, use the notation $\wedge\Theta$ for the wff $[F^1 \wedge \ldots \wedge F^{n-1}] \wedge F^n$ and the notation $\vee\Theta$ for the wff $[F^1 \vee \ldots \vee F^{n-1}] \vee F^n$. We now define an association between wffs$_o$ and sequents in the following manner. Let $p \in \mathcal{V}ar_o$ be a designated propositional variable. Given a sequent $\Gamma \longrightarrow \Delta$, we then say that its *associated formula* is $\wedge\Gamma \supset \vee\Delta$ if neither the antecedent nor the succedent is empty, $\vee\Delta$ if only the antecedent is empty, $\wedge\Gamma \supset p \wedge \sim p$ if only the succedent is empty, and $p \wedge \sim p$ if both the antecedent and the succedent are empty. The notion of equivalence that we are interested in then is the following: We consider the calculus $LKH\Sigma$ to be equivalent to $\mathcal{T}^*$ if the existence of an $LKH\Sigma$ proof figure for a sequent implies that its associated formula is a theorem of $\mathcal{T}^*$ and if the converse is also true.

We show in this section that the two systems are, indeed, equivalent in the sense made precise above. The proof of the first half of this equivalence is provided by outlining a method for transforming an $LKH\Sigma$ proof figure into a proof in $\mathcal{T}^*$. In the proof of the second half we make use of the abstract derivability property introduced in Section 2.3.

2.5.1 Lemma. *There is an $LKH\Sigma$ proof figure for a sequent only if there is a proof in $\mathcal{T}^*$ for its associated formula.*

Proof. The proof of this lemma is based on the methods in [15]. We shall present an auxiliary calculus that has the following virtues: We shall see that an $LKH\Sigma$ proof figure for a given sequent can be transformed into a proof figure in the auxiliary calculus for a sequent that has the same associated formula, ensuring, in so doing, that no sequent in the latter proof figure has an empty succedent. For the auxiliary calculus we shall readily see that (i) proofs in $\mathcal{T}^*$ exist for the associated formulas of the initial sequents, and (ii) if each sequent in an inference figure has a nonempty succedent and if proofs in $\mathcal{T}^*$ exist for the associated formula(s) of the upper sequent(s) of the figure, then a proof in $\mathcal{T}^*$ exists for the associated formula of the lower sequent of the figure. It is clear that the lemma follows from these observations.

The auxiliary calculus is obtained from $LKH\Sigma$ in the following manner. We replace the initial sequents of $LKH\Sigma$ by those obtained from the following schemata by substituting atomic formulas for A and arbitrary wffs$_o$ for F:

$$A \longrightarrow A \qquad\qquad \longrightarrow \top \qquad\qquad F, \sim F \longrightarrow A$$

We restrict the the inference figures obtained from the schemata in Figure 2.4.1 as follows: Only the empty sequence may be substituted for Γ_2 and Δ_1 in all the schemata in which

these two symbols appear. In the schemata λ *in the succedent*, *Contraction in the succedent*, $\vee-$IS, $\wedge-$IS, $\sim-$IS, and $\Sigma-$IS the empty sequence must be substituted for Δ_2 as well. We drop the schemata all $-$ IA and imply $-$ IA; from our previous discussions, it is clear that these may be transformed into other $LKH\Sigma$ figures. Finally, we add the five new inference figure schemata listed below:

Thinning in the antecedent

$$\frac{\Gamma \longrightarrow \Delta}{\Gamma, F \longrightarrow \Delta}$$

Interchange in the antecedent

$$\frac{\Gamma, G, F \longrightarrow \Delta}{\Gamma, F, G \longrightarrow \Delta}$$

$$\frac{\Gamma \longrightarrow F \qquad F \longrightarrow \Delta}{\Gamma \longrightarrow \Delta} \text{ Cut}$$

$$\frac{\Gamma, \sim F \longrightarrow \Delta}{\Gamma \longrightarrow F, \Delta} \sim-\text{EA} \qquad \frac{\Gamma \longrightarrow \sim F, \Delta}{\Gamma, F \longrightarrow \Delta} \sim-\text{ES}$$

We observe, now, that the effect of each $LKH\Sigma$ inference figure may be obtained by a sequence of inference figures in the auxiliary calculus. Consider, for example, the following instance of the schema *Contraction in the antecedent* for $LKH\Sigma$:

$$\frac{\Gamma_1, F, F, \Gamma_2 \longrightarrow \Delta}{\Gamma_1, F, \Gamma_2 \longrightarrow \Delta}$$

Denoting by $\sim\Gamma$ the sequence of wffs that is obtained by negating each wff in Γ, we see that the above $LKH\Sigma$ inference figure may be transformed into the following sequence of inference figures of the auxiliary calculus:

$$\cfrac{\cfrac{\cfrac{\Gamma_1, F, F, \Gamma_2 \longrightarrow \Delta}{\Gamma_1, F, F \longrightarrow \sim\Gamma_2, \Delta}\ \text{sequence of } \sim-\text{IS}}{\Gamma_1, F \longrightarrow \sim\Gamma_2, \Delta}\ \text{Contraction}}{\Gamma_1, F, \Gamma_2 \longrightarrow \Delta}\ \text{sequence of } \sim-\text{ES}$$

A similar transformation may be effected on the figures λ *in the antecedent*, $\vee-$IA, $\wedge-$IA and $\Sigma-$IA of $LKH\Sigma$. In transforming the remaining inference figures we may need the additional figures $\sim-$EA and *Interchange*, but the scheme is clear. In a similar fashion, we add a sequence of figures "above" the $LKH\Sigma$ initial sequents $\Gamma_1, A, \Gamma_2 \longrightarrow \Delta_1, A, \Delta_2$ and

$\Gamma \longrightarrow \Delta_1, \top, \Delta_2$ to obtain the initial sequents $A \longrightarrow A$ and $\longrightarrow \top$ of the auxiliary calculus; this time we may need the figure *Thinning in the antecedent* as well.

From these observations, it is clear that any $LKH\Sigma$ proof figure may be transformed into a proof figure for the same end sequent in the auxiliary calculus. In the latter proof figure there may occur sequents whose succedents are empty. In such cases we write the wff $p \wedge \sim p$ in the succedent. If the succedent of the end sequent was empty, we see that this operation merely produces a sequent that has the same associated formula. However there may result figures of the following four kinds that need justification:

$$\frac{\Gamma \longrightarrow F}{\Gamma, \sim F \longrightarrow p \wedge \sim p} \qquad \frac{\Gamma, \sim F \longrightarrow p \wedge \sim p}{\Gamma \longrightarrow F}$$

$$\frac{\Gamma, F \longrightarrow p \wedge \sim p}{\Gamma \longrightarrow \sim F} \qquad \frac{\Gamma \longrightarrow \sim F}{\Gamma, F \longrightarrow p \wedge \sim p}$$

But such figures may again be transformed into a sequence of inference figures in the auxiliary calculus by the use of the figure *Cut* and the final new initial sequent, and in performing this transformation we may ensure that the succedent of all sequents remain nonempty. We illustrate the transformation for one case below.

$$\dfrac{\Gamma \longrightarrow \sim F \qquad \dfrac{\dfrac{\dfrac{F, \sim F \longrightarrow p \wedge \sim p}{\sim F, F \longrightarrow p \wedge \sim p}\ \sim\text{−IS}}{\sim F \longrightarrow \sim F, p \wedge \sim p}\ \text{Interchange}}{\Gamma \longrightarrow \sim F, p \wedge \sim p}\ \text{Cut}}{\Gamma, F \longrightarrow p \wedge \sim p}\ \sim\text{−ES}$$

Now it is easily seen that the associated formulas of the initial sequents of the auxiliary calculus are theorems of $\mathcal{T}^*$; for instance the associated formula of the sequent $F, \sim F \longrightarrow A$ is $F \wedge \sim F \supset A$ which is a tautology. For each inference figure in which no sequent has an empty succedent we may, similarly, verify that the associated formula of the lower sequent is a theorem if the associated formula(s) of the upper sequent(s) is. For example take the figure:

$$\frac{\Gamma, F \longrightarrow \Delta \qquad \Gamma, G \longrightarrow \Delta}{\Gamma, F \vee G \longrightarrow \Delta}\ \vee\text{−IA}$$

The associated formulas of the upper sequents are $\wedge\Gamma \wedge F \supset \vee\Delta$ and $\wedge\Gamma \wedge G \supset \vee\Delta$, respectively. If these are theorems, we obtain easily that $\Gamma \vdash_{\mathcal{T}^*} F \supset \vee\Delta$ and $\Gamma \vdash_{\mathcal{T}^*} G \supset \vee\Delta$.

But then from the tautologous proposition $[F \supset \vee\Delta] \supset [[G \supset \vee\Delta] \supset [F \vee G \supset \vee\Delta]]$, we obtain $\Gamma \vdash_{\mathcal{T}^*} F \vee G \supset [\vee\Delta]$. Finally by Theorem 2.3.1 it is evident that $\wedge\Gamma \wedge [F \vee G] \supset \vee\Delta$ is a theorem of $\mathcal{T}^*$. The other cases yield to a similar argument. ∎

2.5.2 Lemma. *A wff$_o$ is a theorem of $\mathcal{T}^*$ only if there is a proof figure for a sequent equivalent to it.*

Proof. For this it is enough to show that the property

$$\longrightarrow \mathcal{S} \quad \text{has an LKH}\Sigma \text{ proof figure}$$

is an abstract derivability property of finite sets, $\mathcal{S}$, of wffs$_o$*. But this is easily verified. *ADP1-3, 5, 7* and *10* are immediate from the inference figure schemata. Consider, for instance, *ADP10*. From a proof figure for $\longrightarrow \mathcal{S}_1, \Sigma P, P C, \mathcal{S}_2$ we obtain one for $\longrightarrow \mathcal{S}_1, \Sigma P, \mathcal{S}_2$ by adjoining below the end sequent of the proof figure for the first sequent the inference figures $\Sigma-$IS and Contraction.

For *ADP6, 8,* and *9*, we use Lemma 2.4.3. For instance, consider *ADP9*. If the sequent $\longrightarrow \mathcal{S}_1, \sim P y, \mathcal{S}_2$ has a proof figure, we see that $P y \longrightarrow \mathcal{S}_1, \mathcal{S}_2$ has one too. Further, y does not occur free in the other wffs of the second sequent if such is the case for the first sequent. But then, by a use of the inference figures $\Sigma-$IA and $\sim-$IS we obtain a proof figure for the sequent $\longrightarrow \mathcal{S}_1, \sim\Sigma P, \mathcal{S}_2$.

Finally, one half of *ADP4* follows directly. The other half follows from Lemma 2.4.3. ∎

2.5.3 Corollary. *Let Δ be nonempty sequence of wffs$_o$. Then there is a proof figure for $\Gamma \longrightarrow \Delta$ if and only if $\Gamma \vdash_{\mathcal{T}^*} \vee\Delta$.*

Section 6: Discussion

There has been a certain amount of interest in automating the construction of proofs in a higher-order logic such as $\mathcal{T}^*$, based on the observation [38] that it is "higher-order logic, and not first-order logic, which is the natural technical framework for the 'mechanization of mathematics'." Investigations in this regard have revealed that a higher-order logic of this sort possesses several proof-theoretic properties akin to a first-order logic that are necessary for its automation. For instance, [26] shows that there is a generalisation to Herbrand's Theorem — the basis for automating first-order logics — that holds for the logical system $\mathcal{T}$. Higher-order resolution [1] and unification [22] have also been developed, and based on these principles several theorem-provers (see [21], [4], and the other references in [4]) for higher-order logics have been built. These systems have been able to generate proofs for some theorems of mathematics that appear to be beyond the realm of existing first-order

* When we treat a finite set of wffs as a sequence, we mean actually an arbitrary listing of the members of the set.

theorem-provers; [4], for instance, describes a system that is able to produce a proof for Cantor's Theorem for sets.

Although much success has been encountered in extending theorem-proving techniques for first-order logics to a higher-order logic, there is one problem that makes it difficult to describe a system that is complete and at the same time "tractable" for a higher-order logic. This problem is raised by the fact that predicate variables may occur extensionally in formulas in such a logic. The traditional method for constructing a proof of a formula in a logic that involves quantification may, in a general sense, be thought to consist of substituting expressions for existentially quantified variables and then verifying that the resulting formula is a tautology. The nature of a first-order logic is such that the propositional structure remains invariant under substitutions. As a result, the search for a proof can be based on the propositional structure of the formula, and the role of the substitution (or, more appropriately, unification) process may be reduced to that of a constraint on the search*. However, the situation is different in a logic where predicate variables can occur extensionally. As we observed in Example 2.2.6, applying a substitution to a formula which contains such variables may very well change the propositional structure of the formula. In such cases, the construction of a proof often involves finding the "right" way in which to change the propositional structure. Although some work has been done that gives an indication of useful techniques in this task (*e.g.* in [7]), no good method that is also complete has yet been described for determining these kinds of substitutions. The existing theorem-provers for a higher-order logic either sacrifice completeness or are fairly intractable for this reason; the one in [4], for instance, does not perform a search for such substitutions, and the theorem-prover of [21] performs an exhaustive and undirected search.

While the issue of describing general methods for finding substitutions for predicate variables in a higher-order logic is an important one, we shall consider this problem for only a restricted class of wffs_o of $\mathcal{T}^*$ in this thesis. For this class, we shall observe that a complete proof procedure can be described that finds substitutions for predicate variables almost entirely through the process of unification. Part of the motivation for studying this class of wffs is due to the fact that it provides us with a basis for generalising the paradigm of logic programming in the manner we desire: It enables us to introduce predicate and function variables into (first-order) definite clauses, and to describe a richer domain of terms from which these variables may take values. We now turn to a description of this class of wffs and to a study of its properties.

---

* The appropriateness of this description of the process comes out most clearly if one considers, for example, the method for finding proofs that is described in [3].

Chapter 3
A Class of Higher-Order Formulas

Our objective in this chapter is to provide a basis for higher-order notions within the context of logic programming. In this enterprise, we use $\mathcal{T}^*$ to describe a generalisation of the programming paradigm of first-order logic. In the first section of this chapter we identify wffs of $\mathcal{T}^*$ that we call higher-order definite sentences and goal formulas. These formulas are intended to provide higher-order analogues to the first-order notions of program and query, and we describe the manner in which they generalise these notions. In Section 3.2 we study the nature of $LKH\Sigma$ proof figures for sequents whose antecedents consist of definite sentences and whose consequents contain only goal formulas. We show here the existence of a normal form for proof figures for such sequents. From this analysis, we distill a simplified sequent calculus that we use in studying the properties of definite sentences. In Section 3.3 we return to a consideration of the computational paradigm described in the first section of this chapter. The work here is devoted to showing that our definite sentences possess properties that make this a satisfactory paradigm; briefly, the notion of a proof in $\mathcal{T}^*$ does indeed provide a clear sense for an "answer" to a (higher-order) query, in a manner quite similar to the case with first-order definite clauses. The discussion in this section also provides an insight into the structure of a search for such answers. A satisfactory description of such a search, however, requires an elucidation of the problem of unification of wffs in $\mathcal{T}^*$ and in Section 3.4 we digress briefly to consider this problem. We then describe the notion of a $\mathcal{P}$-derivation that is intended to be a syntactic object encoding the proof of a query from a program $\mathcal{P}$. An interpreter for a logic programming language that is based on our notions of higher-order definite sentences and goal formulas may, in the abstract, be described as a mechanism that actually constructs $\mathcal{P}$-derivations. We take up the description of such a language and the practical considerations in the design of an interpreter for this language in the next chapter.

Section 1: Higher-Order Definite Sentences

In this section we identify the wffs_o of $\mathcal{T}^*$ that we call higher-order definite sentences and goal formulas. Intrinsic to the definition of these wffs, is the notion of a positive formula. These formulas are so named because they are exactly those wffs of $\mathcal{T}^*$ in which the negation symbol does not appear.

3.1.1. Definition. $\mathcal{PF}$, the class of positive formulas, is the smallest collection of wffs of $\mathcal{T}^*$ that satisfies the following properties:

(i) Each variable and each constant except $\sim$ is in $\mathcal{PF}$, and

(ii) If F, G and H are in $\mathcal{PF}$ then so are $\lambda x.F$ and $[G\,H]$.

Implicit in our definition is the fact that each positive formula has a type associated with it. Furthermore, we expect in (ii) that if the type of H is α then the type of G is $\alpha \to \beta$.

3.1.2. Definition. $\mathcal{H}^+$, the *Positive Herbrand Universe*, is the collection of all λ-normal formulas in $\mathcal{PF}$. The *Herbrand Base*, $\mathcal{HB}$, is the collection of all closed wffs in $\mathcal{H}^+$.

$\mathcal{H}^+$ is evidently a "canonical" representation of $\mathcal{PF}$. As will become apparent, $\mathcal{HB}$ in our context plays the same role as the *Herbrand Universe* does in the context of other discussions of logic programming: it is the domain of terms that is used in describing the results of computations.

3.1.3. Definition. A *goal formula* is a formula of type o in $\mathcal{H}^+$. A *positive atom* is an atomic goal formula. A *rigid* positive atom is a positive atom that has a parameter as its head.

A positive atom is thus either $\top$ or a formula of the form $[A\, F^1\, \ldots\, F^n]$ where A is a parameter or a variable of type $\alpha_1 \to \ldots \to \alpha_n \to o$ and, for each $1 \leq i \leq n$, F^i is a positive λ-normal formula of type α_i; it is a rigid positive atom just in case A is also a parameter. It is easily verified that goal formulas have the following inductive characterisation: a positive atom is a goal formula, $A \vee B$ and $A \wedge B$ are goal formulas if A and B are, and $\Sigma\, P$ is a goal formula if $P \in \mathcal{H}^+$ has type $\alpha \to o$ and Σ has type $(\alpha \to o) \to o$; in the last case we observe that $\lambda norm(P\, C)$ is a goal formula for any $C \in \mathcal{H}^+$ of type α.

We now use the notions of a goal formula and a rigid positive atom to define the notion of a higher-order definite sentence.

3.1.4. Definition. Let G be an arbitrary goal formula and A be any rigid positive atom. Let $\bar{x}$ be an arbitrary listing of all the variables free in either G or A. Then the formula $\forall \bar{x}.G \supset A$ is a *(higher-order) definite sentence.*

The definite sentences defined above are a generalisation of the first-order definite clauses. It is of some interest, at this point, to observe the nature of the generalisation. Our definite sentences are formulas of the form

$$\forall x^1.\ \ldots\ \forall x^n.G \supset A$$

where G is a goal formula, A is a rigid positive atom and the variables that are free in either G or A are included in the list $x^1, \ldots, x^n$. First-order definite clauses are contained in our definite sentences under an implicit encoding. The encoding essentially assigns types to the first-order terms and predicates. Specifically, let i be a particular sort. Then the encoding assigns the type i to variables and constants, the type $i \to \cdots \to i \to i$, with $n+1$ occurrences of i, to each n-ary function symbol, and the type $i \to \cdots \to i \to o$, with n occurrences of i, to each n-ary predicate symbol. Looked at differently, our definite sentences contain

within them a many-sorted version of first-order definite clauses. Indeed they contain, in this sense, a larger class of first-order formulas than the (first-order) definite clauses since a goal formula, *i.e.* the formula on the left of the $\supset$ symbol in our definite sentences, may contain nested disjunctions and existential quantifications. This generalisation is inconsequential in the first-order case since we are primarily interested in the logical properties of sets of definite clauses and a first-order formula of the above sort can be rendered into a set of first-order definite clauses that is logically equivalent to it. In the higher-order context it is more natural to retain the embedded disjunctions and existential quantifications because the substitutions that need to be considered in the course of constructing proofs have the potential for reintroducing them. Our definite sentences do embody a genuine generalisation to first-order definite clauses, though, since they may contain complex terms that are constructed by the use of abstractions and applications. Furthermore, these sentences may also include quantifications over variables that correspond to functions and predicates.

In the rest of this chapter we shall examine the properties of definite sentences and closed goal formulas that make them a suitable basis for describing a notion of computation. To preview the main results of this chapter let us introduce the following definition.

3.1.5. Definition. A substitution φ is a positive substitution if its range is contained in $\mathcal{H}^+$. It is a closed positive substitution if its range is contained in $\mathcal{HB}$.

Now let $\mathcal{P}$ be a finite collection of definite sentences, and let G be a goal formula all of whose free variables are contained in the listing $x^1, \ldots, x^n$. We shall see, then, that $\mathcal{P} \vdash_{\mathcal{T}^*} \exists x^1 \ldots \exists x^n . G$ just in case there is a closed positive substitution φ for $x^1, \ldots, x^n$ such that $\mathcal{P} \vdash_{\mathcal{T}^*} \varphi(G)$. This observation shall also facilitate the description of a simple proof procedure that may be used to extract substitutions such as φ. One of the consequences of these observations is that our definite sentences and goal formulas provide the basis for a generalisation to the programming paradigm of first-order logic: In the abstract, a set of definite sentences may be thought of as a description of relations that hold between wffs in the Herbrand Base; concreteness to the nature of this description is provided through the notion of a proof in $\mathcal{T}^*$. In this context, a goal formula constitutes a request to evaluate some of these relations.

It might be useful, at this point, to consider a few examples that illustrate the "higher-order" nature of our definite sentences and shed some light on the nature of the programming paradigm discussed above.

3.1.6. Example. Let `nil` and `cons` be parameters of type $list\,int$ and $int \to (list\,int) \to (list\,int)$ respectively, and let `mapfun` be a parameter of type $(int \to int) \to (list\,int) \to (list\,int) \to o$. Then, the following list of definite sentences constitute a "program."

$\forall$`f.[`$\top \supset$ `[mapfun f nil nil]]` ,
$\forall$`f.`$\forall$`x.`$\forall$`l1.`$\forall$`l2.[[mapfun f l1 l2]` $\supset$

```
                    [mapfun f [cons x l1] [cons [f x] l2]]].
```

If `F`, `L1`, and `L2` are wffs in $\mathcal{HB}$ of types $int \to int$, $list\,int$ and $list\,int$, respectively, then we see that `mapfun F L1 L2` is provable in $\mathcal{T}^*$ from the above definite sentences only if `L2` is the "list" that results from the application of `F` to each element of the list `L1`. Thus, these sentences may intuitively be thought of as specifying the tuples ⟨`F,L1,L2`⟩ that satisfy this property.

Let `g` be a parameter of type $int \to int \to int$. Then the following goal formula is a "query"

```
∃l.[mapfun λx.[g x 1] [cons 1 [cons 2 nil]] l].
```

An "answer" to this query is the substitution {⟨`l,[cons [g 1 1] [cons [g 1 2] nil]]`⟩}. Another example of a query is

```
∃f.[mapfun f
          [cons 1 [cons 2 nil]]
          [cons [g 1 1] [cons [g 1 2] nil]]].
```

An answer to this query requires "computing" the substitution {⟨`f,λx.[g x 1]`⟩}.

3.1.7. Example. Let `primrel` be a parameter of type $(i \to i \to o) \to o$. Then consider the following definite sentences where `rel`, `wife`, `mother`, `jane`, and `mary` are parameters of appropriate types.

```
⊤ ⊃ [primrel mother],
⊤ ⊃ [primrel wife],
∀r.[[primrel r] ⊃ [rel r]],
∀r.∀s.[[primrel r ∧ primrel s] ⊃
        [rel λx.λy.∃z.[[r x z] ∧ [s z y]]]],
⊤ ⊃ [mother jane mary],
⊤ ⊃ [wife john jane].
```

If we assume that i is the type of individuals, then the first two sentences identify "primitive" relations between individuals, and the next two sentences specify other relations that are a result of "joining" primitive relations. The last two sentences specify particular relations that hold between (the individuals denoted by) `jane`, `mary` and `john`. The query

```
∃r.[[rel r] ∧ [r john mary]]
```

asks for a relation (in the sense of `rel`) between `john` and `mary`. An answer to this query is the substitution {⟨`r,λx.λy.∃z.[[wife x z] ∧ [mother z y]]`⟩}; intuitively, that `mary` is related to `john` by being his mother-in-law.

The second example illustrates the use of a predicate variable in a query. We note that when the substitution considered for this variable is applied to the matrix of the query,

it produces a formula that has new logical connectives in it. However, since the range of the substitution is contained in $\mathcal{HB}$, the resulting formula is still a goal formula. Moreover, this is the only substitution for `r` which, when applied to `[rel r]` $\wedge$ `[r john mary]`, produces a wff that is provable from the given set of definite sentences. Consider, however, the query

$\exists$`r.[r john mary]`.

Now there are several substitutions for `r` whose range is not in $\mathcal{HB}$ which may be applied to `r john mary` to yield a wff that is provable from the definite sentences in Example 3.1.7; for instance, each substitution of the form

$$\{\langle \mathtt{r}, \lambda x.\lambda y.[\sim[P\,x\,y] \vee [P\,x\,y]]\rangle\},$$

where P is any predicate of two arguments, serves this purpose. However, we shall see in the next section that each of these substitutions can be "transformed" into a simpler substitution whose range is in $\mathcal{HB}$ and which also serves the purpose; for instance, each of the substitutions considered above can be transformed into the ones $\{\langle \mathtt{r}, \lambda x.\lambda y.[\top \vee [P\,x\,y]]\rangle\}$.

Section 2: A Simplified Sequent System for Definite Sentences

Our objective in this section is to show the result alluded to at the end of the last section, namely that it is sufficient to consider only positive substitutions in constructing a proof for a goal formula from a set of definite sentences. Our demonstration of this fact is based on an analysis of $LKH\Sigma$ proof figures for sequents containing only these formulas. In conjunction with this fact we shall also see the existence of a "normal form" for proof figures for such sequents, in that we need to consider proof figures in which only certain restricted kinds of $LKH\Sigma$ inference figures appear. From this observation we shall extract a simpler sequent calculus that is complete for such sequents, and that is especially convenient for the study of the properties of definite sentences that we undertake in the remainder of this chapter.

Before we begin our analysis of $LKH\Sigma$ proof figures for sequents containing definite sentences and goal formulas, we note that we shall find it necessary to deal with an unabbreviated representation of definite sentences. This representation may be made clear as follows.

3.2.1. Definition. The class of *(higher-order) definite clauses** is the subcollection of wffs_o that is specified inductively by the following rules

* The qualification "higher-order" is intended to distinguish the formulas introduced by the above definition from the first-order formulas of the same name. In the remainder of our discussions we omit this qualification if, by so doing, no confusion should arise.

(i) $\sim G \vee A$ is a definite clause if G is a goal formula and A is a rigid positive atom, and

(ii) $\sim\Sigma\, \lambda x.\sim D$ is a definite clause if D is a definite clause; here Σ has type $(\alpha \to o) \to o$ if x has type α.

It is apparent, then, that a definite sentence is a closed definite clause.

$LKH\Sigma$ Proof Figures for Definite Sentences. We are interested in considering proof figures for sequents in which the antecedent is a sequence of definite sentences and the succedent is a sequence of closed goal formulas. Let us identify the following inference figure schemata

$$\frac{\Gamma_1, \rho([\lambda x.P]\, C), \Gamma_2 \;\longrightarrow\; \Delta}{\Gamma_1, \sim\Sigma\, \lambda x.\sim P, \Gamma_2 \;\longrightarrow\; \Delta} \quad \text{all} - \text{IA}^+$$

$$\frac{\Gamma \;\longrightarrow\; \Delta_1, \rho(P\, C), \Delta_2}{\Gamma \;\longrightarrow\; \Delta_1, \Sigma\, P, \Delta_2} \quad \Sigma\text{–IS}^+$$

in which we expect only wffs from $\mathcal{HB}$ to be substituted for C. These are evidently subcases of the schemata all – IA and Σ–IS, respectively. The main result of this section establishes that if any $LKH\Sigma$ proof figure exists for a sequent of the kind we are interested in, then there is a proof figure for the same sequent in which all occurrences of the inference figures all – IA and Σ–IS are also occurrences of the figures all – IA^+ and Σ–IS^+ respectively. This observation has the consequence that several inference figure schemata may be removed from the calculus $LKH\Sigma$ while preserving the "completeness" of the calculus for the kinds of sequents that we are interested in.

Our proof of the result outlined above is obtained by effecting a transformation on the $LKH\Sigma$ proof figures of the form described in Lemma 2.4.4. The main idea underlying this transformation is as follows. Let us assume that a given proof figure contains an inference figure obtained from the schema Σ–IS or all – IA by substituting a wff for C in which the constant $\sim$ appears. Then, we shall see that no essential role is played by the symbol $\sim$ in the proof figure that cannot also be played by the wff $\lambda z.\top$. Consequently, we may replace each occurrence of this symbol in the wff substituted for C by $\lambda z.\top$, thereby obtaining a positive formula. This is the purpose of the function *pos* of Definition 3.2.4. We define this process of replacement precisely below, and show that it "produces" an alternative proof figure for the same end sequent.

In describing the transformation and in showing its correctness, we find it useful to identify the following class of wffs_o that contains the class of definite clauses.

3.2.2. Definition. A wff_o is an *implicational formula* just in case it has one of the following forms

(i) $\sim F \vee A$ where F and A are λ-normal formulas and in addition A is a rigid atom.

(ii) $\sim\Sigma\,\lambda x.\sim F$ where F is itself an implicational formula.

3.2.3 Lemma. *Let F be an implicational formula and let D be a definite clause. Further, if x is a variable$_\alpha$, let C be a wff$_\alpha$ and C' be a positive formula of type α. Then*

(i) $\rho(F)$ is an implicational formula and $\rho(D)$ is a definite clause.

(ii) $\lambda norm([\lambda x.F]\,C)$ is an implicational formula and $\lambda norm([\lambda x.D]\,C')$ is a definite clause.

Proof. (i) Implicational formulas and definite clauses are, by virtue of their definitions, λ-normal formulas. Hence $\rho(F) \equiv F$ and $\rho(D) \equiv D$. The claim then follows from the fact that an α-conversion preserves the structure of a wff.

(ii) Let A be a wff, y be a variable$_\alpha$ and B be a wff$_\alpha$. The following are then easily seen to be true: If A is a rigid atom then so are $S^y_B\,A$ and $\lambda norm(A)$. If A and B are positive formulas, then so are $S^y_B\,A$ and $\lambda norm(A)$.

We observe, now, that there must be a definite clause D' such that $D \equiv D'$, the bound variables of D' are distinct from x and C is free for x in D'. To show that $\lambda norm([\lambda x.D]\,C')$ is a definite clause, it is, then, sufficient to show that $\lambda norm(S^x_{C'}\,D')$ is. But this follows by an induction on the structure of D': If D' is $\sim G \vee A$ then $\lambda norm(S^x_{C'}\,D')$ is of the form $\sim G^1 \vee A^1$ where G^1 is $\lambda norm(S^x_{C'}\,G)$ and A^1 is $\lambda norm(S^x_{C'}\,A)$. By our previous observations, G^1 is a goal formula and A^1 is a rigid positive atom, and so $\sim G^1 \vee A^1$ is a definite clause. If D' is $\sim\Sigma\,\lambda y.\sim D^1$ then $\lambda norm(S^x_{C'}\,D') \equiv \sim\Sigma\,\lambda y.\sim D^2$ where D^2 is $\lambda norm(S^x_{C'}\,D^1)$. The claim in this case is now evident from the hypothesis.

An identical argument shows that $\lambda norm([\lambda x.F]\,C)$ is an implicational formula. ∎

We define a function below whose purpose is to map wffs onto positive formulas.

3.2.4. Definition. Let $z \in \mathcal{V}ar_o$ be a designated variable. The function *pos* on wffs is then defined by recursion as follows:

(i) F is a constant or a variable. If F is $\sim$ then $pos(F) = \lambda z.\top$, otherwise $pos(F) = F$.

(ii) F is $\lambda x.F^1$. Then $pos(F) = \lambda x.pos(F^1)$.

(iii) F is $[F^1\,F^2]$. Then $pos(F) = [pos(F^1)\,pos(F^2)]$.

It is easily verified that *pos* maps each wff onto a positive formula of the same type. The following lemma states additional properties of *pos* that we shall find useful.

3.2.5 Lemma. *Let F be a wff, let x be a variable$_\alpha$ and let B be a wff$_\alpha$. Then*

(i) $\mathcal{F}(F) = \mathcal{F}(pos(F))$.

(ii) If B is free for x in F, then B is free for x in $pos(F)$.

(iii) $pos(S^x_B\, F) = S^x_{pos(B)}\, pos(F)$.

Proof. We verify these claims by an induction on the structure of F:

(a) F is a constant or a variable. Consider first the case when F is not $\sim$. In this case $pos(F) = F$ and (i) and (ii) are obvious. For (iii) we have two subcases. If F is x then

$$S^x_{pos(B)}\, F = pos(B) = pos(S^x_B\, F).$$

Otherwise F is not x. But then

$$S^x_{pos(B)}\, F = F = S^x_B\, F.$$

Using the fact that $pos(F) = F$, it follows easily in either of these subcases that

$$pos(S^x_B\, F) = S^x_{pos(B)}\, F = S^x_{pos(B)}\, pos(F).$$

To complete the argument we need to consider the case when F is $\sim$. In this case $pos(F) = \lambda z.\top$ and so

$$\mathcal{F}(F) = \mathcal{F}(pos(F)) = \emptyset.$$

Now (i) follows immediately, and (ii) and (iii) follow by observing further that x is free neither in F nor in $pos(F)$.

(b) F is $\lambda y.G$. Now $pos(F) = \lambda y.pos(G)$. But by the hypothesis $\mathcal{F}(G) = \mathcal{F}(pos(G))$ and so (i) follows. Similarly (ii) is also clear. For (iii), if $x = y$ then

$$pos(S^x_B\, F) = pos(F) = S^x_{pos(B)}\, pos(F).$$

Otherwise $x \neq y$. In this case

$$\begin{array}{rcll} \mathrm{pos}(S^x_B\, F) & = & \lambda y.pos(S^x_B\, G) & \text{by definitions} \\ & = & \lambda y.S^x_{pos(B)}\, pos(G) & \text{by the hypothesis} \\ & = & S^y_{pos(B)}\, \lambda y.G & \text{by definitions.} \end{array}$$

(c) F is $[G\, H]$. Now $pos(F) = [pos(G)\, pos(H)]$ and (i) and (ii) follow by the hypothesis. Using definitions, we see that

$$pos(S^x_B\, F) = [pos(S^x_B\, G)\, pos(S^x_B\, H)],$$

and also that

$$S^x_{pos(B)}\, pos(F) = [S^x_{pos(B)}\, pos(G)\, S^x_{pos(B)}\, pos(H)].$$

(iii) now follows from the hypothesis. ∎

We now use *pos* to determine a mapping from wffs to $\mathcal{H}^+$. We need this mapping in order to describe the transformation of proof figures in Theorem 3.2.14.

3.2.6. Definition. Let F be a wff. Then $pc(F) = \rho(pos(F))$; the wff $pc(F)$ is the *positive correspondent* of F.

It is clear that $\mathcal{F}(pc(F)) \subset \mathcal{F}(F)$. Hence, if F is a closed wff, then $pc(F) \in \mathcal{HB}$. The following property of pc is also obvious.

3.2.7 Lemma. *Let F be a λ-normal formula of type o.*

(i) If F is an atom, then $pc(F)$ is a positive atom.

(ii) If F is $\sim F^1$, then $pc(F) = \top$.

(iii) If F is $G * H$ where $*$ is either $\vee$ or $\wedge$, then $pc(F) = pc(G) * pc(H)$.

(iv) If F is $\Sigma\, P$, then $pc(F) = \Sigma\, pc(P)$

Our purpose now is to show that pc and ρ commute as operations on wffs.

3.2.8 Lemma. *For any wff F, $pc(\rho(F)) = \rho(pc(F))$.*

Proof. Let us say that F' results *directly* from F by an application of a rule of λ-conversion if the subformula of F which is replaced by virtue of one of these rules is F itself. The following observations that follow readily from Lemma 3.2.5 show that if F' results directly from F in such a manner, then $pos(F')$ also results directly from $pos(F)$ in a similar manner.

(a) If F is $\lambda x.G$ and F' is $\lambda y.S^x_y\, G$ then $pos(F)$ is $\lambda x.pos(G)$ and $pos(F')$ is $\lambda y.S^x_y\, pos(G)$. Further if y is free for x in G then y is free for x in $pos(G)$.

(b) If F is $[\lambda x.G]\, H$ and F' is $S^x_H\, G$ then $pos(F)$ is $[[\lambda x.pos(G)]\, pos(H)]$ and $pos(F')$ is $S^x_{pos(H)}\, pos(G)$. Further if H is free for x in G then $pos(H)$ is free for x in $pos(G)$.

(c) If F is of type $\alpha \to \beta$ then $pos(F)$ is also of type $\alpha \to \beta$. If F' is $\lambda y.[F\, y]$ then $pos(F')$ is $\lambda y.[pos(F)\, y]$. Further $y \in \mathcal{F}(F)$ if $y \in \mathcal{F}(pos(F))$.

An induction on the structure of F now shows that if F' results from F by an application of a rule of λ-conversion, then $pos(F')$ results from $pos(F)$ by a similar rule: If F is a constant or a variable this is immediate, since F' must result directly from F. Now suppose F is of the form $\lambda y.G$ and F' does not result directly from F. Then F' is of the form $\lambda y.G'$ where G' is obtained from G by a rule of λ-conversion. By the hypothesis, therefore, $pos(G')$ results from $pos(G)$ by a rule of λ-conversion. Now $pos(F) = \lambda y.pos(G)$ and $pos(F') = \lambda y.pos(G')$, and hence $pos(F')$ must also result from $pos(F)$ by a rule of λ-conversion. A similar argument is used for the case when F is $[F^1\, F^2]$.

From the above arguments it is evident that $pos(\rho(F))$ results from $pos(F)$ by the application of the rules of λ-conversion. Hence,

$$\rho(pos(F)) = \rho(pos(\rho(F))).$$

Noting further that $\rho(pos(F)) = \rho(\rho(pos(F)))$, the lemma follows. ∎

The lemma has the following corollary that we need for the proof of the main theorem of this section.

3.2.9 Corollary. *If P and C are wffs of appropriate types, then*

$$pc(\rho([P\,C])) = \rho([pc(P)\,pc(C)]).$$

Proof. The claim is evident from the following equalities:

$$\begin{array}{rcll} pc(\rho([P\,C])) & = & \rho(pc([P\,C])) & \text{by Lemma 3.2.8} \\ & = & \rho([pos(P)\,pos(C)]) & \text{using definitions} \\ & = & \rho([\rho(pos(P))\,\rho(pos(C))]) & \text{by properties of } \lambda\text{-conversion} \\ & = & \rho([pc(P)\,pc(C)]) & \text{using definitions. } \blacksquare \end{array}$$

We now define a function on implicational formulas whose purpose is to transform these formulas into definite clauses.

3.2.10. Definition. The function pos_{I} on implicational formulas is defined as follows

(i) If F is $\sim F^1 \vee A$ then $pos_{\mathrm{I}}(F) = \sim pos(F^1) \vee pos(A)$.

(ii) If F is $\sim\Sigma\,\lambda x.\sim F^1$ then $pos_{\mathrm{I}}(F) = \sim\Sigma\,\lambda x.\sim pos_{\mathrm{I}}(F^1)$

Further, if F is an implicational formula then $pc_{\mathrm{I}}(F) = \rho(pos_{\mathrm{I}}(F))$.

It is clear that if F is a definite clause then $pos_{\mathrm{I}}(F) = F$. The following lemma states additional properties of pc_{I} that we need in the proof of Theorem 3.2.14.

3.2.11 Lemma. *Let F be an implicational formula, let x be a variable$_\alpha$, and let C be a wff$_\alpha$. Then*

(i) $pc_{\mathrm{I}}(F)$ is a definite clause.

(ii) $pc_{\mathrm{I}}(\rho([\lambda x.F]\,C)) = \rho([\lambda x.pc_{\mathrm{I}}(F)]\,pc(C))$.

Proof. (i) follows from an induction on the structure of F: If F is $\sim F^1 \vee A$ then $pc_{\mathrm{I}}(F) = \sim\rho(pos(F^1)) \vee \rho(pos(A))$ and the wff on the right-hand side is apparently a definite clause. If F is $\sim\Sigma\,\lambda x.\sim F^1$ then $pc_{\mathrm{I}}(F) \equiv \sim\Sigma\,\lambda x.\sim\rho(pos_{\mathrm{I}}(F^1))$ and the claim now follows from the hypothesis.

(ii) We first note that by Lemma 3.2.3 $\lambda norm([\lambda x.F]\,C)$ is an implicational formula if F is one. Hence pos_{I} is defined on this wff and, consequently, the left-hand side of the equality is defined.

We now claim that $pos_{\mathrm{I}}(\lambda norm([\lambda x.F]C))$ λ-conv $[\lambda x.pos_{\mathrm{I}}(F)]\,pos(C)$. Given the claim, it is clear that $\rho(pos_{\mathrm{I}}(\rho([\lambda x.F]\,C))) = \rho([\lambda x.pos_{\mathrm{I}}(F)]\,pos(C)$. The lemma now follows from the observation that $\rho([\lambda x.A]\,B) = \rho([\lambda x.\rho(A)]\,\rho(B))$.

It is easily seen that for any implicational formula F^1, if $F^1 \equiv F^2$ then $pos_{\mathrm{I}}(F^1) \equiv pos_{\mathrm{I}}(F^2)$. Thus, in showing the claim, we may assume that the bound variables of F are distinct from x and from the free variables of C. We then use an induction on the structure of F:

(a) F is of the form $\sim F^1 \vee A$. In this case

$$\lambda norm([\lambda x.F]\, C) \equiv \sim \lambda norm(S^x_C\, F^1) \vee \lambda norm(S^x_C\, A).$$

We have seen (in the proof of Lemma 3.2.8) that if A λ-conv B then $pos(A)$ λ-conv $pos(B)$. Using the definition of pos_{I} and Lemma 3.2.5 it follows therefore that $\sim S^x_{pos(C)}\, pos(F^1) \vee S^x_{pos(C)}\, pos(A)$ results from $pos_{\mathrm{I}}(\sim \lambda norm(S^x_C\, F^1) \vee \lambda norm(S^x_C\, A))$ by λ-conversions. From this it is evident that the claim holds in this case.

(b) F is of the form $\sim\Sigma\, \lambda y.\sim F'$. But then it is clear that

$$\lambda norm([\lambda x.F]\, C) \equiv \sim\Sigma\, \lambda y.\sim \lambda norm([\lambda x.F']\, C)$$

and, therefore, that

$$pos_{\mathrm{I}}(\lambda norm([\lambda x.F]\, C)) \equiv pos_{\mathrm{I}}(\sim\Sigma\, \lambda y.\sim \lambda norm([\lambda x.F']\, C)).$$

Now from the hypothesis of the induction we know that $[\lambda x.pos_{\mathrm{I}}(F')]pos(C)$ results from $pos_{\mathrm{I}}(\lambda norm([\lambda x.F']\, C))$ by a sequence of λ-conversions. Using the definition of pos_{I}

$$pos_{\mathrm{I}}(\lambda norm([\lambda x.F]\, C)) \quad \lambda\text{-conv} \quad \sim\Sigma\lambda y.\sim[\lambda x.pos_{\mathrm{I}}(F')]\, pos(C)$$

Finally, the claim is evident from the observations that $x \neq y$ and $\mathcal{F}(C) = \mathcal{F}(pos(C)$. ∎

3.2.12. Definition. We extend pc_{I} to the class of all wffs$_o$. Specifically, let F be a wff$_o$. Then

$$pc_{\mathrm{O}}(F) = \begin{cases} pc_{\mathrm{I}}(F), & \text{if } F \text{ is an implicational formula;} \\ pc(F), & \text{otherwise.} \end{cases}$$

The main purpose of the mappings defined above is to enable us, in the proof of Theorem 3.2.14, to replace sequents by what might be thought of as their positive correlates. This transformation on sequents is now defined.

3.2.13. Definition. The mapping pc_{S} on sequents is defined as follows: $pc_{\mathrm{S}}(\Gamma \longrightarrow \Delta)$ is the sequent that results by replacing each wff$_o$ F in Γ by $pc_{\mathrm{O}}(F)$ and each wff$_o$ G in Δ by $pc(G)$.

3.2.14 Theorem. *Let Γ be a sequence of wffs$_o$ that are either definite sentences or closed positive atoms. Let Δ be a sequence of closed goal formulas. Further, let each F in Γ or Δ be in principal normal form. Then the sequent $\Gamma \longrightarrow \Delta$ has an $LKH\Sigma$ proof figure only if it has an $LKH\Sigma$ proof figure in which*

(i) the only inference figures that appear are Contraction, $\vee$−IS, $\wedge$−IS, Σ−IS, imply − IA and all − IA, and

(ii) each occurrence of the figure all − IA or Σ−IS is also an occurrence of the figure all − IA$^+$ or Σ−IS$^+$ respectively.

Proof. Let us assume that $\Gamma \longrightarrow \Delta$ has an $LKH\Sigma$ proof figure. By Lemma 2.4.4 we see that it has a proof figure in which (i) the principal formula for a $\vee-$IA inference figure is not of the form $\sim F \vee G$, (ii) the principal formula for a $\sim-$IA inference figure is not of the form $\sim\Sigma\,\lambda x.\sim P$, and (iii) the inference figure λ does not appear. We may further assume that it has a proof figure of the above kind in which each occurrence of the figures all $-$ IA and $\Sigma-$IS are obtained from the respective schemata by substituting a closed wff for C; if a variable$_\alpha$ y appears free in the wff substituted for C, then we replace each free occurrence of y in it and in the sequents in the proof figure by a parameter$_\alpha$ that does not already appear in the proof figure; it is easy to see that the result is still a proof figure of the same kind, and that its end sequent is still $\Gamma \longrightarrow \Delta$.

Let $\mathbf{T}$ be a proof figure for $\Gamma \longrightarrow \Delta$ that satisfies these properties. We show below that $\mathbf{T}$ can be transformed into a proof figure satisfying the requirements of the theorem.

We first consider all those paths in $\mathbf{T}$ that contain no sequent which is also the upper sequent of a $\sim-$IS inference figure. We call any sequent that appears on such a path an *essential* sequent of $\mathbf{T}$. We claim that every wff in the antecedent of an essential sequent is either a rigid atom or an implicational formula. From this it also follows that each essential sequent, except the root sequent, is the upper sequent of one of the figures *Contraction*, $\vee-$IS, $\wedge-$IS, $\Sigma-$IS, imply $-$ IA or all $-$ IA.

The claim is shown by an induction on the distance of the essential sequent from the root sequent. If this distance is 0, the claim is obviously true. Let us then assume that the claim is true if the distance is d, and verify it for distance $d+1$. By the assumption, we need to consider only those inference figures in which there is a wff in the antecedent of an upper sequent that is not in the antecedent of the lower sequent; *i.e.* the figures $\vee-$IA, $\wedge-$IA, $\sim-$IS, imply $-$ IA, and all $-$ IA. Since we are considering only essential sequents, the figure is clearly not $\sim-$IS. By the hypothesis, the figure is not $\vee-$IA or $\wedge-$IA either. If the figure in question is imply $-$ IA, *i.e.*

$$\frac{\Gamma_1, \Gamma_2 \longrightarrow \Delta_1, F, \Delta_2 \qquad \Gamma_1, G, \Gamma_2 \longrightarrow \Delta_1, \Delta_2}{\Gamma_1, \sim F \vee G, \Gamma_2 \longrightarrow \Delta_1, \Delta_2}$$

we see that the principal formula must be an implicational formula. But then, G is a rigid atom and so the claim holds. If the figure in question is all $-$ IA, *i.e.*

$$\frac{\Gamma_1, \rho([\lambda x.P]\, C), \Gamma_2 \longrightarrow \Delta}{\Gamma_1, \sim\Sigma\,\lambda x.\sim P, \Gamma_2 \longrightarrow \Delta}$$

we see once again that the principal formula is an implicational formula, and the claim now follows from Lemma 3.2.3.

Now let $e(\mathbf{T})$ be the structure that results from removing all those sequents that

are not essential sequents from $\mathbf{T}$, and let $pe(\mathbf{T})$ be the result of replacing each sequent $\Gamma \longrightarrow \Delta$ in $e(\mathbf{T})$ by $pc_{\mathrm{S}}(\Gamma \longrightarrow \Delta)$. We claim that $pe(\mathbf{T})$ is a proof figure for the same sequent that $\mathbf{T}$ is. Since the end sequent of $\mathbf{T}$ has only positive formulas F that are in principal normal form, and since each implicational formula in the antecedent is a definite clause, it is clear that if $pe(\mathbf{T})$ is a proof figure, then it is a proof figure for the same sequent as $\mathbf{T}$ is. To see that $pe(\mathbf{T})$ is a proof figure we first note that $e(\mathbf{T})$ is obviously a "tree-like" arrangement of sequents and hence $pe(\mathbf{T})$ is too. Further, the leaf sequents of $e(\mathbf{T})$ are of the form $\Gamma_1, A, \Gamma_2 \longrightarrow \Delta_1, A, \Delta_2$ where A is a rigid atom or of the form $\Gamma \longrightarrow \Delta_1, \sim F, \Delta_2$. By Lemma 3.2.7, therefore, the leaf sequents of $pe(\mathbf{T})$ are evidently initial sequents. Thus, it only remains to be verified that each pair of upper sequents and lower sequent in $pe(\mathbf{T})$ are actually instances of the inference figure schemata.

To show this, we consider each of the possible cases in $e(\mathbf{T})$ and check the corresponding pairs in $pe(\mathbf{T})$. The case for *Contraction* is evident. The cases for $\vee-$IS and $\wedge-$IS follow directly from Lemma 3.2.7. The case for imply $-$ IA follows from that observation that the principal formula is an implicational formula of the form $\sim F \vee A$ and for such a formula $pc_{\mathrm{I}}(\sim F \vee A) = \sim pc(F) \vee pc(A)$. If the inference figure in $e(\mathbf{T})$ is $\Sigma-$IS, *i.e.* of the form

$$\frac{\Gamma \longrightarrow \Delta_1, \rho(P\,C), \Delta_2}{\Gamma \longrightarrow \Delta_1, \Sigma\,P, \Delta_2}$$

then the claim follows from Lemma 3.2.7 and Corollary 3.2.9; $pc(\Sigma\,P) = \Sigma\,pc(P)$ and $pc(\rho(P\,C)) = \rho(pc(P)\,pc(C))$. Note, moreover, that since C is a closed wff, $pc(C) \in \mathcal{HB}$ and so the corresponding figure in $pe(\mathbf{T})$ is actually an instance of $\Sigma-\mathrm{IS}^{+}$. Finally, if the inference figure in $e(\mathbf{T})$ is all $-$ IA, *i.e.* of the form

$$\frac{\Gamma_1, \rho([\lambda x.P]\,C), \Gamma_2 \longrightarrow \Delta}{\Gamma_1, \sim\Sigma\,\lambda x.\sim P, \Gamma_2 \longrightarrow \Delta}$$

the claim follows from the Lemma 3.2.11: Obviously $\sim\Sigma\,\lambda x.\sim P$ is an implicational formula and so $pc_{\mathrm{I}}(\rho([\lambda x.P]\,C)) = \rho([\lambda x.pc_{\mathrm{I}}(F)]\,pc(C))$. Moreover, since C is a closed wff, $pc(C) \in \mathcal{HB}$ and, hence, the corresponding figure in $pe(\mathbf{T})$ is actually an instance of all $-$ IA^{+}.

Thus, $pe(\mathbf{T})$ is a proof figure for the same end sequent. Further, as is evident from the above argument, it is a proof figure of the kind claimed in the theorem. ∎

3.2.15. Example. We illustrate the transformation described in the above theorem by considering a simple example. Let P, A, and Q be parameters of suitable types. Consider then the following arrangement of sequents that constitute a proof figure for the sequent $\sim\Sigma\,\lambda x.\sim[\sim x \vee [P\,A]] \longrightarrow [P\,A]$:

$$\dfrac{\dfrac{\dfrac{\dfrac{Q \longrightarrow [P\,A], Q}{\longrightarrow [P\,A], \sim Q, Q}\sim\!\text{IS}}{\longrightarrow [P\,A], \sim Q \vee Q}\vee\!\text{IS} \qquad [P\,A] \longrightarrow [P\,A]}{\sim[\sim Q \vee Q] \vee [P\,A] \longrightarrow [P\,A]}\,\text{imply}-\text{IA}}{\sim\Sigma\,\lambda x.\sim[\sim x \vee [P\,A]] \longrightarrow [P\,A]}\,\text{all}-\text{IA}$$

Using the methods described in the above theorem, this proof figure can be transformed into the proof figure shown below, that has the same end sequent.

$$\dfrac{\dfrac{\dfrac{\longrightarrow [P\,A], \top, Q}{\longrightarrow [P\,A], \top \vee Q}\vee\!\text{IS} \qquad [P\,A] \longrightarrow [P\,A]}{\sim[\top \vee Q] \vee [P\,A] \longrightarrow [P\,A]}\,\text{imply}-\text{IA}}{\sim\Sigma\,\lambda x.\sim[\sim x \vee [P\,A]] \longrightarrow [P\,A]}\,\text{all}-\text{IA}$$

Notice that the wff $\sim Q \vee Q$ that is used to "instantiate" x in the first proof figure is replaced in the second proof figure by $\top \vee Q$.

The Sequent Calculus $LKHD$. The content of Theorem 3.2.14 may be expressed by the description of a simplified sequent calculus for definite sentences. This calculus, which we call $LKHD$, is the following. Its initial sequents are obtained from the schemata

$$\Gamma_1, F, \Gamma_2 \longrightarrow \Delta_1, F', \Delta_2 \qquad\qquad \Gamma \longrightarrow \Delta_1, \top, \Delta_2$$

by replacing F and F' by atoms A and A' such that $A \equiv A'$; the Γs and Δs are to be replaced as usual by finite sequences of wffs$_o$. The inference figures of $LKHD$ are those obtained from the schemata in Figure 3.2.1 by a substitution of the kind described in Section 2.4 with the only exception that C is now to be replaced by a wff from $\mathcal{HB}$.

The discussions in this section are summarised in the following theorem.

3.2.16 Theorem. *Let Γ be a finite sequence of wffs$_o$ that are either definite sentences or closed positive atoms, and let Δ be a finite sequence of closed goal formulas. Then $\Gamma \vdash_{\mathcal{T}^*} \vee\Delta$ if and only if $\Gamma \longrightarrow \Delta$ has an $LKHD$ proof figure.*

Proof. ($\supset$) Let us assume first that each wff F in Γ and Δ is such that $\rho(F) = F$. By Corollary 2.5.3 and Theorem 3.2.14 it follows that $\Gamma \longrightarrow \Delta$ has an $LKHD$ proof figure. Now let Γ' and Δ' be the result of replacing each wff A in Γ and Δ respectively by a wff A' such that $A \equiv A'$. One the one hand, it is clear that $\Gamma' \vdash_{\mathcal{T}^*} \vee\Delta'$ only if $\Gamma \vdash_{\mathcal{T}^*} \vee\Delta$. On the other hand, an easy induction on the height of the proof figure shows that $\Gamma \longrightarrow \Delta$ has a proof figure only if $\Gamma' \longrightarrow \Delta'$ has a proof figure. Since definite clauses, positive atoms

Contraction in the antecedent

$$\frac{\Gamma_1, F, F, \Gamma_2 \longrightarrow \Delta}{\Gamma_1, F, \Gamma_2 \longrightarrow \Delta}$$

Contraction in the succedent

$$\frac{\Gamma \longrightarrow \Delta_1, F, F, \Delta_2}{\Gamma \longrightarrow \Delta_1, F, \Delta_2}$$

$$\frac{\Gamma_1, \Gamma_2 \longrightarrow \Delta_1, F, \Delta_2 \qquad \Gamma_1, G, \Gamma_2 \longrightarrow \Delta_1, \Delta_2}{\Gamma_1, F \supset G, \Gamma_2 \longrightarrow \Delta_1, \Delta_2} \supset -\mathrm{IA}$$

$$\frac{\Gamma \longrightarrow \Delta_1, F, G, \Delta_2}{\Gamma \longrightarrow \Delta_1, , F \wedge G, \Delta_2} \wedge -\mathrm{IS}$$

$$\frac{\Gamma \longrightarrow \Delta_1, F, G, \Delta_2}{\Gamma \longrightarrow \Delta_1, F \vee G, \Delta_2} \vee -\mathrm{IS}$$

$$\frac{\Gamma_1, \rho([\lambda x.P]\, C), \Gamma_2 \longrightarrow \Delta}{\Gamma_1, \forall x.P, \Gamma_2 \longrightarrow \Delta} \forall -\mathrm{IA}$$

$$\frac{\Gamma \longrightarrow \Delta_1, \rho(P\, C), \Delta_2}{\Gamma \longrightarrow \Delta_1, \Sigma\, P, \Delta_2} \Sigma -\mathrm{IS}$$

Figure 3.2.1: The $LKHD$ Inference Figure Schemata

and goal formulas are λ-normal formulas, we may remove the initial restriction placed on the form of Γ and Δ.

($\subset$) From an $LKHD$ proof figure we obtain an $LKH\Sigma$ proof figure by perhaps adding above the initial sequents λ inference figures. The claim now follows from Corollary 2.5.3. ∎

Section 3: Properties of Definite Sentences

As mentioned already, our definite sentences are intended to provide the basis for a generalisation to the programming paradigm of first-order logic. In a manner analogous to the first-order case, we wish to view a finite collection of our definite sentences as a *program*. In the context of such a program, we desire to construe a goal formula as playing

the role of a *query*. The idea of demonstrating that there is a proof in $\mathcal{T}^*$ for a query from a program is then intended to provide us with the notion of a computation.

While the nature of such a generalisation to the computational paradigm of first-order logic is clear, it remains to be shown that it does provide us with a satisfactory basis for programming. Our objective in this section is to demonstrate that our definite sentences, despite the generalisations they provide to first-order definite clauses, retain the properties of their first-order counterparts that are essential to their construal as programs. In the first part of this section we observe some of the characteristics of proofs of goal formulas from a set of definite sentences. A study of these properties enables us to give substance to an abstract view of our definite sentences as specifications and, consequently, to give credence to the view of goal formulas as queries. It also lays the basis for the description of a mechanism for evaluating the results of queries that we take up in the next section.

Before embarking upon these discussions, however, we introduce certain notational conventions that turn out to be convenient. We shall use the symbol $\mathcal{P}$ uniformly to denote an arbitrary, but finite, set of definite sentences. In a similar fashion, we reserve the symbol G, perhaps with subscripts and superscripts, to denote a goal formula; given that the type of a goal formula is o, we hope a subscript, when used, will not be confused for a type symbol. Finally, we shall need the notation that is introduced by the following definition.

3.3.1. Definition. Let D be the definite sentence $\forall \bar{x}.G \supset A$. Then $|D|$ denotes the set of all wffs that can be obtained from $G \supset A$ by substituting closed positive formulas for the variables in $\bar{x}$, *i.e.*

$$|D| = \{\varphi(G \supset A) \mid \varphi \text{ is a closed positive substitution for } \bar{x}\}.$$

This notation is extended to sets of definite sentences:

$$|\mathcal{P}| = \bigcup\{|D| \mid D \in \mathcal{P}\}.$$

We note that, given this definition, $|D|$ is a collection of definite sentences, and so also is $|\mathcal{P}|$.

Proofs from Definite Sentences. Our objective now is to analyse some of the properties of proofs of goal formulas from definite sentences. The main tool that we shall use in this enterprise is the sequent calculus $LKHD$. Consequently, we use the term "proof figure" to mean "$LKHD$ proof figure," except when explicitly qualified. With regard to these proof figures we observe the following property that we need in the remaining proofs in this section.

3.3.2 Lemma. *If the sequent* $\Gamma_1, \Gamma_2 \longrightarrow \Delta_1, \Delta_2$ *has a proof figure of height* h*, then any sequent of the form* $\Gamma_1', \Gamma_1, \Gamma_3', \Gamma_2, \Gamma_2' \longrightarrow \Delta_1', \Delta_1, \Delta_3', \Delta_2, \Delta_2'$ *also has a proof figure of height* h*.*

Proof. By an induction on the height of the proof figure. ∎

The main aim of the discussions below is to show that there is a proof for a goal formula from a finite set of definite sentences if and only if there is also a sequence of goal formulas of a sort to be made precise presently. The property of definite sentences that is critical in showing this fact may be paraphrased in the following statement: There is a proof in $\mathcal{T}^*$ of a disjunctive goal formula from a finite set of definite sentences only if there is also a proof for one of the disjuncts from the same set. The proof of this property is contained in the following lemma.

3.3.3 Lemma. *Let Γ be a sequence of wffs$_o$ that are either definite sentences or closed positive atoms. $\Gamma \longrightarrow G^1, \ldots, G^n$ has a proof figure of height h only if there is an i, $1 \leq i \leq n$ such that the $\Gamma \longrightarrow G^i$ has a proof figure of height $\leq h$. Conversely if the second sequent has an proof figure of height h then the first sequent also has one of height h.*

Proof. ($\supset$) We use an induction on the height h of the purported proof figure for the first sequent to show this part of the claim. If h is 1 then there is a G^i that is either $\top$ or $G^i \equiv A$ for some atom A that appears in Γ. From this, the conclusion is obvious for this case. If the proof figure is of height $h+1$, we consider by cases the inference figure for which the end sequent may be the lower sequent. In the cases when the inference figure is *Contraction*, $\Sigma-$IS, and $\forall-$IA the claim follows easily from the hypothesis. If the figure is an $\wedge-$IS, *i.e.* of the form

$$\frac{\Gamma \longrightarrow G^1, \ldots, G^{j-1}, G_1^j, \ldots, G^n \qquad \Gamma \longrightarrow G^1, \ldots, G^{j-1}, G_2^j, \ldots, G^n}{\Gamma \longrightarrow G^1, \ldots, G^{j-1}, G_1^j \wedge G_2^j, \ldots, G^n}$$

then, from the hypothesis, we see that there must be a proof figure of height $\leq h$ for the sequent $\Gamma \longrightarrow G^i$ for some $i \neq j$ or there must be proof figures of height $\leq h$ for the sequents $\Gamma \longrightarrow G_1^j$ and $\Gamma \longrightarrow G_2^j$. In the first case the claim is obviously true and, in the latter case, we use the two proof figures together with an $\wedge-$IS inference figure to construct a proof figure of height $\leq h+1$ for the sequent $\Gamma \longrightarrow G_1^j \wedge G_2^j$. A similar argument suffices for the case where the inference figure is an $\vee-$IS. If the inference is an $\supset -$IA, *i.e.* of the form

$$\frac{\Gamma_1, \Gamma_2 \longrightarrow G^1, \ldots, G^{k-1}, G, G^k, \ldots, G^n \qquad \Gamma_1, A, \Gamma_2 \longrightarrow G^1, \ldots, G^n}{\Gamma_1, G \supset A, \Gamma_2 \longrightarrow G^1, \ldots, G^n}$$

then, from the hypothesis, it is clear there is a j such that either $\Gamma_1, \Gamma_2 \longrightarrow G^j$ has a proof figure of height $\leq h$ or the sequents $\Gamma_1, A, \Gamma_2 \longrightarrow G^j$ and $\Gamma_1, \Gamma_2 \longrightarrow G$ both have proof figures of height $\leq h$. In the first case the claim is evident from Lemma 3.3.2. In the second case, from Lemma 3.3.2 we see that $\Gamma_1, \Gamma_2 \longrightarrow G^j, G$ has a proof figure of height $\leq h$. We now use this proof figure together with the one for $\Gamma_1, A, \Gamma_2 \longrightarrow G^j$

and an $\supset -$IA inference figure to obtain a proof figure of height $\leq h+1$ for the sequent $\Gamma_1, G \supset A, \Gamma_2 \longrightarrow G^j$.

($\subset$) This follows directly from Lemma 3.3.2. ∎

In the discussions that follow we need the notions of a *derivation sequence* for a goal formula from a set of definite sentences and from a set of closed positive atoms, respectively. These are special cases of the following definition.

3.3.4. Definition. Let Γ be a set of wffs$_o$ that are either closed positive atoms or definite sentences, and let G be a closed goal formula. Then a *derivation sequence* for G relative to Γ is a finite sequence of closed goal formulas $G^1, \ldots, G^n$ such that G is G^n, and for each i, $1 \leq i \leq n$,

(i) If G^i is a closed positive atom, then
  (a) G^i is $\top$, or
  (b) for some $G' \in \Gamma$ it is the case that $G^i \equiv G'$, or
  (c) there is a definite clause $D \in \Gamma$ such that $G' \supset A \in |D|$, $G^i \equiv A$ and for some $j < i$, G^j is G'.

(ii) If G^i is $G^i_1 \vee G^i_2$ then for some $j < i$ G^j is either G^i_1 or G^i_2.

(iii) If G^i is $G^i_1 \wedge G^i_2$ then for some $j, k < i$ G^j is G^i_1 and G^j is G^i_2.

(iv) If G^i is ΣP then there is a $C \in \mathcal{HB}$ and a $j < i$ such that $\lambda norm(P\,C)$ is G^j.

3.3.5 Lemma. *Let Γ be a finite set of wffs$_o$ that are either closed positive atoms or definite sentences, and let G be a closed goal formula. Then there is a proof figure for $\Gamma \longrightarrow G$ if and only if there is a derivation sequence for G relative to Γ*.*

Proof. ($\supset$) We show, by an induction on the height of the proof figure for the given sequent, that there must be a derivation sequence for G relative to Γ. The claim is evident if the height is 1, since, then, either G is $\top$ or there is an atom A in Γ such that $G \equiv A$. Let the proof figure, therefore, be of height $h+1$ and we consider, once again by cases, the inference figures of which the end sequent may be the lower sequent. The case when this figure is *Contraction in the antecedent* is evident from the hypothesis and the definition of a derivation sequence. For the case when the figure is *Contraction in the succedent* we use Lemma 3.3.3 and the hypothesis. If the figure is an $\vee-$IS, then evidently G is $G^1 \vee G^2$ and the inference figure under consideration is of the form

$$\frac{\Gamma \longrightarrow G^1, G^2}{\Gamma \longrightarrow G^1 \vee G^2}$$

* We adopt once again a harmless confusion between finite sets and sequences.

From Lemma 3.3.3 it follows that there is a proof figure of height $\leq h$ for either $\Gamma \longrightarrow G^1$ or $\Gamma \longrightarrow G^2$. By the hypothesis there is a derivation sequence for either G^1 or G^2 relative to Γ. But then, we add G at the end of such a sequence to get one for G. A similar argument suffices for $\Sigma-$IS and $\wedge-$IS; in the latter case we shall have to first append two derivation sequences. If the figure is an $\supset -$IA then it is of one of the forms

$$\frac{\Gamma_1, \Gamma_2 \longrightarrow G^1, G \qquad \Gamma_1, A, \Gamma_2 \longrightarrow G}{\Gamma_1, G^1 \supset A, \Gamma_2 \longrightarrow G}$$

$$\frac{\Gamma_1, \Gamma_2 \longrightarrow G, G^1 \qquad \Gamma_1, A, \Gamma_2 \longrightarrow G}{\Gamma_1, G^1 \supset A, \Gamma_2 \longrightarrow G}$$

where Γ is $\Gamma_1, G^1 \supset A, \Gamma_2$. We note first that a derivation sequence relative to Γ_1, Γ_2 is also a derivation sequence relative to Γ. Using Lemma 3.3.3 and the hypothesis we see, therefore, that either there is a derivation sequence for G relative to Γ or there is a derivation sequence for G^1 relative to Γ and a derivation sequence for G relative to Γ_1, A, Γ_2. In the first case the claim is evident and, in the latter case, we obtain a derivation sequence for G relative to Γ by appending the first derivation sequence to the second. Finally, consider the case when the inference figure is an $\forall-$IA, *i.e.* of the form

$$\frac{\Gamma_1, \rho([\lambda x.D]\, C), \Gamma_2 \longrightarrow G}{\Gamma_1, \forall x.D, \Gamma_2 \longrightarrow G}$$

where Γ is $\Gamma_1, \forall x.D, \Gamma_2$. From the hypothesis, there is a derivation sequence for G relative to $\Gamma_1, \rho([\lambda x.D]C), \Gamma_2$. Now it is evident that if any $G' \supset A \in |\rho([\lambda x.D]C)|$ then $G' \supset A \in |\forall x.D|$. From this it follows that the given derivation sequence for G is also one relative to Γ.

($\subset$) We assume the claim for derivation sequences of length $< l$ and we show it for the case when the length is l. Let G be the last wff in such a sequence. If G is either $\top$ or $G \equiv A$ for some atom A in Γ, then $\Gamma \longrightarrow G$ is evidently an initial sequent of $LKHD$. If G is ΣP then, by our definition, there is a $C \in \mathcal{HB}$ such that there is a derivation sequence of length $< l$ for $\rho(P\, C)$. By the hypothesis there is a proof figure for $\Gamma \longrightarrow \rho(P\, C)$. But by adjoining a $\Sigma-$IS inference figure below the end sequent of this proof figure, we obtain one for $\Gamma \longrightarrow \Sigma\, P$. The cases when G is $G^1 \wedge G^2$ and $G^1 \vee G^2$ are similar; in the latter case we need additionally Lemma 3.3.2. Finally, we have the case when, for some $D \in \Gamma$, $G' \supset A \in |D|$, $G \equiv A$ and G' appears earlier in the sequence. By the hypothesis, $\Gamma \longrightarrow G'$ has a proof figure. From this proof figure we obtain one for $\Gamma \longrightarrow G$ by using the following arrangement of inference figures: We assume here that Γ is $\Gamma_1, \forall \bar{x}.D, \Gamma_2$ and that $G' \supset A \in |\forall \bar{x}.D|$.

$$\dfrac{\dfrac{\dfrac{\Gamma_1, \forall x.D, \Gamma_2 \longrightarrow G' \qquad \Gamma_1, A, \Gamma_2 \longrightarrow G}{\Gamma_1, G' \supset A, \forall x.D, \Gamma_2 \longrightarrow G}\ \supset\text{-IA}}{\Gamma_1, \forall x.D, \forall x.D, \Gamma_2 \longrightarrow G}\ \text{sequence of } \forall\text{-IA}}{\Gamma_1, \forall x.D, \Gamma_2 \longrightarrow G}\ \text{Contraction}$$

∎

As a consequence of this lemma we see that, in order to determine whether there is a proof for a goal formula from a set of definite sentences, we may search the space of derivation sequences. In describing such a search, we shall need the following measure of "complexity" of a goal formula; this measure is to be used in Definition 3.4.20.

3.3.6. Definition. Let G be a closed goal formula. Let k be the length of the shortest derivation sequence for G relative to $\mathcal{P}$; if no such sequence exists, we assume that $k = \omega$. Then

$$\mu_{\mathcal{P}}(G) = \begin{cases} 2^k, & \text{if } k < \omega; \\ \omega, & \text{otherwise.} \end{cases}$$

Our proof of the completeness of $\mathcal{P}$-derivations in the next section depends strongly on the properties of this measure observed in the following lemma. The proof of this lemma is obtained easily from Lemma 3.3.5.

3.3.7 Lemma. *Let G be a closed goal formula such that $\mathcal{P} \vdash_{\mathcal{T}^*} G$. Then $\mu_{\mathcal{P}}(G) > 0$ and $\mu_{\mathcal{P}}(G) < \omega$. Further,*

(i) If G is an atom other than $\top$ then there is a $G' \supset G \in |\mathcal{P}|$ such that $\mu_{\mathcal{P}}(G') < \mu_{\mathcal{P}}(G)$.

(ii) If G is $G^1 \vee G^2$ then $\mu_{\mathcal{P}}(G^i) < \mu_{\mathcal{P}}(G)$ for $i = 1$ or $i = 2$.

(iii) If G is $G^1 \wedge G^2$ then $\mu_{\mathcal{P}}(G^1) + \mu_{\mathcal{P}}(G^2) < \mu_{\mathcal{P}}(G)$.

(iv) If G is ΣP then there is a closed positive formula C such that $\mu_{\mathcal{P}}(\lambda norm(P\,C)) < \mu_{\mathcal{P}}(\Sigma P)$.

Definite Sentences as Programs. Another apparent consequence of Lemma 3.3.5 is the following theorem.

3.3.8 Theorem. *Let G be a closed goal formula. Then the following are true.*

(i) If G is $G^1 \wedge G^2$ then $\mathcal{P} \vdash_{\mathcal{T}^*} G$ if and only if $\mathcal{P} \vdash_{\mathcal{T}^*} G^1$ and $\mathcal{P} \vdash_{\mathcal{T}^*} G^2$.

(ii) If G is $G^1 \vee G^2$ then $\mathcal{P} \vdash_{\mathcal{T}^*} G$ if and only if $\mathcal{P} \vdash_{\mathcal{T}^*} G^1$ or $\mathcal{P} \vdash_{\mathcal{T}^*} G^2$.

(iii) If G is $\Sigma\, P$ then $\mathcal{P} \vdash_{\mathcal{T}^*} G$ if and only if there is a $C \in \mathcal{HB}$ such that $\mathcal{P} \vdash_{\mathcal{T}^*} \lambda norm(P\,C)$.

(iv) If G is an atom then $\mathcal{P} \vdash_{\mathcal{T}^*} G$ if and only if there is a formula $G^1 \supset G \in |\mathcal{P}|$ such that $\mathcal{P} \vdash_{\mathcal{T}^*} G^1$.

Let G be a goal formula that has the variables $x^1, \ldots, x^n$ free in it. As a consequence of clause (iii) of the above theorem we see that $\mathcal{P} \vdash_{\mathcal{T}^*} \exists x^1. \ \ldots \exists x^n.G$ only if there is a closed positive substitution φ for $x^1, \ldots, x^n$ such that $\mathcal{P} \vdash_{\mathcal{T}^*} \varphi G$. Thus, we see that definite sentences possess the property necessary to make our notion of the result of a computation a well-defined one. The theorem also shows that definite sentences, together with the notion of a proof in $\mathcal{T}^*$, provide a paradigm of programming that is based on a non-deterministic search. In this context, the propositional connectives $\wedge$ and $\vee$ provide for the specification of *and* and non-deterministic *or* branches in a search. The quantifier Σ corresponds to an infinite *or* branch where each branch is parameterised by a closed wff in $\mathcal{HB}$. Definite sentences provide the basis for the definition of procedures: A definite sentence $\forall \bar{x}.G' \supset A$ may be thought of as describing how a "goal", the name of which is the head of A, may be solved. Notice that the head of A must be a parameter and, therefore, such a construal makes good sense. Given an atomic "goal" G that "unifies" with A, this procedure may be invoked and would lead to an attempt to solve G'. A precise description of this procedural nature of definite sentences and, consequently, of the nature of the search paradigm, requires an explication of the notion of unification, and we undertake this task in the next section.

The discussions leading up to Lemma 3.3.5 also enable us to give substance to the view of a set of definite sentences as a specification of relationships between the terms in $\mathcal{HB}$. The idea is to associate — in a manner that is in some ways similar to that used for the first-order case in, *e.g.* [5] and [41] — a set of closed positive atoms with each collection of definite sentences. For this purpose, we define an *interpretation* to be any set of closed positive atoms; we use the symbol I as a syntactic variable for interpretations. We then use the notion of a derivation sequence to provide a weak notion of satisfaction corresponding to interpretations and goal formulas.

3.3.9. Definition. An interpretation I *satisfies* a closed goal formula G just in case there is a derivation sequence for G relative to I. We denote this relation of satisfaction by the symbol $\models$.

Assuming the ordering on interpretations that is determined by $\subseteq$, it is evident from this definition that $\models$ is a relation that is monotone on its first element; if $I_1 \subseteq I_2$ and $I_1 \models G$ then $I_2 \models G$. Indeed, it is a relation that is continuous on this element:

3.3.10 Lemma. *Let $\langle I_n \rangle_{n<\omega}$ be a sequence of interpretations such that $I_n \subseteq I_{n+1}$. Then $\bigcup_{n<\omega} I_n \models G$ only if, for some $n < \omega$, $I_n \models G$.*

Proof. Since a derivation sequence is finite, the compactness of our notion of satisfaction is clear: $I \models G$ if and only if there is a finite subset I' of I such that $I' \models G$. From this the lemma follows easily. ∎

We use $\models$ to associate a mapping with each collection $\mathcal{P}$ of definite sentences.

3.3.11. Definition. The mapping $T_{\mathcal{P}}$ from interpretations to interpretations is defined as follows.

$$T_{\mathcal{P}}(I) = \{A \mid G \supset A \in |\mathcal{P}| \text{ and } I \models\!\!\!\models G\}$$

From the monotonicity of $\models\!\!\!\models$ it follows that $T_{\mathcal{P}}$ is a monotone operator. Indeed, from Lemma 3.3.10, it also follows that $T_{\mathcal{P}}$ is continuous; *i.e.* if $\langle I_n\rangle_{n<\omega}$ is an increasing sequence of interpretations, then it is clear that

$$T_{\mathcal{P}}(\bigcup_{n<\omega} I^n) = \bigcup_{n<\omega} T_{\mathcal{P}}(I^n).$$

From this fact it follows that $T_{\mathcal{P}}$ has a least fixed point, under the ordering $\subseteq$, that may be obtained by iterating the effect of $T_{\mathcal{P}}$ on $\emptyset$. To be precise, let us define by recursion on the finite ordinals the sequence $\langle I^n_{\mathcal{P}}\rangle_{n<\omega}$ by $I^n_{\mathcal{P}} = T_{\mathcal{P}}(\bigcup\{I^k_{\mathcal{P}} \mid k < n\})$, and let $I^\omega_{\mathcal{P}} = \bigcup_{n<\omega} I^n_{\mathcal{P}}$. Then

3.3.12 Lemma. *$I^\omega_{\mathcal{P}}$ is the least solution to the equation $T_{\mathcal{P}}(I) = I$.*

Proof. Notice that $I^0_{\mathcal{P}} = T_{\mathcal{P}}(\emptyset)$. Since $T_{\mathcal{P}}$ is monotone, it now follows that $I^n_{\mathcal{P}} \subseteq T_{\mathcal{P}}(I^n_{\mathcal{P}}) = I^{n+1}_{\mathcal{P}}$ for each finite ordinal n. From this observation, we see that (i) $\langle I^n_{\mathcal{P}}\rangle_{n<\omega}$ is an increasing sequence of interpretations, and (ii) $\bigcup_{n<\omega} I^n_{\mathcal{P}} = \bigcup_{n<\omega} T_{\mathcal{P}}(I^n_{\mathcal{P}})$. But now since $T_{\mathcal{P}}$ is continuous, it is clear that

$$T_{\mathcal{P}}(I^\omega_{\mathcal{P}}) = \bigcup_{n<\omega} T_{\mathcal{P}}(I^n_{\mathcal{P}}) = I^\omega_{\mathcal{P}};$$

in other words, that $I^\omega_{\mathcal{P}}$ is a solution to the equation.

Now let I' be any other solution. Using the monotonicity of $T_{\mathcal{P}}$, an induction on the finite ordinals shows that, for $n < \omega$, $I^n_{\mathcal{P}} \subseteq I'$; this is evidently true for $n = 0$ since $\emptyset \subseteq I'$ and $T_{\mathcal{P}}(I') = I'$; since $I^{n+1}_{\mathcal{P}} = T_{\mathcal{P}}(I^n_{\mathcal{P}})$, it follows from the hypothesis that $I^{n+1}_{\mathcal{P}} \subseteq I'$. But then $\bigcup_{n<\omega} I^n_{\mathcal{P}} \subseteq I'$ and so $I^\omega_{\mathcal{P}}$ must be the unique least solution. ∎

3.3.13. Example. (i) Let $\mathcal{P}$ be the set of definite sentences in Example 3.1.6. Then $I^\omega_{\mathcal{P}}$ is evidently the set of all closed atoms of the form `[mapfun F L1 L2]`, where `L2` is the "integer list" that results from applying `F` to each element of the integer list `L1`.

(ii) Let $\mathcal{P}$ be the set of definite sentences in Example 3.1.7. Then $I^\omega_{\mathcal{P}}$ is the following set of atoms

```
{ [mother jane mary],  [wife john jane],
  [primrel mother], [primrel wife],
  [rel mother], [rel wife],
  [rel λx.λy.∃z.[[mother x z] ∧ [wife z y]]],
  [rel λx.λy.∃z.[[wife x z] ∧ [mother z y]]],
  [rel λx.λy.∃z.[[wife x z] ∧ [wife z y]]],
  [rel λx.λy.∃z.[[mother x z] ∧ [mother z y]]] }.
```

Given a set of definite sentences $\mathcal{P}$ it is this least fixed point of $T_{\mathcal{P}}$, *i.e.* $I^{\omega}_{\mathcal{P}}$, that we think of as the set of atoms specified by $\mathcal{P}$. The computation that is involved in answering a query G may be viewed as that of determining whether there is a closed substitution instance of G that is satisfied by the set. The consistency of this view with our earlier discussions is the content of the following theorem:

3.3.14 Theorem. *Let G be a closed goal formula and let $\mathcal{P}$ be a finite set of definite sentences. Then $\mathcal{P} \vdash_{\mathcal{T}^*} G$ if and only if $I^{\omega}_{\mathcal{P}} \models G$.*

Proof. We have seen that $\langle I^n_{\mathcal{P}} \rangle_{n<\omega}$ is an increasing sequence of interpretations. Using Theorem 3.2.16 and Lemma 3.3.5 on the one hand and Lemma 3.3.10 on the other, it follows that it is enough to show there is a derivation sequence for G relative to $\mathcal{P}$ if and only for some $n < \omega$ $I^n_{\mathcal{P}} \models G$. We do this below.

($\supset$) By an induction on the length of the derivation sequence. If G is $\top$ then clearly $I^0_{\mathcal{P}} \models G$. If G is $G^1 \vee G^2$, $G^1 \wedge G^2$, or $\Sigma\, P$ then the claim follows from the hypothesis and the definition of $\models$. Finally let $G^1 \supset G \in |\mathcal{P}|$ be such that G^1 appears earlier in the derivation sequence. By the hypothesis there is an $n < \omega$ such that $I^n_{\mathcal{P}} \models G^1$. From the definition of $T_{\mathcal{P}}$ therefore $I^{n+1}_{\mathcal{P}} \models G$.

($\subset$) First assume the claim true if $I^n_{\mathcal{P}} \models G$. Now we show the claim by an induction on the length of the sequence by virtue of which $I^{n+1}_{\mathcal{P}} \models G$. If G is $\top$ then the claim is obviously true. If G is an atom other than $\top$ then there is a $G^1 \supset G \in |\mathcal{P}|$ such that $I^n_{\mathcal{P}} \models G^1$. By our first hypothesis there is a derivation sequence for G^1 relative to $\mathcal{P}$. By adding G at the end of this sequence we obtain one for G. In the cases when G is $G^1 \vee G^2$, $G^1 \wedge G^2$, or $\Sigma\, P$ the claim follows in a similar manner from our second hypothesis. ∎

Before concluding this section, it is to be noted that the definition of satisfaction in first-order contexts is provided in a compositional manner and, in this respect, our notion may appear to be somewhat unsatisfactory. Considerable care must be exercised, however, in providing such a definition in the context of higher-order languages. Take, for example, an attempt to define the relation of satisfaction between interpretations and goal formulas in the following manner:

(i) $I \models \top$,

(ii) $I \models G$ if G is an atom and $G \in I$,

(iii) $I \models G^1 \vee G^2$ if $I \models G^1$ or $I \models G^2$,

(iv) $I \models G^1 \wedge G^2$ if $I \models G^1$ and $I \models G^2$, and

(v) $I \models \Sigma G$ if there is a closed formula $C \in \mathcal{H}^+$ such that $I \models \lambda norm(G\, C)$.

However, this attempt does not yield a well-defined notion: Let $I = \emptyset$. If $G = \lambda x.[x\, A]$, and $C = \lambda y.[\Sigma\, \lambda x.[x\, y]]$ we see that $\lambda norm(G\, C) = \Sigma\, G$. But then, an attempt to answer the question whether $I \models \Sigma\, G$ apparently brings us back to the same question.

There is potential for providing a better definition of satisfaction than the one used here, and the work on general models in [19] provides an indication in this direction. A thorough examination of this issue, however, is beyond the scope of this thesis.

Section 4: Searching for Proofs from Definite Sentences

Our objective now is to describe a mechanism that determines whether there is a proof in $\mathcal{T}^*$ for the existential closure of a goal formula from a set of definite sentences. Such a mechanism may already be described as one that enumerates all the derivation sequences relative to the given set of definite sentences. We would, however, like to describe a procedure that conducts a search for an appropriate derivation sequence that is directed in a sense by the given goal formula. We have already seen that Theorem 3.3.8 provides us with some insight into the structure of such a search. A more complete description of the nature of the search, however, requires us to consider the task of unifying two wffs. This problem has been studied by several researchers, and in most extensive detail by [22]. In the first part of this section, we describe this problem and detail some of the aspects of its solution in [22] that are pertinent to the remaining discussion in the section. In the second part of this section, we use this understanding of the unification problem, and our discussions in the previous section, to introduce the notion of a $\mathcal{P}$-derivation. $\mathcal{P}$-derivations may be looked upon as a generalisation to the higher-order context of the notion of SLD-derivations that were introduced in [5], and are prevalent in most discussions of first-order definite clauses. At one level, they are intended as syntactic objects that demonstrate the existence of a proof for a goal formula and our discussions show their correctness from this perspective. At another level, they are intended to provide a basis for an actual proof procedure — a symbol manipulating procedure that searches for $\mathcal{P}$-derivations would constitute an interpreter for a programming paradigm that is based on our definite sentences — and we explore some of their properties that are pertinent to the description of such a procedure.

The Higher-Order Unification Problem. Let us call a pair of wffs of the same type a *disagreement pair*. A *disagreement set* is then a finite set, $\{\langle F^i, H^i\rangle \mid 1 \leq i \leq n\}$, of disagreement pairs, and a *unifier* for the set is a substitution σ such that, for $1 \leq i \leq n$, $\sigma(F^i) = \sigma(H^i)$. The *higher-order unification problem* is then the following. Given a disagreement set, we desire to determine whether it has unifiers, and to explicitly provide a unifier if it does have one.

The problem described above is a generalisation of the well-known unification problem for first-order terms. The higher-order unification problem has been studied by several researchers and, as a result of their endeavours, it is known that this problem has properties that are, in a certain sense, divergent from those of the problem in the first-order case. It has been shown, for instance, that the question of whether a unifier exists for an arbitrary

disagreement set is an undecidable question [16, 25, 20], whereas the corresponding question for first-order terms is a decidable one. As another example, it has been shown [18] that the notion of a most general unifier that has been described in the context of first-order terms does not generalise to the higher-order case; *i.e.* there are (higher-order) disagreement sets that have more than one unifier none of which may be obtained from yet another unifier by composition with a substitution. Despite these characteristics of the problem, it has been shown that a systematic search can be made for unifiers for a given disagreement set, and it is this aspect that concerns us at the moment.

Huet, in [22], describes a procedure for determining the existence of unifiers for a given disagreement set and shows that, whenever unifiers exist, the procedure can be used to provide some of the unifiers for the set. The basis for this procedure is the fact that there are disagreement sets of a certain kind for which at least one unifier may easily be provided, and, similarly, there are disagreement sets of another kind for which it is easily manifest that no unifiers can exist. Given an arbitrary disagreement set, the procedure then attempts to reduce it to a disagreement set of one of the above kinds. The reduction proceeds by an iterative use of two kinds of simplifying functions, called SIMPL and MATCH, on disagreement sets. Since our notion of a $\mathcal{P}$-derivation uses these functions in an intrinsic way, we devote some effort to describing them below. The reader who is already familiar with the contents of [22] may, at this stage, wish to proceed to the latter half of this section.

In describing the functions SIMPL and MATCH, and in demonstrating the properties of these functions that are important in our context, we shall find a particular representation of wffs most convenient. This representation is introduced by the following definition.

3.4.1. Definition. A β-normal formula $\tilde{F} = \lambda x^1 \ldots \lambda x^n.[H\ A^1\ \ldots\ A^m]$ is said to be a *unification normal formula* if the type of H is of the form $\alpha_1 \to \cdots \to \alpha_m \to \alpha_0$ — where, of course, α_0 is an atomic type — and, for $1 \leq i \leq m$, A^i is also a unification normal formula. If F is a wff such that F λ-conv $\tilde{F}$, then $\tilde{F}$ is said to be a *unification normal form* of F.

We note that every wff has a unification normal form; such a form may be obtained by first converting the wff to a β-normal formula, and then performing a sequence of η-expansions. It may also be observed that such normal forms are unique upto a renaming of bound variables*. If F is a wff, then we shall use the notation $\tilde{F}$ to denote a unification normal form of F — it may be observed that the ambiguity inherent in this notation is inconsequential in the discussions below. Notice that if σ is a substitution, then $\sigma(F) = \sigma(\tilde{F})$. One of the reasons for representing wffs in unification normal form is that we can

* A proof of the fact that if F^1 and F^2 are unification normal formulas and F^1 λ-conv F^2 then $F^1 \equiv F^2$ may be obtained from the uniqueness of λ-normal forms by an induction on the measure on unification normal formulas that is provided in Definition 3.4.5.

analyse the effects of substitutions easily and, consequently, make decisions about unifiers.

Recalling the definitions in Section 2.2, we see that a unification normal formula of the form $\lambda x^1.\ \ldots\ \lambda x^n.[H\,A^1\,\ldots\,A^r]$ is *rigid* if H is a constant or is an element of $\{x^1,\ldots,x^n\}$ and that it is *flexible* if it is not rigid. We observe now the following lemma that is the basis of the first phase of simplification in the search for unifiers for a given disagreement set. In this lemma, and in the rest of this section, we use the notation $\mathcal{U}(\mathcal{D})$ to denote the set of unifiers for a disagreement set $\mathcal{D}$.

3.4.2 Lemma. *Let*

$$F^1 = \lambda x^1.\ \ldots\ \lambda x^n.[H^1\,A^1\,\ldots\,A^r], \quad \textit{and}$$
$$F^2 = \lambda x^1.\ \ldots\ \lambda x^n.[H^2\,B^1\,\ldots\,B^s]$$

be two rigid unification normal formulas of the same type. Then $\sigma \in \mathcal{U}(\{\langle F^1, F^2\rangle\})$ if and only if

(i) $H^1 = H^2$, *and*

(ii) $\sigma \in \mathcal{U}(\{\langle \lambda x^1.\ \ldots\ \lambda x^n.A^i, \lambda x^1.\ \ldots\ \lambda x^n.B^i\rangle \mid 1 \leq i \leq r\})$*.*

Proof. We shall show the lemma assuming that, for $1 \leq i \leq n$, $x^i \notin \mathcal{F}(\sigma)$. If this is not the case, then we obtain, by a sequence of α-conversions, from F^1 and F^2 the unification normal formulas

$$\hat{F^1} = \lambda y^1.\ \ldots\ \lambda y^n.[\hat{H^1}\,\hat{A^1}\,\ldots\,\hat{A^r}], \quad \text{and}$$
$$\hat{F^2} = \lambda y^1.\ \ldots\ \lambda y^n.[\hat{H^2}\,\hat{B^1}\,\ldots\,\hat{B^s}]$$

for which the assumption is true with respect to the y^is. But now

(a) $\hat{H^1} = \hat{H^2}$ if and only if $H^1 = H^2$,

(b) $\lambda x^1.\ \ldots\ \lambda x^n.A^i \equiv \lambda y^1.\ \ldots\ \lambda y^n.\hat{A^i}$, and

(c) $\lambda x^1.\ \ldots\ \lambda x^n.B^i \equiv \lambda y^1.\ \ldots\ \lambda y^n.\hat{B^i}$.

From these facts it follows that if the lemma is true with respect to $\hat{F^1}$ and $\hat{F^2}$, then it must also be true with respect to F^1 and F^2.

Now, given our assumption about the x^is and the facts that H^1 and H^2 are either constants or one of the variables $x^1,\ldots,x^n$, we see that

$$\sigma(F^1) \equiv \lambda norm(\lambda x^1.\ \ldots\ \lambda x^n.[H^1\,\sigma(A^1)\,\ldots\,\sigma(A^r)]) \tag{1}$$
$$\sigma(F^2) \equiv \lambda norm(\lambda x^1.\ \ldots\ \lambda x^n.[H^2\,\sigma(B^1)\ \ldots\ \sigma(B^s)]) \tag{2}$$

From this observation the lemma follows easily:

($\subset$) Let $H^1 = H^2$, and consequently $r = s$. From (ii) and our assumption about the x^is it follows easily that $\sigma(A^i) = \sigma(B^i)$ for $1 \leq i \leq r$. But then from (1) and (2) it is clear that $\sigma(F^1) = \sigma(F^2)$.

($\supset$) From (1) and (2) it is clear that

---

* If $H^1 = H^2$ it must be the case that $r = s$.

$$\sigma(F^1) \equiv \lambda x^1.\ \ldots\ \lambda x^{n-j}.[H^1\,\sigma(A^1)\,\ldots\,\sigma(A^{r-j})],$$
$$\sigma(F^2) \equiv \lambda x^1.\ \ldots\ \lambda x^{n-k}.[H^2\,\sigma(B^1)\,\ldots\,\sigma(B^{s-k})],$$

where, for $r-k < i \leq r$, $\sigma(A^i) = x^i$ and, for $k-s < i \leq s$, $\sigma(B^i) = x^i$. Since $\sigma(F^1) = \sigma(F^2)$ then it is apparent that $H^1 = H^2$. From this it also follows that $r = s$, that $j = k$, and finally, for $1 \leq i \leq r$, $\sigma(A^i) = \sigma(B^i)$. But from our assumption about the x^is, we see that (ii) must also be true. ∎

Let us say that F is rigid (flexible) just in case $\tilde{F}$ is rigid (flexible), and let us refer to the arguments of $\tilde{F}$ as the arguments of F. If F^1 and F^2 are two wffs of the same type, it is evident that unification normal forms of F^1 and F^2 must have binders of the same length. Furthermore, we may, by a sequence of α-conversions, arrange their binders to be identical. If F^1 and F^2 are both rigid, then Lemma 3.4.2 provides us a means for either determining that F^1 and F^2 have no unifiers or for reducing the problem of finding unifiers for F^1 and F^2 to that of finding unifiers for the arguments of these wffs. This, in fact, is the nature of the simplification effected on a given unification problem by the function SIMPL.

3.4.3. Definition. The function SIMPL on sets of disagreement pairs is defined as follows:

(1) If $\mathcal{D} = \emptyset$ then $\text{SIMPL}(\mathcal{D}) = \emptyset$.

(2) If $\mathcal{D} = \{\langle F^1, F^2\rangle\}$, and

  (a) if F^1 is a flexible wff then $\text{SIMPL}(\mathcal{D}) = \mathcal{D}$; otherwise

  (b) if F^2 is a flexible wff then $\text{SIMPL}(\mathcal{D}) = \{\langle F^2, F^1\rangle\}$;

  (c) otherwise F^1 and F^2 are both rigid wffs. Let $\lambda\bar{x}.[C^1\,A^1\ \ldots\ A^r]$ and $\lambda\bar{x}.[C^2\,B^1\ \ldots\ B^s]$ be unification normal forms for F^1 and F^2.
  If $C^1 \neq C^2$ then $\text{SIMPL}(\mathcal{D}) = \mathbf{F}$;
  otherwise $\text{SIMPL}(\mathcal{D}) = \text{SIMPL}(\{\langle \lambda\bar{x}.A^i, \lambda\bar{x}.B^i\rangle \mid 1 \leq i \leq r\})$.

(3) Otherwise $\mathcal{D}$ has at least two members. Let $\mathcal{D} = \{\langle F^i, G^i\rangle \mid 1 \leq i \leq n\}$.

  (a) If $\text{SIMPL}(\{\langle F^i, G^i\rangle\}) = \mathbf{F}$ for some i then $\text{SIMPL}(\mathcal{D}) = \mathbf{F}$;

  (b) Otherwise $\text{SIMPL}(\mathcal{D}) = \bigcup_{i=1}^{n} \text{SIMPL}(\{\langle F^i, G^i\rangle\})$.

3.4.4. Example. Let $\mathcal{D}$ be the disagreement set

```
{ ⟨[mapfun f1 nil nil],
   [mapfun f2
           [cons 1 [cons 2 nil]]
           [cons [g 1 1] [cons [g 1 2] nil]]]⟩ }.
```

where `f1` and `f2` are variables of type $int \to int$, and the other symbols are as in Example

3.1.6. Then SIMPL($\mathcal{D}$) = **F**. On the other hand, if $\mathcal{D}$ is the set

```
{ ⟨[mapfun f1 [cons x l1] [cons [f1 x] l2]],
   [mapfun f2
           [cons 1 [cons 2 nil]]
           [cons [g 1 1] [cons [g 1 2] nil]]]⟩ },
```

where `x`, `l1`, and `l2` are variables, then SIMPL($\mathcal{D}$) is the disagreement set

```
{ ⟨f1,f2⟩ , ⟨x,1⟩ , ⟨[f1 x],[g 1 1]⟩ ,
  ⟨l1,[cons 2 nil]⟩ , ⟨l2,[cons [g 1 2] nil]⟩ }.
```

It is clear from the definition of SIMPL that it transforms a given disagreement set into either the marker **F** or a disagreement set consisting solely of "flexible-flexible" or "flexible-rigid" wffs. By an abuse of terminology, we shall regard **F** as a disagreement set that has no unifiers. The intention, then, is that SIMPL transforms the given set into a simplified set that has the same unifiers. In order to show that SIMPL is true to this intention, we need the following measure on wffs.

3.4.5. Definition. Let $F = \lambda\bar{x}.[H\ A^1 \ldots A^m]$ be a formula in unification normal form. Then we define the following measure on F:

$$\xi(F) = m + \sum_{i=1}^{m} \xi(A^i).$$

We extend this measure to an arbitrary wff: Let $\tilde{F}$ be a unification normal form of a wff F. Then $\xi(F) = \xi(\tilde{F})$*.

That SIMPL is a transformation that achieves its intended purpose in a finite number of steps is the content of the following lemma.

3.4.6 Lemma. SIMPL *is a total computable function on sets of disagreement pairs. Further, if* $\mathcal{D}$ *is a set of disagreement pairs then* $\sigma \in \mathcal{U}(\mathcal{D})$ *if and only if* SIMPL($\mathcal{D}$) $\neq$ **F** *and* $\sigma \in \mathcal{U}$(SIMPL($\mathcal{D}$)).

Proof. We define a measure ψ on sets of disagreement pairs in the following fashion. If $\mathcal{D} = \{\langle F^i, G^i\rangle \mid 1 \leq i \leq n\}$, then

$$\psi(\mathcal{D}) = n + \sum_{i=1}^{n} \xi(F^i).$$

Using Lemma 3.4.2, an inductive argument based on the measure of a set verifies the lemma. ∎

The first phase in the process of finding unifiers for a given disagreement set $\mathcal{D}$ thus consists of evaluating SIMPL($\mathcal{D}$). If the result of this is **F**, we see that $\mathcal{D}$ has no unifiers.

* It is easily verified that if F^1 and F^2 are unification normal formulas and $F^1 \equiv F^2$ then $\xi(F^1) = \xi(F^2)$. That ξ is well-defined on arbitrary wffs is then a consequence of the uniqueness of unification normal forms upto α-conversions.

On the other hand, if the result is a set that is either empty or has only flexible-flexible pairs, at least one unifier can be provided easily for the set as we shall see in the proof of Theorem 3.4.18; for obvious reasons, therefore, we refer to such a set as a *solved set*. If the set has at least one flexible-rigid pair, then the function MATCH is used to progress the search for a unifier further.

3.4.7. Definition. Let $\mathcal{V}$ be a set of variables, let F^1 be a flexible wff, let F^2 be a rigid wff of the same type as F^1, and let $\lambda\bar{x}.[f\ A^1\ \ldots\ A^r]$, and $\lambda\bar{x}.[C\ B^1\ \ldots\ B^s]$ be unification normal forms of F^1 and F^2. Further, let the type of f be $\alpha_1 \to \cdots \to \alpha_r \to \beta$, and, for $1 \leq i \leq r$, let w^i be a variable$_{\alpha_i}$.

(i) if C is a variable (*i.e.* C appears in $\bar{x}$), then $\mathrm{IMIT}(F^1, F^2, \mathcal{V}) = \emptyset$;

otherwise, letting $h^i \notin \mathcal{V} \cup \{w^1, \ldots, w^r\}$ be variables of suitable types for $1 \leq i \leq s$, $\mathrm{IMIT}(F^1, F^2, \mathcal{V}) = \{\{\langle f, \lambda w^1.\ \ldots\ \lambda w^r.[C\,[h^1\,w^1\,\ldots\,w^r]\ldots[h^s\,w^1\,\ldots\,w^r]]\rangle\}\}$.

(ii) for $1 \leq i \leq r$,

if α_i is not of the form $\beta_1 \to \ldots \to \beta_t \to \beta$ then $\mathrm{PROJ}_i(F^1, F^2, \mathcal{V}) = \emptyset$;

otherwise, letting $h^i \notin \mathcal{V} \cup \{w^1, \ldots, w^r\}$ be variables of suitable types for $1 \leq i \leq t$, $\mathrm{PROJ}_i(F^1, F^2, \mathcal{V}) = \{\{\langle f, \lambda w^1.\ \ldots\ \lambda w^r.[w^i\,[h^1\,w^1\,\ldots\,w^r]\ldots[h^t\,w^1\,\ldots\,w^r]]\rangle\}\}$.

(iii) $\mathrm{MATCH}(F^1, F^2, \mathcal{V}) = \mathrm{IMIT}(F^1, F^2, \mathcal{V}) \cup \bigcup_{1 \leq i \leq r} \mathrm{PROJ}_i(F^1, F^2, \mathcal{V})$

3.4.8. Example. Let F^1 be the wff `[f1 x]`, and F^2 be the wff `[g 1 1]`, where `f1`, `x`, and `g` are as in Example 3.4.4, and let `w` be a variable of type *int*. Further, let `h1`, `h2` be two different variables of type $int \to int$ that are also distinct from `f1`. Then

(i) $\mathrm{IMIT}(F^1, F^2, \{\mathtt{f1},\mathtt{x}\}) = \{\{\langle \mathtt{f1}, \lambda \mathtt{w}.\mathtt{[g\ [h1\ w]\ [h2\ w]]}\rangle\}\}$,

(ii) $\mathrm{PROJ}_1(F^1, F^2, \{\mathtt{f1},\mathtt{x}\}) = \{\{\langle \mathtt{f1}, \lambda \mathtt{w}.\mathtt{w}\rangle\}\}$, and

(iii) $\mathrm{MATCH}((F^1, F^2, \{\mathtt{f1},\mathtt{x}\}) = \{\{\langle \mathtt{f1}, \lambda \mathtt{w}.\mathtt{[g\ [h1\ w]\ [h2\ w]]}\rangle\}, \{\langle \mathtt{f1}, \lambda \mathtt{w}.\mathtt{w}\rangle\}\}$.

The intuitive picture behind MATCH is that it suggests ways in which the flexible wff may be made to resemble the rigid one. One way in which this may be achieved is by making its head "imitate" that of the rigid wff. Indeed, in the context of first-order terms this is the only way in which the two terms may be made the same. If our wffs are higher-order ones, however, the end may also be achieved by "projecting" one of the arguments of the flexible wff out as the head and then getting the resulting term to resemble the rigid one. It is this aspect of higher-order terms that causes the search for a unifier to be a branching search.

The purpose of MATCH, thus, is to suggest a set of substitutions that may form "initial segments" of unifiers and, in this process, bring the search for a unifier closer to resolution. In order to show that MATCH achieves this purpose, we introduce first the following measure on substitutions:

3.4.9. Definition. Let $\varphi = \{\langle f^i, T^i\rangle \mid 1 \leq i \leq n\}$ be a substitution. Then we define a measure on φ as follows

$$\pi(\varphi) = n + \sum_{i=1}^{n} \xi(T^i).$$

The following lemma now demonstrates the correctness of MATCH.

3.4.10 Lemma. *Let $\mathcal{V}$ be a set of variables, let F^1 be a flexible wff and let F^2 be a rigid wff of the same type as F^1. If there is a substitution $\sigma \in \mathcal{U}(\{\langle F^1, F^2\rangle\})$ then there is a substitution $\varphi \in \mathrm{MATCH}(F^1, F^2, \mathcal{V})$ and a corresponding substitution σ' such that*

(i) $\sigma =_\nu \sigma' \circ \varphi$, *and*

(ii) $\pi(\sigma') < \pi(\sigma)$.

Proof. We sketch a proof based on [22]. Let $\sigma \in \mathcal{U}(\{\langle F^1, F^2\rangle\})$, and let

$$\lambda x^1. \ \ldots \ \lambda x^n.[f\, A^1 \ \ldots \ A^r], \quad \text{and}$$
$$\lambda x^1. \ \ldots \ \lambda x^n.[C\, B^1 \ \ldots \ B^s]$$

be unification normal forms for F^1 and F^2 chosen such that, for $1 \leq i \leq n$, $x^i \notin \mathcal{F}(\sigma)$. Since $C \in \{x^1, \ldots, x^n\}$ or is C a parameter, we see that there is a j, with $0 \leq j \leq min(n, s)$, such that

$$\sigma(F^2) \equiv \lambda x^1. \ \ldots \ \lambda x^{n-j}.[C\, \sigma(B^1) \ \ldots \ \sigma(B^{s-j})].$$

Now there must be a pair $\langle f, T\rangle \in \sigma$ with T of the form

$$\lambda z^1. \ \ldots \ \lambda z^{r-k}.[H\, D^1 \ \ldots \ D^t]$$

(with, of course, $0 \leq k \leq r$) where either $H \in \{z^1, \ldots, z^{r-k}\}$ or $H = C$; if neither of these conditions is true we see that there is an $i \leq n$ such that

$$\sigma(F^1) \equiv \lambda x^1. \ \ldots \ \lambda x^i.[H' \ldots],$$

where H', being either f or H, is distinct from C and consequently $\sigma(F^1) \neq \sigma(F^2)$.

Let us assume $H \notin \{z^1, \ldots, z^{r-k}\}$. Then $H = C$, and, given our choice of x^is, C must be a parameter. But then $\mathrm{IMIT}(F^1, F^2, \mathcal{V})$ contains a substitution and we let this be φ. By our definition,

$$\varphi = \{\langle f, \lambda w^1. \ \ldots \ \lambda w^r.[C\, [h^1\, w^1 \ \ldots \ w^r] \ \ldots \ [h^s\, w^1 \ \ldots \ w^r]]\rangle\},$$

where for $1 \leq i \leq s$, $h^i \notin \mathcal{V} \cup \{w^1, \ldots, w^r\}$. Now for a suitable choice of variables $z^{r-k+1}, \ldots, z^r$, we see that T has

$$\lambda z^1. \ \ldots \ \lambda z^r.[C\, \tilde{D}^1 \ \ldots \ \tilde{D}^t\, z^{r-k+1} \ \ldots \ z^r]$$

as a unification normal form; we note also that in this case $t + k = s$. Letting

$$\begin{aligned} \delta \ = \ & \{\langle h^i, \lambda z^1. \ \ldots \ \lambda z^r.D^i\rangle \mid 1 \leq i \leq t\} \cup \\ & \{\langle h^{t+i}, \lambda z^1. \ \ldots \ \lambda z^r.z^{r-k+i}\rangle \mid 1 \leq i \leq k\}, \end{aligned}$$

we define $\sigma' = ((\sigma \uparrow \mathcal{V}) - \{\langle f, T\rangle\}) \cup \delta$. It is easily verified that σ' is a substitution such that $\sigma =_\nu \sigma' \circ \varphi$ and $\pi(\sigma') = \pi(\sigma \uparrow \mathcal{V}) - 1 < \pi(\sigma)$.

If $H = z^i$ for $1 \leq i \leq r - k$, we use $\text{PROJ}_i(F^1, F^2, \mathcal{V})$ in a similar fashion to exhibit a φ and a σ' that satisfies (i) and (ii).

As a final comment, we note that the lemma may, in a certain sense, be strengthened. Let $\langle f, T^1\rangle, \langle f, T^2\rangle \in \text{MATCH}(F^1, F^2, \mathcal{V})$ be two distinct substitutions. An examination of the definition of MATCH assures us that there can be no substitutions δ and δ' such that $\delta(T^1) = \delta'(T^2)$. Thus, in the case that $f \in \mathcal{V}$ we see that corresponding to each σ there is exactly one φ for which the lemma is true. ∎

A unification procedure may now be described based on an iterative use of SIMPL and MATCH. A procedure that searches for a $\mathcal{P}$-derivation, a notion that we describe next, may actually be looked upon as a generalisation of this unification procedure.

$\mathcal{P}$-Derivations. With the above understanding of the unification problem, we are now in a position to introduce the notion of a $\mathcal{P}$-derivation and to study some of the properties of this notion. Let the symbols $\mathcal{G}$, $\mathcal{D}$, θ and $\mathcal{V}$, perhaps with subscripts, denote sets of wffs_o, disagreement sets, substitutions and sets of variables, respectively. We then define the relation of being "$\mathcal{P}$-derived from" between tuples of the form $\langle \mathcal{G}, \mathcal{D}, \theta, \mathcal{V}\rangle$ that is basic to the definition of a $\mathcal{P}$-derivation in the following manner.

3.4.11. Definition. Let $\mathcal{P}$ be a set of definite sentences. We say a tuple $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2, \mathcal{V}_2\rangle$ is *$\mathcal{P}$-derived* from the tuple $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1\rangle$ if $\mathcal{D}_1 \neq \mathbf{F}$ and, in addition, one of the following situations holds:

(1) (*Goal reduction step*) $\theta_2 = \emptyset$, $\mathcal{D}_2 = \mathcal{D}_1$, and there is a goal formula $G \in \mathcal{G}_1$ such that
   (a) G is $\top$ and $\mathcal{G}_2 = \mathcal{G}_1 - \{G\}$ and $\mathcal{V}_2 = \mathcal{V}_1$, or
   (b) G is $G^1 \wedge G^2$ and $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G^1, G^2\}$ and $\mathcal{V}_2 = \mathcal{V}_1$, or
   (c) G is $G^1 \vee G^2$ and, for $i = 1$ or $i = 2$, $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G^i\}$ and $\mathcal{V}_2 = \mathcal{V}_1$, or
   (d) G is ΣP and for some variable $y \notin \mathcal{V}_1$ it is the case that $\mathcal{V}_2 = \mathcal{V}_1 \cup \{y\}$ and $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{\lambda norm(P\, y)\}$.

(2) (*Backchaining step*) Let $G \in \mathcal{G}_1$ be a rigid positive atom, and let $D \in \mathcal{P}$ be such that $D \equiv \forall x^1. \ldots . \forall x^n . G' \supset A$ for some sequence of variables $x^1, \ldots, x^n$ for which no $x^i \in \mathcal{V}_1$. Then $\theta_2 = \emptyset$, $\mathcal{V}_2 = \mathcal{V}_1 \cup \{x^1, \ldots, x^n\}$, $\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G'\}$, and $\mathcal{D}_2 = \text{SIMPL}(\mathcal{D}_1 \cup \{\langle G, A\rangle\})$.

(3) *(Unification step)* $\mathcal{D}_1$ is not a solved set and for some flexible-rigid pair $\langle F^1, F^2\rangle \in \mathcal{D}_1$, either $\text{MATCH}(F^1, F^2, \mathcal{V}_1) = \emptyset$ and $\mathcal{D}_2 = \mathbf{F}$, or there is a $\sigma \in \text{MATCH}(F^1, F^2, \mathcal{V}_1)$ and it is the case that $\theta_2 = \sigma$, $\mathcal{G}_2 = \sigma(\mathcal{G}_1)$, $\mathcal{D}_2 = \text{SIMPL}(\sigma(\mathcal{D}_1))$, and, if $\sigma = \{\langle x, T\rangle\}$, $\mathcal{V}_2 = \mathcal{V}_1 \cup \mathcal{F}(T)$.

Let us call a finite set of goal formulas a *goal set*, and a disagreement set that is **F** or consists solely of pairs of positive wffs a *positive disagreement set*. If $\mathcal{G}_1$ is a goal set and $\mathcal{D}_1$ is a positive disagreement set then it is clear, from an inspection of the above definition, the definitions 3.4.3 and 3.4.7, and the fact that a positive formula remains a positive formula under a positive substitution, that $\mathcal{G}_2$ is a goal set and $\mathcal{D}_2$ a positive disagreement set for any tuple $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2, \mathcal{V}_2 \rangle$ that is $\mathcal{P}$-derived from $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1 \rangle$.

3.4.12. Definition. Let $\mathcal{G}$ be a goal set. Then we say that a sequence $\langle \mathcal{G}_i, \mathcal{D}_i, \theta_i, \mathcal{V}_i \rangle_{1 \leq i \leq n}$ is a $\mathcal{P}$-derivation sequence for $\mathcal{G}$ just in case $\mathcal{G}_1 = \mathcal{G}$, $\mathcal{V}_1 = \mathcal{F}(\mathcal{G}_1)$, $\mathcal{D}_1 = \emptyset$, $\theta_1 = \emptyset$, and, for $1 \leq i < n$, $\langle \mathcal{G}_{i+1}, \mathcal{D}_{i+1}, \theta_{i+1}, \mathcal{V}_{i+1} \rangle$ is $\mathcal{P}$-derived from $\langle \mathcal{G}_i, \mathcal{D}_i, \theta_i, \mathcal{V}_i \rangle$.

From our earlier observations, and an easy induction on the length of the sequence, it is clear that in a $\mathcal{P}$-derivation sequence for a goal set $\mathcal{G}$ each $\mathcal{G}_i$ is a goal set and each $\mathcal{D}_i$ is a positive disagreement set. We make implicit use of this observation in our discussions below. In particular, we intend unqualified uses of the symbols $\mathcal{G}$ and $\mathcal{D}$ to be read as syntactic variables for goal sets and positive disagreement sets respectively.

A $\mathcal{P}$-derivation sequence $\langle \mathcal{G}_i, \mathcal{D}_i, \theta_i, \mathcal{V}_i \rangle_{1 \leq i \leq n}$ terminates, *i.e.* is not contained in a longer sequence, if

(a) $\mathcal{G}_n$ is either empty or is a goal set consisting solely of flexible atoms and $\mathcal{D}_n$ is either empty or consists solely of flexible-flexible pairs, or

(b) $\mathcal{D}_n = \mathbf{F}$.

In the former case we say that it is a *successfully terminated* sequence.

3.4.13. Definition. A $\mathcal{P}$-derivation sequence, $\langle \mathcal{G}_i, \mathcal{D}_i, \theta_i, \mathcal{V}_i \rangle_{1 \leq i \leq n}$, for $\mathcal{G}$ that is a successfully terminated sequence is called a *$\mathcal{P}$-derivation of $\mathcal{G}$* and $\theta_n \circ \cdots \circ \theta_1$ is called its *answer substitution*. If $\mathcal{G} = \{G\}$ then we also say that the sequence is a $\mathcal{P}$-derivation of G.

3.4.14. Example. Let $\mathcal{P}$ be the set of definite sentences in Example 3.1.6. Further, let `f1` be a variable of type $int \to int$ and let G be the goal formula

```
[mapfun f1
        [cons 1 [cons 2 nil]]
        [cons [g 1 1] [cons [g 1 2] nil]]].
```

Then the tuple $\langle \mathcal{G}_1, \mathcal{D}_1, \emptyset, \mathcal{V}_1 \rangle$ is $\mathcal{P}$-derived from $\langle \{G\}, \emptyset, \emptyset, \{\texttt{f1}\} \rangle$ by a backchaining step, if

$\mathcal{V}_1 = \{\texttt{f1}, \texttt{f2}, \texttt{l1}, \texttt{l2}, \texttt{x}\}$,
$\mathcal{G}_1 = \{\texttt{[mapfun f2 l1 l2]}\}$, and
$\mathcal{D}_1 = \{$ $\langle$`f1`,`f2`$\rangle$, $\langle$`x`,`1`$\rangle$, $\langle$`[f1 x]`,`[g 1 1]`$\rangle$,
$\langle$`l1`,`[cons 2 nil]`$\rangle$, $\langle$`l2`,`[cons [g 1 2] nil]`$\rangle$ $\}$;

of course, `f2`, `l1`, `l2`, and `x` are variables here. Similarly, if

$\mathcal{V}_2 = \mathcal{V}_1 \cup \{\texttt{h1,h2}\}$,

$\mathcal{G}_2 = \{$`[mapfun f2 l1 l2]`$\}$,
$\theta_2 = \{\ \langle$`f1,`λ`w.[g [h1 w] [h2 w]]`$\rangle\ \}$, and
$\mathcal{D}_2 = \{\ \langle$`l1,[cons 2 nil]`$\rangle$, $\langle$`l2,[cons [g 1 2] nil]`$\rangle$, $\langle$`x,1`$\rangle$,
$\langle$`[h1 x],1`$\rangle$, $\langle$`[h2 x],1`$\rangle$, $\langle$`f2,`λ`w.[g [h1 w] [h2 w]]`$\rangle$ $\}$,

then the tuple $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2, \mathcal{V}_2 \rangle$ is $\mathcal{P}$-derived from $\langle \mathcal{G}_1, \mathcal{D}_1, \emptyset, \mathcal{V}_1 \rangle$ by a unification step by picking the flexible-rigid pair $\langle$`[f1 x],[g 1 1]`$\rangle$ from $\mathcal{D}_1$ and using the substitution provided by MATCH in Example 3.4.8. If we picked the substitution provided by PROJ_1 instead, we would obtain the tuple $\langle \mathcal{G}_2, \mathbf{F}, \{\langle$`f1,`$\lambda$`w.w`$\rangle\}, \mathcal{V}_1 \rangle$.

There are several $\mathcal{P}$-derivations of G, and all of them have the same answer substitution: $\{\langle$`f1,`λ`w.[g w 1]`$\rangle\}$.

3.4.15. Example. Let $\mathcal{P}$ be a set of definite sentences that contains the definite sentence $\forall x.[[x\,A] \supset [P\,A]]$, where P and A are parameters of type $int \to o$ and int, respectively. Then, the following (sequence of) tuples constitute a $\mathcal{P}$-derivation of $[P\,A]$:

$\langle \{[P\,A]\}, \emptyset, \emptyset, \emptyset \rangle$, $\langle \{[x\,A]\}, \emptyset, \emptyset, \{x\} \rangle$.

Notice that this is a successfully terminated sequence, even though the final goal set contains a flexible atom. We shall see, in Theorem 3.4.18, that a goal set that contains only flexible atoms can be "solved" rather easily. In this particular case, for instance, the final goal set can be solved by applying the substitution $\{\langle x, \lambda y.\top \rangle\}$ to it.

As mentioned at the beginning of this section, a $\mathcal{P}$-derivation of G is intended to be a syntactic object that demonstrates that a proof in $\mathcal{T}^*$ exists for G from $\mathcal{P}$. Our next endeavour, culminating in the Theorems 3.4.18 and 3.4.22, is to show that this notion is true to our intention. In the process, we shall see that a $\mathcal{P}$-derivation of G encodes enough information to make it possible to extract the result of a computation. We shall also observe some properties of $\mathcal{P}$-derivations that are of interest from the perspective of constructing a procedure that searches for such a derivation of a goal formula.

3.4.16. Definition. Let F be a wff. Then a *ground instance* of F is a closed wff F' such that, for some substitution φ, $F' = \varphi(F)$. F' is a *positive ground instance* of F if in addition F' is a positive wff.

It is apparent that F' is a ground instance F if and only if there is a closed substitution φ for the free variables of F such that $F' = \varphi(F)$; if F' is a positive ground instance, then there must be such a φ that is also a positive substitution.

3.4.17 Lemma. *Let $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2, \mathcal{V}_2 \rangle$ be $\mathcal{P}$-derived from $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1 \rangle$, and let $\mathcal{D}_2 \neq \mathbf{F}$. Further let $\sigma \in \mathcal{U}(\mathcal{D}_2)$ be a positive substitution such that for each positive ground instance G' of a wff in $\sigma(\mathcal{G}_2)$ it is the case that $\mathcal{P} \vdash_{\mathcal{T}^*} G'$. Then*

(i) $\sigma \circ \theta_2 \in \mathcal{U}(\mathcal{D}_1)$, and

(ii) $\mathcal{P} \vdash_{\mathcal{T}^*} G$ *for each* G *that is a positive ground instance of a wff in* $\sigma(\mathcal{G}_1)$.

Proof. The lemma is proved by considering the cases in Definition 3.4.12.

A goal reduction or a backchaining step. In these cases $\theta_2 = \emptyset$ and so $\sigma \circ \theta_2 = \sigma$. Further, in a goal reduction step $\mathcal{D}_2 = \mathcal{D}_1$, and in a backchaining step $\mathcal{D}_1 \subseteq \mathcal{D}_2$. From these observations, (i) is evidently true. What remains to be shown, then, is that if $G \in \mathcal{G}_1$ and δ is a closed positive substitution for the free variables of $\sigma(G)$ then $\mathcal{P} \vdash_{\mathcal{T}^*} \delta \circ \sigma(G)$. In the cases when G is also an element of $\mathcal{G}_2$, this follows directly from the assumption of the lemma. Consequently, we only need to consider the case when $G \notin \mathcal{G}_2$. If G is $\top$, then the claim is apparent from the fact that $\vdash_{\mathcal{T}^*} \top$. If G is $G^1 \vee G^2$, from the fact that

$$\delta \circ \sigma(G) = \delta \circ \sigma(G^1) \vee \delta \circ \sigma(G^2),$$

it follows that $\delta \circ \sigma(G^1)$ and $\delta \circ \sigma(G^2)$ are closed goal formulas and are, therefore, positive ground instances of $\sigma(G^1)$ and $\sigma(G^2)$. Since either $\sigma(G^1)$ or $\sigma(G^2)$ is an element of $\sigma(\mathcal{G}_2)$, the claim follows from the assumptions and Theorem 3.3.8. A similar argument may be provided for the case when G is $G^1 \wedge G^2$.

For the remaining cases, we need a substitution that is parameterised by a sequence of variables. Let c_α be an arbitrary parameter of type α. If $\bar{y}$ is a sequence of variables, then we define $\delta_{\bar{y}}$ as follows:

$$\delta_{\bar{y}} = \{\langle x_\alpha, c_\alpha \rangle \mid y^i \text{ is an element of } \bar{y} \text{ and } x_\alpha \in \mathcal{F}(\delta \circ \sigma(y^i))\}.$$

It is apparent that $\delta_{\bar{y}} \circ \delta \circ \sigma(F)$ is a closed positive formula for any positive formula F all of whose free variables are in the sequence $\bar{y}$.

Let us now consider the case when G is $\Sigma\, P$. Since $\delta \circ \sigma(G) = \Sigma\, \delta \circ \sigma(P)$, it follows that $\delta \circ \sigma(P)$ is a closed positive formula. From Definition 3.4.12, we see that there is a variable y such that $\sigma(\lambda norm(P\, y)) \in \sigma(\mathcal{G}_2)$. Now

$$\delta_y \circ \delta \circ \sigma(\lambda norm(P\, y))$$

is evidently a positive ground instance and so, by our assumption,

$$\mathcal{P} \vdash_{\mathcal{T}^*} \delta_y \circ \delta \circ \sigma(\lambda norm(P\, y)).$$

But letting $P' = \delta \circ \sigma(P)$ and $c = \delta_y \circ \sigma \circ \delta(y)$, we see that this wff is also a λ-normal form of $P'\, c$. Hence by Theorem 3.3.8 it follows that $\mathcal{P} \vdash_{\mathcal{T}^*} \Sigma\, P'$, *i.e.* $\mathcal{P} \vdash_{\mathcal{T}^*} \delta \circ \sigma(G)$.

The only other case is that corresponding to a backchaining step. From Definition 3.4.12 and Lemma 3.4.6 we see that there is a $D \in \mathcal{P}$ such that

$$D \equiv \forall \bar{x}.G' \supset A, \quad G' \in \mathcal{G}_2, \quad \text{and} \quad \sigma(G) = \sigma(A).$$

Now let $G'' = \delta_{\bar{x}} \circ \delta \circ \sigma(G')$. Since all the free variables of G' are in the sequence $\bar{x}$, G'' is apparently a positive ground instance of $\sigma(G')$, and so, by our assumption, $\mathcal{P} \vdash_{\mathcal{T}^*} G''$. Since

$$\varphi(G' \supset A) = \varphi(G') \supset \varphi(A)$$

for any substitution φ, and since $\delta \circ \sigma(G)$ is a closed positive formula, it follows easily that

$$G'' \supset \delta \circ \sigma(G) \in |\mathcal{P}|.$$

But then, Theorem 3.3.8 assures us of the truth of the claim.

A unification step. We note first that $\mathcal{D}_2 \neq \mathbf{F}$. Hence, in either of these cases, it follows from Lemma 3.4.6 that if $\sigma \in \mathcal{U}(\mathcal{D}_2)$ then $\sigma \in \mathcal{U}(\theta_2(\mathcal{D}_1))$. But then, it is easy to see that $\sigma \circ \theta_2 \in \mathcal{U}(\mathcal{D}_1)$. Since $\mathcal{G}_2 = \theta_2(\mathcal{G}_1)$ it is evident that every ground instance of a goal formula in $\sigma \circ \theta_2(\mathcal{G}_1)$ is also a ground instance of a goal formula in $\sigma(\mathcal{G}_2)$. From this the second part of the lemma is obvious. ∎

3.4.18 Theorem. (Soundness of $\mathcal{P}$-derivations) *Let $\langle \mathcal{G}_i, \mathcal{D}_i, \theta_i, \mathcal{V}_i \rangle_{1 \leq i \leq n}$ be a $\mathcal{P}$-derivation of G, and let θ be its answer substitution. Then there is a positive substitution such that*

(i) $\sigma \in \mathcal{U}(\mathcal{D}_n)$, and

(ii) $\mathcal{P} \vdash_{\mathcal{T}^*} G'$ for every ground instance G' of a goal formula in $\sigma(\mathcal{G}_n)$.

Further, for every positive substitution σ that satisfies (i) and (ii), it is the case that $\mathcal{P} \vdash_{\mathcal{T}^*} G'$ for every ground instance G' of $\sigma \circ \theta(G)$.

Proof. The second part of the theorem follows easily from Lemma 3.4.17 and a backward induction on i, the index of each tuple in the given $\mathcal{P}$-derivation sequence. For the first part we first exhibit a substitution — that is a simple modification of the one in Lemma 3.5 in [22] — and then show that it satisfies the requirements.

Let $h_\alpha \in \mathcal{V}ar_\alpha$ be a chosen variable for each atomic type α. Then for each type α we define the wff $\hat{E}_\alpha$ in the following fashion:

(a) If α is o, then $\hat{E}_\alpha = \top$.

(b) If α is an atomic type other than o, then $\hat{E}_\alpha = h_\alpha$.

(c) If α is the function type $\beta_1 \to \cdots \to \beta_k \to \beta$ where β is an atomic type, then

$$\hat{E}_\alpha = \lambda x^1_{\beta_1}. \ \ldots \lambda x^k_{\beta_k}.\hat{E}_\beta,$$

where, for $1 \leq i \leq k$, $x^i_{\beta_i}$ is a variable such that $x^i_{\beta_i} \neq h_{\beta_i}$*.

Now let $\gamma = \{\langle y_\alpha, \hat{E}_\alpha \rangle \mid y_\alpha \in \mathcal{V}ar_\alpha\}$. Finally, letting $\mathcal{V} = \mathcal{F}(\mathcal{G}_n) \cup \mathcal{F}(\mathcal{D}_n)$, we define $\sigma = \gamma \uparrow \mathcal{V}$.

We note that if there are any goal formulas in $\mathcal{G}_n$, then they are all of the form $[P\, C^1 \ldots C^n]$ where P is a variable whose type is of the form $\alpha_1 \to \cdots \to \alpha_n \to o$. From this it is apparent that if $G \in \mathcal{G}_n$ then any ground instance of $\sigma(G)$ is identical to $\top$. Thus, it is clear that σ satisfies (ii). If $\mathcal{D}_n$ is empty then $\sigma \in \mathcal{U}(\mathcal{D}_n)$. Otherwise, let $\langle F^1, F^2 \rangle \in \mathcal{D}_n$. Since F^1 and F^2 are two flexible wffs, it may be seen that $\sigma(F^1)$ and $\sigma(F^2)$ are of the form†

---

* The particular choice of $x^i_{\beta_i}$ is unimportant so long as the constraint that $x^i_{\beta_i} \neq h_{\beta_i}$ is met. However, in order to make the wff $\hat{E}_\alpha$ described above unique, it is necessary to fix the choice of $x^i_{\beta_i}$. We implicitly assume this qualification in the definition.

† The subscripts on the ys are not intended to be read as types.

$$\lambda y_1^1 \ldots . \lambda y_1^{m_1} . \hat{E}_{\beta_1}, \quad \text{and}$$
$$\lambda y_2^1 \ldots . \lambda y_2^{m_2} . \hat{E}_{\beta_2},$$

where β_i is a primitive type and $\hat{E}_{\beta_i} \notin \{y_i^1, \ldots, y_i^{m_i}\}$ for $i = 1, 2$. Since F^1 and F^2 have the same types and substitution is a type preserving mapping, it is clear that $\beta_1 = \beta_2$, $m_1 = m_2$ and y_1^i and y_2^i are variables of the same type. But then evidently $\sigma(F^1) = \sigma(F^2)$. ∎

We now desire to show a converse of the above theorem. For this purpose, we shall need the following observation that follows immediately from an inspection of Definition 3.4.12.

3.4.19 Lemma. *Let $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2, \mathcal{V}_2 \rangle$ be $\mathcal{P}$-*derived *from $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1 \rangle$ and let $\mathcal{D} \neq \mathbf{F}$. Then $\mathcal{V}_1 \subseteq \mathcal{V}_2$ and if $\mathcal{F}(\mathcal{G}_1) \cup \mathcal{F}(\mathcal{D}_1) \subseteq \mathcal{V}_1$, then $\mathcal{F}(\mathcal{G}_2) \cup \mathcal{F}(\mathcal{D}_2) \subseteq \mathcal{V}_2$.*

We also need a measure of complexity corresponding to a goal set and a unifier. In defining such a measure, we use those introduced in Definitions 3.3.6 and 3.4.9.

3.4.20. Definition.

(i) Let $\mathcal{G}$ be a set of closed goal formulas. Then $\nu_{\mathcal{P}}(\mathcal{G}) = \sum_{G \in \mathcal{G}} \mu_{\mathcal{P}}(G)$.

(ii) Let $\mathcal{G}$ be a set of goal formulas and let σ be a positive substitution such that each wff in $\sigma(\mathcal{G})$ is closed. Then $\kappa_{\mathcal{P}}(\mathcal{G}, \sigma) = \langle \nu_{\mathcal{P}}(\sigma(\mathcal{G})), \pi(\sigma) \rangle$.

(iii) Let $\prec$ be the lexicographic ordering on the collection of pairs of natural numbers; *i.e.* $\langle m_1, n_1 \rangle \prec \langle m_2, n_2 \rangle$ if either $m_1 < m_2$ or $m_1 = m_2$ and $n_1 < n_2$.

If $\mathcal{G}$ is a finite set of closed goal formulas such that $\mathcal{P} \vdash_{\mathcal{T}^*} G$ for each $G \in \mathcal{G}$, then it is easily seen that $\nu_{\mathcal{P}}(\mathcal{G}) < \omega$. We make implicit use of this fact in the proof of the following Lemma.

3.4.21 Lemma. *Let $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1 \rangle$ be a tuple that is not a terminated $\mathcal{P}$-derivation sequence and for which $\mathcal{F}(\mathcal{G}_1) \cup \mathcal{F}(\mathcal{D}_1) \subseteq \mathcal{V}_1$. If there is a positive substitution $\sigma_1 \in \mathcal{U}(\mathcal{D}_1)$ such that $\sigma_1(G^1)$ is a closed goal formula and $\mathcal{P} \vdash_{\mathcal{T}^*} \sigma_1(G^1)$ for each $G^1 \in \mathcal{G}_1$, then there is a tuple $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2, \mathcal{V}_2 \rangle$ that may be $\mathcal{P}$-*derived *from $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1 \rangle$ and there is a positive substitution $\sigma_2 \in \mathcal{U}(\mathcal{D}_2)$ such that*

(i) $\sigma_1 =_{\mathcal{V}_1} \sigma_2 \circ \theta_2$,

(ii) for each $G^2 \in \mathcal{G}_2$, $\sigma_2(G^2)$ is a closed goal formula such that $\mathcal{P} \vdash_{\mathcal{T}^*} \sigma_2(G^2)$, and

(iii) $\kappa_{\mathcal{P}}(\mathcal{G}_2, \sigma_2) \prec \kappa_{\mathcal{P}}(\mathcal{G}_1, \sigma_1)$.

*Furthermore, when several tuples may be $\mathcal{P}$-*derived *from $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1 \rangle$, then there exist a tuple and a corresponding positive substitution σ_2 that satisfy the conditions (i), (ii), and (iii) regardless of the choice that is exercised in deciding (1) the of the kind of step, (2) the goal formula in a goal reduction or backchaining step, and (3) the flexible-rigid pair in a unification step.*

Proof. Since $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1 \rangle$ is not a terminated $\mathcal{P}$-derivation sequence, it is clear that there must be a tuple $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2, \mathcal{V}_2 \rangle$ that may be $\mathcal{P}$-derived from it. We consider by cases the steps by which such a tuple may result and show that, in each of these cases, there is a tuple and a substitution that satisfies the requirements of the lemma. From our argument it will be evident that such a tuple exists regardless of the choices mentioned in the lemma.

Goal reduction step. If the second tuple is to be obtained by one of these cases, it must be that $\mathcal{D}_2 = \mathcal{D}_1$ and $\theta_2 = \emptyset$.

If one of the subcases 1(a) – 1(c) is applicable, then let $\sigma_2 = \sigma_1$. It is obvious that $\sigma_2 \in \mathcal{U}(\mathcal{D}_2)$, and, since $\sigma_1 = \sigma_2 \circ \theta$, that $\sigma_1 =_{\mathcal{V}_1} \sigma_2 \circ \theta$. Now let $\mathcal{V}_2 = \mathcal{V}_1$ and, by considering each of the subcases 1(a) – 1(c), we demonstrate a value for $\mathcal{G}_2$ that satisfies the remaining requirements of the lemma.

(a) If $\top \in \mathcal{G}_1$, then let $\mathcal{G}_2 = \mathcal{G}_1 - \{\top\}$. The tuple $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2, \mathcal{V}_2 \rangle$ that is described by these various assignments is obviously $\mathcal{P}$-derived from the given tuple. From the assumptions of the lemma, it also follows that this tuple and σ_2 satisfy (ii). That

$$\kappa_{\mathcal{P}}(\mathcal{G}_2, \sigma_2) \prec \kappa_{\mathcal{P}}(\mathcal{G}_1, \sigma_1)$$

follows from the fact that $\mu_{\mathcal{P}}(G) > 0$ for any closed goal formula G.

(b) Let $G^1 \vee G^2 \in \mathcal{G}_1$. Since

$$\sigma_1(G^1 \vee G^2) = \sigma_1(G^1) \vee \sigma_1(G^2),$$

it follows that $\sigma_1(G^1)$ and $\sigma_1(G^2)$ are closed goal formulas. From Lemma 3.3.7 we see that, for $i = 1$ or $i = 2$,

$$\mu_{\mathcal{P}}(\sigma_1(G^i)) < \mu_{\mathcal{P}}(\sigma_1(G^1 \vee G^2))$$

and so also $\mathcal{P} \vdash_{\mathcal{T}^*} \sigma_1(G^i)$. Without loss of generality, we may assume that this is true for $i = 1$. Thus, if we let

$$\mathcal{G}_2 = (\mathcal{G}_1 - \{G^1 \vee G^2\}) \cup \{G^1\},$$

we see that we have a tuple that is $\mathcal{P}$-derived from the given one and that together with σ_2 satisfies (ii) and (iii).

(c) Let $G^1 \wedge G^2 \in \mathcal{G}_1$. In this case we let

$$\mathcal{G}_2 = (\mathcal{G}_1 - \{G^1 \wedge G^2\}) \cup \{G^1, G^2\}.$$

By an argument similar to that in (b) we may see that we have then defined a tuple that together with σ_2 satisfies the remaining requirements of the lemma.

The only remaining subcase to consider is 1(d). If this subcase is applicable, then there is wff_o such that $\Sigma\, P \in \mathcal{G}_1$. In this event, let

$$\mathcal{G}_2 = (\mathcal{G}_1 - \{\Sigma\, P\}) \cup \{\lambda norm(P\, y)\}$$

for some $y \notin \mathcal{V}_1$ and let $\mathcal{V}_2 = \mathcal{V}_1 \cup \{y\}$. If we let $P' = \sigma_1(P)$, it follows, from our assumptions and the fact that

$$\sigma_1(\Sigma\, P) = \Sigma\, P',$$

that P' is a closed positive formula and that $\mathcal{P} \vdash_{\mathcal{T}^*} \Sigma\, P'$. Lemma 3.3.7 now assures us of the existence of a closed positive formula C such that

$$\mu_{\mathcal{P}}(\lambda norm(P'\, C)) < \mu_{\mathcal{P}}(\Sigma\, P').$$

Letting

$$\sigma_2 = \sigma_1 \circ \{\langle y, C\rangle\},$$

we see that the requirements of the lemma are met: Since $y \notin \mathcal{V}_1$ and $\mathcal{F}(\mathcal{D}_1) \subseteq \mathcal{V}_1$, it is clear that $\sigma_2 \in \mathcal{U}(\mathcal{D}_2)$ and that $\sigma_1 =_{\mathcal{V}_1} \sigma_2$. Since $\mathcal{F}(\mathcal{G}_1) \subseteq \mathcal{V}_1$, it is also apparent that (ii) is true for each $G \in \mathcal{G}_2$ that is also in $\mathcal{G}_1$. For the only other $G \in \mathcal{G}_2$, *i.e.* $\lambda norm(P\, y)$, we see that $\sigma_2(G)$ is a λ-normal form of $[P'\, C]$ and so (ii) is obviously true. Finally, (iii) is true since our choice of C is such that

$$\mu_{\mathcal{P}}(\lambda norm(P'\, C)) < \mu_{\mathcal{P}}(\sigma_1(\Sigma\, P)).$$

Backchaining step. If such a step is applicable, there is a rigid positive atom $G \in \mathcal{G}_1$. Let $G^a = \sigma_1(G)$. By our assumptions, G^a is a closed rigid positive atom and $\mathcal{P} \vdash_{\mathcal{T}^*} G^a$. From Lemma 3.3.7 it follows that there is a wff $G'' \supset G^a \in |\mathcal{P}|$ such that

$$\mu_{\mathcal{P}}(G'') < \mu_{\mathcal{P}}(G^a)$$

and, therefore, also such that $\mathcal{P} \vdash_{\mathcal{T}^*} G''$. From this it is easily seen that there is a $D \in \mathcal{P}$ such that

$$D \equiv \forall x^1 \ldots \forall x^n . G' \supset A,$$

where $x^1, \ldots, x^n$ are chosen ensuring that $x^i \notin \mathcal{V}_1$, and that there is a closed positive substitution φ for $\{x^1, \ldots, x^n\}$ such that $G^a = \varphi(A)$ and $G'' = \varphi(G')$. Now if we let

$$\begin{aligned}
&\theta_2 = \emptyset,\\
&\mathcal{V}_2 = \mathcal{V}_1 \cup \{x^1, \ldots, x^n\},\\
&\mathcal{G}_2 = (\mathcal{G}_1 - \{G\}) \cup \{G'\}, \quad \text{and}\\
&\mathcal{D}_2 = \text{SIMPL}(\mathcal{D}_1 \cup (\{\langle G, A\rangle\}),
\end{aligned}$$

we obtain a tuple $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2, \mathcal{V}_2\rangle$ that is $\mathcal{P}$-derived from $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1\rangle$. Letting $\sigma_2 = \sigma_1 \circ \varphi$, we see that this tuple and σ_2 also meet the requirements of the lemma: By our choice of x^is and the fact that

$$\mathcal{F}(\mathcal{G}_1) \cup \mathcal{F}(\mathcal{D}_1) \subseteq \mathcal{V}_1,$$

it follows that

$$\begin{aligned}
&\sigma_1 =_{\mathcal{V}_1} \sigma_2,\\
&\sigma_2 \in \mathcal{U}(\mathcal{D}_1), \quad \text{and}\\
&\sigma_2(G^1) = \sigma_1(G^1)
\end{aligned}$$

for every $G^1 \in \mathcal{G}_1$. We also see that

$$\begin{aligned}
&\sigma_2(A) = \varphi(A), \quad \text{and}\\
&\sigma_2(G') = \varphi(G') = G''.
\end{aligned}$$

Using Lemma 3.4.6, it is now clear that σ_2 is a unifier for $\mathcal{D}_2$. Similarly, it is also clear that $\sigma_2(G^2)$ is a closed goal formula such that $\mathcal{P} \vdash_{\mathcal{T}^*} \sigma_2(G^2)$ for every $G^2 \in \mathcal{G}_2$. Finally, since $\mu_{\mathcal{P}}(G'') < \mu_{\mathcal{P}}(G^a)$ and $\sigma^2(G) = G^a$, we see that

$$\kappa_{\mathcal{P}}(\mathcal{G}_2, \sigma_2) \prec \kappa_{\mathcal{P}}(\mathcal{G}_1, \sigma_1).$$

Unification step. If this case is applicable, then there is evidently a flexible-rigid pair in $\mathcal{D}_1$. Let $\langle F^1, F^2 \rangle$ be an arbitrary such pair. By Lemma 3.4.10 there is a (positive) substitution

$$\varphi \in \mathrm{MATCH}(F^1, F^2, \mathcal{V}_1)$$

and another (positive) substitution δ such that

$$\sigma_1 =_{\mathcal{V}_1} \delta \circ \varphi \text{ and } \pi(\delta) < \pi(\sigma_1).$$

But now by letting

$$\begin{aligned} \theta_2 &= \varphi, \\ \mathcal{G}_2 &= \varphi(\mathcal{G}_1), \quad \text{and} \\ \mathcal{D}_2 &= \mathrm{SIMPL}(\varphi(\mathcal{D}_1)), \end{aligned}$$

and by choosing $\mathcal{V}_2$ appropriately, we see that there is a tuple $\langle \mathcal{G}_2, \mathcal{D}_2, \theta_2, \mathcal{V}_2 \rangle$ that may be $\mathcal{P}$-derived from $\langle \mathcal{G}_1, \mathcal{D}_1, \theta_1, \mathcal{V}_1 \rangle$. Letting $\sigma_2 = \delta$ we see easily that the other requirements of the lemma are also satisfied: Since $\mathcal{F}(\mathcal{D}_1) \subseteq \mathcal{V}_1$ it is clear that

$$\sigma_1(\mathcal{D}_1) = \sigma_2 \circ \theta_2(\mathcal{D}_1).$$

Since $\sigma_1 \in \mathcal{U}(\mathcal{D}_1)$, it follows that

$$\sigma_2 \circ \theta_2 \in \mathcal{U}(\mathcal{D}_1)$$

and, using Lemma 3.4.6, that $\sigma_2 \in \mathcal{U}(\mathcal{D}_2)$. Since $\mathcal{F}(\mathcal{G}_1) \subseteq \mathcal{V}_1$, it is apparent that

$$\sigma_1(\mathcal{G}_1) = \sigma_2(\mathcal{G}_2).$$

That every $G^2 \in \sigma_2(\mathcal{G}_2)$ is a closed goal formula such that $\mathcal{P} \vdash_{\mathcal{T}^*} G^2$ now follows trivially from the assumptions. Finally

$$\kappa_{\mathcal{P}}(\mathcal{G}_2, \sigma_2) \prec \kappa_{\mathcal{P}}(\mathcal{G}_1, \sigma_1)$$

since $\nu_{\mathcal{P}}(\sigma_2(\mathcal{G}_2)) = \nu_{\mathcal{P}}(\sigma_1(\mathcal{G}_1))$ and $\pi(\sigma_2) < \pi(\sigma_1)$. ∎

3.4.22 Theorem. (Completeness of $\mathcal{P}$-derivations) *Let φ be a closed positive substitution for the free variables of G such that $\mathcal{P} \vdash_{\mathcal{T}^*} \varphi(G)$. Then there is a $\mathcal{P}$-derivation of G with an answer substitution θ such that $\varphi \preceq_{\mathcal{F}(G)} \theta$.*

Proof. From Lemmas 3.4.21 and 3.4.19 and the assumption of the theorem, it is evident that there is a $\mathcal{P}$-derivation sequence $\langle \mathcal{G}_i, \mathcal{D}_i, \theta_i, \mathcal{V}_i \rangle_{1 \leq i}$ for $\{G\}$ and a sequence of substitutions σ_i such that

(i) $\sigma_1 = \varphi$,

(ii) σ_{i+1} satisfies the equation $\sigma_i =_{\mathcal{V}_i} \sigma_{i+1} \circ \theta_{i+1}$,

(iii) $\sigma_i \in \mathcal{U}(\mathcal{D}_i)$, and

(iv) $\kappa_{\mathcal{P}}(\mathcal{G}_{i+1}, \sigma_{i+1}) \prec \kappa_{\mathcal{P}}(\mathcal{G}_i, \sigma_i)$.

From (iv) and the definition of $\prec$ it is clear that the sequence must terminate. From (iii) and Lemmas 3.4.6 and 3.4.10 it is evident, then, that it must be a successfully terminated sequence, *i.e.* a $\mathcal{P}$-derivation of G. Using (i), (ii) and Lemma 3.4.19, an induction on the length n of the sequence then reveals that $\varphi \preceq_{\mathcal{V}_1} \theta_n \circ \cdots \circ \theta_1$. But $\mathcal{F}(G) = \mathcal{V}_1$ and $\theta_n \circ \cdots \circ \theta_1$ is the answer substitution for the sequence. ∎

$\mathcal{P}$-derivations, thus, provide us the basis for describing the proof procedure that we desired at the outset. Given a goal formula G, such a procedure starts with the tuple $\langle \{G\}, \emptyset, \emptyset, \mathcal{F}(G) \rangle$ and constructs a $\mathcal{P}$-derivation sequence. If the procedure performs an exhaustive search, and if there is a proof of G from $\mathcal{P}$, the procedure will always succeed in constructing a $\mathcal{P}$-derivation of G from which a result may be extracted. A breadth-first search may be inappropriate if the procedure is intended as an interpreter for a programming language based on our definite sentences. By virtue of Lemma 3.4.21, we see that there are certain cases in which the procedure may limit its choices without adverse effects. The following choices are, however, critical:

(i) Choice of definite sentence in a backchaining step, and

(ii) Choice of substitution in a unification step.

When it encounters such choices, the procedure may, with an accompanying loss of completeness, perform a depth-first search with backtracking. The particular manner in which to exercise these choices is very much an empirical question and might be settled, if at all, by experimentation.

Chapter 4
The Logic Programming Language λProlog

Our focus in the preceding parts of this thesis have largely been on analysing the logical basis of a logic programming language that incorporates higher-order notions. We now turn to the practical concerns regarding a language that is based on this analysis. In this chapter we consider an experimental programming system called λProlog that, in its current conception, is based largely on the logic of definite sentences studied in the preceding chapter. It is to be noted, however, that λProlog is a system that is yet evolving; it is, in this respect, best thought of as a vehicle for experimenting with the usefulness, in the logic programming context, of notions that have been found valuable in other paradigms of programming. To provide one example, it is intended that an ability to structure logic programs be provided in this system based on the extension to the logic of definite clauses described in [28]. Our concerns in this chapter, however, are oriented primarily towards exposing the higher-order features of λProlog and towards discussing the issues relevant to the implementation of a logic programming language that incorporates such features. These concerns are adequately served by restricting our view of the system to the way it is currently conceived, and it is this stance that we adopt in this chapter.

In the first section of this chapter we describe the syntax of λProlog. The language of λProlog may, in a certain sense, be looked upon as a typed, higher-order version of that of Prolog. The main difference in the syntax of the two languages is that types must be associated with every λProlog expression, and we introduce the notion of a *type declaration* through which such an association is achieved. Our experience has revealed that the notion of types provided by the underlying logic is unfortunately too restrictive in practice. We have tried to deal with this issue by introducing the notion of a *type variable* into the system. We discuss, in this section, some of the simplifications that this provides in the presentation of programs and also some of the complications that this might lead to in interpreting programs.

The next two sections focus on the higher-order features of λProlog. In Section 4.2 we outline the nature of predicate and function variables in this system. This discussion also serves to illustrate the value of predicate variables in a logic programming context. The truly novel features of λProlog — and hence also the features whose usefulness is less tangible — are the provision of function variables and the use of λ-terms as data structures. In Section 4.3 we illustrate the advantages these features offer in describing operations on objects that are intrinsically higher-order in nature. The specific tasks that we consider here involve operations on two kinds of objects: formulas and programs. It is our contention here that the data structures of λProlog permit us to represent, and to describe manipulations on, these kinds of objects in a logically perspicuous manner. The understanding of higher-order terms that is manifest in λProlog through λ-conversion

and higher-order unification then allows such descriptions to serve as means for actually performing the necessary manipulations.

In the last section of this chapter, we turn to the issues involved in implementing an interpreter for λProlog. In this section we describe an experimental interpreter that we have designed for this system. This interpreter is akin to the standard ones for Prolog with the difference that our use of higher-order unification requires certain additional choices to be made. We describe the nature of these choices and also relate certain features that we have found useful to incorporate into the interpreter based on our experiences with an implementation. The provision of type variables also introduces certain complications (discussed in Section 4.1) in designing an interpreter and we discuss some techniques that we have adopted for dealing with these in an implementation. It is to be noted that all the examples of λProlog programs in this thesis have been tested with the interpreter described in Section 4.4.

Section 1: The Language of λProlog

As we have noted already, λProlog is a logic programming language that is based on the logic of higher-order definite sentences. Given the similarities observed in Section 3.1 between the logics underlying the two systems, we have found it convenient to adopt several features of the syntax of Prolog [11] in λProlog. These features include the following.

(i) Tokens that correspond to constants and variables in λProlog are identified in much the same way as in Prolog, *i.e.* they are sequences of alphanumeric characters or sequences of 'sign' characters. Since it is necessary to distinguish between those tokens in wffs that correspond to variables and those that are to be construed as constants, we use the convention, again borrowed from Prolog, that those tokens which begin with capital letters stand for variables, and all other tokens represent constants*.

(ii) We have reserved the symbols `,`, `;`, and `:-` to represent the logical constants $\wedge$, $\vee$ and $\supset$ respectively. The last of these constants may appear only as the top-level logical connective in a definite sentence. In keeping with the procedural nature of definite sentences, we have chosen to write the formulas that contain this constant backwards, *i.e.* the formula $G \supset A$ is written in λProlog as the procedure A `:-` G. In the special case when G is the constant $\top$, this last λProlog expression may be abbreviated to A.

(iii) In rendering definite sentences into λProlog expressions, the outermost universal quan-

* Our current implementation includes one "built-in" sort other than the sort `o` that corresponds to propositions, and this is the sort `int` corresponding to the integers. Thus λProlog does make an additional distinction between tokens corresponding to constants in that a token consisting solely of numeric characters is recognised as a constant of type `int`.

tifications are omitted. In keeping with this convention, variables that occur free in expressions representing definite sentences are understood to be implicitly universally quantified and the scope of this quantification is taken to be the entire expression.

There are, however, a few differences between the syntax of λProlog and Prolog that are necessitated by the richer nature of the underlying logic. The higher-order language includes the constant $\top$ and the existential quantifier Σ, and we have reserved the constant symbols `true` and `sigma`* in λProlog to represent these. Similarly, there is a need to represent the operation of abstraction, and we have reserved the infix symbol `\` for this purpose: thus the wff $\lambda X.F$ is rendered into λProlog as $X\backslash F$. We have also found it convenient to import the curried notation of the logical language into λProlog and to use the meta-linguistic conventions, discussed in Chapter 2, for omitting brackets. When brackets are necessary to change the application order in expressions in λProlog, these are represented by parentheses. The following set of 'procedure' declarations that define the append relation between three lists illustrate the use of this notation and also serve to expose the similarities between the syntax of Prolog and λProlog:

```
append nil L L.
append (cons X L1) L2 (cons X L3) :- append L1 L2 L3.
```

It is to be noted, however, that the above declarations do not completely specify a λProlog program, since our logical language requires that a type be associated with every constant and variable symbol. In order to achieve this objective, we have introduced what we call *type declarations*. These are declarations that have the format

```
type  <list-of-tokens>  <type>.
```

The effect of such a declaration is to identify `<type>` as the type of each of the constant or variable symbols in `<list-of-tokens>`. Using declarations of this kind, a complete specification of λProlog procedures to append lists would then be the following†; we observe here that the function type constructor is represented in λProlog by the symbol `->`.

```
type   X   int.
type   nil,L,L1,L2,L3    (list int).
type   cons   int -> (list int) -> (list int).
type   append   (list int) -> (list int) -> (list int) -> o.

append nil L L.
```

---

* Anticipating our discussions later in this section, the type expression associated with the λProlog constant `sigma` is `(A -> o) -> o`.

† It is often useful to be able to restrict the scope of type and procedure declarations, and λProlog provides such a facility through a preliminary notion of a *module*. We do not discuss this matter here, however, since our purpose in this chapter is adequately served by viewing a program as containing only one set of declarations.

```
append (cons X L1) L2 (cons X L3) :- append L1 L2 L3.
```

λProlog programs may also contain *operator declarations*, that serve the purpose of overriding the default prefix application precedence in a manner exactly analogous to that in Prolog. An example of such a declaration is

```
infix  225  xfy  &.
```

This declaration corresponds to the declaration `op(225,xfy,&)` in Prolog [11]; it defines the constant `&` as an infix operator that is right associative and has priority 225.

Type Variables and Type Inference. An examination of the program that appears above reveals to us that the declarations provided therein serve to define only the "data structure" for lists of integers. Thus, if we identify the constants `bob`, `sue`, etc. as being of type `person`, it is clear that we need to use an analogous set of declarations to provide the notion of a list of these kinds of objects.

Such a course is clearly unsatisfactory from a programming point of view since it leads to a proliferation of declarations, and to a complete separation between notions that at some level have a common flavour. An immediate solution to the above problem is to collapse the type distinctions between terms that have a atomic type. Thus we may agree by convention that the only type other than `o` is `i`, and then deem that both `bob` and `3` are of type `i`; in a sense this is the course adopted in Prolog. This does not turn out to be a very happy solution to the given problem, since it may be desirable to maintain distinctions between terms of different atomic types. Consider, for instance, the procedure declarations

```
sum_of_list nil 0.
sum_of_list (cons X L) N :- sum_of_list L N1, add X N1 N.
```

where `add` is defined to be a predicate that performs addition on integers. We do wish to be able to say that these declarations define the predicate `sum_of_list` that is of type `(list int) -> int -> o`, and that they cannot be used to solve the query

```
sum_of_list (cons 1 (cons bob nil)) N,
```

even before such an attempt is made.

Even if we were to accept the suggested solution to the above problem, we notice that it is certainly not a solution if what we desire are representations for lists of terms of each function type. The distinction between the types `i -> i` and `i`, for instance, is intrinsic to the language on which λProlog is based.

The solution that we have adopted to the problem is to permit a use of the same syntactic device that was employed in our discussions about the logical system, namely to allow variables to occur in types. An illustration of this approach is provided by the following revised version of the program defining lists; we note that the tokens that begin

with capital letters in the types in the type declarations here are to be construed as variables.

```
type   X   A.
type   nil,L,L1,L2,L3    (list A).
type   cons   A -> (list A) -> (list A).
type   append   (list A) -> (list A) -> (list A) -> o.

append nil L L.
append (cons X L1) L2 (cons X L3) :- append L1 L2 L3.
```

A type declaration in which variables occur in the type is to be understood in the following fashion: It represents an infinite number of declarations each of which is obtained by substituting, in a uniform manner, closed types for the variables that occur in the type. For instance, the type declaration

```
type cons A -> (list A) -> (list A).
```

represents, amongst others, the type declarations

```
type cons int -> (list int) -> (list int).
type cons (int -> int) -> (list (int -> int)) -> (list (int -> int)).
```

By virtue of the above provision, we see that variables may occur in the types corresponding to the constants and variables that appear in a "procedure declaration." Such a "procedure declaration" is, again, to be thought of as a schema that represents an infinite set of procedure declarations in which no type variables appear. Each member of this set is obtained by substituting closed types for the type variables that occur in the schema. Such a substitution is, of course, constrained by the fact that the resulting instance must be a representation of a definite sentence that is well-formed. We assume, in addition, that the free variables having the same name in a formula are identical — in particular, they have identical types. Thus given the procedure declaration schema (although strictly necessary, we shall often drop this last qualification if no confusion arises as a result)

```
append (cons X L1) L2 (cons X L3) :- append L1 L2 L3.
```

where the types associated with the first and second occurrences of `X` are `A` and `B` respectively and the type associated with the first occurrence of `cons` is `C -> (list C) -> (list C)`, the same types must replace `A`, `B` and `C` in all the permissible instances of this schema.

It is tempting to conclude that our use of type variables actually provides for an enriched class of procedure declarations in which an (implicit) universal quantification is provided over type variables that occur in each such declaration. Such a view, however, is misleading. Given the logical analysis in Chapter 3, it is clear that a procedure declaration in which type variables appear is only correctly interpreted as a schema: It represents a class of definite sentences each of which is obtained by instantiating every type variable

by a closed type, and it is only one of these definite sentences that may be used in the construction of a proof for a goal formula. The implication of this is that the invocation of a procedure declaration must also be accompanied by the determination of the type instance to be used. It is often possible to delay the choice of the particular type instance at run-time until such a time that it can be uniquely determined, but there are situations where the use of type variables causes problems. We shall comment on these aspects in Section 4.4.

The introduction of type variables enables us to remove much of the burden of providing type declarations from the user. The needed type declarations may often be inferred by using the methods described by [29] in the context of ML [17]. The idea is that the requirement that type instances of a procedure declaration be well-formed places constraints on the types that may be associated with the constants and variables that appear in it. Additional constraints are provided by our assumption that the type associated with each occurrence of a constant, whose type is not explicitly declared, within a program must be the same. These constraints may actually be formulated as a system of equations involving first-order terms. A most general solution to this system of equations may be obtained by using the notion of (first-order) unification [37] and we use this solution to provide a set of implicit type declarations. Using such a scheme, we note that from the type declaration

```
type  cons  A -> (list A) -> (list A).
```

all the other type declarations in the program defining lists may be inferred. Even this declaration is not necessary, since without it the type declaration

```
type  cons  A -> B -> B.
```

would be inferred. Providing it, however, enables unintended uses of the procedure declarations in the program to be excluded.

While the syntax of types and the methods of inferring the types associated with terms in λProlog bears a resemblance to that used in the context of functional programming languages (specifically ML), it is important to point out that there is a difference with regard to the role that they play in the overall system. The primary purpose of typing in ML is to provide a partial assurance of the correctness of a program and is in a certain sense orthogonal to the evaluation mechanism: As [29] argues, a program that can be well-typed is semantically sound. Looked at operationally this means that, for example, "an integer is never added to a truth value or applied to an argument, and consequently need not carry its type around for run-time checking." In contrast, in our context, types are an integral part of the underlying language. While they do play a role in the early detection of failed computations*, as illustrated in the `sum_of_list` example, they also play an important

* A polymorphic type system for Prolog is described in [32] and it is argued that "well-typed programs do not go wrong." The semantic basis for the notion of "wrong" in the

role in the evaluation mechanism, specifically in higher-order unification. Consequently, they do not constitute a component that is dispensible at "run-time."

Section 2: The Nature of Function and Predicate Variables

From the discussions earlier in this thesis, it is clear that one of the main differences between Prolog and λProlog is in the higher-order features that the latter provides. One of the sources of this difference is that the logic of higher-order definite sentences permits a quantification over function variables. It is possible, as a consequence, to write procedures in λProlog that take functions as arguments. An illustration of this facility is provided by the following set of procedure declarations that define the operation of mapping a function over a list:

```
mapfun F (cons X L1) (cons (F X) L2) :- mapfun F L1 L2.
mapfun F nil nil.
```

The predicate `mapfun` that is defined by these declarations may be used with its first and second arguments instantiated. When used in this manner it will produce the list that results from applying the first argument to each element of the second argument. To take a concrete example, the query

```
mapfun X\(g a X) (cons a (cons b nil)) L,
```

would produce, as an answer, the substitution `(cons (g a a) (cons (g a b) nil))` for `L`. Answering this query would require an interpreter for λProlog to form the terms `((X\(g a X)) a)` and `((X\(g a X)) b)` and then reduce these terms using the rules of λ-conversion.

The logical considerations underlying λProlog actually permit functions to be treated as first-class, logic programming variables. The implication of this observation is that the values of function variables may in fact be *computed* through unification. An illustration of this particular role of function variables is provided in the context of the above declarations by the query

```
(mapfun F
        (cons a (cons b nil))
        (cons (g a a) (cons (g a b) nil)).
```

There is precisely one answer for this query, namely the substitution `X\(g a X)` for `F`, and an interpreter for λProlog would find this answer. A point to note here is that the search for such "higher-order" substitutions may involve making choices between substitutions, and this adds an extra degree of non-determinism to the programming paradigm. To appreciate

---

context of logic programming is, however, somewhat unclear. It seems more appropriate, following [30], to think of types in the logic programming context as providing an early indication of computations that will not succeed.

the need for a non-deterministic search, consider an interpreter for λProlog that functions in a manner similar to the standard depth-first (unification-first) interpreters for Prolog. In attempting to answer the above query, such an interpreter would first need to consider the task of unifying `(F a)` and `(g a a)`. There are four terms that may be substituted for `F` that satisfy this requirement, and these are listed below:

```
X\(g X X),  X\(g a X),  X\(g X a),  X\(g a a).
```

If the interpreter picked any of these other than the second, this choice would lead to a failure when an attempt is made at a later stage to unify `(F b)` and `(g a b)`. The interpreter would at this stage have to backtrack over its earlier choice of substitution.

The above discussions illustrate that λProlog does embody a genuine understanding of the notion of a function. It is important to note, however, that the understanding of this notion that is embodied in λProlog is fairly limited, since the class of functions that is representable using typed λ-terms is actually a very simple class. To appreciate the sense in which this understanding is limited, let us consider the following query in the context of the declarations defining the predicate `mapfun`.

```
mapfun F (cons a (cons b nil))  (cons c (cons d nil)).
```

There are several functions that in a sense 'satisfy' this query, namely each function that maps `a` to `c` and `b` to `d`. However, *none* of these functions is representable using our λ-terms and, consequently, any interpreter for λProlog would fail in answering this query. Conversely, it is not possible to use `mapfun` to produce the list `(cons c (cons d nil))` from the list `(cons a (cons b nil))`. In a more constructive sense, it is precisely this limitation in the representational ability of our λ-terms that makes the determination of function values through unification a reasonable computational operation. Furthermore, while it is clear that what the use of these terms provides us with is largely a notion of substitution through λ-conversion and a mechanism for "pattern-matching" through unification, these operations are more powerful than the corresponding operations available through the use of first-order terms. There are several situations where this enhanced power is useful, and we shall see some instances of these situations in the next section.

In the example above we have considered a function variable that is not also a predicate variable. Although there is not much difference between predicates and other functions from a purely logical point of view, variables of the former sort may play quite a different role in a logic programming language from non-predicate variables. The reason for this is that predicate variables may appear both *inside* the terms of a goal as well as the *head* of a goal, *i.e.* they may appear intensionally and extensionally. When they appear intensionally, their values are determined through unification just like those of any other kind of variable. When they appear extensionally, they possess the facet that they might be "executed."

To illustrate this dual nature predicate variables, let us consider the following set of

procedure declarations that define the operation of mapping a predicate over a list.

```
mappred P nil nil.
mappred P (cons X L1) (cons Y L2) :- P X Y, mappred P L1 L2.
```

In the second declaration above, we note that the predicate variable `P` appears both intensionally and extensionally. If this procedure is invoked with its first argument instantiated, then the extensional occurrence of `P` would actually lead to the evaluation of the term instantiating this argument. To make this picture more concrete, let us assume that we are given the following additional set of declarations that define the ages of various individuals,

```
age bob 23.
age sue 24.
age ned 23.
```

and now consider the following query:

```
mappred (X\Y\(age X Y))
        (cons ned (cons bob (cons sue nil))
        L.
```

This query essentially asks for the ages of the individuals `ned`, `bob` and `sue`. Evaluating this query would lead to an instantiation of the predicate argument, `P`, of `mappred` with the term `(X\Y\(age X Y))`. Using this instantiation and an appropriate instantiation for `X`, a new goal would be formed out of the term `(P X Y)` and this goal would then be evaluated to determine a value for `Y`. Thus the goals `(age ned Y)`, `(age bob Y)`, and `(age sue Y)` would each be evaluated, and the values found for `Y` as a result would go to form the substitution `(cons 23 (cons 23 (cons 24 nil)))` for `L`. It is clear that the mapping operation defined by `mappred` is thus much stronger than that provided by `mapfun` above, since the evaluation of the goal `(P X Y)` may require *arbitrary* computations to be performed and is not restricted only to λ-conversions. Indeed, the reader familiar with a language like Lisp would recognise the similarities between the use of a predicate variable in this example and the use of apply and lambda terms in Lisp. The value of predicate variables in writing procedures that "abstract over" other procedures should also be fairly clear from this example. It is, in fact, a recognition of this value that has lead to the introduction of extra-logical predicates like `call` and `univ` into Prolog. The advantage of λProlog in this respect is that it provides this feature in a more general and theoretically well-understood fashion.

In the example considered above, the λ-term that was considered as an instantiation for the predicate variable contained no logical connectives or quantifiers. Our logical analysis does permit these terms to contain the operations of conjunction, disjunction and existential quantification, and the use of these operations might have value in certain situations. To consider a particular example of the use of this feature, let us assume that we

are given the following set of procedure declarations

```
sublist P (cons X L) (cons X K) :- P X, sublist P L K.
sublist P (cons X L) K :- sublist P L K.
sublist P nil nil.
```

that define the relation `sublist` between a predicate and two lists. This relation may be understood in the following fashion: if `P`, `L` and `K` are closed terms, then `(sublist P L K)` holds if `K` is a sublist of `L` all of whose members satisfy the property expressed by the predicate `P`. Let us suppose now that we wish to define a predicate `have_age` which is such that `(have_age L K)` is provable if `K` is a sublist of the individuals in `L` whose ages are defined. This predicate may be defined by the following declaration that uses `sublist`:

```
have_age L K :- sublist Z\(sigma X\(age Z X)) L K.
```

The point to note in this definition is the use of a predicate term that contains an explicit quantifier to instantiate the predicate argument of `sublist`; this is the term `(Z\(sigma X\(age Z X)))` that corresponds to $\lambda z\ \exists x\ age(z, x)$ in our logical language and that describes the set of individuals that have an age. Omitting the quantifier from this term, as in the definition of `same_age` below,

```
same_age L K :- sublist Z\(age Z A) L K.
```

defines a predicate with a different property: `(same_age L K)` is true only when all the objects in `K` also have the same age.

The uses of predicate variables in the cases considered above were such that whenever a variable of this sort appeared as the head of a goal, it could always be completely instantiated before the goal had to be invoked. There are cases, however, when the value for the variable head of a goal cannot be fully determined before the goal is to be invoked, and this value, in fact, needs to be computed through an evaluation. The search for such values, however, is fairly unconstrained and there is, consequently, a question as to what value should be returned. To appreciate the unconstrained nature of this search, consider the query

```
P bob 23.
```

One answer to this query, in the context of the declarations that appear earlier in this section, is of course the substitution `(X\Y\(age X Y))` for `P`. There are, however, many other substitutions that also serve this purpose. For instance, if `G` is any closed query that can be satisfied given our procedure declarations, then any substitution of the form `X\Y\G` for `P` may logically be construed as an answer. Clearly there are far too many answers to this query, and the query is, in this sense, an ill-posed one. In deciding what kind of an answer to provide in such cases, we have help from our discussions in Chapter 3. By virtue of our analysis there, we have seen that there is one trivial substitution that always

works in such cases. This substitution, in the example above, is the one that replaces `P` with `X\Y\true`. While this substitution is not very informative, it has the virtue of at least being correct. Furthermore, it is perhaps a better strategy to provide only such a trivial substitution as an answer rather than to embark on an unconstrained search to an ill-posed query. The experimental interpreter that we describe in Section 4.4 is one that adopts this strategy.

It is important to note that picking such a substitution does not necessarily trivialise the kinds of values that can be found for predicate variables. If a predicate variable occurs intensionally as well as extensionally in a goal, then this kind of a trivial substitution may not be a solution. To see this, consider the following set of declarations that are a rendition into the syntax of λProlog of the definite sentences in Example 3.1.7.

```
primrel mother.
primrel wife.
rel R :- primrel R.
rel X\Y\ (sigma Z\(R X Z , S Z Y)) :- primrel R , primrel S.
mother jane mary.
wife john jane.
```

In the context of these sentences, the query

```
rel R, R john mary,
```

is satisfiable only if the substitution `X\Y\(sigma Z\(wife X Z, mother Z Y))` is made for `R`. Notice that answering this query actually requires finding a substitution for a predicate variable — the query asks for a relation, in the sense of `rel`, between `john` and `mary`. The choice of such relations is, however, well constrained by `rel` and therefore the query does beget a meaningful answer.

Section 3: λ-terms as Data Structures

The preceding section examined the nature of predicate and function variables in λProlog. It is hoped that the value of predicate variables in a logic programming language is also clear from this discussion. The truly novel and, in our opinion, the most valuable aspect of λProlog is its use of λ-terms as data structures. There are several kinds of objects whose representation in a manner closely corresponding to their meaning requires the use of a higher-order term language. Examples of objects of this kind are provided by logical formulas and programs. The ability to reason about such objects is also important in several situations, *e.g.* in constructing proof systems or program transformation systems. The task of constructing such reasoning systems is, we believe, greatly facilitated by the use of a programming paradigm that provides natural ways to represent these kinds of objects, and also provides mechanisms for manipulating such representations. The logic programming

paradigm possesses two characteristics that makes it particularly suitable in this respect: The manner in which it supports the notion of search, and the ability that it provides to examine the intension — or, the manner of description — of objects through unification. The use of logic programming in implementing formula or program manipulating systems has, however, been limited, primarily because languages in this paradigm have traditionally been based on first-order logic. By combining a richer representation language, namely, that of typed λ-terms with the basic computational machinery of logic programming we believe that λProlog overcomes this difficulty and, by doing so, opens the way for several new applications for the paradigm of logic programming.

While a complete justification of the above observations is beyond the scope of this thesis, we provide, in this section, some preliminary indications of the usefulness of the enriched term structures of λProlog, and of the potential promise of a language like λProlog in general. We do so by demonstrating how logical formulas and programs may be naturally represented by λ-terms, and by showing how such representations simplify the task of performing certain manipulations on them.

Representing and Manipulating Logical Expressions. The term structures of λProlog contain at least one enrichment over first-order terms in that they incorporate the notion of λ-abstraction. This operation is useful whenever there is a desire to represent objects that involve the concept of a variable being bound over the scope of a sub-expression. A situation of this sort arises, for instance, when there is a need to represent first-order formulas as objects that are to be manipulated by programs. This aspect is brought out clearly if we consider the task of representing the formula $\forall x \exists y (P(x, y) \supset Q(y, x))$ as a term in a logic programming language. Fragments of this formula may be encoded into first-order terms, but there is a genuine problem with representing the quantification. We need to represent the variable being quantified as a genuine variable, since logical operations (such as quantifier instantiation) may involve substituting for the variable. A correct representation, however, requires that we distinguish occurrences of the variable within the scope of the quantifier from occurrences outside of it.

The mechanism of λ-abstraction provides the tool needed to make such distinctions. To illustrate this, let us consider how the formula above may be encoded using the terms of λProlog. For this purpose we first reserve the sort `b` for the types of terms that represent first-order formulas. Further, we assume that the constants `&`, `or` and `=>`, which we shall use to represent the the logical connectives $\wedge$, $\vee$ and $\supset$, are defined to be infix operators of type `b -> b -> b`. Finally, we assume that the constants `all` and `some` are defined to be of type `(i -> b) -> b`; these two constants have the type of "generalised" quantifiers and may be used together with abstraction to represent universal and existential quantification respectively. Assuming these declarations, the formula above may be represented by the λ-term `(all X\(some Y\(p X Y => q Y X)))`.

Encoding Rules of Inference in λ-Prolog Programs. The ability to represent formulas in a manner that captures all the important logical aspects is of interest because it provides a new domain of application for logic programming languages, namely as a vehicle for implementing proof systems based on natural deduction. Consider, for instance, the following rule of inference in a sequent calculus:

$$\frac{\Gamma \longrightarrow F1 \qquad \Gamma \longrightarrow F2}{\Gamma \longrightarrow F1 \wedge F2}$$

This rule embodies a notion of search that is relevant to the construction of proofs. To be precise, it suggests that one way to construct a proof for the sequent $\Gamma \longrightarrow F1 \wedge F2$ is to construct proofs for the sequents $\Gamma \longrightarrow F1$ and $\Gamma \longrightarrow F2$. Logic programming languages provide us with a mechanism for computation that captures exactly this notion of search. Thus, assuming our representation for formulas and that the antecedent of a sequent is represented as lists of formulas, the above rule may be described by the following λProlog procedure

```
prove Gamma (F1 & F2)  :-  prove Gamma F1, prove Gamma F2.
```

Such a procedure may actually be used to search for proofs. Attempting to use it reveals at least one use for higher-order unification; the second-order term `(F1 & F2)` would have to be matched with the term that instantiates it in an invocation.

While it may be argued that much of the same advantages may already be derived from first-order term encodings of formulas, a consideration of quantifier rules alters this picture. Take, for example, the following rule in a sequent style calculus

$$\frac{P(t), \Gamma \longrightarrow F}{\forall x P(x), \Gamma \longrightarrow F}$$

where t is some term. An implementation of this rule requires the instantiation of a quantifier. Given our representation of quantification, this operation may be described rather directly as an application. The intended effect is then achieved by virtue of the rules of λ-conversion. Using this idea, the quantifier rule above may now be easily described:

```
prove (cons (all X\(P X)) Gamma) F  :-  prove (cons (P T) Gamma) F.
```

Notice that `T` is a logic variable in this procedure. In the course of constructing a proof, this `T` may be instantiated by an arbitrary term.

To provide a more complete illustration of the usefulness of a language such as λProlog in the context under discussion, let us consider the task of writing an interpreter for the logic programming language that is described in [28] and [14]. This language extends the conventional first-order logic programming language by permitting implications in goal formulas. To be precise, the definite clauses and goal formulas in this language are described

by mutual recursion in the following manner; we assume here that A, D, and G are syntactic variables for (first-order) atomic formulas, definite clauses and goal formulas respectively.

$$D ::= A \mid G \supset A \mid \forall x D, \quad \text{and}$$
$$G ::= A \mid G_1 \vee G_2 \mid G_1 \wedge G_2 \mid D \supset G \mid \exists x G.$$

A *program* in this language is a finite set of closed definite clauses, and a *query* is a closed goal formula. The relation of being "derived from" between a query G and a program $\mathcal{P}$, denoted by $\mathcal{P} \vdash_O G$, is formalised in [28] in a natural deduction framework by the following proof rules:

(i) $\mathcal{P} \vdash_O \exists x G$ if there is a closed term t such that $\mathcal{P} \vdash_O G[x/t]$.

(ii) $\mathcal{P} \vdash_O G_1 \wedge G_2$ if $\mathcal{P} \vdash_O G_1$ and $\mathcal{P} \vdash_O G_2$.

(iii) $\mathcal{P} \vdash_O G_1 \vee G_2$ if $\mathcal{P} \vdash_O G_1$ or $\mathcal{P} \vdash_O G_2$.

(iv) $\mathcal{P} \vdash_O D \supset G$ if $\mathcal{P} \cup \{D\} \vdash_O G$.

(v) $\mathcal{P} \vdash_O A$ if A is atomic and is an instance of a formula in $\mathcal{P}$.

(vi) $\mathcal{P} \vdash_O A$ if A is atomic and there is an instance $G \supset A$ of a formula in $\mathcal{P}$ such that $\mathcal{P} \vdash_O G$.

These proof rules translate rather directly into procedures in λProlog. Indeed, the following list of declarations define the predicate `interpreter` such that the goal `(interpreter Clauses Goal)` is derivable just in case `Clauses` is a list of terms that represents the formulas in a program $\mathcal{P}$, `Goal` is a term that represents a query G, and $\mathcal{P} \vdash_O G$.

```
interpreter Cl (some G) :- interpreter Cl (G T).
interpreter Cl (G1 & G2) :- interpreter Cl G1 , interpreter Cl G2.
interpreter Cl (G1 or G2) :- interpreter Cl G1 ; interpreter Cl G2.
interpreter Cl (D => G) :- interpreter (cons D Cl) G.
interpreter Cl A :- member Clause Cl, instantiate Clause A.
interpreter Cl A :- member Clause Cl, instantiate Clause (G => A),
                    interpreter Cl G.

instantiate (all P) C :- instantiate (P T) C.
instantiate C C.

member X (cons X Rest).
member X (cons Y Rest) :- member X Rest.
```

An interesting aspect of the declarations above is the manner in which the predicate `instantiate` is used repeatedly to perform universal instantiations and thus produce a new copy of the body of a clause. It is important to notice that writing an interpreter for the language under consideration really requires an explicit mechanism for variable binding since it is necessary to determine the scope of a quantifier. There is, for instance,

a distinction to be made between the two goals $\exists x(\forall y\ p(x,y) \supset q)$ and $(\forall x \forall y\ p(x,y) \supset q)$ since only the latter may be derived from the program $\{p(a,c) \wedge p(b,c) \supset q\}$. First-order terms, of course, do not provide a facility for representing variable binding. It is therefore difficult to see how an interpreter that is equivalent to the one above can be written in Prolog even with the "extra-logical" predicate `clause` [11] that captures some of the behaviour of the predicate `instantiate` above. In general a logic programming language that is based on higher-order terms, such as λProlog, does appear to be the right implementation mechanism for problems of the kind considered here.

Translating between English and Logical Forms. To provide another illustration of the usefulness of λ-terms, in particular of the idea of substitution provided by λ-conversion in their context, we consider the task of translating English sentences to logical forms. In the example that we present here, we assume a familiarity with the formalism of definite clause grammars (DCGs) as presented in [33]. We also assume an extension of this formalism to λProlog; the only real difference is that in λProlog each English word must have a type, and we assume that the type `token` is reserved for this purpose.

The problem that we wish to consider is best explicated by an example. We wish to render the English sentence

"Every man loves a woman"

to the logical form

$$\forall x(man(x) \supset \exists y(woman(y) \wedge loves(x,y))$$

which, in our context, is represented by the λ-term*

```
all X\ ((manp X) => (some Y\((womanp Y) & (lovesp X Y)))).
```

At first sight this seems a formidable problem since the accepted syntactic structures for the given English sentence bear no similarity to the syntactic structure of the logical expression. However Montague, in his work on the semantics of English (*e.g.* in [31]), has shown how such a logical form may be obtained in a simple and elegant manner. The idea is to associate terms of the typed λ-calculus with words in English, and to look upon the final expression as a result of composing these in accordance with the syntactic structure of the English sentence and then applying the rules of λ-conversion. The following DCG in λProlog that specifies the relation between a small subset of English sentences and their

* In representing the above logical form, we have used the λProlog constants `manp`, `womanp` and `lovesp`, rather than `man`, `woman` and `loves`, to encode the predicates *man*, *woman*, and *loves*, respectively. The reason for this is we need the latter constants for representing the English words "man", "woman", and "loves", and they shall, therefore, have a type that makes them inappropriate for representing predicates as well.

logical forms borrows from Montague's ideas*.

```
sentence (P1 P2)  -->  np P1, vp P2.
noun-phrase (P1 P2)  -->  determiner P1, nom P2.
noun-phrase P  -->  proper-noun P.
nom P  --> com-noun P.
nom X\((P1 X) & (P2 X))  -->  noun P1, rel-clause P2.
verb-phrase X\(P2 (P1 X))  -->  trans-verb P1, noun-phrase P2.
verb-phrase P  -->  intrans-verb P.
rel-clause P  -->  [that], vp P.
determiner P1\P2\(all X\(P1 X => P2 X))  -->  [every].
determiner P1\P2\(some X\((P1 X) & (P2 X)))  -->  [a].
com-noun manp  -->  [man].
com-noun womanp  -->  [woman].
proper-noun (P\(P j))  -->  [john].
trans-verb lovesp  -->  [loves].
intrans-verb livesp -->  [lives].
```

In this DCG we note that the type of the (first) argument of `com-noun` (*i.e.* the types of `manp` and `womanp`) is `i -> b`, of `proper-noun` is `((i -> b) -> b)`, of `trans-verb` is `i -> i -> b` and of `intrans-verb` is `i -> b`; the other types should be clear from the context.

Using this DCG to parse a sentence illustrates the usefulness of the notion of substitution provided by λ-conversion in the context of the data structures of λProlog. The point is perhaps best brought out by contrasting the DCG above with a first-order version of it that was presented in [33] and that is reproduced below.

```
sentence(P)  -->  noun-phrase(X,P1,P),verb-phrase(X,P1).
noun-phrase(X,P1,P) -->
   determiner(X,P2,P1,P) , noun(X,P3) , rel-clause(X,P3,P2).
noun-phrase(X,P,P)  -->  prop-noun(X).
verb-phrase(X,P)  -->  trans-verb(X,Y,P1) , noun-phrase(Y,P1,P).
verb-phrase(X,P)  -->  intrans-verb(X,P).
rel-clause(X,P1,(P1 & P2))  -->  [that] , verb-phrase(X,P2).
rel-clause(_,P,P)  -->  [].
determiner(X,P1,P2,all(X,(P1 => P2)))  -->  [every].
determiner(X,P1,P2,exists(X,(P1 & P2)))  -->  [a].
noun(X,man(X))  -->  [man].
noun(X,woman(X))  -->  [woman].
proper-noun(john)  -->  [john].
trans-verb(X,Y,loves(X,Y))  -->  [loves].
intrans-verb(X,lives(X))  -->  [lives].
```

* An example, similar in spirit to the one we present here, also appears in [45].

It is a fairly uncontroversial observation that the higher-order version of the DCG is far more perspicuous than the first-order version. The reason for this is also obvious. There is a need for an operation of substitution corresponding to λ-conversion in order to provide a solution to this problem. However, this operation is required at a conceptual level that is different from that of the logic of the grammar rules. This distinction is provided rather naturally in λProlog by the use of two distinct computational mechanisms to handle to process of substitution and the process of parsing. In contrast, the first-order language necessitates the presentation of the rules of λ-conversion that are pertinent to the solution of the problem *within* the logic of the grammar rules. Thus the arguments of the DCG 'predicates' sometimes perform the task of abstraction, sometimes of application and sometimes of constructing a logical form, and it is not clear which task is being performed in each instance!

The availability of higher-order terms, and the use of these in the DCG in λProlog offers another benefit that should be noted in this context. In the course of parsing a sentence, the arguments of the DCG predicates are instantiated by terms that have a logical significance, and that are, in fact, intended to correspond closely to the meaning of the subpart of the sentence that has been parsed. We have seen earlier in this section that it is possible to encode relationships between the denotations of such terms by a use of the logic of definite sentences. It thus appears that a language such as λProlog provides a framework for integrating some of the syntactic and semantic processes that are necessary for understanding sentences in a natural language into one computational process. While we do not pursue this aspect in this thesis, some indications of the suitability of λProlog in this respect are contained in [27].

One final point to note is that the DCG presented above may also be used to solve the inverse problem, namely that of obtaining a sentence given a logical form, and this illustrates a genuine use of higher-order unification. Consider the task of obtaining a sentence from the logical form that is represented by the expression

```
all X\ ((manp X) => (some Y\((womanp Y) & (lovesp X Y)))).
```

This task involves unifying the term above with the expression `(P1 P2)`. One of the unifiers for these two expressions is

$$\{\ \langle\ \texttt{P1},\ \texttt{P\(all X\((manp X) => (P X)))}\rangle,$$
$$\langle\ \texttt{P2},\ \texttt{X\(some Y\((womanp Y) \& (lovesp X Y))}\rangle\ \}.$$

Once this unifier is found, the task then breaks into that of obtaining a noun phrase from the the expression

```
P\(all X\((manp X) => (P X)))
```

and a verb phrase from

```
X\(some Y\((womanp Y) & (lovesp X Y)).
```

The use of higher-order unification thus seems to provide a top-down decomposition in the search for a solution. This view turns out to be a little simplistic however, since unification permits more structural decompositions than are warranted in this context. Thus, another unifier for the pair considered above is

```
{ ⟨ P1, Z\(all Z)⟩,
  ⟨ P2, X\((manp X) => (some Y\((womanp Y) & (lovesp X Y))))⟩}
```

which does not correspond to a meaningful decomposition in the context of the rest of the rules. It is possible to prevent such decompositions by anticipating the rest of the grammar rules. Alternatively, decompositions may be eschewed altogether; a logical form may be constructed bottom-up and compared with the given one. The first alternative detracts from the clarity and the specificational nature of the solution. The latter involves an exhaustive search over the space of all sentences. The DCG considered here, together with higher-order unification, seems to provide a balance between clarity and efficiency.

Representing and Manipulating Programs. We are now interested in a representation of programs that facilitates the task of analysing their structures and in performing transformations between them based on such an analysis. The data structures of λProlog are particularly apt in this respect, since, being based on a λ-calculus, they provide us with the tools required to make the functional structure of programs explicit. There is, however, an apparent problem since the simply typed λ-terms that are the data structures of λProlog do not embody notions such as recursion and do not provide a concept of data types that is rich enough for representing genuine programs. The deficiency with regard to data types can be overcome by describing suitable encodings for these into λ-terms; such is the approach adopted, for instance, in the Church numeral representation of integers. There is, however, an alternative solution that is more direct and that is adequate for our concerns in this section. This approach involves the use of λProlog constants to represent notions like recursion and data types. The equational properties of the notions represented by such constants are, of course, not understood in a primitive sense within λProlog. This is no obstacle, though, since those properties that require to be understood in manipulating representations of programs containing such constants may be encoded in λProlog procedures.

The ideas mentioned above may be illustrated by considering the task of representing simple functional programs. For simplicity of exposition, we assume that the programs that we consider perform computations over two primitive domains, these being the domains of integers and of truth values. We shall need types for λProlog terms that represent objects in these two different domains, and we reserve the types `bool` and `int` for this purpose. As noted already, our programs might involve operations on objects in these domains, and in representing these in λProlog we shall use appropriately typed constants that bear the same "names" as the operations they represent. The following type declarations in λProlog

provide an illustration of a set of such constants that we shall use in the representation of the sample programs we consider below; we assume also that an additional set of operator declarations define `=`, `<`, `+`, and `-` as infix λProlog operators.

```
type   not   bool -> bool.
type    =    int -> int -> bool.
type    <    int -> int -> bool.
type    +    int -> int -> int.
type    -    int -> int -> int.
```

In order to represent recursive programs, we need two other constants. These are the ones that shall represent the conditional and the fixed-point operators, and are defined by the following declarations.

```
type   cond    bool -> A -> A -> A.
type   fixpt   (A -> A) -> A.
```

These constants have a polymorphic type, but could be specialised for the purpose of the examples in this section. Thus, the only version of `cond` that we shall need is the one that has the type

```
bool  ->  int  ->  int  ->  int.
```

Armed with this vocabulary of constants, we may now represent simple functional programs in the term structures of λProlog. To illustrate this, let us consider the following program, in a syntax that bears some similarities to that of ML [17], that adds two integers:

```
sum(n,m) = if  n = 0
           then  m
           else  sum(n - 1,m + 1)
```

This program may be represented by the λProlog term

```
(fixpt  Sum \ N \ M \ (cond (N = 0)
                            M
                            (Sum (N - 1) (M + 1))))
```

Performing Transformations between Programs. The term representation of the program shown above has certain advantages given that λProlog understands the equational nature of typed λ-terms. The computational characteristics of the program `sum` is, for instance, impervious to the particular choice of name for the function or its arguments. This aspect is captured precisely by the notion of α-conversion in the context of the term representation of the program. Given a particular name for each argument, however, there is a strong correspondence between this name and each of its occurrences within the body of the program. This notion, of course, has a precise counterpart in the operation of binding as it is understood in the context of λ-terms. Finally, the λ-term representation clearly

captures the functional nature of the program in question. This aspect is of particular interest, since several manipulations that we might wish to perform on programs, such as those involved in effecting transformations between them, may be easily described based on their functional structure. In this regard λProlog offers another advantage, namely the ability to examine the functional structure of terms through higher-order unification. This ability makes it possible to implement directly the transformations that may be described through λProlog procedures.

We illustrate the above observations by considering the task of describing, and thus implementing, a particular kind of transformation between programs. The transformation that we consider involves removing tail recursion, in programs of two arguments in our simple functional programming language, in favour of iteration. The nature of the transformation intended may best be illustrated by an example of its application. Let us consider the program `sum` that was presented above. An execution of this program may involve a recursive call to itself. However, such a recursive call, if it occurs, would be the last expression that needs to be evaluated. Consequently, the recursion in this program may be replaced by a computationally less expensive iteration. The following program in an Algol-like syntax would, for instance, return the sum in `result` if `done` were initialised to `false` and `loc1` and `loc2` were initialised to the numbers whose sum needs to be computed.

```
while not(done) do
  begin if (loc1 = 0)
        then begin done := true ;
                   result := loc2
             end
        else begin loc1 := loc1 - 1 ;
                   loc2 := loc2 + 1
             end
  end
```

Given our term representation of functional programs, the tail-recursiveness of at least some of these programs can easily be recognised. The following term, for instance, would unify only with a term that represents a tail-recursive program

```
(fixpt Fun\X\Y\ (cond (C X Y) (H X Y) (Fun (F1 X Y) (F2 Y))))
```

The idea here is that the argument of `fixpt` in this term imposes a constraint on the functional structure on any term that it unifies with. The latter term must be such that the only occurrence of the function variable `Fun` being abstracted must be in the 'second arm' of `cond` and, that too, as the principal functor of that arm. It is clear that terms having such a structure can only correspond to tail-recursive programs.

Before we can describe any transformations from the representations of our functional programs into iterative programs, we clearly need to describe term representations for

iterative constructs. There are several different representations that might be chosen for this purpose, and the one we use here is, perhaps, the simplest. We shall represent iterative programs by (first-order) terms that reflect their "parse" structure*. This representation is made precise as follows. There are essentially two categories of constructs that need to be represented, and these correspond to *commands* and *evaluable expressions* respectively. Constructs of the latter category shall be represented by λProlog terms of the types `int` and `bool` described before, whereas for terms representing constructs of the former kind we reserve the type `cmd`. There is another notion that is of importance in the context of iterative programs, and this is that of *locations* in a store. We shall use λProlog constants to represent each of these locations and we reserve the type `reg` for these constants. In forming evaluable expressions, use might be made of operations on data types, and we assume that these operations are represented in much the same way as in the case of functional programs. There is one additional operation that needs to be represented, though, and this is the one that "coerces" a location in the store into its contents. We use the constant `find` of type `reg -> Val` to represent this operation; notice that this constant is polymorphic to capture the fact that the contents of a location may either be a boolean or an integer. Finally there are operations that are used in forming commands and we assume that these operations are represented by making use of λProlog constants of appropriate types; for the discussions in this section, we shall use the constants `:=` of type `reg -> Val -> cmd`, `&` of type `cmd -> cmd -> cmd`, `if` of type `bool -> cmd -> cmd -> cmd`, and `while` of type `bool -> cmd -> cmd`, in representing the assignment command, the sequencing of commands, and the conditional and the iteration commands respectively. The various aspects of the scheme just discussed may now be illustrated by considering the representation for the iterative program above. Assuming that `result`, `done`, `loc1` and `loc2` are each defined as constants of type `reg`, this program may be represented by the λProlog term shown below; we assume

* There is a certain asymmetry in this choice that should be mentioned. The term representations chosen for functional programs could be thought of as translations of these programs into an intermediate language on the way to explicating their meanings; by associating a denotation with each of the constants used in these representations and by using the antecedently understood meaning of λ-terms, we obtain a denotation for the program that a term represents. A representation that has much the same role for programs with iterative constructs could have been adopted, *e.g.* based on the discussions in [40]. The virtue of such a representation is that the correctness of transformations such as the one that we present below could then be established solely by noting the correspondences between the denotations of λ-terms. Although such a course is to be followed in a serious use of λProlog in implementing program transformations that can be justified to be correct, we have eschewed this approach here. This is because the description of this kind of a representation is fairly involved and is appropriate only if accompanied by arguments pertaining to correctness; our primary intent in this section is to demonstrate the potential usefulness of the data structures of λProlog, and of higher-order unification in their context, in describing, and thus implementing, program transformations and we do not wish to dwell on issues of correctness here.

here that `:=` and `&` are defined to be infix operators and that `&` has higher precedence than `:=`.

```
(while (not (find done))
       (if ((find loc1) = 0)
           (done := true & result := (find loc2))
           (loc1 := ((find loc1) - 1) &
            loc2 := ((find loc2) + 1))))
```

Returning to the issue of effecting a transformation from the representation of the recursive version of `sum` into the representation of the iterative version, we see now that this may be described via a process of "template" matching. Assume that a given term unifies with the "template" for tail-recursive programs that was shown above. Using the substitutions for the variables `C`, `H`, `F1`, and `F2` that make such a unification possible, this term may then be transformed into an alternative one by utilising the template

```
(while (not (find done))
       (if (C (find loc1) (find loc2))
           (done := true &
            result := (H (find loc1) (find loc2)))
           (loc1 := (F1 (find loc1) (find loc2)) &
            loc2 := (F2 (find loc2)))))
```

As a particular instance, consider the term representation for the recursive version of `sum`. Unifying this with the first template yields the substitution

$$\{\ \langle \texttt{C}\ ,\ \texttt{X\textbackslash Y\textbackslash(X = 0)}\rangle\ ,\ \langle \texttt{H}\ ,\ \texttt{X\textbackslash Y\textbackslash Y}\rangle,$$
$$\langle \texttt{F1}\ ,\ \texttt{X\textbackslash Y\textbackslash(X - 1)}\rangle\ ,\ \langle \texttt{F2}\ ,\ \texttt{Y\textbackslash(Y - 1)}\rangle\ \}$$

Applying this substitution to the second template would then yield the term structure corresponding to the iterative version.

The recognition of tail recursion outlined above, and the corresponding conversion to an iterative version, illustrates a novel use of higher-order unification. This kind of use of higher-order unification has been recognised previously by Huet and Lang [23], and has been used there to formalise some of the approaches to program transformations studied by Burstall and Darlington [12]. The applicability of the notion of template matching that is the basis of this approach is, however, limited since only restricted kinds of patterns can be recognised by using it. Consider, for instance, the following term that represents a program that computes the greatest common denominator of two numbers:

```
(fixpt Gcd\X\Y\ (cond (1 = X) 1
                      (cond (X = Y) X
                            (cond (X < Y) (Gcd Y X)
                                  (Gcd (X - Y) Y)))))
```

The program represented by this term is obviously tail-recursive. It is clear, though, that this term does not unify with the pattern that was used to recognise the tail-recursiveness of `sum`. What is worse is that there is no term all of whose instances are representations of tail-recursive programs and which also has the above term and the term representation of `sum` as instances.

There is, nevertheless, a recursive specification of a class of terms that can be used to recognise the tail-recursiveness of both the programs above. Consider a term of the form `(fixpt Prog)`. In the trivial case this term represents a tail recursive program if `Prog` is of the form `F\X\Y\(H X Y)` or of the form `F\X\Y\(F (H X Y) (G X Y))` — i.e. it corresponds to a recursive program in which there are either no recursive calls or there is only a recursive call with modified arguments. However, the term also represents a tail recursive program if `Prog` has the functional structure `F\X\Y\(cond (C X Y) (H1 F X Y) (H2 F X Y))` where `(fixpt H1)` and `(fixpt H2)` themselves represent tail-recursive programs.

It should now be clear how procedures in a logic programming language may be combined together with λ-terms to provide a concise specification of the class of terms described above. Such a specification could then be used to recognise the fact that both `sum` and the program for computing greatest common denominators that is represented by the term shown above are tail-recursive. What is interesting is that the same description also provides us with a means for specifying a transformation into an iterative version of the program. The idea here is as follows. The term `(fixpt Prog)` transforms into the term

```
(while (not (find done)) Body)
```

where `Body` is obtained by a transformation of `Prog`; the intuitive picture is that an iteration (in which `done` is used to flag the end of the iteration) replaces recursion and the "code" to be iterated over is obtained from the form of the functional program. Now the form of `Body` can easily be guessed from the cases corresponding to the form of `Prog`. The case where `Prog` is of the form `F\X\Y\ (H X Y)` corresponds to the "bottoming out" of the recursion and, hence, to iterative code for terminating the iteration. The case when `Prog` is of the form `F\X\Y\ (F (H X Y) (G X Y))` corresponds to resetting of variables within an iteration. Finally, the case when `Prog` has the form of a conditional yields a conditional in iteration.

Putting these observations together with a recognition of some "special" cases in which the iterative code may be simplified, we obtain the following set of λProlog procedures that specify a richer class of tail-recursion transformations than can actually be specified by the mere use of templates.

```
trans_tail (fixpt Prog)
           (while (not (find done)) G) :-
                                trans_body Prog G.
trans_body (F\X\Y\ (H X Y))
           (result := (H (find loc1) (find loc2)) &
            done := true).
trans_body (F\X\Y\ (F (G X Y) Y))
           (loc1 := (G (find loc1) (find loc2))).
trans_body (F\X\Y\ (F (G X Y) (H Y)))
           (loc1 := (G (find loc1) (find loc2)) &
            loc2 := (H (find loc2))).
trans_body (F\X\Y\ (F (G X Y) (H X Y)))
           (temp := (find loc1) &
            loc1 := (G (find temp) (find loc2)) &
            loc2 := (H (find temp) (find loc2))).
trans_body (F\X\Y\ (cond (C X Y) (H1 F X Y) (H2 F X Y)))
           (if (C (find loc1) (find loc2)) G1 G2) :-
                  trans_body H1 G1, trans_body H2 G2.
```

The predicate `trans_tail` defined by the procedures above describes the transformation of a class of tail-recursive programs, that includes both `sum` and the program for computing greatest common denominators, into their respective iterative versions. The clarity and the conceptual elegance of this description is perhaps best appreciated by the reader who goes through the exercise of providing equivalent descriptions in a language that is not equipped with an understanding of the binding operation provided by λ-abstraction. Another point to note is that these procedures also provide an *implementation* of the transformation that they describe. In attempting to perform this transformation on specific programs, a λProlog interpreter would make a non-trivial use of higher-order unification — in particular, of second-order matching* — as the reader may verify.

Section 4: An Experimental Interpreter for λProlog

The discussions in Section 3.4 have examined the logical basis for an interpreter for λProlog. From these discussions, it is clear that such an interpreter may be described as a procedure which, given a program $\mathcal{P}$ and a query G, attempts to construct a $\mathcal{P}$-derivation of G. Abstracting out those components of a tuple in a $\mathcal{P}$-derivation that are relevant to the search

* The *second-order matching problem* is a restricted version of the higher-order unification problem that is described as follows. Given two terms F^1 and F^2 all of whose variables have a type either of the form $\alpha_1 \to \cdots \to \alpha_n \to \beta$ or of the form β where $\alpha_1,\ldots,\alpha_n$, and β are atomic types, we desire to find a substitution θ such that $\theta(F^1)$ λ-converts to F^2.

for a $\mathcal{P}$-derivation, we see that the search space for such a procedure may be characterised by a set of states, each of which is represented by a pair consisting of a goal set and a disagreement set. The initial state in the search corresponds to the pair $\langle\{G\}, \emptyset\rangle$. At each stage in the search, the procedure is confronted with a state that it must attempt to simplify. The process of simplification involves trying either to find a solution to the unification problem posed by the disagreement set, or to reduce the goal set to an empty set or a set of flexible atomic goals. Given a particular state, there are several ways in which a specific step may be chosen in attempting to bring the search closer to a resolution. There are, however, only a finite number of such choices, and we have seen that a procedure that tries, exhaustively, all the steps possible at each stage is bound to find a $\mathcal{P}$-derivation of G if one exists.

A theorem-proving procedure that is complete can thus be described for the logic of definite sentences. In order to be complete, however, such a procedure must perform an exhaustive search. An exhaustive search is clearly an inappropriate basis for a procedure that is also to be an *interpreter* for a programming language, and trade-offs need to be made between completeness and practicality. In order to get an empirical understanding of the nature of the trade-offs involved, we have constructed an experimental interpreter for λProlog*. This interpreter has been implemented in C-Prolog [35], a dialect of Prolog, and has been tested on the examples described in this thesis and on several other examples as well. In attempting to find a $\mathcal{P}$-derivation, this interpreter performs a depth-first search with backtracking that in some ways resembles the kind of search standard Prolog interpreters perform. Indeed, our interpreter was constructed with the intention that it should behave in a manner similar to the Prolog interpreter on the first-order subset of our language. In dealing with the higher-order aspects, however, there are certain additional choices to be made. We describe some of the features of our interpreter below, highlighting, in the process, the additional choices and also some of the experiences we have gained, chiefly in implementing higher-order unification. There are certain problems, discussed briefly in Section 4.1, in constructing an interpreter for λProlog that arise out of our use of type variables, and we also describe below some techniques that we have used to deal with this problem.

Unify-first. An interpreter for λProlog may be thought of as a procedure that, given a program $\mathcal{P}$, and a goal G, starts with a $\mathcal{P}$-derivation sequence for G and attempts to

* As is apparent from the discussions in this chapter, an implementation of λProlog involves more than the implementation of an interpreter. Our implementation effort can, in fact, be factored into two parts: That of constructing a parser for understanding λProlog declarations and queries, for performing type inference, and so on, and that of constructing an interpreter for answering queries. However, it is only questions relating to an interpreter that hold our attention in this section, and we therefore discuss only the latter aspect of our implementation here.

extend this into a $\mathcal{P}$-derivation of G. As we have observed, the choices in attempting to extend a $\mathcal{P}$-derivation sequence can be classified into two broad categories, namely that of trying to solve a unification problem and that of trying to "solve" a goal set. We have seen, in Lemma 3.4.21, that the particular manner in which this choice is exercised does not affect the overall completeness of a proof procedure. Our interpreter chooses between the two courses in the following fashion. If the unification problem corresponds to a solved disagreement set, then the interpreter attempts to solve a goal from the goal set. Otherwise, the interpreter always elects to use a unification step. There are two points that should be noted with regard to this strategy. First, in always trying to solve an unsolved unification problem before looking at the goal set, our interpreter functions much like standard Prolog interpreters which always solve the trivial, first-order unification problems first. Second, the attempt to solve a unification problem always stops short of looking for unifiers for solved disagreement sets. The search for unifiers for such sets can be rather expensive [22], and we avoid this search by "carrying forward" the solved disagreement sets*. In fact, our interpreter never attempts to find unifiers for a solved disagreement set; if it has also succeeded in solving the goal set, then it returns the answer substitution and the final solved disagreement set. We describe below certain heuristic improvements that are incorporated in our interpreter that, in some cases, have the effect of reducing such sets to empty sets. In other cases, it is our belief that the user is not interested in particular unifiers for the final solved set.

Disjunctive Goals. When the interpreter attempts to solve a goal set, it does so by picking the first goal in the set†. If the goal is not atomic and not disjunctive, then there is only one way in which it may be simplified; the interpreter performs this simplification and reinserts the result at the beginning of the set. If the goal set is of the form $\{G^1 \vee G^2\} \cup \mathcal{G}$, then, for the sake of completeness, the interpreter should actually try to solve both $\{G^1\} \cup \mathcal{G}$ and $\{G^2\} \cup \mathcal{G}$ in a breadth-first fashion. In such cases, however, our interpreter attempts to solve $\{G^1\} \cup \mathcal{G}$ first, and returns to $\{G^2\} \cup \mathcal{G}$ only if its first attempt leads to a failure.

Flexible Atomic Goals. If the interpreter encounters a flexible atom in the goal set, then it solves it in a "eager" fashion using the substitution discussed in Theorem 3.4.18. Thus, if the goal is of the form $[P\, C^1 \ldots C^n]$ where P is a variable of the type $\alpha_1 \to \ldots \to \alpha_n \to o$, then the interpreter removes this goal from the goal set and "simplifies" the resulting goal set and the associated disagreement set further by applying the substitution $\{\langle P, \lambda x_{\alpha_1}. \ldots \lambda x_{\alpha_n}.\top\rangle\}$ to them. For the sake of completeness it is necessary that the interpreter should delay solving such a goal if P appears free in the associated disagreement

* Note that the trivial substitution of Theorem 3.4.18 works only if the associated goal set is either empty or consists solely of flexible atoms.

† We make a systematic confusion in this section between a set and its representation in a computer as a list.

set or in any of the other goals in the goal set. To illustrate the need for this, consider the program in Example 3.1.7 a version of which appears in λProlog syntax in Section 4.2. In the context of this program, the query

```
R john mary , rel R
```

has an answer, namely

```
R = X\Y\ (sigma Z\ (mother mary Z , wife Z john)),
```

but our interpreter will fail to find it. Our reason for adopting this course is that we believe reordering goals is an expensive operation, and that the need for this is eliminated if the user is sensitive to some aspects of control. Thus, the reordered query

```
rel R , R john mary
```

will produce the expected answer.

Rigid Atomic Goals. There is finally the case when the goal the interpreter encounters is a rigid atom. In attempting to solve such a goal, the interpreter must use a backchaining step. Now, in performing such a step, it is evidently enough to consider only procedures of the form `A` or `A :- G` where the head of `A` is identical to the head of the goal being solved, since all other cases will cause the disagreement set to be reduced to **F** by SIMPL. Our interpreter considers only such procedures and, in fact, indexes procedures by the name of the predicate they define. For completeness, it is necessary to use each of these procedures in a breadth-first fashion in attempting to solve the goal. Here again we have sacrificed completeness by using the scheme that is used by standard Prolog interpreters: We pick the first appropriate procedure based on some predetermined ordering of the procedures in the program, and return to attempt the next one only if the first choice leads to failure.

Determining Type Instantiations. As we noted in Section 4.1, our use of type variables requires us to pick not only a procedure but also a particular type instance of it, in the last case discussed above. The implication of this observation is that after choosing a particular procedure we also need to instantiate all the type variables that appear in it before we use it in an attempt to solve a rigid atomic goal. If a procedure does have type variables in it, this turns out to be a problem since there are obviously an infinite number of type instances of it, many of which may later turn out to be inappropriate in particular cases due to type considerations. The approach that we have adopted to deal with this proliferation of, often unnecessary, cases is to permit type variables in the types of the goals in a goal set and of the wffs in a disagreement set, thereby delaying the determination of type instances until it is actually needed. The only occasions when type variables might need to be instantiated are when use is made of SIMPL and MATCH. Of these functions, SIMPL can be modified to deal directly with type variables. In fact, the modification to SIMPL, that is incorporated into our interpreter, has the effect of *determining* the appropriate instantiations for type

variables in several cases. We illustrate this by considering an example. Let us assume that we have the goal

`append`^{1} `(cons 1 nil) (cons 2 nil) L`,

where the type of `append`^{1} is evidently `(list int) -> (list int) -> (list int) -> o`. Further, let us assume that wish to use the procedure

`append`^{2} `(cons X L) L2 (cons X L3) :-` `append`^{2} `L1 L2 L3`,

from the program in Section 4.1 in an attempt to solve this goal; we note that `append`^{2} in this procedure has type `(list A) -> (list A) -> (list A) -> o` we have used superscripts on `append` only to distinguish between two occurrences of this constant that have different type expressions associated with them. This attempt to solve the goal will lead to an application of SIMPL on the set

$\{ \langle$`(append`^{2} `(cons X L) L2 (cons X L3))`,
`(append`^{1} `(cons 1 nil) (cons 2 nil) L)`$\rangle$ $\}$.

Since the types associated with the two wffs in the pair in this set must be identical if they are to unify, it is clear that the type variable `A` must be instantiated to `int`. The effect of such an instantiation may be obtained by performing first-order unification on the types associated with the wffs in a disagreement pair, and we have modified SIMPL to do such a unification.

Types of course play a critical role in MATCH and, in determining substitutions here, type expressions containing variables will, in general, need to be instantiated. However, there are cases even here in which such an instantiation may be delayed in the hope that SIMPL will be able to determine a unique instantiation. To see this, let us assume that the flexible wff provided to MATCH is $\lambda\bar{x}.[f\ A^1\ \ldots\ A^n]$ and the rigid wff is $\lambda\bar{x}.[C\ B^1\ \ldots\ B^m]$* and that the type expressions associated with f and C are $\alpha_1^1 \to \ldots \to \alpha_n^1 \to \beta^1$ and $\alpha_1^2 \to \ldots \to \alpha_m^2 \to \beta^2$ respectively. Now if β^1, and consequently also β^2, is not a variable, then IMIT may still generate one substitution. To take a specific example, let the two wffs provided to IMIT be $[F\ X]$ and $[cons\ 2\ [cons\ 3\ nil]]$, where the type associated with the variable F is $A \to (list\ int)$. In this case IMIT may return the substitution

$$\{\langle F, \lambda w.[cons\ [h^1\ w]\ [h^2\ w]]\rangle\}$$

where the types associated with w, h^1 and h^2 are W, $W \to int$ and $W \to (list\ int)$ respectively.

In a similar manner we note that if, in addition to β^1 not being a variable, the target types of α_i^1 are not variables for $1 \leq i \leq n$, then PROJ$_i$ may produce only one substitution without instantiating type variables.

---

* We assume here that SIMPL makes the type expressions associated with the two wffs in each disagreement pair the same. It can be seen, then, that the wffs provided to MATCH have "β-normal forms" in which their binders are identical.

To deal with the other cases, however, MATCH would need to produce instantiations for at least some of the type variables that occur in the wffs it is provided with. This is, of course, a problem since MATCH would in effect have to produce an infinite number of substitutions. When encountered by such cases, our current interpreter gives up and indicates a run-time error. It is clear that there is a need for a better solution for the problem that led us to the introduction of type variables. Such a solution is, perhaps, to be found by using a language that has a richer type structure; a detailed study of this aspect is, however, beyond the scope of this thesis.

Choosing Substitutions. In attempting to solve (the unification problem corresponding to) a disagreement set, the interpreter picks the first flexible-rigid pair in the set and invokes MATCH to generate a set of substitutions corresponding to this pair. It then chooses one of these substitutions and uses it to progress the search further. Since MATCH produces more than one substitution in general, the remaining substitutions are retained and constitute alternative choices in case there is a need to backtrack due to a failure.

There are certain biases that may be incorporated in choosing from the substitutions provided by MATCH, depending on the kinds of solutions that are desired first. To illustrate this, let us consider the unification problem posed by the following disagreement set

$$\{\langle [F\,2], [cons\,2\,[cons\,2\,nil]]\rangle\}$$

where F is a variable of type $int \rightarrow (list\,int)$. Now there are four unifiers for this set and these are listed below:

$$\{\langle F, \lambda x.[cons\,x\,[cons\,x\,nil]]\rangle\},$$
$$\{\langle F, \lambda x.[cons\,2\,[cons\,x\,nil]]\rangle\},$$
$$\{\langle F, \lambda x.[cons\,x\,[cons\,2\,nil]]\rangle\},$$
$$\{\langle F, \lambda x.[cons\,2\,[cons\,2\,nil]]\rangle\}.$$

If the substitutions provided by PROJ$_i$s are chosen first at each stage, then these unifiers will be produced in the order that they appear above, perhaps with the second and third interchanged. On the other hand, choosing the substitution provided by IMIT first results in these unifiers being produced in the reverse order. Now, the above unification problem may arise in a programming context out of the following kind of desire: We wish to unify the function variable F with the result of "abstracting" out all occurrences of a particular constant, which is 2 in this case, from a given data structure, which is an integer list here. If this is the desire, then it is clearly preferable to choose the substitutions provided by the PROJs before those provided by IMIT.

In general, it is necessary to consider the aspect of providing a user of λProlog with a means for controlling the unification process. The ability to provide such controls would, we believe, depend on one's being able to describe the effect of steps within the unification algorithm on its global behaviour. While some understanding of this aspect has arisen out

of an experimentation with various programs, this is at the present time too preliminary to be reported here; we hope that a much clearer picture will emerge out of more extensive experimentation. It is also to be noted that although our language provides for terms of all types, it may be desirable, for the purpose of constructing an interpreter that is "efficient", to tailor it to deal only with certain special cases. For instance, much mileage may be obtained out of using just λ-conversion and second-order matching which is known to be decidable [23]. Once again, this is an issue that needs to be settled on the basis of further experimentation.

Improving the Unification Process. Certain techniques are suggested in [22] that sometimes help in the solution of the unification problem, and we mention one of these that is incorporated into our interpreter. Consider a disagreement pair of the form $\langle x, F\rangle$ where x is a variable and F is a wff in which x does not appear free. It can easily be shown that $\{\langle x, F\rangle\}$ is a most general unifier of $\{\langle x, F\rangle\}$. This observation can actually be strengthened, since we have assumed the rule of η-conversion. Let us suppose that instead of the pair $\langle x, F\rangle$, we now have the pair $\langle F^1, F\rangle$ where F^1 has as a β-normal form the wff

$$\lambda y_1. \ldots \lambda y_n.[x\ y_{p_1}\ \ldots\ y_{p_n}]$$

such that $y_{p_1}, \ldots, y_{p_n}$ is a permutation of the sequence $y_1, \ldots, y_n$ and, further, none of the variables $y_1, \ldots, y_n$, and x occur free in F. It may once again be seen that the substitution

$$\{\langle x, \lambda y_{p_1}. \ldots \lambda y_{p_n}.[F\ y_1\ \ldots\ y_n]\rangle\}$$

is a most general unifier of $\{\langle F^1, F\rangle\}$. Thus, if a pair of this form appears in a disagreement set, then the latter may be simplified by first removing the pair from the set and then applying the most general unifier displayed above to the resulting set and to the associated goal set. We have incorporated this heuristic into SIMPL and have found it useful in several instances. For example, if a program $\mathcal{P}$ and a query G contain only first-order terms, then the final solved set in any $\mathcal{P}$-derivation of G is always reduced to an empty set without calling MATCH if use is made of this heuristic.

In a similar vein, if the pair $\langle F^1, F\rangle$ shown above is such that x appears free in F, there are cases in which it is clear that no unifiers can exist for a disagreement set that contains this pair. One such case is when x and F are first-order terms. Indeed, a check for such cases, known as the *occurs check*, is an integral part of first-order unification algorithms. However, a naive use of the occurs check is not appropriate in the higher-order context. Consider, for instance, the disagreement pair $\langle x, [f\ x]\rangle$, where f is a function variable of type $int \to int$. This pair has a unifier, namely $\{\langle f, \lambda y.y\rangle\}$, although x appears free in the second term in the pair. There are, nevertheless, certain cases when nonunifiability can be detected even in the higher-order context. Huet [22] describes a method for detecting some of these cases that can be incorporated into SIMPL. This method actually subsumes the occurs check for the first-order case. Incorporating it into SIMPL together with the heuristic

discussed earlier in this section, therefore, has the effect that the unification problem for first-order terms is solved entirely within SIMPL.

Chapter 5
Conclusion and Future Work

The motivation for this thesis was provided by the realisation that a logic programming language incorporating an understanding of higher-order objects would be valuable. A language embodying such an understanding could be described in two distinct ways. The first approach would be to retain the first-order logic underlying traditional logic programming languages and to add on higher-order features at the level of an implementation. The second approach would be to abstract out those properties of the underlying first-order logic that are essential to its computational interpretation and then describe a higher-order logic that retained these features. The first approach has the advantage in that it allows one to provide usable "higher-order" extensions in a relatively short period of time, but we believe it is the latter approach that is more profitable in the long run. This belief is justified partly by the observation that one of the main strengths of a logic programming language lies in its clear theoretical basis, and partly by the observation that it is only in the context of a genuine higher-order language that the full value of higher-order features can be realised.

The above observations have led us to seek an appropriate logical basis for the introduction of a notion of higher-order objects within the paradigm of logic programming. To achieve this goal, we have used a higher-order logic to describe a generalisation to the definite clauses of first-order logic. We have studied the proof-theoretic properties of this generalisation and shown that the generalisation can be used to specify computations in much the same way as its first-order counterpart. We have also described a theorem-proving procedure for this generalisation, a procedure which forms the basis for an interpreter that executes specifications provided by using the generalisation. The results have enabled us to describe a logic programming language called λProlog. This language extends a language like Prolog by permitting predicate and function variables. The truly novel feature of λProlog, however, is its use of typed λ-terms as data structures. We believe that this feature has several promising applications, and we have provided examples to bolster this belief.

The work in this thesis has, thus, achieved a large part of its original objective, namely that of describing, in a principled fashion, a logic programming language that realises the full potential of higher-order features within this paradigm of programming. There remain, however, a few questions that require further consideration. These questions concern pragmatic issues relevant to the programming use of λProlog, and we discuss them as possibilities for future work.

Providing a Good Implementation. Our current implementation of λProlog has been motivated mainly by a desire to examine whether higher-order unification can be used to

solve problems in practice as it can in theory, and to understand the nature of the tradeoffs that have to be made in the practical realisation of the logic of definite sentences. Not much heed has been paid, as a result, to the efficiency of the implementation. Having convinced ourselves of the feasibility of λProlog, we now propose to consider, seriously, the issue of efficiency; indeed, the importance of this issue cannot be overemphasised, given that the main key to the success of Prolog has been the description of abstract machines that facilitate extremely efficient implementations of the logic of first-order definite clauses [42, 44]. One of the main challenges in implementing λProlog is that of devising good data structures for λ-terms. Although representations for such terms have been considered in the functional programming context, the nature of the representation in λProlog will have to be significantly different, since there is the additional requirement for examination and manipulation of the internal structure of λ-terms. This concern also arises in the context of formula manipulating systems, and there is, consequently, value in examining the representational techniques (*e.g.* those discussed in [9]) that have been developed in the context of these systems. There are at least two other constraints that must be satisfied by a representation of λ-terms for it to be suitable for λProlog:

(i) The representation should be conservative in its use of space. This is an important requirement, since λ-conversion often leads to an explosion in the sizes of terms. We believe that the techniques of structure-sharing, developed in the context of Prolog and of theorem-proving systems for first-order logic [8], should turn out to be useful in keeping the usage of space low despite the growth in the sizes of terms.

(ii) The representation should be such that it is possible to undo the effects of a β-reduction easily. This requirement arises because the application of a substitution, an operation that corresponds naturally to a β-reduction, may have to be reconsidered in the course of backtracking. A similar problem arises in the context of Prolog. The solution there, however, is much simpler than any that we might have to consider here. This is because the application of a substitution has a very local effect in first-order terms, whereas it may change the entire structure of a λ-term.

It is to be noted that some of the problems mentioned above also arise in paradigms that attempt to combine equational forms of programming with logic programming. This is to be expected, since higher-order unification is a special case of equational unification. A good solution to the implementation problems discussed above should, consequently, be of more general interest.

Improving the Type System. Another question that needs to be considered more closely concerns the aspect of types. The notion of types provided by the logic underlying λProlog is, as we have seen, a little too restrictive in the programming context. In an attempt to redress this situation, we have introduced a flexibility into the typing system by permitting the use of type variables. However, this solution is not ideal, since it has on occasions led to

unanticipated problems in the task of interpreting queries. A better solution would be one that chooses between these two extremes, by providing the user of the system with a means to control the flexibility in the interpretation of types. Such a solution may, perhaps, be realised by the use of a language that permits an explicit quantification over types. This endeavour may require complete reconsideration of the logical basis of λProlog, a task best undertaken after a typing system that is satisfactory from the programming point of view has been enunciated.

Controlling Unification. As we have noted several times in this thesis, higher-order unification is a fairly complex operation. While the use of this operation is justified by the fact that it enables us to provide conceptually elegant solutions to difficult problems, there is, nevertheless, a need to exercise care in the way it is used. It is important, therefore, to devise a method for imparting an understanding of this operation to a user of λProlog who is unacquainted with the unification procedure. It is also necessary to provide such a user with control primitives that might be used for affecting the unification process. With regard to the latter aspect, there are two sorts of control primitives that might be considered. The first sort is one that would permit the user to bias the unification process towards finding certain unifiers before others; we have seen an example of where such a bias might be useful in Section 4.4. The second sort is one that would permit the user to prevent certain kinds of unifiers from being found. To illustrate where such an ability might be useful, let us consider the disagreement set

$$\{\langle [F\,2], [cons\,2\,[cons\,3\,nil]]\rangle\}.$$

This pair has a unifier that corresponds to substituting a constant function for F, *i.e.* the substitution

$$\{\langle F, \lambda x.[cons\,2\,[cons\,3\,nil]]\rangle\}.$$

However, this unifier may be undesirable in a programming context, especially if the user wants to determine whether F can be unified with the result of abstracting at least one occurrence of 2 out of the given list*.

Clearly, much remains to be done in characterising the programming aspects of higher-order unification. A first step in this direction would be to gain an insight into the effect of local choices within the unification procedure on the global behaviour of the process. Such insight should emerge through careful experimentation with several different kinds of programs. Along another vein, such experimentation should enable us to determine whether the full power of higher-order unification is really needed to gain the advantages

* Our current implementation provides the primitive predicates `not` and `=` that are similar to those in Prolog. The user may, therefore, deal with this particular problem by using the goal `(not F = X\Y)`. However, a better way to deal with the situation would be to incorporate the undesirability of constant functions into the unification process, rather than to use it as a criterion for screening out unifiers after all the work in finding them has been done.

of higher-order terms. If the examples in this thesis are any indication in this direction, it appears that there is much to be gained merely through the additions of λ-conversion and second-order matching. Assuming only restricted forms of higher-order unification are needed, it would be of interest to consider the influence this would have on characterising the unification process and also on describing an "efficient" implementation for λProlog.

Exploring the New Applications. The use of λ-terms as data structures raises a host of computational problems that are new in the logic programming context and studying these problems is a worthwhile effort, since the use of λ-terms also provides a source of richness to the logic programming paradigm. We have illustrated how the use of these data structures together with definite clause reasoning facilitates implementation tasks that involve manipulations on formulas as well as on programs. In this respect, λProlog holds substantial promise for being a vehicle of implementation in areas that are not well supported by existing computational formalisms. To consider one example, the language ML [17] has been extensively used in implementing proof systems. The notion of search that is necessary in this implementation task is, however, foreign to the functional programming paradigm on which ML is based, necessitating the introduction of an exception handling mechanism into the language. The use of this mechanism has one disadvantage in that it conflicts with the typing notions the language provides, thereby detracting from the clarity of the implementations. In contrast, the notion of search is fundamental to the programming paradigm underlying λProlog. Furthermore, λProlog also provides the advantage that the intensions of higher-order objects may be examined through unification, and we have seen the usefulness of this ability in implementing formula and program manipulating systems.

It is to be noted that the above arguments and the illustrations in this thesis are only preliminary indications of the usefulness of λProlog. One of the goals for the future is to actually explore and exploit the potential of the language in these arenas.

Appendix A
Abstract Consistency Properties for $\mathcal{T}^*$

The system $\mathcal{T}^*$ differs from the system $\mathcal{T}$ of [1] principally in the choice of connectives, in the use of existential quantification as the primitive notion and in the inclusion of η-conversion as a rule of inference. The definition of an abstract consistency property for $\mathcal{T}^*$ is an adaptation of the corresponding notion for $\mathcal{T}$ that reflects these differences.

A.1. Definition. A property Γ of finite sets of wffs$_o$ is an *abstract consistency property* (relative to $\mathcal{T}^*$) if and only if the following properties hold for all finite sets $\mathcal{S}$ of wffs$_o$ and for all wffs F, G, and P of the appropriate types:

ACP1 If $\Gamma(\mathcal{S})$, then $\sim\top \notin \mathcal{S}$.

ACP2 If $\Gamma(\mathcal{S})$, then there is no atomic formula F such that $F \in \mathcal{S}$ and $\sim F \in \mathcal{S}$.

ACP3 If $\Gamma(\mathcal{S} \cup \{F\})$, then $\Gamma(\mathcal{S} \cup \{\rho(F)\})$.

ACP4 If $\Gamma(\mathcal{S} \cup \{\sim\sim F\})$, then $\Gamma(\mathcal{S} \cup \{F\})$.

ACP5 If $\Gamma(\mathcal{S} \cup \{F \vee G\})$, then $\Gamma(\mathcal{S} \cup \{F\})$ or $\Gamma(\mathcal{S} \cup \{G\})$.

ACP6 If $\Gamma(\mathcal{S} \cup \{\sim[F \vee G]\})$, then $\Gamma(\mathcal{S} \cup \{\sim F\})$ and $\Gamma(\mathcal{S} \cup \{\sim G\})$.

ACP7 If $\Gamma(\mathcal{S} \cup \{F \wedge G\})$, then $\Gamma(\mathcal{S} \cup \{F\})$ and $\Gamma(\mathcal{S} \cup \{G\})$.

ACP8 If $\Gamma(\mathcal{S} \cup \{\sim[F \wedge G]\})$, then $\Gamma(\mathcal{S} \cup \{\sim F\})$ or $\Gamma(\mathcal{S} \cup \{\sim G\})$.

ACP9 If $\Gamma(\mathcal{S} \cup \{\Sigma\, P\})$ and y is a parameter (variable) that does not occur (occur free) in any wff in $\mathcal{S} \cup \{P\}$, then $\Gamma(\mathcal{S} \cup \{P\, y\})$.

ACP10 If $\Gamma(\mathcal{S} \cup \{\sim\Sigma\, P\})$, then $\Gamma(\mathcal{S} \cup \{\sim\Sigma\, P, \sim P\, c\})$ for any wff c of the appropriate type.

The relationship between abstract consistency properties and abstract derivability properties for $\mathcal{T}^*$ that is the content of the following proposition is identical to the one for $\mathcal{T}$ that is described in [26]. The proof of this proposition may also be obtained by arguments similar to those in [26] (Lemma 2.2.4); we assume that Λ is an abstract derivability property and, for any $\mathcal{S}$, we show that $\sim\Lambda(\sim\mathcal{S})$ satisfies the contrapositive form of each of the abstract consistency conditions.

A.2 Proposition. *Let Λ be an abstract derivability property and let Γ be the property of wffs$_o$, $\mathcal{S}$, such that $\Gamma(\mathcal{S})$ if and only if $\sim\Lambda(\sim S)$. Then Γ is an abstract consistency property.*

Let us say that a finite set $\mathcal{S}$ of wffs$_o$ is consistent if it is not the case that $\vdash_{\mathcal{T}^*} (\vee\sim\mathcal{S})$. Then the usefulness of abstract consistency properties is obtained from the following proposition; our particular interest in this proposition is that, together with Proposition A.2, it yields Theorem 2.3.3.

A.3 Proposition. *If Γ is an abstract consistency property and $\mathcal{S}$ is a finite set of wffs$_o$*

such that $\Gamma(\mathcal{S})$*, then* $\mathcal{S}$ *is consistent.*

This proposition is Theorem 3.5 in [1], with the difference that provability in $\mathcal{T}$ is replaced by provability in $\mathcal{T}^*$. The proof of Theorem 3.5 in [1] may be outlined in the following manner. First a notion of semivaluation is defined. This is essentially a partial function from the set of wffs_o to the truth values $\{t, f\}$ that is true to the intended interpretation of the logical constants. It is then shown that if Γ is an abstract consistency property and if $\Gamma(\mathcal{S})$ holds for a set $\mathcal{S}$ of wffs_o, then there is a semivaluation V that assigns t to each wff A in $\mathcal{S}$. Now, each semivaluation may be used to construct a (non-extensional) general model in which the theorems of $\mathcal{T}$ are satisfied. As a consequence of this construction, it turns out that if $\vdash_{\mathcal{T}} A$ holds, then for each semivaluation V it is the case that $V(A) = t$. From these two observations, however, it follows that if $\Gamma(\mathcal{S})$ holds then $\mathcal{S}$ cannot be inconsistent.

A virtually identical argument may be adopted for showing Proposition A.3. The only changes that are necessary are those that take into account the different set of logical constants in $\mathcal{T}^*$ and the inclusion of η-conversion as a rule of inference. These changes are outlined below. In order to avoid having to reproduce the lengthy technical arguments presented therein, we assume that the reader is familiar with the contents of [1].

(i) To the definition of semivaluations, *i.e.* Definition 3.2 in [1], we add the following properties:

$V(\top) = t$,

If $V([A \wedge B]) = t$, then $V(A) = t$ and $V(B) = t$, and

If $V([A \wedge B]) = f$, then $V(A) = f$ or $V(B) = f$.

We also replace the properties 3.2.1, 3.2.6 and 3.2.7 by

If $V(A)$ is defined, then $V(\rho(A)) = V(A)$,

If $V(\Sigma P) = t$, then for some wff C, $V([P\, C]) = t$, and

If $V(\Sigma P) = f$, then for each wff C, $V([P\, C]) = f$,

respectively.

(ii) The definition of the domains of "models", *i.e.* the sets of V-complexes, needs to be changed so as to ensure that the same value gets assigned to two wffs that η-convert to each other. This is done by replacing η and η-wff by ρ and ρ-wff, respectively, throughout Definition 3.4.1. Furthermore, there may be more than one atomic type other than o in $\mathcal{T}^*$, and this collection may also not include ι. Consequently, 3.4.1.2 should be replaced by the following:

For each atomic type α other than o, $\mathcal{D}_\alpha = \{\langle A_\alpha, \alpha\rangle \mid A_\alpha \text{ is a } \rho\text{-wff}_\alpha\}$.

(iii) In extending an assignment of "values" to variables, to an assignment of values to all wffs, we need, first of all, to account for the addition of the rule of η-conversion.

This is done by replacing η by ρ in definition 3.4.4. Furthermore, we need to define the second component of a V-complex that is assigned to each of the new logical constants. This is done in the following manner:

$\mathcal{V}^2_\varphi(\top) = t$,

$\mathcal{V}^2_\varphi(\wedge)(\langle B_o, q\rangle) = \langle[\wedge\, B_o], h\rangle$, where h is the function on $\mathcal{D}_o$ such that $h(\langle E_o, r\rangle) = \langle[\wedge\, B_o]\, E_o, q \wedge r\rangle$ for any $\langle E_o, r\rangle \in \mathcal{D}_o$, and

If the type of Σ is $(\alpha \to o) \to o$, then $\mathcal{V}^2_\varphi(\Sigma)(\langle A, q\rangle) = \langle \Sigma\, A, h\rangle$, where $h = t$ if $q^2(c) = t$ for some $c \in \mathcal{D}_\alpha$, and $h = f$ otherwise.

The last of these clauses actually replaces 3.4.4.7.

With these changes in the definitions, the proofs of the versions of Theorems 3.3 and 3.4 in [1] that correspond to $\mathcal{T}^*$ may be obtained by routine modifications to the arguments in [1]. From these Theorems, Proposition A.3 follows easily.

Bibliography

[1] P. B. Andrews, "Resolution in Type Theory," Journal of Symbolic Logic 36, 1971, 414 – 432.

[2] P. B. Andrews, "General Models and Extensionality," Journal of Symbolic Logic 37, 1972, 395 – 397.

[3] P. B. Andrews, "Theorem Proving via General Matings," JACM 28(2), 1981, 193 – 214.

[4] P. B. Andrews, D. A. Miller, E. L. Cohen and F. Pfenning, "Automating Higher-Order Logic" in *Automated Theorem Proving: After 25 Years*, AMS Contemporary Mathematics Series 29, 1984, 169 – 192.

[5] K. R. Apt and M. H. van Emden, "Contributions to the Theory of Logic Programming," JACM 29(3), 1982, 841 – 862.

[6] H.P. Barendregt, *The Lambda Calculus: Its Syntax and Semantics*, North Holland Publishing Co., 1981.

[7] W. W. Bledsoe, "A Maximal Method for Set Variables in Automatic Theorem-Proving," in *Machine Intelligence 9*, edited by J. E. Hayes, D. Michie, and L. I. Mikulich, Halstead Press, 1979, 53 – 100.

[8] R. S. Boyer and J. S. Moore, "The Sharing of Structure in Theorem Proving Programs," in *Machine Intelligence 7*, edited by D. Michie and B. Meltzer, Halstead Press, 1972, 101 – 116.

[9] N. G. De Bruijn, "Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem," Proceedings, Series A, 75, No 5 and Indag. Math., 34, No 5, 1972.

[10] A. Church, "A Formulation of the Simple Theory of Types," Journal of Symbolic Logic 5, 1940, 56 – 68.

[11] W. F. Clocksin and C. S. Mellish, *Programming in Prolog*, Springer Verlag, 1981.

[12] J. Darlington and R. Burstall, "A System which Automatically Improves Programs," Proceedings of the Third International Joint Conference on Artificial Intelligence, 1973, 479 – 484.

[13] S. Fortune, D. Leivant, and M. O'Donnell, "The Expressiveness of Simple and Second-Order Type Structures," JACM 30(1), 1983, 151 – 185.

[14] D. M. Gabbay, and U. Reyle, "N-Prolog: An Extension to Prolog with Hypothetical Implications. I," Journal of Logic Programming 1, 1984, 319 – 355.

[15] G. Gentzen, "Investigations into Logical Deduction," in *The Collected Papers of Gerhard Gentzen*, edited by M. E. Szabo, North-Holland Publishing Co., 1969, 68 – 131.

[16] W. D. Goldfarb, "The Undecidability of the Second-Order Unification Problem," Theoretical Computer Science 13, 1981, 225 – 230.

[17] M. J. Gordon, A. J. Milner, and C. P. Wadsworth, *Edinburgh LCF*, Springer Verlag, 1979.

[18] W. E. Gould, "A Matching Procedure for ω-Order Logic," Scientific Report No. 4, A F C R L (1976) 66 – 781, AD 646 560.

[19] L. Henkin, "Completeness in the Theory of Types," Journal of Symbolic Logic 15 (1950), 81 – 91.

[20] G. P. Huet, "The Undecidability of Unification in Third Order Logic," Information and Control 22(3), 1973, 257 – 267.

[21] G. P. Huet, "A Mechanization of Type Theory," Proceedings of the Third International Joint Conference on Artificial Intelligence, 1973, 139 – 146.

[22] G. P. Huet, "A Unification Algorithm for Typed λ-Calculus," Theoretical Computer Science 1, 1975, 27 – 57.

[23] G. P. Huet and B. Lang, "Proving and Applying Program Transformations Expressed with Second-Order Patterns," Acta Informatica 11, (1978), 31 – 55.

[24] S. C. Kleene, *Mathematical Logic*, John Wiley & Sons, Inc., 1967.

[25] C. L. Lucchesi, "The Undecidability of the Unification Problem for Third Order Languages," Report C S R R 2059, Dept. of Applied Analysis and Computer Science, University of Waterloo, 1972.

[26] D. A. Miller, "Proofs in Higher-order Logic," Ph. D. Dissertation, Carnegie-Mellon University, 1983.

[27] D. A. Miller and G. Nadathur, "Some Uses of Higher-Order Logic in Computational Linguistics," Proceedings of the 24th Annual Meeting of the Association for Computational Linguistics, 1986, 247 – 255.

[28] D. A. Miller, "A Theory of Modules for Logic Programming," Proceedings of the Symposium on Logic Programming, 1986, 106 – 115.

[29] R. Milner, "A Theory of Type Polymorphism in Programming," Journal of Computer and System Sciences 17, 1978, 348 – 375.

[30] P. Mishra, "Towards a Theory of Types in Prolog," Proceedings of the 1984 International Symposium on Logic Programming, Atlantic City, February 1984, 289 – 299.

[31] R. Montague, "The Proper Treatment of Quantification in Ordinary English," in *Formal Philosophy: Selected Papers of Richard Montague*, edited by R. Thomason, Yale University Press, New Haven, 1974.

[32] A. Mycroft and R. A. O'Keefe, "A Polymorphic Type System for Prolog," Artificial

Intelligence 23(3), 1984, 295 – 307.

[33] F. C. N. Pereira and D. H. D. Warren, "Definite Clause Grammars for Language Analysis – A Survey of the Formalism and a Comparison with Augmented Transition Networks," Artificial Intelligence 13, 1980.

[34] L. C. Paulson, "Natural Deduction as Higher-Order Resolution," The Journal of Logic Programming 3(3), 1986, 237 – 258.

[35] F. Pereira, D. Warren, D. Bowen, L. Byrd, and L. Pereira, "C-Prolog User's Manual: Version 1.5," February 1984.

[36] J. C. Reynolds, "Three Approaches to Type Structure," Proceedings of the International Joint Conference on Theory and Practice of Software Development, March 1985.

[37] J. A. Robinson, "A Machine-Oriented Logic based on the Resolution Principle," JACM 12, 1965, 23 – 41.

[38] J. A. Robinson, "Mechanizing Higher-Order Logic," in *Machine Intelligence 4*, edited by B. Meltzer and D. Michie, Halstead Press, 1969, 151 – 170.

[39] R. M. Smullyan, *First-Order Logic*, Springer-Verlag, New York, 1968.

[40] J. E. Stoy, *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, The MIT Press, Cambridge, 1977.

[41] M. H. van Emden and R. A. Kowalski, "The Semantics of Predicate Logic as a Programming Language," JACM 23(4), 1976, 733 – 742.

[42] D. H. D. Warren, "Implementing Prolog - Compiling Predicate Logic Programs," D. A. I. Research Report Nos 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

[43] D. H. D. Warren, "Higher-order Extensions to PROLOG: Are They Needed?," *Machine Intelligence 10*, 1982, 441 – 454.

[44] D. H. D. Warren, "An Abstract Prolog Instruction Set," Technical Note 309, SRI International, October 1983.

[45] D. S. Warren, "Using λ-Calculus to Represent Meaning in Logic Grammars," Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics, 1983, 51 – 56.

List of Defined Terms