A Proof-theoretic approach to logic programming

Miguel Suarez

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1992
25ème anniversaire

N° 1818

Programme 2
Calcul Symbolique, Programmation
et Génie logiciel

A PROOF-THEORETIC APPROACH TO LOGIC PROGRAMMING

Miguel SUAREZ

Décembre 1992

RR-1818

A Proof–Theoretic Approach to Logic Programming

Une approche de la Programmation en Logique par la Théorie de la Preuve

Miguel Suarez
INRIA - Rocquencourt
B.P. 105
78153 Le Chesnay, FRANCE

Abstract. We develop the foundations of Horn clause Logic Programming in a proof-theoretic style. We present a formal unification system for first order terms including a syntactic representation of environments, and formalisms for the bottom-up and top-down computation of answers in a unified formal setting. In passing, we propose an abstract unification machine. We also discuss the possibility and implications of representing logic program evaluation strategies in the Lambda Calculus. **Key words:** Logic Programming, Proof Theory, Lambda Calculus, Abstract Machine

Résumé. Nous développons les fondements de la Programmation en clauses de Horn dans le style de la Théorie de la Démonstration. Nous presentons un formalisme pour l'unification de termes de premier ordre incluant une représentation syntaxique des environments, et des formalismes pour le calcul chaînage avant et chaînage arrière de réponses dans un cadre unifié. Nous proposons une machine abstraite pour l'unification. Nous discutons la possibilité et les consequences de la représentation de strategies d'évaluation dans le Lambda-Calcul. **Mots clés:** Programmation en Logique, Théorie de la Démonstration, Lambda-Calcul, Machine abstraite.

Chapter 1

Introduction

In Horn clause Logic Programming, we define relations (or predicates) by means of syntactic objects called *Horn clauses* (to be defined later). A *program* is simply a finite set of Horn clauses. Once a program is given, we have the possibility of asking questions through *queries*. For instance, if the program P defines a binary predicate p, we can ask the following questions:

i. Is it true that the objects a and b are in the relation p defined by P, or, simply, $p(a, b)$?

ii. What are the objects x such that $p(x, b)$?

iii. What are all the pairs in the relation p, or, $p(x, y)$?

In *i* we demand a yes/no answer. In *ii*, however, we demand that the specified objects be listed. For each object a, we expect an *answer* of the form $[x/a]$, telling us that $p(a, b)$ holds. Similarly, in *iii*, we expect answers of the form $[x/a, y/b]$. In the Logic Programming literature[Llo87, Pad88] we can find means of defining these notions. Thus, the giving of a Horn clause program and a query precisely defines an 'evaluation problem'. A program defines a set of evaluation problems, one for each possible query.

The next step is to define algorithms for solving such computational problems. An evaluation algorithm (or strategy) is said to be *sound* on the class C of evaluation problems when for all problems in C, it returns only specified answers; it is *complete* on C when for each problem in C, it is capable of returning essentially all specified answers.

It is a known fact that there exist evaluation strategies sound and complete on the class of all the Horn clause evaluation problems. This is a nice and reassuring theoretical result. Unfortunately, such universal strategies are of limited practical use. Taking into account the expressive power of Horn clauses and some computability results, we can affirm that there exists no sound, complete and always terminating Horn clause evaluation strategy. This means that any universal, sound and complete evaluation strategy is bound to not terminate on some problems. The completeness of the strategy only ensures that, if given enough time, the algorithm will compute all demanded answers. The problem is that there may exist no algorithm capable of (correctly) deciding when to stop. There is another obvious practical

limitation, that of efficiency. It is clear then that we need to develop specialized evaluation strategies.

Prolog is such a specialized strategy. Typically, it is defined as follows. First, we define an *SLD-Resolution formalism*. Also, for any given program and query, we define the notion of *SLD-tree,* and define the Prolog evaluation of the given query with respect to the given program as the depth-first, left-right traversal of the corresponding SLD-tree. The main virtue of Prolog is that there exist efficient implementations of it. Its main defect is that it is incomplete and too often non-terminating. The Cut operator has been introduced as a means of modifying the default Prolog strategy, in this way defining new evaluation strategies. For lack of a better name, here we will call these strategies *the Cut-definable strategies.* Good texts on Prolog[CM87, SS90] recommend a careful use of the Cut, since it may produce unexpected effects. A typical argument against its use is that it is 'operational', or 'extra-logical', hence inadequate in a declarative environment. We argue that the Cut is not so extra-logical as it seems. The problem is of a different nature.

The Declarative Paradise, where it is possible to specify a computational problem and ignore the question of defining the evaluation strategy, is a myth. At computation time, *some* algorithm is needed, and if it is to be of any practical use, it must be fast enough. Concerning Horn clause Logic programming, once we accepted that no universal strategy is practically acceptable, we must seriously consider the question of precisely defining 'fast enough' specialized evaluation strategies, and tools for proving that they are safe on the intended class of evaluation problems. In our view, the actual difficulty is that we currently lack rigurous, powerful, and general enough techniques for *defining* and *reasoning* about specialized strategies. There are two branches of Mathematical Logic that may be of great help to this purpose: Proof Theory, and the Lambda Calculus.

Proof theory is logic from the syntactic (formal) viewpoint. The raw materials of proof theory are (in general, finite) syntactic objects called *formulas,*, and proof (or deduction) rules, which are syntactical (formal, mechanical) means of proving (deducing, producing) new formulas from given formulas. We call *formal theory* any set of formulas, the *axioms* of the theory, supplemented with a set of proof rules. With a theory at hand, we can define the notion of *proof* in this theory. The exact definition depends on the exact nature of the formulas and rules. An acceptable approximation is to say that a proof in the theory $\mathcal{T}_0$ is a finite sequence of formulas, each of which is either an axiom of $\mathcal{T}_0$, or is obtainable from previous formulas in the sequence by some rule of the theory. A proof is then a finite syntactic object. Each proof defines a *theorem* of the theory, namely, its last formula. We write $\mathcal{T}_0 \vdash \alpha$ for expressing that α is a theorem of $\mathcal{T}_0$. This is the syntactic (formal) side. On the semantic (informal, but rigurous) side, we introduce the notion of *truth,* allowing us to define what it means for a formula α to be *true in* $\mathcal{T}_0$, that we write $\mathcal{T}_0 \models \alpha$. A typical problem is to ensure that the formally provable formulas are the same as the true formulas. In symbols, we try to ensure that the statement

$$\text{for all formulas } \alpha, \qquad \mathcal{T}_0 \models \alpha \qquad \text{if and only if} \qquad \mathcal{T}_0 \vdash \alpha. \tag{1.1}$$

holds for the system at hand. Note that here we have two levels of reasoning. The formal level, the proofs in $\mathcal{T}_0$, and the informal but hopefully rigurous level, the reasoning allowing us to verify the validity of (1.1). Note also that (1.1) is a statement *about* $\mathcal{T}_0$. We say that it is a *meta-theoretic* property. The distinction, however, is not absolute. Proof Theory also studies the possibility of formalising the proofs of certain meta-theoretic properties. Given a theory $\mathcal{T}_0$, and a meta-theoretic statement like (1.1), the idea is to find a formal theory $\mathcal{T}_1$, and a formula β of this last theory, such that $\mathcal{T}_1 \vdash \beta$ implies (1.1). This may seem an awkward idea. In our opinion, however, it may be quite useful for studying Logic Programming strategies. Let us justify our claim.

Typical versions of SLD-Resolution[Llo87, Pad88] have two main defects. On the one hand, they lack a syntactic representation of answers, and they fail to formally represent the unification operation. Said otherwise, in such systems, the unification operation is informal and external to the system. We present here a version of SLD-Resolution where these gaps have been filled in. This is based on a formal unification system, and a finite syntactic notion of *environment*. In passing, this allows us to define an abstract unification machine that may be of interest to Prolog compiler writers, and in general, logic programming implementors. We also prove the classical results concerning SLD-Resolution in this new setting. Among these results, the completeness of SLD-Resolution, which roughly says that given a program P and a query Q, we can construct an SLD-theory $\mathcal{T}$, depending on P and Q, such that the set of all the proofs in $\mathcal{T}$ encode all the answers we need in a well defined way. Next, we identify a particular kind of proofs, that we call the *canonical* SLD-proofs, which is a proper subset of the set of all the proofs, but that also contains all the answers we need. The set of all the canonical proofs, ordered in a natural way, is a tree, that we define to be the SLD-tree of the given query Q relative to the given program P. Note then that with this definition each node of the tree is an SLD-proof. This allows us to define Prolog as a *canonical-SLD-proof generating* strategy. For this, it suffices to define the Prolog-successor relation $\succ$ between canonical proofs, and say that the Prolog strategy simply consists in the set of all the proofs reachable from the root of the SLD-tree by a finite number of $\succ$-steps, set which is totally ordered by $\succ$. We argue that the Cut-definable strategies can also be defined in this way.

This may seem a pedantic definition of Prolog, but it has interesting consequences. In particular, it makes possible the application of the technique of *Gödelnumbering*, which consists in representing syntactic objects by natural numbers. In this way, we can transform statements involving syntactic objects (in particular, meta-theoretic statements about a given theory) into statements about the set $\mathcal{N}$of natural numbers. For instance, a Horn clause program is a finite syntactic object, hence may be represented by a natural number p (its Gödel number). Analogously, a query Q may be represented by a number q. Now, consider a strategy s defined as above. An SLD-proof is a finite syntactic object, hence representable through a number. The full strategy may involve an infinity of proofs, but the list of the first $n+1$ proofs of the strategy is a finite syntactic object, hence representable by a number $s(n)$ depending on s and n. Since we can represent answers by finite syntactic objects, an answer may be represented by a number. Now, consider the following statement

about the natural numbers:

> for all a, if a is (the Gödel number of) an answer for the query
> (of Gödel number) q with respect to the program p, then
> there exists an n such that a is among the answers encoded in $s(n)$.

Even if we have not given full details, this suggests that the completeness of the given strategy may be expressed through a property of the natural numbers. Since there are formal theories for formally reasoning about $\mathcal{N}$, this shows the possibility of rigurously (even formally) proving properties of evaluation strategies when these are defined as suggested above. We remark that we have not yet used this idea. What interests us here is to explore the applicability of typical proof-theoretic and Lambda Calculus techniques to the study of Logic Programming problems.

There are other interesting possibilities that merit attention. By definition, from the set-theoretic point of view, a unary function f is a set of pairs $(x, f(x))$, that is, a function *is* its graph. According to [Bar84], one of the aims of the founders of the Lambda Calculus was the study of the idea of 'function as rule'. The basic objects of the Lambda Calculus are the *lambda terms*. Let us take a denumerable list of variables, and the symbols 'λ', '(', ')'. By definition, (we follow the syntax of [Kri90]), a variable $\mathbf{x}$ is a (lambda) term, and if $\mathbf{t}$, $\mathbf{u}$ are terms, then $\lambda\mathbf{x}\ \mathbf{t}$, and $(\mathbf{t})\mathbf{u}$ are terms. Intuitively, $\lambda\mathbf{x}\ \mathbf{t}$ denotes a unary function, whose value on the object (denoted by) $\mathbf{u}$ is $(\lambda\mathbf{x}\ \mathbf{t})\mathbf{u}$. There is, however, the problem of defining the equality of values, since two (syntactically) different terms may want to represent the same value. Technically, we introduce *conversion* (or *reduction*) rules, and try to convert $(\lambda\mathbf{x}\ \mathbf{t})\mathbf{u}$ into a *normal* term $\mathbf{w}$, that is, a term to which none of the 'relevant' conversion rules may be applied. If this reduction is possible, at least in principle, we can say that the value of $(\lambda\mathbf{x}\ \mathbf{t})\mathbf{u}$ is its normal form $\mathbf{w}$. Of course, a conversion process is a computation process. Conversely, given a known computation process, we may consider the question of representing it in the Lambda Calculus for studying its properties. In particular, since a conversion strategy may be non-terminating, such representations allow us to represent non-terminating computations. Now recall our discussion on Prolog. We said that we can define Prolog through the 'Prolog successor' relation $\succ$, the notation '$\pi_0 \succ \pi_1$' meaning that the proof π_1 is the Prolog successor of π_0. With the preceding discussion in mind, the analogy is evident: '$\pi_0 \succ \pi_1$' now is read: the lambda term π_0 is $\succ$-convertible into π_1. This idea led us to the Abstract Prolog Machine (APM). The APM is not concerned with questions like memory, registers, or data representation. An APM state is a syntactic object, and the only APM instruction is a conversion rule $\succ$. Even if at present we have not the full details, we argue that coding the APM in the Lambda Calculus is only a matter of time. In our view, however, for this representation to be truly useful, it must allow us to transform the termination of Prolog into a normalisation problem. Without proof, we argue that it is possible to define

i. a conversion rule $\succ$ on lambda terms,

ii. a recursive function mapping any given Horn clause program P into a lambda term $\mathbf{p}$, and

iii. a recursive function mapping any given query Q into a lambda term $\mathbf{q}$

in such a way that the termination of the Prolog evaluation of Q with respect to P is equivalent to the statement: $\succ$ is a $\succ$-normalisation strategy for $(\mathbf{p})\mathbf{q}$, that is, that we can transform $(\mathbf{p})\mathbf{q}$ into a $\succ$-normal term by a finite number of $\succ$-reductions. More, we argue that all the Cut-definable strategies may be represented in an analogous way. The theoretical consequences of such representations are evident: such representations are tools for precisely defining and studying evaluation strategies. In fact, we think that $\succ$ must be some special case of β, or $\beta\eta$ conversion. For instance, $\succ$ may be simply the left β-conversion. If this is done in this way, interesting consequences follow. For instance, [Kri90] presents type systems for the Lambda Calculus. In these systems, a type expression is a syntactic object of the form $\Gamma \vdash \mathbf{t} : A$, where Γ is a (possibly empty) *context* whose exact nature is irrelevant here, $\mathbf{t}$ is a Lambda-term, and A is a *type*, a formula in a language which depends on the system. A type system also includes rules, allowing us to construct type expressions. If $\Gamma \vdash \mathbf{t} : A$ may be constructed in system $\mathcal{S}$, then $\mathbf{t}$ is said of type A in the context Γ in system $\mathcal{S}$. As is shown in [Kri90], the left β reduction of a term $\mathbf{t}$ is terminating if and only if $\mathbf{t}$ may be typed in a *special way* in system $\mathcal{D}\Omega$, where *special way* is a certain syntactic condition. This means that if Prolog may be represented as described above, for any program P and query Q, the Prolog evaluation of Q with respect to P is terminating iff $\mathbf{p}(\mathbf{q})$ may be typed in a *special way* in $\mathcal{D}\Omega$. This being done, the *special* typability in $\mathcal{D}\Omega$ becomes a formal tool for guaranteeing the termination of Prolog.

Let us stop for a while and review the main ideas of the preceding discussion. The ability of precisely specifying evaluation problems is certainly a step towards a more declarative programming style, but the use of unpredictable evaluation strategies is clearly not a step in the same direction. We need precise formal tools for defining and reasoning about evaluation strategies. But the formalisms themselves (Proof theory, Lambda Calculus) are not enough. We also need general principles for defining evaluation strategies. For instance, Prolog is a top-down (backward) strategy, and there also exist bottom-up (forward) strategies. Let us clarify these ideas for readers not used to them. Think a proof rule r as a (for the purposes of this discussion) binary function. The natural use of such a rule is its forward use: given F_0 and F_1, we compute $r(F_0, F_1) = F_2$. This is clearly suggested by the notation $r(F_0, F_1) = F_2?$, meaning that when the rule is applied, the known data are F_0 and F_1, while F_2 is to be computed. Thus, the forward application of r is 'from the axioms towards the theorems'. The opposite strategy is also possible. This is suggested by the expression $r(F_0?, F_1?) = F_2$. Here we start with a candidate theorem and try to construct a proof backwards. Referring to the preceding equation, once we computed F_0 and F_1 making this equality to hold, our problem will be solved once we verified that F_0 and F_1 are themselves theorems. Interestingly enough, it is possible to precisely define complete 'mixed' evaluation strategies, where the forward and backward paradigms coexist. More precisely, we argue that given a program, a query, and an arbitrarily selected set of predicates, it is possible

to define an evaluation strategy computing all demanded answers in such a way that the selected predicates are evaluated top-down, while the others bottom-up. More, even if at present we have not all details, we are firmly persuaded that these strategies can also be represented in the Lambda Calculus following the ideas described above.

We said before that a Lambda Calculus representation of an evaluation strategy gives us a precise formal definition of it, hence the possibility of precisely studying its properties in a sophisticated formal setting like the Lambda Calculus. But there are also practical implications. If we have defined conversion rules and term constructing functions in the manner suggested above, then we have (implicitly or explicitly) identified a set of abstract operations (an abstract instruction set) powerful enough for implementing mixed strategies, in particular Prolog, and we have a compilation algorithm such that, given a program P, and a specification of a strategy, generates machine code $\mathbf{p}$ which, feeded with the appropriate representation $\mathbf{q}$ of any given query Q, is capable of implementing the evaluation of the argument query Q with respect to the compiled program P following the specified strategy. The Prolog languages (actual implementations of the Prolog strategy) are useful programming tools, but may be criticised for not clearly separating the specification (logic) component from the operational (control) component. The previous discussion suggests the possibility of explicitly separating both components through the precise definition of a Logic Programming control language. In principle this appears to be a return to the good old algorithmic approach, but all depends on how this idea is applied. Once we have precise means for defining different evaluation strategies, we can dream of finding general results concerning properties of strategies, results of the form 'strategy s is sound, complete and terminating on all the evaluation problems of the class $\mathcal{C}$'. Since a program defines a class of evaluation problems, with such knowledge at hand we have the possibility of developing sophisticated Logic Programming compilers.

Let us explore this idea. Suppose that for all queries $Q \in D$, the evaluation of Q with respect to P following strategy s terminates. We can then define a function f such that for all $Q \in D$, $f(Q)$ is the finite list of the (candidate) answers returned during this evaluation. Suppose that we have two formulas $D[x]$ and $E[x]$ such that if

$$(\forall Q)(\, D[Q] \rightarrow E[f(Q)]\,)$$

then

$\forall Q \in D$, $f(Q)$ contains only specified answers, and essentially all specified answers. (1.2)

That is, suppose that the validity of the above formula implies that strategy s is sound, complete, and terminating on D. Suppose also that we can express the function f by a lambda term $\mathbf{t}$. Then we can dream of developing a formal system with the property that a proof of the formal expression

$$\vdash \mathbf{t} : \forall x\,(\, D[x] \rightarrow E[f(x)]\,) \tag{1.3}$$

in this system imply (1.2). Such a system would be a tool for formally proving the soundness, completeness, and termination of s on a well defined class of problems. Interestingly enough, there already exists a good candidate for this kind of applications. It is the system AF_2 presented in [Kri90]. Our description here will be only approximate. We simply intend to convey the idea of using AF_2 in this manner.

AF_2 is a type system as described above. In this case, the types are formulas of second order predicate calculus. AF_2 also consists in a certain number of formal (typing) rules. Here we will write $AF_2 \vdash \mathbf{t} : T$ for expressing that $\vdash \mathbf{t} : T$ may be constructed with the rules of AF_2.

Certain formulas are said to *define a data type*, and the corresponding sets are said to be *data types*. Suppose now that we have a function $f : D \rightarrow E$, where the data types D and E are defined by $D[x]$, and $E[x]$, respectively. Then, following [Kri90] (again, this is only an approximate statement), if we find a term $\mathbf{t}$ such that

$$AF_2 \vdash \mathbf{t} : \forall x (D[x] \rightarrow E[f(x)]) \tag{1.4}$$

then the term $\mathbf{t}$ defines an algorithm computing f on the domain D.

Each rule of AF_2 may be partitioned into two independent parts, the term part, and the type part. Restricted to their type parts, the rules of AF_2 define a second order intuitionistic proof system, such that if we have a proof of the type part of (1.4) in this system, then we can transform this proof into a term (algorithm) $\mathbf{t}$ satisfying (1.4). Hence, if we are looking for an algorithm computing $f : D \rightarrow E$, it suffices to find an intuitionistic proof of the type of (1.4), and transform this proof into an algorithm (lambda term) as suggested above. This is a *verification directed* programming style, since instead of writing an algorithm, we construct a proof of a certain specification in a certain formal system. This is a quite impressive result, but its practical use in the short term is unlikely, since, at least for the moment, it appears difficult to control the efficiency of the generated algorithm.

But AF_2 can also be used as a verification system. We first write an algorithm $\mathbf{t}$, and next we try to prove (1.4). If we succeed, our algorithm is correct. Thus, expressing logic programming evaluation strategies in the Lambda Calculus opens the possibility of using AF_2 (or some specialized version of it) as a formal verification system. Of course, it also opens the possibility of representing logic program evaluation strategies and functional programs in a unified formal setting. In our view, this is a quite interesting, and, to our knowledge, unexplored research direction. Its development obviously depends on our ability of expressing logic program evaluation strategies in the Lambda Calculus. As we remarked above, we are firmly persuaded that, at least for Prolog and the Cut-definable strategies, this will be done in the near future.

These are the ideas underlying this work. We are well aware that some of our claims above are highly speculative, but others are simply intuitive descriptions of precisely stated and proved results. This is the subject of this report. Unfortunately, due to limitations in time, we were not able to include full details. Hopefully, after reading this work, the reader will agree with us in that the speculations above deserve serious consideration.

A final remark concerning this report. It seems to us that the interaction between the Logic Programming community on one side, and the Proof Theory/Lambda Calculus communities on the other side is still limited. It is possible then that researchers of one field be unfamiliar with elementary notions of the other field. Since we address both communities, in particular students and young researchers, we decided to start (almost) from scratch, trying to make this report as self-contained as possible, in the limits of time and space we had to respect.

Chapter 2

The Background

2.1 First Order Languages

In this chapter we present some basic definitions and results that we will need in what follows. The reader is supposed to be familiar with the basics of first order predicate calculus. Here we will simply recall some notions in order to fix our terminology and notation. We also list some results for reference purposes. In general, we will not include proofs that may be done in a few lines using preceding definitions and results and standard proof techniques, like induction on terms and formulas.

A first order language (fol) is a set of symbols partitioned into two disjoint classes: the *logical* symbols and the *non-logical* symbols, also called *proper* symbols. As usual, we will suppose that the logical symbols are the same in all the languages we consider.

We take the logical symbols from among the following list: a certain number of *connectives,* typically $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies), the quantifiers $\forall$ (universal quantifier), $\exists$ (existential quantifier), the auxiliary symbols left and right parentheses and comma, and a denumerable set $\mathcal{V}$ of *individual variables.*

The non-logical symbols come in three disjoint classes: *individual constants, function symbols* and *predicate symbols.* Each function and predicate symbol is associated a positive integer, its *arity.* We suppose that a fol contains at least one predicate symbol.

When a fol is under consideration, we need to differentiate between two languages: the fol we are considering, the *object language,* and the language we use in our discussion, here english supplemented with some special signs. This is called the *metalanguage.* In order to avoid confusion, the meaning of the special signs of the metalanguage must be precisely defined. Here are some conventions we will follow, others will appear later. x, y, z (and their subscripted variants) are signs of the metalanguage standing for variables of the object language. They are *metavariables* ranging over the set of object language variables. a, b, c, are metavariables ranging over the (individual) constants. f, g, represent function symbols, p, q, represent predicate symbols.

With the symbols of a fol, we may form *expressions,* that is, finite sequences of symbols.

Two kinds of expressions will receive particular attention: the *terms* and the *clauses*.

A *term* is either a variable, or a constant, or an expression of the form $f(t_1, \ldots, t_n)$, where f is an n-ary function symbol and the t_i's are terms. We will use t, u, and v as metavariables representing arbitrary terms of the language under discussion. The *complexity* of a term t, $\#t$, is the number of occurrences of function symbols in t. More precisely, if t is either a variable or a constant, then $\#t = 0$, and

$$\#f(t_1, \ldots, t_n) = 1 + \sum_{i=1}^{n} \#t_i.$$

A complex term is a term of positive complexity; it is then of the form $f(t_1, \ldots, t_n)$ and in this case we say that f is its *main* function symbol. We write $t \approx u$ and say that t and u are *compatible* iff they are complex terms with the same main function symbol. We denote by $var(t)$ the set of variables appearing in t. A term t is said to be *closed* iff $var(t) = \emptyset$. If T is a set of terms, $var(T)$ is the set of variables appearing in some term of T, that is

$$var(T) = \bigcup_{t \in T} var(t).$$

For reasons that will be apparent below, our definition of *clause* is different from the one normally used. However, it is equivalent in a sense explained later.

Definition 2.1 *A* clause *is either an* atom, *that is, an expression of the form $p(t_1, \ldots, t_n)$, where p is an n-ary predicate symbol and the t_i's terms, or is an implication of the form*

$$A_0 \rightarrow (A_1 \rightarrow \ldots \rightarrow (A_{n-1} \rightarrow A_n) \ldots) \tag{2.1}$$

where the A_i's are atoms and $\rightarrow$ is the usual implication symbol. A_n is said to be the head *of clause (2.1), the head of an atom being the atom itself.*

We use the letters A, B, C, D and their subscripted variants for representing atoms. The full notation (2.1) is too heavy to be systematically used, so we will also represent such a clause by the notation $A_0 \ldots A_n$. Note that this is not a change in the definition of *clause* but a metalanguage notational convention. The lowercase greek letters α, β, γ, and δ, possibly subscripted, will represent arbitrary clauses, while $A \rightarrow \alpha$ and analogous notations will represent arbitrary implications. The head of α will be denoted by $\widehat{\alpha}$. If $\alpha = A_0 \ldots A_n$, then for $j = 0, \ldots, n$, α^j denotes the clause $A_j \ldots A_n$. The clauses α^j and only them will be called *right parts* of α. Note that $\alpha^0 = \alpha$, $\alpha^n = \widehat{\alpha}$, and for $j < n$, $\alpha^j = A_j \rightarrow \alpha^{j+1}$. As for terms, $var(\alpha)$ represents the set of variables appearing in α. The same notation will be used with other syntactic objects without further mention. We extend the compatibility relation to clauses: we write $\beta \approx \delta$ iff $\widehat{\beta}$ and $\widehat{\delta}$ are atoms on the same predicate. Applied to atoms, the definition says that two atoms are compatible iff their predicates are the same.

2.2 Substitutions

The notion of *substitution* will be essential in what follows.

Definition 2.2 *A* substitution *is a total function θ from the set of terms into the set of terms satisfying the following properties:*

i. for all constants c, $c\theta = c$,

ii. for all complex terms $f(t_1, \ldots, t_n)$, $f(t_1, \ldots, t_n)\theta = f(t_1\theta, \ldots, t_n\theta)$.

The identity substitution, which maps each term onto itself, will be denoted by id. Two substitutions are equal when they are equal as functions, that is $\theta = \xi$ iff for all terms t, $t\theta = t\xi$. Once we adopted the postfix notation for substitutions, it is natural to denote by $\theta\xi$ the composition of θ (first) and ξ (second); in symbols, for all t, $t\theta\xi = (t\theta)\xi$. It is immediate that the composition of substitutions is a substitution. By definition, $\theta^0 = id$, and for $k > 0$, θ^k is the composition $\theta \ldots \theta$, k times. An *idempotent* substitution is a substitution θ satisfying $\theta\theta = \theta$. If T is a set of terms, $T\theta = \{ t\theta \mid t \in T \}$. The following statement precisely characterises the set of variables appearing in $t\theta$:

$$var(t\theta) = \bigcup_{x \in var(t)} var(x\theta).$$

Take, for exemple, $f(x, g(y))\theta = f(x\theta, g(y\theta))$. Clearly, the variables in this term are those in $var(x\theta)$ and those in $var(y\theta)$.

A substitution applied to a term only affects its variables, thus if t is closed, $t\theta = t$ for all θ. This property offers an intuitive justification for our terminology: if t contains no variable, no substitution is allowed to 'enter'. We will also need to consider terms containing at least one variable, that is, those that are not closed. We will find then convenient to choose a short name for this kind of terms. By analogy, the choice imposes itself: a term t is *open* iff $var(t) \neq \emptyset$.

We will be interested in talking about equality of substitutions on a given set V of variables. By definition, $\theta_1 = \theta_2$ on V iff for all $x \in V$, $x\theta_1 = x\theta_2$. Since the image of t under θ depends only on the action of θ on the variables of t, we get

Proposition 2.3 *$\theta_1 = \theta_2$ on $var(t)$ if and only if $t\theta_1 = t\theta_2$.* □

As a consequence, $\theta = id$ on $var(t)$ iff $t\theta = t$. Also, if $\theta_1 = \theta_2$ on V, and $var(t) \subseteq V$, then $t\theta_1 = t\theta_2$.

The reader used to the definition of *substitution* normally found in the Logic Programming literature, may feel a little uncomfortable with the definition we presented above. For the purposes of the following discussion, let us call 'standard' substitution, any function from a finite set of variables into the set of terms, denoted by

$$[x_1/t_1, \ldots, x_n/t_n], \tag{2.2}$$

the notation telling us that the domain is $\{ x_1, \ldots, x_n \}$, and that the image of x_i is t_i, for each i. Once introduced, such a partial function is extended to all terms by defining what it means to 'apply' (2.2) to an arbitrary term. If this extension is a substitution, in our sense, it is natural to identify the partial function with its extension. But for the identification to be meaningful, we need to uniquely determine the substitution with which (2.2) is to be identified. The following result ensures that there exists at least one such substitution.

Proposition 2.4 *If f is a function mapping a set V of variables into the set of terms, then f may be extended to a substitution.* □

But the question of the unicity is still open. The following is an immediate consequence of our definition of *substitution.*

Proposition 2.5 *If $\theta_1 = \theta_2$ on V then $\theta_1 = \theta_2$.* □

Otherwise said, for two substitutions to be equal (on all terms), it suffices that they be equal on the set of all variables. But then we have:

Corollary 2.6 *A function from the set of all variables into the set of terms may be extended to exactly one substitution.* □

Simply observe that two substitutions extending such a function are equal on all variables, hence they are equal. What is usually done with (2.2) is to first extend it to all variables making it equal to the identity on the variables not in $\{ x_1, \ldots, x_n \}$, and then to extend this new function to the set of all terms using the conditions of our definition. This extension is then a substitution and, according to the last corollary, it is uniquely determined.

The previous results say that we may define a substitution simply by defining it on all variables, possibly with the help of other substitutions. A definition schema we will find relatively frequently is the following: given θ and ξ two substitutions and V a set of variables, we define a new substitution ϕ by the condition

$$x\phi = \begin{cases} x\theta & \text{if } x \in V. \\ x\xi & \text{otherwise.} \end{cases}$$

We will write this kind of definitions in the form: $\phi = \theta \downarrow V + \xi$.

The *kernel* of θ is the set $ker(\theta) = \{ x \mid x\theta \neq x \}$. It is immediate that for all θ, $\theta = \theta \downarrow ker(\theta) + id$. This equality clearly expresses the fact that θ is completely determined by its action on the set $ker(\theta)$. It is also easy to convince himself that the substitutions that may be obtained by extending a 'standard' substitution in the manner described above are those of finite kernel. This readily implies that there are non-standard substitutions.

Corollary 2.7 *i. If $ker(\theta) = \{ x_1, \ldots, x_n \}$, then $\theta = [x_1/x_1\theta, \ldots, x_n/x_n\theta]$.*

ii. $ker(id) = \emptyset$.

iii. *If* $\theta = \xi$ *on* V, *then* $V \cap ker(\theta) = V \cap ker(\xi)$.

iv. *For all* θ *and* V, $\theta \downarrow V + id = \theta \downarrow V \cap ker(\theta) + id$.

v. $\theta = \xi$ *on* V *iff* $\theta \downarrow V + id = \xi \downarrow V + id$.

□

Note that the equality of substitutions on a set of variables is a ternary relation denoted following the pattern: '$\theta\, R\, \xi$ on V'. Other relations will be defined and denoted analogously. We adopt then the following convention: the notation '$\theta\, R\, \xi$' will be used as an abbreviation of '$\theta\, R\, \xi$ on $\mathcal{V}$', that is, on the set of all variables. With this convention, the notation '$\theta = \xi$' has two meanings: that θ and ξ are equal on all terms, the usual meaning, and that they are equal on the set of all variables, by the convention. However, we have seen that these two statements are equivalent, so there is no ambiguity.

Proposition 2.8 $\theta = id$ *on* V *iff* $ker(\theta) \cap V = \emptyset$. □

Corollary 2.9 $t\theta = t$ *iff* $ker(\theta) \cap var(t) = \emptyset$. □

Now we may easily prove a known characterisation of the idempotent substitutions.

Proposition 2.10 θ *is idempotent if and only if* $ker(\theta) \cap var(ker(\theta)\theta) = \emptyset$.

Proof If θ is idempotent, for all variables x, $x\theta\theta = x\theta$, and the last corollary implies $ker(\theta) \cap var(x\theta) = \emptyset$. Conversely, if $x \notin ker(\theta)$, trivially, $x\theta\theta = x\theta$, and if $x \in ker(\theta)$, by the hypothesis, $ker(\theta) \cap var(x\theta) = \emptyset$, which, with the last corollary, implies $x\theta\theta = x\theta$. □

The following is another useful property of the idempotent substitutions.

Proposition 2.11 *If* θ *is idempotent then, for all* ξ, $\theta \leq \xi$ *if and only if* $\theta\xi = \xi$. □

A special kind of substitutions is widely used in Logic Programming writings, namely, the *variable renamings*.

Definition 2.12 *A substitution* θ *is said to be a variable renaming on the set* V_0 *of variables* into *the set* V_1 *if and only if*

i. *for all* $x \in V_0$, $x\theta \in V_1$ *and*

ii. $\forall x_1 \in V_0, \forall x_2 \in V_0$, *if* $x_1 \neq x_2$, *then* $x_1\theta \neq x_2\theta$.

Trivially, *id* is a variable renaming on any set of variables. $\Omega(V_0)$ will denote the set of all variable renamings on V_0. A variable renaming on the set $\mathcal{V}$ will be called simply a *variable renaming*. The following result says that variable renamings on a set have a kind of 'partial right inverse'.

Proposition 2.13 *If* θ *is a variable renaming on* V, *there exists a substitution* ξ *such that* $\theta\xi = id$ *on* V. □

In some cases we may need to rename the variables in V_0 *without using any variable of another set* V_1. Trivially, if V_1 is finite, this is possible.

Proposition 2.14 *Let V_0 be any set of variables and V_1 a finite set of variables. Then, there exists a variable renaming θ on V_0 into $\mathcal{V} \setminus V_1$.* □

In general, if V is a set of variables and θ a substitution, the set $V\theta$ may contain arbitrary terms. However, if θ is a variable renaming on V, $V\theta$ is a set of variables and $var(V\theta) = V\theta$.

Proposition 2.15 *The composition $\theta\xi$ is a variable renaming on V if and only if θ is a variable renaming on V, and ξ is a variable renaming on $V\theta$.* □

Proposition 2.16 *If ϕ is a variable renaming on V, $x_i \in V$, and t_i is an arbitrary term, for $i = 1, \ldots, n$, then $\phi[x_1\phi/t_1\phi, \ldots, x_n\phi/t_n\phi] = [x_1/t_1, \ldots, x_n/t_n]\phi$ on V.* □

The following result generalises a preceding remark. We will use its corollary.

Proposition 2.17 *Let T be a set of terms and θ a substitution such that $t\theta = t$ for all $t \in T$. Then $\theta = id$ on $var(T)$.* □

Corollary 2.18 *If $\xi\theta_1\theta_2 = \xi$ on V, then $\theta_1 \in \Omega(var(V\xi))$.*

Proof Let $T = V\xi$. The hypothesis implies that for all $t \in T$, $t\theta_1\theta_2 = t$. The proposition now implies $\theta_1\theta_2 = id$ on $var(T)$. But id is a variable renaming, so according to Proposition 2.15, θ_1 is a variable renaming on $var(T) = var(V\xi)$. □

We are going to consider more special variable renamings by demanding a useful additional property. As we stated above, a substitution is a function from terms to terms; in particular, it may be a bijection from the set of terms onto itself. For instance, id is a bijective substitution. The following property is easily proved.

Proposition 2.19 *A bijective substitution maps different variables onto different variables. As a consequence, a bijective substitution is a variable renaming on any set of variables.* □

Ω will denote the set of all bijective substitutions. The convenience of using bijective substitutions is that if $\theta \in \Omega$, then its inverse θ^{-1} exists and is obviously bijective. But it is not immediately obvious that θ^{-1} is itself a substitution. We must verify that this is the case, but this is easily done.

Proposition 2.20 *The inverse of a bijective substitution is a (bijective) substitution.* □

In symbols, if $\theta \in \Omega$, then $\theta^{-1} \in \Omega$. This implies, in particular, that if θ is a bijective substitution, θ^{-1} is a variable renaming on any set of variables.

Bijective substitutions are a useful theoretical tool. For this reason, we prefer working with them instead of with simple variable renamings on a given set. Thus, given $\theta \in \Omega(V)$, we would like to replace it by a bijective substitution ξ with $\xi = \theta$ on V. Even if in general this is not possible, the following proposition holds.

Proposition 2.21 *If V is* finite *and θ is a variable renaming on V, then there exists a bijective substitution ξ such that $\xi = \theta$ on V.* □

This implies that *given a variable renaming on a* finite *set of variables, we can suppose that it is a bijective substitution.*

We define the subsumption relation between substitutions restricted to a particular set of variables.

Definition 2.22 *We write $\theta \leq \xi$ on V and say that θ* subsumes *ξ on V if and only if there exists a substitution ϕ such that $\theta\phi = \xi$ on V. We write $\theta \equiv \xi$ on V and say that θ is a* variant *of ξ on V if and only if there exists a variable renaming ϕ on $var(V\theta)$ such that $\theta\phi = \xi$ on V.*

Corollary 2.23 *The following consequences follow.*

i. If $\theta \leq \xi$ on V and $\xi \leq \theta$ on V, then $\theta \equiv \xi$ on V.

ii. If $\theta \leq \xi$ on V and ϕ is a variable renaming, then $\theta\phi \leq \xi$ on V.

Proof For i, the hypotheses imply the existence of φ_i, for $i = 1, 2$, such that $\theta\varphi_1 = \xi$ on V and $\xi\varphi_2 = \theta$ on V. But then $\theta\varphi_1\varphi_2 = \theta$ on V. By Corollary 2.18, φ_1 is a variable renaming on $var(V\theta)$. Fom this remark, the equality $\theta\varphi_1 = \xi$ on V and the preceding definition, we get $\theta \equiv \xi$ on V. Concerning *ii*, and by hypothesis, there exists a φ such that $\theta\varphi = \xi$ on V. If ϕ is a variable renaming on $var(V\theta)$, by Proposition 2.13, there exists a ϕ' such that $\phi\phi' = id$ on $var(V\theta)$. But then, $(\theta\phi)(\phi'\varphi) = \theta id\varphi = \theta\varphi = \xi$ on V. By definition, $\theta\phi \leq \xi$ on V. □

Definition 2.24 *We write $t \leq u$ and say that t* subsumes *u or that u is an* instance *of t iff for some substitution θ, $t\theta = u$. We say that t is a* variant *of u and write $t \equiv u$ iff there exists $\theta \in \Omega(var(t))$ such that $t\theta = u$.*

An immediate consequence of the definition is that $t \equiv u$ iff there exists $\theta \in \Omega$ such that $t\theta = u$. From this, proving that $\equiv$ is an equivalence relation on the set of terms is an easy exercise. We note that if $\theta \equiv \xi$ on V and t is a term such that $var(t) \subseteq V$, then $t\theta \equiv t\xi$.

2.3 Unifiers

An *equation* is a formal expression $t \asymp u$, where '$\asymp$' is a symbol not appearing in the underlying language. The natural interpretation of this symbol is obviously the equality relation, but we will not assign any particular meaning to it. It will only be a tool in the definition of a unification system. Note that this convention implies that the equations are not atoms and conversely. For the sake of notation, let E, F and their subscripted variants denote finite sets of equations. $\Gamma, \Delta, \Lambda, \Pi$ will denote finite sequences of atoms and/or equations. $|\Delta|$ denotes the number of atoms and/or equations in Δ. The meaning of expressions of the form '$\Delta\theta$' is obvious. In particular, if $|\Delta| = 0$, then $\Delta\theta = \Delta$.

Definition 2.25 *A* unifier *of a set E of equations is a substitution θ such that $t\theta = u\theta$ for all $t \asymp u \in E$; $unf(E)$ will denote the set of all unifiers of E. We say that E is* unifiable *iff $unf(E) \neq \emptyset$. If S is a set of substitutions, a* principal *member of S is a substitution $\theta \in S$ such that for all $\xi \in S$, $\theta \leq \xi$. A* principal *unifier*[1] *of E is a principal member of $unf(E)$. We denote by $pru(E)$ the set of all principal unifiers of E. When talking about unifiers and principal unifiers, a list of equations is identified with the set containing the same equations.*

When the set of equations is a singleton $\{ t \asymp u \}$, we prefer the lighter notations $unf(t,u)$, and $pru(t,u)$ to $unf(\{ t \asymp u \})$, and $pru(\{ t \asymp u \})$ respectively.

It is easy to see that if $\theta \in unf(E)$, then for any ξ, $\theta\xi \in unf(E)$. Also, if θ and ξ are principal members of S, then $\theta \equiv \xi$. In particular, two principal unifiers of a set of equations are variants of one another. By definition, the set of principal unifiers of a set of equations is completely determined by the set of all its unifiers. This implies the following easy

Proposition 2.26 *If $unf(E_0) = unf(E_1)$, then $pru(E_0) = pru(E_1)$.* □

The following is a kind of converse.

Proposition 2.27 *$pru(E_0) \cap pru(E_1) \neq \emptyset$ implies $unf(E_0) = unf(E_1)$.* □

Definition 2.28 *If $t = f(t_1, \ldots, t_n)$ and $u = f(u_1, \ldots, u_n)$, define $\propto(t, u)$ to be the list*

$$t_1 \asymp u_1 \ \ldots \ t_n \asymp u_n{}^2.$$

If t and u are incompatible, $\propto(t, u)$ is the empty list. For A and B atoms, $\propto(A, B)$ is defined analogously.

It is a simple exercise to verify the following

Corollary 2.29 *i. If t and u are compatible terms, then $unf(t,u) = unf(\propto(t, u))$. From this and a previous proposition, $pru(t,u) = pru(\propto(t, u))$.*

ii. For all t, u and θ, $\propto(t\theta, u\theta) = \propto(t, u)\theta$.

iii. If $t = f(t_1, \ldots, t_n)$ and $u = f(u_1, \ldots, u_n)$, then $var(\propto(t, u)) = var(t) \cup var(u)$, and

$$\#t + \#u = 2 + \sum_{i=1}^{n} \#t_i + \#u_i.$$

□

[1] Usually called a *most general* unifier.

[2] Note that $\propto$ is not the greek letter α.

If t and u share no variable and for some v, $t \leq v$ and $u \leq v$, then t and u are unifiable. Also, if $x \neq t$, then x and t are unifiable iff $x \notin var(t)$.

A unification algorithm is a computable function $unify(E)$ such that if E is unifiable, then $unify(E)$ is a principal unifier of E, and if E is not unifiable, then $unify(E)$ is some value encoding the fact that E is not unifiable. Various unification algorithms are known. Trivially, if E has a principal unifier, it also has a unifier. But the converse is also true. If $unify(E)$ is a unification algorithm and $unf(E) \neq \emptyset$, then $unify(E) \in pru(E)$, hence $pru(E) \neq \emptyset$. In short, E has a unifier iff it has a principal unifier. This is not evident from the definitions. Also, the unifiability of a finite set E of equations is decidable: any unification algorithm may be used for determining whether E is unifiable or not.

Below we present a unification proof system and proofs of its correctness and completeness. The system gives rise to a unification algorithm which is essentially the one presented in [AK91] as the general Warren Abstract Machine (WAM) [War83] unification algorithm. We also present a detailed description of SUE, an abstract unification machine that we propose as an alternative to the WAM unification fragment.

Even if unification algorithms and proofs appear elsewhere, the untrained reader may be unfamiliar with them. Also, typical presentations fall into one of the two 'extreme' classes: either an operational description of a practical algorithm but without justification, as in [AK91], or an abstract description and proof of an algorithm not matching the algorithm we want to use in practice, as in [Llo87]. Our presentation is mathematically precise and has the additional merit of fitting almost perfectly with the algorithm in [AK91], so it will help inexperienced readers to understand how the unifiers are stored and the unification operation implemented in the WAM and in SUE. This section is rather technical, so the reader may prefer to skip it on first reading. We note, however, that it is essential for the presentation of our SLD-Resolution formalism.

2.3.1 A Unification System

Let E be a set of equations. A *binding* of x in E is a term t such that $x \asymp t \in E$. If x has a binding in E, it is *bound* in E; otherwise, it is *unbound* in E.

Definition 2.30 *We write $x <_E z$ iff some binding of x in E contains z; $depth(x, E)$ is the least upper bound of the set of $k \in \mathcal{N}$ for which there exists a sequence*

$$x = x_0 <_E x_1 <_E \ldots <_E x_k.$$

Equivalently, it is the least upper bound of the set of $k \in \mathcal{N}$ for which there exists a variable z such that $x <_E^k z$. If this set has no finite upper bound, we write $depth(x, E) = +\infty$.

It is immediate that $depth(x, E) \neq 0$ if and only if there exists a variable z such that $x <_E z$, which is equivalent to saying that x is bound to an open term in E. In particular, a variable of positive $<_E$-depth is bound in E. It follows that $depth(x, E) = 0$ iff either x is unbound in E or all its bindings in E are closed terms. It is also easy to verify that if $depth(x, E)$ is finite

and $x <_E z$, then $depth(z, E) < depth(x, E)$. This implies that if t is a binding of x, then for all $z \in var(t)$, $depth(z, E) < depth(x, E)$. Note also that if $x <_E^+ x$, then $depth(x, E) = +\infty$, where, as usual, $<_E^+$ is the transitive closure of $<_E$.

The notion of *environment* will be fundamental in our presentation of the unification system.

Definition 2.31 *An* environment *is a finite set* $E = \{ x_i \asymp t_i \mid 1 \leq i \leq n \}$ *of equations such that for all variables* $x \in \mathcal{V}$, x *has at most one binding in* E, *and* $depth(x, E)$ *is finite.*

If E is an environment, and $F \subseteq E$, then F is an environment. As a consequence, $\emptyset$ is an environment.

Corollary 2.32 *Let* E *be an environment.*

i. For all x, $x \not<_E^+ x$. *This implies that no variable is bound to itself in* E.

ii. The set of variables of positive $<_E$*-depth is finite.*

iii. If the binding of x *in* E *is the open term* t, *then*

$$depth(x, E) = \max \{ depth(y, E) \mid y \in var(t) \} + 1.$$

In particular, this implies that for some $y \in var(t)$, $depth(x, E) = depth(y, E) + 1$.

□

If E is an environment, we define $depth(E) = \max \{ depth(x, E) \mid x \in \mathcal{V} \}$. The previous corollary implies that $depth(E)$ is finite. Now, if $E = \{ x_i \asymp t_i \mid 1 \leq i \leq n \}$, we write σ_E as an abridged notation for the substitution $[x_1/t_1, \ldots, x_n/t_n]$. Thus, if x is unbound in E, $x\sigma_E = x$, and if x is bound in E, $x\sigma_E$ is the unique binding of x in E. It follows that $ker(\sigma_E)$ is the set of variables bound in E. Note also that

$$\forall x \in ker(\sigma_E) \ \forall z \in \mathcal{V} \ \forall k \in \mathcal{N}, \qquad x <_E^k z \quad \text{iff} \quad z \in var(x\sigma_E^k). \tag{2.3}$$

Proposition 2.33 *Let* E *be an environment,* x *an arbitrary variable and* m, n, *two integers both strictly greater than* $depth(x, E)$. *Then* $x\sigma_E^m = x\sigma_E^n$.

Proof Suppose that $depth(x, E) = k$, and that we have

$$\text{for all } j \in \mathcal{N}, \ x\sigma_E^{(k+1)+j} = x\sigma_E^{k+1}. \tag{2.4}$$

In such a case, if m and n are two integers greater than k, we have $x\sigma_E^m = x\sigma_E^{k+1} = x\sigma_E^n$. It suffices then to prove that for all $x \in \mathcal{V}$, (2.4) holds. This is done by induction on $depth(x, E)$.

Base Case: $depth(x, E) = 0$. If x is unbound in E, $x\sigma_E = x$. Hence, we have $x\sigma_E^{1+j} = x = x\sigma_E^1$. If x is bound to the closed term t,

$$x\sigma_E^{1+j} = (x\sigma_E)\sigma_E^j = t\sigma_E^j = t = x\sigma_E^1.$$

Inductive Step: $depth(x, E) = k > 0$. Letting t be the binding of x in E, we have $x\sigma_E^{(k+1)+j} = (x\sigma_E)\sigma_E^{k+j} = t\sigma_E^{k+j}$. But for all $z \in var(t)$, $depth(z, E) < k$. We may then apply the inductive hypothesis (I.H.) to all variables in $var(t)$ and this implies $t\sigma_E^{k+j} = t\sigma_E^k$. We finally get $x\sigma_E^{(k+1)+j} = t\sigma_E^{k+j} = t\sigma_E^k = x\sigma_E^{k+1}$. □

Corollary 2.34 *If* $depth(E) = m$, *then for all* $j \in \mathcal{N}$, $\sigma_E^{(m+1)+j} = \sigma_E^{m+1}$. □

Definition 2.35 *Let* $depth(E) = m$. *We define* $\sigma_E^+ = \sigma_E^{m+1}$.

Corollary 2.36 *i. For all* $m, n \in \mathcal{N}$, $\sigma_E^m \sigma_E^+ \sigma_E^n = \sigma_E^+$. *In particular,* σ_E^+ *is idempotent.*

ii. For all terms t, *and all* $m, n \in \mathcal{N}$, σ_E^+ *unifies* $t\sigma_E^m \asymp t\sigma_E^n$. *In particular, since all the equations of* E *are of the form* $x \asymp x\sigma_E$, σ_E^+ *is a unifier of* E.

□

We see then that all environments are unifiable. According to our definition, for some $k \in \mathcal{N}$ we have $\sigma_E^+ = \sigma_E^k$, hence $\sigma_E^{k+1} = \sigma_E^k$. The following result shows that the converse is also true.

Proposition 2.37 *If* $\sigma_E^{k+1} = \sigma_E^k$ *on* V, *then* $\sigma_E^+ = \sigma_E^k$ *on* V.

Proof Taking $depth(E) = m$, we have $\sigma_E^+ = \sigma_E^k \sigma_E^{m+1}$. Suppose that for some $j \in \mathcal{N}$ we have already proved that $\sigma_E^+ = \sigma_E^k \sigma_E^{j+1}$ on V. Then by the hypothesis, we readily get $\sigma_E^+ = \sigma_E^{k+1}\sigma_E^j = \sigma_E^k \sigma_E^j$ on V. □

In particular, this implies that if $t\sigma_E^{k+1} = t\sigma_E^k$, then $t\sigma_E^+ = t\sigma_E^k$. Consequently, in order to compute $t\sigma_E^+$, we may successively compute the values $t\sigma_E^k$, for $k = 0, 1, \ldots$, up to a point where we see no change. But if $t\sigma_E^k$ is not modified by σ_E, then all variables of $t\sigma_E^k$ must be unbound in E. Recall that x is unbound in E if and only if $x \notin ker(\sigma_E)$.

Proposition 2.38 *If* E *is an environment and* t *an arbitrary term, all variables in* $var(t\sigma_E^+)$ *are unbound in* E.

Proof We know that $t\sigma_E^+\sigma_E = t\sigma_E^+$. It suffices then to apply Corollary 2.9. □

Proposition 2.39 $ker(\sigma_E^+) = ker(\sigma_E)$.

Proof If $x \notin ker(\sigma_E^+)$, then $x = x\sigma_E^+$, hence, by the preceding proposition, x is unbound in E, that is, $x \notin ker(\sigma_E)$. Conversely, and trivially, if x is unbound in E, $x = x\sigma_E^+$. □

From this result, Proposition 2.10, and the fact that σ_E^+ is idempotent, we get

Corollary 2.40 $ker(\sigma_E) \cap var(ker(\sigma_E)\sigma_E^+) = \emptyset$, *for all environments* E. □

Examples

i. $\sigma_\emptyset = id$ and for all $x \in \mathcal{V}$, $depth(x, \emptyset) = 0$. This implies $depth(\emptyset) = 0$ and $\sigma_\emptyset^+ = id$.

ii. If $E = \{ x \asymp a \}$, $\sigma_E = [x/a]$, $depth(E) = 0$ and $\sigma_E^+ = [x/a]$.

iii. Let $E = \{ x_0 \asymp f(x_1),\ x_1 \asymp x_2,\ x_3 \asymp x_1 \}$. We have then $\sigma_E = [x_0/f(x_1), x_1/x_2, x_3/x_1]$ and $depth(E) = 2$. Thus, $x_0\sigma_E^+ = f(x_2)$

□

Proposition 2.41 *Let E be an environment. Then $\theta \in unf(E)$ if and only if $\theta = \sigma_E^+\theta$.*

Proof Take $\theta \in unf(E)$. We will prove that for all $x \in \mathcal{V}$, $x\sigma_E^+\theta = x\theta$ by induction on $depth(x, E)$. In the proof we will use the fact that, θ, by hypothesis, and σ_E^+, by a previous corollary, are unifiers of E.

Base Case: $depth(x, E) = 0$. If x is unbound in E, we trivially get $x\sigma_E^+\theta = x\theta$. If x is bound to the closed term t, we have

$$x\sigma_E^+\theta = t\sigma_E^+\theta = t\theta = x\theta. \tag{2.5}$$

Inductive Step: $depth(x, E) > 0$. If t is the binding of x in E, applying the I.H. to the variables of t we get $t\sigma_E^+\theta = t\theta$. But then, the equalities in (2.5) also hold in this case. □

Corollary 2.42 *For all environments E, σ_E^+ is an idempotent principal unifier of E.* □

The following corollary gives us the central idea of the unification system presented below.

Corollary 2.43 *If E_0 is an arbitrary set of equations and E_1 an environment such that $unf(E_1) = unf(E_0)$, then $\sigma_{E_1}^+$ is an idempotent principal unifier of E_0.*

Proof The hypotheses imply that $pru(E_1) = pru(E_0)$. From the last corollary, we conclude that $\sigma_{E_1}^+ \in pru(E_0)$. □

Hence, for computing an idempotent principal unifier of a finite set E_0 of equations it suffices to transform E_0 into an environment E_1 with the same unifiers as E_0. Note, however, that if E_0 is not unifiable, no such environment exists.

In the Logic Programming literature we frequently find the notion of 'dereferencing' a term with respect to the 'current' environment. This operation is made precise by the function δ defined below.

Definition 2.44 *Let E be an environment, and t an arbitrary term. We define*

$$\delta(t, E) = \begin{cases} t\sigma_E^+ & \text{if for all } k \in \mathcal{N},\ t\sigma_E^k \in \mathcal{V}. \\ t\sigma_E^{k_0} & \text{otherwise.} \end{cases}$$

where k_0 is the least $k \in \mathcal{N}$ such that $t\sigma_E^k$ is not *a variable.*

We say that $\delta(t, E)$ is the *dereferenced value of t in E*. Note that $\delta(t, E)$ is always of the form $t\sigma_E^j$, hence $\delta(t, E)\sigma_E^+ = t\sigma_E^+$.

Corollary 2.45 *i. The following is the usual operational definition of the dereferencing operation. If t is a constant or a complex term, or a variable unbound in E, then $\delta(t,E) = t$. If t is a variable bound in E, then $\delta(t,E) = \delta(t\sigma_E, E)$.*

ii. If $\#t\sigma_E^+ = 0$, then $\delta(t,E) = t\sigma_E^+$. If $\#\delta(t,E) = 0$, then $\delta(t,E) = t\sigma_E^+$.

iii. If $\#t\sigma_E^+ > 0$, then $\delta(t,E) \approx t\sigma_E^+$. If $\#\delta(t,E) > 0$, then $\delta(t,E) \approx t\sigma_E^+$.

Proof For i, if t is a constant or complex term, since $t\sigma_E^0 = t$, 0 is the least integer k such that $t\sigma_E^k$ is not a variable. From the definition, $\delta(t,E) = t\sigma_E^0 = t$. If t is a variable unbound in E, for all $k \in \mathcal{N}$, $t\sigma_E^k = t$. Thus, $t\sigma_E^k$ is a variable for all $k \in \mathcal{N}$. Now we apply the definition of δ. If t is a variable bound in E, we apply induction on the depth of t in E.

For *ii*, suppose $\#\delta(t,E) = 0$. If $\delta(t,E)$ is a variable, we must be in the first case of the definition of δ, hence $\delta(t,E) = t\sigma_E^+$, and if $\delta(t,E)$ is a constant, we have $t\sigma_E^+ = \delta(t,E)\sigma_E^+ = \delta(t,E)$. Conversely, if $\#t\sigma_E^+ = 0$, since $\delta(t,E) \leq t\sigma_E^+$, $\#\delta(t,E) = 0$, and the preceding argument applies.

Concerning *iii*, recall the equality $\delta(t,E)\sigma_E^+ = t\sigma_E^+$, and note that by *ii*, $\#t\sigma_E^+ > 0$ if and only if $\#\delta(t,E) > 0$. □

Exemple Let $E = \{\ x_0 \asymp f(x_1),\ x_1 \asymp x_2,\ x_3 \asymp x_1\ \}$. We have then: $\delta(x_0,E) = f(x_1)$, $\delta(x_3,E) = x_2$. □

For the sake of conciseness, we introduce the following predicates.

$\Phi_1(t,u,E)$: $\#t\sigma_E^+ = 0$, and $u\sigma_E^+ = t\sigma_E^+$.

$\Phi_2(t,u,E)$: $t\sigma_E^+ \in \mathcal{V} \setminus var(u\sigma_E^+)$.

$\Phi_3(t,u,E)$: $t\sigma_E^+ \notin \mathcal{V}$ and $\Phi_2(u,t,E)$.

$\Phi_4(t,u,E)$: $t\sigma_E^+ \approx u\sigma_E^+$.

$\Phi_5(t,u,E)$: $\bigwedge_{i=1}^4 \neg\Phi_i(t,u,E)$.

The following are immediate consequences of these definitions.

Corollary 2.46 *i. For all $1 \leq i < j \leq 4$, if $\Phi_i(t,u,E)$, then not $\Phi_j(t,u,E)$.*

ii. $\Phi_1(t,u,E)$ iff $\delta(t,E)$ is either a variable or a constant and $\delta(u,E) = \delta(t,E)$.

iii. $\Phi_2(t,u,E)$ iff $\delta(t,E)$ is a variable, and $\delta(t,E) \notin var(u\sigma_E^+)$.

iv. $\Phi_4(t,u,E)$ iff $\delta(t,E)$ and $\delta(u,E)$ are compatible terms.

□

Proposition 2.47 *If $t\sigma_E^+$ and $u\sigma_E^+$ are unifiable, then for some $i = 1,\ldots,4$ $\Phi_i(t,u,E)$. Equivalently, if $\Phi_5(t,u,E)$, then $t\sigma_E^+$ and $u\sigma_E^+$ are not unifiable.*

Proof We simply consider the different possibilities for $t\sigma_E^+$ and $u\sigma_E^+$. If $t\sigma_E^+$ is a variable, since $u\sigma_E^+$ is unifiable with it, we must have either that $u\sigma_E^+ = t\sigma_E^+$, hence $\Phi_1(t, u, E)$, or that $u\sigma_E^+$ is a term not containing $t\sigma_E^+$, hence $\Phi_2(t, u, E)$. The rest is analogous. □

Proposition 2.48 *If ϕ is a variable renaming on $var(t\sigma_E^+) \cup var(u\sigma_E^+)$, $v\sigma_F^+ = t\sigma_E^+\phi$, and $w\sigma_F^+ = u\sigma_E^+\phi$, then for all $i = 1, \ldots, 5$, $\Phi_i(t, u, E)$ if and only if $\Phi_i(v, w, F)$.*

Proof First, let us note that it is enough to prove the following
Claim : For all t, u, v, w, E, F and $\phi \in \Omega$ satisfying the hypothesis of this proposition, and for all $i = 1, \ldots, 4$,

$$\Phi_i(t, u, E) \quad \textit{implies} \quad \Phi_i(v, w, F). \tag{2.6}$$

Assuming the claim and the hypothesis of the proposition, if $\phi \in \Omega$, we get $t\sigma_E^+ = v\sigma_F^+\phi^{-1}$, and $u\sigma_E^+ = w\sigma_F^+\phi^{-1}$, hence, by the claim, and for $i = 1, \ldots, 4$, $\Phi_i(v, w, F)$ implies $\Phi_i(t, u, E)$. Therefore, if the claim holds as is, it also holds replacing 'implies' by 'iff' in (2.6). It is also easy to verify that if the claim holds in this new form, it remains valid replacing '$i = 1, \ldots, 4$' by '$i = 1, \ldots, 5$'.

If $\Phi_1(t, u, E)$, then $\#t\sigma_E^+ = 0$, and $u\sigma_E^+ = t\sigma_E^+$. Hence, since ϕ is a variable renaming, $\#v\sigma_F^+ = \#t\sigma_E^+ = 0$, and $w\sigma_F^+ = u\sigma_E^+\phi = t\sigma_E^+\phi = v\sigma_F^+$. This shows that $\Phi_1(v, w, F)$ is true.

If $\Phi_2(t, u, E)$, then $t\sigma_E^+ \in \mathcal{V} \setminus var(u\sigma_E^+)$, and since ϕ is a variable renaming,

$$v\sigma_F^+ = t\sigma_E^+\phi \in \mathcal{V} \setminus var(u\sigma_E^+\phi) = \mathcal{V} \setminus var(w\sigma_F^+),$$

hence $\Phi_2(v, w, F)$ holds. The rest is analogous. □

Corollary 2.49 *For all $j, k \in \mathcal{N}$, $\Phi_i(t, u, E)$ if and only if $\Phi_i(t\sigma_E^j, u\sigma_E^k, E)$.* □

We will call *equational goal* any expression of the form $\Delta \vdash E$, where Δ is a finite sequence of equations, and E a list of equations such that the corresponding set is an environment. By definition, a goal $\Lambda \vdash F$ is equal to $\Delta \vdash E$ iff $\Lambda = \Delta$, and E and F contain the same equations, independently of their order. The *empty* goals are those of the form $\vdash E$. A *permutation* of $\Delta \vdash E$ is any goal of the form $\Lambda \vdash E$, where Λ is a permutation of Δ. A *unifier* of a goal is a unifier of *all* the equations appearing in the goal. Note the following characterization.

Proposition 2.50 *Let $\Delta \vdash E$ be an equational goal. Then θ is a unifier of $\Delta \vdash E$ iff θ is a unifier of Δ, and $\theta = \sigma_E^+\theta$.*

Proof Easy. Simply recall that by Proposition 2.41, θ is a unifier of E iff $\theta = \sigma_E^+\theta$. □

We will call U the system consisting of the unification rules of Table 2.1. Note that by Corollary 2.46, at most one rule of U can be applied to any given goal. Each pair of goals

$$\frac{\Delta_0 \vdash E_0}{\Delta_1 \vdash E_1} \tag{2.7}$$

The Equation Elimination Rule (ee)

$$\frac{\Delta \quad t \asymp u \quad \vdash \quad E}{\Delta \quad \vdash \quad E} \quad \textit{with } \Phi_1(t,u,E).$$

The Left Binding Rule (lb)

$$\frac{\Delta \quad t \asymp u \quad \vdash \quad E}{\Delta \quad \vdash \quad t\sigma_E^+ \asymp u\sigma_E^k \quad E} \quad \textit{with } \Phi_2(t,u,E), \quad k \in \mathcal{N}.$$

The Right Binding Rule (rb)

$$\frac{\Delta \quad t \asymp u \quad \vdash \quad E}{\Delta \quad \vdash \quad u\sigma_E^+ \asymp t\sigma_E^k \quad E} \quad \textit{with } \Phi_3(t,u,E), \quad k \in \mathcal{N}.$$

The Structure Decomposition Rule (sd)

$$\frac{\Delta \quad t \asymp u \quad \vdash \quad E}{\Delta \quad \propto(\,\delta(t,E)\sigma_E^j, \delta(u,E)\sigma_E^k\,) \quad \vdash \quad E} \quad \textit{with } \Phi_4(t,u,E), \quad j,k \in \mathcal{N}.$$

Table 2.1: The Unification Rules

satisfying the conditions defining rule r is said to be an *instance* of r. $\Delta_0 \vdash E_0$ is the *antecedent* of this instance, while $\Delta_1 \vdash E_1$ is the *conclusion.* A rule is simply the set of all its instances. The *complexity* of a proof in U (or any other system) is the number of rule applications it contains. Note that the rules other than **ee** have integer parameters, and that, as a consequence, these rules may contain various instances with the same antecedent. For example, here we have two instances of **lb**.

$$\frac{x \asymp f(y) \quad \vdash \quad y \asymp a}{\vdash \quad x \asymp f(y) \;\; y \asymp a} \qquad \frac{x \asymp f(y) \quad \vdash \quad y \asymp a}{\vdash \quad x \asymp f(a) \;\; y \asymp a}$$

However, each rule of U contains only a finite number of instances with the same antecedent. See Figure 2.1 for a more complex example. Note that $[x/g(a), y/a]$ is a unifier of the first goal of this exemple, and that if θ is another such unifier, then $\theta = [x/g(a), y/a]\theta$.

We must ensure that our rules are well defined, and for this it will be enough to show that if (2.7) is a candidate rule instance, then E_1 is an environment. This is obvious for all the instances of **ee**, and **sd**. For those of **lb**, and **rb**, we need some further results.

Proposition 2.51 *If for some* $k \in \mathcal{N}$, $x \in var(t\sigma_E^k) \setminus ker(\sigma_E^+)$, *then* $x \in var(t\sigma_E^+)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{f(b,g(y),y) \asymp f(b,x,a) \quad \vdash}{b \asymp b \;\; g(y) \asymp x \;\; y \asymp a \quad \vdash}}{b \asymp b \;\; g(y) \asymp x \quad \vdash \quad y \asymp a}}{b \asymp b \quad \vdash \quad x \asymp g(a) \;\; y \asymp a}}{\vdash \quad x \asymp g(a) \;\; y \asymp a}$$

Figure 2.1: An example proof in U.

Proof From Corollary 2.36, $t\sigma_E^k\sigma_E^+ = t\sigma_E^+$. Therefore,

$$var(t\sigma_E^+) = \bigcup_{z \in var(t\sigma_E^k)} var(z\sigma_E^+).$$

By hypothesis, $x \in var(t\sigma_E^k)$. From this and the previous equality we get that $var(x\sigma_E^+)$ is contained in $var(t\sigma_E^+)$. But, by hypothesis, x is unbound in E, which implies $x\sigma_E^+ = x$. From this and the previous remark we conclude $x \in var(t\sigma_E^+)$. □

Proposition 2.52 *Let E be an environment, x a variable unbound in E, and u a term such that $x \notin var(u\sigma_E^+)$. Take an arbitrary $k \in \mathcal{N}$ and define*

$$F = E \cup \{\, x \asymp u\sigma_E^k \,\}.$$

Then F is an environment and $\sigma_F^+ = \sigma_E^+[x/u\sigma_E^+]$. In particular, the statement holds if we take a $k \in \mathcal{N}$ making $u\sigma_E^k = \delta(u, E)$.

Proof Clearly, F is finite and associates at most one binding to each variable. Therefore, proving that F is an environment reduces to proving that all variables are of finite $<_F$-depth. Suppose we have an infinite sequence

$$x_0 <_F x_1 <_F \ldots <_F x_n <_F \ldots \tag{2.8}$$

It is obvious that if $y <_F z$ and $y \neq x$, then $y <_E z$; so, if for all n, $x_n \neq x$, we would have $depth(x_0, E) = +\infty$, which is impossible. Then, x appears somewhere in the sequence. Without losing generality, we may suppose that $x_0 = x$. The same argument leads us to conclude that there exists a $n \in \mathcal{N}$ such that $x_{n+1} = x$. Taking the least such n, we have the following situation:

$$x <_F x_1 \quad \text{and} \quad x_1 <_E x_2 <_E \ldots <_E x_{n+1} = x. \tag{2.9}$$

From the first condition in (2.9), x_1 appears in the binding of x in F, which is $u\sigma_E^k$. We have then $x_1 \in var(u\sigma_E^k)$. Besides, the second condition in (2.9), and (2.3) imply that $x \in var(x_1\sigma_E^n)$. Consequently, we have $x \in var(u\sigma_E^{k+n})$.

By hypothesis, x is unbound in E, so applying Proposition 2.51, we get $x \in var(u\sigma_E^+)$, which contradicts the hypotheses. We conclude then that there is no infinite sequence like (2.8) and so, all variables are of finite $<_F$-depth. This proves that F is an environment. We now prove that for all variables z, $z\sigma_F^+ = z\sigma_E^+[x/u\sigma_E^+]$.

Base Case: $depth(z, F) = 0$. Suppose first that z is unbound in F. We have then $z\sigma_F^+ = z$. But under this hypothesis, z is also unbound in E and is different from x. This implies $z\sigma_E^+[x/u\sigma_E^+] = z$.

If the binding of z in F is the closed term t, $z\sigma_F^+ = t\sigma_F^+ = t$. If $z \neq x$, then $z \asymp t \in E$ and we get

$$z\sigma_E^+[x/u\sigma_E^+] = t\sigma_E^+[x/u\sigma_E^+] = t.$$

If $z = x$, then $t = u\sigma_E^k$. Since t is supposed to be a closed term, we have

$$u\sigma_E^+ = u\sigma_E^k\sigma_E^+ = t\sigma_E^+ = t.$$

Now, recalling that by hypothesis $x = z$ is unbound in E, we conclude

$$z\sigma_E^+[x/u\sigma_E^+] = x[x/u\sigma_E^+] = u\sigma_E^+ = t.$$

Inductive Step: $depth(z, F) > 0$. We first consider the subcase $z \neq x$. By the case hypothesis, z is bound to an open term t and, from $z \neq x$, we must have $z \asymp t \in E$. This means that both σ_E^+ and σ_F^+ unify $z \asymp t$. Since we may apply the I.H. to the variables of t, we have

$$z\sigma_F^+ = t\sigma_F^+ = t\sigma_E^+[x/u\sigma_E^+] = z\sigma_E^+[x/u\sigma_E^+].$$

If, on the other side, $z = x$, the binding of $z = x$ in F is $u\sigma_E^k$ and we may apply the I.H. on the variables of this term. This gives

$$u\sigma_E^k\sigma_F^+ = u\sigma_E^k\sigma_E^+[x/u\sigma_E^+].$$

But $u\sigma_E^k\sigma_E^+ = u\sigma_E^+$ and from the hypothesis that $x \notin var(u\sigma_E^+)$ we get

$$u\sigma_E^k\sigma_E^+[x/u\sigma_E^+] = u\sigma_E^+.$$

The previous equalities imply $x\sigma_F^+ = u\sigma_E^k\sigma_F^+ = u\sigma_E^+$. On the other hand, since $z = x$ is unbound in E we have $x\sigma_E^+[x/u\sigma_E^+] = u\sigma_E^+$ and the proposition is proved. □

Consequently, if E is an environment, $k \in \mathcal{N}$, and t, u are terms such that $\Phi_2(t, u, E)$, making $x = t\sigma_E^+$ in the above proposition, we get that $E \cup \{ t\sigma_E^+ \asymp u\sigma_E^k \}$ is an environment. For the same reason, if $\Phi_3(t, u, E)$, then $E \cup \{ u\sigma_E^+ \asymp t\sigma_E^k \}$ is an environment. This shows that **lb** and **rb** are well defined.

It is clear that an equational goal $\Delta \vdash E$ encodes both a 'partial unifier' and a list of equations waiting to be unified. At first sight, we may think that these are, respectively, σ_E and Δ, but a careful analysis of the conditions $\Phi_i(t, u, E)$ shows that the partial unifier is σ_E^+ and that the list of 'suspended' equations is $\Delta\sigma_E^+$.

If $\Phi_1(t,u,E)$ holds, $t\sigma_E^+ = u\sigma_E^+$. If, during the unification process, we pick $t \asymp u$ up and discover that $\Phi_1(t,u,E)$ holds, we are implicitly selecting the suspended equation $t\sigma_E^+ \asymp u\sigma_E^+$ and discovering that the two sides of the equation contain either the same variable or the same constant. Such an equation is already unified so we may simply eliminate it. This is what **ee** diligently does.

If $\Phi_2(t,u,E)$ holds, $t\sigma_E^+$ is a variable not appearing in $u\sigma_E^+$. Under these conditions, $[t\sigma_E^+/u\sigma_E^+]$ is a principal unifier of the selected equation $t\sigma_E^+ \asymp u\sigma_E^+$. It is quite natural to guess that the new partial unifier is the result of composing $[t\sigma_E^+/u\sigma_E^+]$ *after* the current partial unifier, that is, that the new partial unifier is $\sigma_E^+[t\sigma_E^+/u\sigma_E^+]$. Proposition 2.52 tells us that adding the equation $t\sigma_E^+ \asymp u\sigma_E^k$ to the current environment is a (highly efficient!) way of storing this new partial unifier.

If $\Phi_4(t,u,E)$ holds, $\delta(t,E)$ and $\delta(u,E)$ are compatible terms,

$$\delta(t,E) = f(t_1,\ldots,t_n) \quad \text{and} \quad \delta(u,E) = f(u_1,\ldots,u_n),$$

say. In this case,

$$\begin{aligned} t\sigma_E^+ &= \delta(t,E)\sigma_E^j\sigma_E^+ = f(t_1\sigma_E^j,\ldots,t_n\sigma_E^j)\sigma_E^+, \\ u\sigma_E^+ &= \delta(u,E)\sigma_E^k\sigma_E^+ = f(u_1\sigma_E^k,\ldots,u_n\sigma_E^k)\sigma_E^+. \end{aligned}$$

sd replaces $t \asymp u$ by $\{ t_i\sigma_E^j \asymp u_i\sigma_E^k \}$ without modifying the current partial unifier. This amounts to replacing $t\sigma_E^+ \asymp u\sigma_E^+$ by $\propto(t\sigma_E^+, u\sigma_E^+)$ in the list of suspended equations. This shows that U takes a goal $\Delta \vdash E$ as an encoded representation of the list $\Delta\sigma_E^+$ of equations. The key point, however, is that U tries not only to determine whether this list is unifiable or not, but also, it tries to compute a unifier. The purpose of the environment in the goal is precisely to 'store' the computed substitution.

When applied to a goal $\Delta \vdash E$, all the rules of U 'reduce' the rightmost equation of Δ. However, it is more or less clear that nothing essential is modified if we choose an equation other than the rightmost. In order to formalise this idea, instead of modifying the rules of U, we will introduce the following notion.

Definition 2.53 *An* exchange rule *is a unary rule X on goals such that*

i. If

$$\frac{\Delta_0 \vdash E_0}{\Delta_1 \vdash E_1}$$

is an instance of X, then $\Delta_1 \vdash E_1$ is a permutation of $\Delta_0 \vdash E_0$.

ii. For all goals $\Delta_0 \vdash E$, X contains at least one instance

$$\frac{\Delta_0 \vdash E}{\Delta_1 \vdash E} \tag{2.10}$$

According to *i*, an exchange rule X only allows us to permute goals, while *ii* ensures that X allows us to permute all goals. But this does not mean that X allows us to get all permutations of all goals in *one* rule application. This is why we talk about exchange *rules*. The convenience of adopting this definition will be clarified later. The *full* exchange rule is the exchange rule containing all the instances of the form (2.10) for all goals $\Delta_0 \vdash E$, and all permutations $\Delta_1 \vdash E$ of the former. From now on, except when explicitly stated otherwise, 'X' will denote an arbitrary exchange rule.

Suppose $\mathcal{S}$ is a deduction system acting on goals. We will denote by $\mathcal{S}^X$ the rule (and the single-rule system) obtained 'merging' all the instances of X with all the instances of the rules of $\mathcal{S}$, when this is possible. More precisely,

$$\frac{\Delta_0 \vdash E_0}{\Delta_1 \vdash E_1} \tag{2.11}$$

is an instance of $\mathcal{S}^X$ iff there exists an instance

$$\frac{\Delta_0 \vdash E_0}{\Lambda \vdash E_0}$$

of X such that

$$\frac{\Lambda \vdash E_0}{\Delta_1 \vdash E_1}$$

is an instance of a rule of $\mathcal{S}$. $\Delta_0 \vdash E_0$ is *reducible* in $\mathcal{S}$ iff there exists an instance of a rule of $\mathcal{S}$ of the form (2.11); otherwise, it is irreducible. Given a proof

$$\begin{array}{c} \Delta \vdash E \\ \vdots \\ \hline \Lambda \vdash F \end{array} \tag{2.12}$$

in $\mathcal{S}$, $\Lambda \vdash F$ is its *conclusion*, while the *interior* goal occurrences are the goal occurrences other than the conclusion. (2.12) is *maximal* iff $\Lambda \vdash E$ is irreducible; it is a *refutation* iff $\Lambda \vdash E$ is an empty goal. Trivially, all refutations in U^X are maximal, but the converse fails. For instance, $a \asymp b \vdash$ is maximal.

Our next task will be to show that U^X is sound. More precisely, we want to show that if (2.12) is a proof in U^X, then $\Lambda \vdash F$ has the same unifiers than $\Delta \vdash E$. Clearly, it suffices to show the

Proposition 2.54 *All the rules of U preserve unifiers.*

Proof

Case **ee** $\dfrac{\Delta \quad t \asymp u \quad \vdash \quad E}{\Delta \vdash E}$ *with* $\Phi_1(t,u,E)$.

Suppose θ is a unifier of $\Delta \vdash E$. Proposition 2.50 implies $\theta = \sigma_E^+\theta$, and from $\Phi_1(t,u,E)$, we get $t\theta = t\sigma_E^+\theta = u\sigma_E^+\theta = u\theta$. The rest is immediate.

Case **lb**

$$\frac{\Delta \quad t \asymp u \quad \vdash \quad E}{\Delta \quad \vdash \quad t\sigma_E^+ \asymp u\sigma_E^k \quad E} \qquad \textit{with } \Phi_2(t,u,E), k \in \mathcal{N}.$$

Let us write $F = E \cup \{ t\sigma_E^+ \asymp u\sigma_E^k \}$. The reader may convince himself that in order to prove that the two goals in the above rule instance have the same unifiers, it suffices to prove that for all θ,

- $\theta = \sigma_E^+\theta$ and $t\theta = u\theta$ implies $t\sigma_E^+\theta = u\sigma_E^k\theta$, and
- $\theta = \sigma_F^+\theta$ implies $t\theta = u\theta$.

For the first claim, if the hypothesis is true, we immediately get

$$t\sigma_E^+\theta = t\theta = u\theta = u\sigma_E^+\theta = u\sigma_E^k\sigma_E^+\theta = u\sigma_E^k\theta.$$

Suppose now $\theta = \sigma_F^+\theta$. By hypothesis, $\Phi_2(t,u,E)$, hence

$$t\sigma_E^+ \text{ is a variable not in } u\sigma_E^+. \tag{2.13}$$

Making $x = t\sigma_E^+$ in Proposition 2.52 we get

$$\theta = \sigma_F^+\theta = \sigma_E^+[t\sigma_E^+/u\sigma_E^+]\theta.$$

Thus, on the one hand,

$$t\theta = t\sigma_E^+[t\sigma_E^+/u\sigma_E^+]\theta = u\sigma_E^+\theta,$$

and on the other hand, recalling (2.13),

$$u\theta = u\sigma_E^+[t\sigma_E^+/u\sigma_E^+]\theta = u\sigma_E^+\theta.$$

This proves the second claim above. Hence, the case **lb** is proved. The proof for **rb** is analogous.

Case **sd**

$$\frac{\Delta \quad t \asymp u \quad \vdash \quad E}{\Delta \quad \propto(\,\delta(t,E)\sigma_E^j, \delta(u,E)\sigma_E^k\,) \quad \vdash \quad E} \qquad \textit{with } \Phi_4(t,u,E), j,k \in \mathcal{N}.$$

By Corollary 2.29, $\propto(\,\delta(t,E)\sigma_E^j, \delta(u,E)\sigma_E^k\,)$ has the same unifiers as $\delta(t,E)\sigma_E^j \asymp \delta(u,E)\sigma_E^k$. Therefore, it suffices to prove that for all θ

- if $\theta = \sigma_E^+\theta$, then $t\theta = u\theta$ if and only if $\delta(t,E)\sigma_E^j\theta = \delta(u,E)\sigma_E^k\theta$.

But this is easy, because if $\theta = \sigma_E^+\theta$, we get

$$t\theta = t\sigma_E^+\theta = \delta(t,E)\sigma_E^+\theta = \delta(t,E)\sigma_E^j\sigma_E^+\theta = \delta(t,E)\sigma_E^j\theta,$$

and similarly, $u\theta = \delta(u,E)\sigma_E^k\theta$. □

In particular, this implies that if

$$\frac{\begin{array}{c}\propto(A,B) \vdash E \\ \vdots \\ \vdash F\end{array}}{}$$

is a refutation in U^X, then σ_F^+ is a principal unifier of $\propto(A,B) \cup E$.

The completeness of U^X may be stated as follows. If $\Delta \vdash E$ is unifiable, then there exists a refutation

$$\frac{\begin{array}{c}\Delta \vdash E \\ \vdots \\ \vdash F\end{array}}{}$$

in U^X. We will see not only that this is true, but also that if $\Delta \vdash E$ is unifiable, then *all* maximal proofs in U^X extending $\Delta \vdash E$ are refutations.

Proposition 2.55 *All non-empty and unifiable goals are reducible in U^X. Equivalently, only the empty goals are both unifiable and irreducible in U^X.*

Proof Suppose that $\Delta \vdash E$ is non-empty and unifiable, choose an arbitrary instance

$$\frac{\Delta \vdash E}{\Lambda \; t \asymp u \vdash E}$$

of X, and let θ be a unifier of $\Lambda \; t \asymp u \vdash E$. By Proposition 2.50, $\theta = \sigma_E^+\theta$, hence

$$t\sigma_E^+\theta = t\theta = u\theta = u\sigma_E^+\theta,$$

which shows that $t\sigma_E^+$ and $u\sigma_E^+$ are unifiable. By Proposition 2.47, $\Lambda \; t \asymp u \vdash E$ is reducible in U. □

Corollary 2.56 *Let*

$$\frac{\begin{array}{c}\Delta \vdash E \\ \vdots \\ \Lambda \vdash F\end{array}}{} \tag{2.14}$$

be a maximal proof in U^X. Then (2.14) is a refutation iff $\Delta \vdash E$ is unifiable.

Proof If (2.14) is a refutation, by the soundness of U^X, σ_F^+ is a principal unifier of $\Delta \vdash E$. Conversely, if $\Delta \vdash E$ is unifiable, by the soundness of U^X and the hypothesis of maximality, $\Lambda \vdash F$ is both unifiable and irreducible. By the preceding proposition, $\Lambda \vdash F$ is empty. □

This corollary suggests the following mechanism for defining a unification algorithm. Given $\Delta \vdash E$, construct a maximal proof (2.14). Once this has been done, if $|\Lambda| = 0$, σ_F^+ is an idempotent principal unifier of $\Delta \vdash E$, and if $|\Lambda| > 0$, $\Delta \vdash E$ is not unifiable. The natural procedure for implementing this idea is as follows. We start with $\Delta \vdash E$. After having constructed *all* the proofs of complexity m, and assuming that there are only a finite

number of them, if one of them is maximal, we stop, otherwise, we construct all the proofs of complexity $m+1$, (there are only a finite number of them), and continue the process. If there exists a maximal proof, we will certainly be able to find it, but at this point we are not guaranteed that this is is the case for all the equational goals. This is essentially the problem of termination of the preceding algorithm.

Consider the set of all triples of non-negative integers (n, m, r) with the lexicographic ordering, that is: $(n_1, m_1, r_1) \leq (n_0, m_0, r_0)$ iff either $n_1 \leq n_0$, or $n_1 = n_0$ and $m_1 \leq m_0$, or $n_1 = n_0$ and $m_1 = m_0$ and $r_1 \leq r_0$. We mention without proof that there exists no infinite sequence of triples strictly decreasing in this partial order.

Definition 2.57 *The* level *of the equational goal $\Delta \vdash E$ is the triple (n, m, r) of natural numbers, where*

- *n is the cardinality of $var(\Delta\sigma_E^+)$, that is, the number of variables in $\Delta\sigma_E^+$, not counting repetitions. Note that according to Proposition 2.38, all these variables are unbound in E.*
- *$m = \sum \#v\sigma_E^+ + \#w\sigma_E^+$, where the sum is on all the equation occurrences $v \asymp w$ of Δ.*
- *$r = |\Delta|$.*

Trivially, the level of a goal does not depend on the order of the equations. Thus, two permutations are of the same level.

Proposition 2.58 *Let*

$$\frac{\Delta_0 \vdash E_0}{\Delta_1 \vdash E_1} \tag{2.15}$$

be an instance of the rule r of U, and let (n_i, m_i, r_i) be the corresponding levels, $i = 0, 1$.

i. If r is **ee**, *then $n_1 \leq n_0$, $m_1 = m_0$, and $r_1 < r_0$.*

ii. If r is **lb**, *or* **rb**, *then $n_1 < n_0$.*

iii. If r is **sd**, *then $n_1 = n_0$, and $m_1 < m_0$.*

Proof

Case **ee**

$$\frac{\Delta \quad t \asymp u \quad \vdash \quad E}{\Delta \vdash E} \quad \text{with } \Phi_1(t,u,E).$$

Since all the variables appearing in $\Delta\sigma_E^+$ also appear in $\Delta\sigma_E^+ \; t\sigma_E^+ \asymp u\sigma_E^+$, we have $n_1 \leq n_0$. Trivially, $m_0 = m_1 + \#t\sigma_E^+ + \#u\sigma_E^+$, and by $\Phi_1(t, u, E)$, $\#t\sigma_E^+ + \#u\sigma_E^+ = 0$. Therefore, $m_1 = m_0$. $r_1 < r_0$ is self-evident.

Case **lb**

$$\frac{\Delta \quad t \asymp u \quad \vdash \quad E}{\Delta \quad \vdash \quad t\sigma_E^+ \asymp u\sigma_E^k \quad E} \quad \text{with } \Phi_2(t,u,E), k \in \mathcal{N}.$$

Let us write $E_1 = E \cup \{ t\sigma_E^+ \asymp u\sigma_E^k \}$. We first show that

$$var(\Delta\sigma_{E_1}^+) \subseteq var(\Delta\sigma_E^+) \cup var(u\sigma_E^+). \tag{2.16}$$

By Proposition 2.52, $\sigma_{E_1}^+ = \sigma_E^+[t\sigma_E^+/u\sigma_E^+]$. Take now an equation $v \asymp w \in \Delta$. We have

$$var(v\sigma_{E_1}^+) = var(v\sigma_E^+[t\sigma_E^+/u\sigma_E^+]) \subseteq var(v\sigma_E^+) \cup var(u\sigma_E^+) \subseteq var(\Delta\sigma_E^+) \cup var(u\sigma_E^+). \tag{2.17}$$

Since the same inclusion holds for $var(w\sigma_{E_1}^+)$, (2.16) is verified, hence $n_1 \leq n_0$. But $\Phi_2(t, u, E)$ implies

$$t\sigma_E^+ \in var(\Delta\sigma_E^+) \cup var(t\sigma_E^+) \cup var(u\sigma_E^+),$$

and since $t\sigma_E^+$ is bound in E_1, it does not appear in $\Delta\sigma_{E_1}^+$. Thus, $n_1 < n_0$.

Case **sd**

$$\frac{\Delta \quad t \asymp u \quad \vdash \quad E}{\Delta \quad \propto(\, \delta(t,E)\sigma_E^j, \delta(u,E)\sigma_E^k\,) \quad \vdash \quad E} \quad \text{with } \Phi_4(t,u,E), j,k \in \mathcal{N}.$$

By hypothesis, $t\sigma_E^+ \approx u\sigma_E^+$, and by Corollary 2.29,

$$\propto(\, \delta(t,E)\sigma_E^j, \delta(u,E)\sigma_E^k\,)\sigma_E^+ = \propto(\, \delta(t,E)\sigma_E^j\sigma_E^+, \delta(u,E)\sigma_E^k\sigma_E^+\,) = \propto(\, t\sigma_E^+, u\sigma_E^+\,). \tag{2.18}$$

Hence, by Corollary 2.29,

$$var(\propto(\, \delta(t,E)\sigma_E^j, \delta(u,E)\sigma_E^k\,)\sigma_E^+) = var(t\sigma_E^+) \cup var(u\sigma_E^+).$$

This readily implies $n_1 = n_0$. Finally, again by Corollary 2.29, we have

$$\sum \#v\sigma_E^+ + \#w\sigma_E^+ < \#t\sigma_E^+ + \#u\sigma_E^+,$$

where the sum is on the equations of (2.18). This entails $m_1 < m_0$. □

Corollary 2.59 *Any instance (2.15) of a rule of U^X may be extended to a maximal proof in U^X.*

Proof By induction on the level of $\Delta_1 \vdash E_1$ using the above proposition, and the fact that two permutations are of the same level. □

Note the difference between the statement of this corollary, and the following one: all goals may be extended to a maximal proof in U^X. The latter only ensures that the procedure described above will terminate, while the former ensures that the following optimised algorithm is enough. For a given goal, suppose we have already constructed *one* proof of complexity m. If this proof is maximal, we are done. Otherwise, we arbitrarily choose *one* extension of complexity $m + 1$, and continue the process. A further optimisation is still possible. The conclusion of a proof contains enough information for determining whether the proof is maximal or not, and if not, to construct an extension. Hence in the preceding algorithm, instead of remembering the full proof, it suffices to store its conclusion.

Here we have a more operational description. Let s be a stack of equations, and let $\langle\rangle$ denote the empty stack. For t and u compatible terms, let $\propto(t,u) \Rightarrow s$ denote the result of pushing onto s all the equations in $\propto(t,u)$ in some unspecified order. Suppose that Φ is a function defined on all the triples (t,u,E) with values in $[1,5]$ such that $\Phi(t,u,E) = i$ iff $\Phi_i(t,u,E)$. Also, let $bind(t,u,E) = E \cup \{ t \asymp u \}$. In the algorithm below we make the rule **lb** functional. For this we choose the parameter k such that $u\sigma_E^k = \delta(u,E)$, and analogously for **rb**, and **sd**. We are left with the following widely known unification algorithm.

```
unify(s, E)
begin
        if  ( s = ⟨⟩ ) return(E);
        t ≍ u := pop(s);
        t = δ(t, E);  u = δ(u, E);
        case  ( Φ(t, u, E) ) of
        1:    return  unify(s, E);
        2:    return  unify(s, bind(t, u, E));
        3:    return  unify(s, bind(u, t, E));
        4:    return  unify(∝(t, u) ⇒ s, E);
        5:    return  (⊥);
        esac
end
```

Table 2.2: A Unification Algorithm

Note that here we are testing the conditions Φ_i *after* dereferencing. According to Corollary 2.49, this is correct. We end this section with additional simple properties of environments.

Proposition 2.60 *If x is bound in E, then for all $j \in \mathcal{N}$, $var(x\sigma_E^j) \subseteq var(E)$.* □

The proof may be easily done by induction on $depth(x,E)$.

Proposition 2.61 *If $E_0 \subseteq E_1$, then $\sigma_{E_1}^+ = \sigma_{E_0}^+ \sigma_{E_1}^+$.*

Proof Proposition 2.41 implies that for all $\theta \in unf(E_0)$, $\theta = \sigma_{E_0}^+ \theta$. But the same result and the hypothesis imply $\sigma_{E_1}^+ \in unf(E_1) \subseteq unf(E_0)$. □

Proposition 2.62 *$\sigma_E^+ \equiv \sigma_F^+$ implies $unf(E) = unf(F)$.* □

This follows immediately from Proposition 2.27.

2.3.2 An Abstract Unification Machine

Introduction

Unification is a fundamental operation in Logic Programming. According to some authors, [Mai90], [MD], it accounts for between 50% and 80% of Prolog's execution time. It is then natural to look for efficient implementation techniques.

For some time, the WAM has monopolized the market of abstract Prolog architectures. However, after some years of experience, researchers arrived at a deeper understanding of the unification operation and realized that the WAM unification fragment suffers from some flaws. Alternative approaches have then been developed. SUE is such an alternative. We present here a detailed account of it, and describe an algorithm for generating what we call the 'canonical' SUE code for a term or a PROLOG clause head. The canonical code of a term contains exactly one abstract instruction per term symbol, not counting parentheses and commas and is optimal in write mode. It corresponds to a depth-first traversal of the compiled term, instead of breath-first traversal as in the typical WAM code. The proposed architecture includes a certain number of substructure pointers S_i specialized to a given subterm 'depth' (this is made precise below), and instructions specialized to symbol 'positions' and depths. Finally, the canonical SUE code does not need temporary registers for storing structure arguments. This is made possible through the use of multiple substructure pointers.

The Proposed Architecture

In what follows, a *(SUE) word* is either a memory word or a register. All words are the same size. The registers are: H, the *Heap pointer*, E, the *environment pointer*, M, the *mode register*, a certain number of *substructure pointers* S_d, of *argument registers* A_j, and a stack, the Pushdown List. Of course, there is also a Program Counter, or Program Pointer, but we do not need to make explicit reference to it.

We introduce now some terminology and notation, and make some comments on data representation. In general, we will write terms using neither parentheses nor commas. By definition, $size(f) = arity(f) + 1$, for all functors f. The *size* of a term is the sum of the sizes of its functor *occurrences*.

A *value* is a sequence of binary digits of the same length as a SUE word. We use values for specifying SUE word contents. As the WAM, SUE uses *tagged values* for representing terms. A tagged value is a value partitioned into two fields: the *tag* field and what we call the *secondary* field (sfield). The contents of the tag field indicates what kind of term a given value represents. If the tag contents is *ctag,* (respectively, *vtag*, *stag*, *ltag,*) then the represented term is an *individual constant*[3] (respectively a variable, a structured term, a list). Lists are but a special case of structured terms, but for efficiency reasons, it is convenient to add a tag for them, store them in a special way and add some instructions to

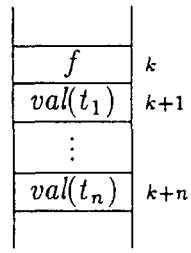
[3]also called *atom* elsewhere,

the architecture. However, once the basic principles are understood, these additions may be easily done, so we will not treat them in a special way. The reader is referred to the WAM literature.

We assume that at compile time each constant is associated a uniquely determined ctagged value in such a way that two constants are equal if and only if so are their respective values. From now on, then, we may use the same letter, typically, a, b, c, for representing both an individual constant and its associated value. The context will clearly indicate what the exact meaning is. Similarly, each functor is associated a value allowing SUE to differentiate between different functors, but we also require that the value associated to f encode the *size* of f. As for constants, we will use the same letter for denoting a functor and its associated value.

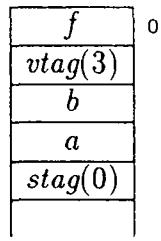
SUE represents variables through values of the form $vtag(k)$, with tag $vtag$ and sfield k. The secondary field contains a memory address, the *binding address* of variable $vtag(k)$. The tagged value contained in the memory word at address k, m[k], is the *binding* of $vtag(k)$. If the (current) binding of $vtag(k)$ is $vtag(k)$ itself, then the variable $vtag(k)$ is said to be (currently) *unbound.*

Structured terms may be arbitrarily long. For this reason, a value is in general not enough for storing all the information necessary to their representation. A term $ft_1 \ldots t_n$ will be stored in the form



where $val(t_i)$ is a value representing t_i in the current memory state. In this situation, the tagged value $stag(k)$ is a tagged value representing $ft_1 \ldots t_n$.

At run time the memory state encodes a substitution. But a variable may be bound to another variable, so SUE must differentiate between the current binding of a variable and the term associated to this variable by the current substitution. For exemple, in the memory state



the binding of $vtag(1)$ is $vtag(3)$, but the term associated to $vtag(1)$ is a, the binding of $vtag(3)$. This implies that the term assigned to $vtag(4)$ is $f\ a\ b$. SUE needs then to dereference a tagged value with respect to the current state. This is as in the unification system U. A tagged value is said to be *dereferenced* if it equals its dereferenced value. Table 2.3 lists the

instruction mnemonics. The d postfix in the m- and mt- instructions denote integers $1, 2, \ldots$, up to a certain number. This is discussed below.

The h-family	The m-family	The mt-family	The t-family
hfunct f,A$_j$	**mfunct**d f,k	**mtfunct**d f,k	**tfunct** f,k
hconst c,A$_j$	**mconst**d c,k	**mtconst**d c,k	**tconst** c,k
hvarfo i,A$_j$	**mvarfo**d i,k	**mtvarfo**d i,k	**tvarfo** i,k
hvarnfo i,A$_j$	**mvarnfo**d i,k	**mtvarnfo**di,k	**tvarnfo** i,k

Table 2.3: Instruction Mnemonics

A Compilation Algorithm

For compiling a structured term $t = ft_1 \ldots t_n$, we need to determine the *position* of each symbol occurrence in $ft_1 \ldots t_n$, the *depth* and the *offset* of each symbol occurrence in $t_1 \ldots t_n$. First, note that a functor occurrence g in t is followed by $m = arity(g)$ terms $u_1 \ldots u_m$. We say that $gu_1 \ldots u_m$ is the term occurrence *determined* by this occurrence of g.

We say that u is in *m-position*[4] in t iff for some i, $1 \leq i < n$, u is t_i, or, for some i, $1 \leq i \leq n$, u is in m-position in t_i; u is in *t-position*[5] in t iff u is t_n or u is in t-position in t_n; u is in *mt-position* in t iff for some term occurrence v in m-position in t, u is in t-position in v. For the sake of regularity, we say that u is in *h-position*[6] in t iff u is t itself.

By definition, a simple term is in h-position in itself, but it is neither in m- nor in mt- nor in t-position in itself. It follows that a subterm occurrence in t is, in t, in exactly one of the four defined positions. A functor occurrence f is said to be in x-position in t iff the term occurrence determined by f is in x-position in t. See below for an example.

Let u be a term occurrence in $t_1 \ldots t_n$. If, for some i, $1 \leq i \leq n$, u is t_i, the *depth* of u in t is 1. Let now $1 \leq i \leq n$, $t_i = gu_1 \ldots u_m$, and u a proper subterm occurrence in t_i. For defining the *depth* of u in t we need to differentiate the cases $i < n$, and $i = n$. If $i < n$, the *depth* of u in t equals the depth of u in t_i plus 1. If $i = n$, the *depth* of u in t equals the depth of u in $t_i = t_n$. As a consequence, the depth of a term in t-position is 1. From this and the definition of *depth*, the depth of all the t_i's in the list $[t_1, \ldots, t_n]$ is 1. An analogous remark holds for any right-sided structure.

Given a term occurrence u in $t_1 \ldots t_n$, there exists a uniquely determined integer j and a uniquely determined term occurrence $gu_1 \ldots u_{j-1}uu_{j+1} \ldots u_m$ in $ft_1 \ldots t_n$, $m = arity(g)$, such that u is the j-th argument of this term occurrence. By definition, the *offset* of the term occurrence u in $ft_1 \ldots t_n$ is the number

$$-(size(g) - j + \sum_{i=1}^{j-1} size(u_i)). \tag{2.19}$$

[4] middle-position

[5] terminal-position

[6] head-position

Given a symbol occurrence s in $t_1 \ldots t_n$, if s is a simple term, the depth and the offset of s in t are already defined; if s is a functor occurrence, the depth (respectively, the offset) of s in t, is the depth (respectively, the offset) in t of the subterm occurrence determined by s.

Suppose $size(f) = 4$, $size(g) = 3$ and $size(h) = 2$. In 2.20 we give a structured term, and the positions, depths and offsets of all the symbol occurrences, when they are defined.

$$\begin{array}{rrrrrrrrrr} f & g & a & h & Y_0 & b & g & a & h & Y_0 \\ h & m & m & mt & mt & m & t & m & t & t \\ & 1 & 2 & 2 & 2 & 1 & 1 & 1 & 1 & 1 \\ & -3 & -2 & -1 & -1 & -7 & -6 & -2 & -1 & -1 \end{array} \tag{2.20}$$

We may now show how to compile a clause head $p(t_1, \ldots, t_n)$ whose variables, if any, are $Y_0, \ldots, Y_m$, for some m. From this convention, the reader familiar with the WAM will correctly infer that in the algorithm described below, all variables are considered permanent. Further research is needed for the definition of non-trivial register allocation algorithms for temporary variables.

Here, the value representing variable Y_i will be $vtag(e + i)$, where e is the contents of register E during unification. We will not justify this. The reader is referred to the WAM literature for a more detailed discussion. Note that E is not modified by the instructions of the proposed architecture.

Let now t_j be an arbitrary head argument. The following rules define the *canonical SUE code* for t_j. Recall that A_j is an argument register.

- If t_j is the constant c, its canonical code is **hconst** c,A_j.
- If t_j is Y_i, and this is the first occurrence of Y_i in the compiled head, starting from the left, the code for t_j is **hvarfo** i,A_j.
- If t_j is Y_i and this is not the first occurrence of Y_i in the compiled head, the code for t_j is **hvarnfo** i,A_j.
- Suppose now that t_j is structured. We translate each symbol occurrence in t_j into a uniquely determined instruction.

  If f is the functor in h-position in t_j, it is translated into **hfunct** f,A_j.

  Consider now a symbol occurrence s other than the first in t_j. Suppose that s is in x-position in t_j, where x- is either m-, or mt-. Suppose that the depth (respectively, offset) of s in t_j is d (respectively, k).

  If s is an occurrence of c, its canonical code is x**const***d* c,k. If s is an occurrence of Y_i and this is the first occurrence of Y_i in the compiled head, s is translated into x**varfo***d* i,k. If s is an occurrence of Y_i and this is not the first occurrence of Y_i, s is translated into x**varnfo***d* i,k. If s is an occurrence of the functor g, it is translated into x**funct***d* g,k.

  If s is in t-position, the rules are analogous, except that we may ignore its depth, which is uniquely determined.

At this point, the meanings of the instruction mnemonics should be clear. For instance, **hfunct** stands for 'functor occurrence in head position', **mvarfo1** stands for 'first occurrence of a variable in m-position at depth 1', etc.

Assuming that (2.20) is the j-th argument of the compiled head, according to the previous rules, its canonical code is the following.

hfunct	f, A_j
mfunct1	$g, -3$
mconst2	$a, -2$
mtfunct2	$h, -1$
mtvarfo2	$0, -1$
mconst1	$b, -7$
tfunct	$g, -6$
mconst1	$a, -2$
tfunct	$h, -1$
tvarnfo	$0, -1$

Low Level Operations

For the specification of our instructions, we need *value specifiers*, *word specifiers*, and a certain number of low level operations. If c and f denote a constant and a functor, respectively, we use c, f and $size(f)$ as value specifiers, their meaning being clear from previous conventions. Integers are also value specifiers. We use *addressing modes* both as value specifiers and as word specifiers. They are listed in Table 2.4, where r denotes the contents of register R, and s the sfield of r. The value denoted by an addressing mode is the contents of the specified word. The addressing modes other than the first are the *memory word specifiers*; they specify a memory word through its address. Note that some addressing modes produce side-effects.

Addressing Mode	Specified Word	Side Effects
R	R	none
(R)	m[r]	none
(R+k)	m[r+k]	none
(R)++	m[r]	R $\leftarrow r+1$
[R]	m[s]	none
[R]++	m[s]	R $\leftarrow s+1$

Table 2.4: Addressing Modes

Table 2.5 lists the operations we will use. In this table, v, v_i (respectively, w, w_i) denote value (respectively, word) specifiers.

Basic Operations					
`move`	v, w	`inc`	v, w	`tst_struct`	v, l
`mvvar`	w_1, w_2	`dec`	v, w	`tst_const`	v, l
`mvstr`	f, w	`deref`	v, w	`continue`	
`compare`	v_1, v_2	`unify`	v_1, v_2	`fail`	

Table 2.5: Low Level Operations

We will not precisely specify the `fail` operation. Readers familiar with the WAM may fill in the details. Another possibility is to think it as a halt operation. This will be enough for our purposes. The first argument w_1 of `mvvar` must be a memory word specifier; if, at run time, it returns address k, say, then `mvvar` moves *vtag*(k) into w_2. If H = h, `mvstr` moves *stag*(h) into w, moves f into m[h], and increments H of *size*(f). This will be clarified below. `deref` stores the dereferenced value of v into w. `unify` is a general unification algorithm. It tries to unify its arguments in the current memory state. If this is not possible, SUE `fail`'s; otherwise, control continues at the next low level operation. `unify` is the only instruction allowed access to the Pushdown List and it is responsible for its management. `continue` marks the end of a SUE instruction.

`compare` compares its arguments. If they are equal, control continues at the next low level operation; otherwise, SUE `fail`'s. `tst_struct` analyzes the tag of v; if it is *vtag,* control continues at the next operation; if it is *stag,* control continues at the label l; otherwise, a failure occurs. `tst_const` is analogous. Note that none of the instructions mentioned in this paragraph dereferences its value argument(s).

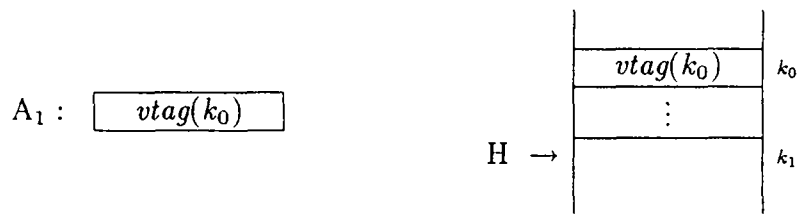
Execution Modes

In the following discussion, we suppose *size*(f) = 3 and *size*(g) = 2. According to our definition, if $f\ g\ a\ b$ is the first argument of a clause head, its canonical code is

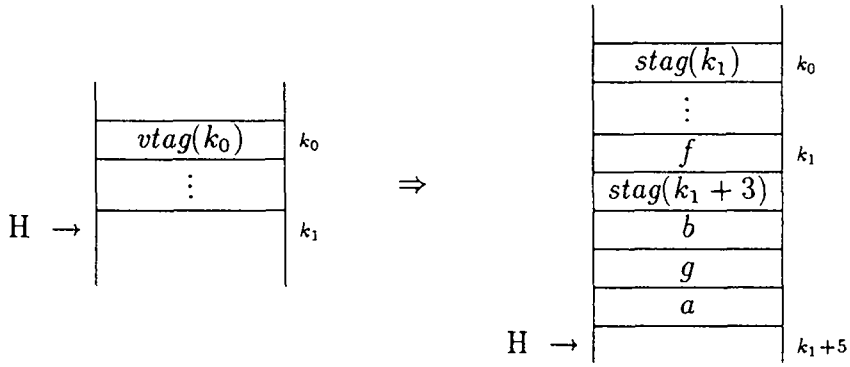
$$\begin{array}{ll} \textbf{hfunct} & f, \mathrm{A}_1 \\ \textbf{mfunct1} & g, -2 \\ \textbf{mtconst2} & a, -1 \\ \textbf{tconst} & b, -3 \end{array} \tag{2.21}$$

We suppose that at run time, when control arrives at the **hfunct** instruction above, A_1 will contain a dereferenced value, so, if it is a variable, it is unbound. The **hfunct** instruction of (2.21) will determine the kind of term the contents of A_1 represents, and will react according to the result of this analysis. Three essentially different situations may arrive.

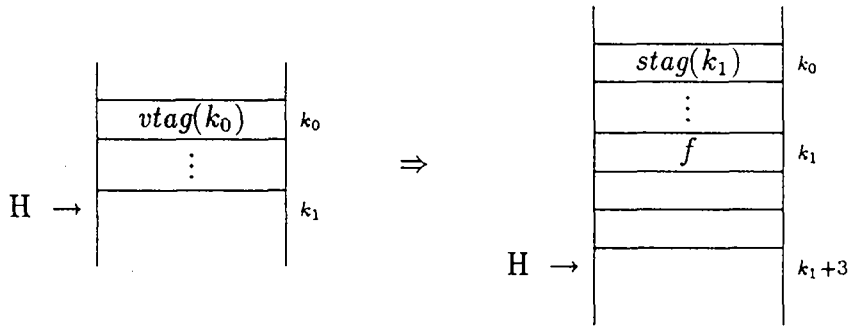
The first possibility is that A_1 contain an unbound variable. For instance, our machine state may be the following.



For unifying *f g a b* with $vtag(k_0)$ it suffices to construct a representation of the compiled term, store it at m[k_1], modify H appropriately and bind $vtag(k_0)$ to the constructed representation, that is $stag(k_1)$. We want then that our code (2.21) produce the following transformation.



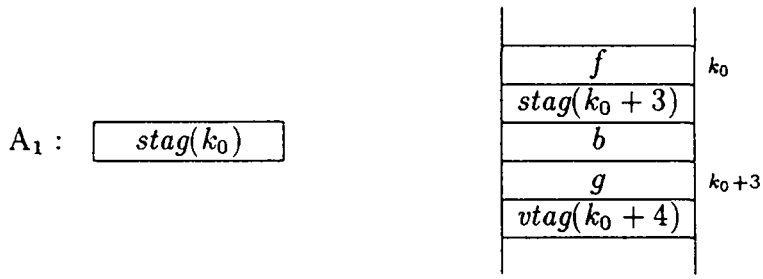
The **hfunct** instruction will fire the process with the transformation



and will execute `set_wmode` (i.e: `move` 1,M), declaring SUE to be in *write mode*. The following instructions, finding themselves in write mode, will complete the creation of the compiled term.

This reveals the meaning of the offset of a symbol occurrence in a structured term. In write mode, SUE represents structures in a canonical way, and in this representation, a structured term t occupies $size(t)$ contiguous memory words. The creation of such a representation starts with an **hfunct** or an *x***funct***d* instruction which, among other things, writes the functor at the memory word pointed at by H, and reserves space for the values representing the arguments (recall the operation `mvstr`). The following symbols, (i.e. instructions), knowing their own offsets, will be able to write themselves at the appropriate place. This clearly shows why the offsets are negative integers.

A second possibility is that (2.21) start running in a state like this.



In this case, SUE must traverse the arguments of the incoming structure, trying to unify them with the arguments of the compiled term. The **hfunct** instruction will `set_rmode` (i.e: `move 0,M`), declaring SUE to be in *read mode* and will leave S_1 pointing at the first depth-1 incoming argument, that is, $S_1 = k_0 + 1$. **mfunct1**, a depth-1 instruction, starting in read mode, will analyze the dereferenced value of the value pointed at by S_1, here, a structure with the expected functor g. It will then leave $S_2 = k_0 + 4$ pointing at the first depth-2 argument, and $S_1 = k_0 + 2$ pointing at the next depth-1 argument. **mtconst2**, a depth-2 instruction, will find its corresponding argument through S_2, and **tconst**, a depth-1 instruction, through S_1. If we guarantee that S_1 will not be modified by **mtconst2**, its saving/unsaving is unecessary.

Another subcase is possible here. **mfunct1** may find S_1 pointing at an unbound variable. It must then bind this variable, write the functor g on top of the Heap, reserve space for the argument list, and `set_wmode`. **mtconst2**, starting in write mode, must complete the creation of the substructure, and must `pop_wmode` (`dec 1,M`), for putting SUE back in read mode.

In general, an x**funct** instruction starting in read mode, may leave either in read or in write mode. In the first (respectively, second) case, we say it runs in *rr-mode* (respectively, *rw-mode*). It may also run in *ww-mode*, as we saw in the first case analyzed above, but it will never run in wr-mode. If a substructure starts in write mode, its surrounding structure is writing itself onto the Heap, in which case the substructure must participate in this task, hence it must also run in write mode.

The third possibility is that (2.21) start running with an incoming argument which is either a constant or a structured term with main functor other that f. In this case the unification is impossible, so SUE must `fail`.

Design Principles

Consider a compiled term occurrence t in m-position and at depth d in a head argument,

$$\ldots g\, u_1 \ldots u_{j-1}\, t\, u_{j+1} \ldots u_m \ldots . \tag{2.22}$$

If t starts running in read mode with $S_d = s$, it must try to unify its current instance[7] with the term represented by the value in m[s]. If it succeeds, it must leave SUE in read mode

[7] it may contain variables currently bound

with $S_d = s + 1$, for allowing u_{j+1}, which is also at depth d, to start in the appropriate state. But t may be structured, and some portions of it may run in write mode. It seems convenient to include a stack of mode flags for managing the possible mode changes. By convention, the current mode is the mode on top of the mode stack. Suppose now that our t above is structured, $t = ft_1 \ldots t_n$. When entered in read mode, f must decide whether t must run in read or write mode. In the first (respectively, second) case, it will push an R (respectively, a W) onto the mode stack. The arguments $t_1, \ldots, t_{n-1}$, all of which are in m-position, will leave the mode stack as they found it, and t_n, which is is mt-position, will pop the mode stack. But if a structured term runs in write mode, the same holds for its substructures. This implies that the contents of the mode stack will necessarily be W...WR...R, where the top is on the left. This allows us to apply the following optimization rules: a functor in m-position running in rr-mode, will not modify the mode stack, and a term in mt-position starting in read mode will not pop the mode stack. With some thought, we realize that we may replace the mode stack with an integer variable storing the number of W's on top of the replaced mode stack. This is precisely the role of the register M. Therefore, $M = 0$ means that SUE is in read mode, $M > 0$ that it is in write mode. With our canonical code, M cannot take negative values. Saying that a functor runs in rw-mode means, in fact, that it starts with $M = 0$ and executes `set_wmode`.

A slight difficulty appears, however. Suppose that t is $f(g(a, h(b)), c)$, and that we follow the rules mentioned above. If f runs in rr-mode and g in rw-mode, at a we will have $M = 1$; a is in m-position, so it will not modify M, h will `push_wmode` (i.e. `inc` 1,M), and b will `pop_wmode`, leaving $M = 1$. This violates our conventions: $g(a, h(b))$ is in m-position, hence it must leave M as it found it, $M = 0$ in this case. The solution is simple: a functor in mt-position, as h above, running in ww-mode, must *not* `push_wmode`.

The reader familiar with the WAM has certainly remarked various differences between SUE and WAM. SUE prefers the depth-first traversal of structured terms to the breath-first traversal of the typical WAM code. In write mode, SUE uses H-relative addressing, which is allowed by the compile-time computation of offsets. In a sense, the depth-first traversal of structures forces us to introduce the stack of mode flags for the correct management of mode changes. Finally, SUE provides itself with substructure pointers and instructions specialized on a given depth. It ensures that a depth-d' instruction, $d' > d$, will not modify S_d, which makes unnecessary the saving and unsaving of substructure pointers.

According to [Mai88], Prolog programmers do not use deeply nested structures, except, possibly, right-sided ones, like lists, which, with our definition of *depth*, are not very deep. For practical purposes, then, a limited number of registers S_d and of instructions **xxx***d* are enough.

The Instructions

Most instructions are specified on the next page. This is followed by hints for those not appearing there.

```
hfunct          f, A_j
        tst_struct A_j, l0
        mvstr      f, [A_j]
        set_wmode
        continue
l0:     compare    f, [A_j]++
        move       A_j, S_1
        set_rmode
        continue

mfunctd_read    f, k
        deref      (S_d)++, S_d+1
        tst_struct S_d+1, l1
        mvstr      f, [S_d+1]
        set_wmode
        continue
l1:     compare    f, [S_d+1]++
        continue

mfunctd_write   f, k
        mvstr      f, (H+k)
        push_wmode
        continue

mconstd_read    c, k
        deref      (S_d)++, S_d+1
        tst_const  S_d+1, l2
        move       c, [S_d+1]
        continue
l2:     compare    c, S_d+1
        continue

mconstd_write   c, k
        move       c, (H+k)
        continue

mvarfod_read    i, k
        move       (S_d)++, (E+i)
        continue

mvarfod_write   i, k
        mvvar      (H+k), (H+k)
        mvvar      (H+k), (E+i)
        continue

mvarnfod_read   i, k
        unify      (S_d)++, (E+i)
        continue

mvarnfod_write  i, k
        move       (E+i), (H+k)
        continue

mtfunctd_read   f, k
        deref      (S_d), S_d
        tst_struct S_d, l3
        mvstr      f, [S_d]
        set_wmode
        continue
l3:     compare    f, [S_d]++
        continue

mtfunctd_write  f, k
        mvstr      f, (H+k)
        continue

mtconstd_read   c, k
        deref      (S_d), S_d
        tst_const  S_d, l4
        move       c, [S_d]
        continue
l4:     compare    c, S_d
        continue

mtconstd_write  c, k
        move       c, (H+k)
        pop_wmode
        continue
```

Instructions corresponding to variable occurrences in mt-position are analogous to the corresponding version of **mtconst**. **tfunct** is like **mtfunct1**. **tconst_read** is like **mtconst1_read**.

tconst_write c, k

```
    move        c,(H+k)
    continue
```

Variable occurrences in t-position are treated similarly.

We do not include trailing, which is irrelevant to the unification proper. Note that each instruction i of the m-, mt- and t- families, has two codes: i_read and i_write, executed when i is entered in read and write mode, respectively. The h-family corresponds, roughly, to the WAM `get` family with differences discussed below. Each instruction transfers control to the instruction following it, that is, there are no jumps.

Note that simple term occurrences in t-position differ from the corresponding mt-versions in that the write code of the former does not include `pop_wmode`. This is a slight optimization allowed by the fact that the h-instructions are mode independent.

How to Use It

Some authors argue that a language for implementing unification must include 'enough' low level operations for allowing a smart compiler to generate optimized code, when additional static information is available. Following this proposal, we consider that the instructions of Table 2.5, or some variants of them, are part of the architecture; in particular, the `deref` instruction.

When we discussed execution modes, we remarked that our canonical code expects dereferenced arguments. But argument registers are loaded at well defined places. The compiler may then generate specialized instruction sequences guaranteeing that this convention will be respected. In the WAM, the argument registers are loaded by the instructions of the `put` family; but the `structure`, `list`, `constant` and `variable` members of the family, all put dereferenced values, while a `put_value` may be replaced by a `move` or a `deref`, depending on the compiler's ability to determine whether the moved value will be dereferenced or not at run time. At worst, the canonical code of each clause head argument may be preceded by an appropriate `deref` instruction.

It is useful to compare the efficiency of an ideal hardware implementation of SUE with that of the PLM, [Dob90], a loosely coupled coprocessor implementation of the WAM.

We note that mode testing in the PLM takes no time. If an instruction has different read and write codes, the microroutines implementing them are stored at different microstore addresses. At run time, when such an instruction is read, the microengine uses both the instruction opcode and the processor state (mode flag) for computing the appropriate microstore entry point.

It is sometines argued that the WAM is inefficient because of its intensive mode testing. The PLM shows that the criticism does not apply to a well designed hardware implementa-

tion. Of course, the same remark holds for our proposal. This explains why we have specified separately the read and write codes of our mode-sensitive instructions.

The typical WAM code includes one instruction, a `unify_variable`, per embedded functor occurrence, without counterpart in our canonical code. Consider, for instance, a clause head with $f\,g\,a$ as first argument.

WAM code		SUE code	
`get_structure`	f,A1	**hfunct**	f,A1
`unify_variable`	A1	**tfunct**	g,-1
`get_structure`	g,A1	**tconst**	a,-1
`unify_constant`	a		

Note that `mvstr` f, w writes two memory words and increments H. We argue that on a Data Path similar to that of the PLM, incrementing H may be done in parallel with the second memory write. With this in mind, we may safely say that **hfunct** is not less efficient than `get_structure`. In write mode, `unify_variable` initializes to unbound a variable in memory and loads A_1 with this unbound, and so, dereferenced, variable. Among other things, the following `get_structure` dereferences the already dereferenced incoming variable and trails it. Even after the addition, where needed, of trailing in our instructions, none of these operations are executed by our code above. In fact, in write mode our code is optimal in the following sense. A structured term t executed in write mode must write at least $size(t)$ memory words for constructing the term representation, plus one word for binding the incoming variable, plus one word per variable occurrence in t which is a first occurrence in the compiled head. Our code writes exactly this number of memory words, which is not the case of the WAM code. This has already been noted in [MD]. Our code also executes a `set_wmode` and some `push_wmode`/`pop_wmode`, but on a specialized processor, this may be done in parallel with the memory writes. Our code is similar to that in [MD]. However, according to the compilation rules suggested there, each embedded functor generates two instructions, a `push_structure` and a `push_functor`. Our code seems more appropriate for a specialized processor: it takes less instructions, executes the same number of memory writes and does not include jumps at the macroinstruction level.

A similar analysis leads us to argue that our x**funct***d* in read mode are comparable to `get_structure` in read mode. But our code does not need `unify_variable`.

We must note that even if our code typically includes less instructions than the corresponding WAM code, it may occupy more space. This is due to the offset argument in the m-, mt-, and t- families of instructions. Note, however, that this offset is bounded by the size of the compiled head argument and so, in practice, a byte should be enough for storing it.

A compiler generating code for a general purpose machine could use our instructions as intermediate language. In this case, however, the criticism concerning mode testing applies. It is better to follow the suggestions of [Mai90], [MD] and [Van89] of explicitly separating the read and write codes. Given a head argument, we write two copies of its canonical code, but in the first (respectively, the second) copy, we systematically replace each mode-sensitive

instruction by its read (respectively, write) code, with minor modifications. This is better understood through an exemple. Consider the term $f(g(a), h(foo))$. Our modified code is

```
            hfunct              f,A1,wa1
ra1:        mfunct1_read        g,wa1_alist
            mtconst2_read       a
ra2:        tfunct_read         h,wa2_alist
            tconst_read         foo
            jump                end
wa1:        mfunct_write        g,-2
wa1_alist:  mtconst_write       a,-1,ra2
wa2:        tfunct_write        h,-3
wa2_alist:  tconst_write        foo,-1
end:
```

No read code needs the offset argument, and no write code needs the depth postfix, so we simply eliminate them. We add a `label` argument to **hfunct**; in its code, we replace the first `continue` by `jump label` and we delete `set_rmode` and the last `continue`. We also add a `label` argument to **mtconst_write** and replace its `continue` by a `jump to label if M = 0`. The resulting code is quite close to the one proposed in [MD], except that we treat arguments in their natural order, which is not necessarily better.

Related Works

Our main source on the WAM is [AK91]. [GLLO88] is also good but less detailed. [Dob90] discusses the PLM. It gives a good idea of the actual cost of the various low level operations executed during unification. [Van89], [Mai90] and [MD] present alternatives to the typical WAM code. They discuss the problem of mode testing on general purpose machines and propose solutions. [Mai90] treats the compiled term depth-first, as we do, instead of breath-first, as the typical WAM code does. In write mode his code uses the S register for writing the argument list of structured terms; but then he is forced to saving/unsaving S in write mode, which makes his code non-optimal. Our code is freed from this flaw through the use of H-relative addressing, as in [Van89]. Our register M is similar to the L register of [MD]. However, the authors give no general compilation algorithm, and in their exemples, the problematic case of a functor in mt-position is not treated. They propose to modify the order of structure arguments. They argue that this may lead to a more efficient code in some cases. However, some results of the next section suggest that when the incoming arguments are unifiable with the compiled head, the number of unification steps is independent of the order of the arguments. This seems a debatable question. We know no other proposal where instructions and substructure pointers are specialized to a given depth.

2.3.3 Optimisation Issues

At each rule application (unification step) in system U^X, we have different choices at our disposal: what equation we will try to reduce next, that is, essentially, which exchange rule we allow ourselves to use, and eventually, what value(s) we will assign to the integer parameter(s) in the construction of the new environment. In this section we will explore how these different choices may affect the efficiency of the whole process. We will also consider the question of whether certain operations executed in the general algorithm may be safely eliminated.

It is easy to give examples of different maximal proofs of different complexities extending a given goal. For instance

$$\frac{a \asymp b \quad b \asymp b \quad \vdash}{a \asymp b \quad \vdash}$$

is maximal in U, while if

$$\frac{a \asymp b \quad b \asymp b \quad \vdash}{b \asymp b \quad a \asymp b \quad \vdash}$$

is the only instance of X with $a \asymp b \quad b \asymp b \vdash$ in the antecedent, then $a \asymp b \quad b \asymp b \vdash$ is maximal in U^X. Thus, at least when the unification is to fail, the choice of the equation to reduce, that is, of the exchange rule, may affect the maximality of the proof, hence the final number of unification steps.

For the purposes of the following discussion, let us denote by X_0 the full exchange rule. It is easy to see that[8]

$$\bigcup \mathrm{U}^X = \mathrm{U}^{X_0},$$

where the union is on all the exchange rules X. For a given goal $\Delta \vdash E$, consider now the set of all the instances

$$\frac{\Delta \vdash E}{\Pi \vdash F} \in \mathrm{U}^{X_0}.$$

It is easily seen that this set is finite, since there exists only a finite number of permutations $\Lambda \vdash E$ of $\Delta \vdash E$, and for each of them, eash rule of U contains only a finite number of instances

$$\frac{\Lambda \vdash E}{\Pi \vdash F}$$

Using this remark, and by induction on the level of $\Delta \vdash E$, we see that the set of all the proofs

$$\frac{\begin{array}{c}\Delta \vdash E \\ \vdots \end{array}}{\Lambda \vdash F}$$

(for a fixed goal $\Delta \vdash E$) which are maximal in *some* system U^X is finite, which implies that among them there exists one of minimal complexity. For the purposes of the following

[8]Recall that a rule is the set of all its instances.

discussion, let us say that such a proof is *optimal.* It is easy to define an optimal-proof-generating algorithm. Given any equational goal, we list all the maximal proofs extending it, and choose one of minimal complexity. This means that we have an algorithm which, given an equational goal, defines a unification strategy taking the least possible number of unification steps for this goal. The algorithm given above is too inefficient to be used in practice, but the fact that such an algorithm *exists* is in itself an interesting property. This suggests us to consider the following problem. Does there exists an 'efficient' optimal-proof-generating algorithm? At present, we have no answer to this question.

As the example above shows, if a goal is not unifiable, there may exist various maximal proofs of different complexities extending it. Interestingly enough, if the goal is unifiable and satisfies an additional syntactic condition, all the maximal proofs extending it are of the same complexity. This means that during the unification of such goals, the choice of the equation to reduce has no effect on the final number of unification steps.

Proposition 2.63 *Let* $\Delta \vdash E$ *be unifiable, and suppose that for all variables* x*, and all equations* $t \asymp u$ *of* Δ*,* $\#x\sigma_E^+ = \#t = \#u = 0$*. Then all refutations of* $\Delta \vdash E$ *are of complexity* $|\Delta|$*.*

Proof By induction on $|\Delta|$. Simply note that neither **rb**, nor **sd** can be applied to such goals, and that **ee**, and **lb** preserve the condition of the hypothesis. □

Note that the condition of the proposition is decidable, since the statement: 'for all $x \in \mathcal{V}$, $\#x\sigma_E^+ = 0$' is equivalent to 'for all $x \in ker(\sigma_E)$, $\#x\sigma_E^+ = 0$'.

This statement is of limited applicability due to the severe syntactic restrictions of the hypothesis, but it supports our belief that an analogous statement holds for *all* the unifiable goals. More precisely, we conjecture that *if* $\Delta \vdash E$ *is unifiable, then all its refutations on all the systems* U^X *are of the same complexity.* Otherwise said, if a goal is unifiable, the choice of the equation to reduce does not affect the number of unification steps. Several examples that we have tried have this property, and as we mentioned above, the last proposition is also an indication in this sense. Another indication is the result we want to show next, which says that *all maximal proofs in U extending any given goal are of the same complexity.* In order to prove this claim, we are naturally lead to apply induction on proofs. Suppose then that we have two instances of a rule of U.

$$\frac{\Delta \vdash E}{\Lambda_0 \vdash F_0} \qquad \frac{\Delta \vdash E}{\Lambda_1 \vdash F_1}$$

We quickly realise that we cannot inductively apply the statement we are willing to prove since $\Lambda_0 \vdash F_0$ and $\Lambda_1 \vdash F_1$ are different goals. The solution we will adopt is as follows. We will define an equivalence relation such that $\Lambda_0 \vdash F_0$ and $\Lambda_1 \vdash F_1$ are in this relation, and instead of proving the claim above, we will show that if two goals are in this relation, then all their maximal proofs in U are of the same complexity, and their conclusions are equivalent.

Definition 2.64 $\Delta \vdash E$ subsumes $\Lambda \vdash F$ *on* V *iff there exists a substitution* ϕ *such that*

$$\sigma_F^+ = \sigma_E^+\phi \text{ on } V, \qquad \text{and} \qquad \Lambda\sigma_F^+ = \Delta\sigma_E^+\phi. \tag{2.23}$$

$\Lambda \vdash F$ *is a* variant *of* $\Delta \vdash E$ *on* V *iff there exists a variable renaming* ϕ *on*

$$var(V\sigma_E^+) \cup var(\Delta\sigma_E^+), \tag{2.24}$$

such that (2.23) holds.

Corollary 2.65 *i. If* $\Lambda \vdash F$ *is a variant of* $\Delta \vdash E$ *on* V*, then the latter subsumes the former on* V*.*

ii. If $\Delta \vdash E$ *subsumes* $\Lambda \vdash F$ *on some set, and the latter is unifiable, then so is the former. As a consequence, if two goals are variants of each other, then either both of them, or none of them is unifiable.*

iii. If two goals subsume each other on some set, then they are variants on this set.

iv. The relation of being variants on a fixed set is an equivalence relation.

Proof We will only consider parts *ii* and *iv*. Concerning *ii*, suppose that $\Delta \vdash E$ subsumes $\Lambda \vdash F$ on V, that ϕ satisfies (2.23), and that θ is a unifier of $\Lambda \vdash F$. We will show that $\sigma_E^+\phi\theta$ is a unifier of $\Delta \vdash E$.

Trivially, $\sigma_E^+\phi\theta$ is a unifier of E. Also, since θ is a unifier of F, we have $\theta = \sigma_F^+\theta$, and from (2.23) we get $\Delta\sigma_E^+\phi\theta = \Lambda\sigma_F^+\theta = \Lambda\theta$. Since θ is a unifier of Λ, all the equation occurrences in $\Lambda\theta$ are of the form $t \asymp t$. Hence $\sigma_E^+\phi\theta$ is a unifier of Δ.

Concerning *iv*, let V be an arbitrary set of variables. Taking $\phi = id$ in the above definition, we readily get that all goals are variants of themselves on V. For the symmetry, suppose that we have a variable renaming ϕ on (2.24), such that (2.23) holds. Then there exists a ξ such that $\phi\xi = id$ on (2.24), which implies that ξ is a variable renaming on $var(V\sigma_E^+\phi) \cup var(\Delta\sigma_E^+\phi)$. But according to (2.23), this set is $var(V\sigma_F^+) \cup var(\Lambda\sigma_F^+)$, and we have $\sigma_E^+ = \sigma_E^+\phi\xi = \sigma_F^+\xi$ on V, and $\Delta\sigma_E^+ = \Delta\sigma_E^+\phi\xi = \Lambda\sigma_F^+\xi$, which shows that $\Delta \vdash E$ is a variant of $\Lambda \vdash F$ on V. The rest is left to the reader. □

Note that by part *ii*, for any two variants we have that either they are both refutable in U, or none of them is refutable in U. However, if they are both refutable, at this point we are unable to compare the lengths of their refutations.

Proposition 2.66 *Let* V *be an arbitrary set of variables, and suppose that* $\Lambda\ v \asymp w \vdash F$ *is a variant of* $\Delta\ t \asymp u \vdash E$ *on* V*. Then for all* $i = 1, \ldots, 5$*,* $\Phi_i(t, u, E)$ *iff* $\Phi_i(v, w, F)$*. As a consequence, a rule of U can be applied to* $\Delta\ t \asymp u \vdash E$ *iff it can be applied to* $\Lambda\ v \asymp w \vdash F$*.*

Proof By hypothesis, there exists a variable renaming ϕ on a set containing $var(t\sigma_E^+)$ and $var(u\sigma_E^+)$ such that, among other things, $v\sigma_F^+ = t\sigma_E^+\phi$, and $w\sigma_F^+ = u\sigma_E^+\phi$, which allows us to apply Proposition 2.48. □

Proposition 2.67 *Let V be a set of variables, $\Lambda_0\ v \asymp w \vdash F_0$ a variant of $\Delta_0\ t \asymp u \vdash E_0$ on V, let r be an arbitrary rule of U, and let*

$$\frac{\Delta_0\ t \asymp u \vdash E_0}{\Delta_1 \vdash E_1} \qquad \frac{\Lambda_0\ v \asymp w \vdash F_0}{\Lambda_1 \vdash F_1} \tag{2.25}$$

be two instances of r. Then $\Lambda_1 \vdash F_1$ is a variant of $\Delta_1 \vdash E_1$ on V.

Proof By hypothesis, there exists a variable renaming ϕ on

$$var(V\sigma^+_{E_0}) \cup var(\Delta_0\sigma^+_{E_0}) \cup var(t\sigma^+_{E_0}) \cup var(u\sigma^+_{E_0}) \tag{2.26}$$

such that

$$\sigma^+_{F_0} = \sigma^+_{E_0}\phi \text{ on } V, \quad \Lambda_0\sigma^+_{F_0} = \Delta_0\sigma^+_{E_0}\phi, \quad v\sigma^+_{F_0} = t\sigma^+_{E_0}\phi, \quad w\sigma^+_{F_0} = u\sigma^+_{E_0}\phi. \tag{2.27}$$

If r is **ee**, using (2.26), and (2.27), we can easily show that the conclusions of the instances in (2.25) are variants on V. Suppose now that r is **lb**, that the corresponding instances are

$$\frac{\Delta_0\ \ t \asymp u \vdash E_0}{\Delta_0 \vdash t\sigma^+_{E_0} \asymp u\sigma^j_{F_0}\ \ E_0} \qquad \frac{\Lambda_0\ \ v \asymp w \vdash F_0}{\Lambda_0 \vdash v\sigma^+_{F_0} \asymp w\sigma^k_{F_0}\ \ F_0} \tag{2.28}$$

and denote by E_1, and F_1 the environments in the left and right conclusions respectively. We know that

$$\sigma^+_{E_1} = \sigma^+_{E_0}[t\sigma^+_{E_0}/u\sigma^+_{E_0}], \qquad \sigma^+_{F_1} = \sigma^+_{F_0}[v\sigma^+_{F_0}/w\sigma^+_{F_0}].$$

From (2.26), (2.27), and Proposition 2.16, we get the following equalities on V.

$$\sigma^+_{F_1} = \sigma^+_{F_0}[v\sigma^+_{F_0}/w\sigma^+_{F_0}] = \sigma^+_{E_0}\phi[t\sigma^+_{E_0}\phi/u\sigma^+_{E_0}\phi] = \sigma^+_{E_0}[t\sigma^+_{E_0}/u\sigma^+_{E_0}]\phi = \sigma^+_{E_1}\phi,$$

and

$$\Lambda_0\sigma^+_{F_1} = \Lambda_0\sigma^+_{F_0}[v\sigma^+_{F_0}/w\sigma^+_{F_0}] = \Delta_0\sigma^+_{E_0}\phi[t\sigma^+_{E_0}\phi/u\sigma^+_{E_0}\phi] = \Delta_0\sigma^+_{E_0}[t\sigma^+_{E_0}/u\sigma^+_{E_0}]\phi = \Delta_0\sigma^+_{E_1}\phi.$$

This shows that the conclusions of (2.28) are variants on V. Finally, suppose that r is **sd**, and that our instances are

$$\frac{\Delta_0\ \ t \asymp u \vdash E_0}{\Delta_0 \propto (\,\delta(t, E_0)\sigma^{j_0}_{E_0}, \delta(u, E_0)\sigma^{k_0}_{E_0}\,) \ \vdash\ E_0} \qquad \frac{\Lambda_0\ \ v \asymp w \vdash F_0}{\Lambda_0 \propto (\,\delta(v, F_0)\sigma^{j_1}_{F_0}, \delta(w, F_0)\sigma^{k_1}_{F_0}\,) \ \vdash\ F_0}$$

From (2.27) we get

$$\delta(v, F_0)\sigma^{j_1}_{F_0}\sigma^+_{F_0} = v\sigma^+_{F_0} = t\sigma^+_{E_0}\phi = \delta(t, E_0)\sigma^{j_0}_{E_0}\sigma^+_{E_0}\phi,$$

and similarly,

$$\delta(w, F_0)\sigma^{k_1}_{F_0}\sigma^+_{F_0} = \delta(u, E_0)\sigma^{k_0}_{E_0}\sigma^+_{E_0}\phi.$$

Hence, by Corollary 2.29

$$\propto (\,\delta(v, F_0)\sigma^{j_1}_{F_0}, \delta(w, F_0)\sigma^{k_1}_{F_0}\,)\sigma^+_{F_0} = \propto (\,\delta(t, E_0)\sigma^{j_0}_{E_0}, \delta(u, E_0)\sigma^{k_0}_{E_0}\,)\sigma^+_{E_0}\phi.$$

From this equality, (2.26), (2.27), and Corollary 2.29, we conclude that the conclusions of the above instances of **sd** are variants on V. □

Corollary 2.68 *If $\Delta \vdash E$ and $\Lambda \vdash F$ are variants on V, then all the maximal proofs in U extending $\Delta \vdash E$ and all the maximal proofs in U extending $\Lambda \vdash F$ are of the same complexity. In particular, all the maximal proofs in U extending $\Delta \vdash E$ are of the same complexity.*

Proof By induction on $m \in \mathcal{N}$, we show that if we have a maximal proof in U of complexity m extending $\Delta \vdash E$, then all maximal proofs in U extending $\Lambda \vdash F$ are of complexity m. We use Proposition 2.66, and the above proposition. □

The number of unification steps certainly affects the efficiency of the unification process, but the testing of the conditions Φ_i for selecting the rule to apply must also be taken into account. Note that $\Phi_2(t, u, E)$ may be written in the form: $t\sigma_E^+ \in \mathcal{V}$ and $t\sigma_E^+ \notin var(u\sigma_E^+)$. The operational implementation of the second condition is usually termed 'the occur-check', and has been recognised as a relatively expensive operation. This has given rise to works on occur-check elimination.

We define the rule $\mathbf{lb}^-$ like **lb**, except that instead of demanding $\Phi_2(t, u, E)$, we simply demand $t\sigma_E^+ \in \mathcal{V}$, that is, $\mathbf{lb}^-$ is **lb** without the occur-check. $\mathbf{rb}^-$ is defined analogously. Let U^- be the system consisting of the rules **ee**, $\mathbf{lb}^-$, $\mathbf{rb}^-$, and **sd**.

It is evident that $\mathbf{lb} \subseteq \mathbf{lb}^-$, but the converse fails, since

$$\frac{x \asymp f(x) \quad \vdash}{\vdash \quad x \asymp f(x)}$$

is an instance of $\mathbf{lb}^-$. The example above shows that U^- allows us to 'unify' non-unifiable goals. This rises the question of characterising the set of *all* the goals for which U^- is a correct unification system. Here we will define a *proper* subset of this set.

A term is said to be *linear* iff no variable has multiple occurrences in it. It is immediate that if $f(t_1, \ldots, t_n)$ is linear, the same holds for the t_i's. Now, given a list Δ of equations, we define what does it mean for x to have a left (or right) occurrence in Δ, and for Δ to be left-linear.

Definition 2.69 *If $|\Delta| = 0$, then no variable has either a left or a right occurrence in Δ, and Δ is left-linear. x has a left (right) occurrence in Δ $t \asymp u$ iff either x has a left (right) occurrence in Δ, or $x \in var(t)$ ($x \in var(u)$). Δ $t \asymp u$ is left-linear iff Δ is left-linear, t is linear, and no variable of t has a left occurrence in Δ.*

Consider now the following conditions on the arbitrary goal $\Delta \vdash E$.

a. Δ is left-linear.

b. If x has a left occurrence in Δ, then x is unbound in E.

c. If x has a right occurrence in Δ, then neither x, nor the variables in $x\sigma_E^+$ have a left occurrence in Δ.

Proposition 2.70 *Suppose that the goal* $\Delta \;\; t \asymp u \;\vdash\; E$ *satisfies the conditions a-c above. Then*

$$\frac{\Delta \;\; t \asymp u \;\;\vdash\;\; E}{\Lambda \;\;\vdash\;\; F} \tag{2.29}$$

is an instance of a rule of U iff it is an instance of a rule of U^-*. Also, if (2.29) is an instance of a rule of U, then* $\Lambda \vdash F$ *satisfies a-c.*

Proof Note that by c

$$\text{no variable in } u\sigma_E^+ \text{ has a left occurrence in } \Delta \;\; t \asymp u. \tag{2.30}$$

From this and b, we get

$$var(t) \cap var(u\sigma_E^+) = \emptyset, \qquad \text{and} \qquad t\sigma_E^+ = t. \tag{2.31}$$

The first statement of our claim is trivial if (2.29) is an instance of **ee**, or **sd**. In order to prove that (2.29) is in **lb** iff it is in $\mathbf{lb}^-$, we need to show that $t\sigma_E^+ \in \mathcal{V} \setminus var(u\sigma_E^+)$ iff $t\sigma_E^+ \in \mathcal{V}$, but this is immediate from (2.31). The case of **rb** is similar.

Suppose now that (2.29) is an instance of **lb**, and that $\Lambda \vdash F$ is $\Delta \vdash t \asymp u\sigma_E^k \; E$. By the hypothesis a, Δ is left linear. By b, if x has a left occurrence in Δ, then x is unbound in E, and by a, $x \neq t$, hence x is unbound in F. Finally, suppose that x has a right occurrence in Δ. Then

$$var(x\sigma_F^+) = var(x\sigma_E^+[t/u\sigma_E^+]) \subseteq var(x\sigma_E^+) \cup var(u\sigma_E^+).$$

From this inclusion, the hypothesis c, and (2.30), we conclude that no variable of $x\sigma_F^+$ has a left occurrence in Δ. □

Corollary 2.71 *If* $\Delta \vdash E$ *satisfies a-c, then any maximal proof*

$$\frac{\begin{array}{c}\Delta \vdash E \\ \vdots \end{array}}{\Lambda \vdash F}$$

in U is a maximal proof in U^-*, and conversely.* □

This shows that U^- is a sound and complete unification system on the set of all the goals satisfying a-c. Note, however, that $x \asymp x \vdash$ is unifiable, it may be unified by U^-, but it does not satisfy condition c. This means that the set of goals satisfying conditions a-c is a proper subset of the goals that may be unified by U^-.

2.3.4 More on Unifiers

When working with unifiers, it is useful to have additional properties at hand. For instance, we have already seen that if $\theta \in unf(E)$, for any substitution ξ we trivially have $\theta\xi \in unf(E)$. Otherwise said, the set of unifiers of an arbitrary set E of equations is closed under composition with arbitrary substitutions. The set of principal unifiers of E is closed under composition with bijective substitutions.

Proposition 2.72 *If $\theta \in pru(E)$ and ϕ is a bijective substitution, then both $\theta\phi \in pru(E)$ and $\phi^{-1}\theta \in pru(E\phi)$ hold.* □

In fact, the first part of the previous statement holds in a slightly more general form.

Proposition 2.73 *If $\theta \in pru(E)$ and ϕ is a variable renaming, then $\theta\phi \in pru(E)$.*

Proof By hypothesis, if ξ is a unifier of E, $\theta \leq \xi$. Hence, by Corollary 2.23.*ii*, $\theta\phi \leq \xi$. □

The following two results give necessary conditions for θ to be a principal unifier of a set of equations. Both properties are, in some sense, merged in the corollary following them, so they may be safely ignored by the hurried reader.

Proposition 2.74 *Suppose that $\theta \in unf(E)$ and that x_0 is a variable not appearing in E. Then the following two conditions hold:*

i. If $x_0\theta$ is not a variable, then there exists a $\xi \in unf(E)$ such that $\theta \not\leq \xi$.

ii. If $x_0\theta$ is a variable and for some $y \in var(E)$ we have $x_0\theta \in var(y\theta)$, then there exists a $\xi \in unf(E)$ such that $\theta \not\leq \xi$.

Proof Take a variable renaming φ such that for all variables z, $z\varphi \neq x_0$. As a consequence, for all terms t, $x_0 \notin var(t\varphi)$. Define now ξ to be the substitution $\xi = id \downarrow \{ x_0 \} + \theta\varphi$. Obviously, $\xi \in unf(E)$.

According to the hypothesis of part *i*, $x_0\theta$ is not a variable. Hence none of its instances is a variable. This implies that for any substitution ϕ, $x_0\theta\phi \neq x_0 = x_0\xi$. Therefore, there exists no ϕ such that $\theta\phi = \xi$.

Considering now part *ii*, let us suppose that $\theta\phi = \xi$. By hypothesis, we have $x_0\theta \in var(y\theta)$ and since $x_0 = x_0\xi$, we have

$$x_0 = x_0\theta\phi \in var(y\theta\phi).$$

Under the current hypotheses, $y\theta\phi = y\xi$, so we have $x_0 \in var(y\xi)$. By hypothesis, $y \neq x_0$ and so, by the definition of ξ, $y\xi = y\theta\varphi$ and we conclude that $x_0 \in var(y\theta\varphi)$. This contradicts the choice of φ. □

Proposition 2.75 *Let $\theta \in unf(E)$, x_1 and x_2 two different variables not appearing in E and suppose that $x_1\theta = x_2\theta$. Then there exists a $\xi \in unf(E)$ such that $\theta \not\leq \xi$.*

Proof The hypotheses imply that for all ϕ, $x_1\theta\phi = x_2\theta\phi$. Defining $\xi = id \downarrow \{ x_1, x_2 \} + \theta$, we get $\xi \in unf(E)$ and $x_i\xi = x_i$, for $i = 1, 2$. If $\xi = \theta\phi$ were true for some ϕ, we would get

$$x_1 = x_1\xi = x_1\theta\phi = x_2\theta\phi = x_2\xi = x_2,$$

which is a contradiction. □

Corollary 2.76 *If $\theta \in pru(E)$, then θ is a variable renaming on the set of variables not appearing in E and if x is such a variable, $x\theta \notin var(E\theta)$.* □

Exemple: At first sight, $\theta = [x/z, y/z]$ may seem a principal unifier of $f(x) \asymp f(y)$, but the corollary tells us that this is not the case, because $z\theta = z$ appears in $f(x)\theta = f(z)$. This conclusion is confirmed by the fact that θ does not subsume the unifier $[x/y]$. □

2.4 Representability

Environments are syntactic tools for representing and computing substitutions. For θ and V given, the question arises of whether there exists an environment F such that $\sigma_F^+ = \theta$ on V. More general, we will need conditions ensuring that given θ, V, and E, there exists an environment $F \supseteq E$ such that $\sigma_F^+ = \theta\sigma_E^+ \downarrow V + \sigma_E^+$. This and related questions are at the heart of this section.

Definition 2.77 *We say that the environment E is* normal *iff $\sigma_E^+ = \sigma_E$.*

Proposition 2.78 *The following conditions are equivalent for all environments E.*

i. E is normal.

ii. σ_E is idempotent.

iii. $ker(\sigma_E) \cap var(ker(\sigma_E)\sigma_E) = \emptyset$.

Proof For all environments E, σ_E^+ is idempotent, hence i implies ii. The converse follows easily from Proposition 2.37. The equivalence of ii and iii follows from Proposition 2.10. □

Corollary 2.79 *If E is normal and $F \subseteq E$, F is normal.*

Proof If E is normal, it satisfies condition iii above, and this implies that any $F \subseteq E$ satisfies the same condition, hence is normal. □

We write $E \downarrow V = \{ x \succ t \mid x \in V \}$, and $E^n = \{ x \succ x\sigma_E^+ \mid x \in ker(\sigma_E) \}$, for E an environment.

Corollary 2.80 *i. If E is normal, $\sigma_{E\downarrow V}^+ = \sigma_E^+ \downarrow V + id$.*

ii. E^n is normal and $\sigma_{E^n}^+ = \sigma_E^+$.

□

Definition 2.81 *We say that the environment E* represents θ on V *iff $\sigma_E^+ = \theta$ on V. θ is* representable on V *iff there exists an environment representing θ on V.*

Note that by the above corollary, θ is representable on V if and only if it is representable on V by a normal environment. Following an already stated convention, 'E represents θ', and 'θ is representable' mean that E represents θ on $\mathcal{V}$, and that θ is representable on $\mathcal{V}$, respectively.

Proposition 2.82 *θ is representable on V if and only if $\theta \downarrow V + id$ is representable.*

Proof If E is a normal environment representing θ on V, we have $\sigma_E^+ = \theta$ on V, and by Corollary 2.7 and Corollary 2.80,

$$\theta \downarrow V + id = \sigma_E^+ \downarrow V + id = \sigma_{E\downarrow V}^+.$$

The converse follows easily. □

Corollary 2.83 *θ is representable if and only if it is representable on $ker(\theta)$.*

Proof For all θ, $\theta = \theta \downarrow ker(\theta) + id$. Hence, according to the preceding proposition, θ is representable on $ker(\theta)$ iff $\theta = \theta \downarrow ker(\theta) + id$ is representable. □

The following statement gives us a general mechanism for extending an environment, and precisely characterises the substitution represented by the result.

Proposition 2.84 *Let E_0 be an environment, $V = \{ x_1, \ldots, x_n \}$ a finite set of variables, and θ a substitution such that*

i. $V \cap var(V\theta) = \emptyset$, and

ii. all variables in $V \cup var(V\theta)$ are unbound in E_0.

Then

$$E_1 = E_0 \cup \{ x_i \asymp x_i\theta \mid 1 \leq i \leq n \}$$

is an environment, and $\sigma^+_{E_1} = \sigma^+_{E_0}[x_1/x_1\theta, \ldots, x_n/x_n\theta]$.

Proof Define $F_0 = E_0$, and for $1 \leq j \leq n$,

$$F_j = E_0 \cup \{ x_i \asymp x_i\theta \mid 1 \leq i \leq j \}.$$

We prove by induction on $j < n$ that if F_j is an environment with

$$\sigma^+_{F_j} = \sigma^+_{E_0}[x_1/x_1\theta, \ldots, x_j/x_j\theta], \tag{2.32}$$

then F_{j+1} is also an environment with

$$\sigma^+_{F_{j+1}} = \sigma^+_{E_0}[x_1/x_1\theta, \ldots, x_{j+1}/x_{j+1}\theta]. \tag{2.33}$$

From *ii*, x_{j+1} and all the variables in $var(x_{j+1}\theta)$ are unbound in E_0, and so, using *i* and the definition of F_j, we deduce that x_{j+1} and all the variables in $var(x_{j+1}\theta)$ are unbound in F_j. In particular, this implies

$$x_{j+1}\theta\sigma^+_{F_j} = x_{j+1}\theta. \tag{2.34}$$

Thus, under the current hypotheses, x_{j+1} is unbound in F_j and

$$x_{j+1} \notin var(x_{j+1}\theta\sigma^+_{F_j}).$$

Making $E = F_j$, $x = x_{j+1}$, $u = x_{j+1}\theta$, and $k = 0$ in Proposition 2.52, we get that F_{j+1} is an environment with

$$\sigma^+_{F_{j+1}} = \sigma^+_{F_j}[x_{j+1}/x_{j+1}\theta],$$

where, in this last equality, we used (2.34). But now, from (2.32) and condition *i* above, we may conclude (2.33). □

For the sake of notation, let us write
$R_1(\theta, V) : V \cap ker(\theta)$ is finite.
$R_2(\theta, V) : (V \cap ker(\theta)) \cap var((V \cap ker(\theta))\theta) = \emptyset$.

Theorem 2.85 *If $R_1(\theta, V)$, $R_2(\theta, V)$, and $V \cap var(E) = \emptyset$, then there exists an environment $F \supseteq E$ such that $\sigma_F^+ = \theta\sigma_E^+ \downarrow V + \sigma_E^+$.*

Proof Write $W = V \cap ker(\theta)$, and $\xi = \theta\sigma_E^+ \downarrow W + \sigma_E^+$. It is not difficult to verify that $\xi = \theta\sigma_E^+ \downarrow V + \sigma_E^+$. For all $x \in W$, we have $var(x\xi) = var(x\theta\sigma_E^+) \subseteq var(x\theta) \cup var(E)$. Hence

$$var(W\xi) \subseteq var(W\theta) \cup var(E).$$

Claim : $W \cap var(W\xi) = \emptyset$, *and all variables in* $W \cup var(W\xi)$ *are unbound in* E.
$R_2(\theta, V)$ says that $W \cap var(W\theta) = \emptyset$, while $V \cap var(E) = \emptyset$ implies $W \cap var(E) = \emptyset$. These observations and the above inclusion imply $W \cap var(W\xi) = \emptyset$.

Since $W \cap var(E) = \emptyset$, all variables in W are unbound in E, while Proposition 2.38 ensures that the same holds for those in $var(W\xi) = var(W\theta\sigma_E^+)$.

By $R_1(\theta, V)$, we may write $W = \{ x_1, \ldots, x_n \}$. Now, the above claim and Proposition 2.84 imply that $F = E \cup \{ x_i \asymp x_i\xi \mid 1 \leq i \leq n \}$ is an environment, with

$$\sigma_F^+ = \sigma_E^+[x_1/x_1\xi, \ldots, x_n/x_n\xi].$$

If $x \in W$, x is unbound in E, and for some i, $x = x_i$. Therefore,

$$x\sigma_F^+ = x_i[x_1/x_1\xi, \ldots, x_n/x_n\xi] = x_i\xi = x\xi.$$

If $x \notin W$ and x is bound in F, it is bound in E, and by Proposition 2.60, $var(x\sigma_E^+) \subseteq var(E)$. But we have already remarked that W and $var(E)$ are disjoint, hence $var(x\sigma_E^+) \cap W = \emptyset$. This implies

$$x\sigma_F^+ = x\sigma_E^+[x_1/x_1\xi, \ldots, x_n/x_n\xi] = x\sigma_E^+ = x\xi.$$

If $x \notin W$ and x is unbound in F, it is unbound in E. Thus $x\sigma_F^+ = x = x\sigma_E^+ = x\xi$. □

Corollary 2.86 *If V is finite, and $V \cap var(V\theta) = \emptyset = V \cap var(E)$, then there exists an environment $F \supseteq E$ such that $\sigma_F^+ = \theta\sigma_E^+ \downarrow V + \sigma_E^+$.*

Proof V finite implies $R_1(\theta, V)$, and $V \cap var(V\theta) = \emptyset$ implies $R_2(\theta, V)$. This means that we may apply the preceding proposition. □

Corollary 2.87 *If V is finite, $V \cap var(V\theta) = \emptyset = V \cap var(E)$, and all the variables in $var(V\theta)$ are unbound in E, then there exists an $F \supseteq E$ such that $\sigma_F^+ = \theta \downarrow V + \sigma_E^+$.*

Proof We simply apply the preceding corollary, taking into account that under the current hypotheses, $\theta\sigma_E^+ = \theta$ on V. □

The following statement gives a necessary and sufficient condition for θ to be representable on V.

Proposition 2.88 *θ is representable on V iff both $R_1(\theta, V)$ and $R_2(\theta, V)$ hold.*

Proof By Proposition 2.82, θ is representable on V iff $\theta \downarrow V + id$ is representable. If $R_1(\theta, V)$ and $R_2(\theta, V)$, we may apply the last theorem with $E = \emptyset$, which gives us an environment $\sigma_F^+ = \theta\sigma_\emptyset^+ \downarrow V + \sigma_\emptyset^+$. Now we simply recall that $\sigma_\emptyset^+ = id$. Conversely, suppose $\sigma_F^+ = \theta \downarrow V + id$. Hence,

$$V \cap ker(\theta) = V \cap ker(\sigma_F^+) \subseteq ker(\sigma_F^+) = ker(\sigma_F).$$

This inclusion readily implies $R_1(\theta, V)$, and with Corollary 2.40, it implies $R_2(\theta, V)$. □

Making $V = \mathcal{V}$ in this proposition, we get

Corollary 2.89 *θ is representable iff $ker(\theta)$ is finite and $ker(\theta) \cap var(ker(\theta)\theta) = \emptyset$.* □

For instance, $[x/f(x)]$ is not representable.

Proposition 2.90 *If V is finite and $\sigma_E^+ \leq \theta$ on V, there exists and environment $F \supseteq E$ such that $\sigma_F^+ \equiv \theta$ on V.*

Proof By hypothesis, there exists a ξ such that $\sigma_E^+\xi = \theta$ on V. Choose a variable renaming ϕ into $\mathcal{V}\backslash(var(E) \cup V)$. Trivially, $\sigma_E^+\xi\phi \equiv \theta$ on V. Defining $W = (var(E) \cup V)\backslash ker(\sigma_E)$, we readily get that $W \cap var(W\xi\phi) = \emptyset$, and that all variables in $W \cup var(W\xi\phi)$ are unbound in E. By Proposition 2.84, $F = E \cup \{ x \asymp x\xi\phi \mid x \in W \}$ is an environment, with $\sigma_F^+ = \sigma_E^+\xi\phi$ on V. □

Corollary 2.91 *If V is finite, for all θ, there exists an environment F such that $\sigma_F^+ \equiv \theta$ on V.*

Proof By the proposition, since $\sigma_\emptyset^+ = id \leq \theta$ on V, there exists an F such that $\sigma_F^+ \equiv \theta$ on V. □

2.5 Semantics

The classical mechanism for assigning truth values to formulas is the notion of *interpretation* of a given first order language[9].

Definition 2.92 *An* interpretation *of a first order language is a pair $(U, \mathcal{I})$, where U is a non-empty set, called the* universe *of the interpretation, and $\mathcal{I}$ is a function on the non-logical symbols of the language satisfying the conditions:*

- *for every constant c, $\mathcal{I}(c) \in U$,*
- *for every n-ary function symbol f, $\mathcal{I}(f)$ is an n-ary function from U into itself,*
- *for every n-ary predicate symbol p, $\mathcal{I}(p)$ is an n-ary relation in U.*

[9] also called *model* of the language.

By an abuse of notation, given an interpretation $(U, \mathcal{I})$, we will also use '$\mathcal{I}$' for denoting the interpretation itself.

Intuitively, a term acts as a name of an individual or object of the universe we are talking about. Given an interpretation $(U, \mathcal{I})$, this is clearly seen in the case of a constant c, which names the object $\mathcal{I}(c)$. However, an interpretation assigns no meaning to the variables, so we cannot talk about the object named by a term containing variables. This gap is filled in by the notion of *valuation.*

Definition 2.93 *Given an interpretation $\mathcal{I}$, a* valuation *in $\mathcal{I}$ is a function ν from the variables of the language into the universe of $\mathcal{I}$.*

Given an interpretation $\mathcal{I}$ and a valuation ν in $\mathcal{I}$, we have enough information for determining the meaning of any term t. If we let both the term t and the interpretation $\mathcal{I}$ fixed and let the valuations vary, we get a function defined on the set of all valuations in $\mathcal{I}$, that we denote by $t^{\mathcal{I}}$. Its value on the valuation ν, $t^{\mathcal{I}}(\nu)$, is the object of the universe represented by t if we interpret the proper symbols of t according to $\mathcal{I}$ and its variables according to ν. This is made precise as follows.

Definition 2.94 *Let $\mathcal{I}$ be an interpretation and ν a valuation in $\mathcal{I}$. Then*

- *for all constants c, $c^{\mathcal{I}}(\nu) = \mathcal{I}(c)$,*
- *for all variables x, $x^{\mathcal{I}}(\nu) = \nu(x)$,*
- *for all complex terms $f(t_1, \ldots, t_n)$, $f(t_1, \ldots, t_n)^{\mathcal{I}}(\nu) = \mathcal{I}(f)(t_1^{\mathcal{I}}(\nu), \ldots, t_n^{\mathcal{I}}(\nu))$.*

It is immediate that the value $t^{\mathcal{I}}(\nu)$ depends only on the values of ν on the variables of t. Thus if for all $x \in var(t)$, $\nu_1(x) = \nu_2(x)$, then $t^{\mathcal{I}}(\nu_1) = t^{\mathcal{I}}(\nu_2)$. As an immediate corollary, if t is closed, $t^{\mathcal{I}}$ is a constant, that is, for all ν_1, ν_2, $t^{\mathcal{I}}(\nu_1) = t^{\mathcal{I}}(\nu_2)$.

The assignement of truth values to formulas is done in a similar way. Given an atom $p(t_1, \ldots, t_n)$, we want it to be true if the n-tuple of objects denoted by $(t_1, \ldots, t_n)$ is in the relation denoted by p. But for identifying the n-tuple of objects denoted by $(t_1, \ldots, t_n)$ we need not only an interpretation but also a valuation. So, given $\mathcal{I}$ and ν, we want that $p(t_1, \ldots, t_n)$ be true iff $(t_1^{\mathcal{I}}(\nu), \ldots, t_n^{\mathcal{I}}(\nu)) \in \mathcal{I}(p)$. We express this by saying that ν *satisfies* $p(t_1, \ldots, t_n)$ *in* $\mathcal{I}$; in symbols, $\mathcal{I}, \nu \models p(t_1, \ldots, t_n)$. The definition is extended to more complex formulas taking into account the intuitive meanings of the connectives and quantifiers of the language.

The notion of *satisfaction* formalises the notion of *truth* of a formula for a given assignement of meanings to its proper symbols and free variables. The notion of *model* of a formula makes precise the idea of the formula being true independently of the meanings of its free variables. By definition, $\mathcal{I}$ is a model of α, written $\mathcal{I} \models \alpha$, if and only if for all valuations ν, $\mathcal{I}, \nu \models \alpha$.

A *universal closure* of α, written $\forall(\alpha)$, is any formula $(\forall x_1) \ldots (\forall x_n)(\alpha)$, where $x_1, \ldots, x_n$ is any list of variables containing all the variables having a free occurrence in α. The

existential closures of α, $\exists(\alpha)$ are defined analogously. It may be proved that $\mathcal{I} \models \alpha$ if and only if $\mathcal{I} \models \forall(\alpha)$. Otherwise said, a formula and its universal closures have the same models. Another simple property we will need is the following: given a clause $\alpha = A_0 \dots A_n$, $\mathcal{I} \models \alpha$ if and only if for all valuations ν, the following condition holds:

$$\text{if} \quad \text{for all } j,\ 0 \leq j < n, \quad \mathcal{I}, \nu \models A_j, \quad \text{then} \quad \mathcal{I}, \nu \models A_n. \tag{2.35}$$

An immediate consequence of this is that

$$\mathcal{I} \models A_0 \dots A_n \quad \text{if and only if} \quad \mathcal{I} \models A_{i_0} \dots A_{i_{n-1}} A_n, \tag{2.36}$$

where $A_{i_0}, \dots, A_{i_{n-1}}$ is any permutation of $A_0, \dots, A_{n-1}$.

A model of a set $\mathcal{A}$ of formulas is a model of all the formulas in $\mathcal{A}$. A formula α is said to be a *logical consequence* of the set $\mathcal{A}$ of formulas iff all models of $\mathcal{A}$ are also models of α. This will be denoted by $\mathcal{A} \models \alpha$.

We end this section with some technicalities that we will need in what follows. The Herbrand Universe of a language is the set of all its closed terms. Of course, for it to be non-empty, the language must contain at least one constant. If the Herbrand Universe is not empty, a set $\mathcal{A}$ of closed atoms uniquely determines a *Herbrand interpretation* $\mathcal{I}$ as follows: the universe of $\mathcal{I}$ is the Herbrand Universe and the interpretation function is defined:

- for all constants c, $\mathcal{I}(c) = c$,
- for all closed terms $f(t_1, \dots, t_n)$, $\mathcal{I}(f)(t_1, \dots, t_n) = f(t_1, \dots, t_n)$,
- for all closed atoms $p(t_1, \dots, t_n)$, $(t_1, \dots, t_n) \in \mathcal{I}(p)$ iff $p(t_1, \dots, t_n) \in \mathcal{A}$.

Thus, the universe and the interpretation of constant and function symbols of a Herbrand interpretation are uniquely determined by the language and that of the predicates is determined by the given set $\mathcal{A}$ of closed atoms. This idea may be naturally generalized taking arbitrary terms, not only closed ones.

Definition 2.95 *A set $\mathcal{A}$ of arbitrary atoms defines an interpretation $\mathcal{I}$ in the following way: the universe of $\mathcal{I}$ is the set of* all *terms and the interpretation function is defined*

- *for all constants c, $\mathcal{I}(c) = c$,*
- *for* all *terms $f(t_1, \dots, t_n)$, $\mathcal{I}(f)(t_1, \dots, t_n) = f(t_1, \dots, t_n)$,*
- *for* all *atoms $p(t_1, \dots, t_n)$, $(t_1, \dots, t_n) \in \mathcal{I}(p)$ iff $p(t_1, \dots, t_n) \in \mathcal{A}$.*

By definition, a valuation in such an interpretation is a function from the variables into the set of all terms. But previous remarks ensure that such a function may be identified with the unique substitution extending it. We leave to the reader the task of verifying the following result.

Proposition 2.96 *Let $\mathcal{A}$ be an arbitrary set of atoms and $\mathcal{I}$ the interpretation determined by $\mathcal{A}$ according to Definition 2.95. Then*

- *for any term t and substitution θ, $t^{\mathcal{I}}(\theta) = t\theta$,*
- *for any atom A and substitution θ, $\mathcal{I}, \theta \models A$ iff $A\theta \in \mathcal{A}$,*
- *for any atom A, $\mathcal{I} \models A$ iff all instances of A are in $\mathcal{A}$.*

□

2.6 Programs and Evaluators

Definition 2.97 *A* program *is a finite set of clauses. A* query *is a conjunction of atoms. An* answer *for query Q in program P is a substitution θ such that $P \models Q\theta$. $ans(Q,P)$ will denote the set of all answers for Q in P.*

Intuitively, a program P states what is accepted as true. If the query Q is closed, asking Q to P amounts to asking whether Q is true under the hypotheses P. Things are a little trickier if Q contains variables. In this case, asking Q amounts to asking for those assignements of values to the variables of Q making it true under the hypotheses P.

As was the case for substitutions, there are some differences between the notions we defined above and corresponding ones usually found in the literature. In order to discuss their relationship, let us recall some definitions from [Llo87], using the terminology we find there. A *definite program clause* is a formula $\forall(\neg A_0 \vee \ldots \vee \neg A_{n-1} \vee A_n)$[10], with $n \geq 0$; a *definite goal* is a formula $\forall(\neg A_0 \vee \ldots \vee \neg A_n)$; a *definite program* is a finite set of definite program clauses; given a definite program P and definite goal $G = \forall(\neg A_0 \vee \ldots \vee \neg A_n)$, an *answer* for $P \cup \{ G \}$ is a substitution θ such that $ker(\theta) \subseteq var(G)$, where $var(G)$ denotes the union of all the $var(A_i)$'s; a *correct answer* for $P \cup \{ G \}$ is an answer θ for $P \cup \{ G \}$ such that

$$P \models \forall((A_0 \wedge \ldots \wedge A_n)\theta).$$

With our definition of model, a clause $\forall(\neg A_0 \vee \ldots \vee \neg A_{n-1} \vee A_n)$ has the same models as $\neg A_0 \vee \ldots \vee \neg A_{n-1} \vee A_n$. Recalling that $\neg A \vee B$ and $A \rightarrow B$ have the same models, we see that $\neg A_0 \vee \ldots \vee \neg A_{n-1} \vee A_n$ has the same models as

$$A_0 \rightarrow (A_1 \rightarrow \ldots \rightarrow (A_{n-1} \rightarrow A_n) \ldots).$$

We can then safely say that our definition of clause is essentially the same as the standard one and the same holds for programs. In particular, we see that with the definition of model presented here, we may get rid of the universal quantifier in the object language. In [Llo87]

[10] Recall that the A_i's are atoms.

it is needed because there the notion of model is restricted to closed formulas. Interestingly enough, in standard writings the universal quantifier disappears by a metalanguage notational convention: a clause $\forall(\neg A_0 \vee \ldots \vee \neg A_{n-1} \vee A_n)$ is written $A_n \leftarrow A_0, \ldots, A_{n-1}$.

Consider now a goal $G = \forall(\neg A_0 \vee \ldots \vee \neg A_n)$. We know that

$$P \models \forall((A_0 \wedge \ldots \wedge A_n)\theta) \quad \text{iff} \quad P \models (A_0 \wedge \ldots \wedge A_n)\theta.$$

It follows that θ is an answer for $P \cup \{G\}$ iff it is an answer for the query $A_0 \wedge \ldots \wedge A_n$ in P. Otherwise said, the query $A_0 \wedge \ldots \wedge A_n$ and the goal G determine the same answers. Therefore, in this sense, they may be identified. Note, however, that these formulas have no common model. By our previous comments, $\forall(\neg A_0 \vee \ldots \vee \neg A_n)$ has the same models as $\forall(\neg(A_0 \wedge \ldots \wedge A_n))$, hence the same models as $\neg(A_0 \wedge \ldots \wedge A_n)$.

We know that if θ is an answer for $\forall(\neg A_0 \vee \ldots \vee \neg A_n)$, then $P \models (A_0 \wedge \ldots \wedge A_n)\theta$ and so, $P \models \exists(A_0 \wedge \ldots \wedge A_n)$. This shows that no model of P is a model of $\neg\exists(A_0 \wedge \ldots \wedge A_n)$; but this formula has the same models as the goal $\forall(\neg A_0 \vee \ldots \vee \neg A_n)$. We see then that a goal asks for those substitutions *refuting* it under the hypetheses P. This seems a rather contrieved way of conceptualizing the notion of answer, but may be justified as follows. Goals are used for defining SLD-Resolution and the *SLD-computed answers* for $P \cup \{G\}$. This being done, the completeness of SLD-Resolution takes the form: for every correct answer θ for $P \cup \{G\}$, there exists a computed answer ξ with $\xi \leq \theta$ on $var(G)$. In our view, this shows that the standard approach adopts an unnatural definition of *answer* for allowing a natural statement of the completeness of SLD-Resolution. This is quite acceptable if we plan to restrict ourselves to this formalism. But our work is essentially based on a different formalism, so we preferred to define the notion of *answer* as we did before, which seems more natural to us. Note, in passing, that with the standard definition of *answer*, it is quite natural to consider SLD-Resolution as a refutation formalism. We will see that a different but also appropriate interpretation is possible.

It seems conceptually useful to give an abstract characterization of answer computing algorithms before considering answer computing formalisms. We will be interested in *complete* algorithms, that is, those capable of computing all answers we need. So, defining *complete* evaluators amounts to defining 'how many' answers we need. The definition of *answer* places no restrictions on the action of an answer on the variables not appearing in the given query Q. This means that two answers θ_1 and θ_2 equal on $var(Q)$ may be seen as one and the same.

Definition 2.98 *A* complete program evaluator *is a function eval(Q,P) such that, given Q and P, eval(Q,P) is a set of answers for Q in P satisfying*

$$\forall\, \theta_1 \in ans(Q,P),\ \exists\, \theta_2 \in eval(Q,P) \text{ such that } \theta_2 = \theta_1 \text{ on } var(Q).$$

With this definition, a complete evaluator gives esentially 'all' answers. From a purely abstract point of view, this is perhaps the most natural definition, but practical considerations will lead us to replace it with a more appropriate one.

Once we agreed on what complete evaluators are, the natural step is to develop formal tools for representing them, or, equivalently, for computing answers. By far, the formal mechanism most widely used for this purpose is SLD-Resolution. Later we will see that other simple formal deductive systems are also appropriate to this purpose. There we will find useful to use atoms as a syntactic encoding of answers.

Definition 2.99 *Given P and $Q = A_0 \wedge \ldots \wedge A_n$, let $x_1 \ldots x_m$, where m may be 0, be a list containing all variables in Q in an arbitrary order and without repetitions. We choose an m-ary predicate symbol @ appearing neither in P nor in Q and an arbitrary permutation $A_{i_0} \ldots A_{i_n}$ of the atoms in Q. Now we define $P + Q$ to be the program*

$$P + Q = P \cup \{ A_{i_0} \ldots A_{i_n} @(x_1, \ldots, x_m) \}. \tag{2.37}$$

According to (2.36), if $A_{j_0}, \ldots, A_{j_{n-1}}$ is any other permutation, the programs

$$P \cup \{ A_{j_0} \ldots A_{j_n} @(x_1, \ldots, x_m) \} \quad \text{and} \quad P \cup \{ A_{i_0} \ldots A_{i_n} @(x_1, \ldots, x_m) \}$$

have the same models. Thus, the statement '$\mathcal{I} \models P + Q$' is unambiguous, even if the program denoted by '$P + Q$' is not uniquely determined. Now, proving the following result is a standard exercise.

Proposition 2.100 *The substitution*

$$[x_1/t_1, \ldots, x_m/t_m] \tag{2.38}$$

is an answer for Q in P if and only if

$$P + Q \models @(t_1, \ldots, t_m). \tag{2.39}$$

□

Knowing $@(t_1, \ldots, t_m)$ is not enough for constructing (2.38); we also need to know $var(Q)$ and the correspondence between variables and terms. We obtain all this information with the atom $@(x_1, \ldots, x_m)$, the head of the clause added to P in (2.37) for constructing $P + Q$. Therefore, for computing the answer (2.38) it suffices to being able to compute the atom (2.39) having $@(x_1, \ldots, x_m)$ at hand. From now on, then, the atom (2.39) will be identified with the corresponding substitution (2.38) and will also be called *answer*.

Chapter 3

To Answers through Proofs

3.1 Basic Proof Systems

It is a known fact that there exist sound and complete first order proof systems, that is, proof systems $\vdash$ satisfying the property:

$$\mathcal{A} \models \alpha \qquad \text{if and only if} \qquad \mathcal{A} \vdash \alpha, \tag{3.1}$$

for any set $\mathcal{A}$ of first order formulas and α a formula. With such a system at hand, the following corollary holds.

Corollary 3.1 *If* $var(Q) = \{ x_1, \ldots, x_m \}$, *then* $[x_1/t_1, \ldots, x_m/t_m]$ *is an answer for* Q *in* P *if and only if* $P + Q \vdash @(t_1, \ldots, t_m)$.

Proof According to Proposition 2.100, $[x_1/t_1, \ldots, x_m/t_m]$ is an answer for Q in P if and only if $P + Q \models @(t_1, \ldots, t_m)$, hence if our system satisfies (3.1), the preceding statement is equivalent to $P + Q \vdash @(t_1, \ldots, t_m)$. □

If we define $Th(\mathcal{A}) = \{ \alpha \mid \mathcal{A} \vdash \alpha \}$, and $@(\mathcal{A}) = \{ @(t_1, \ldots, t_m) \mid @(t_1, \ldots, t_m) \in \mathcal{A} \}$, for $\mathcal{A}$ a set of formulas in the language of $P + Q$, according to Definition 2.98, the function $eval(Q,P) = @(Th(P + Q))$ is a complete evaluator. This means that, in principle, any sound and complete first order proof system may be used as a formal mechanism for defining complete logic program evaluators. Practical considerations, however, suggest us to eliminate those deduction rules and formulas that are not needed to our purposes. For this, we note that the proof of Corollary 3.1 shows that for the validity of the result it suffices to have a system $\vdash$ acting on Horn clauses and satisfying the soundness and completeness property (3.1) for $\mathcal{A}$ a program and α an atom. A first possibility is to use the system SMP, whose rules are the following:

The Substitution Rule (S)

$$\frac{\beta}{\beta\theta}$$

for β an arbitrary clause and θ an arbitrary substitution,

Modus Ponens (MP)

$$\frac{B \rightarrow \beta \qquad B}{\beta}$$

for $B \rightarrow \beta$ an arbitrary implication.

SMP is a restriction of system HCC defined in []. We will write $\mathcal{A} \vdash_{SMP} \alpha$ for expressing that α is provable in $\mathcal{A}$ by a finite number of applications of **S** and **MP**. $Th_{SMP}(\mathcal{A})$ will denote the set

$$Th_{SMP}(\mathcal{A}) = \{\alpha \mid \mathcal{A} \vdash_{SMP} \alpha\}.$$

Similar notations will be used with other systems introduced below. The soundness of SMP, that is,

$$\mathcal{A} \vdash_{SMP} \alpha \qquad \text{implies} \qquad \mathcal{A} \models \alpha,$$

is easily established. The following theorem is the main result in our proof of the kind of completeness we are interested in here. The idea of the proof is essentially the same as that used in [] for proving the completeness of HCC.

Theorem 3.2 *For any set $\mathcal{A}$ of clauses, there exists an interpretation $\mathcal{I}$ such that*

i. $\mathcal{I} \models \mathcal{A}$,

ii. for all atoms A, if $\mathcal{I} \models A$ then $\mathcal{A} \vdash_{SMP} A$.

Proof Let $\mathcal{B} = \{A \mid \mathcal{A} \vdash_{SMP} A\}$, and let $\mathcal{I}$ be the interpretation determined by $\mathcal{B}$ according to Definition 2.95. We recall that in such an interpretation we may identify a valuation with the unique substitution extending it. For proving part *i*, we need to prove that for all $\alpha \in \mathcal{A}$, $\mathcal{I} \models \alpha$, that is,

$$\text{for all } \alpha \in \mathcal{A}, \text{ and for all substitutions } \theta, \quad \mathcal{I}, \theta \models \alpha.$$

Take then $\alpha = A_0 \ldots A_n \in \mathcal{A}$, and θ an arbitrary substitution. According to (2.35), it suffices to prove that for all θ,

$$\text{if} \quad \text{for all } j,\ 0 \leq j < n, \quad \mathcal{I}, \theta \models A_j, \quad \text{then} \quad \mathcal{I}, \theta \models A_n.$$

But from Proposition 2.96, for all $j \leq n$, $\mathcal{I}, \theta \models A_j$ if and only if $A_j\theta \in \mathcal{B}$, that is, if and only if $\mathcal{A} \vdash_{SMP} A_j\theta$. It suffices then to prove that for all substitutions θ, if

$$\text{for all } j < n, \quad \mathcal{A} \vdash_{SMP} A_j\theta \tag{3.2}$$

then

$$\mathcal{A} \vdash_{SMP} A_n\theta. \tag{3.3}$$

We supposed $A_0 \ldots A_n \in \mathcal{A}$, thus an application of **S** gives

$$\mathcal{A} \vdash_{SMP} (A_0 \ldots A_n)\theta.$$

But then, if (3.2) holds, n applications of **MP** give (3.3).

For part *ii*, if $\mathcal{I} \models A$, by Proposition 2.96, all instances of A are in $\mathcal{B}$. In particular, $A \in \mathcal{B}$. Recalling the definition of $\mathcal{B}$, this means that $\mathcal{A} \vdash_{SMP} A$. □

Corollary 3.3 (Completeness of SMP) *For all sets $\mathcal{A}$ of clauses and all atoms A, if $\mathcal{A} \models A$, then $\mathcal{A} \vdash_{SMP} A$. Otherwise said, SMP is capable of proving all atomic logical consequences of any given set of clauses.*

Proof For $\mathcal{A}$ given, take $\mathcal{I}$ an interpretation satisfying conditions *i* and *ii* of the theorem. By part *i*, $\mathcal{I} \models \mathcal{A}$. Thus, if $\mathcal{A} \models A$, we also have $\mathcal{I} \models A$. Now, $\mathcal{A} \vdash_{SMP} A$ follows from part *ii*. □

Since all the programs denoted by $P + Q$ have the same models, the statement '$P + Q \vdash_{SMP} A$' is unambiguous. More precisely, all the programs represented by the notation $P + Q$ prove the same atoms in SMP.

Corollary 3.4 *$eval(Q,P) = @(Th_{SMP}(P + Q))$ is a complete evaluator.* □

Other similar systems may be used. A *sequent* is an expression $\Gamma \vdash \Delta$, where Γ and Δ are finite, possibly empty sequences of formulas. The *intuitionistic* sequents are those containing at most one formula to the right of the turnstyle symbol $\vdash$. In particular, the intuitionistic sequents containing only atoms are of one of the two forms

$$A_0 \ldots A_{n-1} \vdash A_n, \qquad \text{or} \qquad A_0 \ldots A_{n-1} \vdash ,$$

that is, Horn clauses with a different syntax. We are allowed to identify the sequent $A_0 \ldots A_{n-1} \vdash A_n$ with the clause $A_0 \ldots A_n$ and the sequent $A_0 \ldots A_{n-1} \vdash$ with Lloyd's definite goal $\forall(\neg A_0 \vee \ldots \vee \neg A_{n-1})$. Consequently, we will freely mix these notations.

A sequent calculus is a set of formal rules acting on sequents. Typical examples of such rules, as given in [?], are the *left exchange* rule,

$$\frac{\Gamma\ G\ H\ \Delta \vdash \Lambda}{\Gamma\ H\ G\ \Delta \vdash \Lambda}$$

and the **CUT** rule,

$$\frac{\Gamma \vdash G\ \Delta \qquad \Lambda\ G \vdash \Pi}{\Gamma\ \Lambda \vdash \Delta\ \Pi}$$

Swapping the premises of this rule, if $|\Gamma| = |\Delta| = 0$, and Π is a sequence containing exactly one formula H, we get the following special case of the **CUT**

$$\frac{\Lambda\ G \vdash H \qquad \vdash G}{\Lambda \vdash H}$$

Taking into account the sequent representation of clauses and the evident similarities between **MP** and this restricted version of the **CUT**, we may be tempted to identify both rules. However, the identification is not completely satisfactory. For instance, if A, B and C are three different atoms, Modus Ponens may be applied in the form

$$\frac{A \rightarrow (B \rightarrow C) \qquad A}{B \rightarrow C}$$

while we cannot apply the **CUT**

$$\frac{A\ B \vdash C \qquad \vdash A}{?}$$

However, we are not particularly interested in identifying **MP** with a restricted version of the **CUT**, but in defining formal systems for computing atomic logical consequences of sets of clauses, and in this sense, replacing **MP** with a restricted form of the **CUT** *is* possible. Let us denote by **S&C** the system whose rules are:

The Substitution Rule (on sequents)

$$\frac{\Delta \vdash A}{\Delta\theta \vdash A\theta}$$

The (Restricted) CUT

$$\frac{\vdash A \qquad \Delta\ A \vdash B}{\Delta \vdash B}$$

It is not difficult to prove that given a set $\mathcal{A}$ of sequents of the form $A_0 \ldots A_{n-1} \vdash A_n$, the sequent $\vdash A$ is **S&C**-provable in $\mathcal{A}$ iff $\vdash A$ is a logical consequence of $\mathcal{A}$. This means that **S&C** proves essentially the same atoms as SMP does, even if this is not true for all clauses. Consequently, **S&C** could also be used for defining complete logic program evaluators. There is still another possibility, namely, the system HCC mentioned above.

We have seen that at least three quite similar but strictly speaking different formal deductive systems can be used as the formal base for the development of conceptually simple and complete logic program evaluators. Unfortunately, none of them is of practical utility. Any actual implementation of $@(Th_{SMP}(P+Q))$ must generate, sooner or later, all SMP-theorems of $P+Q$, set which is 'almost always' infinite. A similar remark holds for the other systems we saw above. Clearly, the responsible for this annoying situation is the Substitution Rule, which allows us to generate all instances of a clause already generated. However, we should not blame our poor systems for this; they only do what we demanded them to do. By the very definition of answer, if $@(t_1, \ldots, t_m)$ is an answer, so are all of its instances. This means that with our current definition of *completeness,* we want our evaluators to generate all the instances of the answers they generate. For practical purposes, then, our current notion of *completeness* is inappropriate. A complete evaluator should *characterise* the set of all answers, but not necessarily through the computation of all of them.

Definition 3.5 *A* sufficiently rich *(s.r.) set of answers for Q in P is a set $\mathcal{A}$ of answers for Q in P such that for any answer $@(t_1,\ldots,t_m)$, there exists $@(u_1,\ldots,u_m) \in \mathcal{A}$ satisfying*

$$@(u_1,\ldots,u_m) \leq @(t_1,\ldots,t_m).$$

From now on, a complete *evaluator will be a function eval(Q,P) such that, for all Q and P, eval(Q,P) is a sufficiently rich set of answers for Q in P.*

This means that we will content ourselves with the computation of a sufficiently rich set of answers. The obvious idea behind this decision is that once we computed an answer $@(u_1,\ldots,u_m)$, if $@(t_1,\ldots,t_m)$ is an instance of it, since we already know that it is an answer, its computation gives us no additional information. Note that the definition of *s.r. set of answers* does not depend on any particular formal system.

Of course, SMP is complete in this new sense, but now we are not only interested in the theoretical property of *completeness* but also in the computational property of *termination.* For this reason, and recalling our previous discussion, we must reject rule **S**. But **MP** alone is not sufficiently powerful for generating all atomic logical consequences of a program. Consider the example $P + Q$:

$$\begin{array}{l} p(x,y) \\ p(f(x),y) \rightarrow @(x,y) \end{array} \tag{3.4}$$

It is easy to see that $P + Q \vdash_{SMP} @(x,y)$, which implies that $@(x,y)$ is an answer for Q in P. Therefore, a complete evaluator must generate some variant of $@(x,y)$. It is evident, however, that this cannot be done with **MP** alone.

The situation is then the following: SMP is complete, but too powerful from a computational point of view, while **MP** alone is incomplete. It is natural to try an intermediate solution: given $B \rightarrow \beta$ and A, we try first to find two 'simple' substitutions θ_1 and θ_2 such that $B\theta_1 = A\theta_2$. If we find such substitutions, we construct the following deduction:

$$\dfrac{\dfrac{B \rightarrow \beta}{B\theta_1 \rightarrow \beta\theta_1}\,\theta_1 \qquad \dfrac{A}{A\theta_2}\,\theta_2}{\beta\theta_1}$$

This may be expressed by the rule

$$\dfrac{B \rightarrow \beta \qquad A}{\beta\theta} \tag{3.5}$$

where, for some bijective substitution ϕ satisfying

$$var(B \rightarrow \beta) \cap var(A\phi) = \emptyset, \tag{3.6}$$

$\theta \in pru(B,A\phi)$. The use of the variable renaming ϕ satisfying (3.6) is necessary for the completeness of the rule; otherwise, it cannot be applied to our previous example (3.4).

Intuitively, the soundness of the rule is suggested by the following argument: we have seen that we identify A with its universal closure $(\forall x_1)\ldots(\forall x_n)(A)$ from the semantical point of view. But bound variables have no actual identity, in the sense that we identify the previous universal closure with $(\forall z_1)\ldots(\forall z_n)(A[x_1/z_1,\ldots,x_n/z_n])$, and, as we remarked above, we identify this with $A[x_1/z_1,\ldots,x_n/z_n]$. So, for us, using A or $A[x_1/z_1,\ldots,x_n/z_n]$ is quite the same thing.

As was defined, the rule is not functional, in the sense that the conclusion is not uniquely determined by the premises. However, we have the following

Proposition 3.6 *Let $B \rightarrow \beta$, A, $\phi_i \in \Omega$ and $\theta_i \in pru(B,A\phi_i)$ be given, with*

$$var(B \rightarrow \beta) \cap var(A\phi_i) = \emptyset,$$

for $i = 0, 1$. Then $\beta\theta_0 \equiv \beta\theta_1$.

Proof Define $\xi_1 = \theta_1 \downarrow var(B \rightarrow \beta) \; + \; \phi_0^{-1}\phi_1\theta_1$. Then

$$B\xi_1 = B\theta_1 = A\phi_1\theta_1 = (A\phi_0)\phi_0^{-1}\phi_1\theta_1 = (A\phi_0)\xi_1.$$

From the hypotheses, $\theta_0 \leq \xi_1$, which implies $\beta\theta_0 \leq \beta\xi_1$. This and the definition of ξ_1 imply $\beta\theta_0 \leq \beta\theta_1$. Since the roles of θ_0 and θ_1 are symmetric, $\beta\theta_1 \leq \beta\theta_0$ also holds, and we are done. □

Even if this proposition is valid, we will find it convenient to make our rule functional through the introduction of the notion of *normal clause*. In passing, this will eliminate the need for the variable renaming we mentioned before, and will simplify the metamathematical treatement of our rule.

We suppose that the set $\mathcal{V}$ of all variables is partitioned into two disjoint and denumerable sets $\mathcal{V}_a$ and $\mathcal{V}_i$ that we suppose given by infinite sequences. We say that an atom is *normal* iff for each of its variable occurrences, either the corresponding variable appears to the left of this occurrence, or it is the first variable in $\mathcal{V}_a$ *not* appearing to the left. The normal implications are defined analogously, but using variables from $\mathcal{V}_i$. The normal programs are those all of whose clauses are normal. Trivially, if $B \rightarrow \beta$ and A are normal, they share no variable.

Example Suppose that x_0 and x_1 are the first two variables in $\mathcal{V}_a$. Then $p(a, x_0)$ and $p(x_0, f(x_1))$ are normal atoms, whereas $p(f(x_1), x_0)$ is not. □

It is easy to see that if $\beta \equiv \delta$ and both clauses are normal, then they are equal. Also, for any clause β, there exists a normal clause δ such that $\beta \equiv \delta$. This implies that each clause β has exactly one variant which is a normal clause, denoted by $\overline{\beta}$. We conclude that $\beta \equiv \delta$ if and only if $\overline{\beta} = \overline{\delta}$.

Definition 3.7 (The Rule UMP)

$$\frac{B \rightarrow \beta \qquad A}{\overline{\beta\theta}}$$

where $B \rightarrow \beta$ *and* A *are normal clauses and* $\theta \in pru(B,A)$.

Note that if $\theta \in pru(B,A)$, we can always find a $\xi \in \Omega$ such that $\beta\theta\xi = \overline{\beta\theta}$. By Proposition 2.72, $\theta\xi \in pru(B,A)$. Thus we could have defined the rule **UMP** in the form:

$$\frac{B \rightarrow \beta \qquad A}{\beta\theta}$$

where $B \rightarrow \beta$, A *and* $\beta\theta$ *are normal clauses and* $\theta \in pru(B,A)$. We may then write a rule application in this last form in which case we are implicitly assuming that $\beta\theta$ is normal.

Rule **UMP** allows us to think a normal clause as a partial function on normal clauses. For an implication $B \rightarrow \beta$, the domain of the associated function, $dom(B \rightarrow \beta)$, is the set of normal atoms A unifiable with B. The image of such an atom A, written $(B \rightarrow \beta)(A)$, is $\overline{\beta\theta}$, with $\theta \in pru(B,A)$. Atoms may be given an analogous interpretation. Under these conventions,

$$(B \rightarrow \beta)(A) = (A)(B \rightarrow \beta).$$

Also, we may write a rule application in the form

$$\frac{B \rightarrow \beta \qquad A}{(B \rightarrow \beta)(A)}$$

The reader familiar with logic program and database evaluation strategies, has certainly recognized rule **UMP** as the basic rule used by bottom up evaluation strategies with an unusual syntax. In these contexts, however, the rule is treated in an informal way.

In general, $P \vdash_{UMP} \alpha$ does not imply that α is normal, but if P is normal, so is α. From now on, except when explicitly stated otherwise, we agree in that 'program' means 'normal program'. This convention also applies to programs of the form $P + Q$.

It is easily seen that if $P \vdash_{UMP} \alpha$, then $P \vdash_{SMP} \alpha$. Hence the soundness of UMP follows from that of SMP. The completeness of UMP follows from the following

Proposition 3.8 *For all programs* P, *and clauses* α, *if* $P \vdash_{SMP} \alpha$, *then there exists a* β *such that* $P \vdash_{UMP} \beta$ *and* $\beta \leq \alpha$.

Proof By induction on the SMP-proofs. We only consider the Inductive Step over the application of **MP**. The rest is left to the reader. Suppose we have an SMP-proof

$$\frac{\begin{array}{cc}\vdots & \vdots \\ D \rightarrow \delta & D\end{array}}{\delta}$$

and assume, as I.H., that there exist $B \rightarrow \beta$, A, ξ_0, ξ_1 such that

- $P \vdash_{UMP} B \rightarrow \beta, \qquad P \vdash_{UMP} A,$

- $(B \rightarrow \beta)\xi_0 = D \rightarrow \delta, \qquad A\xi_1 = D.$

The second condition and the fact that $B \rightarrow \beta$ and A share no variable imply that we may suppose $\xi_0 = \xi_1$. Therefore, there exists a $\theta \in pru(B,A)$ such that

$$\frac{B \rightarrow \beta \qquad A}{\beta\theta}$$

is a correct application of UMP. Thus $P \vdash_{UMP} \beta\theta$. But ξ_0 is a unifier of B and A, hence $\theta \leq \xi_0$. From this and the equality $\beta\xi_0 = \delta$, we conclude $\beta\theta \leq \beta\xi_0 = \delta$. □

Corollary 3.9 (Completeness of UMP) *For any atom A and any set $\mathcal{A}$ of clauses, if $\mathcal{A} \models A$, then there exists B such that $B \leq A$ and $\mathcal{A} \vdash_{UMP} B$.*

Proof Immediate from the completeness of SMP, and the previous proposition. □

Corollary 3.10 *$eval(Q,P) = @(Th_{UMP}(P + Q))$ is a complete evaluator.*

Proof If $@(t_1, \ldots, t_m)$ is an answer, by Proposition 2.100, and the previous corollary, there exists an $@(u_1, \ldots, u_m)$ such that

$$P + Q \vdash_{UMP} @(u_1, \ldots, u_m)$$

and $@(u_1, \ldots, u_m) \leq @(t_1, \ldots, t_m)$. □

For any $P + Q$, $Th_{UMP}(P + Q) \subseteq Th_{SMP}(P + Q)$, thus any evaluator implementing the computation of $@(Th_{UMP}(P + Q))$ has better chances of stopping than one implementing $@(Th_{SMP}(P + Q))$. However, we cannot be satisfied with this. Consider the following example $P + Q$:

$$\begin{array}{l} p(x) \rightarrow p(f(x)) \\ p(y) \\ p(x) \rightarrow @(x) \end{array} \tag{3.7}$$

where we suppose that x (respectively y) is the first variable in $\mathcal{V}_i$ (respectively $\mathcal{V}_a$). It is not difficult to verify that $Th_{UMP}(P + Q)$ contains all the atoms of the form $p(f^k(y))$, for all $k \in \mathcal{N}$, and so, it is infinite, even if there exists a *finite* s.r. set of answers, namely, $\{ @(y) \}$. In general, then, we should try to compute not the whole $Th_{UMP}(P + Q)$ but a smaller set $\mathcal{A}$ of clauses, provided we are guaranteed that $\mathcal{A}$ contains enough answers.

Definition 3.11 *A* saturated extension *of a program P is a set $\mathcal{A}$ of clauses satisfying:*

i. $P \subseteq \mathcal{A} \subseteq Th_{UMP}(P)$.

ii. For every implication $B \rightarrow \beta \in \mathcal{A}$ and every atom $A \in \mathcal{A}$, if A is in the domain of $B \rightarrow \beta$, then there exists a clause $\delta \in \mathcal{A}$ such that $\delta \leq (B \rightarrow \beta)(A)$.

Clearly, for all P, $Th_{UMP}(P)$ is a saturated extension of P, but there may exist a smaller one. In example (3.7), $(P+Q) \cup \{ @(y) \}$ is a *finite* saturated extension of $P+Q$, even if $Th_{UMP}(P+Q)$ is infinite. The following proposition implies that computing saturated extensions is enough to our purposes.

Proposition 3.12 *If $\mathcal{A}$ is a saturated extension of P, then for all $\beta \in Th_{UMP}(P)$, there exists a $\delta \in \mathcal{A}$ such that $\delta \leq \beta$.*

Proof By induction on the UMP-proofs in P. The proposition trivially holds for $\beta \in P$. Suppose now we have an UMP-proof

$$\dfrac{\begin{array}{cc} \vdots & \vdots \\ D \rightarrow \delta & C \end{array}}{\delta\theta}$$

with $\theta \in pru(D,C)$. By I.H., there exist $B \rightarrow \beta$, $A \in \mathcal{A}$, and a substitution ξ such that

$$(B \rightarrow \beta)\xi = D \rightarrow \delta \qquad \text{and} \qquad A\xi = C.$$

We have then $B\xi\theta = D\theta = C\theta = A\xi\theta$, which means that $\xi\theta \in unf(B,A)$. There exists then a $\varphi \in pru(B,A)$ such that

$$(B \rightarrow \beta)(A) = \beta\varphi \leq \beta\xi\theta = \delta\theta.$$

Since $\mathcal{A}$ is saturated, there exists an $\alpha \in \mathcal{A}$ such that $\alpha \leq (B \rightarrow \beta)(A) \leq \delta\theta$. □

Corollary 3.13 *If $\mathcal{A}$ is a saturated extension of $P+Q$, then $@(\mathcal{A})$ is a sufficiently rich set of answers for Q in P.*

Proof $\mathcal{A} \subseteq Th_{UMP}(P+Q)$, so all the members of $@(\mathcal{A})$ are answers for Q in P. On the other hand, if $@(t_1,\ldots,t_m)$ is an answer for Q in P, by Corollary 3.10 there exists $@(u_1,\ldots,u_m)$ such that $@(u_1,\ldots,u_m) \leq @(t_1,\ldots,t_m)$ and

$$P+Q \vdash_{UMP} @(u_1,\ldots,u_m).$$

Now the previous proposition implies that there exists $@(v_1,\ldots,v_m) \in \mathcal{A}$ such that

$$@(v_1,\ldots,v_m) \leq @(u_1,\ldots,u_m) \leq @(t_1,\ldots,t_m).$$

□

We may then conclude that if $saturate(P)$ is any function returning a saturated extension of P, $@(saturate(P+Q))$ is a complete evaluator.

3.2 SLD-Resolution

3.2.1 Introduction

We have already remarked that SLD-Resolution is usually considered a refutation formalism. Here we will see that another view is possible. More precisely, we argue that SLD-Resolution may be seen as a backwards SMP-proof constructing mechanism. Let us clarify this idea.

Suppose we are given a program P, an atomic query B, and are asked to compute all the answers for B in P. We must then find those substitutions θ such that $P \models B\theta$. We know that SMP is capable of proving all atomic logical consequences of P, so our problem may be stated as follows: given P and B, find those pairs (π, θ) such that π is an SMP-proof of $B\theta$ in P. The naive solution, that of generating all pairs (π, θ) and rejecting those that are not useful for us, is in general non-terminating. We will then try to reduce the number of pairs we need to construct. However, if we want that our procedure return enough answers, this reduction cannot be done arbitrarily.

An idea we may explore is the following: we will try to identify a special kind of proofs such that, if a clause is provable, then it has a special proof. This being done, our problem will be reduced to the generation of those pairs (π, θ) where π is a special proof. For fixing ideas, consider the following program P,

$$p(x, g(y))$$
$$p(x, z) \rightarrow p(f(x), z)$$

and the following two SMP-proofs of $p(f(a), g(b))$ in P.

$$\dfrac{\dfrac{\dfrac{p(x,z) \rightarrow p(f(x),z)}{p(x,g(y)) \rightarrow p(f(x),g(y))} \quad p(x,g(y))}{p(f(x),g(y))}}{\dfrac{p(f(a),g(y))}{p(f(a),g(b))}}$$

$$\dfrac{\dfrac{p(x,z) \rightarrow p(f(x),z)}{p(a,g(b)) \rightarrow p(f(a),g(b))} \quad \dfrac{p(x,g(y))}{p(a,g(b))}}{p(f(a),g(b))}$$

The structure of the second proof is simpler that that of the first, thus more appealing from a computational point of view. Interestingly enough, the first proof may be transformed into the second through the application of the following rules:

- Two consecutive applications of **S** may be merged into one.

$$\dfrac{\dfrac{\beta}{\beta\theta}}{\beta\theta\xi} \quad \Rightarrow \quad \dfrac{\beta}{\beta\theta\xi}$$

- An application of **MP** and one of **S** following it may be swapped over as follows:

$$\dfrac{\dfrac{B \to \beta \qquad B}{\beta}}{\beta\theta} \quad \Rightarrow \quad \dfrac{\dfrac{B \to \beta}{B\theta \to \beta\theta} \qquad \dfrac{B}{B\theta}}{\beta\theta}$$

These rules allow us to transform any SMP-proof into a proof of the same clause from the same axioms, but where **S** is only applied to axioms, if applied at all. More formally, define $\mathcal{A}^{\Sigma}$ to be the set of all the clauses of the form $\alpha\theta$, where $\alpha \in \mathcal{A}$ and θ is an arbitrary substitution, that is, $\mathcal{A}^{\Sigma}$ is the set of all the instances of all the clauses in $\mathcal{A}$. Also, define $\mathcal{A}^{\equiv}$ to be the set of clauses $\alpha\phi$ with $\alpha \in \mathcal{A}$ and $\phi \in \Omega$, that is, $\mathcal{A}^{\equiv}$ is the set of all the variants of all the clauses in $\mathcal{A}$. Let us also write $\mathcal{A} \vdash^{m}_{MP} \alpha$ for expressing that α is provable in $\mathcal{A}$ with an MP-proof of complexity m. More precisely,

- $\mathcal{A} \vdash^{0}_{MP} \alpha$ if and only if $\alpha \in \mathcal{A}$.
- $$\dfrac{\mathcal{A} \vdash^{m_1}_{MP} B \to \beta \qquad \mathcal{A} \vdash^{m_2}_{MP} B}{\mathcal{A} \vdash^{m_1+m_2+1}_{MP} \beta}$$

The reader may verify the following

Proposition 3.14 *Let $\mathcal{A}$ be any set of clauses.*

i. If $\mathcal{A}^{\Sigma} \vdash^{m}_{MP} \alpha$, then for all θ, $\mathcal{A}^{\Sigma} \vdash^{m}_{MP} \alpha\theta$.

ii. For all α, $\mathcal{A} \vdash_{SMP} \alpha$ if and only if $\mathcal{A}^{\Sigma} \vdash_{MP} \alpha$.

□

Part *i* says that if α is MP-provable in $\mathcal{A}^{\Sigma}$ with a proof of complexity m, the same holds for all the instances of α. This implies that rule **S** is unnecessary for the generation of the SMP-theorems of $\mathcal{A}^{\Sigma}$. Part *ii* says that the **MP**-theorems of $\mathcal{A}^{\Sigma}$ are exactly the same as the SMP-theorems of $\mathcal{A}$. With this in mind, we can state our computational problem as follows: given P and B, find those pairs (π, θ) such that π is an MP-proof of $B\theta$ in P^{Σ}.

For the purposes of the following discussion, let us say that π is *atomic* iff the theorem proved by π, $th(\pi)$, is an atom. It is easy to see that an MP-proof in a set $\mathcal{A}$ of clauses is of

the form

$$\dfrac{\dfrac{\dfrac{\alpha^0 \qquad A_0}{\alpha^1} \qquad A_1}{\vdots}}{\dfrac{\alpha^{j-1} \qquad A_{j-1}}{\alpha^j}}$$

where $\alpha = A_0 \ldots A_n \in \mathcal{A}$ and $j \leq n$ [1]. In the figure, we do not suppose that the A_i's belong to $\mathcal{A}$; only that they are MP-provable in $\mathcal{A}$. We do not make explicit their proofs simply because we are not interested in analyzing them. We say that such a proof is *based on* α. Thus, an atomic MP-proof in $\mathcal{A}$ is either an atom $A \in \mathcal{A}$, or, for some clause $\alpha = A_0 \ldots A_n \in \mathcal{A}$ with $n > 0$, it is a proof of the form

$$\dfrac{\dfrac{\dfrac{\alpha^0 \qquad A_0}{\alpha^1} \qquad A_1}{\vdots}}{\dfrac{\alpha^{n-1} \qquad A_{n-1}}{\alpha^n}}$$

This means that if π is an atomic MP-proof in $\mathcal{A}$ not reducing to an atom, there exists an implication $\alpha \in \mathcal{A}$ such that π is based on α and $th(\pi) = \widehat{\alpha}$. In particular, if π is an atomic MP-proof in P^{Σ}, there exist an $\alpha \in P$, and a substitution φ such that π is based on $\alpha\varphi$, and $th(\pi) = \widehat{\alpha}\varphi$. More, suppose that π is an MP-proof of $B\theta$ in P^{Σ}. The previous discussion tells us that there exist an $\alpha \in P$, and a substitution φ such that π is based on $\alpha\varphi$, and $B\theta = th(\pi) = \widehat{\alpha}\varphi$. Take now a bijective substitution ϕ such that $var(B) \cap var(\alpha\phi) = \emptyset$. Trivially, $\beta = \alpha\phi \in P^{\equiv}$, π is based on

$$\alpha\varphi = (\alpha\phi)\phi^{-1}\varphi = \beta\phi^{-1}\varphi$$

and $\widehat{\beta}\phi^{-1}\varphi = B\theta$. Consider now the substitution $\xi = \theta \downarrow var(B) + \phi^{-1}\varphi$. From the previous discussion, we deduce that $B\xi = B\theta = \widehat{\beta}\xi$, and that π is based on $\beta\xi$. We have then the following

Fact If θ is an answer for B in P, there exist a formula $\beta \in P^{\equiv}$, and a pair (π, ξ) such that $var(B) \cap var(\beta) = \emptyset$, π is an MP-proof in P^{Σ} based on $\beta\xi$, and $\theta = \xi$ on $var(B)$. □

Only for the purposes of the following discussion, let us introduce the following terminology: we say that (π, θ) is *useful* for B iff there exists an $\alpha \in P^{\equiv}$ such that

$$var(B) \cap var(\alpha) = \emptyset,$$

[1] Recall that for $j < n$, $\alpha^j = A_j \rightarrow \alpha^{j+1}$ and $\alpha^n = \widehat{\alpha} = A_n$.

and π is an MP-proof in P^{Σ} of $B\theta = \widehat{\alpha}\theta$ based on $\alpha\theta$. The previous Fact now says that if θ is an answer for B in P, there exists a pair (π, ξ) useful for B in P with $\theta = \xi$ on $var(B)$. If we are only interested in determining the action of an answer on the variables of the query, our computational problem may be stated as follows: given P and B, compute all useful pairs for B in P. We need then an algorithm for generating useful pairs for B in P.

By definition, if (π, θ) is useful for B in P, there exist an $\alpha \in P$, and a $\phi \in \Omega$ such that $var(B) \cap var(\alpha\phi) = \emptyset$, and π is based on $\alpha\phi\theta$. This observation leads us to the following algorithm:

for all $\alpha \in P$
  for all $\phi \in \Omega$
    if $var(B) \cap var(\alpha\phi) = \emptyset$
      generate all useful pairs
      (π, θ) for B in P, with
      π based on $\alpha\phi\theta$.

The algorithm is clearly non-terminating due to the loop on all $\phi \in \Omega$. Fortunately, as we are going to see below, the following simplified version is enough to our purposes.

for all $\alpha \in P$
  choose one arbitrary $\phi \in \Omega$ with $var(B) \cap var(\alpha\phi) = \emptyset$;
  generate all useful pairs for B
  in P whose proofs are based
  on some *instance* of $\alpha\phi$.

This may be justified as follows. We are only interested in computing answers for B, so, if (π_0, θ_0) and (π_1, θ_1) are two useful pairs for B with $\theta_1 = \theta_0$ on $var(B)$, once we computed (π_0, θ_0), the computation of (π_1, θ_1) becomes useless.

Proposition 3.15 *Let $\alpha \in P$ and $\phi_i \in \Omega$ with $var(B) \cap var(\alpha\phi_i) = \emptyset$, for $i = 0, 1$. If (π, θ_1) is a useful pair for B with π based on $\alpha\phi_1\theta_1$, then there exists θ_0 such that $\theta_0 = \theta_1$ on $var(B)$, π is based on $\alpha\phi_0\theta_0$, and (π, θ_0) is useful for B in P.*

Proof Define $\theta_0 = \phi_0^{-1}\phi_1\theta_1 \downarrow var(\alpha\phi_0) + \theta_1$. It is immediate that $\theta_1 = \theta_0$ on $var(B)$. Also,

$$(\alpha\phi_0)\theta_0 = (\alpha\phi_0)(\phi_0^{-1}\phi_1\theta_1) = \alpha\phi_1\theta_1,$$

which implies that π is based on $\alpha\phi_0\theta_0$. Finally, and by hypothesis, $B\theta_1 = \widehat{\alpha}\phi_1\theta_1$. Hence

$$B\theta_0 = B\theta_1 = \widehat{\alpha}\phi_1\theta_1 = \widehat{\alpha}\phi_0\theta_0.$$

This implies that (π, θ_0) is useful for B. □

Suppose then that we fixed $\alpha \in P$, and $\phi_0 \in \Omega$, with $\alpha\phi_0$ sharing no variable with B, and suppose that we have an algorithm generating all the useful pairs (π, θ_0) for B, with π based

on $\alpha\phi_0\theta_0$. Now, the proposition tells us that choosing a different $\phi_1 \in \Omega$ with $\alpha\phi_1$ sharing no variable with B, and computing useful pairs (π_1, θ_1) based on $\alpha\phi_1\theta_1$ will give us no new answer.

Let then $\alpha \in P$, $\phi \in \Omega$ with $var(B) \cap var(\alpha\phi) = \emptyset$. If (π, θ) is a useful pair for B with π based on $\alpha\phi\theta$, then from the definition of *useful pair*, we get that $B\theta = \hat{\alpha}\phi\theta$; therefore B and $\hat{\alpha}\phi$ are unifiable. Reversing the argument, we get the following

Fact If $\hat{\alpha}\phi$ and B *are not unifiable,* then there exists no useful pair for B based on an instance of $\alpha\phi$.

This means that, in our algorithm, before trying to construct any useful pair (π, θ) with π based on $\alpha\phi\theta$, we should try to unify B with $\hat{\alpha}\phi$. If the unification fails, the previous Fact tells us that we may let both α and ϕ drop and try another clause $\alpha' \in P$ and another substitution $\phi' \in \Omega$; no answer will be lost.

Suppose now that for α and ϕ as before, we have computed a unifier θ of B and $\hat{\alpha}\phi$. If there exists a useful pair (π, θ) with π based on $\alpha\phi\theta$, by the definition of *useful pair*, we must have

$$P^{\Sigma} \vdash_{MP} B\theta. \tag{3.8}$$

However, the sole fact that θ is a unifier of B and $\hat{\alpha}\phi$, does not imply (3.8), hence does not imply that there exists a useful pair for B based on $\alpha\phi\theta$. Take, for instance, the program P

$$\begin{array}{l} q(y) \rightarrow p(y) \\ q(a) \end{array}$$

and $B = p(x)$. Choosing $\alpha = q(y) \rightarrow p(y)$ and $\phi = id$, we have that $\theta = [x/y]$ is a unifier of B and $\hat{\alpha}\phi$, but

$$P^{\Sigma} \vdash_{MP} B\theta = p(y)$$

does not hold. Note, however, that

$$\frac{q(a) \rightarrow p(a) \qquad q(a)}{p(a)}$$

is an MP-proof in P^{Σ}, but not of $B\theta = p(y)$, but of an *instance* of it. The moral of the tale is that given $\theta \in unf(B,\hat{\alpha}\phi)$, even if there may not exist useful pairs (π, θ) with π based on $\alpha\phi\theta$, there may exist useful pairs $(\pi, \theta\varphi)$ with π based on $\alpha\phi\theta\varphi$, for some φ. So, once we computed the unifier θ as before, in order to give our algorithm more chances of succeeding, instead of trying to construct useful pairs (π, θ), we should try to construct useful pairs $(\pi, \theta\varphi)$, for some φ. Let us see how.

Suppose then that P and B are given, that we have chosen an $\alpha \in P^{\equiv}$ sharing no variable with B, and that we computed an unifier $\theta_2 \in unf(B,\hat{\alpha})$. We will try to define a formal process for constructing useful pairs for B of the form $(\pi, \theta_2\varphi)$ based on $\alpha\theta_2\varphi$, for some φ. For fixing ideas, suppose that $\alpha = A_0A_1A_2$.

Our first step may be to construct the pair

$$\dfrac{\dfrac{\alpha^0\theta_2 \qquad A_0\theta_2}{\alpha^1\theta_2} \qquad A_1\theta_2}{\alpha^2\theta_2} \quad , \quad \theta_2 \tag{3.9}$$

Under the current hypotheses, we have the following facts:

- $\alpha^0\theta_2 \in P^\Sigma$. In particular, $\alpha^0\theta_2$ is MP-provable in P^Σ.
- The object on the left of (3.9) is an MP-proof in $P^\Sigma \cup \{ A_0\theta_2, A_1\theta_2 \}$.
- $B\theta_2 = \alpha^2\theta_2$.

From this we conclude that if the proof on the left of (3.9) is an MP-proof in P^Σ, then (3.9) is a useful pair for B. Unfortunately, at this point we are unable to guarantee that the $A_i\theta_2$'s are MP-provable in P^Σ. In order to explicitly represent this state of affairs, we may write

$$\dfrac{\dfrac{\alpha^0\theta_2 \qquad A_0\theta_2\ ?}{\alpha^1\theta_2} \qquad A_1\theta_2\ ?}{\alpha^2\theta_2} \quad , \quad \theta_2 \tag{3.10}$$

It is almost obvious what our next step should be: we should try to construct a useful pair for $A_1\theta_2$. If such a pair exists, and we are smart enough, after some time we will come back with a useful pair (π_1, θ_1) for $A_1\theta_2$, hence we will be guaranteed that $A_1\theta_2\theta_1 = th(\pi_1)$ is MP-provable in P^Σ. In general, we cannot replace $A_1\theta_2$ with $A_1\theta_2\theta_1$ and its proof π_1 in (3.10) because this would result in a wrong application of MP. However, we may replace the whole pair (3.10) with

$$\dfrac{\dfrac{\alpha^0\theta_2\theta_1 \qquad A_0\theta_2\theta_1\ ?}{\alpha^1\theta_2\theta_1} \qquad A_1\theta_2\theta_1}{\alpha^2\theta_2\theta_1} \quad , \quad \theta_2\theta_1 \tag{3.11}$$

where we do not represent the proof π_1. Note that the process is sound. From $B\theta_2 = \alpha^2\theta_2$, we get $B\theta_2\theta_1 = \alpha^2\theta_2\theta_1$. Therefore, if the proof in (3.11) is an MP-proof in P^Σ, (3.11) is a useful pair for B. Repeating the process with $A_0\theta_2\theta_1$, and with some chance, we will end with

$$\dfrac{\dfrac{\alpha^0\theta_2\theta_1\theta_0 \qquad A_0\theta_2\theta_1\theta_0}{\alpha^1\theta_2\theta_1\theta_0} \qquad A_1\theta_2\theta_1\theta_0}{\alpha^2\theta_2\theta_1\theta_0} \quad , \quad \theta_2\theta_1\theta_0 \tag{3.12}$$

a useful pair for B.

The previous discussion clearly suggests another formal representation of this process. For instance,

$$\dfrac{\dfrac{\alpha^0 \qquad A_0\ ?}{\alpha^1 \qquad A_1\ ?}}{\alpha^2} \quad , \quad \theta_2 \tag{3.13}$$

is a sufficiently accurate representation of (3.10); after all, both representations contain the same amount of information. Furthermore, if we use environments for representing substitutions, (3.12), say, may be represented by

$$\dfrac{\dfrac{\alpha^0 \qquad A_0}{\alpha^1 \qquad A_1}}{\alpha^2} \quad , \quad E_0 \tag{3.14}$$

provided that $\sigma^+_{E_0} = \theta_2\theta_1\theta_0$. If this is the case, $\sigma^+_{E_0}$ is an answer for B in P.

If we are only interested in computing this E_0, that is, the answer, the proof in (3.14) is of no use for us, so we may safely delete it. However, in the analogous representations of (3.10) and (3.11), we must keep enough information for reconstructing the atoms whose provability is not guaranteed at each point. This leads us to representing the process (3.10)-(3.12) by the formal expression

$$\dfrac{\dfrac{A_0\ A_1 \ \vdash \ E_2}{A_0 \ \vdash \ E_1}}{\vdash \ E_0}$$

where, among other things, we suppose $\sigma^+_{E_2} = \theta_2 \in unf(B,A_2)$, which makes $A_0\ A_1 \vdash E_2$ a sufficiently accurate representation of (3.10). In this expression, A_i represents the atomic query $A_i\sigma^+_{E_2}$, $i = 0, 1$. Recalling that $id = \sigma^+_\emptyset$, our representation principles lead us to representing B by $B \vdash \emptyset$.

We also need the formal rule allowing us to mimic the process described before, but now this is (almost) self-evident: it must be something like

$$\dfrac{A_0\ A_1 \vdash A_2 \qquad B \vdash \emptyset}{A_0\ A_1 \vdash E_2}$$

where, as we remarked before, $\sigma^+_{E_2} \in unf(A_2, B\sigma^+_\emptyset)$. The experienced reader has certainly recognized this rule as the SLD-Resolution Rule, modulo some technicalities.

3.2.2 The Resolution Rules

Definition 3.16 *We will call* goal *any expression of the form $\Delta \vdash E$, where Δ is a finite sequence of atoms and/or equations, and E is as in Definition 2.37. The equality between goals is defined in the obvious way.*

The previous discussion suggests us to consider the goal $A_0 \ldots A_n \vdash E$ as an encoded representation of some MP-proof in

$$P^\Sigma \cup \{ A_i\sigma_E^+ \mid 0 \leq i \leq n \},$$

for some P, proof that we are trying to extend into a proof in P^Σ. We will see that it may also be seen as an encoded representation of the sequent $A_0\sigma_E^+ \ldots A_n\sigma_E^+ \vdash$.

Definition 3.17 (The GSLD-Resolution Rule)

$$\frac{B_0 \ldots B_{k-1} \vdash B_k \qquad \Delta\ A \vdash E_0}{\Delta\ B_0 \ldots B_{k-1} \vdash E_1} \tag{3.15}$$

with the following provisos:

- $B_0 \ldots B_{k-1} \vdash B_k$ *shares no variable with* $\Delta\ A \vdash E_0$.
- $E_0 \subseteq E_1$.
- $\sigma_{E_1}^+ \in unf(B_k, A\sigma_{E_0}^+)$.

The restrictions in the definition merit some comments. The obvious idea behind the calculus is that given P, and a query $Q = A_0 \wedge \ldots \wedge A_n$, we will try to transform the goal $A_0 \ldots A_n \vdash \emptyset$ into a goal $\vdash E$ such that σ_E^+ is an answer for Q in P. The second proviso in the definition ensures that if a variable of Q is bound during the process, we will be able to find its binding in the resulting environment E. Also, by Proposition 2.61, $\sigma_{E_1}^+ = \sigma_{E_0}^+\sigma_{E_1}^+$, which implies $\sigma_{E_0}^+ \leq \sigma_{E_1}^+$. We see then that the formalism computes more and more instantiated substitutions. This is precisely what our discussion in the introduction suggested us to do. Note also that the equality $\sigma_{E_1}^+ = \sigma_{E_0}^+\sigma_{E_1}^+$ implies

$$\sigma_{E_1}^+ \in unf(B_k, A\sigma_{E_0}^+) \quad \text{if and only if} \quad \sigma_{E_1}^+ \in unf(B_k, A) \tag{3.16}$$

and also

$$\sigma_{E_1}^+ \in unf(B_k, A\sigma_{E_0}^+) \quad \text{if and only if} \quad \sigma_{E_1}^+ \in unf(B_k\sigma_{E_0}^+, A\sigma_{E_0}^+). \tag{3.17}$$

This shows that the third condition in Definition 3.17 could have been expressed as on the right of (3.16), or of (3.17). It seems to us, however, that the most natural statement of this condition is precisely that of Definition 3.17, which clearly suggests that the rule application is a first step towards the construction of a useful pair for $A\sigma_{E_0}^+$ based on some instance of $B_0 \ldots B_{k-1} \vdash B_k$.

The previous remarks also show that the rule application (3.15) may be seen as an encoded representation of the following deduction using **CUT** and the Substitution Rule.

$$\dfrac{\dfrac{B_0 \ldots B_{k-1} \vdash B_k}{B_0\sigma_{E_1}^+ \ldots B_{k-1}\sigma_{E_1}^+ \vdash B_k\sigma_{E_1}^+} \qquad \dfrac{\Delta\sigma_{E_0}^+\ A\sigma_{E_0}^+ \vdash}{\Delta\sigma_{E_1}^+\ A\sigma_{E_1}^+ \vdash}}{\dfrac{B_0\sigma_{E_1}^+ \ldots B_{k-1}\sigma_{E_1}^+\ \Delta\sigma_{E_1}^+ \vdash}{\Delta\sigma_{E_1}^+\ B_0\sigma_{E_1}^+ \ldots B_{k-1}\sigma_{E_1}^+ \vdash}}$$

where the double line represents an application of the full exchange rule. However, the SLD-Rule intends to be an answer-computing formalism. This is why we store the computed substitution in the goal's environment.

The GSLD-Rule is the General SLD-Resolution Rule. According to our definition, in each rule instance, the clause and the goal in the antecedent share no variable, while the environment in the conclusion encodes an *arbitrary* unifier of the clause head and the rightmost goal atom. We define the UGSLD (Unrestricted GSLD) Rule to be like the GSLD-Rule, except that now we allow that the clause and the goal in the antecedent share variables. We will see that the UGSLD Rule is sound and complete.

The SLD-Rule must be defined in such a way that the environment in the conclusion of each rule instance encode a principal unifier of the clause head and the rightmost goal atom. More precisely, (3.15) is an instance of SLD iff $B_k \approx A$, and

$$\frac{\begin{array}{c} \propto(B_k, A) \vdash E_0 \\ \vdots \end{array}}{\vdash E_1}$$

is a refutation in U^X, where we recall that the notation $B_k \approx A$ means that the corresponding atoms are compatible, that is, their predicates are equal. With this definition, we are allowed to use an arbitrary unification algorithm at each SLD Resolution step. But we may want (or need) to impose particular restrictions on it. For instance, we may choose a fixed deterministic unification algorithm and restrict ourselves to using this fixed algorithm in all applications of the SLD-Rule. We will see that even with this restriction the SLD-Rule is a sufficiently powerful answer computing formalism. In our discussion of SLD-Resolution, we will adopt the following

Definition 3.18 *A* unification algorithm *is a ternary computable function* $\mu(A, B, E)$ *such that if* $A \approx B$ *and* $\propto(A, B) \cup E$ *is unifiable, then* $F = \mu(A, B, E)$ *is an environment such that*

$$\frac{\begin{array}{c} \propto(B, A) \vdash E \\ \vdots \end{array}}{\vdash F}$$

is a refutation in some system U^X. *Otherwise,* $\mu(A, B, E) = \perp$, *where* $\perp$ *is some object different from all environments.*

The SLD-Resolution Rule *determinded by* μ is defined as in Definition 3.17 replacing the last two provisos with

- $E_1 = \mu(B_k, A, E_0) \neq \perp$.

Hence, strictly speaking, with our definitions, there exist multiple SLD-Rules. For instance,

$$\frac{\vdash p(x) \qquad p(y) \vdash}{\vdash x \asymp y} \qquad\qquad \frac{\vdash p(x) \qquad p(y) \vdash}{\vdash y \asymp x}$$

are instances of different SLD-Rules. For simplicity, however, the expression 'the SLD-Rule' will mean 'an arbitrary SLD-Rule'.

In our definition of the SLD-Rule, the role of the unification algorithm is to hide the internals of the unification process, but, since a unification algorithm is a relatively complex operation, our rules are 'high-level' rules. Another possibility is to define a 'low-level' system, capable of internally and explicitly representing the unification process. The system SLDS is the low level SLD-Resolution system consisting of the rules of U, supplemented with

The Atom Decomposition Rule (ad)

$$\frac{\Gamma \vdash C \qquad \Delta\ A \vdash E}{\Delta\ \Gamma\ \propto(C, A) \vdash E}$$

provided $C \approx A$, and $\Gamma \vdash C$ share no variable with $\Delta\ A \vdash E$.

Consider a fixed SLD-Rule. Since U is a complete unification system, any instance (3.15) of SLD may be transformed into a proof

$$\frac{B_0 \ldots B_{k-1} \vdash B_k \qquad \Delta\ A \vdash E_0}{\lambda \left\{ \begin{array}{c} \Delta\ B_0 \ldots B_{k-1} \propto(B_k, A) \vdash E_0 \\ \vdots \\ \Delta\ B_0 \ldots B_{k-1} \vdash E_1 \end{array} \right.} \tag{3.18}$$

in SLDS, where λ is a proof in U. The converse is not true, that is, there exist SLDS proofs (3.18) such that (3.15) is not an instance of the chosen SLD-Rule. However, under these conditions, (3.15) is an instance of GSLD.

Examples

1. This is an instance of SLD.

$$\frac{\vdash p(x) \qquad p(a) \vdash}{\vdash x \asymp a}$$

2. This is an instance of GSLD, but not of SLD.

$$\frac{\vdash p(x) \qquad p(a) \vdash}{\vdash x \asymp a\ y \asymp b}$$

3. This is an instance of UGSLD, but not of GSLD.

$$\frac{\vdash p(x) \qquad p(a) \vdash x \asymp a}{\vdash x \asymp a}$$

In what follows, we will need to deal with the different systems we defined above. In order to avoid confusion, it will be convenient to identify them clearly once and for all. System U is the unification system of Table 2.1. The UGSLD (respectively, the GSLD, SLD) system is the system consisting only of the corresponding rule, while system SLDS consists of the unification rules plus the Atom Decomposition Rule. If $\mathcal{S}$ is one of these systems, and X an exchange rule, we already know how to define $\mathcal{S}^X$.

We will call *SLD-theory* any set $\mathcal{A} \cup \mathcal{G}$, where $\mathcal{A}$ is a set of clauses, and $\mathcal{G}$ is a set of goals. Given such a theory, it is easily seen that once we started a proof in one of the systems defined above with a goal $\Delta \vdash E$ as axiom, all other goals appearing in the proof are provable in $\mathcal{A} \cup \{ \Delta \vdash E \}$[2] in the same system.

Given a program P and a query $Q = A_0 \wedge \ldots \wedge A_n$, we may think that for computing all answers we need, it suffices to consider the proofs in P, $A_0 \ldots A_n \vdash \emptyset$. In general, however, this is not the case. Consider the program $P = \{ p(x) \}$ and the query $p(f(x))$. Clearly, id is an answer for $p(f(x))$ in P, but we will find no UGSLD instance of the form

$$\frac{\vdash p(x) \qquad p(f(x)) \vdash}{?}$$

simply because $p(x)$ and $p(f(x))$ are not unifiable. However, taking $\vdash p(y) \in P^{\equiv}$, we get

$$\frac{\vdash p(y) \qquad p(f(x)) \vdash}{\vdash y \asymp f(x)}$$

and $\sigma^+_{\{y \asymp f(x)\}} = id$ on $var(p(f(x)))$. Given a program P and a goal $\Delta\ A \vdash E_0$, we must take a clause $B_0 \ldots B_{k-1} \vdash B_k \in P^{\equiv}$ such that, at least, $var(B_k)$ and $var(A\sigma^+_{E_0})$ are disjoint, but things are more tricky than this.

Consider the program $\{ p(x) \}$ and the goal $p(f(y)) \vdash x \asymp a$. For applying one of the above rules in the form

$$\frac{\vdash p(x) \qquad p(f(y)) \vdash x \asymp a}{\vdash E_1}$$

we need that $p(x)$ and $p(f(y))\sigma^+_{\{x \asymp a\}} = p(f(y))[x/a] = p(f(y))$ be unifiable, which is the case. However, we will find no environment E_1 such that $\{ x \asymp a \} \subseteq E_1$, and

$$\sigma^+_{E_1} \in unf(p(x), p(f(y))[x/a]). \tag{3.19}$$

This is because, as was remarked in (3.17), (3.19) is equivalent to

$$\sigma^+_{E_1} \in unf(p(x)[x/a], p(f(y))[x/a]) = \emptyset.$$

The obvious problem here is that we took a clause $\vdash p(x)$ with a variable bound in the goal's environment $\{ x \asymp a \}$. In other cases, however, violating this condition does not hurt,

[2] from now on, written $\mathcal{A}, \Delta \vdash E$.

as the following trivial example shows.

$$\frac{\vdash p(x) \qquad p(x) \vdash x \asymp a}{\vdash x \asymp a}$$

3.2.3 Soundness

The UGSLD^X system is *sound* if any UGSLD^X-computed answer is an answer. This is precisely stated in Corollary 3.20 below.

Proposition 3.19 *If*

$$\frac{\begin{array}{c} A_0 \ldots A_n \vdash E_0 \\ \vdots \end{array}}{B_0 \ldots B_{k-1} \vdash E_1} \tag{3.20}$$

is a proof in UGSLD^X, then

$$P \models (\bigwedge_{i=0}^{k-1} B_i \rightarrow \bigwedge_{i=0}^{n} A_i)\sigma_{E_1}^{+}.$$

In particular, if (3.20) is a refutation,

$$P \models \bigwedge_{i=0}^{n} A_i \sigma_{E_1}^{+}.$$

Proof By induction on the complexity of the given proof. If the given proof is of complexity 0, the result is immediate. Suppose we have a proof

$$\frac{\begin{array}{cc} & A_0 \ldots A_n \vdash E_0 \\ & \vdots \\ D_0 \ldots D_{r-1} \vdash D_r & C_0 \ldots C_k \vdash E \end{array}}{C_{j_0} \ldots C_{j_{k-1}} D_0 \ldots D_{r-1} \vdash E_1}$$

where $C_{j_0} \ldots C_{j_k}$ is a permutation of $C_0, \ldots, C_k$, and

$$D_r \sigma_{E_1}^{+} = C_{j_k} \sigma_{E_1}^{+}. \tag{3.21}$$

We want to prove

$$P \models ((\bigwedge_{i=0}^{k-1} C_{j_i}) \wedge (\bigwedge_{i=0}^{r-1} D_i) \rightarrow \bigwedge_{i=0}^{n} A_i)\sigma_{E_1}^{+},$$

under the inductive hypothesis

$$P \models (\bigwedge_{i=0}^{k} C_i \rightarrow \bigwedge_{i=0}^{n} A_i)\sigma_{E}^{+}. \tag{3.22}$$

Note that $\sigma_E^{+} \leq \sigma_{E_1}^{+}$, hence from (3.22)

$$P \models (\bigwedge_{i=0}^{k} C_i \rightarrow \bigwedge_{i=0}^{n} A_i)\sigma_{E_1}^{+}. \tag{3.23}$$

Take then a model $\mathcal{I}$ of P, and a valuation ν such that

$$\mathcal{I}, \nu \models (\bigwedge_{i=0}^{k-1} C_{j_i}\sigma_{E_1}^+) \wedge (\bigwedge_{i=0}^{r-1} D_i\sigma_{E_1}^+). \tag{3.24}$$

Since $D_0 \ldots D_r$ is a clause of $P^{\equiv}$, we have

$$P \models (D_0 \ldots D_r)\sigma_{E_1}^+.$$

According to (3.24), $\mathcal{I}, \nu \models \bigwedge_{i=0}^{r-1} D_i\sigma_{E_1}^+$, and by hypothesis $\mathcal{I}$ is a model of P. Hence $\mathcal{I}, \nu \models D_r\sigma_{E_1}^+$. But by (3.21), $D_r\sigma_{E_1}^+ = C_{j_k}\sigma_{E_1}^+$, which means that (3.24) implies

$$\mathcal{I}, \nu \models \bigwedge_{i=0}^{k} C_{j_i}\sigma_{E_1}^+.$$

From this and (3.23), $\mathcal{I}, \nu \models \bigwedge_{i=0}^{n} A_i\sigma_{E_1}^+$. □

Corollary 3.20 *If (3.20) is a refutation in UGSLD^X, then $\sigma_{E_1}^+$ is an answer for*

$$(A_0 \wedge \ldots \wedge A_n)\sigma_{E_0}^+$$

in P. In particular, if $E_0 = \emptyset$, $\sigma_{E_1}^+$ is an answer for $A_0 \wedge \ldots \wedge A_n$ in P. □

3.2.4 Structural Results

In this section we will consider structural properties of the deductions in the systems considered here. The reader will find no difficulty in verifying these results.

Our first observation is that if we have a proof

$$\cfrac{\cfrac{\cfrac{\Delta_0 \vdash E_0}{\vdots}}{\Delta_1 \vdash E_1}}{\vdots}$$
$$\Delta_2 \vdash E_2$$

then $E_0 \subseteq E_1 \subseteq E_2$.

Let us say that the *length* of the goal $\Delta \vdash E$ and of the clause $\Delta \vdash D$ is $|\Delta|$. The following statement says that if π is a proof in UGSLD, there exists a well-defined relation between the complexity of π, the length of the conclusion, the length of the goal axiom, and the lengths of the clauses appearing in the proof.

Proposition 3.21 *Consider a proof*

$$\cfrac{\cfrac{\cfrac{\Pi_{n-1} \vdash B_{n-1} \qquad \cfrac{\cfrac{\Pi_0 \vdash B_0 \qquad \Delta_0 \vdash E_0}{\Delta_1 \vdash E_1}}{\vdots}}{\Delta_{n-1} \vdash E_{n-1}}}{\Delta_n \vdash E_n}}{}$$

in UGSLD of complexity n. Then $n + |\Delta_n| = |\Delta_0| + \Sigma_{i=0}^{n-1} |\Pi_i|$. □

In particular, if the proof is a refutation, $|\Delta_n| = 0$, and we get

$$n = |\Delta_0| + \Sigma_{i=0}^{n-1} |\Pi_i| \geq |\Delta_0|.$$

Corollary 3.22 *The complexity of any refutation of* $\Delta_0 \vdash E_0$ *is greater than or equal to* $|\Delta_0|$. □

An application of UGSLD deletes at most one atom from the current goal. Therefore, it is clear that if $|\Delta| > 0$, a refutation

$$\frac{\begin{array}{c}\Delta \wedge A \vdash E_0 \\ \vdots \end{array}}{\vdash E_1} \tag{3.25}$$

in UGSLD must contain at least one interior goal occurrence of the form $\Delta \vdash E_2$. But an application of UGSLD may also introduce new atom occurrences, hence (3.25) may contain multiple goal occurrences of this form. A simple structural condition uniquely determines the 'first' of such occurrences. Let us call *prefix* of $\Delta \vdash E$ any prefix of Δ, the *proper* prefixes being the prefixes other than Δ.

Proposition 3.23 *Suppose that* $|\Delta| > 0$, *and that (3.25) is a refutation of complexity* m *in UGSLD (respectively, GSLD, SLD). Then there exist two uniquely determined subproofs* π_1, π_2 *of (3.25)*

$$\pi_1 \left\{ \frac{\begin{array}{c}\Delta \wedge A \vdash E_0 \\ \vdots \end{array}}{\Delta \vdash E_2} \right. \qquad \pi_2 \left\{ \frac{\begin{array}{c}\Delta \vdash E_2 \\ \vdots \end{array}}{\vdash E_1} \right.$$

of positive complexities m_1 *and* m_2, *such that* $m = m_1 + m_2$, *and* Δ *is a proper prefix of all the interior goal occurrences of* π_1. □

3.2.5 Operations on Proofs

Proposition 3.24 *If in an instance of UGSLD (respectively, GSLD)*

$$\frac{\Gamma \vdash C \qquad \Delta\ A \vdash E_0}{\Delta\ \Gamma \vdash E_1} \tag{3.26}$$

we replace E_0 *with an environment* $F_0 \subseteq E_0$, *we obtain another instance of UGSLD (respectively, GSLD).* □

Note, however, that the statement does not hold for SLD. If (3.26) is an instance of SLD, we have

$$\sigma_{E_1}^{+} \in pru(\propto(C, A) \cup E_0). \tag{3.27}$$

If the replacement of E_0 with F_0 yields another instance of SLD, we also have

$$\sigma_{E_1}^{+} \in pru(\propto(C, A) \cup F_0). \tag{3.28}$$

But two sets of equations with a common principal unifier have the same unifiers, hence (3.27), and (3.28) imply

$$unf(\propto(C, A) \cup F_0) = unf(\propto(C, A) \cup E_0). \tag{3.29}$$

Therefore, if the above proposition holds for SLD, (3.29) holds for C and A arbitrary compatible atoms, E_0 an arbitrary environment, and F_0 an arbitrary subset of E_0. It is easily seen, however, that this last statement is not true, hence the proposition fails for SLD. As a counter-example, consider the following instance of SLD.

$$\frac{\vdash p(a) \qquad p(a) \vdash x \asymp a}{\vdash x \asymp a}$$

Replacing the environment in the antecedent by the empty environment, we get

$$\frac{\vdash p(a) \qquad p(a) \vdash}{\vdash x \asymp a}$$

which is an instance of GSLD. but not of SLD.

Proposition 3.25 *Suppose that*

$$\frac{\Gamma \vdash C \qquad \Delta\ A \vdash E_0}{\Delta\ \Gamma \vdash E_1} \tag{3.30}$$

is an instance of UGSLD (respectively, GSLD). Then for all environments $E_2 \supseteq E_1$, the replacement of E_1 by E_2 in (3.30) yields an instance of UGSLD (respectively, GSLD). □

It is clear that this statement fails for SLD. The next few statements deal with the deletion and addition of prefixes in all the goal occurrences of a proof.

Proposition 3.26 *If*

$$\frac{\Gamma \vdash C \qquad \Delta\ \Lambda\ A \vdash E_0}{\Delta\ \Lambda\ \Gamma \vdash E_1} \tag{3.31}$$

is an instance of UGSLD (respectively, GSLD, SLD), then

$$\frac{\Gamma \vdash C \qquad \Lambda\ A \vdash E_0}{\Lambda\ \Gamma \vdash E_1} \tag{3.32}$$

is an instance of UGSLD (respectively, GSLD, SLD). Conversely, if (3.32) is an instance of UGSLD (respectively, GSLD, SLD), and Δ shares no variables with $\Gamma \vdash C$, then (3.31) is an instance of UGSLD (respectively, GSLD, SLD). If Δ shares variables with $\Gamma \vdash C$, then (3.31) is an instance of UGSLD, but not of GSLD. □

This statement may be extended to arbitrary proofs in the obvious way. This is left to the reader.

Corollary 3.27 *If* $|\Delta| > 0$, *and there exists a refutation of complexity* m *in UGSLD (respectively, GSLD, SLD)*

$$\dfrac{\begin{array}{c}\Delta\ A \vdash E_0 \\ \vdots \end{array}}{\vdash E_1}$$

then there exists a refutation

$$\dfrac{\begin{array}{c}A \vdash E_0 \\ \vdots \end{array}}{\vdash E_2}$$

in UGSLD (respectively, GSLD, SLD) of complexity $0 < m_1 < m$, *and* $E_0 \subseteq E_2 \subseteq E_1$. □

This statement easily generalizes to the following

Corollary 3.28 *If we are given a refutation* π *in UGSLD (respectively, GSLD, SLD)*

$$\pi \left\{ \dfrac{\begin{array}{c}A_0 \ldots A_n \vdash E_0 \\ \vdots \end{array}}{\vdash E_1} \right. \tag{3.33}$$

of complexity m, *then for each* $i \leq n$, *we can construct a refutation* π_i

$$\dfrac{\begin{array}{c}A_i \vdash F_{i+1} \\ \vdots \end{array}}{\vdash F_i} \tag{3.34}$$

in UGSLD (respectively, GSLD, SLD) of complexity m_i, *with* $m = \Sigma_{i=0}^{n} m_i$,

$$E_0 = F_{n+1} \subseteq F_n \subseteq \ldots \subseteq F_1 \subseteq F_0 = E_1,$$

and $var(\pi) = \bigcup_{i=0}^{n} var(\pi_i)$. □

A restricted form of converse is also true.

Corollary 3.29 *If for each* $i \leq n$ *we are given the refutation* π_i *of (3.34) of complexity* m_i, *then we can construct a refutation* π *in UGSLD of the form (3.33) of complexity* $\Sigma_{i=0}^{n} m_i$, *with* $E_0 = F_{n+1}$, $E_1 = F_0$, *and* $var(\pi) = \bigcup_{i=0}^{n} var(\pi_i)$.

Proof For each $i \leq n$, by Proposition 3.26, from (3.34) we can construct a proof

$$\dfrac{\begin{array}{c}A_0 \ldots A_i \vdash F_{i+1} \\ \vdots \end{array}}{A_0 \ldots A_{i-1} \vdash F_i}$$

in UGSLD. Next, we construct (3.33) in the obvious way. □

Note, however, that we can only ensure that the resulting proof is UGSLD, even if the given proofs were SLD. The following example clearly shows why the 'SLDness' may be lost. Applying the method described above to the two SLD-proofs

$$\frac{\vdash p(x) \qquad p(y) \vdash}{\vdash y \asymp x} \qquad\qquad \frac{\vdash q(a) \qquad q(x) \vdash y \asymp x}{\vdash x \asymp a \quad y \asymp x}$$

we get a proof in UGSLD, which is not GSLD.

$$\dfrac{\vdash q(a) \qquad \dfrac{\vdash p(x) \qquad\qquad q(x) \quad p(y) \vdash}{q(x) \vdash y \asymp x}}{\vdash x \asymp a \quad y \asymp x}$$

Definition 3.30 *An instance of UGSLD*

$$\frac{\Gamma \vdash C \qquad \Delta \; A \vdash E_0}{\Delta \; \Gamma \vdash E_1}$$

is said to be elementaty *iff* $E_0 = E_1$. *The elementary proofs and refutations are defined in the obvious way.*

Proposition 3.31 *If* σ_E^+ *unifies all the equations in* Δ, *and*

$$\frac{\begin{array}{c}\Delta \vdash E \\ \vdots \end{array}}{\vdash F}$$

is a refutation in U, then $F = E$.

Proof Since the rules **ee** and **sd** do not modify the environment, it suffices to show that under the current hypotheses, the rules **lb** and **rb** cannot be applied.

Suppose that $t \asymp u$ is the rightmost equation of Δ. By hypothesis, $t\sigma_E^+ = u\sigma_E^+$. If $t\sigma_E^+$ is a variable, then it appears in $u\sigma_E^+ = t\sigma_E^+$, hence $\Phi_2(t, u, E)$ does not hold, and **lb** cannot be applied. The case of **rb** is analogous. □

Corollary 3.32 *If* σ_E^+ *unifies the atoms* C *and* A, *then for any unification algorithm* μ, $E = \mu(C, A, E)$. □

This implies that an elementary instance of GSLD is also an elementary instance of SLD.

Examples

i. An elementary instance of GSLD.

$$\frac{\vdash p(a) \qquad p(x) \vdash x \asymp a}{\vdash x \asymp a}$$

ii. An elementary instance of UGSLD, not of GSLD.

$$\frac{\vdash p(x) \qquad p(x) \vdash x \asymp a}{\vdash x \asymp a}$$

□

The next statement shows how to transform an arbitrary proof into an elementary one.

Proposition 3.33 *Let π be the proof*

$$\frac{\begin{array}{c}\Delta_0 \vdash E_0 \\ \vdots \end{array}}{\Delta_1 \vdash E_1}$$

in UGSLD. If we replace all the environment occurrences in π with an environment E_2 such that $\sigma^+_{E_1} \leq \sigma^+_{E_2}$, then we obtain an elementary proof λ in UGSLD (obviously of the same complexity). Moreover, if $E_2 \supseteq E_1$, we also have $var(\lambda) = var(\pi) \cup var(E_2)$. □

The following result follows easily from Corollary 3.28 and Corollary 3.29.

Corollary 3.34 *$A_0 \ldots A_n \vdash E$ has an elementary refutation π of complexity m in UGSLD iff for each $i \leq n$, there exists an elementary refutation π_i of $A_i \vdash E$ of complexity m_i in UGSLD, with $m = \Sigma_{i=0}^n m_i$, and $var(\pi) = \bigcup_{i=0}^n var(\pi_i)$.* □

This readily entails the following corollary, which, intuitively speaking, says that if we are trying to construct an elementary refutation in UGSLD, then the choice of the atom to reduce is irrelevant.

Corollary 3.35 *If $\Delta \vdash E$ has an elementary refutation π of complexity m in UGSLD, then for any permutation Λ of Δ, $\Lambda \vdash E$ has an elementary refutation λ of complexity m in UGSLD, with $var(\lambda) = var(\pi)$.* □

3.2.6 Equivalence of the Resolution Rules

Trivially, a proof in SLD (respectively, GSLD), is also a proof in GSLD (respectively, UGSLD). The converse does not hold, but if we restrict ourselves to refutations, then a form of converse holds. Our first task will be to show that for all refutations π in UGSLD, there exists a refutation π^* in GSLD computing essentially the same answer. To this end, suppose we have fixed a theory

$$P^{\equiv}, \Delta_0 \vdash E_0. \tag{3.35}$$

We will need the following definition.

Definition 3.36 *A* renaming system *is a family $\mathcal{R} = \{ \phi_j \mid j \in \mathcal{N} \}$ of variable renamings such that, writing $\mathcal{V}_j = \mathcal{V}\phi_j$, the following conditions hold.*

i. $\{ \mathcal{V}_j \mid j \in \mathcal{N} \}$ *is a partition*[3] *of* $\mathcal{V}$.

ii. The set $nuc(\mathcal{R}) = \{ x \in \mathcal{V}_0 \mid x\phi_0 = x \}$ *is denumerable.*

It is not difficult to see that given any set V of variables such that $\mathcal{V} \setminus V$ is infinite, there exists a renaming system $\mathcal{R}$ such that $V \subseteq nuc(\mathcal{R})$.

Once we fixed the theory (3.35), let us choose a renaming system $\mathcal{R}$ such that

$$var(\Delta_0) \cup var(E_0) \subseteq nuc(\mathcal{R}). \tag{3.36}$$

If π is a proof of $A_0 \ldots A_n \vdash E_1$ in (3.35), for each atom *occurrence* A_i in the conclusion of π, we will be interested in determining which rule application introduced this atom occurrence. This will be formalized through the notion of *depth* of each atom occurrence in the conclusion of π.

Definition 3.37 *All atom occurrences in* Δ_0 *are of depth 0 in* $\Delta_0 \vdash E_0$. *Suppose now that* π *is the proof*

$$\frac{B_0 \ldots B_{k-1} \vdash B_k \qquad \pi_2 \left\{ \begin{array}{c} \vdots \\ \Delta\ A \vdash E_2 \end{array} \right.}{\Delta\ B_0 \ldots B_{k-1} \vdash E_1} \tag{3.37}$$

of complexity m. *Each atom occurring in* Δ *in the conclusion of* π *is, in* π, *of the same depth as the corresponding occurrence in the conclusion of* π_2. *All the* B_i*'s are of depth* m *in* π.

The following notations will be useful in the proof of the next proposition. If π is the proof (3.37), we will denote by $V_0(\pi)$ the set $var(\Delta_0)$. For $0 < j \leq m$, $V_j(\pi)$ will denote the set of variables of the clause used in the j-the rule application in π. If π is the proof (3.37), we define

$$e(\pi) = \{ z\phi_j \asymp z \mid 0 < j \leq m, \text{ and } z \in V_j(\pi) \},$$

and if π reduces to $\Delta_0 \vdash E_0$, $e(\pi) = \emptyset$.

Several remarks are in order. If the atom occurrence A in the conclusion of π is of depth j, then $var(A) \subseteq V_j(\pi)$. Also, referring to (3.37), $V_j(\pi) = V_j(\pi_2)$, for all $j < m$.

Consider now a *positive* $j_0 \leq m$, and $z_0 \in V_{j_0}(\pi)$. Since $\mathcal{R}$ is a renaming system, $z_0\phi_{j_0} \in \mathcal{V}_{j_0}$, hence $z_0\phi_{j_0} \notin \mathcal{V}_0$. Also, if the pair (j_1, z_1) is different from (j_0, z_0), then $z_1\phi_{j_1} \neq z_0\phi_{j_0}$. This implies that there exists a substitution θ such that

$$\forall j \forall z,\ 0 < j \leq m, \text{ and } z \in V_j(\pi) \text{ implies } z\phi_j\theta = z. \tag{3.38}$$

Finally, note that if π is the proof (3.37), then

$$e(\pi_2) \subseteq e(\pi). \tag{3.39}$$

[3]Note that since ϕ_j is a variable renaming, $\mathcal{V}_j$ is a denumerable set of variables.

Proposition 3.38 *If π is a UGSLD-proof of $C_0 \dots C_n \vdash E_1$ of complexity m in (3.35), and $var(\pi) \subseteq nuc(\mathcal{R})$, then there exists a GSLD-proof π^* of $D_0 \dots D_n \vdash F_1$ in (3.35) such that*

i. $F_1 = E_1 \cup e(\pi)$.

ii. $\sigma^+_{F_1} = \sigma^+_{E_1}$ *on* $\mathcal{V}_0$.

iii. For each $i \leq n$, *if* C_i *is of depth* j *in* π, *then* $D_i = C_i\phi_j$.

iv. π^* *is of complexity* m, *and* $var(\pi^*) \subseteq \bigcup_{j=0}^{m} \mathcal{V}_j$.

Proof If π is of complexity 0, it suffices to take $\pi^* = \pi$. Suppose now that π is (3.37), and that (the rightmost) A in the conclusion of π_2 is of depth j in π_2. By the I.H., there exists a GSLD-proof π_2^* of complexity $m-1$

$$\frac{\begin{array}{c}\Delta_0 \vdash E_0 \\ \vdots \\ \Pi\, A\phi_j \vdash F_2\end{array}}{}$$

with $F_2 = E_2 \cup e(\pi_2)$, and $var(\pi_2^*) \subseteq \bigcup_{j=0}^{m-1} \mathcal{V}_j$. Defining F_1 as in i above, and recalling (3.39), we readily get $F_2 \subseteq F_1$.

Consider now the set $V = \{\, z\phi_j \mid 0 < j \leq m,\ z \in V_j(\pi) \,\}$, and take a θ satisfying (3.38). Thus, $e(\pi) = \{\, x \asymp x\theta \mid x \in V \,\}$. Note that by the choice of $\mathcal{R}$, and the hypotheses on π,

$$var(V\theta) \cup var(E_1) \subseteq var(\pi) \subseteq nuc(\mathcal{R}) \subseteq \mathcal{V}_0,$$

and by a previous remark, if $x \in V$, then $x \notin \mathcal{V}_0$. By Corollary 2.86, F_1 is an environment, and

$$\sigma^+_{F_1} = \theta\sigma^+_{E_1} \downarrow V \ +\ \sigma^+_{E_1}. \tag{3.40}$$

Since $V \cap \mathcal{V}_0 = \emptyset$, $\sigma^+_{F_1} = \sigma^+_{E_1}$ on $\mathcal{V}_0$, which is *ii.* (3.38), and (3.40) imply that for all j, $0 < j \leq m$, and $z \in V_j(\pi)$,

$$z\phi_j\sigma^+_{F_1} = z\sigma^+_{E_1}. \tag{3.41}$$

We have chosen the renaming system $\mathcal{R}$ such that

$$V_0(\pi) = var(\Delta_0) \subseteq nuc(\mathcal{R}) \subseteq \mathcal{V}_0.$$

Consequently, since $\phi_0 = id$ on $nuc(\mathcal{R})$, (3.41) also holds for $j = 0$, and $z \in V_0(\pi)$.

But by (3.37), $\sigma^+_{E_1}$ unifies B_k and A, $var(B_k) \subseteq V_m(\pi)$, and since we supposed that A is of depth j in π_2, $var(A) \subseteq V_j(\pi)$. Thus, from (3.41), we get

$$B_k\phi_m\sigma^+_{F_1} = B_k\sigma^+_{E_1} = A\sigma^+_{E_1} = A\phi_j\sigma^+_{F_1},$$

Consequently,

$$\frac{B_0\phi_m \dots B_{k-1}\phi_m \vdash B_k\phi_m \qquad \pi_2^*\left\{\begin{array}{c}\vdots \\ \Pi\, A\phi_j \vdash F_2\end{array}\right.}{\Pi\, B_0\phi_m \dots B_{k-1}\phi_m \vdash F_1}$$

is a proof π^* as demanded. □

Corollary 3.39 *If π is a UGSLD-refutation in (3.35) of conclusion $\vdash E_1$ and complexity m, then for any finite set V of variables such that $var(\pi) \subseteq V$, there exists a GSLD-refutation in (3.35) of conclusion $\vdash F_1$ and complexity m, with $\sigma^+_{F_1} = \sigma^+_{E_1}$ on V.*

Proof Choose a renaming system $\mathcal{R}$ such that $V \subseteq nuc(\mathcal{R})$ and apply the proposition. We get a GSLD-refutation of conclusion $\vdash F_1$, and $\sigma^+_{F_1} = \sigma^+_{E_1}$ on $\mathcal{V}_0 \supseteq nuc(\mathcal{R}) \supseteq V$. □

We may then safely say that any UGSLD-computable answer is also GSLD-computable. This is another indication of the soundness of UGSLD. Our next step will be to show that any GSLD-computable answer is subsumed by an SLD-computable answer.

Proposition 3.40 *Suppose that*

$$\frac{\Gamma_0 \vdash C_0 \qquad \Delta_0\ A_0 \vdash E_0}{\Delta_0\ \Gamma_0 \vdash F_0}$$

is an instance of GSLD, that $\Delta_1\ A_1 \vdash E_1$ subsumes $\Delta_0\ A_0 \vdash E_0$ on V, and that $\Gamma_1 \vdash C_1$ is a variant of $\Gamma_0 \vdash C_0$ sharing variables neither with V, nor with $\Delta_1\ A_1 \vdash E_1$. Then there exists an environment F_1 such that

$$\frac{\Gamma_1 \vdash C_1 \qquad \Delta_1\ A_1 \vdash E_1}{\Delta_1\ \Gamma_1 \vdash F_1} \tag{3.42}$$

is an instance of SLD, and if F_1 is any such environment, then $\Delta_1\ \Gamma_1 \vdash F_1$ subsumes $\Delta_0\ \Gamma_0 \vdash F_0$ on V.

Proof By hypothesis, there exists a substitution φ_0 such that

$$\sigma^+_{E_0} = \sigma^+_{E_1}\varphi_0 \text{ on } V, \quad \text{and} \quad (\Delta_0\ A_0)\sigma^+_{E_0} = (\Delta_1\ A_1)\sigma^+_{E_1}\varphi_0, \tag{3.43}$$

and a variable renaming ϕ such that $\Gamma_0\phi \vdash C_0\phi$ is $\Gamma_1 \vdash C_1$. Hence, there exists also a ξ such that $\phi\xi = id$. Let W be the set of variables of $\Gamma_1 \vdash C_1$. Define

$$\theta = \xi\sigma^+_{F_0} \downarrow W\ +\ \sigma^+_{E_1}\varphi_0\sigma^+_{F_0}. \tag{3.44}$$

By hypothesis,

$$C_0 \approx A_0, \quad \text{and} \quad \sigma^+_{F_0} \in unf(\propto(C_0, A_0) \cup E_0). \tag{3.45}$$

The definition of θ, and the hypothesis that $\Gamma_1 \vdash C_1$ shares no variable with either V, or $\Delta_1\ A_1 \vdash E_1$, imply that

$$\theta = \sigma^+_{E_1}\varphi_0\sigma^+_{F_0} \text{ on } V, \text{ and on the variables of } \Delta_1\ A_1 \vdash E_1. \tag{3.46}$$

Hence, $\sigma^+_{E_1} \leq \theta$ on $var(E_1)$, which readily implies that θ is a unifier of E_1. Also, by (3.44), (3.46), and recalling that $E_0 \subseteq F_0$,

$$C_1\theta = (C_0\phi)(\xi\sigma^+_{F_0}) = C_0\sigma^+_{F_0} = A_0\sigma^+_{F_0} = A_0\sigma^+_{E_0}\sigma^+_{F_0} = A_1\sigma^+_{E_1}\varphi_0\sigma^+_{F_0} = A_1\theta.$$

We have then that

$$C_1 \approx A_1, \quad \text{and} \quad \theta \in unf(\propto(C_1, A_1) \cup E_1). \tag{3.47}$$

Consequently, there exists an environment F_1 such that

$$\sigma^+_{F_1} \in pru(\propto(C_1, A_1) \cup E_1).$$

For any such environment, (3.42) is an instance of SLD. Also, by (3.47), there exists a φ_1 such that $\sigma^+_{F_1}\varphi_1 = \theta$. Recalling (3.46), and (3.43), we see that the following equalities hold on V.

$$\sigma^+_{F_1}\varphi_1 = \theta = \sigma^+_{E_1}\varphi_0\sigma^+_{F_0} = \sigma^+_{E_0}\sigma^+_{F_0} = \sigma^+_{F_0}.$$

Also, for the same reasons,

$$\Delta_1\sigma^+_{F_1}\varphi_1 = \Delta_1\theta = \Delta_1\sigma^+_{E_1}\varphi_0\sigma^+_{F_0} = \Delta_0\sigma^+_{E_0}\sigma^+_{F_0} = \Delta_0\sigma^+_{F_0},$$

and

$$\Gamma_1\sigma^+_{F_1}\varphi_1 = \Gamma_1\theta = (\Gamma_0\phi)(\xi\sigma^+_{F_0}) = \Gamma_0\sigma^+_{F_0}.$$

This proves that $\Delta_1\ \Gamma_1 \vdash F_1$ subsumes $\Delta_0\ \Gamma_0 \vdash F_0$ on V. □

Corollary 3.41 *If there exists a GSLD-refutation*

$$\dfrac{\begin{array}{c}\Delta_0 \vdash E_0 \\ \vdots \end{array}}{\vdash F_0} \tag{3.48}$$

of complexity m, then for any goal $\Delta_1 \vdash E_1$ subsuming $\Delta_0 \vdash E_0$ on V, there exists an SLD-refutation

$$\dfrac{\begin{array}{c}\Delta_1 \vdash E_1 \\ \vdots \end{array}}{\vdash F_1}$$

of complexity m, with $\sigma^+_{F_1} \leq \sigma^+_{F_0}$ on V. □

Corollary 3.42 *If there exists a GSLD-refutation (3.48) of complexity m, then for any set V of variables with $var(\Delta_0) \subseteq V$, and any environment E_1 such that $\sigma^+_{E_1} \leq \sigma^+_{E_0}$ on V, there exists an SLD-refutation*

$$\dfrac{\begin{array}{c}\Delta_0 \vdash E_1 \\ \vdots \end{array}}{\vdash F_1}$$

of complexity m, with $\sigma^+_{F_1} \leq \sigma^+_{F_0}$ on $V \supseteq var(\Delta_0)$.

Proof By hypothesis, there exists a φ_0 such that $\sigma^+_{E_1}\varphi_0 = \sigma^+_{E_0}$ on V. Trivially, we also have $\Delta_0\sigma^+_{E_1}\varphi_0 = \Delta_0\sigma^+_{E_0}$. We may then apply the preceding corollary, making $\Delta_1 = \Delta_0$. □

3.2.7 Completeness

Proposition 3.43 *Let $\Delta\ A \vdash E$ be a goal, V a finite set of variables, and suppose that for some $m \in \mathcal{N}$*

$$P^{\Sigma} \vdash^{m}_{MP} A\sigma^{+}_{E}.$$

Then there exists a clause $\beta = B_0 \dots B_k \in P^{\equiv}$, a substitution θ, and an environment F such that

i. $B_k\theta = A\sigma^{+}_{E}$.

ii. All variables in $\beta\theta$ are unbound in E.

iii. β shares no variable with $\Delta\ A \vdash E, \beta\theta, V$.

iv. There exists an MP-proof of $B_k\theta = A\sigma^{+}_{E}$ in P^{Σ} of complexity m based on $\beta\theta$.

v. $E \subseteq F$, *and* $\sigma^{+}_{F} = \theta \downarrow var(\beta) + \sigma^{+}_{E}$.

vi. $$\frac{B_0 \dots B_{k-1} \vdash B_k \qquad \Delta\ A \vdash E}{\Delta\ B_0 \dots B_{k-1} \vdash F}\ GSLD.$$

Proof By hypothesis, there exists a clause $\gamma = C_0 \dots C_k \in P$, a substitution ξ, and an MP-proof

$$\dfrac{\dfrac{\dfrac{\gamma^0\xi \qquad C_0\xi}{\gamma^1\xi} \qquad C_1\xi}{\vdots} \qquad \vdots}{\dfrac{\gamma^{k-1}\xi \qquad C_{k-1}\xi}{\gamma^k\xi}} \tag{3.49}$$

in P^{Σ} of complexity m, with $\gamma^k\xi = C_k\xi = A\sigma^{+}_{E}$. Let $\phi_0 \in \Omega$ such that $\gamma\phi_0$ share no variable with either $\Delta\ A \vdash E$, or V, and define $\beta = \gamma\phi_0$. With this definition, $C_k\xi = B_k\phi_0^{-1}\xi$. None of the variables in $\gamma^k\xi = A\sigma^{+}_{E}$ is in $ker(\sigma^{+}_{E})$, and by the choice of ϕ_0, none of them is in $var(\beta)$. Hence, there exists a variable renaming ϕ_1 into $\mathcal{V} \setminus (ker(\sigma^{+}_{E}) \cup var(\beta))$ such that $\phi_1 = id$ on $var(C_k\xi) = var(B_k\phi_0^{-1}\xi)$. Define $\theta = \phi_0^{-1}\xi\phi_1$. With β and θ so defined, the conditions *i-iii* are satisfied. Concerning *iv*, simply note that applying ϕ_1 to all the formulas in (3.49), we obtain an MP-proof in P^{Σ} of $B_k\theta$ based on $\beta\theta$ of complexity m, as demanded.

Note that by conditions *i-iii*, we have that

$$var(\beta) \cap var(\beta\theta) = \emptyset = var(\beta) \cap var(E),$$

and all the variables in $var(\beta\theta)$ are unbound in E. Hence, by Corollary 2.87, there exists an environment $F \supseteq E$ such that $\sigma^{+}_{F} = \theta \downarrow var(\beta) + \sigma^{+}_{E}$, which is *iv*. But then $B_k\sigma^{+}_{F} = B_k\theta = A\sigma^{+}_{E} = A\sigma^{+}_{F}$, which implies that *vi* is an instance of GSLD. □

Proposition 3.44 *Let P be a set of clauses, $A_0 \ldots A_n$ atoms, V a finite set of variables such that $\bigcup_{i=0}^{n} var(A_i) \subseteq V$, and E_0 an environment such that for all $i \leq n$,*

$$P^{\Sigma} \vdash_{MP}^{m_i} A_i\sigma_{E_0}^{+}.$$

Then there exists a GSLD-refutation

$$\frac{\begin{array}{c} A_0 \ldots A_n \vdash E_0 \\ \vdots \end{array}}{\vdash E_1} \tag{3.50}$$

in $P^{\equiv}$, $A_0 \ldots A_n \vdash E_0$ of complexity $(n+1) + \Sigma_{i=0}^{n} m_i$, and $\sigma_{E_1}^{+} = \sigma_{E_0}^{+}$ on V.

Proof By induction on $(n+1) + \Sigma_{i=0}^{n} m_i$.

Base Case: $(n+1) + \Sigma_{i=0}^{n} m_i = 1$. Hence $n = 0 = m_n$. By hypothesis, $P^{\Sigma} \vdash_{MP}^{0} A_0\sigma_{E_0}^{+}$. By Proposition 3.43, there exists an atom $\beta = B \in P^{\equiv}$, a substitution θ, and an environment F such that

i. $var(\beta) \cap V = \emptyset$.

ii. $\sigma_F^{+} = \theta \downarrow var(\beta) \; + \; \sigma_{E_0}^{+}$,

and

$$\frac{\vdash B \qquad A_0 \vdash E_0}{\vdash F}$$

is an instance of GSLD, hence a GSLD-refutation of complexity $(n+1) + \Sigma_{i=0}^{n} m_i$. Also, by *i* and *ii*, $\sigma_F^{+} = \sigma_{E_0}^{+}$ on V.

Inductive Step: $(n+1) + \Sigma_{i=0}^{n} m_i > 1$. Note that in this case we have either $n > 0$, or $m_n > 0$. By hypothesis, $P^{\Sigma} \vdash_{MP}^{m_n} A_n\sigma_{E_0}^{+}$. By Proposition 3.43, there exists a clause $\beta = B_0 \ldots B_k \in P^{\equiv}$, a substitution θ, and an environment $F \supseteq E_0$ satisfying *i* and *ii* above, and there exists an MP-proof

$$\cfrac{\cfrac{\cfrac{\beta^0\theta \qquad \lambda_0 \left\{ \begin{array}{c} \vdots \\ B_0\theta \end{array} \right.}{\beta^1\theta} \\ \vdots \\ \cfrac{\beta^{k-1}\theta \qquad \lambda_{k-1} \left\{ \begin{array}{c} \vdots \\ B_{k-1}\theta \end{array} \right.}{\beta^k\theta}}} \tag{3.51}$$

of complexity m_n, and such that

$$\frac{B_0 \ldots B_{k-1} \vdash B_k \qquad A_0 \ldots A_n \vdash E_0}{A_0 \ldots A_{n-1}\, B_0 \ldots B_{k-1} \vdash F} \tag{3.52}$$

is an instance of GSLD. By (3.51), if for $j < k$, r_j is the complexity of λ_j, we have

$$m_n = k + \Sigma_{j=0}^{k-1} r_j. \tag{3.53}$$

By (3.51), and *ii*, for all $j < k$, $P^\Sigma \vdash_{MP}^{r_j} B_j\sigma_F^+$. Also, by hypothesis, for all $i < n$, $P^\Sigma \vdash_{MP}^{m_i} A_i\sigma_{E_0}^+$, and since $\sigma_F^+ = \sigma_{E_0}^+\sigma_F^+$, by Proposition 3.14, for all $i < n$, $P^\Sigma \vdash_{MP}^{m_i} A_i\sigma_F^+$. Also, by (3.53),

$$(n+k) + (\Sigma_{i=0}^{n-1} m_i) + (\Sigma_{j=0}^{k-1} r_j) = ((n+1) + \Sigma_{i=0}^{n} m_i) - 1. \tag{3.54}$$

This means that we may apply the I.H. to the conclusion of (3.52), and the set $V \cup var(\beta)$. This gives us a refutation

$$\frac{\begin{array}{c} A_0 \ldots A_{n-1} B_0 \ldots B_{k-1} \vdash F \\ \vdots \end{array}}{\vdash E_1} \tag{3.55}$$

of complexity (3.54), with $\sigma_{E_1}^+ = \sigma_F^+$ on $V \cup var(\beta)$, hence, $\sigma_{E_1}^+ = \sigma_{E_0}^+$ on V. Putting together (3.52), and (3.55), we get a refutation as demanded. □

Theorem 3.45 (Completeness of GSLD) *Let $Q = A_0 \wedge \ldots \wedge A_n$ a query, P a program, and θ an answer for Q in P. Then for any environment E_0 such that $\sigma_{E_0}^+ \leq \theta$ on $var(Q)$, there exists a refutation*

$$\frac{\begin{array}{c} A_0 \ldots A_n \vdash E_0 \\ \vdots \end{array}}{\vdash E_1}$$

in GSLD, with $\sigma_{E_1}^+ \equiv \theta$ on $var(Q)$.

Proof By Proposition 2.86, there exists an environment $F_0 \supseteq E_0$ such that $\sigma_{F_0}^+ \equiv \theta$ on $var(Q)$. Since θ is an answer, so is $\sigma_{F_0}^+$. Hence, by the completeness of SMP, and Proposition 3.14, for all $i \leq n$, $P^\Sigma \vdash_{MP} A_i\sigma_{F_0}^+$. The preceding proposition implies that there exists a GSLD-refutation

$$\frac{\begin{array}{c} A_0 \ldots A_n \vdash F_0 \\ \vdots \end{array}}{\vdash E_1} \tag{3.56}$$

with $\sigma_{E_1}^+ = \sigma_{F_0}^+$ on $var(Q)$. Now it suffices to replace F_0 with E_0 in the axiom of (3.56), which is allowed by Proposition 3.24. □

Corollary 3.46 (Completeness of SLD) *If θ is an answer for Q in P, then for any environment E_0 with $\sigma_{E_0}^+ \leq \theta$ on $var(Q)$, there exists a refutation*

$$\frac{\begin{array}{c} A_0 \ldots A_n \vdash E_0 \\ \vdots \end{array}}{\vdash E_1}$$

in SLD, with $\sigma_{E_1}^+ \leq \theta$ on $var(Q)$.

Proof Immediate from the completeness of GSLD, and Proposition ??. □

3.2.8 Independence of the Selection Rule

As was the case for the unification system U, the introduction of an exchange rule gives us some freedom in the choice of the atom to reduce during a resolution refutation. The next statement shows that for any fixed exchange rule X, any UGSLD-computable answer is subsumed by an SLD^X-computable answer.

Proposition 3.47 *If $\Delta_0 \vdash E_0$ has a refutation π_0 of conclusion $\vdash E_1$ and complexity m in UGSLD, then for any finite set V of variables such that $var(\pi_0) \subseteq V$, and any exchange rule X, $\Delta_0 \vdash E_0$ has a refutation of conclusion $\vdash F_1$ and complexity m in SLD^X, with $\sigma^+_{F_1} \leq \sigma^+_{E_1}$ on V.*

Proof The statement trivially holds for $m = 0$. Suppose now that the given refutation π_0 is of complexity $m > 0$, and that we have already chosen the set V of variables with $var(\pi_0) \subseteq V$. According to Proposition 3.33, we may transform π_0 into an elementary refutation π_1 of $\Delta_0 \vdash E_1$ of complexity m, with $var(\pi_1) = var(\pi_0) \subseteq V$. Now, choose and instance

$$\frac{\Delta_0 \vdash E_0}{\Lambda_0 \vdash E_0}$$

of X. Since $\Delta_0 \vdash E_1$ has the elementary refutation π_1, of complexity m, Corollary 3.35 ensures that there exists an elementary refutation π_2 of $\Lambda_0 \vdash E_1$ of complexity m, with $var(\pi_2) = var(\pi_1) \subseteq V$. By Corollary 3.39, there exists a refutation λ_3 of $\Lambda_0 \vdash E_1$ of complexity m in GSLD with conclusion $\vdash F_3$, with $\sigma^+_{F_3} = \sigma^+_{E_1}$ on V.

Since $\sigma^+_{E_0} \leq \sigma^+_{E_1}$, Corollary 3.42 tells us that there exits an SLD-refutation

$$\dfrac{\Gamma \vdash C \qquad \Lambda_0 \vdash E_0}{\lambda_2 \left\{ \dfrac{\Pi \vdash E_2 \quad \vdots}{\vdash F_2} \right.}$$

of complexity m, with $\sigma^+_{F_2} \leq \sigma^+_{F_3}$ on V.

We may now apply the I.H. of this proposition to the proof λ_2 of complexity $m - 1$, and the set $W = V \cup var(\lambda_2)$. This gives us a refutation λ_1 of $\Pi \vdash E_2$ of complexity $m - 1$ and conclusion $\vdash F_1$ in SLD^X, with $\sigma^+_{F_1} \leq \sigma^+_{F_2}$ on W. But then

$$\dfrac{\Gamma \vdash C \qquad \Delta_0 \vdash E_0}{\lambda_1 \left\{ \dfrac{\Pi \vdash E_2 \quad \vdots}{\vdash F_1} \right.}$$

is a refutation in SLD^X of complexity m, with $\sigma^+_{F_1} \leq \sigma^+_{F_2} \leq \sigma^+_{F_3} = \sigma^+_{E_1}$ on V. □

3.3 Delayed Unification

We already remarked that an application of SLD may be transformed into an application of **ad** followed by a finite number of unification steps. This implies that any maximal proof in SLD may be naturally transformed into a maximal proof in SLDS. In particular, any refutation in SLD may be transformed into a refutation in SLDS.

We assumed that '$\asymp$' is not a program symbol, and according to our definition, an application of **ad** introduces at least one equation. Hence a proof in SLDS cannot contain two consecutive applications of **ad**. Things are different if we introduce an exchange rule.

In this section, let X denote the full exchange rule. We know that if r is a unification rule, any instance

$$\frac{\Delta_0 \vdash E_0}{\Delta_1 \vdash E_1} \tag{3.57}$$

of r^X may be seen as an abbreviation of a proof

$$\cfrac{\cfrac{\Delta_0 \vdash E_0}{\Gamma\ t \asymp u \vdash E_0}}{\Gamma\ \Lambda \vdash E_1} \tag{3.58}$$

with $\Gamma\ \Lambda = \Delta_1$, where the double line represents an application of X, and the single line an application of r. We may find convenient to represent the instance (3.57) in its 'expanded' form (3.58). Note that in (3.58) Λ is a possibly empty list of equations.

An instance occurring in a proof π uniquely determines a subproof of π, namely, that consisting of the selected instance, plus those above it. In the particular case of SLDS^X, the rule applications are totally ordered in the obvious way: the instance ι_0 precedes ι_1 iff the former is above the latter in π. This allows us to introduce the following notion. Let ι be an instance of $\mathbf{ad}^X$ occurring in π. The *deviation* of ι is the number of instances of U^X preceding ι in π. The deviation of π is the sum of the deviations of all the instances of $\mathbf{ad}^X$ in π. Consequently, in a proof

$$\begin{array}{c} \Delta \vdash E \\ \vdots \\ \hline \Pi \vdash F \end{array} \tag{3.59}$$

of deviation 0, an application of $\mathbf{ad}^X$ cannot be preceded by instances of U^X. This means that the above proof may be partitioned into two segments: an initial segment

$$\begin{array}{c} \Delta \vdash E \\ \vdots \\ \hline \Lambda \vdash E \end{array}$$

containing only applications of $\mathbf{ad}^X$, followed by a proof

$$\begin{array}{c} \Lambda \vdash E \\ \vdots \\ \hline \Pi \vdash F \end{array} \tag{3.60}$$

in U^X. Otherwise said, the proofs of deviation 0 are proofs where the unification steps are dalyed. We will see that the delayed SLDS proofs are enough for computing answers. This result has been suggested to us by G. Ferrand in a personal communication. We simply adapted it to our formalism.

Since the instances occurring in a proof π are totally ordered, if π is of positive complexity, we can unambiguously talk about *the first instance of* $\mathbf{ad}^X$ *of positive deviation in* π. It is easy to see that such an instance is immediately preceded by an instance of U^X. As a final remark, let r be a unification rule, and let

$$\frac{\Delta \vdash E}{\Lambda \vdash F}$$

be an instance of r^X. Since X is the full exchange rule, if Π is any permutation of Δ,

$$\frac{\Pi \vdash E}{\Lambda \vdash F}$$

is also an instance of r^X. This observation can be generalised in the obvious way to the instances of $\mathbf{ad}^X$. Hence, if

$$\frac{\begin{array}{c}\Delta \vdash E \\ \vdots \end{array}}{\Lambda \vdash F} \tag{3.61}$$

is a proof in SLDS^X, and we replace the axiom $\Delta \vdash E$ by one of its permutations, then we obtain a proof in SLDS^X.

Proposition 3.48 *If we are given the proof (3.61) in* $SLDS^X$*, then we can construct a proof*

$$\frac{\begin{array}{c}\Delta \vdash E \\ \vdots \end{array}}{\Pi \vdash F} \tag{3.62}$$

in $SLDS^X$*, of the same complexity, and deviation 0, with* Π *a permutation of* Λ.

The idea of the proof is to show that we can obtain (3.62) from (3.61) by a finite number of applications of the transformation presented next.

Proposition 3.49 *Let r be a unification rule, and suppose that*

$$\frac{\Gamma \vdash C \qquad \dfrac{\dfrac{\dfrac{\Delta \vdash E_0}{\Delta_0\ t \asymp u \vdash E_0}}{\Delta_0\ \Lambda_0 \vdash E_1}}{\Lambda_1\ A \vdash E_1}}{\Lambda_1\ \Gamma \propto (C, A) \vdash E_1}$$

is an application of r^X *followed by one of* $\mathbf{ad}^X$. *Then there exist* Π_0 *and* Π_1 *such that*

$$\dfrac{\dfrac{\Gamma \vdash C \qquad \Delta \vdash E_0}{\Pi_0 \vdash E_0}}{\Pi_1 \vdash E_1}$$

is an application of $\mathbf{ad}^X$, *followed by one of* r^X, *and* Π_1 *is a permutation of* $\Lambda_1\ \Gamma \propto (C, A)$. *Intuitively, we can always conmute an application of a unification rule followed by an application of* $\mathbf{ad}^X$.

Proof By hypothesis, $\Delta_0\ t \asymp u$ is a permutation of Δ, Λ_0 is a list of equations, and $\Lambda_1\ A$ is a permutation of $\Delta_0\ \Lambda_0$. Hence, there exist Δ_1, and Δ_2 such that $\Delta_0 = \Delta_1\ A\ \Delta_2$, and Λ_1 is a permutation of $\Delta_1\ \Delta_2\ \Lambda_0$. This implies that $\Delta_1\ \Delta_2\ \Gamma \propto (C, A)\ \Lambda_0$ is a permutation of $\Lambda_1\ \Gamma \propto (C, A)$, and

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash C \quad \cdot \quad \dfrac{\Delta \vdash E_0}{\Delta_1\ \Delta_2\ t \asymp u\ A \vdash E_0}}{\Delta_1\ \Delta_2\ t \asymp u\ \Gamma \propto (C, A) \vdash E_0}}{\Delta_1\ \Delta_2\ \Gamma \propto (C, A)\ t \asymp u \vdash E_0}}{\Delta_1\ \Delta_2\ \Gamma \propto (C, A)\ \Lambda_0 \vdash E_1}$$

is a proof as demanded. □

Proof of Proposition 3.48 It suffices to show that if (3.61) is of deviation $d+1$, then we can construct a proof like (3.62) of deviation d. Consider the fist application of $\mathbf{ad}^X$ of positive deviation in π. Then, using the notation of the preceding proposition, π is of the form

$$\dfrac{\Gamma \vdash C \qquad \dfrac{\vdots}{\dfrac{\Delta \vdash E_0}{\Delta_0\ \Lambda_0 \vdash E_1}}}{\lambda \left\{ \begin{array}{c} \Lambda_1\ \Gamma \propto (C, A) \vdash E_1 \\ \vdots \end{array} \right.}$$

If we transform these instances as indicated before, we get something like

$$\dfrac{\dfrac{\Gamma \vdash C \qquad \dfrac{\vdots}{\Delta \vdash E_0}}{\Delta_1\ \Delta_2\ t \asymp u\ \Gamma \propto (C, A) \vdash E_0}}{\lambda' \left\{ \begin{array}{c} \Delta_1\ \Delta_2\ \Gamma \propto (C, A)\ \Lambda_0 \vdash E_1 \\ \vdots \end{array} \right.} \tag{3.63}$$

λ' is obtained from λ by a permutation in the axiom of the latter. By a previous remark, λ' is also a proof in SLDS^X, hence the same holds for the whole (3.63). It is also easy to verify that (3.63) is of deviation d. □

Corollary 3.50 *If*

$$\dfrac{\begin{array}{c}\Delta \vdash E \\ \vdots \\ \vdash F\end{array}}{}$$

is a refutation in SLD, there exists an equational goal $\Lambda \vdash E$ *obtainable from* $\Delta \vdash E$ *by a finite number of applications of* $\mathbf{ad}^X$, *and such that* σ_F^+ *is a principal unifier of* $\Lambda \vdash E$. □

Hence, F may be computed delaying the application of the unification rules. Here we have an example.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash p(a)} \quad \dfrac{p(x) \vdash p(f(x))}{p(x)\ f(x) \asymp f(a) \vdash} \quad \dfrac{}{p(f(a)) \vdash}}{f(x) \asymp f(a)\ a \asymp x \vdash}}{f(x) \asymp f(a) \vdash x \asymp a}}{x \asymp a \vdash x \asymp a}}{\vdash x \asymp a}$$

The example also shows that such a strategy may have a bad termination behaviour. If we use $p(x) \vdash p(f(x))$ in each application of **ad**, we will never arrive at an equational goal for firing the unification process. Despite this fact, it seems to us that the result merits attention at least for its theoretical interest.

Bibliography

[AK91] H. Ait-Kaci. Warren's Abstract Machine: A Tutorial reconstruction. Technical report, 8th ICLP. Tutorial Number 4, June 1991.

[Bar84] H Barendregt. *The Lambda-Calculus.* North-Holland, 1984.

[CM87] W. Clocksin and C Mellish. *Programming in Prolog.* Springer-Verlag, 1987.

[Dob90] Tep Dobry. *A High Performance Architecture for Prolog.* Kluwer Academic Publishers, 1990.

[GLLO88] G. Gabriel, T. Lindholm, E. Lusk, and R. Overbeek. A Tutorial on the Warren Abstract Machine for Computational Logic. Technical report, Argonne National Laboratory, 1988.

[Kri90] J.-L. Krivine. *Lambda Calcul. Types et modèles.* Masson, 1990.

[Llo87] John W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[Mai88] M. Maier. Analysis of Prolog Procedures for Indexing Purposes. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 800–807, November 1988.

[Mai90] M. Maier. Compilation of Compound Terms in Prolog. In *Proceedings of the NACLP '90*, pages 63–79, October 1990.

[MD] A. Mariën and B. Demoen. A New Scheme for Unification in WAM.

[Pad88] Peter Padawitz. *Computing in Horn Clause Theories.* Springer-Verlag, 1988.

[SS90] L. Sterling and E. Shapiro. *L'art de Prolog.* Masson, 1990.

[Van89] P. Van Roy. An Intermediate Language to Support Prolog's unification. In E. Lusk and R. Overbeek, editor, *Proceedings of the NACLP '89*, pages 1148–1164, Cleveland, October 1989.

[War83] D. H. Warren. An Abstract Prolog Instruction Set. Technical report, SRI International, Menlo Park, CA, 1983.

ISSN 0249-6399