Chapter 1

Introduction

The *Unification Problem* has several variants and has been studied in a number of fields of computer science (Knight, 1989; Baader and Snyder, 2001), including theorem proving, logic programming, automatic program transformation, computational linguistics, etc.

Abstractly, unification means: *given two descriptions d_1 and d_2, can we find an object o that fits both descriptions?*

In a more mathematical sense, unification consists of solving equations, i.e., given a pair of "terms" (*equation*) with some possibly common "unknowns" (*variables*), the problem is to decide whether or not there exists a possible assignation (*substitution*) to these unknowns that makes both objects "equal" modulo some equality theory. If such substitution exists, it is called *unifier*. When we have unknowns in only one of the terms, we talk about *matching* instead of unification. When we are not just considering a set (conjunction) of equations, but we allow to have more complex formulae combining equations and involving in particular negation, the problem is called *Disunification* (Comon, 1991).

Depending on what are the terms, the unknowns and the equality notion that we are considering, we get distinct kinds of unification. For instance, solving arithmetic equations can be seen as solving unification where the objects are arithmetic expressions, the unknowns are the variables that will be instantiated by numbers and the equality relation is the relation defined by the field structure of numbers. Therefore, the 10th Hilbert's problem, can be formalised as an (undecidable) unification problem.

1.1 Main Distinct Unification Problems

1.1.1 First-Order Unification

In First-Order Unification, terms are first-order terms, the unknowns are first-order variables, i.e. variables that can be instantiated by first-order terms, and the notion of equality corresponds to the syntactic equality.

The First-Order Unification Problem can be stated as: *Given two first-order terms s and t, does there exist a substitution σ of terms for the variables in s and t such that $\sigma(s) = \sigma(t)$?*

Note that $\sigma(s)$ and $\sigma(t)$ denote the application of substitution σ to terms s and t respectively i.e., $\sigma(s)$ and $\sigma(t)$ denote terms s and t respectively where all variables have been replaced by their corresponding values in σ. In general we will denote the unknowns by capital letters, constant symbols by lower case ones and substitutions by greek letters.

As a simple example, consider the following equation:

$$f(f(X_3, X_2), X_1) \overset{?}{=} f(X_1, f(X_2, X_3))$$

A possible unifier, could consist of assigning $f(a, a)$ to X_1 and a to X_2 and X_3. This assignment can be represented by means of the *substitution* σ:

$$\sigma = [X_1 \mapsto f(a, a),\ X_2 \mapsto a,\ X_3 \mapsto a]$$

Now, as we can see, the application of σ to both sides of the equation gives us the same term:

$$\sigma(f(f(X_3, X_2), X_1)) \ = \ f(f(a, a), f(a, a)) \ = \ \sigma(f(X_1, f(X_2, X_3)))$$

First-Order Unification was firstly studied by Herbrand in 1931, although its main crucial role in automated deduction was not considered until the 60's when J. A. Robinson invented the simple and powerful inference rule named *resolution* (Robinson, 1965). First-Order Unification is the cornerstone of the rule discovered by Robinson. In some sense this was also the beginning of automated logic. Robinson also showed that the First-Order Unification problem is decidable and that whenever a solution, or *unifier* exists, there always exists what is called a *most general unifier*, i.e. a unifier from which all other unifiers can be generated. Even more, in First-Order Unification whenever this most general unifier exists, it is unique up to variable renaming. Robinson's algorithm was quite inefficient requiring exponential time and space in the worst case. A great deal of effort has gone into improving the efficiency of unification. Among several other results, there are the ones of Venturini-Zilli (1975) reducing the complexity of Robinson's algorithm to quadratic time, and of Martelli and Montanari (1976) when they proved that a linear time algorithm for unification exists.

It was soon realised that resolution is better behaved in the restricted context of Horn clauses, a subset of first-order logic for which SLD-resolution is complete. Colmenauer and Kowalski considered this class of clauses and defined the elegant programming language PROLOG, which spurred the whole new field of logic programming (Kowalski, 1974).

First-Order Unification is not just used in resolution, but for many other purposes: in type inferencing for polymorphic programming languages (Milner, 1978), in expert systems, or in the calculation of critical pairs in the Knuth/Bendix completion procedure (Knuth and Bendix, 1967). Even in another theorem proving method that does not use resolution, like the so called *matings*, unification is required (Andrews, 1981).

Since the early 60's there have been many attempts to generalise the basic paradigm of theorem proving, and these attempts have stimulated research into more general forms of unification. On the one hand there was the goal of adding equality into the theorem proving procedure, and on the other hand there was the goal of automate higher-order logic. Both goals propitiated the two main generalisations of First-Order Unification: *E-Unification* and *Higher-Order Unification.*

1.1.2 E-Unification

The relevance of equational reasoning, i.e., the replaceability of equals by equals, in ordinary mathematical reasoning, and the expressive power of first-order logic to define algebraic structures, naturally lead to the introduction of equality into resolution. In this framework, Robinson and Wos introduced a new deduction rule dedicated to equality, *paramodulation* (Robinson and Wos, 1969; Nieuwenhuis and Rubio, 2001). Due to efficiency reasons, the best option is to split the deduction mechanism considering just the non-equational part in the refutation mechanism and using *E-Unification*, that is, First-Order Unification modulo an equational theory, instead of simply First-Order Unification to deal with the equational reasoning during the unification steps (Plotkin, 1972).

In E-Unification the terms are again first-order terms, and the unknowns first-order variables, but now the notion of equality is not just *syntactic equality* but equality modulo a given equality theory E.

The E-Unification Problem can be stated as: *Given a (finite) set E of equalities and two first-order terms s and t, does there exist a substitution σ of terms for the variables in s and t such that $\sigma(s)$ and $\sigma(t)$ are provably equal from the equations in E?*

As an example, consider the equation:

$$f(f(X_1, X_2), c) \stackrel{?}{=} f(X_1, f(X_2, c))$$

and the equational theory:

$$E = \{\ f(x, f(y, z)) = f(f(x, y), z)\ \}$$

a possible solution[1] is the substitution:

$$\sigma = [X_1 \mapsto a,\ X_2 \mapsto b]$$

$$\sigma(f(f(X_1, X_2), c)) \ = \ f(f(a, b), c) \ =_E f(a, f(b, c)) \ = \ \sigma(f(X_1, f(X_2, c)))$$

where here $=_E$ means the equality modulo the theory E.

Unlike First-Order Unification, E-Unification is undecidable in general. For instance, due to the undecidability of the word problem for semi-groups, E-Unification is undecidable when considering the semi-groups theory. Another

[1]Notice that if we were just considering first-order unification, this equation would have no solution.

major difference with respect to First-Order Unification, is that if an equation is solvable then there may not be a single most general unifier.

Nevertheless there are some theories known to be decidable, for instance $\emptyset$-Unification (i.e. First-Order Unification), AC-Unification (i.e. *Associative-Commutative* Unification), D-Unification (i.e. *Distributive* Unification), and A-Unification (i.e. *Associative* Unification). For a good summarisation of E-Unification see (Siekmann and Szabó, 1984) and (Baader and Siekmann, 1993).

A-unification is of special interest to us because it corresponds to the well known *Word Unification Problem*, shown to be decidable by Makanin (1977).

1.1.3 Word Unification

Word (String) Unification is unification where the terms are words, the unknowns can be instantiated by words and equality means to be the same word.

The Word Unification Problem can be stated as: *Given two words w_1 and w_2, does there exist a substitution σ of words for the variables in w_1 and w_2 such that $\sigma(w_1)$ and $\sigma(w_2)$ are the same word?*

Word Unification can also be seen as A-Unification where we use an associative symbol to form the words. Word Unification is decidable, but solvable Word Unification Problem instances do not always have just one most general unifier but possibly infinitely many independent unifiers, as illustrated by the following example:

$$aX \overset{?}{=} Xa$$

for which for all $n \geq 0$, any substitution of the form:

$$[X \mapsto \overbrace{a \dots a}^{n}]$$

is a unifier, not comparable to any other.

The race for the decidability proof of Word Unification was long and full of little steps: firstly it was shown that the case when no variable occurs more than twice is decidable, then the three occurrences fragment was also proved decidable, and finally, thanks to the *exponent of periodicity* lemma, the general case was proved decidable by Makanin (1977). Since then, a lot of work has been made looking for lower upper bounds on the exponent of periodicity and trying to get the precise complexity of the problem (Kościelski and Pacholski, 1995, 1996). Two recent and independent works, due to Plandowski (1999a,b) and Gutiérrez (1998, 2000), show, with an alternative to Makanin's proof, that Word Unification is in the NEXP class and in the PSPACE class. These are the best complexity classes known for Word Unification. Nevertheless, some people believe that Word Unification is in NP.

It has also been proved that word equations, where variables can be constrained to belong to regular languages, is also decidable (Schulz, 1991). Several attempts to simplify Makanin's proof and trying to give a practical implementation of the algorithm, have been made (Jaffar, 1990; Schulz, 1993). Expressiveness of word equations has also been subject of study in (Karhumäki *et al.*, 1997).

Word Unification has applications, for instance, in deduction systems (Huet, 1976) and in constraint logic programming (Colmerauer, 1988).

Closely related to Word Unification there is the *Context Unification problem.* In fact, Word Unification can be seen also as "*Monadic Context Unification*", i.e. Context Unification considering just a monadic signature.

1.1.4 Context Unification

In Context Unification the terms are first-order terms and the unknowns are first-order and context variables. Substitutions assign first-order terms to first-order variables and contexts to context variables. *Contexts* are terms with "holes", i.e. terms with a special constant symbol called the *hole*, that is denoted by '$\bullet$'. When a context is "applied" to some terms (arguments), the result is the term formed by the context where the holes have been replaced by the argument terms.

The Context Unification Problem can be stated as: *Given two terms s and t, does there exist a substitution σ of first-order terms and contexts for the variables in s and t such that $\sigma(s)$ and $\sigma(t)$ are the same term?*

As an example consider:

$$f(F(a), b) \stackrel{?}{=} F(f(a, b))$$

where F is a context variable. One of its solutions is the substitution:

$$\sigma = [F \mapsto \bullet]$$

which, when applied to the equation, identifies both sides:

$$\sigma(f(F(a), b)) = f(a, b) = \sigma(F(f(a, b)))$$

But as we can easily observe, as in the previous Word Unification example, there are infinitely many incomparable solutions for this equation, all of them having this form:

$$F \to \overbrace{f(\dots f(}^{n} \bullet \overbrace{, b) \dots, b)}^{n}$$

Therefore, solvable Context Unification equations can have infinitely many most general unifiers.

The Context Unification problem was firstly defined by Comon (1992a) and its decidability still remains unsolved. Context Unification has applications in rewriting (Comon, 1992a,b, 1998; Niehren *et al.*, 1997a, 2000), in unification theory (Schmidt-Schauß, 1996, 1998), and in computational linguistics (Pinkal, 1995; Niehren *et al.*, 1997b; Egg *et al.*, 1998; Niehren and Villaret, 2002, 2003).

Context Unification can also be seen as a variant of *Higher-Order Unification*, in fact it is closely related to Linear Second-Order Unification (Levy, 1996; Levy and Villaret, 2000, 2001), a variant of *Second-Order Unification.*

1.1.5 Higher-Order and Second-Order Unification

Higher-Order Unification serves for solving equations in the Simple Typed λ-Calculus (Church, 1940). In this unification problem, the terms considered are simply typed λ-terms, the unknowns are higher-order variables, i.e. variables that can be instantiated by simply typed λ-terms of the same type, and the equality relation is the congruence defined by the α, β and η congruencies of the λ-calculus.

The Higher-Order Unification Problem can be stated as: *Given two simply typed λ-terms s and t, does there exist a substitution σ of λ-terms for the variables in s and t such that $\sigma(s)$ and $\sigma(t)$ are λ-equivalent?*

For example, let s and t be:

$$F(\lambda y.\ y,\ b) \stackrel{?}{=} G(a)$$

one of its infinitely many solutions is:

$$\sigma = [F \mapsto \lambda x_1 x_2.\ x_1 x_2,\ G \mapsto \lambda x.\ b]$$

$$\sigma(s) = (\lambda x_1 x_2.\ x_1 x_2)(\lambda y.\ y,\ b) =_\beta (\lambda y.\ y)b =_\beta b\ {}_\beta{=} (\lambda x. b)a = \sigma(t)$$

where $=_\beta$ and ${}_\beta{=}$ denote β-equivalence in λ-calculus.

Like E-unification, Higher-Order Unification is undecidable in general and most general unifiers may not exist.

Second-order variables are variables that stand for functions on individuals. When variables are at most second-order, we talk about *Second-Order Unification*. Second-Order Unification is also undecidable (Goldfarb, 1981). There exists some recent work based on the number of distinct variables and the number of occurrences per variable, that draws a frontier between decidable and undecidable subclasses of Second-Order Unification (Levy, 1998; Levy and Veanes, 1998, 2000). Also the signature has been considered in the decidability question (Farmer, 1988, 1991).

There is a variant of Second-Order Unification problem called *Linear Second-Order Unification* (Levy, 1996). In this variant, it is imposed a limitation on the possible instances of variables: they are just allowed to be instantiated by linear terms, i.e. terms in normal form where each bound variable occurs in the body of the term once and only once. As we will show, this problem is closely related to Context Unification (Levy and Villaret, 2000). Although its decidability is still an open question, the fact that some subclasses that are undecidable in Second-Order Unification have been shown decidable in Context Unification (Levy, 1996; Schmidt-Schauß and Schulz, 1999), supports the common belief that Context Unification is decidable.

When variables are allowed to be instantiated by terms where bound variables can occur in the body of the term a bounded number of times, we talk about *Bounded Second-Order Unification*. Bounded Second-Order Unification is defined and shown decidable by Schmidt-Schauß (1999a, 2004). Adding the constraint that the number of lambdas in the unifiers is also bounded, and considering no just second-order variables, but variables of any order, the *Bounded*

Higher-Order Unification Problem is obtained. This last problem is also decidable (Schmidt-Schauß and Schulz, 2002a).

1.2 Higher-Order Applications

Despite of its undecidability, Higher-Order Unification is useful and necessary in many fields of computer science.

1.2.1 Automated Theorem Proving

Higher-Order Unification is required when automating higher-order logic. In this logic, quantification over sets or predicates and functions is allowed. This feature permits us, for instance, to axiomatise Peano arithmetic, which cannot be axiomatised just using first-order logic. But this increase on the expressive power is not for free. One of the major objections to the use of this logic comes from the Gödel first incompleteness theorem that states that no system that can formalise Peano arithmetic admits a complete deduction system. Nevertheless, Henkin generalised the notion of model theory with the so-called *general models* and proved that within this model theory, appropriate generalisations of first-order calculi to higher-order logics exist and are sound and complete. Since then, a wide range of methods for higher-order automated theorem proving has been proposed (Robinson, 1969; Darlington, 1971; Pietrzykowski, 1973; Huet, 1973a; Jensen and Pietrzykowski, 1976; Miller and Nadathur, 1987; Felty *et al.*, 1990; Paulson, 1990; Miller, 1991a; Paulson, 1993; Benzmüller and Kohlhase, 1998a,b). A textual cite from (Jensen and Pietrzykowski, 1976) illustrates the interest in higher-order logic despite its difficulty: *"...The attractiveness of higher-order methods in computational logic is not what you can or cannot prove, but rather that many proofs are more natural in a higher-order setting."*.

The first successful attempts to mechanise and implement higher-order logic were those of Pietrzykowski (1973); Jensen and Pietrzykowski (1976) and of Huet (1973a). They combined the resolution principle with Higher-Order Unification. As we have already said, Higher-Order Unification is undecidable, in fact semidecidable, and researchers were looking for procedures that were capable of completely enumerate all set of unifiers (notice that there can be infinitely many). The first implementation of a procedure for Higher-Order Unification already revealed that the search space for unifiers is far too large to be feasible for practical applications. Huet made a major contribution in showing that a restricted form of unification (also undecidable), called *preunification*, is sufficient for most refutation methods, and in defining a method for solving this restricted problem which is used by most current higher-order systems (Huet, 1975, 1976).

Nevertheless, since Higher-Order Unification is undecidable and when solutions exist there can be infinitely many, incorporating unification into the resolution inference rule would not result in an effectively computable rule. As a remedy, the unification process can be delayed by capturing the unification equations as constraints and effectively interleaving the search for empty clauses by resolution with the search of unifiers.

But Higher-Order Unification is not only used in automated theorem proving but also in *higher-order logic programming*, in *program synthesis* and *program transformation* and in *computational linguistics* among other areas. We will illustrate now some of these applications.

1.2.2 Higher-Order Logic Programming

As in Prolog, unification plays a crucial role in higher-order logic programming. But now, the variables considered are not just first order but higher-order ones, therefore we can write programs which are parameterised not just by values but by functions. This feature is not just a privilege of higher-order logic programming, but of higher-order programming in general like functional programming.

According to the use of variables there are distinct approaches, on the one hand there is the logical framework *Isabelle* (Paulson, 1990, 1993) that just allow higher-order variables as functions but not as predicates, and on the other hand there is λ*Prolog* (Felty *et al.*, 1990; Miller, 1991a,b; Müller and Niehren, 1998) that allows both uses of variables.

One simple example to illustrate the higher-order features is the definition of a mapping function: a function that takes a function and a list as arguments and produces a new list by applying the given function to each element of the former list. We show this example from the perspective of quantifying over predicates, in a relational style.

We write the predicates in a higher-order logic program style, *a la* Prolog, as follows:

```
mappred(P, [], []).
mappred(P, [X|L], [Y|K]) :- P(X, Y), mappred(P, L, K).

age(mateu, 31).
age(gemma, 29).
```

We have also added two facts that define the predicate `age` over two elements. Using this program, now we could get the list of ages of `mateu` and `gemma` with the query:

```
?- mappred(age, [mateu, gemma], L).
```

the answer of which would be the substitution `[31, 29]` for `L`. Tracing a successful solution path for this query we can observe that Higher-Order Unification has been required; for instance, to shoot the first rule, `P` has been matched with:

```
 \x y. age(x, y)
```

As we can also notice, when predicate variables get instantiated and after being supplied with appropriate arguments, they become new queries. In fact the first new goal to solve becomes `age(mateu, X')`. Therefore, one needs to be careful because Prolog only considers Horn clauses, therefore the predicate variables when instantiated need to be correct *Horn goals*. Accordingly with this fact, we can only use conjunctions, disjunctions and existential quantifications to instantiate predicate variables that can become queries.

In (Miller *et al.*, 1991), it is proposed the use of *Hereditary Harrop Formulae*, a generalisation on the formulae considered to overcome these Horn clauses limitations.

The previous example illustrates how predicate variables can be used and the power increase that they provide. But not everything one could expect to obtain can be achieved. Consider the following query:

```
?- mappred( R, [mateu, gemma], [31, 29]).
```

that could be used to find out what is the relation that exists between the two lists `[mateu, gemma]` and `[31, 29]`. One could expect the answer to be:

```
R -> \x y. age(x, y)
```

but this is too much optimistic. In fact, there are infinitely many relations that satisfy this query, and enumerating these does not seem to be a meaningful computational task. The problem can be stated in the intensional/extensional role of predicate variables. A broad discussion about these problems can be found in (Nadathur and Miller, 1998).

In the next subsection we will show how higher-order logic programming has nice properties to perform program transformations. We will also see that there are some other techniques that use Higher-Order Unification and Matching for program synthesis and program transformation.

1.2.3 Program Synthesis and Transformation

Automatic program synthesis consists of generating programs from specifications in an automatic manner. One of the pioneers of these techniques was Darlington (1973). His technique consisted of generating SNOBOL programs given a set of axioms based on those of Hoare, and employing a resolution based theorem prover incorporating a restricted Higher-Order Unification algorithm.

Program transformation is the process of converting a piece of code from one form to another whilst preserving its essential meaning. We will show a couple of perspectives to this field.

For instance, there is the work of Miller and Nadathur in λ-Prolog. Considering the higher-order facilities that λ-Prolog provides, basically its Higher-Order Unification features, we can see that it is possible to give rules that apply to certain patterns only matchable using Higher-Order Unification, and that allows us to re-build programs. One of such pattern examples occur in tail-recursive programs. From this recognition we can translate such programs into equivalent imperative programs.

Consider for instance, the following tail-recursive program that sums two non-negative integers, written in a pseudo λ-calculus style, and using `fixpt` as a recursive combinator:

$$\texttt{fixpt}\,\lambda sum.\lambda n.\lambda m.\,\texttt{if (n=0) then}\,m\ \ \texttt{else (}sum\ \texttt{(}n\texttt{-1) (}m\texttt{+1))}$$

The tail-recursiveness of this program can be easily recognised by using Higher-Order Unification (Matching). The program is in fact, an instance of the term:

$$\texttt{fixpt}\,\lambda f.\lambda x.\lambda y.\,\texttt{if (C}\ x\ y\texttt{) then (H}\ x\ y\texttt{) else (}f\ \texttt{(F1}\ x\ y\texttt{) (F2}\ x\ y\texttt{))}$$

as substitution σ shows:

$$\begin{array}{lll} [\ \texttt{C} & \mapsto & \lambda z_1 z_2.\ (z_1 = 0) \\ \ \ \texttt{H} & \mapsto & \lambda z_1 z_2.\ z_2 \\ \ \ \texttt{F1} & \mapsto & \lambda z_1 z_2.\ (z_1 - 1) \\ \ \ \texttt{F2} & \mapsto & \lambda z_1 z_2.\ (z_2 + 1)\ ] \end{array}$$

In fact, any closed term that unifies with this last "second-order template" must be a representation of a recursive program of two arguments whose body is a conditional and in which the only recursive call appears as the head of the expression that constitutes the "*else*" branch of the conditional. Clearly any functional program that has such a structure must be tail-recursive.

Now we should use these matched parts to form the corresponding imperative version of the program that will return the result in variable `result`. We use an imperative style *a la* PASCAL:

```
done := false
while (not done) do
      if (C par1 par2) then
         begin
           done := true;
           result := (H par1 par2)
         end
      else
         begin
           par1 := (F1 par1 par2);
           par2 := (F2 par1 par2)
         end
```

Now, if we apply substitution σ to this imperative program template term, we obtain the imperative version of the summing program:

```
done:=false
while (not done) do
      if (par1 = 0) then
         begin
```

```
          done := true;
          result := par2
        end
    else
        begin
          par1 := par1 - 1;
          par2 := par2 + 1
        end
```

Notice also that this template does not recognise all tail-recursive programs but just the ones that have two parameters and just one conditional in their body. A deeper study and discussion of how to solve this problem, and some other nice examples about program transformation can be found in (Miller and Nadathur, 1987).

There is a more recent work due to de Moor and Sittampalam (2001); Sittampalam and de Moor (2001). As they notice themselves, the automatic program transformation field has its major impact in easing the tension between program efficiency and program abstraction. The purpose of these program transformers is to translate an abstract and readable human-written code into an efficient one. But in general this task cannot be fully automatised and some human annotations are required.

These annotations are made by means of conditional higher-order rewriting rules that lead the transformed program to the transformation intended by the programmer. These rules require Higher-Order Matching.

The work of de Moor and Sittampalam on the development of the system MAG for HASKELL program transformation provides some nice examples that illustrate the power of Higher-Order Matching. One of their examples is the `reverse` list function expressed by means of a `fold`:

$$\texttt{reverse = foldr (}\lambda x.\lambda xs.\ xs\ \texttt{++ [}x\texttt{]) []}$$

In this definition, we first apply to each element of the list the "switching side" function and then the list concatenation function, therefore this `reverse` definition has quadratic time complexity. Our goal is to transform it into a linear time program using the so called "fold fusion" law:

$$\begin{gathered}\text{if}\\ \lambda x.\lambda y.\ (O_2)\ x\ (Fy)\ = \lambda x.\lambda y.\ F(O_1\ x\ y)\\ \text{then}\\ F\ \texttt{(foldr}\ (O_1)\ E\ \texttt{xs) = foldr}\ (O_2)\ (FE)\ \texttt{xs}\end{gathered}$$

The application of this law to our definition requires Higher-Order Matching and provides us with this substitution for the new fold operator:

$$[O_2 \mapsto \lambda x.\lambda g.\lambda xs.\ \ g(x\!:\!xs)]$$

Then, applying the resulting substitution[2] we obtain a linear time version of the `reverse` function:

```
reverse l = foldr (λx.λg.λxs. g(x:xs)) ((++) []) l []
```

1.2.4 Natural Language Semantics

Now we will illustrate the Higher-Order Unification applications in computational linguistics, like scope ambiguity (Pinkal, 1995; Niehren and Koller, 2001). We will dedicate a particular attention to the field of characterising the interpretative possibilities generated by elliptical constructions in natural language. In contrast to the previously presented applications of Higher-Order Unification, a part of our work, mainly Chapter 7 which is based on (Niehren and Villaret, 2002, 2003), is closely related to these natural language semantics topic. Hence, we will introduce this field in more detail than the previous ones.

In computational semantics, the formal description of the meaning of an expression often requires the use of sets and higher-order notions. The task of representing and reasoning about meaning in a computational setting was dealt, for instance, by Montague (1988), or by Miller and Nadathur (1986) who showed how it is possible to integrate syntactic and semantic analysis with λ-Prolog.

We now illustrate scope ambiguity and ellipsis, the two main linguistic phenomena where Context Unification has been used. Then we will introduce the approach that we will study and relate with Context Unification, in the last part of the thesis.

Scope Ambiguity

Scope Ambiguity consists of having more than one possibility for determining the scope of some elements (usually quantifiers) of the sentence. One of the examples in (Pinkal, 1995) is this scope ambiguity example where Linear Second-Order Unification is used, hence substitutions of Second-Order variables are required to be linear. The following sentence:

`Every researcher visited a company`

which is represented by the following equation:

$$C_1(@(every\_researcher, \lambda_{x_1}.(C_3(@(visit, @(x_1, x_2))))))$$
$$\stackrel{?}{=}$$
$$C_2(@(a\_company, \lambda_{x_2}.(C_4(@(visit, @(x_1, x_2))))))$$

has the following two possible solutions corresponding to the two possible readings:

[2]Notice that in fact O_2 is a third-order term because its second abstraction is a bound variable of order two, i.e. a function.

1. *every researcher visited a company which is not necessarily the same as the one that the others researchers visited.* To obtain this reading we consider the following substitution:

$$\left[\begin{array}{lll} C_1 & \mapsto & \lambda x.\ x \\ C_2 & \mapsto & \lambda x.\ @(every\_researcher, \lambda_{x_1}.(x)) \\ C_3 & \mapsto & \lambda x.\ @(a\_company, \lambda_{x_2}.(x)) \\ C_4 & \mapsto & \lambda x.\ x \end{array}\right]$$

   which, applied to the equation gives us the following term:

$$@(every\_researcher, \lambda_{x_1}.(@(a\_company, \lambda_{x_2}.(visit, @(x_1, x_2)))))$$

2. or *there exists a company (the same for all) that is visited by every researcher.* To obtain this reading we consider the following substitution:

$$\left[\begin{array}{lll} C_1 & \mapsto & \lambda x.\ @(a\_company, \lambda_{x_2}.(x)) \\ C_2 & \mapsto & \lambda x.\ x \\ C_3 & \mapsto & \lambda x.\ x \\ C_4 & \mapsto & \lambda x.\ @(every\_researcher, \lambda_{x_1}.(x)) \end{array}\right]$$

   which, applied to the equation, gives us the following term:

$$@(a\_company, \lambda_{x_2}.(@(every\_researcher, \lambda_{x_1}.(visit, @(x_1, x_2)))))$$

As we can appreciate, all substitutions are linear[3].

The work of Pinkal has been extended by Niehren *et al.* (1997b) where it is shown how Context Unification also deals with Ellipses.

Ellipses

Ellipses consist of omitting from a sentence, words needed to complete the construction or sense. In (Dalrymple *et al.*, 1991) it is shown how Higher-Order Unification correctly predicts a wide range of interactions between ellipsis and other semantic phenomena such as quantifier scope and bound anaphora. As a particular example, we can reproduce the verb phrase ellipsis phenomenon example from (Dalrymple *et al.*, 1991):

`Dan likes golf, and George does too.` (1.1)

The intended meaning of the sentence is that that Dan and George both like golf: $like(dan, golf) \wedge like(george, golf)$. The *source* clause, "`Dan likes golf`", parallels the target "`George does too`", with the subjects "`Dan`" and "`George`" being parallel elements, and the verb phrase of the target sentence being represented by the target phrase "`does too`".

[3]Notice that there is a distinction between the lambdas and the bound variables of the object language like λ_{x_1} and x_1, and the lambdas and bound variables of the substitutions, which disappear when applied to the term.

Now, we know that the property, let's say P, being predicated of George in the second sentence is such that when it is predicated on Dan, it means that Dan likes golf. We might state this by means of a Higher-Order Unification equation as follows:

$$P(dan) \stackrel{?}{=} like(dan, golf) \tag{1.2}$$

where P is a predicate variable. A possible value for P in this equation is the property represented by the λ-term $\lambda x.\ like(x, golf)$. Applying this predicate to George, we obtain $like(george, golf)$, and the full sentence meaning becomes the intended one:

$$like(dan, golf) \wedge like(george, golf)$$

Nevertheless not all the solutions of equation 1.2 have a meaningful counterpart in the linguistic semantic world. Consider now the substitution of P by $\lambda x.\ like(dan, golf)$. This is also a solution for the equation but when applied to George we obtain $like(dan, golf)$ and the following meaning for our elliptical sentence:

$$like(dan, golf) \wedge like(dan, golf)$$

which is not an interesting semantic interpretation, in fact it is wrong because it is not the intended meaning of sentence 1.1.

The way to solve this problem is to forbid some kind of substitutions, basically those that instantiate variables by terms that contain *primary occurrences* of the parallel elements (Dalrymple *et al.*, 1991). In this example then, the proposed second substitution is not a valid substitution because it contains a primary occurrence: *dan*. The technique of Dalrymple *et al.* (1991), computes reasonably enough solutions in comparison with other systems. But this way of filtering substitutions was not fully satisfactory. The goal was not to filter among a huge set of generated solutions, but rather to filter beforehand those solutions which are correct from those which are not.

There have been several researchers who have approached this problem, for instance Gardent and Kohlhase (1996), who deal with the primary occurrence constraint, or the one that we have shown in the scope ambiguity example of Pinkal (1995), using Linear Higher-Order Unification.

The Underspecified Semantic Representation Approach of The Constraint Language for Lambda Structures

The use of Context Unification has been broadly studied and related with other formalisms like *Dominance Constraints* (Koller *et al.*, 1998) and *Parallelism Constraints* (Erk and Niehren, 2000), in the works of Egg *et al.* (1998, 2001); Erk *et al.* (2002). Although Parallelism Constraints are equivalent to Context Unification, the procedures used to solve these constraints have a nicer behaviour than the ones for solving Context Unification (Koller, 1998; Erk and Niehren, 2000; Erk *et al.*, 2002), for instance the implementation of an incomplete Context Unification procedure in (Koller, 1998), runs into combinatoric explosion when dealing with scope ambiguities, and it does not perform well enough on the Context Unification equivalent of Dominance Constraints.