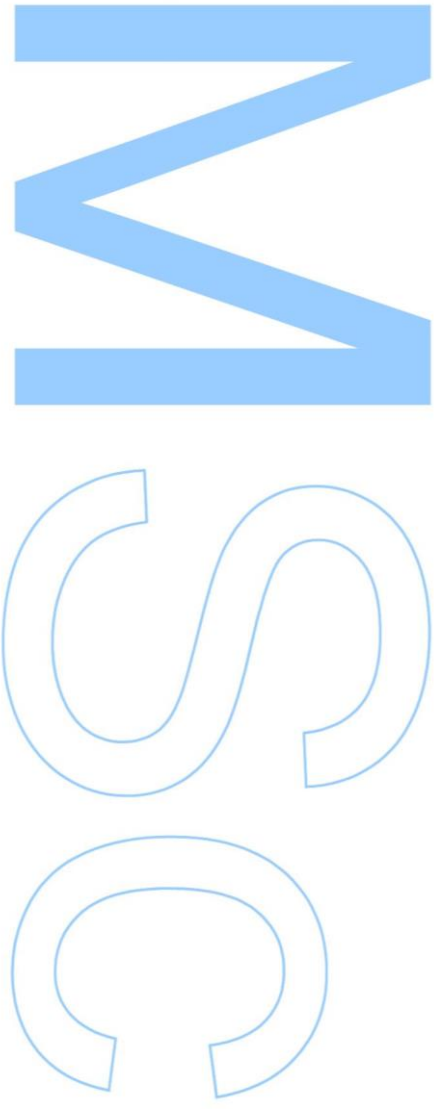
Closed Types for Logic Programming

João Luís Alves Barbosa
Master's degree in Computer Science
Computer Science Department
2016

Supervisor
Mário Florido, Associate Professor, Faculty of Science, University of Porto

Co-Supervisor
Vítor Santos Costa, Associate Professor, Faculty of Science, University of Porto



All corrections determined by the jury, and only those, were incorporated.

The President of the Jury,

Porto, _____/_____/_____

Acknowledgments

I want to thank both my supervisors for the help they provided in the making of this work as well as insisting on me not to stop working and always being available to help solving all the problems that appeared.

I also want to thank my parents for supporting me in my academic choices and providing everything I needed and wanted to finish this course. Without them it would not be possible.

To all my friends a huge thank you for being there for me and encouraging me to work and study as well as providing a lot of fun time that gave me strength and distracted me when I needed to.

Abstract

In this work, we present a new type system for logic programming and a type inference algorithm that we prove to be sound with respect to the type system.

The types inferred by our algorithm are closed, meaning that some programs will not be accepted because they will be too general. The programs that are not accepted by our type system include those where there is a single occurrence of a free variable in the head of a clause defining a predicate and programs that implicitly define open data types. When a program is accepted by the algorithm and we infer types for the predicates in the program, then those types will be closed and, as such, a closure step is needed to ensure that. The closure step is the final step of our algorithm and we believe it is not too restrictive nor too broad so that the final types are closer to the intended types of the programmer than they were before this step.

The semantics we present for types is quite different from the traditional semantics for types in logic programs, as types do not represent sets of terms, but sets of predicates. This relates to the functional programming view of types and it has an important role in our work, since parametric polymorphism comes naturally from this interpretation.

Resumo

Neste trabalho, apresentamos um novo sistema de tipos para programção lógica e um algoritmo de inferência de tipos que provamos ser correto em relação ao sistema de tipos.

Os tipos inferidos pelo algoritmo são fechados, o que significa que alguns programas não são aceites por serem demasiado gerais. Os programas que não são aceites incluem os em que existe uma única ocorrência de uma variável livre na cabeça de uma cláusula que define um predicado e os que explicitamente definem tipos de dados abertos. Quando um programa é aceite pelo algoritmo e tipos são inferidos para os predicados do programa, então esses tipos são fechados e, como tal, um passo de fecho é necessário para assegurar isso. O passo de fecho é o último passo do nosso algoritmo e nós acreditamos que não é demasiado restritivo nem demasiado abrangente, para que os tipos finais estejam mais próximos dos tipos pretendidos pelo programador.

A semântica que apresentamos para tipos de predicados é bastante diferente da semântica tradicional para tipos em programação lógica, uma vez que os tipos não representam conjuntos de termos, mas conjuntos de predicados. Esta visão está relacionada com o ponto de vista da programação funcional e é uma interpretação importante no nosso trabalho, já que o polimorfismo paramétrico surge naturalmente dessa interpretação.

Contents

List of Figures

Chapter 1

Introduction

Type systems have become an important component of most modern programming languages. Originally designed as a tool for program verification, types are particularly useful in describing and verifying the correct use of complex data-structures, a key feature in the development of large programs. Moreover, by providing a description of the program, types can be a powerful building block in constructing programming analyses or in devising safe extensions to a programming language.

Logic programming offers a challenging domain for the design and development of type systems. Types have been proposed since the early days of the field [Mis84, Zob87, PR89, Pfe92]. The approaches are very diverse, ranging from strict type disciplines [MO84, LR91], often inspired by work in the functional language community, to very permissive frameworks that can type most logic programs. More recently, several researchers have challenged the logic programming community by making the point that type systems should be an available tool in logic programming systems.

This work is motivated by this challenge. The goal is pragmatic: we would like to extend the Prolog language with a type system that facilitates the task of the programmer. To do so, first we must understand how the type system can achieve this aim. We start from Milner's insight on the nature of a type system: we would like "for a programming language, a (denotational) semantics definition which is not over-generous in a certain sense". Next, we describe the meaning for over-generous (informally). Clearly, different researchers will have different perspectives on what should be allowed by a type system. The definition of this work is motivated by the goal: data-structure oriented programming that can be verified with least programming effort.

In more detail, the argument is that logic programs are very expressive, and in general the use of the logic variables may cause the acceptance of more terms than intended in a successful computation. We argue that such programs are overly generous and that in a fully typed Prolog we should have type constraints over all variables.

Thus it is proposed that *all uses of the logic variables in a program should be type constrained*, where by constraints it is meant that either the type of the variable is strictly smaller than the set of all possible terms, or that there is an equality constraint between different types. The types in which all variables are constrained are named *closed types*. Consider the following example:

Example 1. Let τ^i_{append} denote the type of the i-th argument of the Prolog *append* predicate and + denote a disjunction of types in a type definition.

`append`([ ], `X`, `X`).
`append`([`X`|`L1`], `L2`, [`X`|`L3`]) : −`append`(`L1`, `L2`, `L3`).

Then:

$\tau^1_{append} = [\,] + [\alpha \mid \tau^1_{append}]$
$\tau^2_{append} = \beta$
$\tau^3_{append} = \beta + [\alpha \mid \tau^3_{append}]$

is a (semantically valid) open type for *append* and:

$\tau^1_{append} = [\,] + [\alpha \mid \tau^1_{append}]$
$\tau^2_{append} = [\,] + [\alpha \mid \tau^2_{append}]$
$\tau^3_{append} = [\,] + [\alpha \mid \tau^3_{append}]$

is a closed type for *append*.

The predicate in the example above is also typed in the traditional way, requiring the first argument to be a list, but the second argument is not typed constrained, whereas closed types, having a more restrict type discipline, require the second and third argument to be type constrained and as such to be a list as well.

The second point is based on the principle of self-contained definitions: given that we have a set of constants C and function symbols F occurring in a predicate p, a type τ must either have a principal functor in F, a constant type generalizing C,

or be a sum type, which is a disjunction of the previous types, that is, it should be context-independent.

In what follows, a polymorphic type discipline for logic programming is developed based on these two principles. The goal is to achieve some important features of previous work on types for logic programming (and programming in general):

- Type checking at compile time. This work defines a type inference algorithm that can also be used to type check declarations explicitly given by the programmer. The types defined are *closed types*, which, like regular types can be described by tree automata [CDG+07]. Type checking can be done dynamically (at run-time) using a well-known relation between regular types, which are types that can be described by a regular term grammar, and a class of monadic logic programs [YFS92, FSVY91, FD92].

- The programmer may not declare types at all, because the type inference algorithm will automatically infer a type for a given untyped program.

- Polymorphism. For now, we deal with parametric polymorphism [PR89, Zob87]. The main advantage of using *closed types* instead of regular types is that closed types can describe polymorphic types that are connected between them, for instance, the type $list(\alpha) \times list(\alpha)$ will be the type for a predicate with two lists as arguments, such that the lists contain terms of the same type on both predicate arguments. This differs from monomorphic regular types because it is not possible to define a monomorphic type that acts the same way, since α, in the example, is describing any term and as such one of the lists could be a list of integers and the other a list of strings. In the future work section, there is a discussion about overloading and coercion.

The major novel contributions of the type system include:

- Closed types: as discussed above, closed types mean that one has a bound on all objects valid in a typed program. The motivation stems from experience in both functional and logic programming languages. First, functional languages have a similar challenge in union types, and require a similar restriction on union types to enable type inference. A second motivation stems from Datalog [UZ90]. Consider the following program:

```
i(X,Y) :- e(X).
```

This program is not allowed because `Y` matches any object in the database. Datalog implementations address this problem by explicitly disallowing unconstrained head free variables as they match the full Herbrand base. We argue that such a program should not be considered well typed.

- Type information. The motivation is to get closer to the programmer's 'intended' definitions [Nai92]. Zobel said about previous type inference algorithms "inferred types may have no relation to a predicate's 'intended' types, and are simply cartesian products, of sets of ground terms, that contain all tuples of ground terms that can occur in the predicate's success set" [DZ92]. Using the assumptions about what is not a too-generous program, we develop an approximation to the programmer's 'intended' types for a predicate.

- A new type system. The type system is the link between inference and semantics as we will show and it assumes an attribution of types to variables and a set of type rules in Zobel style [Zob87]. It is a different type system from others developed by other authors.

- Semantics. Previously, types in logic programming have been described as an approximation to the success set of a program. This work takes a different approach on types, where their interpretation is that they correspond to all the predicates that can have such type, in a more functional programming way of thinking.

We would like to see this proposal as following in the line of Lee Naish's "Specification = Program +Types" [Nai92]. In this case, our type system can be seen as a declarative way to complement control, by restricting programs in order to facilitate the development of large applications, while preserving the flavor of logic programming.

In Chapter 2, we will discuss some work from other authors, related to types in logic programming, showing different approaches to the subject and presenting the main definitions that are important and influential in the type system and type inference algorithm we defined.

In Chapter 3, we will present the type system and discuss some of the reasons behind the definitions, as well as define all the necessary properties we want in types defined by the type system.

In Chapter 4, we will present the type inference algorithm as well as a proof of its soundness regarding the type system.

In Chapter 5, we will show the core of the implementation of the type inference algorithm.

In Chapter 6, conclusions will be presented and some future work topics will be discussed.

Chapter 2

Type Systems for Declarative Languages

Type systems have been used as a paradigmatic method to ensure program correctness. One of the most obvious benefit of using type checking is error detection. When a language is richly typed, the type checker often captures bugs that would be very hard to detect later. The effectiveness of the type checker depends as much on the expressiveness of the type system as on the use the programmer gives it. For instance, if a programmer encodes all his data structures as *ListOfPaths*, then he will not benefit as much from the type checker as if he had used different data types for each structure. There are many other reasons to use type systems as a tool for a programming language such as abstraction of a program by modules in a safer way or as a helpful way to understand a program, by reading the types, as well as the safety that type systems provide.

There have been two major ways of introducing types in logic programming. One is by approximation to the success set of the program. The other is to follow functional languages and more often a Hindley-Milner like type system, where the interpretation of types is not a superset of the program success set, but filters it by defining a set of well-typed programs. In this work, we are going to talk about both of them a little in this section.

Since the early works on the area [YFS92, FSVY91, BJ88, DZ92, Mis84], type systems for logic programming describe types as recursive sets of ground terms. Often, these approaches are based on the notion of regular types[1]. For this type the intersection of

[1]The meaning of regular types is different across literature. In this work, we assume that regular

two types and type comparison are decidable [Zob87]. Regular types can be written as logic programs, namely monadic logic programs themselves as types for logic programs [FSVY91], using unary-predicate programs to describe types in a natural way.

Both type verification and type inference algorithms have been proposed [Zob87, YFS92, BJ88, HJ92, SBG08, GdW94]. Approaches differ on whether types are considered approximations of the success set of a logic program, or whether one wants to ensure that a type signature will be respected.

2.1 Hindley-Milner-Damas Type System

The Hindley-Milner-Damas type system [Mil78, DM82] is a type system for functional languages that treats types as sets of functions from input to output. In this type system, we also have type variables, representing any type, and basic types such as *Bool*, *Int*, *Char*, etc.

Types are built from *type operators* such as $\rightarrow$ to form types for functions and a definition of *type scheme*, which is a type quantified for every variable appearing in it.

Example 2. Let *id* be the identity function defines as follows:

`id x = x`

The type for this function is $\forall\alpha.\alpha \rightarrow \alpha$.

This type system is the basis for many other type systems for either functional programming or logic programming. It is the foundation for a lot of different variations of what is actually implemented on languages such as Haskell, ML, Curry and Mercury [AH16], etc.

Another definition from Damas and Milner in [DM82] is the type assignment algorithm W, that receives as input a set A of assumptions of the form $x : \sigma$ and an expression e and either fails or returns a substitution S and a type τ. They proved that the substitution S and the type τ returned by the algorithm are the most general. Type inference relies on the unification algorithm that given two types, either fails or returns a substitution V that is the most general unifier.

types are types that can be described by regular trees

Let $\overline{A}(\tau)$ be the type closure, i.e. $\overline{A}(\tau) = \forall\alpha_1 \ldots \alpha_n\tau$, where $\alpha_1 \ldots \alpha_n$ are occurring free in τ but not in A. The algorithm itself is defined as follows:

$W(A, e) = (S, \tau)$, where

1. If e is x and there is an assumption $x : \forall\alpha_1 \ldots \alpha_n\tau'$ in A then $S = Id$ and $\tau = [^{\beta_i}/_{\alpha_i}]\tau'$, where β_is are new variables.
2. If e is e_1e_2 then let $W(A, e_1) = (S_1, \tau_1)$ and $W(S_1A, e_2) = (S_2, \tau_2)$ and $U(S_2\tau_1, \tau_2 \rightarrow \beta) = V$ where β is new; then $S = VS_2S_1$ and $\tau = V\beta$.
3. If e is $\lambda x.e_1$ then let β be a new type variable and $W(A_x \cup \{x : \beta\}, e_1) = (S_1, \tau_1)$; then $S = S_1$ and $\tau = S_1\beta \rightarrow \tau_1$.
4. If e is *let* $x = e_1$ *in* e_2 then let $W(A, e_1) = (S_1, \tau_1)$ and $W(S_1A_x \cup \{x : \overline{S_1A}(\tau_1)\}, e_2) = (S_2, \tau_2)$; then $S = S_2S_1$ and $\tau = \tau_2$.

Damas and Milner also proved that W is sound and complete with respect to the Hindley-Milner type system.

2.2 Regular Types

Zobel's work is one of the first examples of type inference based on the success set of a logic program [Zob87]. Zobel used regular types, which are types that can be described with a regular term grammar, as types for logic programs. Type terms can be:

- A type variable (X,Y, ...)
- A type symbol (defined in a set of type rules) (α, β, ...)
- A type constant (1, a, ...)
- A function symbol with arity n, applied to an n-tuple of regular types ($f(\tau_1, \ldots, \tau_n)$)

It is assumed that every type constant can be typed by a base type, for instance, *1* can be typed with the base type *int*. There is a set T of type rules, each of them defining a type symbol. Zobel also introduces two types representing every term (μ) or no term

(ϕ). We can associate a definite program Φ_T to this set consisting of the definitions for μ, ϕ, any base type and the set of clauses associated with the type rules defined in T. For example, the logic program corresponding to the type $\tau \rightarrow \{[\,], [\alpha|\tau]\}$ is:

$\mathtt{t}([\,]).$
$\mathtt{t}([\mathtt{X}|\mathtt{R}]) : -\alpha(\mathtt{X}), \mathtt{t}(\mathtt{R}).$

The interpretation of a type symbol according to T, $[\alpha]_T$ is the set of terms occurring as arguments of the unary predicate α in the least model M_{Φ_T} of the program Φ_T. This association is defined not only for type symbols but also variable-free type terms and it is defined as follows:

$$[\tau]_T = \begin{cases} \{c\} & \text{if } \tau \text{ is a constant symbol } c \\ \{t \mid \alpha(t) \in M_{\Phi_T}\} & \text{if } \tau \text{ is a type symbol } \alpha \\ \{f(t_1, \ldots, t_n) \mid t_i \in [\tau_i], 1 \leq i \leq n\} & \text{if } \tau \text{ is a variable-free type term } f(\tau_1, \ldots, \tau_n) \end{cases}$$

Regular types correspond to tree automata [CDG+07], which is a class of languages where intersection, subset and unification are decidable. Types are defined through type rules T.

Example 3. Let α be a type symbol. The following is a type rule defining a list of β with the list constructor '.':
$\alpha \rightarrow \{nil, .(\beta, \alpha)\}$
where α and β are type symbols, and there should be a another rule defining β.

One important characteristic of this algorithm is that the regular types infered are tuple distibutive [LC98], which means that inter-argument dependencies are ignored when inferring the types. For instance, if we have a set of terms that use the same function symbol such as $\{g(a, b), g(c, d)\}$, then with the tuple distributivity property, we can ensure $g(c, b)$ and $g(a, d)$ have the same type. This is an essential property for the correctness and completeness of the algorithm [LC98].

To build a type inference algorithm using regular types, Zobel also defined four main functions [Zob87, DZ92].

The **empty** function is given a variable-free type term and a set of type rules defining all necessary type symbols and returns true if the type term is empty, and false otherwise. For a type term to be empty, either it is the empty type ϕ, or it is a function

symbol applied to an empty type among its arguments, or it is a type symbol, whose definition is empty.

The **subset** function takes as input type terms τ_1 and τ_2 and a set of type rules defining all necessary type symbols and returns true if τ_1 is a subset of τ_2. There is a number of possibilities, either they are the same, or τ_1 is a constant type and τ_2 is the base type of that constant, or they have the same function symbol and the arguments of τ_1 are a subset of the arguments of τ_2. It may also be that τ_1 is the empty type or τ_2 is μ, or τ_1 is a type symbol whose definition is a subset of τ_2.

Example 4. Let κ be a base type symbol denoting the base type *atomic* and let $T = \{\alpha \to \{nil, f(a, f(a, \alpha))\}, \beta \to \{nil, f(\kappa, \beta)\}$. Consider the comparison $subset(\alpha, \beta)$.

As we can see, α denotes the set of structures of even lengths of a and β denotes the set of structures of *atomic* symbols. So the result of the comparison is true, since, in fact, $[\alpha]_T \subseteq [\beta]_T$.

The function **intersect** takes two variable-free type terms τ_1 and τ_2 and a set of type rules defining all necessary type symbols occurring in both input type terms and returns a type term that represents the intersection of the sets represented by τ_1 and τ_2. It also returns a new set of type rules, since intersection of some types may create new type symbols, with rules associated.

Example 5. Let us consider $T = \{\alpha \to \{s(0), s(s(\alpha))\}, \beta \to \{0, s(s(\beta))\}\}$, where α represents the set of odd length structures with the function symbol s and β the even ones. The result of applying $intersection(\alpha, \beta, T)$ is the pair (γ, T'), where $T' = T \cup \{\gamma \to \{s(s(\gamma))\}\}$, which is now an empty type, as can be verified by $empty(\gamma) = true$.

The last function and the one which is used on the type inference algorithm is **unify**. The **unify** function takes two type terms τ_1 and τ_2 and, again, a set of type rules and returns a type corresponding to the unification of τ_1 and τ_2 (given a substitution S, $\tau_1 S = \tau_2 S$), a new set of type rules, since new type symbols may be necessary for the unification, as well as a substitution for the variables appearing on both types. The **unify** function calls **intersect** when both type terms are variable-free, since the unification makes no sense with two variable-free type terms because $\tau_1 S = \tau_1$ and $\tau_2 S = \tau_2$. Here, we present the full algorithm as defined in [Zob87]:

$unify(\tau_1, \tau_2, T, \Theta) =$

1. Suppose τ_1 and τ_2 are identical, return (τ_1, T, Θ).

2. Suppose one of τ_1 or τ_2 is μ. If τ_1 is μ, return (τ_2, T, Θ), otherwise return (τ_1, T, Θ).

3. Suppose at least one of τ_1 and τ_2 is a variable.
   If $\tau_1 < \tau_2$, let ${}^{\tau_1}/_{\omega}$ be the type binding for τ_1 in Θ. If τ_1 occurs in τ_2, return (ϕ, T, Θ), otherwise, let $(\omega', T_f, \Theta') := unify(\omega, \tau_2, T, \Theta)$. Replace ${}^{\tau_1}/_{\omega}$ in Θ' by ${}^{\tau_1}/_{\omega'}$ giving Θ_f and return (τ_1, T_f, Θ_f). Conversely for τ_2.

4. Suppose both τ_1 and τ_2 are variable-free type terms.
   Let $(\tau_f, T_f) :=$**intersect**(τ_1, τ_2, T) and return (τ_f, T_f, Θ).

5. Suppose one of τ_1 and τ_2 is a type symbol defined in T.
   Suppose τ_1 is a type symbol defined in T, $\tau_1 \to \Upsilon$, and τ_2 has top-level n-ary function symbol f, $n \geq 0$. Let $(\tau, T') :=$**intersect**$(\alpha, \tau_1, T \cup \{\alpha \to \{f(\mu, \ldots, \mu)\}\})$. Then τ is a type symbol, $\tau \to \Upsilon \in T'$, and each $\omega \in \Upsilon$ is of the form $f(\omega_1, \ldots, \omega_n)$. Let T'' be

   $$T' \cup \{\alpha_k \to \{\omega_k \mid f(\ldots, \omega_k, \ldots) \in \Upsilon\} | 1 \leq k \leq n\}$$

   where each α_k is a new type symbol.
   Let $(\tau_f, T_f, \Theta_f) := unify(\tau_2, f(\alpha_1, \ldots, \alpha_n), T'', \Theta)$ and return (τ_f, T_f, Θ_f). (conversely if τ_2 is a type symbol defined in T).

6. Suppose τ_1 is $f(\tau_1^1, \ldots, \tau_n^1)$ and τ_2 is $f(\tau_1^2, \ldots, \tau_n^2)$.
   For each i, $1 \leq i \leq n$, let $(\tau_i^f, T, \Theta) := unify(\tau_i^1, \tau_i^2, T, \Theta)$.
   Return $(f(\tau_1^f, \ldots, \tau_n^f), T, \Theta)$.

7. Otherwise, τ_1 and τ_2 are type terms with distinct top-level function symbols, or one of τ_1 or τ_2 is a complex term and the other is a base type symbol.
   Return (ϕ, T, Θ).

Example 6. Let $T = \{\alpha \to \{f(a, b), f(c, d)\}\}$. The call $unify(f(x, y), \alpha, T, \emptyset)$ returns the triple $(f(x, y), T_f, \Theta_f)$, where $T_f = T \cup \{\beta \to \{a, c\}, \gamma \to \{b, d\}\}$ and $\Theta_f = \{{}^{\beta}/_{x}, {}^{\gamma}/_{y}\}$. So, $[f(x, y)\Theta_f]_{T_f}$ is $\{f(a, b), f(a, d), f(c, b), f(c, d)\}$ and the result given by the function is the most general type unifier of α and $f(x, y)$. Here we can see the *tuple distributivity* property of this algorithm.

Example 7. Consider the call $unify(x, f(y,z), \emptyset, \{^{g(w)}/_{x}\})$. This call will fail, and return the empty type as a result, since the unification of different function symbols fails.

Using the unification described above, Zobel [Zob87, DZ92] inferred types from logic programs as follows:

Definition 1. *Let P be a definite logic program.*
$supertypes(P) =$
let $k := 0$;
let $T_0 := \{\alpha_i^p \rightarrow \{\mu\} \mid 1 \leq i \leq n,\ p$ is an n-ary predicate defined in $P\}$;
repeat
let $k := k + 1$;
let $T_k := programtype(P, T_{k-1})$;
until $I_{P,T_k} = I_{P,T_{k-1}}$
return T_k.

This function *supertypes* applies consecutively the function *programtype*, which generates the types for the predicates defined in the program, to a program P, until a fixed point is reached. There are some cases when a fixed point is not reached, but applying a cut off step is a solution for those cases.

The function that infers types for the program, *programtype* is defined as follows:

Definition 2. *Let P be a logic program and T a set of type rules defining the type symbols α_i^p for each predicate p in P.*
$programtype(P, T) =$
for each clause C in P of the form $p(t_1, \ldots, t_n) \leftarrow B$ ***do*** :
let $(flag, T^C, \Theta^C) := goaltype(\leftarrow B, T, vars(C))$;
if $flag = success$ then :
for each $i, 1 \leq i \leq n$ **do** :
let $\tau_{i,j}^p := t_i\Theta^C$ (where C is the jth definition of p);
else
for each $i, 1 \leq i \leq n$ **do** :
let $\tau_{i,j}^p := \phi$;
discard from T^C any type rules with α_i^p in the head, $1 \leq i \leq n$;
return $\bigcup_{C \in P} T^C \cup \{\alpha_i^p \rightarrow \{\tau_{i,1}^p, \ldots, \tau_{i,l}^p\} \mid 1 \leq i \leq n,\ p$ is an n-ary predicate in P whose definition has l clauses$\}$.

This function infers types for every clause C. If they are not empty, which would mean there would be a type error in the program, add that information to the types for each predicate argument. We only need to define now, how to get types for each clause. This is achieved by looking at each subgoal in the body of the clause and infer types, with the **unify** function, and use the resulting substitution to the head of the clause.

Definition 3. *Let G be a goal $G = B_1, \ldots, B_m$, T a type declaration for G and V a set of variables, containing all variables on G.*
goaltype$(G, T) =$
let $\Theta = \{\mu/v \mid v \in V\}$;
for each $i, 1 \leq i \leq m$ **do** :
if B_i is an equality $t_1 = t_2$ **then** :
let $(t, T, \Theta) :=$ **unify**(t_1, t_2, T, Θ);
else B_i is an atom with n-ary predicate symbol p;
let $(t, T, \Theta) :=$ **unify**$(Bi, p(\alpha_1, \ldots, \alpha_n), T, \Theta)$;
if $[t\Theta]_T$ *is* ϕ **then**
return $(failure, T, \Theta)$;
return $(success, T, \Theta)$.

We can clearly see now that a type is inferred by unifying terms in the clause and getting substitutions for the head. The substitutions are the output of the **unify** function mentioned above. Initially every variable is given the value *any* as a substitution, since a type variable can have any type and then the substitution is changed in the **unify** step of the algorithm.

Finally, we will show an example of Zobel's type inference algorithm in action.

Example 8. Consider the following program P:

```
list(nil).
list([X|R]) :-list(R).
```

Then $T_0 = \{\alpha_1^{list} \rightarrow \{\mu\}\}$ and $T_1 = programtype(P, T_0)$ is $\{\alpha_1^{list} \rightarrow \{nil, [\mu|\alpha_1^{list}]\}$. We can see that $I_{P,T_1} = \{nil, [\mu|nil], \ldots\}$ is a subset of $I_{P,T_0} = \{\mu\}$. Now, $T_2 = programtype(P, T_1)$ is $\{\alpha_1^{list} \rightarrow \{nil, [\mu|\alpha_1^{list}]\}$, therefore $I_{P,T_2} = I_{P,T_1}$ so the algorithm stops and returns T_2.

2.3 Type Systems for Logic Programming

Most work on type systems for programming languages has been influenced by the traditional definition of type systems for λ-calculus and functional programming [Mil78, DM82]. Both this type system and its evolution in modern functional languages such as Haskell, have also been very influential in the design of type systems for functional-logic programming, such as Curry [Han13] or Mercury [SHC96].

Mycroft and O'Keefe [MO84] formulated a type system for Prolog, which Lakshman and Reddy later called Typed Prolog [LR91]. The type system used a grammar that described every Prolog object as follows:

Term ::= Var | Functor(Term*)
Atom ::= Pred(Term*)
Clause ::= Atom $\leftarrow$ Atom*
Sentence ::= Clause*
Program ::= Sentence; Atom
Resolvent ::= Atom*

The type rules that defined types in Mycroft and O'Keefe type system are clearer in [LR91], where the type rules are shown below. It assumes a type is a type term, represented generally by τ, or everything else containing well-typings, such as *Atoms*, *Formulas*, *Clauses* and *Programs*. Γ is a set of assertions of types for variables that can be viewed as a mapping from variables to type terms.

The rules introduced in Figure 2.1 define a well-typed Prolog program. The first rule is very intuitive and states that a variable is well-typed if in the environment the variable has such type associated with it. A complex term is well-typed if every argument is well-typed and matches the type of the function symbol, which is reasonable as well. The predicate is given the type *Atom* if it is well-typed, meaning that if all the arguments of the predicate are well-typed, and their type matches the type for the predicate arguments, then the type is correct and it is *Atom*. The empty formula is always well-typed and if we have a type *Atom* for something, then we have the type *Formula* for that. The equality is well-typed if the type for both sides of the equality are the same, this is also our approach for equality. If two formulas are well-typed, then the sequence is also well-typed. A clause is well-typed if the body is well-typed and the types of the terms in the head of the clause matched the predicate type unless for a renaming of variables. Finally, if all the clauses in a program are well-typed,

$$\Gamma \vdash x : \tau \qquad if (x : \tau) \in \Gamma$$

$$\frac{\Gamma \vdash t_1 : \theta(\tau_1) \ , \ldots , \ \Gamma \vdash t_n : \theta(\tau_n)}{\Gamma \vdash f(t_1, \ldots, t_n) : \theta(\tau')} \qquad if \ f : \tau_1 \times \cdots \times \tau_n \to \tau'$$

$$\frac{\Gamma \vdash t_1 : \theta(\tau_1) \ , \ldots , \ \Gamma \vdash t_n : \theta(\tau_n)}{\Gamma \vdash p(t_1, \ldots, t_n) : Atom} \qquad if \ p : Pred(\tau_1 \times \cdots \times \tau_n)$$

$$\Gamma \vdash \epsilon : Formula$$

$$\frac{\Gamma \vdash A : Atom}{\Gamma \vdash A : Formula}$$

$$\frac{\Gamma \vdash t_1 : \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash (t_1 = t_2) : Formula}$$

$$\frac{\Gamma \vdash \phi_1 : Formula \quad \Gamma \vdash \phi_2 : Formula}{\Gamma \vdash (\phi_1, \phi_2) : Formula}$$

$$\frac{\Gamma \vdash t_1 : \theta(\pi_1) \ \ldots \ \Gamma \vdash t_k : \theta(\pi_k) \quad \Gamma \vdash \phi : Formula}{\vdash [\forall X_1 : \tau_1, \ldots, X_n : \tau_n](p(t_1, \ldots, t_n) \leftarrow \phi) : Clause} \quad if \ \theta \ is \ a \ renaming \ substitution$$

$$\frac{\Gamma \vdash C_1 : Clause \ \ldots \ \Gamma \vdash C_l : Clause}{\Gamma \vdash (C_1 \ldots C_l) Program}$$

Figure 2.1: Rules of the Mycroft-O'Keefe Typed System for Prolog

then the program is well-typed.

We can see some paralels between this type system and the one defined by [Mil78]. An expression is only well-typed if we can derive it from the rules above. The resulting types are parametric since, if a function or predicate symbol have a type, then they have every instance of that type. One other assumption of this type system is the existence of basic types, such as *Int* or *Bool*. We would also like to denote the fact that for an equality, in a program, the type is only correct if both sides of the equality have the same type, which we can compare to the unify function from Zobel [DZ92], that works for the same goal.

This type system was developed without a type inference algorithm in mind, but the authors also made it possible to typecheck your program provided you declared the types, which is called a prescriptive type system.

2.4 Types in Prolog

Type systems have not been widely adopted by Prolog systems. Arguably, Ciao has been the major exception through the support of parametric types in its declaration system [HBC+11]. Ciao also includes libraries for regular and Hindley-Milner types. More recently, there has been a revival of interest in Hindley-Milner types for mainstream Prolog systems, such as SWI-Prolog and YAP, where a new module was introduced for typechecking that allows for the mixing of typed and untyped code, with type declarations, and run-time type cheking [SCWD08]. Their type system is based on Hindley-Milner types and it has some limitation, which the authors admit to be working on.

A similar approach for XSB was proposed in Hadjichristodoulou's gradual discovery of Hindley-Milner types [Had12], where programs need not a type declaration, but gradually become typed with similar types to the ones used recently in SWI-Prolog and YAP. These systems inferred types from untyped programs gradually and make suggestions for types for each clause, that become definitive if the programmer agrees with them.

Chapter 3

A Type System for Logic Programming

In this chapter, we will describe a new type system for logic programming. First, we will describe the syntax that is used, then some preliminaries, namely useful operations for the understanding of the type system that will be defined. We will then present the type rules of our system, explain their meaning and explain in detail the semantics of types, which differs from most work in types in logic programming.

3.1 Syntax

Our syntax is similar to the used by most of other authors. A predicate type is a tuple of possibly several argument types. Each argument type itself is a *sum type*, which is a disjunction of possibly several types. We shall use τ, σ or τ_p^i for type symbols and *sum types*. Furthermore, τ_p^i will be used to represent the type for the i-th argument of the predicate p. We will assume that every constant in a logic program can be typed by a base type, as it happens in most other programming languages. These base types are int, $float$, num, $char$, $string$, $atom$ and nil. Subtyping is allowed for base types, for example int and $float$ are contained in the base type num. We will separate the type nil from the type $atom$ to differentiate between nil and other atoms like a or b, since we want to make sense of things like *lists* which can only terminate in nil, and not other atoms. Type variables are types that represent no specific term and whose interpretation is every term that we can build, will be represented by α, β, γ.

Example 9. These are examples of possible argument types according to our syntax:

$\tau_1 = int + atom,$
$\tau_2 = f(\alpha) + g(\alpha),$
$\tau_3 = \tau_g^1 + h(\tau_f^1)$

The operands of a sum type are called *summands*. Notice that in a sum type, or as an argument for a functor, we can have some other argument type.

3.2 Closed Types

Closed types are, intuitively, types that are constrained in some way. If a type is open, then its possibilities extend to any term constructed from any type constructor, including type constructors not present in the program defining that predicate. This is what we want to avoid and by forcing types to be closed, we get control on which function symbols and base types are accepted by the program.

The definition follows:

Definition 4 (Closed Types). *A type for a predicate argument τ is closed with respect to a set Θ defining all types for the predicate if:*

1. *$\tau \neq \alpha + \Phi$, for any non-empty Φ*
2. *$\tau = \alpha \in \Theta$ iff α occurs in $\Theta \setminus \{\tau = \alpha\}$*

The first case of this definition avoid open data types while the second case avoids unconstrained type variables which could be instantiated by the whole Herbrand universe.

Example 10. $\tau_1 = \alpha + f(\beta)$ is not a closed type with respect to any Θ, since it does not respect the first case of our definition.
$\tau_2 = int + f(\alpha)$ is a closed type with respect to any set Θ, since it does not have variables as *summands*.

Closed types create a controlled environment, meaning that if we were to build the Herbrand model of a logic program, the terms given to variables would not be instantiated with function symbols and constant symbols that were not present in the

program. Or, if they were, there would be some restriction to the possibilities of such types. This is our motivation, coming from the following intuition: closed types can only contain terms built from function and constant symbols defined in the program, or in the case they are represented by a variable, there is a constraint on that variable.

3.3 Type Intersection

Type intersection is fundamental in the type system in order to type the conjunction of two goals that share at least one variable. This was previously defined for regular types [DZ92] and discussed in Section 2.2. Here we discuss in depth the definition of the operation with the alterations we made for our type system.

Let T be a set of type rules defining a certain number of type symbols and I, a set of triples $\langle \tau_1, \tau_2, \tau \rangle$ that stores information on the result of intersecting two types to guarantee termination (I always starts as the empty set).

$intersection(\tau_1, \tau_2, T, I) =$

1. Suppose τ_1 and τ_2 are identical. Return (τ_1, T).

2. Suppose one of τ_1 and τ_2 is a variable. If τ_1 is a variable, return (τ_2, T), otherwise, return (τ_1, T).

3. Suppose there exists α such that $\langle \tau_1, \tau_2, \alpha \rangle \in I$. Return (α, T).

4. Suppose at least one of τ_1 and τ_2 is a type symbol defined in T.
   If τ_1 is defined in T, then let Υ_1 be such that $\tau_1 \rightarrow \Upsilon_1$, else let $\Upsilon_1 := \{\tau_1\}$.
   Define Υ_2 similarly for τ_2.
   Let $(\Upsilon_f, T) := cpi(\Upsilon_1, \Upsilon_2, T, I \cup \{\langle \tau_1, \tau_2, \alpha_f \rangle\})$, where *cpi* calculates the intersection of every term of Υ_1 with every term of Υ_2, and return $(\alpha_f, T \cup \{\alpha_f \rightarrow \Upsilon_f\})$.

5. Suppose τ_1 is $f(\tau_1^1, \ldots, \tau_n^1)$ and τ_2 is $f(\tau_1^2, \ldots, \tau_n^2)$, $k \geq 0$.
   For each i, $1 \leq i \leq n$, let $(\tau_i^f, T) := intersection(\tau_i^1, \tau_i^2, T, I)$.
   Return $(f(\tau_1^f, \ldots, \tau_n^f), T)$.

6. Otherwise, either τ_1 and τ_2 are different base types or have different function symbols of one is a complex term and the other is a base type, in that case fail the intersection.

We can see that the intersection of two base types is themselves if they are equal, and it fails if they are different and the intersection of two type symbols is the intersection of the summands in their definition. The intersection of a type symbol and a term type is another type symbol whose definition is the intersection of the summands in the previous type symbol's definition and the term. The intersection of two complex type terms with the same function symbol is the complex type term built by intersecting argument by argument the initial type terms.

It may happen that a variable is used to intersect with another type term in some cases, for instance, if a variable is present in the definition of a type symbol, in which case the intersection is always the less general type term.

Example 11. The intersection of types:

$\tau_f^1 = int + f(int)$ and
$\tau_h^1 = int + atom$ is:
$\tau_0 = int$

It is important to note that type intersection was previously only defined for ground type terms, but we needed to extend such definition, since the type *any* does not exist in this syntax as it existed in [Zob87], from where this definition was based. We defined the intersection of a variable (which will only happen when this variable represents the type *any*) with any other type to be the type itself. It may happen that both are variables and in that case it is indifferent which one is chosen.

3.4 Type System

The type system defines a relation $\Theta, \Gamma \vdash p : \tau$, where Θ is a set of type definitions, defining all type symbols, $\Gamma = \{X_1 : \tau_1, \ldots, X_n : \tau_n\}$ is the environment of type declarations for logic variables, p is an expression (a term, a goal, or a predicate definition), and τ is a type. This relation should be read as expression p has type τ defined in Θ, given the type environment Γ. We write $\Theta(\alpha)$ for the type τ such that $\alpha = \tau$ is in Θ, $\Gamma(X)$ for the type paired with X in Γ and Γ_X for the type environment Γ with any pair for the variable X removed. We also write $def(\Theta)$ for the set $\{\alpha | \exists_\tau . \alpha = \tau \in \Theta\}$.

Let us now define two auxiliary operations on type definitions.

If Θ_1 and Θ_2 are two sets of type definitions, we define $\Theta_1 \oplus \Theta_2$ and $\Theta_1 \otimes \Theta_2$ as follows: for each $\alpha \in def(\Theta_1) \cup def(\Theta_2)$

$$(\Theta_1 \oplus \Theta_2)(\tau) = \begin{cases} \Theta_1(\tau), & \tau \notin def(\Theta_2) \\ \Theta_2(\tau), & \tau \notin def(\Theta_1) \\ \Theta_1(\tau) + \Theta_2(\tau), & otherwise \end{cases}$$

and

$$(\Theta_1 \otimes \Theta_2)(\tau) = \begin{cases} \Theta_1(\tau), & \tau \notin def(\Theta_2) \\ \Theta_2(\tau), & \tau \notin def(\Theta_1) \\ \textbf{intersect}(\Theta_1(\tau), \Theta_2(\tau)), & otherwise \end{cases}$$

One can easily show that $\oplus$ and $\otimes$ are commutative and associative operators.

To simplify the presentation of the type system let us assume that all predicates are defined in a single definition in disjunctive normal form. Note that it is always possible to transform a logic program consisting of several clauses into this form by substituting every comma with a $\wedge$ and separating each body in the same destination body by a $\vee$.

Example 12. In the case of the predicate *add*:

$\mathtt{add}(\mathtt{0}, \mathtt{X}, \mathtt{X})$.
$\mathtt{add}(\mathtt{s}(\mathtt{X}), \mathtt{Y}, \mathtt{s}(\mathtt{Z})) : -\mathtt{add}(\mathtt{X}, \mathtt{Y}, \mathtt{Z})$.

The normal form is:

$$\mathtt{add}(\mathtt{X_1}, \mathtt{X_2}, \mathtt{X_3}) : -(\mathtt{X_1} = \mathtt{0} \wedge \mathtt{X_2} = \mathtt{X_3}) \vee (\mathtt{X_1} = \mathtt{s}(\mathtt{X}) \wedge \mathtt{X_2} = \mathtt{Y} \wedge \mathtt{X_3} = \mathtt{s}(\mathtt{Z}) \wedge \\ \wedge \mathtt{Y_1} = \mathtt{X} \wedge \mathtt{Y_2} = \mathtt{Y} \wedge \mathtt{Y_3} = \mathtt{Z} \wedge \mathtt{add}(\mathtt{Y_1}, \mathtt{Y_2}, \mathtt{Y_3})).$$

We can see the rules of the type system in Figure 3.1. Let Σ represent a *sum type* as the sum of parcels that represents the disjunction of the several possibilities of the type and let $h(\vec{x})$ be the abbreviation of the predicate $h(x_1, \ldots, x_n)$. The VAR rule types a variable by seeing the definition of the type that is declared in Γ and either giving as a type the attributed type itself or a subset of the *summands* defining it. The operator $\sqsubseteq$ represents that the *sum type* on the left side has either some or all the *summands* that are on the right side *sum type*. The CONS rule types a constant assuming that its types are given by a built-in function. The TERM, GOAL and

$$VAR \quad \Theta_\tau \cup \{\tau = \Sigma\}, \Gamma_x \cup \{x : \tau\} \vdash x : \Sigma' \ , \quad \text{where } \Sigma' \sqsubseteq \Sigma \text{ or } \Sigma' = \tau$$

$$CONS \quad \Theta, \Gamma \vdash c : \tau \ , \quad \text{where } \tau \text{ is the base type for } c$$

$$TERM \quad \frac{\Theta, \Gamma \vdash t_1 : \tau_1, \ldots, \Theta, \Gamma \vdash t_n : \tau_n}{\Theta, \Gamma \vdash f(t_1, \ldots, t_n) : f(\tau_1, \ldots, \tau_n)}$$

$$UNIF \quad \frac{\Theta, \Gamma \vdash t_1 : \tau \quad \Theta, \Gamma \vdash t_2 : \tau}{\Theta, \Gamma \vdash t_1 = t_2 : \tau}$$

$$GOAL \quad \frac{\Theta, \Gamma \vdash t_1 : \tau_1, \ldots, \Theta, \Gamma \vdash t_n : \tau_n}{\Theta, \Gamma \vdash p(t_1, \ldots, t_n) : \tau_1 \times \cdots \times \tau_n}$$

$$CONJ \quad \frac{\Theta_1, \Gamma \vdash t_1 : \tau_1 \quad \Theta_2, \Gamma \vdash t_2 : \tau_2}{\Theta_1 \otimes \Theta_2, \Gamma \vdash t_1 \wedge t_2 : \tau_1 * \tau_2}$$

$$DISJ \quad \frac{\Theta_1, \Gamma \vdash t_1 : \tau_1 \quad \Theta_2, \Gamma \vdash t_2 : \tau_2}{\Theta_1 \oplus \Theta_2, \Gamma \vdash t_1 \vee t_2 : \tau_1 + \tau_2}$$

$$CLS \quad \frac{\Theta, \Gamma_x \cup \{\vec{x} : \vec{\tau}\} \vdash h(\vec{x}) : \vec{\tau} \quad \Theta, \Gamma_x \cup \{\vec{x} : \vec{\tau}\} \vdash b : \tau_1}{\Theta, \Gamma_x \vdash (h(\vec{x}) : -b) : \vec{\tau}}$$

Figure 3.1: Type rules

UNIF rules are straightforward. In the CONJ and DISJ rules, typing respectively conjunctions and disjunctions of conjunctions of goals, the symbols $*$ and $+$ on the types are used only as infix type constructors. The clause rule, CLS, propagates the type of the head of the clause to the main predicate definition and discards the types used for its arguments from the environment, given that the body of the clause can be typed with some type with the same sets Γ and Θ (note the similarity with the usual abstraction rule to type lambda-terms or functions in functional languages) [DM82].

Example 13. Here we present a derivation tree for the type of the predicate *list*, defined as follows:

`list`([ ]).
`list`([`X`|`Xs`]) : −`list`(`Xs`).

The derivation tree assumes a set of attributions of types to variables and a set of type rules defining the types attributed to those variables. To simplify the presentation of the tree, we will assume that along it, the set of attributions will be $\Gamma = \{X_1 : \tau, X : \sigma, Xs : \tau\}$ and the set of type rules, $\Theta = \{\tau = nil + [\sigma|\tau], \sigma = \alpha\}$.

Let $Tree_1$ be:

$$\frac{\Theta, \Gamma \vdash X_1 : nil \qquad \Theta, \Gamma \vdash [\,] : nil}{\Theta, \Gamma \vdash X_1 = [\,] : nil}$$

And $Tree_2$ be:

$$\dfrac{\dfrac{\dfrac{\Theta, \Gamma \vdash X : \sigma \qquad \Theta, \Gamma \vdash Xs : \tau}{\Theta, \Gamma \vdash [X|Xs] : [\sigma|\tau]} \qquad \Theta, \Gamma \vdash X_1 : [\sigma|\tau]}{\Theta, \Gamma \vdash X_1 = [X|Xs] : [\sigma|\tau]} \qquad \dfrac{\Theta, \Gamma \vdash Xs : \tau}{\Theta, \Gamma \vdash list(Xs) : \tau}}{\Theta, \Gamma \vdash (X_1 = [X|Xs] \wedge list(Xs)) : ([\sigma|\tau] \wedge \tau)}$$

The full derivation tree is:

$$\dfrac{\dfrac{\Theta, \Gamma \vdash X_1 : \tau}{\Theta, \Gamma \vdash list(X_1) : \tau} \qquad \dfrac{\begin{matrix} \vdots \\ Tree_1 \end{matrix} \qquad \begin{matrix} \vdots \\ Tree_2 \end{matrix}}{\Theta, \Gamma \vdash (X_1 = [\,]) \vee (X_1 = [X|Xs] \vee list(Xs)) : (nil \vee ([\sigma|\tau] \wedge \tau))}}{\Theta, \Gamma \vdash list(X_1) : -(X_1 = [\,]) \vee (X_1 = [X|Xs] \wedge list(Xs)) : \tau}$$

3.5 Semantics

Usually types for logic programming are interpreted as a conservative approximation to the program semantics. This is not the case of the type system defined in this work. In fact, for instance, the *append* predicate may have type $list(int)$ for its three arguments although it can be applied to lists of other types. We thus follow the view of types as descriptions of program properties and not as approximations of the program semantics.

The semantics of base types and argument types are defined (as usual in previous works on types for logic programming), as a set of terms, meaning $[\![int]\!] = [0, 1, 2, \dots]$ and $[\![int + atom]\!] = [\![int]\!] \cup [\![atom]\!]$.

However, the semantics defined for a predicate type is quite different from the ones in other literature on types for logic programs. Instead of interpreting a type as an approximation to the success set of a program, a descriptive view of types is taken (details about the descriptive versus prescriptive view of types can be found in [Bar92]), interpreting them as all the predicates that can be described by the type. So, for a *ground* predicate type the semantics is:

$$[\![\tau_1 \times \cdots \times \tau_n]\!] = \{p \mid \exists k_1 \dots k_n . p(k_1, \dots, k_n) \textit{ succeeds and } k_1 \in [\![\tau_1]\!], \dots, k_n \in [\![\tau_n]\!]\}$$

This is close to the view of descriptive type systems for functional languages. For example, in a descriptive functional the type $Int \rightarrow Int$ types every function from integer to integer, regardless of what the function is. Likewise, in this system, the type $int \times int$ is a type for every predicate that relates two integers.

3.5.1 Polymorphism

Given a parametric type $\tau(\alpha)$, where α is a type variable occurring in τ, its semantics can be seen as follows:

$$[\![\tau(\alpha)]\!] = \bigcap_{\forall k} [\![\tau(k)]\!]$$

With this interpretation, a type such as $\tau = list(\alpha) \times list(\alpha)$ describes all the predicates with two arguments that are lists of a parameter, such as *reverse*, *sublist*, etc., but not those that work on specific lists of integers or strings as they will not belong to

the intersection with other types: lists of integers do not belong to the semantics of the type $list(string)$ or vice-versa.

A consequence of this interpretation is that it can now easily be defined the notion of most general type. For instance, the predicate $reverse$, can have type $\tau_1 = list(int) \times list(int)$, but also $\tau_2 = list(atom) \times list(atom)$ and $\tau_3 = list(\alpha) \times list(\alpha)$. The type τ_3 is the most general type, as it is the one that for any type σ that we can give to the predicate $reverse$, $[\![\tau_3]\!] \subseteq [\![\sigma]\!]$. This is true since if there is a predicate that only accepts lists of integers, it will not have type τ_3, but if a predicate uses general polymorphic lists, in particular it will have all the types of instantiated lists, such as types τ_1 and τ_2.

The focus of this work is automatic type inference, mainly the definition of a descriptive type system for logic programming and a type inference algorithm which is sound with respect to the type system. In [LR91] a semantic relation between programs and types was presented using a type-theoretical model semantics for Typed Prolog based on many-sorted logic. There, the semantics of base types was viewed as sorts and the usual definitions of model semantics for logic programming (interpretations, minimal models and the immediate consequence operator T_P) were extended for typed predicates. We conjecture that this should also hold for our type system and we leave it to future work.

Chapter 4

Type Inference

4.1 Stratification

The type system of the previous chapter relates programs with types but it does not provide a method for finding, given a program p, a type τ, a set of type definitions Θ and a type environment Γ such that $\Theta, \Gamma \vdash p : \tau$. We now present an algorithm for this purpose. The algorithm assumes that input programs are in normal form. This normal form is the one presented in Section 3.4 and illustrated in Example 12 and was used in implementation settings for logic programming, being called *flatten form* [AK91]. It represents a general definition or call of a predicate $p(t_1, \ldots, t_n)$ by $p(X_1, \ldots, X_n)$ and a conjunction of equations of the form $X_1 = t_1, \ldots, X_n = t_n$. We use type unification and type intersection algorithms similar to the ones defined in [Zob87]. Another assumption we make is that the input program of our algorithm is stratified [Ull88]. To understand the meaning of stratified programs, let us define a dependency directed graph of a program as the graph that has one node representing each predicate in the program and an edge from p' to p for each call from a predicate p to a predicate p'.

Definition 5 (Stratified Program). *A stratified program P is such that the dependency directed graph of P has no cycles of size more than one.*

What this means is that programs that have some predicate p that depends on a predicate q and vice versa will not be typed by our algorithm. The same holds for other cycles of sizes of length greater than two. Stratification is relevant because for stratified programs we can infer types for each predicate, beginning by the ones whose

definition consists of facts or is self-recursive, and step-by-step infer types for predicates that only depend on types from previous steps and, possibly, are self-recursive as well.

4.2 Type Constraints

Our algorithm relies on solving type equations corresponding to type unification constraints. The constraint for a predicate p with arity n on argument i will be $\tau_p^i = term$, where $term$ is the content of the predicate in that argument in some clause and logic variables are replaced by type variables.
The algorithm follows the program according to its dependency graph: we start by inferring types for predicates that only depend on themselves or on no predicate at all (a predicate whose definition consists of facts only) and then we progressively infer types for predicates whose predicates they depend already have a type. We will use the type constraints present on the body due to the *flatten form* and get a substitution for the type variables on the head of the clause.

Example 14. The constraints we get for the following clause:

$\mathtt{p(X,Y)} : -\mathtt{f(X)}, \mathtt{g(X)}, \mathtt{q(1,Y)}.$

are:

$$\tau_f^1 = \alpha,\ \tau_g^1 = \alpha,\ \tau_q^1 = int,\ \tau_q^2 = \beta$$

Some of these constraints, such as $\tau_q^1 = int$ will be a simple type checking for the first argument of predicate q, the other constraints will generate a substitution for the variables on the head of the clause. As we mentioned before, we can assume that all predicates on the body are typed since the program is stratified.

4.3 Open vs Closed Type Inference

In some cases, when the definition of a predicate has a single occurrence of a variable in the heads of all the clauses defining the predicate, then type will be open. This happens because there is no information on the type of possible instances of the variable, since it is free.

Example 15. Let us consider the following definition for predicate p
$\mathtt{p}(\mathtt{X}_1, \mathtt{X}_2) : -(\mathtt{X}_1 = \mathtt{1} \wedge \mathtt{X}_2 = \mathtt{X})$.
The type we get for the second argument is $\tau_p^2 = \alpha$, where α is a type variable, that appears only once in the definition of the predicate and as such, it will appear only once on the set of types for the predicate. This makes the type for the second argument open.

Example 16. Consider the predicate:

$\mathtt{identity}(\mathtt{X}, \mathtt{X})$.

The type of both arguments is the same and equal to a type variable: $\tau^1_{\mathtt{identity}} = \tau^2_{\mathtt{identity}} = \alpha$. Note that X occurs twice in the predicate definition, thus there is a constraint on the type, so it is considered closed with respect to $\{\tau^1_{\mathtt{identity}} = \alpha, \tau^2_{\mathtt{identity}} = \alpha\}$

To ensure that all inferred types are closed, we introduce a closure operation. Such closure must be based on the predicate only, since our algorithm works step-by-step and it must include some kind of domain for which types can be closed on. The next definition defines this domain.

Definition 6 (Proper Domain). *Given a set of type definitions T, the proper domain of T is the set of closed types with respect to T.*

With this definition we can define a closure for an open type τ, assuming there are closed types with which we can close t. This will mean that a predicate of the form $p(X)$ is not accepted by our type system.

Definition 7 (Closure). *Given a type $\tau = \alpha + \Phi$ and a set of types S, a closure of τ with respect to S is $\tau[^{d_1+\cdots+d_n}/_{\alpha}]$, where $\{d_1 + \cdots + d_n\}$ is the proper domain of the subset S' of S, such that all types in S' are closed with respect to S and share a constructor with τ.*

Example 17. Given $\tau_1 = \alpha + c(\beta) + a(\gamma)$ and $S = \{\tau_2 = nil + c(\beta), \tau_3 = e(\epsilon) + a(\gamma)\}$, the closure of τ_1 is $\tau_1 = nil + e(\epsilon) + c(\beta) + a(\gamma)$.

In the case where no such types exist for the closure, the algorithm will close the type with itself, meaning the "open" case will just be removed from the sum of types in τ. This means that there is no other argument on that predicate that uses any type constructor on the argument in question, so we cannot use more information to close it.

4.4 Unification

Type inference relies on solving type constraints by a type unification algorithm. Type unification is also used as a kind of intrinsic type checking, because, for instance if a there is a unification of a type and a term that is not a possibility for that type then the algorithm will fail.

Example 18. Let us consider the following definition for a predicate p.
$\mathtt{p(X)} : -\mathtt{q(a)}.$
Let us also assume that the type previously inferred for q was $\tau_q^1 = int + g(int)$. Then the algorithm will try to match *atom* with the type for q and it will fail.

We use an algorithm based on Zobel's work [Zob87] on unification of regular terms. The algorithm consists of two main predicates, one is the predicate *unify*, that given two terms to unify and a set of types from previously typed predicates returns a triple with the type that unifies them, a new set of types and a substitution for the variables occurring in the body of the clause. When unifying two terms, it may happen that both of them are ground terms or are type symbols defined in the rules and as such the unification requires the intersection of those terms. That is the other main predicate, *intersection*. It receives as input two type terms and a set of type rules and returns their intersection and a new set of type rules. Because of differences on the formalization, such as the existence of a type μ that represents all possible term, in Zobel's work, we needed to do some changes on his algorithm that is described in detail in Section 2.2: instead of initializing the substitution for the variables with the type μ, that we do not have in our type system, we initialize it with a fresh type variable. This means that in the end, the substitution for a variable may still be a variable. We also will not have to deal with the case of unifying the type μ with another type. What will happen is that if the substitution of a variable is a type variable, then there is a step where we unify that type variable with another type and the result will always

be the other type. Note that these changes keep the correctness of the algorithm since the result will remain the same as if the type variable was the type μ.

After the unification process is finished, we have a substitution for all variables that occur in the body of the clause, and we are going to use it to add types to the sum types for each argument of the predicate in the head of the clause. Note that a variable may occur in the head of a clause without occurring in the body and therefore have no substitution coming from the *unify* function. In that case, we substitute it for a fresh type variable.

We will now define a normal form for the type rules that define the types for predicate arguments. The normal form contains no repetitions of *summands* in the rule and does not contain the type itself in the rule as one of the *summands*. This comes from the interpretation that the sum type is a set of possibilities for the type and if a set S is equal to $A \cup A$, then it is equal to A, as well as if it is equal to $A \cup S$, then it is equal to A. Therefore, we have a normalization step after getting the types for a predicate, that removes duplicates and τ itself from the type rule defining τ.

Example 19. Consider we have the following rule defining the type symbol τ_p^1:

$$\tau_p^1 = int + f(int) + int + \tau_g^1 + \tau_p^1$$

After simplification, it becomes:

$$\tau_p^1 = int + f(int) + \tau_g^1$$

There are two important properties of the *unify* predicate.

Proposition 1 *The substitution resulting from the unification of types does not change* $def(\Theta)$ *for any* Θ *as input of the unification.*

Proposition 2 *If* $\Theta, \Gamma \vdash t : \tau$ *for some* Θ*, then* $S(\Theta), \Gamma \vdash t : S(\tau)$ *still holds, for any substitution* S*.*

The correctness of these properties comes from a simple case analysis of the *unify* algorithm.

4.5 Type Inference Algorithm

In this section, we will define the type inference algorithm and explain all its auxiliary functions. Starting from the ordering of predicates, to the algorithm itself and the

closure of types inferred, we will explain the reasoning behind all the components that constitute our algorithm and show some examples.

4.5.1 Ordering of Predicates

We assume the program is stratified, hence we need to get the dependencies of each predicate in order to build the dependency graph.

For this we find all predicates P' called from a predicate P and build a dependency graph. Given this graph, we can topologically sort the graph, obtaining a full order. The graph will have no cycle bigger than one, since we need the program to be stratified and that follows from Definition 5.

Example 20. The program:
$\mathtt{p}(\mathtt{X}_1) : -\mathtt{X}_1 = \mathtt{1}.$
$\mathtt{r}(\mathtt{X}_1) : -\mathtt{X}_1 = \mathtt{a}.$
$\mathtt{q}(\mathtt{X}_1, \mathtt{X}_2) : -\mathtt{X}_1 = \mathtt{X},\ \mathtt{X}_2 = \mathtt{Y},\ \mathtt{p}(\mathtt{X}),\ \mathtt{r}(\mathtt{Y}).$
$\mathtt{f}(\mathtt{X}_1) : -\mathtt{X}_1 = \mathtt{X},\ \mathtt{q}(\mathtt{X}, \mathtt{Y}).$
Will have the following order for the predicates: $p/1 \leq q/2,\ r/1 \leq q/2,\ q/2 \leq f/1.$

Note that the order is partial. The same program with a different order for the clauses would hold the same result, with the exception of predicates p and r, whose order is irrelevant and when that happens, we choose the order they were written with.

It is obvious to see that the order of the first two predicates is irrelevant since either starting with p or r, the second one would not depend on the previous and, as such, the result would be the same. On the other hand, q could never have a type inferred to it before p and r and neither could f before any other predicate, since, for instance, the type checking that occurs in $p(X)$ on the third clause would be impossible to perform, and a type could not be inferred for X other than a variable, representing every term.

4.5.2 Inference Step

In this step of the algorithm, the input is a program, possibly with several different predicates, and the order of application of the algorithm to the predicates and the output is a triple with a set of type definitions, a set with attributions of types to

variables and a type for the predicate, or types for the predicates, in the program. The core of the inference step follows:

$\mathit{infer}(X) = (\{\alpha = \beta\}, \{X : \alpha\}, \beta)$

$\mathit{infer}(c) = (\emptyset, \emptyset, \tau)$, where $\mathit{basetype}(c) = \tau$

$\mathit{infer}(f(t_1, \ldots, t_n)) = (\Theta_1 \cup \ldots \cup \Theta_n, \Gamma_1 \cup \ldots \cup \Gamma_n, f(\alpha_1, \ldots, \alpha_n))$,
where $(\Theta_i, \Gamma_i, \alpha_i) = \mathit{infer}(t_i)$

$\mathit{infer}(t_1 = t_2) = (S(\Theta\prime), \Gamma_1 \cup \Gamma_2, \tau)$,
where $(\Theta_1, \Gamma_1, \tau_1) = \mathit{infer}(t_1)$, $(\Theta_2, \Gamma_2, \tau_2) = \mathit{infer}(t_2)$ and $(\tau, \Theta\prime, S) = unify(\tau_1, \tau_2, \Theta_1 \cup \Theta_2)$

$\mathit{infer}(p(X_1, \ldots, X_n)) =$
$(\{\alpha_1 = \tau_p^1, \ldots, \alpha_n = \tau_p^n\}, \{X_1 : \alpha_1, \ldots, X_n : \alpha_n\}, \alpha_1 \times \ldots \times \alpha_n)$

$\mathit{infer}(cl_1 \vee cl_2) = (\Theta_1 \oplus \Theta_2, \Gamma_1 \cup \Gamma_2, \tau_1 + \tau_2)$,
where $(\Theta_1, \Gamma_1, \tau_1) = \mathit{infer}(cl_1)$ and $(\Theta_2, \Gamma_2, \tau_2) = \mathit{infer}(cl_2)$

$\mathit{infer}(p_1 \wedge p_2) = (\Theta_1 \otimes \Theta_2, \Gamma_1 \cup \Gamma_2, \tau_1 * \tau_2)$, where $(\Theta_1, \Gamma_1, \tau_1) = \mathit{infer}(p_1)$ and $(\Theta_2, \Gamma_2, \tau_2) = \mathit{infer}(p_2)$

$\mathit{infer}(h(\vec{X}) \leftarrow \mathit{body}) = (\Theta, \Gamma_X, \vec{\tau})$, where $(\Theta, \Gamma, \alpha) = \mathit{infer}(\mathit{body})$, such that $\{\vec{X} : \vec{\tau}\} \in \Gamma$ and $\Gamma_X = \Gamma \setminus \{\vec{X} : \vec{\tau}\}$

The initial type for variables is a generic type variable α representing all the possible terms, but during other steps of the algorithm, it may change, due to unification and to intersection. In the end, α may have a different definition, but note that $X : \alpha$ will still be true. This means that each logic variable will be attributed a type symbol only once and it will never change, all changes will happen only in Γ.

Example 21. Let `isList` be the predicate defined by:

`isList([ ]).`
`isList([X|Xs]) :- isList(Xs).`

The definition in *flatten form* is:

$\texttt{isList}(\texttt{X}_1) :- (\texttt{X}_1 = [\,]) \vee\ (\texttt{X}_1 = [\texttt{X}|\texttt{Xs}] \wedge \texttt{Xs} = \texttt{X}_2 \wedge\ \texttt{isList}(\texttt{X}_2))$.

The *infer* algorithm applied to this predicate will be as follows:

$infer(isList(X_1) :- (X_1 = [\,]) \ \vee\ (X_1 = [X|Xs] \ \wedge\ Xs = X_2 \ \wedge\ isList(X_2)).) =$
$infer((X_1 = [\,]) \vee\ (X_1 = [X|Xs] \ \wedge\ Xs = X_2 \ \wedge\ isList(X_2)).) =$
$infer(X_1 = [\,]) =$
$infer(X_1) = (\{\tau_1 = \alpha\}, \{X_1 : \tau_1\}, \tau_1)$
$infer([\,]) = (\emptyset, \emptyset, nil)$
$unify(\tau_1, nil, \{\tau_1 = \alpha\}) = (nil, \{\tau_1 = \alpha\}, \{(\alpha, nil)\})$
$(\{\tau_1 = nil\}, \{X_1 : \tau_1\}, nil)$
$infer((X_1 = [X|Xs] \ \wedge\ Xs = X_2 \ \wedge\ isList(X_2)) =$
$infer(X_1 = [X|Xs]) =$
$infer(X_1) = (\{\tau_1 = \alpha\}, \{X_1 : \tau_1\}, \tau_1)$
$infer([X|Xs]) =$
$infer(X) = (\{\tau_2 = \beta\}, \{X : \tau_2\}, \tau_2)$
$infer(Xs) = (\{\tau_3 = \gamma\}, \{Xs : \tau_3\}, \tau_3)$
$(\{\tau_2 = \beta, \tau_3 = \gamma\}, \{X : \tau_2, Xs : \tau_3\}, [\tau_2|\tau_3])$
$unify(\tau_1, [\tau_2|\tau_3], \{\tau_1 = \alpha, \tau_2 = \beta, \tau_3 = \gamma\}) = ([\tau_2|\tau_3], \{\tau_1 = \alpha, \tau_2 = \beta, \tau_3 = \gamma\}, \{(\alpha, [\tau_2|\tau_3]), (\beta, \beta_2), (\gamma, \gamma_2)\})$
$(\{\tau_1 = [\tau_2|\tau_3], \tau_2 = \beta_2, \tau_3 = \gamma_2\}, \{X_1 : \tau_1, X : \tau_2, Xs : \tau_3\}, [\tau_2|\tau_3])$
$infer(Xs = X_2) =$
$infer(Xs) = (\{\tau_3 = \gamma\}, \{Xs : \tau_3\}, \tau_3)$
$infer(X_2) = (\{\tau_1 = \delta\}, \{X_2 : \tau_1\}, \tau_1)$
$unify(\tau_3, \tau_1, \{\tau_3 = \gamma, \tau_1 = \delta\}) = (\tau_4, \{\tau_3 = \gamma, \tau_1 = \delta, \tau_4 = \gamma_2\}, \{(\gamma, \tau_4), (\delta, \tau_4)\})$
$(\{\tau_3 = \tau_4, \tau_1 = \tau_4, \tau_4 = \gamma_2\}, \{Xs : \tau_3, X_2 : \tau_1\}, \tau_4)$
$infer(isList(X_2)) =$
$infer(X_2) = (\{\tau_1 = \delta\}, \{X_2 : \tau_1\}, \tau_1)$
$(\{\tau_1 = \delta\}, \{X_2 : \tau_1\}, \tau_1)$
$(\{\tau_1 = [\tau_2|\tau_3], \tau_2 = \beta, \tau_3 = \tau_4, \tau_4 = \tau_1\}, \{X_1 : \tau_1, X : \tau_2, Xs : \tau_3, X_2 : \tau_1\}, [\tau_2|\tau_3] * \tau_4 * \tau_1)$
$(\{\tau_1 = nil + [\tau_2|\tau_3], \tau_2 = \beta, \tau_3 = \tau_4, \tau_4 = \tau_1\}, \{X_1 : \tau_1, X : \tau_2, Xs : \tau_3, X_2 : \tau_1\}, nil + ([\tau_2|\tau_3] * \tau_4 * \tau_1))$
$(\{\tau_1 = nil + [\tau_2|\tau_3], \tau_2 = \beta, \tau_3 = \tau_4, \tau_4 = \tau_1\}, \{X : \tau_2, Xs : \tau_3, X_2 : \tau_1\}, nil + [\tau_2|\tau_3])$

After simplification, the result will be $(\{\tau_1 = [\tau_2|\tau_1], \tau_2 = \beta\}, \{X : \tau_2, X_s : \tau_1, X_2 : \tau_1\}, \tau_1)$.

4.5.3 Closure Step

The last step in our algorithm is the closure of types. We perform the closure after all clauses of the definition of each predicate are analyzed and before a type is inferred for the next predicate.

Example 22. For instance the type for the *append* predicate arguments:

$$\tau^1_{append} = [\,] + [\alpha \mid \tau^1_{append}]$$
$$\tau^2_{append} = \beta$$
$$\tau^3_{append} = \beta + [\alpha \mid \tau^3_{append}]$$

will be closed, and the result will be:

$$\tau^1_{append} = [\,] + [\alpha \mid \tau^1_{append}]$$
$$\tau^2_{append} = [\,] + [\alpha \mid \tau^2_{append}]$$
$$\tau^3_{append} = [\,] + [\alpha \mid \tau^3_{append}]$$

Example 23. The types we infer for the predicate:

$\texttt{length}([], \texttt{0}).$
$\texttt{length}([\texttt{X}|\texttt{Xs}], \texttt{N}) : -\texttt{length}(\texttt{Xs}, \texttt{N1}), \texttt{N}\texttt{is}\texttt{N1} + \texttt{1}.$

are as follows:

$$\tau^1_{length} = [] + [\alpha \mid \tau^1_{length}]$$
$$\tau^2_{length} = int$$

Example 24. For the predicate *add* defined as follows:

$\texttt{add}(\texttt{0}, \texttt{X}, \texttt{X}).$
$\texttt{add}(\texttt{s}(\texttt{X}), \texttt{Y}, \texttt{s}(\texttt{Z})) : -\texttt{add}(\texttt{X}, \texttt{Y}, \texttt{Z}).$

The open types would be:

$$\tau^1_{add} = int + s(\tau^1_{add})$$

$$\tau^2_{add} = \alpha$$
$$\tau^3_{add} = \alpha + s(\tau^3_{add})$$

And when we apply the closure, we get:

$$\tau^1_{add} = int + s(\tau^1_{add})$$
$$\tau^2_{add} = int + s(\tau^2_{add})$$
$$\tau^3_{add} = int + s(\tau^3_{add})$$

Example 25. Let *flatten* be the predicate whose first argument is a nested list of lists and the second is the flat version of that nested list, defined as follows:

```
flatten([], []).
flatten([L|R], Flat) :- flatten(L, F1), flatten(R, F2), append(F1, F2, Flat).
flatten(L, [L]).
```

The output of our algorithm before closure is:

$$flatten = flatten.1 \times flatten.2$$
$$flatten.1 = [flatten.1 \mid flatten.1] + [\,] + x1.$$
$$flatten.2 = append.3 + [\,] + [x1].$$

The type for the first argument is open and we can see that with the closure, we will not have the case were the nested list has an integer. This problem can be solved by changing the predicate definition as follows:

```
flatten([], []).
flatten([L|R], Flat) :- flatten(L, F1), flatten(R, F2), append(F1, F2, Flat).
flatten(elem(L), [elem(L)]).
```

Now, the types are:

$$flatten = flatten.1 \times flatten.2$$
$$flatten.1 = [flatten.1 \mid flatten.1] + [\,] + elem(x1).$$
$$flatten.2 = append.3 + [\,] + [elem(x1)].$$

which are closed types.

The closure step ensures that the resulting types will be closed, so they will not have free variables or that if they have, then those variables are type constrained. This is

visible in the predicate $first$ that is defined as follows:

`first`((`X`, `Y`), `X`).

The types we get for that predicate arguments are:

$\tau^1_{first} = (\alpha, \beta)$.
$\tau^2_{first} = \alpha$.

Although the type for the second argument is a variable, the variable occurs in the type for the first argument and as such, the types are closed. On the other hand, a predicate `first` defined like this:

`first`(`X`, `Y`, `X`).

Is not accepted by our algorithm because now the second argument, Y, is a free variable with a single occurrence, which means it can be representing any term, so the type τ^2_{first} is open and it is not possible to close it, since we have no more constraints for that argument.

4.5.4 Soundness

Here we show that our algorithm is sound, i.e., infers types derived by our type system.

Theorem 1 (Soundness) *If* $infer(p) = (\Theta, \Gamma, \tau)$, *then there are* $\Theta_1 \supseteq \Theta$ *and* $\Gamma_1 \supseteq \Gamma$ *such that* $\Theta_1, \Gamma_1 \vdash p : \tau$

Proof: By structural induction on p.

Base cases:
VAR: $infer(X) = (\{\alpha = \beta\}, \{X : \alpha\}, \beta)$. Let $\Theta_1 = \Theta$ and $\Gamma_1 = \Gamma$, then $\Theta_1, \Gamma_1 \vdash X : \beta$.

CONS: $infer(c) = (\emptyset, \emptyset, \tau)$, where basetype(c) $= \tau$. Let $\Theta_1 = \Theta$ and $\Gamma_1 = \Gamma$, then $\Theta_1, \Gamma_1 \vdash c : \tau$, where basetype(c) $= \tau$.

Induction step:
TERM: $infer(f(t_1, \ldots, t_n)) = (\Theta_1 \cup \ldots \cup \Theta_n, \Gamma_1 \cup \ldots \cup \Gamma_n, f(\alpha_1, \ldots, \alpha_n))$.

By the induction hypothesis there are $\Theta_1\prime, \ldots, \Theta_n\prime, \Gamma_1\prime, \ldots, \Gamma_n\prime$ such for $1 \leq i \leq n$, $\Gamma_i\prime \supseteq \Gamma_i$ and $\Theta_i\prime \supseteq \Theta_i$ such that $\Theta_i\prime, \Gamma_i\prime \vdash t : \alpha_i$. Then, by the TERM rule, for $\Theta = \Theta_1\prime \cup \ldots \cup \Theta_n\prime$ and $\Gamma = \Gamma_1\prime \cup \ldots \cup \Gamma_n\prime$, it follows $\Theta, \Gamma \vdash f(t_1, \ldots, t_n) : f(\alpha_1, \ldots, \alpha_n)$.

UNIF: $\mathit{infer}(t_1 = t_2) = (S(\Theta), \Gamma_1 \cup \Gamma_2, \tau)$.
By the induction hypothesis, there are $\Theta\prime, \Gamma_1\prime$ and $\Gamma_2\prime$ such that $\Theta\prime \supseteq \Theta$, $\Gamma_1\prime \supseteq \Gamma_1$ and $\Gamma_2\prime \supseteq \Gamma_2$ such that $\Theta\prime, \Gamma_1\prime \vdash t_1 : \tau$ and $\Theta\prime, \Gamma_2 \vdash t_2 : \tau$. Then, by the rule UNIF, Proposition 1 and Proposition 2, for $\Theta_1 = S(\Theta)$ and $\Gamma = \Gamma_1 \cup \Gamma_2$, then $\Theta_1, \Gamma \vdash t_1 = t_2 : \tau$. (Note that according to the definition of unify, $S(\tau) = \tau$)

GOAL: $\mathit{infer}(p(X_1, \ldots, X_n)) = (\Theta = \{\alpha_1 = \tau_p^1, \ldots, \alpha_n = \tau_p^n\}, \Gamma = \{X_1 : \alpha_1, \ldots, X_n : \alpha_n\}, \alpha_1 \times \ldots \times \alpha_n))$.
By the induction hypothesis, there are $\Theta_1\prime, \ldots, \Theta_n\prime, \Gamma_1\prime, \ldots, \Gamma_n\prime$ such for $1 \leq i \leq n$, $\Gamma_i\prime \supseteq \Gamma_i$ and $\Theta_i\prime \supseteq \Theta_i$ such that $\Theta_i\prime, \Gamma_i\prime \vdash X_i : \alpha_i$. Then, by the GOAL rule, for $\Theta = \Theta_1\prime \cup \ldots \cup \Theta_n\prime$ and $\Gamma = \Gamma_1\prime \cup \ldots \cup \Gamma_n\prime$, it follows $\Theta, \Gamma \vdash p(X_1, \ldots, X_n) : \alpha_1 \times \cdots \times \alpha_n$.

CONJ: $\mathit{infer}(p_1 \wedge p_2) = (\Theta_1 \oplus \Theta_2, \Gamma_1 \cup \Gamma_2, \tau_1 * \tau_2)$.
By the induction hypothesis, there are $\Theta_1\prime, \Theta_2\prime, \Gamma_1\prime$ and $\Gamma_2\prime$, such that $\Theta_1\prime \supseteq \Theta_1$, $\Theta_2\prime \supseteq \Theta_2$, $\Gamma_1\prime \supseteq \Gamma_1$ and $\Gamma_2\prime \supseteq \Gamma_2$ such that $\Theta_1\prime, \Gamma_1\prime \vdash p_1 : \tau_1$ and $\Theta_2\prime, \Gamma_2\prime \vdash p_2 : \tau_2$. Then, by the CONJ rule, for $\Theta = \Theta_1\prime \oplus \Theta_2\prime$ and $\Gamma = \Gamma_1\prime \cup \Gamma_2\prime$, it follows $\Theta, \Gamma \vdash p_1 \wedge p_2 : \tau_1 * \tau_2$.

DISJ: $\mathit{infer}(p_1 \vee p_2) = (\Theta_1 \otimes \Theta_2, \Gamma_1 \cup \Gamma_2, \tau_1 + \tau_2)$.
By the induction hypothesis, there are $\Theta_1\prime, \Theta_2\prime, \Gamma_1\prime$ and $\Gamma_2\prime$, such that $\Theta_1\prime \supseteq \Theta_1$, $\Theta_2\prime \supseteq \Theta_2$, $\Gamma_1\prime \supseteq \Gamma_1$ and $\Gamma_2\prime \supseteq \Gamma_2$ such that $\Theta_1\prime, \Gamma_1\prime \vdash p_1 : \tau_1$ and $\Theta_2\prime, \Gamma_2\prime \vdash p_2 : \tau_2$. Then, by the DISJ rule, for $\Theta = \Theta_1\prime \otimes \Theta_2\prime$ and $\Gamma = \Gamma_1\prime \cup \Gamma_2\prime$, it follows $\Theta, \Gamma \vdash p_1 \vee p_2 : \tau_1 + \tau_2$.

CLS: $\mathit{infer}(h(\vec{X}) : -\mathit{body}) = (\Theta, \Gamma_X, \vec{\tau})$.
By the induction hypothesis, there are $\Theta\prime$ and $\Gamma\prime$, such that $\Theta\prime \supseteq \Theta$ and $\Gamma\prime \supseteq \Gamma$ for which $\Theta\prime, \Gamma\prime \vdash \mathit{body} : \tau_1$ and $\Theta\prime, \Gamma\prime \vdash h(\vec{X}) : \vec{\tau}$. Then, by the CLS rule, it follows $\Theta\prime, \Gamma\prime \vdash (h(\vec{X}) : -\mathit{body}) : \vec{\tau}$. $\square$

Chapter 5

Implementation

In this chapter, we give an overview of the implementation and practical examples of the results obtained by our implementation. We will also show some top level functions and explain the components that are called from those main functions.

5.1 Overview

The implementation of the type inference algorithm follows almost directly from the definition. In Figure 5.1 we can see the different modules of the algorithm.

The output of our algorithm represents type symbols that are not associated with a predicate argument as terms of the form $type(n)$, where n is an integer. The method we chose for printing the result is similar to how we represented types in our syntax during this work. A type for a predicate will be the tuple of the types for its arguments and the type for each argument will be defined bellow by a type equation, followed by any type symbols that may be used in the equations for that predicate.

5.2 Top-Level Predicates

Here we will now present the top-level predicate in the implementation of the type inference algorithm:

```
type_program(InputName, Types) :-
    open(InputName, read, InputStream),
```

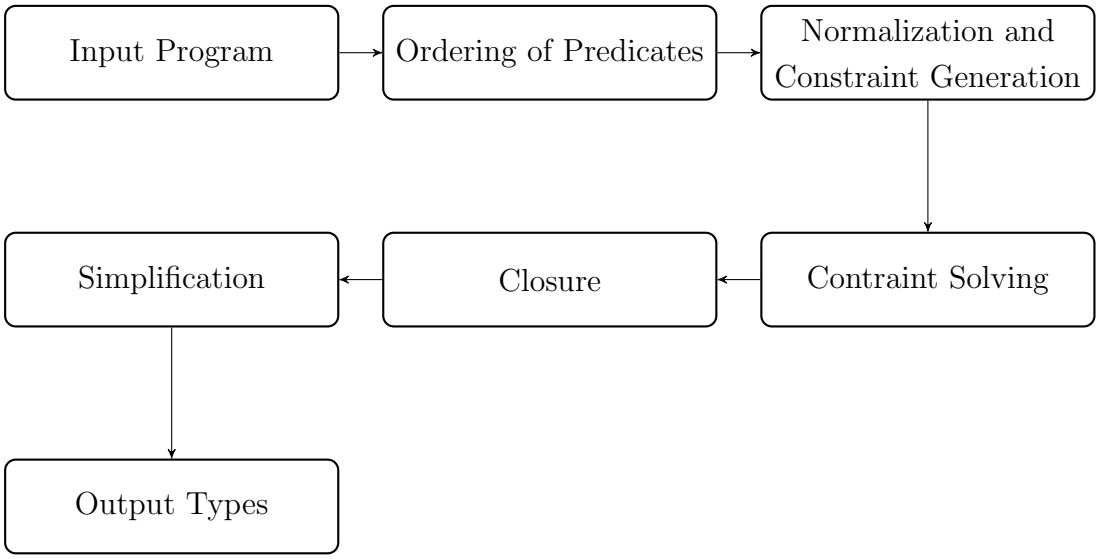


Figure 5.1: Modules of the type inference algorithm's implementation

```
type_program_input(InputStream, [], Rules),
order(Rules, Order),
typing(Rules, Order, [], Types),
close(InputStream),
print_aux_types(Types),
write('EndTypes!'), nl.
```

The predicate *open* reads the input file and gets our program, then the predicate *type_program_input* devides the program in several rules, each corresponding to a Prolog clause. The predicate *order* will find the order that the rest of the algorithm will follow according to the dependency graph of the program. After this, the *typing* predicate will do most of the work on the actual inference of types, including the normalization component as well as the constraint generation, constraint solving and the closure step. Then we use *close* to close the file we opened before and the predicate *print* prints the types that were not printed in the *typing* predicate.

We will know show more in detail the predicate *typing* since it does most of the work on the actual inference of types:

```
typing(_, [], Tf, Tf).
typing(Clauses, [Pred|Ps], Ac, Types) :-
    take_pred_only(Clauses, Pred, CP),
```

```
take_facts(CP, Facts, Rest),
build_types(Facts, [], Types1),
append(Ac, Types1, Types2),
type_clauses(Rest, Types2, AcNovo),
final_simplification(AcNovo, Types3),
closure(Types3, Types3, [], Types4),
pretty_printing(Types4, Pred),
typing(Clauses, Ps, Types4, Types).
```

It is clear now how the algorithm works, we first take the clauses that define the predicate on the current position of the ordering list, then we use the facts to immediately get a *summand* for the types of the arguments of the predicates and we infer types from the clauses that have bodies in *type_clauses*. In the end, we apply our closure heuristics and after that we just print the resulting types for that predicate. Here, we argue that taking the facts first instead of changing the definition of a predicate to *flatten form* does not change the result, since all constraints that come from facts on the *flatten form* will end up just adding a *summand* to the type of each argument, which is what we do in the first place.

5.3 Examples

We will now present the results we got from testing our implementation on some more predicates.

Example 26. Let P be the following program:

```
f(1).
h(1).
h(a).
p(X) :- f(X), h(X).
```

The output of our algorithm follows:

$f :: f.1$
$f.1 = num.$

$h :: h.1$

$h.1 = num + atom.$

$p :: p.1$
$p.1 = type(0).$

$type(0) = num.$

As we can see, we just substitute the τ as a symbol for the types and put the predicate name with its arity and position as symbol. It will be unique because even if two predicates have the same name, they will not have the same arity. If a predicate has more than one argument, the types for the predicate arguments are presented in order and the type for the predicate will have the types for the arguments separated by $\times$.

Example 27. Let P be a program that defines the predicate *append* as it has been defined in other places in this work. The resulting types from our implementation follow:

$append = append.1 \times append.2 \times append.3$
$append.1 = [x0 \mid append.1] + [\,].$
$append.2 = [x0 \mid append.2] + [\,].$
$append.3 = [x0 \mid append.3] + [\,].$

Our implementation deals with equality, inequality, and the $is/2$ predicate. For the inequality, both sides have type *num*. For the equality, the type for both terms must be equal, and for the attribution, the constraint is that the left-side argument has type *num*. We will improve the constraints obtained from attributions by including constraints for every variable with type *num*.

Example 28. Let `p` be the predicate defined as follows:

`p(0).`
`p(X) :- X1 is X - 1, p(X1).`

The types inferred are the following:

$p = p.1$
$p.1 = num.$

This is an example where we show how we deal with the constraints for the predicate $is/2$. The type num is attributed to $X1$ and that is the result we get for the type of the whole predicate argument, since 0 also has type num.

Chapter 6

Conclusions

On this thesis, we defined a type system for logic programming that accepts programs typed by closed types. We also defined a type inference algorithm that is sound according to the type system. Both the type system and the algorithm followed our definition of closed types that says types must be constrained in order for a program to be well-typed.

Our definition of closed types corresponds to what we understand by not too over-generous programs and the results we got from the tests performed on several programs are what we intended.

One of the advantages of our type system is that the types will have better approximation to terms that should be accepted by the program than open types, according to the programmer's intention.

Analyzing the results obtained by the implementation of our algorithm, we would like to point out some characteristics. First of all, we were very pleased with the results for the tests we performed as they matched the intended types for our definitions. Secondly, we think the results obtained are easy to understand and although they constrain the terms accepted by the program, we think that they are not too-restrictive.

In the future we will work on proving the type system is sound with respect to the logic programming semantics we defined and that the type inference algorithm is complete with respect to the type system. We also want to increase the number of typed Prolog built-ins.

Bibliography

[AH16] Sergio Antoy and Michael Hanus. Default rules for curry. *CoRR*, abs/1605.01352, 2016.

[AK91] H. Ait-Kaci, editor. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, Cambridge, MA, 1991.

[Bar92] H. P. Barendregt. Handbook of logic in computer science (vol. 2). chapter Lambda Calculi with Types, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

[BJ88] Maurice Bruynooghe and Gerda Janssens. An instance of abstract interpretation integrating type and mode inferencing. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*, pages 669–683, 1988.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007. release October, 12th 2007.

[DM82] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, 1982.

[DZ92] Philip W. Dart and Justin Zobel. A regular type language for logic programs. In *Types in Logic Programming*, pages 157–187. 1992.

[FD92] Mário Florido and Luís Damas. Types as theories. In *Proc. of post-conference workshop on Proofs and Types, Joint International Conference and Symposium on Logic Programming*, 1992.

[FSVY91] Thom W. Frühwirth, Ehud Y. Shapiro, Moshe Y. Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Proceedings of the Sixth*

Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991, pages 300–309, 1991.

[GdW94] John P. Gallagher and D. Andre de Waal. Fast and precise regular approximations of logic programs. In *Logic Programming, Proceedings of the Eleventh International Conference on Logic Programming, Santa Marherita Ligure, Italy, June 13-18, 1994*, pages 599–613, 1994.

[Had12] Spyros Hadjichristodoulou. A gradual polymorphic type system with subtyping for prolog. In *Technical Communications of the 28th International Conference on Logic Programming, ICLP 2012, September 4-8, 2012, Budapest, Hungary*, pages 451–457, 2012.

[Han13] Michael Hanus. Functional logic programming: From theory to curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, pages 123–168, 2013.

[HBC+11] Manuel V. Hermenegildo, Francisco Bueno, Manuel Carro, Pedro López-García, Rémy Haemmerlé, Edison Mera, José F. Morales, and Germán Puebla. An overview of the ciao system. In *Rule-Based Reasoning, Programming, and Applications - 5th International Symposium, RuleML 2011 - Europe, Barcelona, Spain, July 19-21, 2011. Proceedings*, page 2, 2011.

[HJ92] Nevin Heintze and Joxan Jaffar. Semantic types for logic programs. In *Types in Logic Programming*, pages 141–155. 1992.

[LC98] Lunjin Lu and John G. Cleary. On dart-zobel algorithm for testing regular type inclusion. *CoRR*, cs.LO/9810001, 1998.

[LR91] T. L. Lakshman and Uday S. Reddy. Typed prolog: A semantic reconstruction of the mycroft-o'keefe type system. In *Logic Programming, Proceedings of the 1991 International Symposium, San Diego, California, USA, Oct. 28 - Nov 1, 1991*, pages 202–217, 1991.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[Mis84] Prateek Mishra. Towards a Theory of Types in Prolog. In *International Logic Programming Symposium/International Symposium on Logic Programming/North American Conference on Logic Programming/Symposium on Logic Programming*, pages 289–298, 1984.

[MO84] Alan Mycroft and Richard A. O'Keefe. A polymorphic type system for prolog. *Artif. Intell.*, 23(3):295–307, 1984.

[Nai92] Lee Naish. Types and the intended meaning of logic programs. In *Types in Logic Programming*, pages 189–216. 1992.

[Pfe92] Frank Pfenning, editor. *Types in logic programming*. Logic programming. MIT Press, Cambridge, Mass., London, 1992.

[PR89] Changwoo Pyo and Uday S. Reddy. Inference of polymorphic types for logic programs. In *Logic Programming, Proceedings of the North American Conference 1989, Cleveland, Ohio, USA, October 16-20, 1989. 2 Volumes*, pages 1115–1132, 1989.

[SBG08] Tom Schrijvers, Maurice Bruynooghe, and John P. Gallagher. From monomorphic to polymorphic well-typings and beyond. In *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008, Valencia, Spain, July 17-18, 2008, Revised Selected Papers*, pages 152–167, 2008.

[SCWD08] Tom Schrijvers, Vítor Santos Costa, Jan Wielemaker, and Bart Demoen. Towards typed prolog. In *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, pages 693–697, 2008.

[SHC96] Zoltan Somogyi, Fergus Henderson, and Thomas C. Conway. The execution algorithm of mercury, an efficient purely declarative logic programming language. *J. Log. Program.*, 29(1-3):17–64, 1996.

[Ull88] Jeffrey D. Ullman. *Principles of Database and Knowledge-base Systems, Vol. I.* Computer Science Press, Inc., New York, NY, USA, 1988.

[UZ90] Jeffrey D. Ullman and Carlo Zaniolo. Deductive databases: Achievements and future directions. *SIGMOD Rec.*, 19(4):75–82, December 1990.

[YFS92] Eyal Yardeni, Thom W. Frühwirth, and Ehud Y. Shapiro. Polymorphically typed logic programs. In *Types in Logic Programming*, pages 63–90. 1992.

[Zob87] Justin Zobel. Derivation of polymorphic types for PROLOG programs. In *Logic Programming, Proceedings of the Fourth International Conference, Melbourne, Victoria, Australia, May 25-29, 1987 (2 Volumes)*, pages 817–838, 1987.