Type Assignment in Logic Programming

João Luís Alves Barbosa
Doutoramento em Ciência de Computadores
Departamento de Ciência de Computadores
2023

Orientador
António Mário da Silva Marcos Florido, Professor Associado,
Faculdade de Ciências da Universidade do Porto

Coorientador
Vítor Manuel da Silva Santos Costa, Professor Associado,
Faculdade de Ciências da Universidade do Porto


U.PORTO
FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Declaração de Honra

Eu, João Luís Alves Barbosa, inscrito(a) no Programa Doutoral em Ciência de Computadores da Faculdade de Ciências da Universidade do Porto declaro, nos termos do disposto na alínea a) do artigo 14.º do Código Ético de Conduta Académica da U.Porto, que o conteúdo da presente tese reflete as perspetivas, o trabalho de investigação e as minhas interpretações no momento da sua entrega.

Ao entregar esta tese, declaro, ainda, que a mesma é resultado do meu próprio trabalho de investigação e contém contributos que não foram utilizados previamente noutros trabalhos apresentados a esta ou outra instituição.

Mais declaro que todas as referências a outros autores respeitam escrupulosamente as regras da atribuição, encontrando-se devidamente citadas no corpo do texto e identificadas na secção de referências bibliográficas. Não são divulgados na presente tese quaisquer conteúdos cuja reprodução esteja vedada por direitos de autor.

Tenho consciência de que a prática de plágio e auto-plágio constitui um ilícito académico.

Porto, 23 de Janeiro de 2023

Agradecimentos

Em primeiro lugar quero agradecer ao meu supervisor, Mário Florido, e co-supervisor, Vítor Santos Costa, pelo tempo dedicado a discussões essenciais à concretização desta tese e pela ajuda e orientação proporcionada durante todos estes anos. A investigação teórica não é particularmente fácil, com muitos passos para a frente e passos para trás, e foi a enorme quantidade de horas dedicadas a discussões e reuniões, muitas vezes cansativas, mas sempre produtivas, que me fez continuar em frente e concluir este trabalho. Agradeço em particular ao Professor Florido por se ter apercebido do meu potencial desde cedo, uma pessoa vinda duma área bastante diferente, e me ter guiado no mundo da investigação teórica ao longo de tantos anos.

Quero também agradecer ao meu parceiro Jorge, por partilhar a sua vida comigo, por me apoiar e por fazer com que tudo vá pelo melhor. Não há palavras para agradecer tudo o que faz por mim.

Preciso de agradecer muito aos meus pais, Luís e Dina, por me terem apoiado nas minhas escolhas, por me terem propocionado tudo o que precisei para chegar a este ponto na minha vida e carreira, e por me terem aguentado em momentos mais difíceis. No final de contas sou o fruto do seu trabalho como pais e da sua dedicação a mim como pessoa. Também quero agradecer à minha irmã Mariana, cunhado Renato, tios e tias, avós e Lurdes, por me terem ajudado a crescer e ser quem sou, sendo sempre uma fonte de apoio quando foram precisos.

Quero agradecer aos meus amigos Gonçalo e Rui, por terem revisto esta tese, e por serem excelentes amigos com quem ter discussões interessantes, que me fizeram ter melhores valores e ser melhor pessoa em geral.

Finalmente quero agradecer aos meus amigos, especialmente à Flávia, ao João Paulo e à Joana, por me terem proporciado momentos tão bonitos na minha vida, que a tornaram mais agradável e divertida.

Todo este trabalho não teria sido possível se esse lado da minha vida não existisse. Toda a amizade e amor que recebi do meu namorado, família e amigos são o que guardo de mais querido.

Este trabalho foi apoiado financeiramente pela agência portuguesa de financiamento, FCT — Fundação para a Ciência e Tecnologia, através da Bolsa de Doutoramento SFRH/BD/136018/2018.

FCT Fundação para a Ciência e a Tecnologia

UNIVERSIDADE DO PORTO

Abstract

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

Doctor of Philosophy in Computer Science

Type Assignment in Logic Programming

by João Luís Alves Barbosa

The goal of this thesis is to describe a type discipline for logic programming, where one can dynamically and statically define well-typed programs, automatically infer useful type information, and use it to detect type errors.

Types play an important role in the verification and debugging of programming languages, and have been the subject of significant research in the logic programming community. However, most Prolog compilers usually do not include any formal static type verification, and the dynamic type verification for some built-in predicates lacks a typed semantics as basis.

The following are the contributions of this work:

- a three-valued declarative semantics for logic programming, where the third value corresponds to a type error;
- a typed operational semantics for logic programming, where we define exactly what is meant by a type error in a program and in a query, and show how the semantics can dynamically detect type errors;
- a type system that defines which programs are well-typed with which types through type rules, and a proof that the type system is sound;
- a type inference algorithm that, through type constraint generation followed by constraint solving, automatically infers types from programs, and a proof that the algorithm is sound;
- a *closure* operation that given input types returns closed types that are instances of the input types, and that are closer to the programmer's intention.

UNIVERSIDADE DO PORTO

Resumo

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto

Doutoramento em Ciência de Computadores

por João Luís Alves Barbosa

O objetivo desta tese é definir uma disciplina de tipos para programação em lógica, onde se pode definir, tanto estatica como dinamicamente, programas bem tipados, inferir tipos úteis e usar essa informação para detetar erros de tipos.

Tipos são essenciais na verificação e desenvolvimento de programas em várias linguagens de programação. Por isso, foram o alvo de bastante investigação na comunidade de programção em lógica. Contudo, a maioria dos compiladores de Prolog não inclui nenhuma verificação de tipos estática e a verificação de tipos dinâmica para alguns built-ins não tem base semântica.

As contribuições desta tese são as seguintes:

- uma semântica declarativa de três valores para programação em lógica, onde o terceiro valor corresponde a um erro de tipos;
- uma semântica operacional tipada para programação em lógica, onde definimos exatamente o que corresponde a um erro de tipos num programa e numa chamada ao programa, e mostramos como a semântica pode ser usada para detetar erros de tipos;
- um sistema de tipos que define que programas estão bem tipados e com que tipos e uma prova de que o sistema está correto;
- um algoritmo de inferência que, através da geração e resolução de restrições de tipos, automaticamente infere tipos para programas, assim como uma prova de que o algoritmo está correto;
- uma operação de *fecho* que dados tipos de input retorna tipos fechados que são instâncias dos tipos originais mais próximos da intenção do programador.

List of Publications

This work originated the following publications:

[1] João Barbosa, Mário Florido, & Vítor Santos Costa. *Closed types for logic programming.* In 25th Int. Workshop on Functional and Logic Programming (WFLP) (2017).

[2] João Barbosa, Mário Florido, & Vítor Santos Costa. *A three-valued semantics for typed logic programming.* In Proceedings 35th International Conference on Logic Programming (Technical Communications), ICLP 2019 Technical Communications, Las Cruces, NM, USA, September 20-25, 2019, vol. 306 of EPTCS, 36–51 (2019).

[3] João Barbosa, Mário Florido, & Vítor Santos Costa. *Data type inference for logic programming.* In De Angelis, E. & Vanhoof, W. (eds.) Logic-Based Program Synthesis and Transformation, LOSPTR 2021, 16–37 (Springer International Publishing, Cham, 2022).

[4] João Barbosa, Mário Florido, & Vítor Santos Costa. *Typed SLD-Resolution: Dynamic typing for logic programming.* In Villanueva, A. (ed.) Logic-Based Program Synthesis and Transformation, LOSPTR 2022, 123–141 (Springer International Publishing, Cham, 2022).

[5] Philipp Körner, João Barbosa *et al. Fifty years of prolog and beyond.* Theory and Practice of Logic Programming 1–83 (2022).

Contents

List of Figures

List of Abbreviations

SLD	**S**elective **L**inear **D**efinite
MGU	**M**ost **G**eneral **U**nifier
CAS	**C**omputed **A**nswer **S**ubstitution
FOL	**F**irst **O**rder **L**ogic
MM	**M**artelli and **M**ontanari
TSLD	**T**yped **S**elective **L**inear **D**efinite
TDC	**T**uple **D**istributive **C**losure
SCC	**S**trongly **C**onnected **C**omponent

Chapter 1

Introduction

Types play an important role in the verification and debugging of programming languages, and have been the subject of significant research in the logic programming community [2, 3, 6–16]. Most research has been driven by the desire to perform compile-time type checking.

One important line of this work views types as approximation of the program semantics [6, 7, 9, 10, 17]. This kind of approaches use descriptive types, which are types that try to, in some way, describe a program property. In order to describe the semantics of a program, tuple distributive closures of regular types are usually used [6, 7, 9, 10, 17, 18]. Regular types are types that can be described by regular term grammars [6], and they have decidable intersection, union, subset, and unification operations. The tuple distributive closure assures that type inference is decidable [8]. However, type inference in these approaches sometimes results in overly broad and uninteresting types, that end up not being useful for type error detection. This is mostly due to the way logic programmers write their programs.

Example 1: Let *append* be the predicate where the the third argument is a list corresponding to the concatenation of the lists in the first two arguments, defined traditionally. The following example shows its types as an approximation of the program semantics. Let `ti` be the type of the *i-th* argument of `append`, "+" mean type disjunction and "A" and "B" be type variables:

```
t1 = [] + [A | t1]
t2 = B
t3 = B + [A | t3]
```

The goal in this previous line of research was either to infer types from programs that are over approximations of the program semantics - type inference [6, 7, 17, 19–22]-, or to verify, given a program and types, that the types are an over approximation of the program semantics - type verification [23, 24].

Another type of approach used prescriptive types as in Mycroft and O'Keefe type system [11], later reinterpreted by Lakshman and Reddy [12]. Some of the advantages come from having a more strict type discipline, therefore more type errors can be detected at compile-time, and type information is much closer to the programmer's intention as defined by Naish [25]. These approaches, however, tend to limit the expressiveness of logic programs. This limitation of expressive power can be a turn off for a class of programmers that wants to use logic programming for real world applications which require some complex properties that were lost.

There was also an approach to infer well-typings of a program by Schrijvers, Bruynooghe, and Gallagher [26, 27]. These types do not describe the success set of predicates, but instead describe conditions for which the program succeeds. This type information was used in termination analysis [27].

In practice, static type-checking is not widely used in actual Prolog systems, with the notable exception of the CIAO-Prolog system that uses an assertion language in order to provide pre and post type conditions, that are verified at compile-time and run-time [28–33]. A more in depth survey on all Prolog implementations and the type information that each one uses can be found in [5] and in Section 2.3.

Note that, Prolog systems do rely on dynamic type checking to ensure that system built-in parameters are called with acceptable arguments, such as *is/2*. In fact, the Prolog ISO standard defines a set of predefined types and typing violations [34].

Dynamic type checking has the major disadvantage that the program needs to run in order to detect type errors. In contrast, static typing allows for the static detection of type errors, which in turn allows for a faster development of programs and a bigger confidence in the program correctness.

1.1 Contributions

In the type discipline we describe in this thesis, we allow for the dynamic and static detection of type errors. The goal is to define a semantically sound type discipline that pragmatically is able to infer useful type information. With this goal in mind, we have the following contributions.

Three-Valued Semantics

Our first contribution is a three-valued semantics where the third value, *wrong*, corresponds to a type error. The semantics of a logic program given some interpretation for symbols is no longer either *true* or $false$, but it can also be *wrong*.

Example 2: One example of a program that has a *wrong* semantic value, and therefore a type error, assuming a different type for *integers* and *atoms*, is:

```
p(1).

q(a).

r(X) :- p(X), q(X).
```

Example 3: One example of a program that does not have a *wrong* semantic value, even though the result for any query for predicate r is $false$:

```
p(1).

q(2).

r(X) :- p(X), q(X).
```

Note that in the first example one unifies terms of different types, while in the second example unification is applied to terms of the same type.

Dynamic Typing

Our second contribution is a dynamic type checker. We extend the Martelli-Montanari unification algorithm [35] to be interpreted in a three-valued logic called Weak Kleene Logic, where the third value *wrong* corresponds to a type error. After that, we define a new operational semantics for logic programming called TSLD-resolution, based on SLD-resolution, which is the *de facto* operational semantics of logic programming, extending it with a typed unification algorithm. In this semantics, we define what is a type error in a program and what is a type error in a query. Moreover, we prove that TSLD-resolution is sound with respect to the three-valued declarative semantics. This work was published in [4].

Static Typing

We then present a type system that describes which programs are well-typed. Firstly, we present a type language that corresponds to regular types that uses type symbols defined by (possibly recursive) type definitions. We provide a semantics for types, both ground and polymorphic. We then present the rules of our type system, which define which programs are well-typed and with which types. The type system uses several important auxiliary

definitions, such as subtyping and type equivalence. Finally, we prove that the type system is sound with respect to the semantics, in the sense that, for proper interpretations with tuple distributive domains, if there is a type derivation in the type system, then the types are semantically correct.

This work was published in [2] and slightly modified for presentation in this thesis.

Type Inference

We also present a type inference algorithm that given a program infers types that can be derived by the type system. This algorithm uses type constraint generation followed by constraint solving. We prove termination, and correctness of the algorithm with respect to the type system. This work was published in [3] and was modified for presentation in this thesis.

Through this thesis, we follow the seminal Milner's work [36] on types for programming languages, where we have:

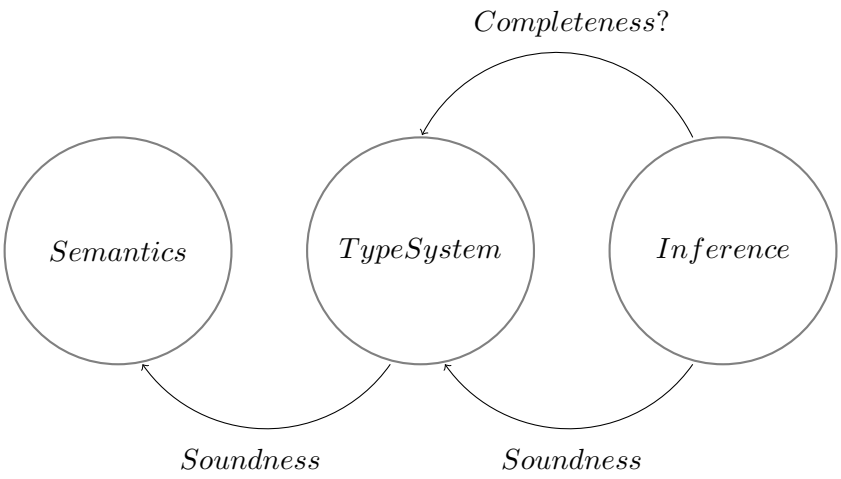


FIGURE 1.1: An overview of our approach

Closed Types

Finally, we define a heuristic that can be applied to the types resulting from type inference in order to get types that are closer to the programmer's intention in the sense defined by Naish in [25]. Although it is a heuristic, it is based on some fundamental principles. We provide some examples of how the application of the algorithm can improve the resulting types. Furthermore, we introduce the concept of data type definitions and show how they can be added to programs to considerably improve type inference. This notion is based on data definitions in functional programming and drastically reduces the extra amount of work that is involved in providing type signatures for every predicate. We also show the results we got on some example programs.

This work was originally published in [1], and extended for this thesis.

Both the type inference algorithm and the closure operation were implemented in Prolog and are available in

https://github.com/JoaoLBarbosa/TypeInferenceAlgorithm

1.2 Outline

The rest of this thesis is organized as follows:

Chapter 2: Background

In Chapter 2, we provide the basis for the rest of this thesis. We provide a brief description of logic programming syntax and semantics, both declarative and operational. We also provide a detailed description of the most influential approaches to introduce types in logic programming, including prescriptive types, descriptive types, well-typings, amongst others. Finally, we compare other approaches to the one presented in the rest of this thesis, including parts that are similar, but most importantly where they differ.

Chapter 3: Dynamic Typing

In Chapter 3, we present a typed operational semantics which uses a novel typed unification algorithm, based on a three-valued logic that uses type information in order to unify terms.

We then describe how to dynamically detect type errors in programs, and in queries. We also prove the correctness of the operational semantics with respect to a typed declarative semantics.

Chapter 4: Static Typing

In Chapter 4, we present present a type system that describes which programs are well-typed. We prove correctness of the type system with respect to a three-valued declarative semantics. The types used in the type system are tuple distributive closures of regular types.

Chapter 5: Type Inference

In Chapter 5, we present a type inference algorithm that uses type constraint generation followed by constraint solving in order to automatically infer types for logic programs. We prove that our algorithm is sound in the sense that types inferred by the algorithm can be derived in the type system.

Both in this chapter and in the previous one we provide extensions to the type system and the algorithm to deal with arithmetic built-ins that are common in most Prolog compilers.

Chapter 6: Closed Types

In Chapter 6, we describe a heuristic, the *closure operation*, that is applied to the types resulting from type inference, in order to solve the problem of overly-broad types, and show the results we got from applying it to several examples. We also describe how we enable optional data type declarations, not in the form of type signatures for predicates, but on the form of data type definitions that can be used during type inference, and show some interesting results we got.

Chapter 7: Conclusions and Further Work

In Chapter 7, we summarize our results, and discuss some of the limitations of dynamic type checking, type inference, and the closure operation. We then provide several possible lines of future work, based on the work presented in this thesis.

Chapter 2

Background

Logic programming is a programming paradigm with a long history. In logic programming, programs are represented by Horn clauses, and Selective Linear Definite(SLD)-resolution is used as a computational procedure to execute such programs [37–39]. One of the advantages of logic programming is its declarative nature that provides an easy semantic interpretation of logic programs. This semantics can be described operationally, in terms of *how* programs compute answers, or declaratively, in terms of *what* answers programs compute [40]. The SLD-resolution, chosen as the *de facto* operational semantics for logic programming, is proven sound, and complete for a large class of programs.

On the other hand, types are one of the key components in most programming languages. They allow for control, documentation, and program verification. Logic programming was initially thought of and designed without types, but, since then, several authors have tried to introduce types in logic programming to take advantage of the huge benefits they could provide in the verification and debugging of programs.

In this chapter, we will describe the basics of syntax and semantics of logic programming, as well as give a walk-through several previous works on types for logic programming languages.

2.1 Logic Programming

The way programs are represented in logic programming is through Horn clauses. Queries, which are calls to programs, are represented by logic formulas. The goal of computation, via SLD-resolution, in a logic program, given a query, is to find a refutation to the query, by finding a substitution to the variables in it such that the query is a logical consequence of the program.

This computation, via resolution, can have several obstacles, such as non-termination, inefficient code, *etc.* In this section, we will not worry with the implementation of the concepts but instead give the formal definitions behind them, in order to present the work that has been done.

2.1.1 Syntax of Logic Programming

Logic programs assume an alphabet composed of symbols from disjoint classes. We assume an infinite set of variables **Var**, an infinite set of function symbols **Fun**, parenthesis and the comma. Using this alphabet we are able to define terms [40, 41].

Terms are defined as follows:

- a variable is a term,
- if f is an n-ary function symbol and $t_1, \ldots, t_n$ are all terms, then $f(t_1, \ldots, t_n)$ is a term,
- if f is a function symbol of arity zero, then f is a term and it is called a constant.

We will call *ground terms* to terms that have no variables, and *complex terms* to terms that start with a function symbol of arity > 1.

We can now extend this alphabet in order to create a language for programs. We start by adding an infinite set of predicate symbols **Pred** and the reverse implication symbol $\leftarrow$ [40, 41]. The definition of atoms, queries and programs is as follows:

- an atom is a predicate symbol p associated with an arity n, applied to terms $t_1, \ldots, t_n$, which we write as $p(t_1, \ldots, t_n)$;
- a query is a finite conjunction of atoms;
- a clause is of the form $H \leftarrow \bar{B}$, where H is an atom and $\bar{B}$ is a query;
- a program is a finite set of clauses, which we will represent by P.

All variables in clauses are universally quantified while all variables in queries are existentially quantified.

Example 4: The following is the definition of a common logic program, called `append` that represents the concatenation of two lists. We replaced $\leftarrow$ with `:-` for implication, since it is the notation used in logic programming languages. The meaning remains the same.

```
append([ ], X, X).
append([H|T], Y, [H|Z]) :- append(T, Y, Z).
```

Throughout the rest of this thesis, whenever we refer to the predicate `append`, we are referring to this definition.

2.1.2 Semantics of Logic Programming

Here we will give a brief overview of the semantics of logic programs. For a detailed description, the reader may use [40, 41].

Operational Semantics

Logic programming uses logic formulas as representation of programs and a refutation procedure, based on the resolution inference rule, for computation. This resolution method is the SLD-resolution, first proposed by Kowalski [42].

The main computational mechanism at work in SLD-resolution is *unification*. Unification between two logic terms consists of finding a substitution such that when applied to both terms makes them equal. A substitution is a set of pairs of the form $X \mapsto t$, and it is interpreted as variable X being replaced by term t. The most used unification algorithms were initially defined by Robinson [43], and later redefined by Martelli and Montanari [35]. Both these algorithms output most general unifiers (MGU) of the terms they are applied to. Most general unifiers of two terms are such that every other unifier is less general, *i.e.*, either a renaming or an instance.

The main goal of SLD-resolution is, given a program P and a query Q, to find a *computed answer substitution* (CAS) θ, *i.e.*, a substitution for the variables in Q, such that $\theta(Q)$ is a logical consequence of the program.

For this we will apply a SLD-derivation step several times until we reach the empty query.

Definition 1 - SLD-derivation step: Given a program P and a query $Q = A_1, \ldots, A_n$, we select an atom A_i from Q, and a clause $H \leftarrow B_1, \ldots, B_m$ from P. Suppose A_i and H unify and let θ be an MGU. Then we can replace A_i by $B_1, \ldots, B_m$ in Q and apply θ to the query. We will represent an SLD-derivation step by $A_1, \ldots, A_i, \ldots, A_n \implies \theta(A_1, \ldots, B_1, \ldots, B_m, \ldots, A_m)$.

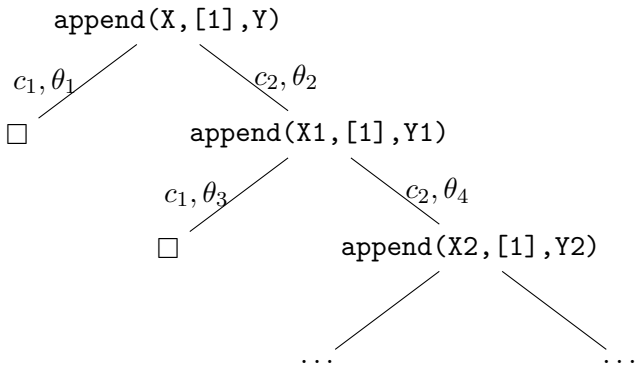
Now an SLD-derivation corresponds to successive applications of the SLD-derivation step.

There are some choices we need to make for applying each SLD-derivation step. We need to choose the selected atom from the query, the selected clause from the program, the MGU, and the renaming of the clause that may need to be performed to make sure that the clause and the selected atom share no variables. It is proven that the choice of the selected atom does not change the success of an SLD-derivation [40]. Also, the choice of MGU is not important so as long as it is idempotent [40], and the choice of renaming

is irrelevant [40]. The choice of the selected clause from the program gives rise to the concept of an SLD-tree.

An SLD-tree is a tree where the root is the query Q, and each child of each node corresponds to a different clause applied in the SLD-derivation step for the selected atom. Note that here we assume a fixed selection rule of an atom in Q, since the results are independent from the selected atom.

Example 5: Let us see the SLD-tree for a program consisting of the `append` predicate and query `append(X,[1],Y)`. We will call c_1 the first clause in the program, and c_2 the second.



The labels θ_i in the SLD-tree correspond to the MGU at each step: $\theta_1 = \{X \mapsto [\,], Y \mapsto [1]\}$, $\theta_2 = \{X \mapsto [H \mid X1], Y \mapsto [H \mid Y1]\}$, $\theta_3 = \{X \mapsto [H], Y \mapsto [H,\ 1]\}$, and $\theta_4 = \{X \mapsto [H \mid [H1 \mid X2]], Y \mapsto [H \mid [H1 \mid Y2]]\}$. If the MGU labels a branch that ends in success, it is the CAS.

As we can see, this SLD-tree is successful an infinite number of times, corresponding to the different possibilities of lists we can append the list [1], in order to obtain another list.

Declarative Semantics

Since logic programming uses logic formulas, one can reason about the truth values of those formulas. First, we need to give meaning to the symbols of the program. To every function symbol we associate a mapping from terms to terms, such that each complex term builds a semantic value that corresponds to a tree, where the root is the function symbol that starts the term, and the children are the values of the terms in the tuple. To every predicate symbol we associate a set of accepted tuples of semantic values, corresponding to the tuples in the relation represented by the predicate symbol.

Then, an interpretation can be seen as a particular set of accepted tuples of terms for each predicate symbol occurring in a program. Some of these interpretations can result in clauses having the value true, and other clauses having the value false.

If an *interpretation* I is such that for any substitution for variables in a program, every clause in the program has the value true, then that interpretation is called a *model* of the program.

There are several interpretations that are models of a program P, and it is proven in [40, 41] that the intersection of these interpretations is the minimal model of the program.

Whenever all models of the set of formulas P are also models of the set of formulas Q, we say that $P \models Q$.

We can then use the definition of models to prove the soundness of SLD-resolution. The theorem is as follows [40, 41].

Theorem 1 - Soundness of SLD-resolution: Given a program P and a query Q. If there is a successful derivation of $P \cup \{Q\}$ with CAS θ, then $P \models \theta(Q)$.

SLD-resolution is also proven complete for a large set of programs.

Theorem 2 - Strong Completeness of SLD-resolution: Given a program P, a query Q, and a substitution θ suppose that $P \models \theta(Q)$. Then for every selection rule there is a successful SLD-derivation of $P \cup \{Q\}$ with CAS η, such that $\eta(Q)$ is more general than $\theta(Q)$.

Fix-point Semantics

The immediate consequence operator T_P [40, 41] takes a set of atoms and calculates the consequence of such set of atoms in the program P. The application of this operator n number of times is represented by $T_P \uparrow n$. We represent by $T_P \uparrow \omega(\emptyset)$ the least fix-point of the T_P operator. It is also proven that $T_P \uparrow \omega(\emptyset)$ corresponds to the minimal model for the program P [40, 41].

This completes the classical semantics of logic programming, both operational and declarative. There has also been some work done on a denotational semantics for logic programming [44], where clauses are interpreted as functions from substitutions to substitutions. This interpretation was proven equivalent to the SLD-resolution and was used to interpret Prolog programs, arguing that the traditional declarative, model-theoretical, semantics is not appropriate to reason about the behavior of Prolog programs.

2.2 Types in Logic Programming

Types are used in programming languages for several reasons. Besides improving documentation of programs and making programs clearer to read

and develop, they also provide some confidence on the program correctness, help in termination analysis, and in the process of debugging during program development.

Logic programming was initially defined as an untyped programming paradigm and Prolog, in particular, was designed with only one type in mind, the *term*. For this reason, pure Prolog has no type checking or type information.

Due to the utility provided by types in program development and verification, several authors have proposed type disciplines for logic programming. Some of these approaches are based on strict type disciplines and use *prescriptive types*, where they become part of the syntax and semantics of programs, and restrict the class of programs accepted by the language to those well-typed. Other approaches follow a *descriptive approach* where types are properties that follow from the program and describe its semantics.

2.2.1 Type Language

One of the first steps towards introducing types into an untyped programming language is to define the language of types. Several authors [6, 8, 9, 20, 22, 23, 45, 46] use *regular types* as the language of types in logic programming. Regular types can be described by regular term grammars [6, 20, 47], and comparison and intersection of regular types is decidable. Regular types may be tuple distributive [6, 8, 20] and can also be described by regular unary logic programs [9, 10, 22, 23, 45], meaning they can be translated into logic programs and integrated easily in the program itself for type verification at run-time.

Tuple distributive types are types for which the following condition holds:

Definition 2 - Tuple Distributive Types: Let τ be a type, containing the set of semantic values S. We say that τ is a tuple distributive type if for every pair $f(t_1^1, \ldots, t_n^1) \in S$ and $f(t_1^2, \ldots, t_n^2) \in S$, we know that $S \supseteq \{f(t_1^{i_1}, \ldots, t_n^{i_n}) \mid 1 \leq i_1, \ldots, i_n \leq 2\}$.

It is easier to understand the definition through an example.

Example 6: Let τ be a tuple distributive type containing the values $f(a, 1)$ and $f(2, b)$. Then we know that τ also contains $f(a, b)$ and $f(2, 1)$.

Types with this property were first described by Mishra in [20] and are called cartesian closed types. Zobel included this property as a condition for types to be deterministic [6]. Other authors [9, 45] also use tuple-distributive regular types.

Some other authors, such as Naish, define types as arbitrary sets of terms [24] that can be described by full logic programming. This is different from approaches that use regular unary logic programs, since it includes a larger class of programs, and therefore, of types.

Mycroft and O'Keefe [11,12] described a type language, where types can be formed using type variables and type constructors with rank (arity) ≥ 0. This approach is closer to functional programming types.

2.2.2 Well-Typedness

After a type language has been defined, the goals of introducing types in logic programming can be several.

Prescriptive Types

In a prescriptive system, types are part of the semantics of programs and, therefore, well-typed programs correspond to programs that are well-formed with respect to the type rules defined by the system. This is the case of Mycroft and O'Keefe system [11]. There, a polymorphic type system is described for Prolog. For this, type declarations are added to the language, where types are declared for function symbols (including constants), predicate symbols and variables. These declarations are built using type variables and type constructors. Then, a definition of well-typed programs is given, based on the well-formedness of programs with respect to the types of each symbol.

The lack of semantics for this approach was later solved by Lakshman and Reddy [12]. There, the authors redefine the language of well-typed programs and give the semantics for this language, which they call Typed Prolog. Typed Prolog is a prescriptive typed language, where a type system characterizes well-formed Typed Prolog programs. Then, both a typed model-theoretic semantics and a fix-point semantics are described for Typed Prolog, and they are proven equivalent.

In Typed Prolog, the language of programs for logic programming is extended with an infinite set of type constructors $\mathbf{T}$, where every type constructor is associated with an arity, and an infinite set of type variables $\mathbf{\Lambda}$. Then, given a finite set of unique type assertions to variables Γ, the type rules are in Figure 2.1, taken from [12]. A well-formed expression can be derived from these rules.

Example 7: Consider the following alphabets:

$$T = \{list^1\}$$

$$F = \{nil : list(\alpha),\ [\ |\ ] : \alpha \times list(\alpha) \rightarrow list(\alpha)\}$$

$$\Gamma \vdash X : \tau \qquad \text{if } (X : \tau) \in \Gamma$$

$$\frac{\Gamma \vdash t_1 : \theta(\tau_1) \quad \ldots \quad \Gamma \vdash t_k : \theta(\tau_k)}{\Gamma \vdash f(t_1, \ldots, t_k) : \theta(\tau\prime)} \qquad \text{if } f : \tau_1 \times \cdots \times \tau_k \to \tau\prime$$

$$\frac{\Gamma \vdash t_1 : \theta(\tau_1) \quad \ldots \quad \Gamma \vdash t_k : \theta(\tau_k)}{\Gamma \vdash p(t_1, \ldots, t_k) Atom} \qquad \text{if } p : Pred(\tau_1 \times \cdots \times \tau_k)$$

$$\Gamma \vdash \epsilon \; Formula$$

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

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

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

$$\frac{\Gamma \vdash t_1 : \theta(\pi_1) \quad \ldots \quad \Gamma \vdash t_k : \theta(\pi_k) \quad \Gamma \vdash \phi \; Formula}{\vdash [\forall X_1 : \tau_1, \ldots, X_n : \tau_n](p(t_1, \ldots, t_k) : -\phi) \; Clause} \qquad \begin{array}{l} \text{if } \Gamma = \{X_1 : \tau_1, \ldots, X_n : \tau_n\} \\ p : Pred(\pi_1 \times \cdots \times \pi_k), \text{ and} \\ \theta \text{ is a renaming substitution} \end{array}$$

$$\frac{\vdash C_1 \; Clause \quad \ldots \quad \vdash C_2 \; Clause}{(C_1 \ldots C_2) \; Program}$$

FIGURE 2.1: Type Rules for Typed Prolog

$$P = \{append : Pred(list(\alpha) \times list(\alpha) \times list(\alpha))\}$$

Then, the following is a version of a well-formed Typed Prolog program, defining `append`:

$$[\forall Y : list(\alpha)] \quad append(nil, Y, Y) \leftarrow$$
$$[\forall A : \alpha, X : list(\alpha), Y : list(\alpha), Z : list(\alpha)]$$
$$append([A|X], Y, [A|Z]]) \leftarrow append(X, Y, Z).$$

It is obviously bothersome for the programmer to declare types for every single symbol in the program, so type reconstruction can be performed if part of the declarations for variables and predicates are missing [12].

One big limitation of this type system is the fact that predicate symbols only have one type, and as we can see in the rule for a clause, every occurrence of a predicate in the head of a clause must be equivalent to the type of the predicate, up to renaming. The authors call this restriction *definitional genericity*. This limitation prevents predicates to work on more than one type as illustrated in the following example from [12].

Example 8: Given the alphabets $T = \{man^0, woman^0\}$, $F = \{john : man, mary : woman\}$, and $P = \{married : Pred(man \times woman)\}$, the following is a well-typed program:

```
married(john, mary).
```

However, we can never define a predicate that works on man and woman at the same time, for example:

```
person(john).
person(mary).
```

since now the type signature for this predicate needs to include a sum of the two types, which is not allowed as it goes against the definitional genericity restriction.

Other logic programming and functional logic programming languages use prescriptive types [48–50]. Mostly all of them are based on similar concepts as the approach presented above, with more expressive type constructors based on functional programming language such as Haskell-style data declarations.

Descriptive Types

In descriptive systems for logic programming, well-typedness is usually more difficult to describe. For instance, if we assume that the type of a predicate is the set of tuples of terms for which the predicate succeeds, what does it mean for a program to be well-typed?

Several authors assume that the only way to know is for the programmer to declare the types for the predicate and then check if the success set of the predicate falls within the declaration [10, 15, 24].

Mishra and Zobel argue that if a clause never succeeds, so it does not contribute to the success set of the predicate, then the clause has a type error [6, 20]. They also argue that if the empty type is inferred for a predicate argument, then the predicate is ill-typed [6, 20]. Zobel [6] also argues that the notion of well-typedness applies only to programs and queries together and says that if $P \cup \{Q\}$ is well-typed, then at every step of the derivation the atoms are well-typed. This is undecidable in general [6].

Schrijvers, Bruynooghe, and Gallagher define well-typings for programs [26, 27, 51]. A well-typing for a predicate is defined given types for variables and can be smaller, or larger, than the success set of the predicate. By definition a well-typing of a program gives some guarantees about the program, but very little information about the true intention of the programmers or the semantics of the program.

One other alternative is the one given by Naish [24], where the types for a predicate come from the theoretical and intended specification thought by the programmer. Naish argues that logic programs are not consequences of the specifications intended by the programmer. As an example, the append

predicate has a specification "append(A,B,C) is true if and only if the list C is the concatenation of list B onto list A". But the usual definition of append accepts the atom $append([\,], 2, 2)$.

As a solution, Naish proposes the addition of type declarations to the program. Through program transformation, we would get a program that includes type information and programs would only accept atoms that are correct and type correct.

Example 9: Given a program defining the predicate `append` and the following type declaration and type signature:

```
append_type(A, B, C) :- list(A), list(B), list(C).
list([]).
list([A|B]) :- list(B).
```

We get the following transformed program for `append`:

```
append([], A, A) :- append_type([], A, A).
append([A|B], C, [A|D]) :- append(B, C, D),
                           append_type([A|B], C, [A|D]).
```

Note that the predicate definition in the transformed program includes the type verification for each argument. Naish discussed the inefficiency of this process and points to possible improvements [24].

Naish also argues that this model does not exclude most of the well-defined and bug-free programs, but does not prove so. An algorithm that outputs a list of the clauses from the transformed program that always fail is described. It is recognized that "a compromise between the time spent and the number of possible errors detected" [24] must be made.

In a later paper, Naish [52] describes a three-valued semantics for logic programming, where the third value represents "inadmissible" or "ill-typed" atoms. A predicate is well-typed when it accepts terms from the *intended types*, while the behavior for *inadmissible* terms, meaning terms that are not of the intended types for the predicate, is uncertain [25, 52, 53].

2.2.3 Type Inference

In order to avoid type declarations, there has been a lot of research into type inference [6, 9, 17, 18, 20, 22, 27, 54]. Several of these works infer types [6, 55], or well-typings [27], from programs using constraint generation and solving. Several others use abstract interpretation [17–19, 23, 54]. Other lines of research use program transformation [9, 22].

In particular, Zobel's approach [6] was a very influential one. There, Zobel builds an algorithm that can infer regular types for predicate arguments. Zobel argues that programs have implicit types that can be defined in terms of the program's success set. The goal of his type inference algorithm is to compute approximations to those implicit types, which may have no relation to the programmers intended types. He also argues that the inferred types are often crude approximations and, therefore, type inference is not a very useful tool [6]. Type inference is performed through an iterative function application, that starts with the universal type for each predicate argument and refines the definitions of those types at each iteration, until a fix-point is reached. At each iteration, constraints are generated and solved from the predicate definition.

In [26, 27], the authors infer well-typings for the program, through constraint generation and solving. A normal form and a solved form are defined and constraints are solved until those forms are reached. This process is decidable, so it always finishes. This approach results in very different types, as can be seen in the example below, since they do not try to approximate the success set of the predicates.

Example 10: Zobel describes type rules as: $\alpha \rightarrow \{\tau_1, \ldots, \tau_n\}$, where $\tau_1, \ldots, \tau_n$ are type terms, constructed from type constants, base types, type variables, type symbols, and type function symbols applied to type terms. The resulting types from type inference as presented in [6] for the predicate `append` are:

$T = \{\alpha_1^{append} \rightarrow \{[\,], [\mu|\alpha_1^{append}]\},$
$\alpha_2^{append} \rightarrow \{\mu\},$
$\alpha_3^{append} \rightarrow \{\mu, [\mu|\alpha_3^{append}]\}\}$

where μ is the universal type, and α_i^{append} is the type for the i-th argument of the predicate.

In [27], the inference of well-typings for the `append` predicate results in the following type signature for `append(`a_1`(T),`a_2`(T),`a_2`(T))`, where the types are defined as:

a_1`(T)` $\rightarrow$ `[ ]; [T|`a_1`(T)]`
a_2`(T)` $\rightarrow$ `[T|`a_2`(T)]`

We can clearly see that the types obtained from the inference of well-typings are much more informative and closer to the programmer's intention according to Naish [25], however, the well-typings are not a conservative

approximation of the success set. In particular, they do not even include the atom `append([],[],[])`.

In any approach, there are some restrictions to the results of type inference we can possibly get. For instance, when inferring polymorphic types for recursive predicates, we know that, in general, type inference with polymorphic recursion reduces to semi-unification, which is known to be undecidable [56]. There are however some works on types that argue that even thought that is the case, the results in practice are decidable for every tested program [56,57]. Another restriction is the language of types. Checking if a type term belongs to a type, when types are not tuple distributive is not decidable in general [8, 20].

2.2.4 Type Checking

After we infer types from a program, or we have declared types for the predicates, so we have a type signature for the predicates, the approaches for how to use this information vary.

On the one hand, one can use this information statically to help in the more efficient compilation of code, or to analyse characteristics of the program, such as termination [18]. Also, during type inference, at compile-time, type errors can be detected, which leads to early bug-detection and more efficient programming.

On the other hand, one can also use this information to check that given a query, the query is well-typed, meaning that it will call a predicate with arguments that are of its type, with respect to the program [6, 45]. One typical problem, in the approaches where types are approximations to the success set of a predicate, is to be able to distinguish between (intentional) failure due to the lack of solutions and (unintentional) failure due to type errors [6].

We can use the type information at run-time to check calls from untyped code to typed code, or vice versa [15].

In fact, the definition of type error has not been consistent throughout the literature on logic programming. As said before, some can argue that a type error exists if a predicate has no success set, but in practice some programmers define such predicates and do not want to change neither their old programs nor their programming style. Because of this, some authors proposed optional type checking [15], or gradual type systems [55].

Some other authors believe that the semantics of logic programming has been assumed as being a two-valued logic, when, in fact, it is a three-valued one [2, 53], where the third-value corresponds to nonsensical or inadmissible uses of predicates.

2.3 Applications in Prolog Compilers

Ciao Prolog is a Prolog dialect that uses type information [5]. Using its assertion language that extends the language semantics with pre-conditions, post-conditions, and several other properties of programs, Ciao does a form of type verification. If the programmer declares some assertions about types for some predicate, they are then checked at compile- and run-time. The type system can be based on Hindley-Milner types (in package `hmtypes`), on regular types (in package `regtypes`), or on types defined by the programmer as a `prop`. An example of a programmer defined type is the type for sorted lists given in Figure 6 in [58], that we show below.

Example 11: A programmer can define a type for a sorted list using the following piece of code:

```
:- prop sorted/1. sorted := [] | [_].
                  sorted([X,Y|Z]) :- X @< Y, sorted([Y|Z]).
```

Vaucheret and Bueno present a type inference strategy based on abstract interpretation [59]. They present a new widening operator, implement it in Ciao, and show some results in terms of efficiency. However, termination is not guaranteed so the implementation needs to enforce a limit which can be paid in loss of precision.

Several authors also present a combination of techniques to improve type analysis in [60]. The authors incentivize type declarations provided by the programmer. The experimental results suggest that this strategy improves on the efficiency of type analysis. Some of the advantages of declared types are pointed out, such as readability (since inferred types often have automatically generated names) of types and type signatures, and better precision in terms what the programmer intended for the program.

Schrijvers, Santos Costa, Wielemaker, and Demoen also implemented the type system defined by Mycroft and O'Keefe [61] as an add-on library for SWI-Prolog and YAP [15]. The type system is optional for each predicate, which allows for a gradual migration from untyped to typed code. Special care was put into the interface from typed code to untyped code and vice-versa. The types in the system are tuple distributive regular types and the programmer needs to provide both type declarations and type signatures for the predicates.

2.4 Comparison with our Approach

Our work is the first one to fully encompass types in logic programming from a semantic basis for types, to a type system, and to a type inference

algorithm with practical applications. Our semantics differs from previous ones because it is a three-valued semantics. Previous work on declarative semantics for logic programming is based on model theory, where some interpretation function either makes clauses true or false in a certain domain [62]. In our work, we extend this model theory approach by using a three-valued logic, using a specific semantic value for denoting erroneous programs. This approach goes back to early work on declarative debugging in logic programming [63]. In these early works whether inadmissible atoms succeed or fail was not important.

This idea was further formalized in logic programming through making explicit use of the third semantic value, in a previously defined three-valued semantics [53]. Naish's third value [52, 53] represents inadmissible calls to predicates, which Naish argued had unimportant results, meaning they could be either true or false. The semantics was based on a generalization of the T_P operator for the strong Kleene logic, which captured inadmissibility with respect to a specification containing mode and type information. The main difference to our work is that we use the weak Kleene logic [64, 65] to denote the propagating effect of type errors, which enables us to use our semantics to establish the semantic soundness of a type system for logic programming, using the third semantic value as the interpretation of ill-typed atoms. For us, the third-value corresponds to a wrongful use of a predicate in the program and therefore it is *wrong*. This wrong usage of a predicate is not permitted, so this information needs to propagate and not disappear.

Previous semantics for typed logic programming, based on different domains of interpretation, were defined before using many-sorted logics [12,48]. These semantics were defined for languages where type declarations formed an integral part of program syntax and were also used to determine their semantics. Our work differs from these approaches by defining separate semantics for untyped programs and types. Three-valued domains of interpretation revealed to be crucial in this separation of both semantics.

While other authors defined a Typed SLD-resolution [25], using a traditional unification algorithm, we used a typed unification algorithm, which means the procedure itself returns three possible values. In [25], Typed SLD-resolution just corresponds to normal SLD-resolution of transformed programs that contain type information.

In our work, dynamic type checking is possible due to the new typed unification algorithm, that can be used during resolution, resulting in three possible answers – true, false, and *wrong*. In [15], type checking is performed only on calls from untyped to typed code using program transformation, and on calls to untyped code from typed code to check whether the untyped code

satisfies previously made type annotations. In this previous work, type annotations for predicates were necessary in both scenarios and the semantic soundness of these run-time checks was not studied. Here we do a semantic study of dynamic typing and use it to show that a new operational mechanism detecting run-time errors is sound. For this we use predefined types only for constants and function symbols.

In several previous works, types approximated the success set of a predicate [6, 7, 9, 17]. This approach often leads to overly broad and even useless types, because the way logic programs are written can be very general and accept more than what was initially intended. Due to this, several of the authors recommend the declaration of type signature for predicates as an alternative [6, 45]. These approaches differ from ours in the sense that in our work types can filter the set of terms accepted by a predicate. Furthermore, we do not incentivize declarations of type signatures. Instead, programmers are incentivized to declare either data type definitions, or to declare nothing and rely on heuristics to obtain more informative types.

Our type system is also new and different from the previous ones in several ways. A rather influential type system was Mycroft and O'Keefe type system [61], which was later reconstructed as Typed Prolog by Lakshman and Reddy [12]. This system has types declared for the constants, function symbols and predicate symbols used in a program. Besides the difference in the language of types, one major difference is that in the Mycroft and O'Keefe type system, each clause of a predicate must have the same type. We lift this limitation extending the type language with sums of types (union types), such as in regular type languages, where the type of a predicate is the sum of the types of its clauses. The semantics of the Typed Prolog system and our semantics are quite different. The semantics of Typed Prolog was itself typed, while our semantics uses an independent semantics for programs and types.

Although we also perform type inference through constraint generation and solving, the algorithms for these procedures are new and different from previous ones. In [27], the constraints are generated from the entire program, meaning that calls to a predicate could affect the type inferred for that predicate, which is not true in our algorithm. In [26], the constraints are generated and solved from stratified strongly connected components, but the algorithm is different and results in different types for the same predicates. Besides, in these previous works, the inference of well-typings never fails, while that is not the case in our algorithm, where for some programs, inference will detect ill-typed programs and fail.

Other relevant works on type systems and type inference in logic programming include types used in the logic programming systems Ciao Prolog [58, 59, 66], SWI-Prolog and YAP [15]. These systems were dedicated to the

type inference problem and are not based on a declarative semantics with an explicit notion of type error. We plan to integrate our system into YAP as an optional feature.

Furthermore, our definition of *closed types* is new and, although it follows from a similar intuition to the ones in [25, 45, 57], it is based on different principles and results in different types.

Chapter 3

Dynamic Typing

Logic programming uses logic formulas, in particular Horn clauses, to represent both programs and calls to programs, *i.e.*, queries. This enables us to describe the program's semantics in terms of models for its corresponding logic formulas. The logic underlying all these concepts is first-order logic (FOL) and it has been shown that we can describe the semantics of logic programs using concepts of FOL. This is not without limitations, for instance, non-termination is not a part of the semantics for first-order logic, and the choice for the order of formulas, or atoms in a query, which is irrelevant in logic, can affect the result of computation. Besides that, as soon as we try to make the logic programming language useful in terms of expressiveness and efficiency, we have to resort to features whose semantics is often not related to FOL.

One other problem that immediately occurs when going from the concept of logic programming to a logic programming language is the idea of an error. The most widespread logic programming language, Prolog, defines a number of errors, such as *instantiation errors*, *type errors*, *evaluation errors*, among others. It is not clear what is the correspondence of errors with logic, since formulas can either evaluate to true or false, while errors stop execution of programs without returning any solution, which ends up being a different behavior from both true and false.

We argue that the logic that best describes this behavior is a three-valued logic. We will be focusing on *type errors*, and for us the third value will represent a type error. We then build a three-valued semantics for logic programming using the three-valued logic called weak Kleene logic [64]. The first change to the untyped semantics is in unification. We describe a new typed unification algorithm and build the operational semantics from there. We prove that the operational and the declarative semantics are equivalent, and clearly define what we mean by type errors, both in the program and in the query. The results presented in this chapter were partially presented in our previous paper [4].

3.1 Three-Valued Logic

The three-valued logic we chose is the weak Kleene Logic [64], where the third value, *wrong* propagates through connectives. This logic, initially introduced by Kleene, was later reinterpreted by Bochvar [67] and Beall [65], so that the third value represents *nonsensical* information. This is the reason why whenever a connective joins nonsense with any formula, the result is always nonsense. This is akin to the behavior and the propagation of run-time errors in a programming language. Particularly in logic programming, the use of this third semantic value for run-time type errors allows one to distinguish a program that simply fails from a program that erroneously uses its function and predicate arguments. This change in the semantics of logic programming from a two-valued semantics to a three-valued semantics captures the notion of type error and well-typedness and thus it will be the key in establishing the precise meaning of what is a *semantically sound type system* for logic programming. We will consider that the semantics of our programs follows a three-valued logic, where the values are $true$, $false$ and $wrong$.

In Figure 3.1 we can see the behavior of the *wrong* value in the $\wedge$ and $\vee$ connectives. The negation of logic values is defined as: $\neg true = false$, $\neg false = true$ and $\neg wrong = wrong$. The connective for implication is such that $a \rightarrow b \equiv \neg a \vee b$.

$\wedge$	*true*	*false*	*wrong*
true	*true*	*false*	*wrong*
false	*false*	*false*	*wrong*
wrong	*wrong*	*wrong*	*wrong*

$\vee$	*true*	*false*	*wrong*
true	*true*	*true*	*wrong*
false	*true*	*false*	*wrong*
wrong	*wrong*	*wrong*	*wrong*

FIGURE 3.1: Connectives of the three-valued logic - conjunction and disjunction

3.2 Typed Unification

In order to describe a typed unification algorithm, we will first introduce our language of types.

3.2.1 Types

We fix the set of base types *int, float, atom* and *string*, an enumerable set of compound types $f(\sigma_1, \cdots, \sigma_n)$, where f is a function symbol and σ_i are types, and an enumerable set of functional types of the form $\sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma$, where σ_i and σ are types.

We use this specific choice of base types because they correspond to types already present, to some extent, in Prolog. Some built-in predicates already expect integers, floating point numbers, or atoms.

3.2.2 From Traditional Unification to Typed Unification

Solving equality constraints using a unification algorithm [35, 43] is the main computational mechanism in logic programming. Logic programming usually uses an untyped term language and assumes a semantic universe composed of all semantic values: the Herbrand universe [40, 68].

However, in our work, we assume that the semantic values are split among several disjoint semantic domains and thus equality only makes sense inside each domain. Moreover each type will be mapped to a non-empty semantic domain. To reflect this, unification may now return three different outputs. Besides being successful or failing, unification can now return the *wrong* value. This is the logical value of nonsense and reflects the fact that we are trying to perform unification between terms with different types corresponding to a type error during program evaluation.

Since the goal of unification is to find a substitution, we will start by describing substitution and some of their properties.

3.2.3 Substitutions

A substitution is a mapping from variables to terms, which binds each variable X in its domain to a term t. We will represent bindings by $X \mapsto t$, substitutions by symbols such as $\theta, \eta, \delta \ldots$, and applying a substitution θ to a term t will be represented by $\theta(t)$. We say $\theta(t)$ is an instance of t.

Substitution composition is represented by $\circ$, *i.e.*, the composition of the substitutions θ and η is denoted $\theta \circ \eta$ and applying $(\theta \circ \eta)(t)$ corresponds to $\theta(\eta(t))$. We can also calculate substitution composition, *i.e.*, $\delta = \theta \circ \eta$ as defined below [40].

Definition 3 - Substitution Composition: Suppose θ and η are substitutions, such that $\theta = [X_1 \mapsto t_1, \ldots, X_n \mapsto t_n]$ and $\eta = [Y_1 \mapsto t_1\prime, \ldots, Y_m \mapsto t_m\prime]$. Then, the composition $\eta \circ \theta$ is calculated by following these steps:

- remove from the sequence $X_1 \mapsto \eta(t_1), \ldots, X_n \mapsto \eta(t_n), Y_1 \mapsto t_1\prime, \ldots, Y_m \mapsto t_m\prime$ the bindings $X_i \mapsto \eta(t_i)$ such that $X_i = \eta(t_i)$ and the elements $Y_i \mapsto t_i\prime$ for which $\exists X_j . Y_i = X_j$
- form a substitution from the resulting sequence.

A substitution θ is called a unifier of two terms t_1 and t_2 if and only if $\theta(t_1) = \theta(t_2)$. If such a substitution exists, we say that the two terms are unifiable. In particular, a unifier θ is called a most general unifier (MGU) of two terms t_1 and t_2 if for every other unifier η of t_1 and t_2, $\eta = \delta \circ \theta$, for some substitution δ.

3.2.4 Typed Unification Algorithm

First order unification [43] assumes an untyped universe, so unification between any two terms always makes sense. Therefore, it either returns an MGU of the terms, if it exists, or halts with failure.

We argue that typed unification only makes sense between terms of the same type. Here we will extend a previous unification algorithm by Martelli and Montanari [35] to define a *typed unification algorithm*, where failure will be separated into $false$, where two terms are not unifiable but may have the same type, and *wrong*, where the terms cannot have the same type.

Definition 4 - Typed Unification Algorithm: Let t_1 and t_2 be two terms, and F be a flag that starts *true*. We create the starting set of equalities as $S = \{t_1 = t_2\}$, and we will rewrite the pair (S, F) by applying the following rules until it is no longer possible to apply any of them, or until the algorithm halts with *wrong*. If no rules are applicable, then we output $false$ if the flag is $false$, or output the solved set S, which can be seen as a substitution.

1. $(\{f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)\} \cup Rest, F) \rightarrow (\{t_1 = s_1, \ldots, t_n = s_n\} \cup Rest, F)$
2. $(\{f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)\} \cup Rest, F) \rightarrow wrong$, if $f \neq g$ or $n \neq m$
3. $(\{c = c\} \cup Rest, F) \rightarrow (Rest, F)$
4. $(\{c = d\} \cup Rest, F) \rightarrow (Rest, false)$, if $c \neq d$, and c and d have the same type
5. $(\{c = d\} \cup Rest, F) \rightarrow wrong$, if $c \neq d$, and c and d have different types
6. $(\{c = f(t_1, \ldots, t_n)\} \cup Rest, F) \rightarrow wrong$
7. $(\{f(t_1, \ldots, t_n) = c\} \cup Rest, F) \rightarrow wrong$
8. $(\{X = X\} \cup Rest, F) \rightarrow (Rest, F)$
9. $(\{t = X\} \cup Rest, F) \rightarrow (\{X = t\} \cup Rest, F)$, where t is not a variable and X is a variable
10. $(\{X = t\} \cup Rest, F) \rightarrow (\{X = t\} \cup [X \mapsto t](Rest), F)$, where X does not occur in t and X occurs in $Rest$
11. $(\{X = t\} \cup Rest, F) \rightarrow (Rest, false)$, where X occurs in t and $X \neq t$

In order to clarify some of these steps we will present three examples for the three possible results of the algorithm.

Example 12: Let t_1 be $f(X, a)$ and t_2 be $f(g(a), Y)$. We generate the pair $(\{f(X, a) = f(g(a), Y)\}, true)$ and proceed to apply the rewriting rules.
$(\{f(X, a) = f(g(a), Y)\}, true) \rightarrow_1 (\{X = g(a), a = Y\}, true) \rightarrow_9$
$(\{X = g(a), Y = a\}, true) \rightarrow \{X = g(a), Y = a\}$
Now we can see $\{X = g(a), Y = a)\}$ as the substitution $[X \mapsto g(a), Y \mapsto a]$.

This example illustrates that for terms that are unifiable the algorithm behaves like the untyped version of Martelli and Montanari [35], as the rules that can be applied to reach an MGU are equal to the ones in the untyped algorithm.

Example 13: Let t_1 be $g(X, a, f(1))$ and t_2 be $g(b, Y, f(2))$. We generate the pair $(\{g(X, a, f(1)) = g(b, Y, f(2))\}, true)$, and proceed to apply the rewriting rules.
$(\{g(X, a, f(1)) = g(b, Y, f(2))\}, true) \rightarrow_1$
$(\{X = b, a = Y, f(1) = f(2)\}, true) \rightarrow_{10}$
$(\{X = b, Y = a, f(1) = f(2)\}, true) \rightarrow_1$
$(\{X = b, Y = a, 1 = 2\}, true) \rightarrow_5$
$(\{X = b, Y = a\}, false) \rightarrow false.$

In the previous example, the terms are not unifiable. In fact, the untyped algorithm would simply fail. However, if we take the substitution $\theta = [X \mapsto b, Y \mapsto a]$, the terms $\theta(t_1)$ and $\theta(t_2)$ have the same type.

Example 14: Let t_1 be $f(g(X, 1, a), h(1))$ and t_2 be $f(h(2), g(4, b, Y))$. We generate the pair $(\{f(g(X, 1, a), h(1)) = f(h(2), g(4, b, Y))\}, true)$ and proceed to apply the rewriting rules.
$(\{f(g(X, 1, a), h(1)) = f(h(2), g(4, b, Y))\}, true) \rightarrow_1$
$(\{g(X, 1, a) = h(2), h(1) = g(4, b, Y)\}, true) \rightarrow_2 wrong$

The previous example illustrates a case where a *wrong* is reached and it also would result in failure in the Martelli and Montanari algorithm [35].

The previously mentioned fact that successful cases of this algorithm are the same as for untyped first order unification [35] is proved by the following theorem.

Theorem 3 - Conservative with respect to term unification: Let t_1 and t_2 be two terms. If we apply the Martelli-Montanari algorithm (MM algorithm) to t_1 and t_2 and it returns a set of solved equalities S, then the typed unification algorithm applied to the same two terms is also successful and returns the same set of equalities.

Proof: The proof follows from induction on S.
Base cases:

- Suppose we have an equality of the form $f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$ in S. Then the MM algorithm halts with failure, and our algorithm halts with $wrong$.
- Suppose we have an equality in S of the form $c = d$. Then the MM algorithm halts with failure, and our algorithm either halts with $wrong$ or changes F to $false$, depending on the types of c and d. In either case, it will not be successful.
- Suppose we have an equality in S of the form $c = f(t_1, \ldots, t_n)$ or, reversely, $f(t_1, \ldots, t_n) = c$. Then the MM algorithm halts with failure, and our algorithm halts with $wrong$.
- Suppose we have an equality in S of the form $X = t$, where X occurs in t. Then, the MM algorithm halts with failure and our algorithm changes F to $false$, so it is never successful.

Inductive cases:

- Suppose we have an equality of the form $f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$ in S. Both algorithms generate the same new equalities and replace the selected one with those in S. Then, by the induction hypothesis, if the MM algorithm succeeds and outputs a set of solved equalities $S\prime$, so does our algorithm.
- Suppose we have an equality in S of the form $c = c$. Both algorithms delete this equality from S. Then, by the induction hypothesis, if the MM algorithm succeeds and outputs a set of solved equalities $S\prime$, so does our algorithm.
- Suppose we have an equality in S of the form $X = X$. Both algorithms delete this equality from S. Then, by the induction hypothesis, if the MM algorithm succeeds and outputs a set of solved equalities $S\prime$, so does our algorithm.
- Suppose we have an equality in S of the form $t = X$, where t is not a variable and X is a variable. Both algorithms replace this equality from S with the same new one. Then, by the induction hypothesis, if the MM algorithm succeeds and outputs a set of solved equalities $S\prime$, so does our algorithm.
- Suppose we have an equality in S of the form $X = t$, where X does not occur in t and X occurs somewhere else in S. Both algorithms apply the same substitution to $S \setminus \{X = t\}$, therefore resulting in the

same set of equalities. Then, by the induction hypothesis, if the MM algorithm succeeds and outputs a set of solved equalities $S\prime$, so does our algorithm.

In any other case, none of the algorithms apply. □

The new typed unification algorithm always terminates. We prove this in the following theorem and use it to prove a later property.

Theorem 4 - Termination of the Typed Unification Algorithm: For any two terms t_1 and t_2, the typed unification algorithm terminates.

Proof: We define the following metric for the algorithm:

- NV: number of variables that occur more than once on the set of equalities and that occurrence is as the left-hand side of some equality
- NS: number of occurrences of non-variable symbols
- NX: number of equalities of the form $X = X$ or $t = X$, where X is a variable and t is not a variable.

We prove termination by showing that NX reduces to zero. Termination of the algorithm is proven by a measure function that maps the set to a tuple (NV, NS, NX). The following table shows that each step decreases the tuple w.r.t. the lexicographical order of the tuple.

	NV	NS	NX
1.	$\leq$	$<$	
2.	0	0	0
3.	$=$	$<$	
4.	$=$	$<$	
5.	0	0	0
6.	0	0	0
7.	0	0	0
8.	$\leq$	$=$	$<$
9.	$\leq$	$=$	$<$
10.	$<$		
11.	$<$		

□

The following theorem proves typed unification detects run-time type errors, in the sense that if we try to perform typed unification on terms with no possible type in common, then the algorithm returns *wrong*.

Theorem 5 - Ill-typed unification: If the output of the typed unification algorithm is *wrong*, then there is no substitution θ such that $\theta(t_1)$ and $\theta(t_2)$ have the same type.

Proof: The proof follows induction on the number of steps of the algorithm from the starting t_1 and t_2, that are represented in each case by $t_1 = t_2$. The number of steps is finite from Theorem 4.
Base cases:

- Suppose we have $c = d$ and c has the same type as d. Then the algorithm outputs $false$, not $wrong$.
- Suppose we have $c = d$ and c has a different type from d. Then the algorithm outputs $wrong$, but there is no θ such that $\theta(c)$ has the same type as $\theta(d)$, since $\theta(c) = c$ and $\theta(d) = d$, and c has a different type from d.
- Suppose we have $X = t$, where X occurs in t. Then the algorithm outputs $false$, not $wrong$.
- Suppose we have $f(t_1, \ldots, t_n) = g(s_1, \ldots, s_m)$. Then the algorithm outputs $wrong$, but we know there is no θ such that $\theta(f(t_1, \ldots, t_n))$ and $\theta(g(s_1, \ldots, s_m))$ have the same type, since terms starting with different function symbols always have a different type.
- Suppose we have $c = c$, then the algorithm outputs the empty substitution, not $wrong$.
- Suppose we have $X = t$, where X does not occur in t, or $X = X$. Then the algorithm ends with success, and does not output $wrong$.
- Suppose we have $t = X$, where t is not a variable. Then, in one step the constraint set becomes $\{X = t\}$ and we know, from above, that in this case our algorithm never outputs the value $wrong$.

Inductive step:

- Suppose we have $f(t_1, \ldots, t_n) = f(s_1, \ldots, s_n)$. Then, by the induction hypothesis, if the algorithm outputs $wrong$ for the input $(\{t_1 = s_1, \ldots, t_n = s_n\}, true)$, we know that there is no θ such that $\forall i.\theta(t_i)$ and $\theta(s_i)$ have the same type. Therefore, no θ such that $\theta(f(t_1, \ldots, t_n))$ and $\theta(f(s_1, \ldots, s_n))$ have the same type exists either, by the properties of substitution.

So, we prove that whenever the typed unification algorithm halts with $wrong$ for some pair of terms t_1 and t_2, then there is no substitution θ such that $\theta(t_1)$ and $\theta(t_2)$ have the same type. □

Example 15: Let $t_1 = f(1, g(h(X, 2)), Y)$ and $t_2 = f(Z, g(h(W, a)), 1)$. The typed unification algorithm outputs *wrong*. We can see in Figure 3.2 that there is no substitution θ such that $\theta(t_1) = \theta(t_2)$, nor any substitution θ such that $\theta(t_1)$ has the same type as $\theta(t_2)$, since the highlighted terms cannot have the same type for any substitution.

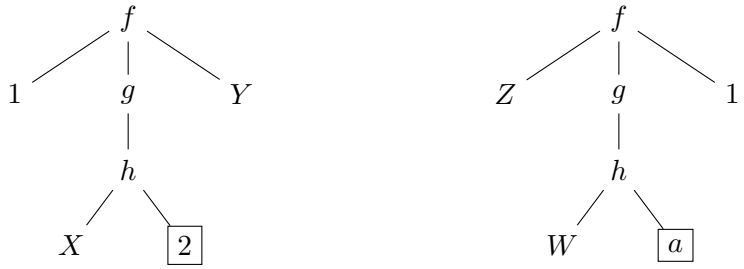


FIGURE 3.2: Tree representation of terms t_1 and t_2

3.3 Typed SLD-resolution

The operational semantics of logic programming describes how answers are computed. Here we define Typed SDL(TSLD)-resolution which returns the third value *wrong* whenever it finds a type error. We start by defining a TSLD-derivation step, which is a variation on the basic mechanism for computing answers to queries in the untyped semantics for logic programming, the SLD-derivation step. The major difference is the use of the typed unification algorithm. Then we create TSLD-derivations by iteratively applying these singular steps. After this, we introduce the concept of TSLD-trees and use it to represent the search space for answers in logic programming. Finally, we interpret the contents of the TSLD-tree.

3.3.1 TSLD-derivation

To compute in logic programming, we need a program P and a query Q. We can interpret P as being a set of statements, or rules, and Q as being a question that will be answered by finding an instance $\theta(Q)$ such that $\theta(Q)$ follows from P. The essence of computation in logic programming is then to find such θ, CAS [40].

In our setting the basic step for computation is the TSLD-derivation step. It corresponds to having a non-empty query Q and selecting from Q an atom A. If A unifies with H, where $H \leftarrow \bar{B}$ is an input clause, we replace A in Q by $\bar{B}$ and apply an MGU of A and H to the query.

Definition 5 - TSLD-derivation step: Consider a non-empty query $Q = \bar{A}_1, A, \bar{A}_2$ and a clause c of the form $H \leftarrow \bar{B}$. Suppose that A unifies (using

typed unification) with H and let θ be an MGU of A and H. A is called the *selected atom* of Q. Then we write

$$\bar{A}_1, A, \bar{A}_2 \underset{c}{\Longrightarrow} \theta(\bar{A}_1, \bar{B}, \bar{A}_2)$$

and call it a *TSLD-derivation step*. $H \leftarrow \bar{B}$ is called its *input clause*. If typed unification between the selected atom A and the input clause c outputs $wrong$ (or $false$) we write the TSLD-derivation step as $Q \Longrightarrow wrong$ (or $Q \Longrightarrow false, \bar{A}_1, \bar{A}_2$).

In this definition we assume that A is variable disjoint with H. It is always possible to rename the variables in $H \leftarrow \bar{B}$ in order to achieve this, without loss of generality.

Definition 6 - TSLD-derivation: Given a program P and a query Q a sequence of TSLD-derivation steps from Q with input clauses of P reaching the empty query, $false$, or $wrong$, is called a *TSLD-derivation* of Q in P.

If the program is clear from the context, we speak of a TSLD-derivation of the query Q and if the input clauses are irrelevant we drop the reference to them. Informally, a TSLD-derivation corresponds to iterating the process of the TSLD-derivation step. We say that a TSLD-derivation is *successful* if we reach the empty query, further denoted by $\square$. The composition of the MGUs $\theta_1, \ldots, \theta_n$ used in each TSLD-derivation step is the CAS of the query. A TSLD-derivation that reaches $false$ is called a *failed derivation* and a TSLD-derivation that reaches $wrong$ is called an *erroneous derivation*.

In a TSLD-derivation, at each TSLD-derivation step we have several choices. We choose an atom from the query, a clause from the program, and an MGU. It is proven in [40] that the choice of MGU does not affect the success or failure of an SLD-derivation, as long as the resulting MGU is idempotent. Since for TSLD-derivations the success set is the same as the ones in a corresponding SLD-derivation, then the result still holds for TSLD.

The *selection rule*, i.e, how we choose the selected atom in the considered query, does not influence the success of a TSLD-derivation either [40], however if you stop as soon as unification returns false, it could prevent us from detecting a type error, in a later step. Let us show this in the following example.

Example 16: Consider the logic program P consisting of only one fact, $p(X, X)$, and the selection rule that chooses the leftmost atom at each step. Then, if we stopped when reaching $false$, the query $Q = p(1, 2), p(1, a)$ would have the TSLD-derivation $Q \Longrightarrow false$, since typed unification

between $p(X, X)$ and $p(1, 2)$ outputs $false$. However, the query $Q\prime = p(1, a), p(1, 2)$ has the TSLD-derivation $Q \implies wrong$, since typed unification between $p(X, X)$ and $p(1, a)$ outputs $wrong$.

In fact, as the comma stands for conjunction, and since $wrong \wedge false = false \wedge wrong = wrong$, we have to continue even if typed unification outputs $false$ in a step, and check if we ever reach the value $wrong$. In general, for any selection rule S we can construct a query Q such that it is necessary to continue when typed unification outputs $false$ for some atom in Q. Therefore, when we reach the value $false$ in a TSLD-derivation step, we continue applying steps until either we obtain a value $wrong$ from typed unification or we have no more atoms to select. In this last case, we can safely say that we reached $false$. This guarantees independence of the selection rule. For the following example we use the selection rule that always chooses the leftmost atom in a query, which is the selection rule of Prolog.

Example 17: Let us continue Example 16. The TSLD-derivation for Q is $Q \implies false, p(1, a) \implies wrong$. Let $Q\prime\prime = p(1, 2), p(1, 1)$. Then the TSLD-derivation is $Q\prime\prime \implies false, p(1, 1) \implies false$.

Note that when we get to $false$ for a typed unification in a TSLD-derivation, we can only output $false$ or $wrong$, so either way it is not a successful derivation.

The selected clause from the program is another choice point we have at each TSLD-derivation step. We will discuss the impact of this choice in the next section.

3.3.2 TSLD-tree

When we want to find a successful TSLD-derivation for a query, we need to consider the entire search space, which consists of all possible derivations, choosing all possible clauses for a selected atom. We are considering a fixed selection rule here, so the only thing that changes between derivations is the selected clause. We say that a clause $H \leftarrow \bar{B}$ is *applicable* to an atom A if H and A have the same predicate symbol with the same arity.

Definition 7 - TSLD-tree: Given a program P and a query Q, a *TSLD-tree* for $P \cup \{Q\}$ is a tree where the branches are TSLD-derivations of $P \cup \{Q\}$ and every node Q has a child for each clause from P applicable to the selected atom of Q.

We will present some terms to classify a TSLD-tree based on what occurs in its branches.

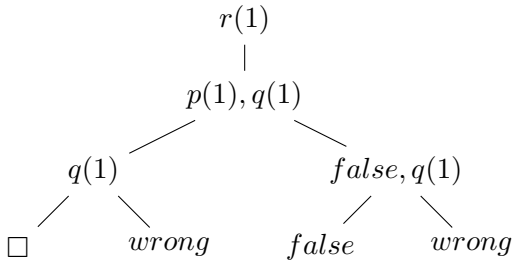
Definition 8 - TSLD-tree classification:

- If a TSLD-tree contains the empty query, we call it *successful*.
- If a TSLD-tree is finite and all its branches are erroneous TSLD-derivations, we call it *finitely erroneous*.
- If a TSLD-tree is finite and it is not successful nor finitely erroneous, we say it is *finitely failed*.

Example 18: Let program P be:

```
p(1).
p(2).
q(1).
q(a).
r(X) :- p(X),q(X).
```

and let query Q be $r(1)$. The TSLD-tree for Q and P is the following successful TSLD-tree:



We will now present some auxiliary definitions which are needed to clearly define the notion of a type error in a program.

Definition 9 - Generic Query: Let Q be a query and P a program. We say that Q is a *generic query* of P iff Q is composed of an atom of the form $p(X_1, \ldots, X_n)$ for some predicate symbol p that occurs in the head of at least one clause in P, where $X_1, \ldots, X_n$ are variables that occur only once in the query.

Example 19: Let P be the program defined as follows:

```
p(X,X).
q(X) :- p(1,a).
```

Then, given the generic query $Q_2 = q(X_1)$, we have the following TSLD-derivation: $q(X_1) \implies p(1, a) \implies wrong$.

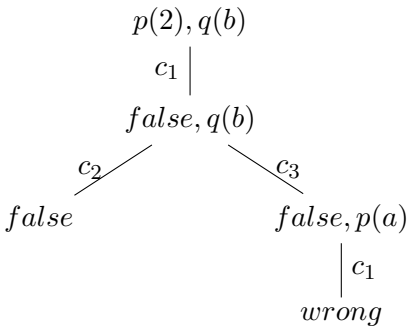
Definition 10 - Blamed Clause: Given a program P and a query Q, a clause c is a *blamed clause* of the TSLD-tree for $P \cup \{Q\}$ if all derivations where c is an input clause are erroneous.

The blamed clause is a clause in the program which causes a type error. A similar notion was first defined for functional programming languages with the blame calculus [69].

Example 20: Let P be the following program, with clauses c_1, c_2, and c_3, respectively:

```
p(1).
q(a).
q(X) :- p(a).
```

Then for the query $Q = p(2), q(b)$, we have the following TSLD-tree:



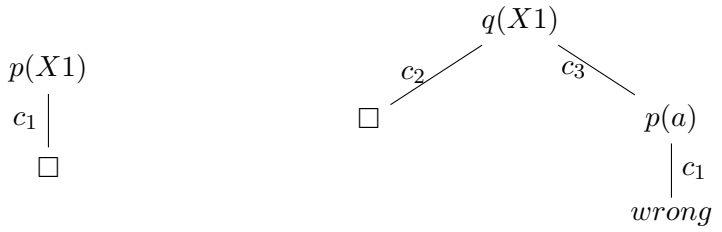
In this case, c_3 is a blamed clause, since every derivation that uses it eventually reaches *wrong*. Note that c_1 is not a blamed clause, because the leftmost branch of the TSLD-tree uses c_1 but is $false$.

Definition 11 - Blamed Set: Suppose we have a program P. We call the *blamed set* of P the set of blamed clauses of each generic query of P.

Definition 12 - Type Error in the Program: Suppose we have a program P. We say that P has a *type error* if at least one clause c in the *blamed set* of P is a blamed clause in every TSLD-tree where it occurs for $P \cup \{Q\}$, where Q is any generic query. We call c an *erroneous clause* of P.

Note that if a program does not have a type error, then there is no *blamed clause* in the blamed set of P that always leads to erroneous derivations.

Example 21: Assume the same program from Example 8. Let $Q_1 = p(X1)$ and $Q_2 = q(X1)$ be generic queries of P. The TSLD-trees for $P \cup \{Q_1\}$ and $P \cup \{Q_2\}$ are:



As we can see, the blamed set of P is $\{c_1, c_3\}$, and c_3 is an *erroneous clause* of P. Note that c_1 is not an erroneous clause of P since in the leftmost TSLD-tree, it is used but not a blamed clause.

Intuitively, having a type error in the program means that somewhere in the program we will perform typed unification between two terms that do not have the same type.

Consider a generic query $Q = p(X_1, \ldots, X_n)$. For some derivation, after one step, we will have $\theta(\bar{B})$, where $H \leftarrow \bar{B}$ is a clause in P and θ is a unifier of $p(X_1, \ldots, X_n)$ and H. Since θ, or any other idempotent MGU of $p(X_1, \ldots, X_n)$ and H, is a renaming of $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$, where $H = p(t_1, \ldots, t_n)$, and since the variables $X_1, \ldots, X_n$ do not occur in $\bar{B}$ because the clause is variable disjoint from the query by definition, then $\theta(\bar{B}) = \bar{B}$.

After selecting the *erroneous clause* c, every TSLD-derivation is such that $Q_0 \implies \cdots \implies Q_n \implies \mathit{wrong}$. Thus, at step Q_n, the selected atom comes from the program and every MGU applied up to this point is from substitutions arising from the program itself and not the query. Therefore, the type error was in the program.

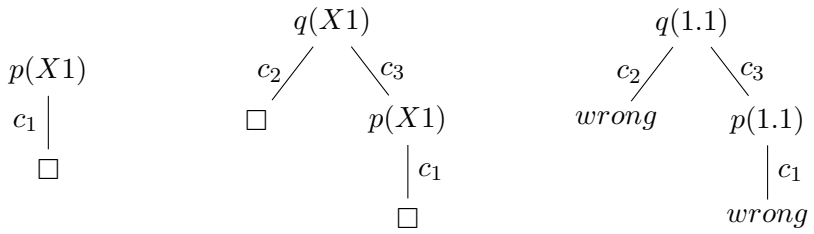
Definition 13 - Type Error in the Query: Let P be a program and Q be a query. If there is no type error in P and the TSLD-tree is finitely erroneous, then we say that there is a *type error in the query* Q with respect to P.

If there is no type error in the program P but the TSLD-tree is finitely erroneous, then that error must have occurred in a unification between terms from the query and the program. We then say that the type error is in the query.

Example 22: Now suppose we have the following program:

```
p(1).
q(a).
q(X) :- p(X).
```

Let us name the clauses c_1, c_2, and c_3, respectively. The following trees are the TSLD-tree for generic queries, and the TSLD-tree for the query $Q = q(1.1)$.



From the two leftmost TSLD-trees, we can conclude that the program has no *erroneous clause*, which means that there is no type error in the program. The rightmost tree is finitely erroneous, therefore there is a type error in query Q.

3.4 Typed Interpretations

The declarative semantics of logic programming is, in opposition to the operational one, a definition of what the programs compute. The fact that logic programming can be interpreted this way supports the fact that logic programming is declarative [40]. In this section, we will introduce the concept of interpretations, which takes us from the syntactic programs we saw and used so far into the semantic universe, giving them meaning. With this interpretation we will redefine a declarative semantics for logic programming first defined in [2] and prove a connection between both the operational and the declarative semantics.

3.4.1 Domains

Instead of interpreting the universe of semantics values in a single set containing every term, we will divide the universe into domains. Let U be a non-empty set of semantic values, which we will call the universe. We assume that the universe is divided into domains such that each ground term is mapped to a semantic value in a non-empty domain. Thus, U is divided into domains as follows: $U = Int + Float + Atom + String + A_1 + \ldots + A_n + F + Bool + W$, where Int is the domain of integer numbers, $Float$ is the domain of floating point numbers, $Atom$ is the domain of non-numeric constants, $String$

is the domain of strings, A_i are domains for trees, where each domain has trees whose root is the same functor symbol and its n-children belong to n domains and F is the domain of functions. Moreover, we define *Bool* as the domain containing *true* and $false$, and W as the domain with the single value *wrong*, corresponding to a run-time error. We will call Int, $Float$, *Atom*, and *String* the base domains, and $A_1, \ldots, A_n$ the tree domains.

In particular, we can see that constants are separated into several predefined base domains, one for each base type, while complex terms, *i.e.* trees, are separated into domains depending on the principal function symbol (root) and the n-tuple inside the parenthesis (n-children).

3.4.2 Interpretations

Every constant of some type T is associated with a semantic value in one of the base domains, Int, $Float$, *Atom*, or *String*, corresponding to T. Every function symbol f of arity n in our language is associated with a mapping f_U from any n-tuple of base or tree domains $\delta_1 \times \cdots \times \delta_n$ to the domain $F(\delta_1, \ldots, \delta_n)$, which is the domain of trees whose root is f and the n-children are in the domains δ_i.

To define the semantic value for terms, we will first have to define *states*. States, Σ, are bindings from variables into values of the universe. We also define a function *domain* that when applied to a semantic value returns the domain it belongs to. The semantic value of a term is defined as follows:

$$[\![X]\!]_\Sigma = \Sigma(X)$$
$$[\![c]\!]_\Sigma = c_U$$
$$[\![f(t_1, \ldots, t_n)]\!]_\Sigma = f_U([\![t_1]\!]_\Sigma, \ldots, [\![t_n]\!]_\Sigma)$$

An *interpretation* I associates every predicate symbol p with a single function $I(p)$ in F, such that the output of the function $I(p)$ is the domain *Bool* and the input is a union of tuples of domains. For each tuple that is in its domain, the function $I(p)$ either returns *true* or $false$. We will use $[\![\ ]\!]_{I,\Sigma}$ to denote the semantics of an *expression* E, which can be an atom, a query, or a clause, in an interpretation I, and define it as follows:

$$[\![p(t_1, \ldots, t_n)]\!]_{I,\Sigma} = \textbf{if } (domain([\![t_1]\!]_\Sigma), \ldots, domain([\![t_n]\!]_\Sigma)) \subseteq domain(I(p))$$
$$\textbf{then } I(p)([\![t_1]\!]_\Sigma, \ldots, [\![t_n]\!]_\Sigma)$$
$$\textbf{else } wrong$$
$$[\![A_1, \ldots, A_n]\!]_{I,\Sigma} = [\![A_1]\!]_{I,\Sigma} \wedge \ldots \wedge [\![A_n]\!]_{I,\Sigma}$$
$$[\![q(t_1, \ldots, t_n) : -\bar{B}]\!]_{I,\Sigma} = ([\![\bar{B}]\!]_{I,\Sigma} \longrightarrow ([\![q(t_1, \ldots, t_n)]\!]_{I,\Sigma}))$$

Note that if the clause is of the form $H \leftarrow$, then its semantics is equivalent to that of H. Also note that interpretations can differ both on the type

of $I(p_i)$ for some p_i or on the set of terms that is accepted by $I(p_i)$, even if the type of the function is the same.

3.4.3 Models

The term language and their semantic values are fixed, thus each interpretation I is determined by the interpretation of the predicate symbols. Interpretations differ from each other only in the functions $I(p)$ they associate to each predicate p defined in P.

We now define a *context* as a set Γ of pairs of the form $X : D$, where X is a variable that occurs only once in the set, and D is a domain. We say that Σ complies with Γ if every binding $X : v$ in Σ is such that $(X : D) \in \Delta$ and $v \in D$.

An interpretation I is a *model* of E in the context Γ iff for every state Σ that complies with Γ, $[\![E]\!]_{I,\Sigma} = true$. We will denote this as $\Gamma \models [\![E]\!]_I$. Given a program P, we say that an interpretation I is a model of P in context Γ if I is a model of every clause in P in context Γ. Here we assume, without loss of generality, that all clauses are variable disjoint with each other.

If two expressions E_1 and E_2 are such that every model of E_1 in a context Γ is also a model of E_2 in the context Γ, then we say that E_2 is a semantic consequence of E_1 and represent this by $E_1 \models E_2$.

Suppose two interpretations I_1 and I_2 are models of program P in some context Γ. Suppose, in particular, that for some predicate p of P the associated function is $I_1(p)$ for I_1 and $I_2(p)$ for I_2. Let us call T_i the set of tuples of terms for which $I_i(p)$ outputs $true$, and F_i to the set of tuples of terms for which $I_i(p)$ outputs $false$. We say that I_1 is smaller than I_2 if $T_1 \subseteq T_2$ and, if $T_1 = T_2$, then $F_1 \subseteq F_2$.

We say that a model I of P in context Γ is *minimal* if for every other model $I\prime$ of P in context Γ, I is smaller than $I\prime$.

Example 23: Consider the program P defined below:

```
father(john,mary).
father(phil,john).
grandfather(X,Y) :- father(X,Z), father(Z,Y).
```

Suppose that interpretation in I_1, $I_1(p)$ is associated with $grandfather$ and $I_1(p) :: Atom \times Atom \rightarrow Bool$. Also, suppose that $I_2(p)$ is associated with $grandfather$ in I_2, with the same domain. Suppose that $I_3(p)$, associated in I_3 with $grandfather$, is such that $I_3(p) :: Atom \times Atom \cup Int \times Int \rightarrow Bool$. Let the sets $T_1 = \{(phil, mary)\}$, $T_2 = \{(phil, mary), (john, caroline)\}$, and $T_3 = \{(phil, mary)\}$ be the sets of accepted tuples for $I_1(p)$, $I_2(p)$, and $I_3(p)$, respectively.

Thus, if these interpretations associate the same function $I(q) :: Atom \times Atom \rightarrow Bool$ to $father$, and $T = \{(john, mary), (phil, john)\}$ the set of accepted tuples for $I(q)$, then all I_i are models of P in context $\Gamma = \{X : Atom, Y : Atom, Z : Atom\}$. In fact, all states Σ that comply with Γ are such that $[\![grandfather(X, Y) : -father(X, Z), father(Z, Y)]\!]_{I_i,\Sigma}$ is true, for all $i = 1, 2, 3$.

But note that $T_1 \subseteq T_2$, and $T_1 = T_3$, but $F_1 \subseteq F_3$. In fact, any smaller domain or set T_k would not model P. Therefore I_1 is the minimal model of P.

3.4.4 Type Errors

We can calculate the atoms accepted by a program, using the immediate consequence operator, T_P. The T_P operator is traditionally used in logic programming to iteratively calculate the minimal model of a logic program as presented in [40, 41, 70]. The minimal model is defined as the least fixed point of this operator in the untyped semantics for logic programming.

Suppose we have a program P and we apply the T_P operator to the empty set of tuples once. Then $T_P(\emptyset) = S$, where S contains all atoms that are instances of heads of clauses that have no body. We can now apply the T_P operator again to this set, represented by $T_P(S)$. In order to present this iterative process, we will represent successive applications of the operator by $T_P \uparrow (n + 1) = T_P(T_P \uparrow n)$. Using this representation, $T_P(S) = T_P(T_P(\emptyset)) = T_P \uparrow 2(\emptyset)$.

If we keep applying this operator, eventually, the application $T_P \uparrow \omega(\emptyset)$ will be such that $T_P \uparrow (\omega + 1)(\emptyset) = T_P \uparrow \omega(\emptyset)$. So, we reached a fixed point of this operator, which corresponds to the minimal model of program P in the untyped semantics of logic programming.

Since for us interpretations for predicates are typed, $T_P \uparrow \omega(\emptyset)$ does not generate an interpretation. Instead it generates a set of atoms S. Then we say that any interpretation I derived from S is such that for all predicates p occurring in S, $I(p) :: (D_{(1,1)} \times \cdots \times D_{(1,n)}) \cup \cdots \cup (D_{(k,1)}, \ldots, D_{(k,n)}) \rightarrow Bool$, where for all $i = 1, \ldots, k$ there is at least one atom $p(v_1, \ldots, v_n) \in S$ such that $v_1 \in D_{(i,1)}, \ldots, v_n \in D_{(i,n)}$. Note that these interpretations may not be models of P using our new definition of a model. We are now able to define the notion of *ill-typed program*.

Definition 14 - Ill-typed Program: Let P be a program. If no interpretation derived from $T_P \uparrow \omega(\emptyset)$ is a model of P, we say that P is an ill-typed program.

Example 24: Let P be the program defined as:

```
p(1).
p(a).

q(X) :- p(1.1).
```

Then $S = T_P \uparrow \omega(\emptyset) = \{p(1), p(a)\}$. So any interpretation I derived from S is such that $I(p) :: Int \cup Atom \rightarrow Bool$. Therefore for any context Γ, for every Σ that complies with Γ, $[\![q(X) : -p(1.1)]\!]_{I,\Sigma} = wrong$. Therefore no such I is a model of P.

The reason why $T_P \uparrow \omega(\emptyset)$ is always a minimal model of P in the untyped semantics, comes from the fact that whenever a body of a clause is $false$ for all states, then the clause is trivially *true* for all states. However in our semantics, since we are separating these cases into $false$ and *wrong*, the *wrong* ones do not trivially make the formula *true*, making it *wrong* instead. These are the ill-typed cases.

Lemma 1 - Erroneous Clause Type Error: Suppose there is a type error in the program with erroneous clause $H \leftarrow A_1, \ldots, A_m$. Then, $\exists A_i = p(t_1, \ldots, t_n)$ such that $\forall p(s_1, \ldots, s_n) \in T_P \uparrow \omega(\emptyset).\forall\Sigma.\exists j.domain([\![t_j]\!]_\Sigma) \neq domain([\![s_j]\!]_\Sigma)$.

Proof: We will prove this by contradiction. Suppose that for all A_i there is some $p(s_1, \ldots, s_n) \in T_P \uparrow \omega(\emptyset)$ such that there is a Σ for which, $\forall i \in [1, \ldots, n].domain([\![t_i]\!]_\Sigma) = domain([\![s_i]\!]_\Sigma)$. Then, there would be a derivation of the form $A_1, \ldots, A_m, \bar{B} \implies \cdots \implies \bar{B}$ or $A_1, \ldots, A_m, \bar{B} \implies \cdots \implies false, \bar{B}$ in the TSLD-tree for $P \cup \{Q\}$, where Q is a generic query, since the output for the unification between A_i and $p(s_1, \ldots, s_n)$ would not return *wrong*. But then c would not be an erroneous clause. Therefore we proved the lemma. □

Effectively what this means is that if there is a type error in the program, then the erroneous clause is such that it will not be used to calculate the $T_P \uparrow \omega(\emptyset)$, since at least one of the atoms in its body will never be able to be used in an application of T_P. We can also have a ill-typed query, and we define it as follows.

Definition 15 - Ill-typed Query: Let P be a program. If any interpretation I derived from $T_P \uparrow \omega(\emptyset)$, such that I models P in some context Γ, is such that I is not a model of Q in the context Γ, then we say that Q is an ill-typed query with respect to P.

Even if the program is not ill-typed, if the query is ill-typed then we still are going to get a type error. A parallel can be made with functional programming languages, where even if a program is well-typed, we can call it with an ill-typed application of a function in the program and get a type error.

3.4.5 Soundness of TSLD-resolution

In this section we will prove that TSLD-resolution is sound, *i.e.*, if there is a successful derivation of a query Q in program P with a CAS θ, then every model of P is also a model of $\theta(Q)$; if there is a type error in the program, then the program is ill-typed; and if there is a type error in the query, the query is ill-typed with respect to the program. To prove this we will introduce the following auxiliary concept.

Definition 16 - Resultant: Suppose we have a TSLD-derivation step $Q_1 \implies \theta(Q_2)$. Then we define the resultant associated with this step as $\theta(Q_1) \leftarrow Q_2$.

Lemma 2 - Soundness of resultants: Let $Q_1 \implies \theta(Q_2)$ be a TSLD-derivation step using input clause c and r be the resultant associated with it. Then:

1. $c \models r$;
2. for any TSLD-derivation of $P \cup \{Q\}$ with resultants $r_1, \ldots, r_n$, $P \models r_i$ (for all $i \geq 0$).

Proof of this lemma for the SLD-resolution is in [40]. Since for unifiable terms the typed unification algorithm behaves like first-order unification, the proof still holds.

Theorem 6 - Soundness of TSLD-resolution: Let P be a program and Q a query. Then:

1. Suppose that there exists a successful derivation of $P \cup \{Q\}$, with the correct answer substitution θ. Then $P \models \theta(Q)$.
2. Suppose there is a type error in the program. Then P is ill-typed.
3. Suppose there is a type error in the query. Then Q is an ill-typed query with respect to P.

Proof: (1) Let $\theta_1, \ldots, \theta_n$ be the MGUs obtained in a successful derivation. Therefore, $\theta = \theta_n \circ \ldots \circ \theta_1$. The proof follows directly from lemma 2 applied to $P \models \theta_n \circ \cdots \circ \theta_1(Q) \leftarrow \square$.

(2) If there is a type error in the program, then there is an erroneous clause c in the TSLD-tree for $P \cup \{Q\}$, where Q is any generic query that uses c. Then by lemma 1 we know that there is at least one $A_i = p(t_1, \ldots, t_n)$, in the body of c, such that $\forall p(s_1, \ldots, s_n) \in T_P \uparrow \omega(\emptyset).\exists i \in [1, \ldots, n].domain([\![t_i]\!]) = domain([\![s_i]\!])$. Any interpretation I derived from $T_P \uparrow \omega(\emptyset)$ is such that for any Σ, $[\![A_i]\!]_{I,\Sigma} = wrong$. This implies that no such I is a model of c and, therefore, no such I is a model of P, which means P is ill-typed.

(3) This is the case where there is a type error in the query Q. Now, consider program $P \cup \{p() \leftarrow Q\}$, where p is a predicate that does not occur in P. Then note that every TSLD-derivation for a generic query with input clause $p() \leftarrow Q$ leads to $wrong$, so this is an erroneous clause. Therefore, from (2), no interpretation I derived from $T_P \uparrow \omega(\emptyset)$ models this clause. Since the set of atoms for p in $T_P \uparrow \omega(\emptyset)$ is empty, we can give any interpretation to p in any interpretation derived from $T_P \uparrow \omega(\emptyset)$. So we can choose an interpretation I derived from $T_P \uparrow \omega(\emptyset)$ such that I models $p()$ in some context Γ. Therefore, as no interpretation derived from $T_P \uparrow \omega(\emptyset)$ models $P \cup \{p() \leftarrow Q\}$ in any context Γ and we can build an interpretation which models predicate p in some context, no interpretation can model Q. Thus, by the definition of ill-typed query, Q is ill-typed with respect to P.

□

A short note about completeness. As for untyped SLD-resolution, completeness is related to the search for answers in a TSLD-tree. If we use Prolog sequential, top-down, depth-first search with backtracking, then it may result in incompleteness for same cases where the TSLD-tree is infinite, because the exploration of an infinite computation may defer indefinitely the exploration of some alternative computation capable of yielding a correct answer.

3.5 Extensions and Discussion

The major limitation of the semantics presented in this chapter is the fact that types are still far from what programmers would expect from a programming language. Functors are uninterpreted, such as in Prolog, in the sense that they are just symbols used to build new trees.

An obvious extension of this work is to extend the system to dynamically detect type errors relating to the semantic interpretation of some specific functors, for instance the list constructor. For this, we would have for the

list constructor not the Herbrand-based interpretation $[\ |\ ] :: \forall A, B.A \times B \rightarrow [A|B]$, but the following interpretation $[\ |\ ] :: \forall A.A \times list(A) \rightarrow list(A)$. Moreover, we would have the empty list $[\ ]$ with type $\forall D.list(D)$. This would necessarily change the typed unification algorithm by introducing a new kind of constraints. As an example, consider the unification $[1|2] = [1|2]$, where the second argument in both terms is not a list: considering a specially interpreted list constructor the result should be *wrong*, although the traditional untyped result is *true*. The same issues appear for arithmetic expressions. Arithmetic interpretations of $+, -, \times$, and $/$ can be introduced in the typed unification algorithm, so that in this context, unifications such as $a + b = a + b$ would now return *wrong* instead of *true*. These extensions are left for future work.

One very relevant consequence of this change is the fact that type errors, until now, occurred in the unification of terms from different types, but with a special interpretation of functors, we could have a type error in the term itself. For instance the term $[1|2]$ in the interpretation where $[\ |\ ] :: \forall A.A \times list(A) \rightarrow list(A)$, has a type error on itself, without performing unification. This dramatically changes the semantics and it is something we are going to discuss further in the next chapter.

Another limitation is the fact that TSLD-derivations cannot stop when reaching a value $false$ for a unification. In practical terms, this would drastically decrease the efficiency in certain cases, when comparing to SLD-resolution. What we argue is that a compromise could be reached in order to improve efficiency, where execution could stop when reaching the value $false$, without continuing, but there were no guarantees that there was no type error, it just was not detected. This would mean that only some type errors would be detected, both in the program and in the query, but it would be a more efficient approach for dynamic typing in logic programming.

The same could be said about unification, where we also continue when we reach the value $false$ in order to detect a type error further in the term. The effects on efficiency of this continuation are less significant than the TSLD-derivations. Even so, a compromise could be reached where unification halts with $false$ as soon as we reach that value, but in this case not all type errors would be detected.

There is a balance between making type checking complete and efficient that needs to be further analysed. However, a lot of these issues would go away if we could have type information at compile-time (statically), instead of dynamically. In the next chapter we present a type system where we can check if a program is well-typed, or detect type errors, statically.

Chapter 4

Static Typing

In the previous chapter, we presented and discussed a typed semantics that allows for the dynamic type checking of logic programs. In this chapter we present a static type system. A type system consists of a set of rules, through which we define which programs are well-typed at compile-time. Our type rules were first presented in [2], and later reformulated in [3]. We then prove that our type system is sound.

Note that when the type system was presented in [2], the semantics was slightly different and, because of that, new proofs for soundness are presented here.

4.1 Semantics

The declarative semantics we present here is based on the one presented in the previous chapter but with the following alterations: first of all, the basic domains are the same, but we include tree domains built from a set of constructors; secondly, the interpretations also interpret the function symbols, in addition to the predicate symbols, allowing for more interesting functions, other than the ones that build trees of some form.

We start by assuming that the Universe is divided into disjoint primitive domains as follows: $\mathbf{U} = \mathbf{Int} + \mathbf{Float} + \mathbf{String} + \mathbf{Nil} + \mathbf{Atom} + \mathbf{Cons} + \mathbf{F} + \mathbf{Bool} + \mathbf{W}$.

Int is the domain composed of all integer numbers, **Float** is the domain composed of all floating-point numbers, **String** is the domain composed of all strings, **Nil** is the domain composed of a single value representing an empty list, and **Atom** is the domain matching all symbolic constants. These are called *basic domains* and are the domains of constant values. **Cons** is the domain of constructors, each with some arity n. **F** is the domain of functions, such that each function maps values from a tuple of domains to another domain. **W** is the domain that contains the value *wrong*, standing for a type error. **Bool** is the domain which contains the values $true$ and $false$. A domain will be represented throughout the rest of the thesis by D or D_i, for some i.

Definition 17 - Simple Domains: *Simple domains* are defined by the following rules:

- A basic domain is a simple domain.
- If D_1 and D_2 are disjoint simple domains, then $D = D_1 + D_2$ is also a simple domain, that contains all values in D_1 and all values in D_2.
- If $D_1, \ldots, D_n$ are simple domains and F is a constructor from $Cons$ with arity n, then $D = F(D_1, \ldots, D_n)$ is also a simple domain that contains all trees with root as the functor F and the n-children being values from $D_1, \ldots, D_n$.

We allow any solution to these equations to be a domain, as long as the solution is not empty. For instance if we have $D = Nil + F(Int, D)$, this corresponds to the domain of lists of integers, where F is the list constructor, which is defined recursively. On the other hand, $D = F(D)$ has the empty domain as a solution, so it is not considered a simple domain.

Each ground term in our language is associated with a semantic value, contained in a simple domain, by an interpretation function. Let **Var** be an infinite and enumerable set of variables, **Func** be an infinite and enumerable set of function symbols and **Pred** be an infinite and enumerable set of predicate symbols.

There is an interpretation function $I :: \mathbf{Func} \cup \mathbf{Pred} \rightarrow \mathbf{Val}$ which associates each constant to their semantic value in a basic domain and each function symbol to a function f from a tuple of simple domains into a simple domain. Predicate symbols are associated with functions from a tuple of simple domains into the domain **Bool**. By definition, whenever a function is applied to a value outside its domain, the result is *wrong*.

Note that I is a function, thus for a given I, each constant, function symbol and predicate symbol can only be associated with one semantic value. In particular, since basic domains are disjoint, meaning $\forall i, j. i \neq j \implies D_i \cap D_j = \emptyset$, each semantic value for constants belongs to a unique basic domain. Complex terms can be associated with some value that is built from constructors, which we will call trees. These trees, can belong to several simple domains (consider the + operand), however, if we take the intersection of all simple domains, we get the smallest domain to which the tree belongs. We define $domain(v)$ as the smallest domain which contains v, if v is constant or a tree. We will use the same notation $domain(v)$ to denote the tuple with the domains of the arguments of v, if v is a function.

We shall now reintroduce the concept of a *state*, first introduced in the previous chapter. A state binds variables with semantic values. Each state Σ specifies a value, written $\Sigma(X)$, for each variable X of **Var**.

We assume that logic programs are normalized. In this representation, each predicate is defined by a single clause ($H : -B$), where the head H contains distinct variables as arguments and the body B is a disjunction (represented by the symbol ;) of queries. There are no common variables between queries, except for the variables that occur in the head of the clause, without loss of generality.

Example 25: Let *add* be a predicate defined by:

```
add(0,X,X).
add(s(X),Y,s(Z)) :- add(X,Y,Z).
```

The normal form of this predicate is:

```
rules
add(X1,X2,X3) :- ( X1 = 0, X2 = X, X3 = X ) ;
                 ( X1 = s(X'), X2 = Y, X3 = s(Z),
                 X4 = X', X5 = Y, X6 = Z, add(X4,X5,X6) ) .
```

Note that it is always possible to normalize a program using program transformation [71]. We will assume that predicate definitions are always in normal form.

Also note that in clauses in normal form $q(X_1, \ldots, X_n) : -b_1; \ldots; b_m.$, the same variable symbols $X_1, \ldots, X_n$ are used in the body $b_1; \ldots; b_m$ but denote possible different values in the different queries $b_1; \ldots; b_m$. Thus, to define the semantics of queries and clauses, we will need a list of possible different states. Each of the states $\Sigma_1, \ldots, \Sigma_m$ in the definition of the semantics for a clause will correspond to different variants of variables $X_1, \ldots, X_n$, one for each query b_i, for $1 \leq i \leq m$.

For simplicity of presentation, throughout the rest of this paper we will use Σ for a list with a single state Σ, and $\overrightarrow{\Sigma}$ for a list with several states, which can also appear explicitly $\overrightarrow{\Sigma} = [\Sigma_1, \ldots, \Sigma_n]$. The semantics of a term, given an interpretation I and a list of states is defined as follows:

$$[\![X]\!]_{I,\Sigma} = \Sigma(X)$$
$$[\![k]\!]_{I,\Sigma} = I(k)$$
$$[\![f(t_1, \ldots, t_n)]\!]_{I,\Sigma} = \textbf{if } (domain([\![t_1]\!]_{I,\Sigma}, \ldots, domain([\![t_n]\!]_{I,\Sigma}) \subseteq domain(I(f))$$
$$\textbf{then } I(f)([\![t_1]\!]_{I,\Sigma}, \ldots, [\![t_n]\!]_{I,\Sigma})$$
$$\textbf{else } wrong$$

Note that the domain check performed to the arguments of the complex term is necessary since we are not assuming the same interpretation for function symbols from the previous chapter, *i.e.*, they do not accept any

value. Instead, they have a certain input which is a subset of the Universe, such that some values are outside of this domain. And by definition, the application of a function to a value outside its domain returns *wrong*. This is exactly what we mean by having type errors in logic terms.

Example 26: Suppose we have the term [1|2], and in an interpretation I, $I(1) = 1$ and $domain(1) = Int$, $I(2) = 2$ and $domain(2) = Int$, and $I([\ |\ ]) = f$ and $domain(f) = Int \times List\_Int \to List\_Int$, where Int is a basic domain containing all integers and $List\_Int$ is the tree domain of all lists of integers.

Then we have a type error, since the second argument belongs to a domain that is not the domain expected by the function. This would be reflected in the semantics of this term, since for any Σ, $[\![[1|2]]\!]_{I,\Sigma} = wrong$.

The semantics for a predicate p with arity n corresponds to a function $I(p)$, given by I, that given values from simple domains $D_1 \times \cdots \times D_n$ outputs values in **Bool**, or in case the values do not belong to the domains of the function, returns the value *wrong*.

$[\![p]\!]_{I,\Sigma} = I(p)$, where $I(p) :: D_1 \times \cdots \times D_n \to Bool$.

Given this, the semantics for programs is given as follows:

$$
\begin{array}{l}
[\![t_1 = t_2]\!]_{I,\Sigma} = \\
\quad \textbf{if } ([\![t_1]\!]_{I,\Sigma} = [\![t_2]\!]_{I,\Sigma} \wedge [\![t_1]\!]_{I,\Sigma} \neq wrong) \\
\qquad \textbf{then } true \\
\quad \textbf{else} \\
\qquad \textbf{if } (domain([\![t_1]\!]_{I,\Sigma}) = domain([\![t_2]\!]_{I,\Sigma}) \wedge [\![t_1]\!]_{I,\Sigma} \neq wrong\ ) \\
\qquad\quad \textbf{then } false \\
\qquad \textbf{else } wrong \\
[\![p(t_1, \ldots, t_n)]\!]_{I,\Sigma} = \\
\quad \textbf{if } (domain([\![t_1]\!]_{I,\Sigma}), \ldots, domain([\![t_n]\!]_{I,\Sigma})) \subseteq domain([\![p]\!]_{I,\Sigma}) \\
\qquad \textbf{then } [\![p]\!]_{I,\Sigma}([\![t_1]\!]_{I,\Sigma}, \ldots, [\![t_n]\!]_{I,\Sigma}) \\
\quad \textbf{else } wrong \\
[\![g_1, \ldots, g_n]\!]_{I,\sigma} \ = \ [\![g_1]\!]_{I,\sigma} \wedge \ldots \wedge [\![g_n]\!]_{I,\Sigma} \\
[\![b_1; \ldots; b_m]\!]_{I,[\Sigma_1,\ldots,\Sigma_m]} \ = \ [\![b_1]\!]_{I,\Sigma_1} \vee \ldots \vee [\![b_m]\!]_{I,\Sigma_m} \\
[\![q(X_1, \ldots, X_n) : -b_1; \ldots; b_m.]\!]_{I,[\Sigma_1,\ldots,\Sigma_m]} \ = \ [\![b_1; \ldots; b_m]\!]_{I,[\Sigma_1,\ldots,\Sigma_m]} \implies \\
\qquad\qquad ([\![q(X_1, \ldots, X_n)]\!]_{I,[\Sigma_1]} \wedge \ldots \wedge [\![q(X_1, \ldots, X_n)]\!]_{I,[\Sigma_m]})
\end{array}
$$

Note that conjunction, disjunction and implication in the previous definitions are interpreted in the three-valued logic defined by the truth tables

presented in Chapter 3. Also note that, as different states are only needed for disjunctions, in the previous rules, the number of states in the list of states is one, except for the last two cases.

Example 27: Let I be an interpretation function such that $I(1) = 1$, and $I(a) = \text{a}$, where $domain(1) = \mathbf{Int}$, and $domain(\text{a}) = \mathbf{Atom}$. Also, let $I(p) = f$, such that $f :: D \rightarrow \mathbf{Bool}$, where $D = \mathbf{Int} + \mathbf{Atom}$. Let our program be defined as follows:

```
p(X) :- X = a ; X = 1.
```

Then, for $\Sigma_1 = [X \mapsto a]$ and $\Sigma_2 = [X \mapsto 1]$, we have the following semantics:

$[\![p(X) : -X = a; X = 1]\!]_{I,[\Sigma_1,\Sigma_2]} =$
$[\![X = a; X = 1]\!]_{I,[\Sigma_1,\Sigma_2]} \implies [\![p(X)]\!]_{I,\Sigma_1} \wedge [\![p(X)]\!]_{I,\Sigma_2} =$
$[\![X = a]\!]_{I,\Sigma_1} \vee [\![X = 1]\!]_{I,\Sigma_2} \implies [\![p(X)]\!]_{I,\Sigma_1} \wedge [\![p(X)]\!]_{I,\Sigma_2} =$
$true \vee true \implies true \wedge true = true$

The next function, called *or_degree*, gives the number of states needed for the semantics of disjunctions.

Definition 18 - ***or_degree*:** Let M be a term, an atom, a query, or a clause. Its *or_degree* is defined as follows:

- $or\_degree(M) = \text{k}$, if $M = b_1; \ldots; b_k$ or $M = p(X_1, \ldots, X_n) : -b_1; \ldots; b_k$.
- $or\_degree(M) = 1$, otherwise.

Example 28: $or\_degree(X = 1) = 1$
$or\_degree(p(X) : -X = 1) = 1$
$or\_degree(p(X) : -X = 1; X = a) = 2$

We have defined a three-valued semantics for logic programming. In order to determine which programs are well-typed (or ill-typed) we need to describe a type language and relate it to the semantics. That is what we will do in the next section.

4.2 Types

Types as they were described in the previous chapter had a major limitation of having a mandatory interpretation for function symbols that is not enough to describe a lot of interesting and widely used data structures, such as

lists and trees. We now define domains in a different way and we keep the property that every type for a term is associated with a simple domain, while functional types are associated with a subset of the domain of functions **F**. Since simple domains are not disjoint in general, then we can have types that have non-empty intersections with other types.

4.2.1 Syntax of Types

We now define a new class of expressions, which we shall call *types*. We first define the notion of *type term* built from an infinite set of type variables $TVar$, a finite set of base types $TBase$, an infinite set of function symbols $TFunc$, and an infinite set of type symbols, $TSymb$.

Type terms can be:

- a type variable $\alpha \in TVar$
- a base type $bs \in TBase$
- a type function symbol $f \in TFunc$ associated with an arity n $(n > 0)$ applied to an n-tuple of type terms
- a type symbol $\sigma \in TSymb$ associated with an arity n $(n \geq 0)$ applied to an n-tuple of type terms.

A *ground type term* is a type variable-free type term. Type symbols are defined in a *type definition*. Type definitions are of the form:

$$\sigma(\alpha_1, \ldots, \alpha_k) = \tau_1 + \ldots + \tau_n,$$

where each τ_i is a type term and σ is the type symbol being defined. In general these definitions are polymorphic, which means that type variables $\alpha_1, \ldots, \alpha_k$, for $k \geq 0$, are the type variables occurring in $\tau_1 + \ldots + \tau_n$, and are called *type parameters*. If we instantiate one of those type variables, we can replace it in the parameters and everywhere it appears on the right-hand side of the definition. The sum $\tau_1 + \ldots + \tau_n$ is a *union type*, describing values that may have one of the types $\tau_1, \ldots, \tau_n$, called the summands. The '+' is an idempotent, commutative, and associative operation. Throughout the rest of the thesis, to condense notation, we will use the symbol $\widetilde{\tau}$ to denote union types. We will also use the notation $\tau \in \widetilde{\tau}$ to denote that τ is a summand in the union type $\widetilde{\tau}$.

Note that type definitions may be recursive. *A deterministic type definition* [6] is a type definition where, on the right-hand side, none of τ_i starts with a type symbol and if τ_i is a type term starting with a type function symbol f, then no other τ_j starts with f.

Example 29: Assuming a base type *int* for the set of all integers, the type list of integers is defined by the type definition $list = [\,] + [int \mid list]$[1].

Let $\overrightarrow{\tau}$ stand for a tuple of types $\tau_1 \times \cdots \times \tau_n$. A *functional type* is a type of the form $\overrightarrow{\tau} \rightarrow \tau$. A *predicate type* is a functional type from a tuple of the type terms defining the types of its arguments to *bool*, *i.e.*, $\tau_1 \times \ldots \times \tau_n \rightarrow bool$. A *type* can be a *type term*, a *union type*, or a *functional type*.

Our type language enables parametric polymorphism through the use of type schemes. A *type scheme* is defined as $\forall_{\alpha_1} \ldots \forall_{\alpha_n} T$, where T is a predicate type and $\alpha_1, \ldots, \alpha_n$ are the type variables that occur in T. In logic programming, there have been several authors that have dealt with polymorphism with type schemes or in a similar way [6,9,10,51,54,56,72–74]. Type schemes have type variables as generic place-holders for ground type terms. Parametric polymorphism comes from the fact these type variables can be instantiated with any type. We say that a type σ_1 is an *instance* of a type scheme σ_2 and write $\sigma_1 \preceq \sigma_2$ iff either σ_2 is of the form $\forall\alpha_1 \ldots \alpha_n.\tau$ and there are types $\tau_1, \ldots, \tau_n$ such that $\sigma_1 = \tau[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]$, or $\sigma_1 = \sigma_2$.

Unless it is relevant in a particular situation, we will omit the $\forall\alpha_i$ in type schemes and the parameters of type symbols, for simplicity of presentation.

Example 30: A polymorphic list is defined by the following type definition: $list(\alpha) = [\,] + [\alpha \mid list(\alpha)]$

Now that we have a detailed description of the syntax of our type language, we need to know the semantics of types, meaning, which domain is associated with each type term and union type, and which subset of **F** is associated with each functional type.

4.2.2 Semantics of Types

We assume that each base type *bs* is associated with a basic domain. Therefore, there are exactly as many base types as there are basic domains, and we also know the association between them. Let $\sim$ denote the association between types and domains. If a base type *bs* is associated with a basic domain B_i, we will denote this by $bs \sim B_i$. The association between base types and basic domains is considered to be predefined.

The same is true for type function symbols and constructors. Every type function symbol $f \in TFunc$ is associated with a constructor $F \in \mathbf{Cons}$.

[1] Type definitions will use the user friendly Prolog notation for lists instead of the list constructor '.'

We can now expand the $\sim$ relation to include all ground type terms and ground union types, given the set of type definitions Δ defining type symbols, in the following way:

- $bs \sim D$ is predefined.
- $\tau_1 + \ldots + \tau_n \sim D_1 \cup \ldots \cup D_n \iff \tau_1 \sim D_1 \wedge \ldots \wedge \tau_n \sim D_n$.
- $f(\tau_1, \ldots, \tau_n) \sim F(D_1, \ldots, D_n) \iff f \sim F \wedge \forall i.\tau_i \sim D_i$.
- $\sigma \sim D \iff \Delta(\sigma) \sim D$, where $\sigma \sim D$ and $\Delta(\sigma)$ is the definition for the type symbol σ.

Note that our type language allows us to write type terms that are associated with empty domains. Consider, for instance, $\sigma = f(\sigma)$, which would be associated with a domain $D = F(D)$. Since we disallow empty domains and types are associated with a domain, we will also disallow such types. For the rest of this thesis, we assume all type terms are not associated with the empty domain.

Given the relation $\sim$, an interpretation I, a set of type definitions Δ defining type symbols, and a substitution for type variables S that binds each type variable in its domain to a ground type term, the semantics for types is given by the following rules. $\mathbf{T}[\![\ ]\!]$ defines the semantics of types of terms, which are type terms and union types, as well as tuples of types and the type *bool*, which is the type of the output of a predicate. This function uses an auxiliary function $\mathbf{GT}[\![\ ]\!]$ that defines the semantics of ground types for terms.

$$\mathbf{T}[\![\tau]\!]_\Delta = \bigcup_{\forall S} \mathbf{GT}[\![S(\tau)]\!]_\Delta$$

$\mathbf{GT}[\![bool]\!]_\Delta = \{true,\ false\}$
$\mathbf{GT}[\![bs]\!]_\Delta = D$, where $bs \sim D$
$\mathbf{GT}[\![\tau_1 + \ldots + \tau_n]\!]_\Delta = D$, where $\tau_1 + \ldots + \tau_n \sim D$.
$\mathbf{GT}[\![f(\tau_1, \ldots, \tau_n)]\!]_\Delta = F(D_1, \ldots, D_n)$, where $f(\tau_1, \ldots, \tau_n) \sim F(D_1, \ldots, D_n)$
$\mathbf{GT}[\![\sigma]\!]_\Delta = D$, where $\sigma \sim D$ and $\Delta(\sigma)$ is the definition of type symbol σ
$\mathbf{GT}[\![\tau_1 \times \cdots \times \tau_n]\!]_\Delta = \{(v_1, \ldots, v_n) \mid v_1 \in \mathbf{GT}[\![\tau_1]\!]_\Delta \wedge \ldots \wedge v_n \in \mathbf{GT}[\![\tau_n]\!]_\Delta\}$.

$\mathbf{P}[\![\ ]\!]$ defines the semantics of functional types, including predicate types. The semantics of these types is a subset of the domain F.

$$\mathbf{P}[\![\forall \alpha_1, \ldots, \alpha_m.\tau_1 \times \cdots \times \tau_n \to \tau]\!]_\Delta = \bigcap_{\forall S} \mathbf{GP}[\![S(\tau_1) \times \ldots \times S(\tau_n) \to S(\tau)]\!]_\Delta$$

$$\begin{aligned}&\mathbf{GP}[\![\tau_1 \times \cdots \times \tau_n \to \tau]\!]_\Delta \\ &\qquad = \{f \mid \forall (v_1, \ldots, v_n).(v_1, \ldots, v_n) \in \mathbf{GT}[\![\tau_1 \times \ldots \times \tau_n]\!]_\Delta \implies\end{aligned}$$

$$f(v_1, \ldots, v_n) \in \mathbf{GT}[\![\tau]\!]_\Delta\}$$

Note that for ground types, both $\mathbf{T}[\![\tau]\!]_\Delta = \mathbf{GT}[\![\tau]\!]_\Delta$ and $\mathbf{P}[\![\tau]\!]_\Delta = \mathbf{GP}[\![\tau]\!]_\Delta$, since $S(\tau) = \tau$ for every S.

We need to explain the rule for the semantics of a type symbol. Since type definitions can be recursive, then the domain associated with a type symbol may be a recursive one.

Example 31: Assume the type symbol σ, defined by the following definition $\sigma = [\,] + [int \mid \sigma]$. Then we get the following association:
$\sigma \sim D \iff [\,] + [int \mid \sigma] \sim D \iff$
$D = Nil + D\prime \wedge [\,] \sim Nil \wedge [int \mid \sigma] \sim D\prime \iff$
$D = Nil + list(\mathbf{Int}, D) \wedge [\,] \sim Nil \wedge [\mid] \sim list \wedge int \sim \mathbf{Int} \wedge \sigma \sim D$,
which finally means that $D = Nil + list(\mathbf{Int}, D)$, which is the domain of lists of integers.

Example 32: Now consider the predicate type for a predicate defining polymorphic lists: $\forall\alpha.\Big[list(\alpha) \rightarrow bool\Big]$. The semantics of this type is the set of functions (in this case predicates) which define lists of elements of type τ, for every ground instance of α. Therefore its semantics is the set of all functions that have polymorphic lists as the argument.

Tuple Distributivity

Most type languages in logic programming use tuple distributive closures of types. The notion of tuple distributivity was given by Mishra [20]. Here we recall the definition of tuple distributive types presented in Chapter 2.

Definition 19 - Tuple Distributive Types: Let τ be a type, corresponding to the set of semantic values V. We say that τ is a tuple distributive type if for every pair of values $f(v_1^1, \ldots, v_n^1) \in V$ and $f(v_1^2, \ldots, v_n^2) \in V$, we know that $V \supseteq \{f(v_1^{i_1}, \ldots, v_n^{i_n}) \mid 1 \leq i_1, \ldots, i_n \leq 2\}$.

Example 33: Let τ be a tuple distributive type containing the values $f(a, 1, c)$ and $f(2, b, d)$. Then we know that τ contains $f(a, b, c)$, $f(a, 1, d)$, $f(a, b, d)$, $f(2, b, c)$, $f(2, 1, c)$, and $f(2, 1, d)$ too.

Type inference is decidable for tuple distributive types, and it was proven in [6], that types defined by deterministic type definitions are tuple distributive.

However, some operations that are performed to sets of deterministic type definitions may change type definitions in a way so that they are no longer deterministic. We define an operation to transform a set of type definitions into a set of deterministic type definitions.

Definition 20 - Tuple Distributive Closure (TDC): Let Δ be a set of type definitions. We define $\overline{\Delta}$ as a set of type definitions such that $\forall\sigma.(\sigma = \overrightarrow{\tau}) \in \Delta$, $(\sigma = \overrightarrow{\tau\prime}) \in \overline{\Delta}$, and $\sigma = \overrightarrow{\tau\prime}$ is a deterministic type definition.

The algorithm to calculate the TDC of a set of type definitions is as follows.

Definition 21 - TDC Operation: Let Δ be a set of type definitions. Let H be the set of all type definitions in Δ. We apply the following rules:

1. $(\{\sigma = \sigma + \overrightarrow{\tau\prime}\} \cup \Delta\prime, H) \rightarrow (\{\sigma = \overrightarrow{\tau\prime}\} \cup \Delta\prime, H \cup \{\sigma = \overrightarrow{\tau\prime}\})$
2. $(\{\sigma = \sigma\prime + \overrightarrow{\tau\prime}\} \cup \Delta\prime, H) \rightarrow (\{\sigma = \Delta\prime(\sigma\prime) + \overrightarrow{\tau\prime}\} \cup \Delta\prime, H \cup \{\sigma = \Delta\prime(\sigma\prime) + \overrightarrow{\tau\prime}\})$
3. $(\{\sigma = f(\tau_1^1, \ldots, \tau_n^1) + \ldots + f(\tau_1^k, \ldots, \tau_n^k) + \overrightarrow{\tau\prime}\} \cup \Delta\prime, H) \rightarrow (\{\sigma = f(\sigma\prime_1, \ldots, \sigma\prime_n) + \overrightarrow{\tau\prime}\} \cup \Delta\prime\prime, H \cup \{\sigma = f(\sigma\prime_1, \ldots, \sigma\prime_n) + \overrightarrow{\tau\prime}, \sigma\prime_j = \tau_j^1 + \ldots + \tau_j^k\})$, where either $(\sigma\prime_i = \tau_i^1 + \ldots + \tau_i^k) \in H$, or for all j such that $(\sigma\prime_j = \tau_j^1 + \ldots + \tau_j^k) \notin H$, $\Delta\prime\prime = \Delta\prime \cup \{\sigma\prime_j = \tau_j^1 + \ldots + \tau_j^k\}$.

Let us show an example of how to calculate the TDC of a set of type definitions.

Example 34: Let $\Delta = \{\sigma_1 = [\,] + [int \mid \sigma_2], \sigma_2 = [float \mid \sigma_1], \sigma_3 = \sigma_1 + \sigma_2\}$ be a set of type definitions. We can see that the definition for the type rule defining σ_3 is not deterministic. Applying the TDC operation, step-by-step, is shown below:

$(\{\sigma_1 = [\,] + [int \mid \sigma_2], \sigma_2 = [float \mid \sigma_1], \sigma_3 = \sigma_1 + \sigma_2\}, \{\sigma_1 = [\,] + [int \mid \sigma_2], \sigma_2 = [float \mid \sigma_1], \sigma_3 = \sigma_1 + \sigma_2\}) \rightarrow$

$(\{\sigma_1 = [\,] + [int \mid \sigma_2], \sigma_2 = [float \mid \sigma_1], \sigma_3 = [\,] + [int \mid \sigma_2] + [float \mid \sigma_1]\}, \{\sigma_1 = [\,] + [int \mid \sigma_2], \sigma_2 = [float \mid \sigma_1], \sigma_3 = \sigma_1 + \sigma_2, \sigma_3 = [\,] + [int \mid \sigma_2] + [float \mid \sigma_1]\}) \rightarrow$

$(\{\sigma_1 = [\,] + [int \mid \sigma_2], \sigma_2 = [float \mid \sigma_1], \sigma_3 = [\,] + [\sigma_4 \mid \sigma_3], \sigma_4 = int + float\}, \{\sigma_1 = [\,] + [int \mid \sigma_2], \sigma_2 = [float \mid \sigma_1], \sigma_3 = \sigma_1 + \sigma_2, \sigma_3 = [\,] + [int \mid \sigma_2] + [float \mid \sigma_1], \sigma_4 = int + float\})$

This final set of type definitions only contains deterministic type definitions.

We will prove that the resulting set of type definitions contains only deterministic type definitions that include the tuple distributive closure of the original types.

Theorem 7 - Correctness of the TDC Operation: Let Δ be a set of type definitions. Then the TDC operation outputs $\overline{\Delta}$.

Proof: We suppose that TDC operation always terminates.

Now suppose the algorithm has terminated and output the set of type definitions $\Delta\prime$. Then all type definitions for type symbols in $\Delta\prime$ are deterministic because if they were not, then either a) there is a type symbol as a summand on a type definition, and rules 1 or 2 would still apply; or b) there are two type terms starting with the same function symbol on a type definition and rule 3 would still apply.

Since no type definition is deleted from the initial set of type definitions Δ, then for all $(\sigma = \widetilde{\tau}) \in \Delta$ there is $(\sigma = \widetilde{\tau\prime}) \in \Delta\prime$ and $\widetilde{\tau\prime}$ is deterministic, therefore $\Delta\prime = \overline{\Delta}$. □

Because the TDC operation calculates exactly the TDC $\overline{\Delta}$ of a set of type definitions Δ, we will abuse notation and represent the operation itself by $\overline{\Delta}$.

Suppose we have a domain D associated with a type symbol σ defined by a deterministic type definition in Δ. Then D is in a sense tuple distributive, since $D = D_1 + \ldots + D_n$, and all D_i are either base domains or a constructor applied to an n-tuple of domains, such that no D_i and D_j start with the same constructor, for $i \neq j$. We will call these *tuple distributive domains*.

4.2.3 Semantic Typing

We shall now define what is meant by a value v semantically having a type represented by σ, defined in the set of type definitions Δ. Note that values may have many types, or have no type at all. For example, the value *wrong* has no type.

Using $\mathbf{T}[\![\ ]\!]$ we can define, for each state Σ, a relation between the value associated with a variable in Σ and a type τ, by:

Definition 22 - Semantics of Assumptions: Let X be a variable and σ a type symbol defined in the set of type definitions Δ. Then for some interpretation I, $X :_{I,\Sigma,\Delta} \sigma \iff [\![X]\!]_{I,\Sigma} \in \mathbf{T}[\![\sigma]\!]_{\Delta}$.

An *assumption* is a type declaration for a variable, written $X : \sigma$, where X is a variable and σ is a type symbol. Variable X is called the *subject* of the assumption.

We define a *context* Γ as a set of assumptions with distinct variables as subjects (alternatively *contexts* can be defined as functions from variables to type symbols). We call $def(\Gamma)$ to the set of variables used as subjects in Γ. We can extend the above relation to contexts.

Definition 23 - Context Semantics: Given a context Γ, $[\![\Gamma]\!]_{I,\Sigma,\Delta} \iff \forall (X : \sigma) \in \Gamma.\ X :_{I,\Sigma,\Delta} \sigma$

One operation that we can perform on contexts is the *sum of contexts*. Basically, we sum the definitions of the type symbols associated with each variable. Note that by performing the union of two type definitions, the resulting type definition may end up not being deterministic. However, the sum of contexts is guaranteed to return a set of deterministic type definitions, since it performs the TDC operation.

Definition 24 - Sum of n Contexts: Let $\Gamma_1, \ldots, \Gamma_n$ be contexts and $\Delta_1, \ldots, \Delta_n$ be sets of type definitions, each Δ_i defining the types in Γ_i, such that no two type symbols are in common amongst the different Δs. Let V be the set of variables that occur in more than one context.
$\oplus\big((\Gamma_1, \ldots, \Gamma_n), (\Delta_1, \ldots, \Delta_n)\big) = (\Gamma, \overline{\Delta})$, where:
$\Gamma(X) = \sigma\prime$, where $\sigma\prime$ is a fresh type symbol, for all $X \in V$, and $\Gamma(X) = \Gamma_i(X)$, for all $X \notin V \wedge X \in def(\Gamma_i)$;
$\Delta(\sigma) = \Gamma_{i_1}(X) + \ldots + \Gamma_{i_k}(X)$, for all type symbols $\sigma \notin \Delta_1 \cup \cdots \cup \Delta_n$, such that Γ_{i_j} are the Γs where $X \in def(\Gamma_{i_j})$ occurs, and $\Delta(\sigma) = \Delta_i(\sigma)$, otherwise.

We will show an example of the sum of contexts.

Example 35: Let:
$\Gamma_1 = \{X : \sigma_1, Y : \sigma_2\}$,
$\Gamma_2 = \{X : \sigma_3, Z : \sigma_4\}$,
$\Delta_1 = \{\sigma_1 = f(int), \sigma_2 = atom\}$,
$\Delta_2 = \{\sigma_3 = f(\alpha) + g(\sigma_3), \sigma_4 = float\}$.

Then $\oplus((\Gamma_1,\Gamma_2),(\Delta_1,\Delta_2)) = (\Gamma,\Delta)$, where
$\Gamma = \{X:\sigma_5, Y:\sigma_2, Z:\sigma_4\}$
$\Delta = \{\sigma_1 = f(int), \sigma_2 = atom, \sigma_3 = f(\alpha) + g(float), \sigma_4 = float, \sigma_5 = f(\sigma_6) + g(\sigma_3), \sigma_6 = int + \alpha\}$.

Finally we give a semantic meaning to assertions of the form $\Gamma,\Delta \models_I M : \tau$ stating that if the assumptions in Γ hold, with type definitions Δ, then M yields a value of type τ.

Definition 25 - Semantic Typing: Let Γ be a context, Δ a set of type definitions, I an interpretation, M either a term, an atom, a query or a clause, and τ a type.
$\Gamma,\Delta \models_I M:\tau \iff \exists[(\Gamma_1,\Delta_1),\ldots,(\Gamma_n,\Delta_n)].\forall\overrightarrow{\Sigma} = [\Sigma_1,\ldots,\Sigma_n].$
$\Big[[\![\Gamma_1]\!]_{I,\Sigma_1,\Delta_1} \wedge \ldots \wedge [\![\Gamma_n]\!]_{I,\Sigma_n,\Delta_n} \implies [\![M]\!]_{I,\overrightarrow{\Sigma}} \in \mathbf{T}[\![\tau]\!]_\Delta\Big]$,
where $\oplus\big((\Gamma_1,\ldots,\Gamma_n),(\Delta_1,\ldots,\Delta_n)\big) = (\Gamma,\Delta)$ and $n = or\_degree(M)$. If $n = 1$, then the only sum possible is (Γ,Δ) itself.

Example 36: Let p be a predicate with the following predicate definition:

```
p(X) :- X = 1 ; X = a.
```

Let interpretation I be such that $I(\mathtt{1}) = 1$ and $I(\mathtt{a}) = a$ and B_1 and B_2 be two basic domains such that $1 \in B_1$ and $a \in B_2$. Let $I(\mathtt{p}) = f_p$, such that $f_p :: D \to Bool$, and $D = B_1 + B_2$.

Lets assume we have $\Gamma = \{X:\sigma\}$ and $\Delta = \{\sigma = int + atom\}$, where $int \sim B_1$ and $atom \sim B_2$. We will show that $\Gamma,\Delta \models_I (p(X) :- X = 1; X = a.) : bool$. This corresponds to showing that $\exists[(\Gamma_1,\Delta_1),(\Gamma_2,\Delta_2)].\forall\overrightarrow{\Sigma} = [\Sigma_1,\Sigma_2].\forall S.[[\![\Gamma_1]\!]_{I,\Sigma_1,\Delta_1,S} \wedge [[\![\Gamma_2]\!]_{I,[\Sigma_2]} \implies [\![p(X) :- X = 1; X = a.]\!]_{I,\overrightarrow{\Sigma}} \in \mathbf{T}[\![bool]\!]_{\Delta,S}$.

Suppose $\Gamma_1 = \{X:\sigma_1\}$, $\Delta_1 = \{\sigma_1 = int\}$, $\Gamma_2 = \{X:\sigma_2\}$, and $\Delta_2 = \{\sigma_2 = atom\}$. Then $\oplus\big((\Gamma_1,\Gamma_2),(\Delta_1,\Delta_2)\big) = (\Gamma,\Delta)$. If $\Sigma_1(X) \in B_1$ and $\Sigma_2(X) \in B_2$, the left-hand side of the implication is true. The right-hand side is also true, since applying $I(p)$ to any of the $\Sigma_i(X)$ does not return *wrong* and neither does any of the unifications on the bodies of the clause. Therefore the semantic value of the clause is either $true$ or $false$. If one of the $\Sigma_i(X)$ does not yield a value in the previous domains, the left-hand side of the implication is false, since $[\![X]\!]_{I,\Sigma_1} \notin \mathbf{T}[\![int]\!]_\Delta$ or $[\![X]\!]_{I,\Sigma_2} \notin \mathbf{T}[\![atom]\!]_\Delta$, thus the whole implication is trivially true.

We want to interpret programs as being a set of predicate definitions, where the clauses define what is accepted by a predicate. Therefore, it does not make sense to have interpretations where the body of a clause is $true$, or

$false$, but the head is $wrong$. We also want to have only tuple distributive types, so we require the domains involved in types for function symbols, or predicate symbols to be tuple distributive as well. We include the following property.

Definition 26 - Proper Interpretation: Let I be an interpretation and P be a program, we say that I is a proper interpretation of P, iff for every clause $p(X_1, \ldots, X_n) : -body.$ in P, we have that $\forall \overrightarrow{\Sigma} [\![p(X_1, \ldots, X_n) : -body.]\!]_{I,\overrightarrow{\Sigma}} = wrong \implies [\![body]\!]_{I,\overrightarrow{\Sigma}} = wrong$, and all domains in $I(f)$ for all function symbols and predicate symbols are tuple distributive domains.

The interpretation of the previous example is a proper interpretation. In fact, if we go back to the previous chapter and reason about the T_P operator, if the body of a clause is true, then we are assuming the head is too in any model for I. This is the intuition that led to this restriction.

Every type in our type language is inhabited, therefore, we also have the following property:

Lemma 3 - State Existence: Given an environment Γ, a set of type definitions Δ, there exists at least one state Σ, such that $[\![\Gamma]\!]_{I,\Sigma,\Delta}$ is true.

Proof: From the fact that every type is inhabited, for all $(X_i : \sigma_i) \in \Gamma$, there is at least one $v_i \in \mathbf{T}[\![\sigma_i]\!]_\Delta$. Let $\Sigma = [X_i \mapsto v_i]$. Then for this Σ, we have that $[\![\Gamma]\!]_{I,\Sigma,\Delta}$. □

So it will never be the case where a statement is trivially true because the left-hand side of an implication is trivially false.

Now that we have defined a semantics for logic programming, a type language, and we related the types with the semantics, we are ready to finally present the type system, that determines under which conditions a program is well-typed.

4.3 Type System

In this section we define a type system which, statically, relates logic programs with types. The type system defines a relation $\Gamma, \Delta \vdash_P M : \tau$, where Γ is a context as defined in the previous section, Δ is a set of deterministic type definitions, M is a term, an atom, a query, or a clause, and τ is a type. This relation should be read as expression M has type τ given the context Γ and the set of type definitions Δ, in program P.

We will write $\Gamma \cup \{X : \sigma\}$ to represent the context that contains all assumptions in Γ and the additional assumption $X : \sigma$ (note that because

each variable is unique as a subject of an assumption in a context, in $\Gamma \cup \{X : \sigma\}$, Γ does not contain assumptions with X as subject). Similarly $\Delta \cup \{\sigma = \widetilde{\tau}\}$ will represent the set of type definitions that contains all type definitions in Δ and the additional one $\sigma = \widetilde{\tau}$, such that σ is not already defined in Δ. We will write a sequence of variables $X_1, \ldots, X_n$ as $\overrightarrow{X}$, and a sequence of types as $\overrightarrow{\tau}$. We assume that clauses are normalized and, therefore, every call to a predicate in the body of a clause contains only variables.

The type system also uses a function *type* which gives the type of constants and function symbols. We assume that *type* is defined for all constants and function symbols that occur in P and types given by *type* never include *bool* and always have the right arity, *i.e.* for a function symbol of arity n it will be of the form $\tau_1 \times \cdots \times \tau_n \rightarrow \tau\prime$. If $n = 0$, *i.e.*, for a constant, then the output is a type term τ.

Since predicate definitions can be parametrically polymorphic or we can have several queries in the body of a clause assuming different types for some variables, in general calls to predicates will use a subtype of the whole type for the predicate.

We will define a subtyping relation, which will be used in the type system.

Definition 27 - Subtyping: Let Δ be a set of type definitions, S_1 and S_2 be substitutions for type variables, and Λ be a set of equations of the form $\tau \sqsubseteq_\Delta \tau\prime$. A statement of the form $\tau \sqsubseteq_\Delta \tau\prime$ is true iff $\forall S_1.\exists S_2.\emptyset, \Delta \vdash S_1(\tau) \sqsubseteq S_2(\tau\prime)$, where $\Lambda, \Delta \vdash \tau_1 \sqsubseteq \tau_2$ is given by the rules in Figure 4.1.

Note that the rules are only applicable to ground types.

Subtyping of functional types is contravariant in the argument type, meaning that the order of subtyping is reversed. This is standard in functional languages and guarantees that when a functional type $\overrightarrow{\tau_1} \rightarrow \tau_2$ is a subtype of another $\overrightarrow{\tau\prime_1} \rightarrow \tau\prime_2$ it is safe to use a function f of type $\overrightarrow{\tau_1} \rightarrow \tau_2$ in place of a function g of type $\overrightarrow{\tau\prime_1} \rightarrow \tau\prime_2$, since f accepts as input all values that g does, and possibly more, returning values in τ_2 which are also in $\tau\prime_2$. For example, predicates of type $int + float \rightarrow bool$ can be used wherever an $int \rightarrow bool$ was expected.

Lemma 4 - Soundness of the Subtyping Rules: Let $\emptyset, \Delta \vdash \tau_1 \sqsubseteq \tau_2$, then either, $\mathbf{GT}[\![\tau]\!]_\Delta \subseteq \mathbf{GT}[\![\tau\prime]\!]_\Delta$ if τ and $\tau\prime$ are types of terms, or $\mathbf{GP}[\![\tau]\!]_\Delta \subseteq \mathbf{GP}[\![\tau\prime]\!]_\Delta$ if τ and $\tau\prime$ are functional types.

Proof: The proof follows from induction on the rules for subtyping.

- [Reflexivity]: It is trivially true, from set theory.

$$[\textsc{Reflexivity}]\ \Lambda, \Delta \vdash \tau \sqsubseteq \tau$$

$$[\textsc{Transitivity}]\ \frac{\Lambda, \Delta \vdash \tau \sqsubseteq \tau\prime \quad \Lambda, \Delta, S \vdash \tau\prime \sqsubseteq \tau\prime\prime}{\Lambda, \Delta \vdash \tau \sqsubseteq \tau\prime\prime}$$

$$[\textsc{Assumption}]\ \Lambda \cup \{\tau \sqsubseteq \tau\prime\}, \Delta \vdash \tau \sqsubseteq \tau\prime$$

$$[+\ \textsc{Introduction}]\ \frac{\Lambda, \Delta \vdash \tau_1 \sqsubseteq \tau \quad \ldots \quad \Lambda, \Delta \vdash \tau_n \sqsubseteq \tau}{\Lambda, \Delta \vdash \tau_1 + \cdots + \tau_n \sqsubseteq \tau}$$

$$[+\ \textsc{Elimination}]\ \frac{\Lambda, \Delta \vdash \tau \in \widetilde{\tau}}{\Lambda, \Delta \vdash \tau \sqsubseteq \widetilde{\tau}}$$

$$[\textsc{Complex}]\ \frac{\Lambda, \Delta \vdash \tau_1 \sqsubseteq \tau_1\prime \quad \ldots \quad \Lambda, \Delta \vdash \tau_n \sqsubseteq \tau_n\prime}{\Lambda, \Delta \vdash f(\tau_1, \ldots, \tau_n) \sqsubseteq f(\tau_1\prime, \ldots, \tau_n\prime)}$$

$$[\textsc{Left Symbol}]\ \frac{\Lambda \cup \{\sigma \sqsubseteq \tau\}, \Delta \vdash \widetilde{\tau} \sqsubseteq \tau \quad \Delta(\sigma) = \widetilde{\tau}}{\Lambda, \Delta \vdash \sigma \sqsubseteq \tau}$$

$$[\textsc{Right Symbol}]\ \frac{\Lambda \cup \{\tau \sqsubseteq \sigma\}, \Delta \vdash \tau \sqsubseteq \widetilde{\tau} \quad \Delta(\sigma) = \widetilde{\tau}}{\Lambda, \Delta \vdash \tau \sqsubseteq \sigma}$$

$$[\textsc{Both Symbol}]\ \frac{\Lambda \cup \{\sigma_1 \sqsubseteq \sigma_2\}, \Delta \vdash \widetilde{\tau_1} \sqsubseteq \widetilde{\tau_2} \quad \Delta(\sigma_1) = \widetilde{\tau_1} \quad \Delta(\sigma_2) = \widetilde{\tau_2}}{\Lambda, \Delta \vdash \sigma_1 \sqsubseteq \sigma_2}$$

$$[\textsc{Tuple}]\ \frac{\Lambda, \Delta \vdash \tau_1 \sqsubseteq \tau\prime_1 \quad \ldots \quad \Lambda, \Delta \vdash \tau_n \sqsubseteq \tau\prime_n}{\Lambda, \Delta \vdash \tau_1 \times \cdots \times \tau_n \sqsubseteq \tau\prime_1 \times \cdots \times \tau\prime_2}$$

$$[\textsc{Contravariance}]\ \frac{\Lambda, \Delta \vdash \tau\prime_1 \sqsubseteq \tau_1 \quad \Lambda, \Delta \vdash \tau_2 \sqsubseteq \tau\prime_2}{\Lambda, \Delta \vdash \tau_1 \to \tau_2 \sqsubseteq \tau\prime_1 \to \tau\prime_2}$$

FIGURE 4.1: Subtyping relation rules ($\Gamma, \Delta \vdash \tau \sqsubseteq \tau\prime$)

- [Transitivity]: It is trivially true, from set theory.
- [Assumption]: Since we are assuming Λ to be true, then it is trivially true.
- [+ Introduction]: Suppose that we have, for $i = 1, \ldots, n$, $\emptyset, \Delta \vdash \tau_i \sqsubseteq \tau$. Then by the induction hypothesis, $\forall i. \mathbf{GT}[\![\tau_i]\!]_\Delta \subseteq \mathbf{GT}[\![\tau]\!]_\Delta$. So, it is also true that $\mathbf{GT}[\![\tau_1]\!]_\Delta \cup \ldots \cup \mathbf{GT}[\![\tau_n]\!]_\Delta \subseteq \mathbf{GT}[\![\tau]\!]_\Delta$, from set theory. Therefore $\mathbf{GT}[\![\tau_1 + \ldots + \tau_n]\!]_\Delta = \mathbf{GT}[\![\tau_1]\!]_\Delta \cup \ldots \cup \mathbf{GT}[\![\tau_n]\!]_\Delta \subseteq \mathbf{GT}[\![\tau]\!]_\Delta$.
- [+ Elimination]: Suppose that we have $\tau \in \widetilde{\tau}$. Then we know that $\widetilde{\tau} = \tau + \ldots$. So, $\mathbf{GT}[\![\widetilde{\tau}]\!]_\Delta = \mathbf{GT}[\![\tau]\!]_\Delta \cup \ldots$. Therefore, we can then conclude that $\mathbf{GT}[\![\tau]\!]_\Delta \subseteq \mathbf{GT}[\![\widetilde{\tau}]\!]_\Delta$.

- [Complex]: Suppose that $\forall i.\emptyset, \Delta \vdash \tau_i \sqsubseteq \tau_i\prime$. Then, by the induction hypothesis we also know that $\forall i.\mathbf{GT}[\![\tau_i]\!]_\Delta = D_i \subseteq D_i\prime = \mathbf{GT}[\![\tau_i\prime]\!]_\Delta$. Let $f \sim F$, $\tau_i \sim D_i$ and $\tau_i\prime \sim D_i\prime$. Then $\mathbf{GT}[\![f(\tau_1, \ldots, \tau_n)]\!]_\Delta = F(D_1, \ldots, D_n)$ contains all trees with root F and n-children with values from $D_1, \ldots, D_n$, but since $D_i \subseteq D_i\prime$, all of these trees are also in $F(D_1\prime, \ldots, D_n\prime)$, so $F(D_1, \ldots, D_n) \subseteq F(D_1\prime, \ldots, D_n\prime)$. But, since $\mathbf{GT}[\![f(\tau_1\prime, \ldots, \tau_n\prime)]\!]_\Delta = F(D_1\prime, \ldots, D_n\prime)$, we can finally conclude that $\mathbf{GT}[\![f(\tau_1, \ldots, \tau_n)]\!]_\Delta \subseteq \mathbf{GT}[\![f(\tau_1\prime, \ldots, \tau_n\prime)]\!]_\Delta$.

- [Left Symbol]: Suppose that $\{\sigma \sqsubseteq \tau\}, \Delta \vdash \widetilde{\tau} \sqsubseteq \tau$, where $\Delta(\sigma) = \widetilde{\tau}$. By the induction hypothesis, we know that $\emptyset \cup \{\sigma \sqsubseteq \tau\} \vdash \widetilde{\tau} \sqsubseteq \tau$, so $\mathbf{GT}[\![\widetilde{\tau}]\!]_\Delta = D \subseteq D\prime = \mathbf{GT}[\![\tau]\!]_\Delta$ if we assume that $\sigma \sqsubseteq \tau$. But $\mathbf{GT}[\![\sigma]\!]_\Delta = D$, since $\sigma \sim D \iff \Delta(\sigma) \sim D$. Therefore we can conclude that $\emptyset, \Delta \vdash \sigma \sqsubseteq \tau$. So we conclude that $\mathbf{GT}[\![\sigma]\!]_\Delta = D \subseteq \mathbf{GT}[\![\tau]\!]_\Delta$.

- [Right Symbol]: The proof for the Right Symbol rule is similar to the previous one.

- [Both Symbol]: Suppose that $\{\sigma_1 \sqsubseteq \sigma_2\}, \Delta \vdash \widetilde{\tau_1} \sqsubseteq \widetilde{\tau_2}$, where $\Delta(\sigma_1) = \widetilde{\tau_1}$ and $\Delta(\sigma_2) = \widetilde{\tau_2}$. Then, by the induction hypothesis, we know that $\mathbf{GT}[\![\widetilde{\tau_1}]\!]_\Delta = D_1 \subseteq D_2 = \mathbf{GT}[\![\widetilde{\tau_2}]\!]_\Delta$. But, since $\sigma_1 \sim D_1 \iff \Delta(\sigma_1) \sim D_1$ and $\sigma_2 \sim D_2 \iff \Delta(\sigma_2) \sim D_2$, we have $\mathbf{GT}[\![\sigma_1]\!]_\Delta = D_1 \subseteq D_2\mathbf{GT}[\![\sigma_2]\!]_\Delta$.

- [Tuple]: Suppose that $\forall i.\emptyset, \Delta \vdash \tau_i \sqsubseteq \tau\prime_i$. By the induction hypothesis, we know that $\forall i.\mathbf{GT}[\![\tau_i]\!]_\Delta \subseteq \mathbf{GT}[\![\tau\prime_i]\!]_\Delta$, so by the definition of $\mathbf{GT}[\![\,]\!]_\Delta$, we know that $\mathbf{GT}[\![\tau_1 \times \ldots \times \tau_n]\!]_\Delta \subseteq \mathbf{GT}[\![\tau\prime_1 \times \ldots \times \tau\prime_n]\!]_\Delta$.

- [Contravariance]: Suppose that $\emptyset, \Delta \vdash \tau\prime_1 \sqsubseteq \tau_1$ and $\emptyset, \Delta \vdash \tau_2 \sqsubseteq \tau\prime_2$. Then we know that $\mathbf{GT}[\![\tau\prime_1]\!]_\Delta \subseteq \mathbf{GT}[\![\tau_1]\!]_\Delta$ and $\mathbf{GT}[\![\tau_2]\!]_\Delta \subseteq \mathbf{GT}[\![\tau\prime_2]\!]_\Delta$. Also, from the semantics we have $\mathbf{GP}[\![\tau_1 \to \tau_2]\!]_\Delta = \{f_1 \mid \forall v_1 \in \mathbf{GT}[\![\tau_1]\!]_\Delta \; f_1(v_1) \in \mathbf{GT}[\![\tau_2]\!]_\Delta\}$, and $\mathbf{P}[\![\tau\prime_1 \to \tau\prime_2]\!]_\Delta = \{f_2 \mid \forall v_2 \in \mathbf{GT}[\![\tau\prime_1]\!]_\Delta \; f_2(v_2) \in \mathbf{GT}[\![\tau\prime_2]\!]_\Delta\}$. But, since all v_2 are also in $\mathbf{GT}[\![\tau_1]\!]_\Delta$, then for all v_2, $f_1(v_2) \in \mathbf{GT}[\![\tau_2]\!]_\Delta$. Also, since $\mathbf{GT}[\![\tau_2]\!]_\Delta \subseteq \mathbf{GT}[\![\tau\prime_2]\!]_\Delta$, all $f_1(v_2) \in \mathbf{GT}[\![\tau\prime_2]\!]_\Delta$, which means that all f_1 are also in $\mathbf{GP}[\![\tau\prime_1 \to \tau\prime_2]\!]_\Delta$. Therefore $\mathbf{GP}[\![\tau_1 \to \tau_2]\!]_\Delta \subseteq \mathbf{GP}[\![\tau\prime_1 \to \tau\prime_2]\!]_\Delta$.

□

Now that we have proven that subtyping in sound for ground types, we will prove it is also sound for types that contain variables.

Theorem 8 - Soundness of Subtyping: Let τ_1 and τ_2 be types. If $\tau_1 \sqsubseteq_\Delta \tau_2$ then either $\mathbf{T}[\![\tau_1]\!]_\Delta \subseteq \mathbf{T}[\![\tau_2]\!]_\Delta$, if τ_1 and τ_2 are types of terms, or $\mathbf{P}[\![\tau_1]\!]_\Delta \subseteq \mathbf{P}[\![\tau_2]\!]_\Delta$, if τ_1 and τ_2 are functional types.

Proof: Suppose τ_1 and τ_2 are types of terms. We know that $\tau_1 \sqsubseteq_\Delta \tau_2$ implies $\forall S_1.\exists S_2.\emptyset, \Delta \vdash S_1(\tau_1) \sqsubseteq S_2(\tau_2)$. We also know that $\mathbf{T}[\![\tau_1]\!]_\Delta = \bigcup_{\forall S} \mathbf{GT}[\![S(\tau_1)]\!]_\Delta$. But from Lemma 4, since for all S there is a substitution $S\prime$ such that $\emptyset, \Delta \vdash S(\tau_1) \sqsubseteq S\prime(\tau_2)$, then $\mathbf{GT}[\![S(\tau_1)]\!]_\Delta \subseteq \mathbf{GT}[\![S\prime(\tau_2)]\!]_\Delta$. So, $\bigcup_{\forall S} \mathbf{GT}[\![S(\tau_1)]\!]_\Delta \subseteq \bigcup_{\forall S\prime} \mathbf{GT}[\![S\prime(\tau_2)]\!]_\Delta$, which ultimately implies $\mathbf{T}[\![\tau_1]\!]_\Delta \subseteq \mathbf{T}[\![\tau_2]\!]_\Delta$.

The proof is similar for functional types. □

Some type symbols may be representing the same set of values, having similar, but not equal definitions. We define a property of types that deals with this.

Definition 28 - Type Equivalence: Let τ_1 and τ_2 be types, and Δ be a set of type definitions. We say τ_1 and τ_2 are equivalent and represent it by $\tau_1 \equiv_\Delta \tau_2$, if for every S, $S(\tau_1) \sqsubseteq_\Delta S(\tau_2)$ and $S(\tau_2) \sqsubseteq_{\Delta)} S(\tau_1)$.

Lemma 5 - Equivalent Types: If $\tau_1 \equiv_\Delta \tau_2$, then $\mathbf{T}[\![\tau_1]\!]_\Delta = \mathbf{T}[\![\tau_2]\!]_\Delta$.

Proof: Since $\tau_1 \equiv_\Delta \tau_2$, then for every S, $S(\tau_1) \sqsubseteq_\Delta S(\tau_2)$ and $S(\tau_2) \sqsubseteq_\Delta S(\tau_1)$. Since $S(\tau_1) \sqsubseteq_\Delta S(\tau_2)$, then we know that $\mathbf{T}[\![S(\tau_1)]\!]_\Delta \subseteq \mathbf{T}[\![S(\tau_2)]\!]_\Delta$. Since $S(\tau_2) \sqsubseteq_\Delta S(\tau_1)$, then we know that $\mathbf{T}[\![S(\tau_2)]\!]_{\Delta)} \subseteq \mathbf{T}[\![S(\tau_1)]\!]_\Delta$. Therefore for every S, $\mathbf{T}[\![S(\tau_1)]\!]_\Delta = \mathbf{T}[\![S(\tau_2)]\!]_\Delta$, and $\mathbf{GT}[\![S(\tau_1)]\!]_\Delta = \mathbf{GT}[\![S(\tau_2)]\!]_\Delta$, because $S(\tau_1)$ and $S(\tau_2)$ are ground. Therefore, $\mathbf{T}[\![\tau_1]\!]_\Delta = \bigcup_{\forall S} \mathbf{GT}[\![S(\tau_1)]\!]_\Delta = \mathbf{T}[\![S(\tau_1)]\!]_\Delta = \mathbf{T}[\![S(\tau_2)]\!]_\Delta = \bigcup_{\forall S} \mathbf{GT}[\![S(\tau_1 2)]\!]_\Delta = \mathbf{T}[\![\tau_2]\!]_\Delta$, so $\mathbf{T}[\![\tau_1]\!]_\Delta = \mathbf{T}[\![\tau_2]\!]_\Delta$. □

Equivalent types represent the same set of values as proven in Lemma 5. Moreover, they have the same type variable in the same position of the syntactic tree of the type. This means that either they are both type symbols that have equal definitions up to the unfolding of type definitions (due to recursion), or one of them is a type symbol and the other is the definition of that type symbol, or a combination of the two.

Now we present a type system (see Figure 4.2) defining the typing relation, which relates terms, atoms, queries, and predicate definitions with types. If there is a context Γ, a set of type definitions Δ, and a type τ such that $\Gamma, \Delta \vdash_P M : \tau$ we say that M is *(statically) well-typed*. This type system can be implemented to type check programs. For that, one would provide the required sets: Γ and Δ, p and the type τ, and you would follow the rules in the type system to see if a derivation can be constructed. We will later define a type inference algorithm, and for the algorithm to be decidable, the following property must be ensured.

$$\text{VAR}\ \frac{}{\Gamma \cup \{X : \sigma\}, \Delta \cup \{\sigma = \widetilde{\tau}\} \vdash_P X : \sigma}$$

$$\text{CST}\ \frac{type(c) = \tau\prime \quad \tau \preceq \tau\prime}{\Gamma, \Delta \vdash_P c : \tau} \qquad \text{CPL}\ \frac{\begin{array}{c} type(f) = \sigma \\ \tau\prime_1 \times \cdots \times \tau\prime_n \to \tau \preceq \sigma \\ \tau_1 \equiv_\Delta \tau_1 \quad \ldots \quad \tau_n \equiv_\Delta \tau\prime_n \\ \Gamma, \Delta \vdash_P t_1 : \tau_1 \quad \ldots \quad \Gamma, \Delta \vdash_P t_n : \tau_n \end{array}}{\Gamma, \Delta \vdash_P f(t_1, \ldots, t_n) : \tau}$$

$$\text{UNF}\ \frac{\begin{array}{c} \tau_1 \equiv_\Delta \tau_2 \\ \Gamma, \Delta \vdash_P t_1 : \tau_1 \quad \Gamma, \Delta \vdash_P t_2 : \tau_2 \end{array}}{\Gamma, \Delta \vdash_P t_1 = t_2 : bool}$$

$$\text{CLL}\ \frac{\begin{array}{c} \overrightarrow{\sigma\prime} \to bool \sqsubseteq_{\Delta \cup \{\overrightarrow{\sigma\prime} = \overrightarrow{\tau\prime}, \overrightarrow{\sigma} = \overrightarrow{\tau}\}} \overrightarrow{\sigma} \to bool \quad p(\overrightarrow{X}) : -body. \in P \\ \Gamma \cup \{\overrightarrow{Y} : \overrightarrow{\sigma\prime}\}, \Delta \cup \{\overrightarrow{\sigma\prime} = \overrightarrow{\tau\prime}\} \vdash_P (p(\overrightarrow{Y}) : -body.) : bool \end{array}}{\Gamma\prime \cup \{\overrightarrow{X} : \overrightarrow{\sigma}\}, \Delta\prime \cup \{\overrightarrow{\sigma} = \overrightarrow{\tau}\} \vdash_P p(\overrightarrow{X}) : bool}$$

$$\text{CON}\ \frac{\Gamma, \Delta \vdash_P g_1 : bool \quad \ldots \quad \Gamma, \Delta \vdash_P g_n : bool}{\Gamma, \Delta \vdash_P g_1, \ldots, g_n : bool}$$

$$\text{CLS}^{(a)}\ \frac{\begin{array}{c} \oplus((\Gamma_1, \ldots, \Gamma_m), (\Delta_1, \ldots, \Delta_m)) = (\Gamma, \Delta) \\ \Gamma_1, \Delta_1 \vdash_P b_1 : bool \quad \ldots \quad \Gamma_m, \Delta_m \vdash_P b_m : bool \end{array}}{\Gamma, \Delta \vdash_P (p(\overrightarrow{X}) : -b_1; \ldots; b_m.) : bool}$$

$$\text{RCLS}^{(b)}\ \frac{\Gamma \cup \{\overrightarrow{X} : \overrightarrow{\tau}, \overrightarrow{Y_i} : \overrightarrow{\tau}\}, \Delta \vdash_P p(\overrightarrow{X}) : -b_1; \ldots; b_{m+n}. : bool}{\begin{array}{c} \Gamma \cup \{\overrightarrow{X} : \overrightarrow{\tau}, ref\overrightarrow{Y_i} : \overrightarrow{\tau}\}, \Delta \vdash_P (p(\overrightarrow{X}) : -b_1; \ldots; b_m; \\ b_{m+1}, p(\overrightarrow{Y}_{11}), \ldots, p(\overrightarrow{Y}_{1k_1}); b_{m+n}, p(\overrightarrow{Y}_{n1}), \ldots, p(\overrightarrow{Y}_{nk_n}).) : bool \end{array}}$$

(a) This rule is for non-recursive predicates only.
(b) This rule is for recursive predicates. Note that all variables in recursive calls in a certain sequence of goals have the same type as the variables in the head in that clause. Also $\overrightarrow{Y_i}$ represents all $i = 1, \ldots, k_n$

FIGURE 4.2: Type System

Definition 29 - Monomorphism Restriction: Let p be a recursive predicate of arity n, typed with type $\tau_1 \times \cdots \times \tau_n \to bool$ using a context Γ, and a set of type definitions Δ. Then, the types of the variables $X_1, \ldots, X_n$ for all recursive calls of p are $\tau_1, \ldots, \tau_n$, respectively, in Γ.

It is well-known that type inference in the presence of polymorphic recursion is not decidable [56, 75], thus we do not allow polymorphic recursion in the system. This is achieved by the previous restriction on recursive predicates. We choose to define this restriction locally in each predicate definition for the sake of simplicity of presentation. The alternative would be to define a new syntax for logic programming to group together mutually recursive predicates as a single syntactic entity (in functional programming

this would correspond to nested *letrec* expressions). The *monomorphism restriction* (Definition 29) holds in our type system by rule RCLS for typing recursive predicates. In this rule we use the same type for the variables in the head of the clause in its recursive calls in the body.

Let us describe the other rules of the type system. Rule Var types a variable with the type it has in the context.

Rule CST says that we can type a constant with any instance of its type. In general, types for constants are base types, so the only instance is the type itself. However, a constant can be the base case in some polymorphic recursive type (such as [ ] in lists). In that case, we can say that the constant has any instance of its type.

Rule CPL types complex terms with any type that is equivalent with an instance of the type for the function symbol. So a complex term $f(t_1, \ldots, t_n)$ has any type that is equivalent to an instance of $type(f)$ and the output type is the output type of that instance.

Rule UNF types an equality as *bool* if the types for both sides of the equality are equivalent.

Rule CLL types predicate (non-recursive) calls: for a call to a predicate p to be well typed, the type for each variable in the call needs to be a subtype of the type of the variables in the definition of p in program P. Note that the rest of the context and set of type definitions have nothing in common, since we just require that the types for the call are a subtype for the types of the predicate in the definition.

Rule CON just checks that every goal is *bool* using the same context and set of type definitions.

Rule CLS types non-recursive clauses: if we type each body of a clause using some context and some set of type definitions, then we can type the entire clause with the sum of all those contexts.

Note that, from rules CLS and RCLS, the type of a clause is *bool*. However, the interesting type information is the type for a predicate, determined by the types of its arguments which, in the end of the type derivation, are in context Γ.

Example 37: Here we give an example of a type derivation. Let p be a predicate defined by *p(X) :- X = 1; X = a.*. Let *type* be such that $type(1) = int$ and $type(a) = atm$. For simplicity of presentation let: $\Gamma_1 = \{X : \sigma_1\}$, $\Delta_1 = \{\sigma_1 = int\}$,
$\Gamma_2 = \{X : \sigma_2\}$, $\Delta_2 = \{\sigma_2 = atm\}$,
$\Gamma_3 = \{X : \sigma_3\}$, and $\Delta_3 = \{\sigma_3 = int + atm\}$.

Note that $\oplus\big((\Gamma_1, \Gamma_2), (\Delta_1, \Delta_2)\big) = (\Gamma_3, \Delta_3)$.

By two applications of rule UNF followed by an application of rule CLS we have:

$$\text{UNF}\ \frac{\sigma_1 \cong \mathit{int} \qquad \text{VAR}\ \dfrac{}{\Gamma_1, \Delta_1 \vdash_P X : \sigma_1} \qquad \text{CST}\ \dfrac{\mathit{type}(1) = \mathit{int} \quad \mathit{int} \preceq \mathit{int}}{\Gamma_1, \Delta_1 \vdash_P 1 : \mathit{int}}}{\Gamma_1, \Delta_1 \vdash_P X = 1 : \mathit{bool}}$$

$$\text{UNF}\ \frac{\sigma_2 \cong \mathit{atm} \qquad \text{VAR}\ \dfrac{}{\Gamma_2, \Delta_2 \vdash_P X : \sigma_2} \qquad \text{CST}\ \dfrac{\mathit{type}(a) = \mathit{atm} \quad \mathit{atm} \preceq \mathit{atm}}{\Gamma_2, \Delta_2 \vdash_P a : \mathit{atm}}}{\Gamma_2, \Delta_2 \vdash_P X = a : \mathit{bool}}$$

$$\text{CLS}\ \frac{\Gamma_1, \Delta_1 \vdash_P X = 1 : \mathit{bool} \quad \Gamma_2, \Delta_2 \vdash_P X = a : \mathit{bool}}{\Gamma_3, \Delta_3 \vdash_P p(X) : -X = 1; X = a. : \mathit{bool}}$$

From the type of X in the final context, the type of p is $\mathit{int} + \mathit{atom} \to \mathit{bool}$.

The function *type* corresponds to type declaration for constants and function symbols. Constant, in general are in base types, but some constants are used as the base case in some (possibly polymorphic) recursive type. For instance, if we have the list type $\mathit{list}(\alpha) = [\,] + [\alpha \mid \mathit{list}(\alpha)]$ then in the type system the function *type* assigns the type $\mathit{list}(\alpha)$ to the constant $[\,]$ and the type $\alpha \times \mathit{list}(\alpha) \to \mathit{list}(\alpha)$ to the function symbol $[\mid]$.

These declaration are defined in *type*, thus the correctness of the type system, which we will prove in the next section, is ultimately connected to the definition of the *type* function.

4.3.1 Type Preservation by Substitution

One property of the type system is that if a clause, query, or term M is well-typed with some type τ, given a context Γ and a set of type definitions Δ, then every instance of Δ also types M with some type $\tau\prime$ that is an instance of τ. We prove this property in Lemma 7. First, we need the following auxiliary lemma.

Lemma 6 - Equivalent Instances: Let τ and $\tau\prime$ be types, such that $\tau \equiv_\Delta \tau\prime$, Δ a set of type definitions, and S be a substitution for type variables. Suppose $S(\tau) = \tau\prime\prime$. Then $S(\tau\prime) = \tau\prime\prime\prime$, and $\tau\prime\prime\prime \equiv_\Delta \tau\prime\prime$.

Proof: Let us describe how τ and $\tau\prime$ can be such that $\tau \equiv_\Delta \tau\prime$. Either:

- $\tau = \tau\prime$: if this is the case, clearly $S(\tau\prime) \equiv_\Delta S(\tau)$, since $S(\tau) = S(\tau\prime)$;
- τ is a type symbol and $\tau\prime$ is a union type: then since $\tau \equiv_\Delta \tau\prime$, we know that $\Delta(\tau) = \tau\prime$ and therefore $S(\tau) \equiv_\Delta S(\tau\prime)$.
- τ and $\tau\prime$ are two different type symbols: then since $\tau \equiv_\Delta \tau\prime$, we know that $\Delta(\tau)$ and $\Delta(\tau\prime)$ are equal up to the renaming of type symbols. Therefore, $S(\Delta(\tau)) \equiv_\Delta S(\Delta(\tau\prime))$.

In any case, we get the result we wanted. □

We are now ready to prove the substitution lemma.

Lemma 7 - Substitution Lemma: Let Γ be a context, Δ a set of type definitions, P a program, M a clause, a query, or a term, and τ a type. If $\Gamma, \Delta \vdash_P M : \tau$, then, for any substitution S, we have a derivation for $\Gamma, \overline{S(\Delta)} \vdash_P M : S(\tau)$.

Proof: The proof follows from induction on the size of the derivation.

- Suppose we have $\Gamma \cup \{X : \sigma\}, \Delta \cup \{\sigma(\overrightarrow{\alpha}) = \widetilde{\tau}\} \vdash_P X : \sigma(\overrightarrow{\alpha})$. Then, for any substitution S, we would get $\overline{S(\Delta \cup \{\sigma(\overrightarrow{\alpha}) = \widetilde{\tau}\})} = \overline{S(\Delta)} \cup \overline{S(\{\sigma(\overrightarrow{\alpha}) = \widetilde{\tau}\})}$. By one application of rule VAR we would get $\Gamma \cup \{X : \sigma\}, \overline{S(\Delta) \cup \{S(\sigma(\overrightarrow{\alpha})) = S(\widetilde{\tau})\}} \vdash_P X : S(\sigma(\overrightarrow{\alpha}))$.

- Suppose we have $\Gamma, \Delta \vdash_P c : \tau$, where $\tau \preceq type(c)$. Since for any S, either $S(\tau) = \tau$, if τ is ground, and therefore $S(\tau) \preceq type(c)$, or $S(\tau)$ is an instance of τ, which implies $S(\tau) \preceq type(c)$. Then the derivation would still be true for $\Gamma, \overline{S(\Delta)} \vdash_P c : S(\tau)$, and trivially $S(\tau) \preceq \tau$.

- Suppose we have $\Gamma, \Delta \vdash_P f(t_1, \ldots, t_n) : \tau$, where $type(f) = \sigma$, $\tau\prime_1 \times \cdots \times \tau\prime_n \to \tau \preceq \sigma$, $\forall i . \tau_i \equiv_\Delta \tau\prime_i$, and $\Gamma, \Delta \vdash_P t_i : \tau_i$. By the induction hypothesis, we know that $\forall i . \Gamma, \overline{S(\Delta)} \vdash_P t_i : S(\tau_i)$. Since $\tau_i \equiv_\Delta \tau\prime_i$, then by Lemma 6, we know that for any S, $S(\tau_i) \equiv_\Delta S(\tau\prime_i)$. Since $S(\tau\prime_i) \preceq \tau\prime_i$, then $S(\tau\prime_1) \times \cdots \times S(\tau\prime_n) \to S(\tau) \preceq \sigma$. By one application of rule CPL we get $\Gamma, \overline{S(\Delta)} \vdash_P f(t_1, \ldots, t_n) : S(\tau)$.

- Suppose we have $\Gamma, \Delta \vdash_P t_1 = t_2 : bool$, where $\Gamma, \Delta \vdash_P t_1 : \tau_1$ and $\Gamma, \Delta \vdash_P t_2 : \tau_2$, where $\tau_1 \equiv_\Delta \tau_2$. By the induction hypothesis, we have that $\Gamma, \overline{S(\Delta)} \vdash_P t_1 : S(\tau_1)$ and $\Gamma, \overline{S(\Delta)} \vdash_P t_2 : S(\tau_2)$. Since $\tau_1 \equiv_\Delta \tau_2$, then by Lemma 6, we know that for any S, $S(\tau_1) \equiv_\Delta S(\tau_2)$. Therefore, in one application of rule UNF, we get $\Gamma, \overline{S(\Delta)} \vdash_P t_1 = t_2 : bool$.

- Suppose we have $\Gamma \cup \{\overrightarrow{X} : \overrightarrow{\sigma}\}, \Delta \cup \{\overrightarrow{\sigma} = \overrightarrow{\tau}\} \vdash_P p(\overrightarrow{X}) : bool$, where $\overrightarrow{\sigma\prime} \to bool \sqsubseteq_\Delta \overrightarrow{\sigma} \to bool$. Since $\overrightarrow{\sigma} \sqsubseteq \overrightarrow{\sigma\prime}$, then for any S, we know that $S(\overrightarrow{\sigma}) \sqsubseteq_\Delta \overrightarrow{\sigma\prime}$, by the definition of $\sqsubseteq$. Therefore we can have the same derivation $\Gamma\prime \cup \{\overrightarrow{Y} : \overrightarrow{\sigma\prime}\}, \Delta\prime \cup \{\overrightarrow{\sigma\prime} = \overrightarrow{\tau\prime}\} \vdash_P p(\overrightarrow{Y}) : -body. : bool$, and in one application of rule CLL, we get $\Gamma \cup \{\overrightarrow{X} : \overrightarrow{\sigma}\}, \overline{S(\Delta \cup \{\overrightarrow{\sigma} = \overrightarrow{\tau}\})} \vdash_P p(\overrightarrow{X}) : bool$.

- Suppose we have $\Gamma, \Delta \vdash_P g_1, \ldots, g_n : bool$, where $\Gamma, \Delta \vdash_P g_i : bool$, for all i. By the induction hypothesis, we have $\Gamma, S(\Delta) \vdash_P g_1, \ldots, g_n : bool$, for all i, therefore we can apply rule CON, we get $\Gamma, \overline{S(\Delta)} \vdash_P g_1, \ldots, g_n : bool$.

- Suppose we have $\Gamma, \Delta \vdash_P (p(\overrightarrow{X}) : -b1; \ldots; b_m.) : bool$, where $\Gamma_i, \Delta_i \vdash_P b_i : bool$, for $i = 1, \ldots, m$, and $\oplus\Big((\Gamma_1, \ldots, \Gamma_m), (\Delta_1, \ldots, \Delta_m)\Big) = (\Gamma, \Delta)$. By the induction hypothesis, we know for any S, $\Gamma_i, \overline{S(\Delta_i)} \vdash_P b_i : bool$, for $i = 1, \ldots, m$.

  Suppose that for all $X \in def(\Gamma_i)$, $X \notin def(\Gamma_j)$. Then $\Delta_i(\Gamma_i(X)) = \Delta(\Gamma(X))$. In that case, we have $S(\Delta_i(\Gamma_i(X))) = S(\Delta(\Gamma(X)))$. Now suppose there is some variable Y such that $Y \in def(\Gamma_i) \wedge \cdots \wedge Y \in def(\Gamma_j)$, for some $1 \leq i, j \leq m$. Then $\Delta(\Gamma(X))$ is the TDC of $\Delta_i(\Gamma_i(X)) + \ldots + \Delta_j(\Gamma_j(X))$, therefore $\overline{S(\Delta(\Gamma(X)))}$ is the TDC of $S(\Delta_i(\Gamma_i(X))) + \ldots + S(\Delta_j(\Gamma_j(X)))$. In whichever case, we prove that $\oplus\Big((\Gamma_1, \ldots, \Gamma_m), (S(\Delta_1), \ldots, S(\Delta_m))\Big) = (\Gamma, \overline{S(\Delta)})$. By one application of the CLS rule, we have that $\Gamma, \overline{S(\Delta)} \vdash_P (p(\overrightarrow{X}) : -b1; \ldots; b_m.) : bool$.

- Suppose we have that $\Gamma, \Delta \vdash_P (p(\overrightarrow{X}) : -b1; \ldots; b_m; b_{m+1}, p(\overrightarrow{Y_{11}}), \ldots, p(\overrightarrow{Y_{1k_1}}); \ldots; b_{m+k}, p(\overrightarrow{Y_{n1}}), \ldots, p(\overrightarrow{Y_{nk_n}})) : bool$, where $\Gamma, \Delta \vdash_P (p(\overrightarrow{X}) : -b1; \ldots; b_m.) : bool$. By the induction hypothesis, we have $\Gamma, \overline{S(\Delta)} \vdash_P (p(\overrightarrow{X}) : -b1; \ldots; b_m.) : bool$. Since we do not require any extra condition, with one application of RCLS rule, we get $\Gamma, \overline{S(\Delta)} \vdash_P (p(\overrightarrow{X}) : -b1; \ldots; b_m; b_{m+1}, p(\overrightarrow{Y_{11}}), \ldots, p(\overrightarrow{Y_{1k_1}}); \ldots; b_{m+k}, p(\overrightarrow{Y_{n1}}), \ldots, p(\overrightarrow{Y_{nk_n}})) : bool$.

□

4.3.2 Soundness of the Type System

The type system allows for type verification, but we need to know that, in fact, if we have some derivation in the type system then that derivation is semantically correct. The soundness of the type system is shown by proving that every derivation of the form $\Gamma, \Delta \vdash_P M : \tau$ implies that semantically $\Gamma, \Delta \models_I M : \tau$. To prove this, we need a few auxiliary definitions.

Definition 30 - Consistency of I and *type*: Given a function *type* and an interpretation I, we say I is consistent with *type* if the following holds:

- for any constant c: $type(c) = \tau$, and for any Δ, $I(c) \in \mathbf{T}[\![\tau]\!]_\Delta$
- for any function symbol f: $type(f) = \tau_1 \times \ldots \times \tau_n \to \tau$, and for any Δ, $I(f) \in \mathbf{P}[\![\tau_1 \times \ldots \times \tau_n \to \tau]\!]_\Delta$.

If an interpretation I is not consistent with our function *type* we can assume that that interpretation does not follow the lines of the programmer's intention. Similarly to proper interpretations, where it would not make sense

to have interpretations that, for a type error-free body of a clause, would still give that clause the value *wrong* because of the interpretation of the head, it does not make sense to have an interpretation that interprets constants and function symbols differently than the programmer does.

Example 38: Let us consider again the interpretation I described in Example 27. $I(1) = 1$, and $I(a) = \text{a}$, where $domain(1) = \mathbf{Int}$, and $domain(\text{a}) = \mathbf{Atom}$. Also, $I(p) = f$, such that $f :: D \rightarrow \mathbf{Bool}$, where $D = \mathbf{Int} + \mathbf{Atom}$. Let our program be defined as follows:

```
p(X) :- X = a ; X = 1.
```

One *type* function that would be consistent with I is a type function such that $type(a) = atom$ and $type(1) = int$. One *type* function that would not be consistent with I is a type function such that $type(a) = type(1) = atom$.

We have an important property of constants used in polymorphic types, for proper interpretations.

Lemma 8 - Polymorphically Typed Constants: Let c be a constant and σ a type symbol defining a polymorphic type. For any proper interpretation, if $type(c) = \sigma$, we know that for any S, $I(c) \in \mathbf{T}[\![S(\sigma)]\!]_{\Delta}$.

Proof: Since $type(c) = \sigma$, then $\sigma = c + \widetilde{\tau}$. For any S, we know that $S(c) = c$, since c is a constant. Therefore $S(\sigma) = S(c) + S(\widetilde{\tau}) = c + S(\widetilde{\tau})$. So, for every S, $I(c) \in \mathbf{T}[\![S(\sigma)]\!]_{\Delta}$. □

What this lemma says is that a constant that has a polymorphic type has every instance of that type.

One important thing we need to know is that is a clause is well-typed with some type, then a call to the predicate being defined in that clause is well-typed with the same type. This is proven by the following lemma.

Lemma 9 - Model for a Query: For any proper interpretation I, if $\Gamma, \Delta \models_I p(X_1, \ldots, X_n) : -body. : bool$, then $\Gamma, \Delta \models_I p(X_1, \ldots, X_n) : bool$.

Proof: We know that $\exists[(\Gamma_1, \Delta_1), \ldots, (\Gamma_n, \Delta_n)].\forall\overrightarrow{\Sigma} = [\Sigma_1, \ldots, \Sigma_m]$. $\Big[[\![\Gamma_1]\!]_{I,\Sigma_1,\Delta_1} \wedge \ldots \wedge [\![\Gamma_m]\!]_{I,\Sigma_m,\Delta_m} \implies [\![p(X_1, \ldots, X_n) : -body.)]\!]_{I,\overrightarrow{\Sigma}} \in \mathbf{T}[\![bool]\!]_{\Delta}\Big]$, where $\oplus((\Gamma_1, \ldots, \Gamma_m), (\Delta_1, \ldots, \Delta_m)) = (\Gamma, \Delta)$.

Therefore $\forall\overrightarrow{\Sigma} = [\Sigma_1, \ldots, \Sigma_m].\Big[[\![\Gamma_1]\!]_{I,\Sigma_1,\Delta_1} \wedge \ldots \wedge [\![\Gamma_m]\!]_{I,\Sigma_m,\Delta_m} \implies [\![p(X_1, \ldots, X_n)]\!]_{I,\Sigma_1} \in \mathbf{T}[\![bool]\!]_{\Delta_1} \wedge \ldots \wedge [\![p(X_1, \ldots, X_n)]\!]_{I,\Sigma_m} \in \mathbf{T}[\![bool]\!]_{\Delta_n}\Big]$.

This means that $I(p) \in \mathbf{P}[\![\Gamma_i(X_1) \times \ldots \times \Gamma_i(X_n) \to \mathit{bool}]\!]_{\Delta_1}$, for all i. Let $D_1^i, \ldots, D_n^i$ be the domains associated with $\Gamma_i(X_1) \times \cdots \times \Gamma_i(X_n)$, for each i. Since I is a proper interpretation, $\mathit{domain}(I(p)) \supseteq (D_1, \ldots, D_n)$, where D_i are the tuple distributive closure of $D_1^i + \ldots + D_n^i$. We can suppose $f(v_1, \ldots, v_n)$ and $f(v\prime_1, \ldots, v\prime_n)$ are in a D_i, then D_i must include $\{f(v_1^{i_1}, \ldots, v_n^{i_n}) \mid 1 \leq i_1, \ldots, i_n \leq 2\}$, because I is a proper interpretation.

But since $\oplus((\Gamma_1, \ldots, \Gamma_m), (\Delta_1, \ldots, \Delta_m)) = (\Gamma, \Delta)$, then $\Gamma(X_i)$ is the tuple distributive closure of $\Gamma_1(X_i) + \ldots + \Gamma_n(X_i)$. This means that $I(p) \in \mathbf{P}[\![\Gamma(X_1) \times \ldots \times \Gamma(X_n) \to \mathit{bool}]\!]_{\Delta}$.

Therefore, we conclude that $\forall\Sigma.[\![\Gamma]\!]_{I,\Sigma,\Delta} \implies [\![p(X_1, \ldots, X_n)]\!]_{I,\Sigma} \in \mathbf{T}[\![\mathit{bool}]\!]_{I,\Sigma,\Delta}$, so $\Gamma, \Delta \models_I p(X_1, \ldots, X_n) : \mathit{bool}$. □

Finally, our main result shows that the type system is semantically sound, meaning that if a program has a type in our type system, then the program and its type are related by the semantic typing relation defined in Definition 25.

Theorem 9 - Semantic Soundness: Let P be a program, Γ a context, Δ a set of type definitions, and I a proper interpretation consistent with *type*, then $\Gamma, \Delta \vdash_P M : \tau \implies \Gamma, \Delta \models_I M : \tau$.

Proof: The proof of this theorem follows by structural induction on M.

- VAR: We know that $(X : \sigma) \in \Gamma$ and $(\sigma = \tau) \in \Delta$ and we want to prove that for any proper interpretation I consistent with *type*, then $\Gamma, \Delta \models_I X : \tau$, which corresponds to proving $\forall\Sigma.\Big[[\![\Gamma]\!]_{I,\Sigma,\Delta} \implies [\![X]\!]_{I,\Sigma} \in \mathbf{T}[\![\tau]\!]_{\Delta}\Big]$, since $\mathit{or\_degree}(X) = 1$. For any I, suppose for some Σ $[\![\Gamma]\!]_{I,\Sigma,\Delta}$ is false, then the implication is true and we get the result we wanted. Suppose for some other Σ, $[\![\Gamma]\!]_{I,\Sigma,\Delta}$ is true. Then it follows by Definitions 22 and 23 that the right side of the implication is true, which means the whole implication is also true.

- CST: We know that $\mathit{type}(c) = \tau\prime$, and we want to prove that, for any proper interpretation I consistent with *type*, $\Gamma, \Delta \models_I c : \tau$, which corresponds to proving $\forall\Sigma.\Big[[\![\Gamma]\!]_{I,\Sigma,\Delta} \implies [\![c]\!]_{I,\Sigma} \in \mathbf{T}[\![\tau]\!]_{\Delta}\Big]$, since $\mathit{or\_degree}(c) = 1$. From Definition 30, we know that if $\mathit{type}(c) = \tau\prime$, then for any Δ, we have that $I(c) \in \mathbf{T}[\![\tau\prime]\!]_{\Delta}$. We know that $\tau \preceq \tau\prime$, so either $\tau = \tau\prime$, or there is a substitution S, such that $S(\tau\prime) = \tau$. If the types are equal, then $I(c) \in \mathbf{T}[\![\tau]\!]_{\Delta}$, trivially. If $S(\tau\prime) = \tau$, then by Lemma 8, we know that $I(c) \in \mathbf{GT}[\![\tau]\!]_{\Delta} = \mathbf{T}[\![\tau]\!]_{\Delta}$. So, since for any Σ, $[\![c]\!]_{I,\Sigma} = I(c)$, we know that the right-hand side of the implication is true. Therefore the implication is true for any Σ.

- CPL: We want to prove that, for any proper interpretation I consistent with *type*, $\Gamma, \Delta \models_I f(t_1, \ldots, t_n) : \tau$, which corresponds to proving $\forall\Sigma.\Big[[\![\Gamma]\!]_{I,\Sigma,\Delta} \implies [\![f(t_1, \ldots, t_n)]\!]_{I,\Sigma} \in \mathbf{T}[\![\tau]\!]_\Delta\Big]$, since we know that $or\_degree(f(t_1, \ldots, t_n)) = 1$. By the induction hypothesis, we know that $\Gamma, \Delta \models_I t_i : \tau_i$, for $i = 1, \ldots, n$. We also know that $type(f) = \sigma$. From Definition 30, we know that for any Δ, $I(f) \in \mathbf{P}[\![\sigma]\!]_\Delta$. We know that $\tau\prime_1 \times \cdots \times \tau\prime_n \to \tau \preceq \sigma$, so either $\tau\prime_1 \times \cdots \times \tau\prime_n \to \tau = \sigma$, or there is a substitution S, such that $S(\sigma) = \tau\prime_1 \times \ldots \times \tau\prime_n \to \tau \preceq \sigma$. If both types are equal, then $I(f) \in \mathbf{P}[\![\tau\prime_1 \times \ldots \times \tau\prime_n \to \tau]\!]_\Delta$ trivially. If for some S, $S(\sigma) = \tau\prime_1 \times \ldots \times \tau\prime_n \to \tau \preceq \sigma$, then, since $\mathbf{P}[\![\sigma]\!]_\Delta = \bigcap_{\forall S} \mathbf{GP}[\![S(\sigma)]\!]_\Delta$, we know that $I(f) \in \mathbf{GP}[\![\tau\prime_1 \times \ldots \times \tau\prime_n \to \tau]\!]_\Delta = \mathbf{P}[\![\tau\prime_1 \times \ldots \times \tau\prime_n \to \tau]\!]_\Delta$. Since for all $i = 1, \ldots, n$ $\tau_i \equiv \tau_i\prime$, then $\mathbf{T}[\![\tau_1 \times \ldots \times \tau_n]\!]_\Delta = \mathbf{T}[\![\tau\prime_1 \times \ldots \times \tau\prime_n]\!]_\Delta$, which implies that $I(f) \in \mathbf{P}[\![\tau_1 \times \ldots \times \tau_n \to \tau]\!]_{\Delta,S}$. So, for any Σ, $\Gamma, \Delta \models_I f(t_1, \ldots, t_n) : \tau$.

- UNF: We want to prove that for any proper interpretation I consistent with *type*, $\Gamma, \Delta \models_I t_1 = t_2 : bool$, which corresponds to proving $\forall\Sigma.\forall S.\Big[[\![\Gamma]\!]_{I,\Sigma,\Delta} \implies [\![t_1 = t_2]\!]_{I,\Sigma} \in \mathbf{T}[\![bool]\!]_\Delta\Big]$, since $or\_degree(t_1 = t_2) = 1$. By the induction hypothesis, we know that for any I consistent with *type*, $\Gamma, \Delta \models_I t_1 : \tau_1$ and $\Gamma, \Delta \models_I t_2 : \tau_2$. Therefore we know that for any Σ, $[\![t_1]\!]_{I,\Sigma} \in \mathbf{T}[\![\tau_1]\!]_\Delta$ and $[\![t_2]\!]_{I,\Sigma} \in \mathbf{T}[\![\tau_2]\!]_\Delta$. Since $\tau_1 \equiv \tau_2$, $\mathbf{T}[\![\tau_1]\!]_\Delta = \mathbf{T}[\![\tau_2]\!]_\Delta$. Therefore, for any Σ, $[\![t_1 = t_2]\!]_{I,\Sigma} = true$ or $[\![t_1 = t_2]\!]_{I,\Sigma} = false$, so $([\![t_1 = t_2]\!]_{I,\Sigma} \in \mathbf{T}[\![bool]\!]_\Delta) = true$.

- CLL: Let $\Gamma\prime = \Gamma_1 \cup \{X_1 : \sigma_1, \ldots, X_n : \sigma_n\}$, $\Gamma\prime\prime = \Gamma_2 \cup \{Y_1 : \sigma_1\prime, \ldots, Y_n : \sigma_n\prime\}$, $\Delta\prime = \Delta_1 \cup \{\sigma_1 = \widetilde{\tau_1}, \ldots, \sigma_n = \widetilde{\tau_n}\}$, and $\Delta\prime\prime = \Delta_2 \cup \{\sigma_1\prime = \widetilde{\tau_1}\prime, \ldots \sigma_n\prime = \widetilde{\tau_n}\prime\}$, and we want to prove that for any proper interpretation I consistent with *type*, $\Gamma\prime, \Delta\prime \models_I p(X_1, \ldots, X_n) : bool$, which corresponds to proving $\forall\Sigma.\Big[[\![\Gamma\prime]\!]_{I,\Sigma,\Delta\prime} \implies [\![p(X_1, \ldots, X_n)]\!]_{I,\Sigma} \in \mathbf{T}[\![bool]\!]_{\Delta\prime}\Big]$, since $or\_degree(p(X_1, \ldots, X_n)) = 1$. By the induction hypothesis, we know that $\Gamma\prime\prime, \Delta\prime\prime \models_I p(Y_1, \ldots, Y_n) : -body. : bool$. By Lemma 9, we also know that $\Gamma\prime\prime, \Delta\prime\prime \models_I p(Y_1, \ldots, Y_n)$, therefore $[\![p(Y_1, \ldots, Y_n)]\!]_{I,\Sigma} \in \mathbf{T}[\![bool]\!]_{\Delta\prime\prime}$, for any Σ such that $[\![\Gamma\prime\prime]\!]_{I,\Sigma,\Delta\prime\prime}$, which means that $\forall(v_1, \ldots, v_n).\Sigma(Y_1, \ldots, Y_n) = (v_1, \ldots, v_n) \in \mathbf{T}[\![\sigma\prime_1 \times \ldots \times \sigma\prime_n]\!]_{\Delta\prime\prime}$, we have that $I(p)(v_1, \ldots, v_n) \in \mathbf{T}[\![bool]\!]_{\Delta\prime\prime}$. Note that we can replace Γ_2 by Γ_1 in $\Gamma\prime\prime$ and the condition would still hold, since the variables in Γ_2 are irrelevant for $p(Y_1, \ldots, Y_n)$. Since $\sigma\prime_1 \times \cdots \times \sigma\prime_n \to bool \sqsubseteq_{\Delta\prime\prime \cup \{\vec{\sigma} = \vec{\tau}\}} \sigma_1 \times \cdots \times \sigma_n \to bool$, we know that $\sigma_i \sqsubseteq_{\Delta\prime\prime \cup \{\vec{\sigma} = \vec{\tau}\}} \sigma\prime_i$, then all $(v_1, \ldots, v_n) \in \mathbf{T}[\![\sigma_1 \times \ldots \times \sigma_n]\!]_\Delta$ are also such that $(v_1, \ldots, v_n) \in \mathbf{T}[\![\sigma_1 \times \ldots \times \sigma_n]\!]_{\Delta\prime\prime}$. We can therefore see that $\forall\Sigma.([\![\Gamma_1 \cup \{Y_1 : \sigma_1, \ldots, Y_n : \sigma_n\}]\!]_{I,\Sigma,\Delta\prime} \implies [\![p(Y_1, \ldots, Y_n)]\!]_{I,\Sigma} \in \mathbf{T}[\![bool]\!]_{\Delta\prime}$. Now if we replace the variables Y_i for X_i both in the atom $p(Y_1, \ldots, Y_n)$ and in $\Gamma_1 \cup \{Y_1 : \sigma_1, \ldots, Y_n : \sigma_n\}$, the condition still holds

trivially. Therefore we have $\forall\Sigma.[\![\Gamma\prime]\!]_{I,\Sigma,\Delta\prime} \implies [\![p(X_1,\ldots,X_n)]\!]_{I,\Sigma} \in \mathbf{T}[\![bool]\!]_{\Delta\prime}$, which means, $\Gamma\prime,\Delta \models_I p(X_1,\ldots,X_n) : bool$.

- CON: We want to prove that for any proper interpretation I consistent with type, $\Gamma,\Delta \models_I b_1,\ldots,b_n : bool$, which means $\forall\Sigma.[\![\Gamma]\!]_{I,\Sigma,\Delta} \implies [\![b_1,\ldots,b_n]\!]_{I,\Sigma} \in \mathbf{T}[\![bool]\!]_\Delta$, since $or\_degree(b_1,\ldots,b_n) = 1$. By the induction hypothesis, we know that for any I consistent with *type*, $\Gamma,\Delta \models_I b_i : bool$, for all $i = 1,\ldots,n$. Therefore for any Σ and S such that $[\![\Gamma]\!]_{I,\Sigma,\Delta}$, we know that $[\![b_i]\!]_{I,\Sigma} \in \mathbf{T}[\![bool]\!]_\Delta$, so $[\![b_1,\ldots,b_n]\!]_{I,\Sigma} = ([\![b_1]\!]_{I,\Sigma} \wedge \ldots \wedge [\![b_n]\!]_{I,\Sigma}) \in \mathbf{T}[\![bool]\!]_\Delta$. Therefore $\Gamma,\Delta \models_I b_1,\ldots,b_n : bool$.

- CLS: We want to prove that for any proper interpretation I consistent with type, $\Gamma,\Delta \models_I p(X_1,\ldots,X_n) : -b_1;\ldots;b_m : bool$, which means $\exists[(\Gamma_1,\Delta_1),\ldots,(\Gamma_m,\Delta_m)].\forall\Sigma\prime = [\Sigma_1,\ldots,\Sigma_m].\forall S.[\![\Gamma_1]\!]_{I,\Sigma_1,\Delta_1} \wedge \ldots \wedge [\![\Gamma_m]\!]_{I,\Sigma_m,\Delta_m} \implies [\![p(X_1,\ldots,X_n) : -b_1;\ldots;b_m]\!]_{I,\Sigma\prime} \in \mathbf{T}[\![bool]\!]_\Delta$, since $or\_degree(b_1,\ldots,b_n) = m$, and $\oplus((\Gamma_1,\ldots,\Gamma_m),(\Delta_1,\ldots,\Delta_m)) = (\Gamma,\Delta)$. By the induction hypothesis, we know that for any proper interpretation I consistent with *type*, $\Gamma_i,\Delta_i \models_I b_i : bool$, for all $i = 1,\ldots,n$. Which means that $\forall\overrightarrow{\Sigma} = [\Sigma_1,\ldots,\Sigma_m].[\![\Gamma_1]\!]_{I,\Sigma_1,\Delta_1} \wedge \ldots \wedge [\![\Gamma_m]\!]_{I,\Sigma_m,\Delta_m} \implies [\![b_1;\ldots;b_m]\!]_{I,\overrightarrow{\Sigma}} \in \mathbf{T}[\![bool]\!]_\Delta$. Since I is a proper interpretation, $\forall\overrightarrow{\Sigma} = [\Sigma_1,\ldots,\Sigma_m].[\![b_1;\ldots;b_m]\!]_{I,\overrightarrow{\Sigma}} \in \mathbf{T}[\![bool]\!]_\Delta \implies \forall i.[\![p(X_1,\ldots,X_n)]\!]_{I,\Sigma_i} \in \mathbf{T}[\![bool]\!]_\Delta$. Therefore, $\forall\overrightarrow{\Sigma}.[\![\Gamma_1]\!]_{I,\Sigma_1} \wedge \ldots \wedge [\![\Gamma_m]\!]_{I,\Sigma_m} \implies [\![p(X_1,\ldots,X_n) : -b_1;\ldots;b_m]\!]_{I,\overrightarrow{\Sigma}} \in \mathbf{T}[\![bool]\!]_\Delta$.

- RCLS: Let $\Gamma\prime = \Gamma \cup \{\overrightarrow{X} : \overrightarrow{\tau}, \overrightarrow{Y_{11}} : \overrightarrow{\tau},\ldots,\overrightarrow{Y_{nk_n}} : \overrightarrow{\tau}\}$, and $body = b_1;\ldots;b_m;b_{m+1},p(\overrightarrow{Y}_{11}),\ldots,p(\overrightarrow{Y}_{1k_1});b_{m+n},p(\overrightarrow{Y}_{n1}),\ldots,p(\overrightarrow{Y}_{nk_n})$. We want to prove that for any proper interpretation I consistent with type $\Gamma\prime,\Delta \models_I (p(\overrightarrow{X}) : -body.) : bool$, which can also be written as $\exists[(\Gamma_1,\Delta_1),\ldots,(\Gamma_{m+n},\Delta_{m+n})].\forall\Sigma\prime = [\Sigma_1,\ldots,\Sigma_{m+n}].[\![\Gamma_1]\!]_{I,\Sigma_1,\Delta_1,S} \wedge \ldots \wedge [\![\Gamma_{m+n}]\!]_{I,\Sigma_{m+n},\Delta_{m+n}} \implies [\![p(\overrightarrow{X}) : -body.]\!]_{I,\overrightarrow{\Sigma}} \in \mathbf{T}[\![bool]\!]_\Delta$. By the induction hypothesis, $\Gamma\prime,\Delta \models_I p(\overrightarrow{X}) : -b_1;\ldots;b_{m+n} : bool$. So for any $\overrightarrow{\Sigma} = [\Sigma_1,\ldots,\Sigma_{m+n}]$, such that $[\![\Gamma_1]\!]_{I,\Sigma_1,\Delta_1} \wedge \ldots \wedge [\![\Gamma_{m+n}]\!]_{I,\Sigma_{m+n},\Delta_{m+n}}$ we know that $[\![p(\overrightarrow{X})]\!]_{I,\Sigma_1} \in \mathbf{T}[\![bool]\!]_\Delta \wedge \ldots \wedge [\![p(\overrightarrow{X})]\!]_{I,\Sigma_{m+n}} \in \mathbf{T}[\![bool]\!]_\Delta$. Let the values for $\overrightarrow{X}$ in any such $\overrightarrow{\Sigma}$ be called $\overrightarrow{V_1},\ldots,\overrightarrow{V_l}$. Then, since the types for $\overrightarrow{Y_i}$, for any $i = 11,\ldots,nk_n$ are the same as the types for $\overrightarrow{X}$, the values possible for each $\overrightarrow{Y_i}$ will be $\overrightarrow{v_1},\ldots,\overrightarrow{v_l}$. But we know that for those values, $I(p)(\overrightarrow{v_j}) \in \mathbf{T}[\![bool]\!]_\Delta$, therefore for any $\overrightarrow{\Sigma}$ and any S, such that $[\![\Gamma_1]\!]_{I,\Sigma_1,\Delta_1} \wedge \ldots \wedge [\![\Gamma_{m+n}]\!]_{I,\Sigma_{m+n},\Delta_{m+n}}$, we know that $\forall i = 11,\ldots,nk_n.[\![p(\overrightarrow{Y_i})]\!]_{I,\Sigma_1} \in \mathbf{T}[\![bool]\!]_\Delta \wedge \ldots \wedge [\![p(\overrightarrow{Y_i})]\!]_{I,\Sigma_{m+n}} \in \mathbf{T}[\![bool]\!]_\Delta$. So we prove what we wanted, that $\Gamma\prime,\Delta \models_I (p(\overrightarrow{X}) : -body.) : bool$.

□

Note that the value *wrong* has no type, thus, as a corollary of the soundness theorem, we have that if a predicate is statically well-typed, then for any proper interpretation I consistent with *type*, following Milner's motto "well-typed programs can not go wrong" [36], the predicate semantics is not *wrong*.

4.4 Arithmetic

Some of the main built-ins in any programming language are arithmetic and relational operators. In Prolog, the main logic programming language, the common arithmetic operations have built-in operators. The binary arithmetic operations (binOp) $+$, $-$, $*$, $/$, $//$, $\hat{}$, and max, and the unary arithmetic operations (unOp) $-$, sin, cos, abs, and $sqrt$ are specially interpreted. In fact, the term $1 + 3$ has two interpretations: as the tree with root $+$ and children 1 and 3, and as the sum of 1 and 3. The latter is only used when using built-in arithmetic predicates. This is called *overloading* of function symbols. Overloaded function symbols have multiple types, one for each of the possible interpretations. For the rest of this section, we will only describe the special interpretation of these operators as arithmetic operators, to simplify presentation.

4.4.1 Semantics

Let $D = \mathbf{Int} + \mathbf{Float}$. The semantics for terms that start with these operators, for any I is:

$$
\begin{aligned}
[\![binOp(t_1, t_2)]\!]_{I,\Sigma} = & \textbf{if } (domain([\![t_1]\!]_{I,\Sigma}, domain([\![t_2]\!]_{I,\Sigma}) \subseteq (D, D)) \\
& \quad \textbf{then } binOp([\![t_1]\!]_{I,\Sigma}, [\![t_2]\!]_{I,\Sigma}) \\
& \quad \textbf{else } wrong \\
[\![unOp(t)]\!]_{I,\Sigma} = & \textbf{if } (domain([\![t]\!]_{I,\Sigma} \subseteq D) \\
& \quad \textbf{then } unOp([\![t]\!]_{I,\Sigma}) \\
& \quad \textbf{else } wrong
\end{aligned}
$$

Note that we have two rules since some operators are unary and some are binary.

These function symbols can only be interpreted as arithmetic operators if used inside an arithmetic predicate. The arithmetic predicates for the main relational operators (relOp) are : $=:=$, $>$, $<$, $>=$, $=/=$, and $=<$, and for the arithmetic equality it is is. The interpretation for these predicates, in any program, for any interpretation I, is as follows:

$$
\begin{aligned}
[\![is(t_1, t_2)]\!]_{I,\Sigma} = & \textbf{if } (domain([\![t_1]\!]_{I,\Sigma}, domain([\![t_2]\!]_{I,\Sigma}) \subseteq (D, D)) \\
& \quad \textbf{then if } [\![t_1]\!]_{I,\Sigma} = [\![t_2]\!]_{I,\Sigma}
\end{aligned}
$$

$$
\begin{array}{lll}
 & & \textbf{then } true \\
 & & \textbf{else } false \\
 & \textbf{else } wrong & \\
[\![relOp(t_1,t_2)]\!]_{I,\Sigma} = \textbf{if } & (domain([\![t_1]\!]_{I,\Sigma}, domain([\![t_2]\!]_{I,\Sigma}) \subseteq (D,D)) & \\
 & \textbf{then if } relOp([\![t_1]\!]_{I,\Sigma}, [\![t_2]\!]_{I,\Sigma}) & \\
 & & \textbf{then } true \\
 & & \textbf{else } false \\
 & \textbf{else } wrong &
\end{array}
$$

Let us show one example that uses some arithmetic built-ins.

Example 39: Let *len* be a predicate that calculates the length of a list, defined as follows:

```
len([],0).
len([X|Xs],N) :- len(Xs,N1), N is N1 + 1.
```

The normalized predicate definition is:

```
len(L,S) :- L = [], S = 0;
       L = [X | Xs], S = N, N is M, M = N1 + 1, len(Xs,N1).
```

Let I be the interpretation for which $I(\texttt{0}) = 0$ and $domain(0) = \mathbf{Int}$, $I(\texttt{1}) = 1$ and $domain(1) = \mathbf{Int}$, $I(\texttt{[]}) = [\ ]$, and $domain([\ ]) = \mathbf{D}$, where $\mathbf{D} = \mathbf{Nil} + List(\mathbf{Int}, \mathbf{D})$, and $I(\texttt{[|]}) = [\ |\ ]$ and $domain([\ |\ ]) = \mathbf{D}$. Also let $I(len)$ be such that $I(len) :: \mathbf{D} \times \mathbf{Int} \rightarrow \mathbf{Bool}$ Then, for any Σ_1 and Σ_2 that attribute integers to S, M, N, $N1$, and X, and lists of integers to L and Xs, the semantics of this predicate is not *wrong*.
$[\![len(L,S) : -L = [\ ], S = 0; L = [X|Xs], S = N, N\ is\ M, M = N1 + 1, len(Xs, N1).]\!]_{I,[\Sigma_1,\Sigma_2]}$
=
$[\![L = [\ ], S = 0; L = [X|Xs], S = N, N\ is\ M, M = N1{+}1, len(Xs, N1)]\!]_{I,[\Sigma_1,\Sigma_2]}$
$\Longrightarrow [\![len(L,S)]\!]_{I,\Sigma_1} \wedge [\![len(L,S)]\!]_{I,\Sigma_2}$
=
$[\![L = [\ ], S = 0]\!]_{I,\Sigma_1} \vee [\![L = [X|Xs], S = N, N\ is\ M, M = N1 + 1, len(Xs, N1)]\!]_{I,\Sigma_2} \Longrightarrow [\![len(L,S)]\!]_{I,\Sigma_1} \wedge [\![len(L,S)]\!]_{I,\Sigma_2}$
=
$([\![L = [\ ]]\!]_{I,\Sigma_1} \wedge [\![S = 0]\!]_{I,\Sigma_1}) \vee ([\![L = [X|Xs]]\!]_{I,\Sigma_2} \wedge [\![S = N]\!]_{I,\Sigma_2} \wedge [\![N\ is\ M]\!]_{I,\Sigma_2} \wedge [\![M = N1{+}1]\!]_{I,\Sigma_2} \wedge [\![len(Xs, N1)]\!]_{I,\Sigma_2}) \Longrightarrow [\![len(L,S)]\!]_{I,\Sigma_1} \wedge [\![len(L,S)]\!]_{I,\Sigma_2}$

Now note that in the expression $S = N$, both S and N have the same type (integers), so the result is boolean. In the expression $N\ is\ M$, M and N are integers, so the result is also boolean. Same for every expression, including $M = N1 + 1$, where M, $N1$, and 1 are all integers. Therefore, the result is either *true* or $false$, but never *wrong*.

As we can see from the example above, the introduction of the arithmetic built-ins does not really complicate the semantics. In fact, the only change is the overloading of function symbols. In this section, we assume that the arithmetic operators are only interpreted as such, but in reality, we could think of the semantics as having a choice point, where we would decide which interpretation for the function symbol we could use.

4.4.2 Type System Rules

Now that we have the semantics for these symbols, we can present the rules that extend the type system in order to type these special predicates and function symbols.

$$\textsc{BinOP1}\ \frac{\sigma_1 \equiv_\Delta int \quad \sigma_2 \equiv_\Delta int \quad \Gamma, \Delta \vdash t_1 : \sigma_1 \quad \Gamma, \Delta \vdash t_2 : \sigma_2}{\Gamma, \Delta \vdash_P binOp(t_1, t_2) : int}$$

$$\textsc{BinOP2}\ \frac{\sigma_1 \equiv_\Delta int \quad \sigma_2 \equiv_\Delta float \quad \Gamma, \Delta \vdash t_1 : \sigma_1 \quad \Gamma, \Delta \vdash t_2 : \sigma_2}{\Gamma, \Delta \vdash_P binOp(t_1, t_2) : float}$$

$$\textsc{BinOP3}\ \frac{\sigma_1 \equiv_\Delta float \quad \sigma_2 \equiv_\Delta int \quad \Gamma, \Delta \vdash t_1 : \sigma_1 \quad \Gamma, \Delta \vdash t_2 : \sigma_2}{\Gamma, \Delta \vdash_P binOp(t_1, t_2) : float}$$

$$\textsc{BinOP4}\ \frac{\sigma_1 \equiv_\Delta float \quad \sigma_2 \equiv_\Delta float \quad \Gamma, \Delta \vdash t_1 : \sigma_1 \quad \Gamma, \Delta \vdash t_2 : \sigma_2}{\Gamma, \Delta \vdash_P binOp(t_1, t_2) : float}$$

$$\textsc{UnOP}\ \frac{\Gamma, \Delta \vdash t : \sigma \quad \sigma \sqsubseteq_\Delta int + float}{\Gamma, \Delta \vdash unOp(t) : \sigma}$$

$$\textsc{RelOp/Is}\ \frac{\sigma_1 \sqsubseteq_\Delta int + float \quad \sigma_2 \sqsubseteq_\Delta int + float \quad \Gamma, \Delta \vdash t_1 : \sigma_1 \quad \Gamma, \Delta \vdash t_2 : \sigma_2}{\Gamma, \Delta \vdash relOp/is(t_1, t_2) : bool}$$

FIGURE 4.3: Arithmetic Extension to the Type System

Example 40: Let us continue with the previous example, and show the type rules being applied to some of the sub-expressions. We assume $type([\,]) = l(\alpha)$ and $type([\,|\,]) = \alpha \times l(\alpha) \rightarrow l(\alpha)$, where $l(\alpha) = [\,] + [\alpha \mid l(\alpha)]$. We also assume $\Gamma = \{X : \sigma, M : \sigma, N : \sigma, N1 : \sigma, Xs : \sigma_2, L : \sigma_2\}$, and $\Delta = \{\sigma =$

$int, \sigma_2 = [\,] + [int \mid \sigma_2]\}$ in what follows:

$$\frac{\sigma \cong int \quad \Gamma, \Delta \vdash_P M : \sigma \qquad \textbf{BinOp1}\ \dfrac{\sigma \cong int \quad \Gamma, \Delta \vdash_P N1 : \sigma \qquad \dfrac{type(1) = int \quad int \preceq int}{int \cong int} \quad \Gamma, \Delta \vdash_P 1 : int}{\Gamma, \Delta \vdash_P N1 + 1 : int}}{\Gamma, \Delta \vdash_P M = N1 + 1 : bool}$$

$$\textbf{RelOp/Is}\ \frac{\sigma \cong \sigma \quad \Gamma, \Delta \vdash_P M : \sigma \quad \Gamma, \Delta \vdash_P N : \sigma}{\Gamma, \Delta \vdash_P is(N, M) : bool}$$

$$\frac{\sigma_2 \cong l(int) \quad \Gamma, \Delta \vdash_P L : \sigma_2 \qquad \dfrac{\begin{array}{c} type([\mid]) = \alpha \times l(\alpha) \to l(\alpha) \\ int \times l(int) \to l(int) \preceq \alpha \times l(\alpha) \to l(\alpha) \\ \sigma \cong int \qquad \sigma_2 \cong l(int) \\ \Gamma, \Delta \vdash_P X : \sigma \qquad \Gamma, \Delta \vdash_P Xs : l(int) \end{array}}{\Gamma, \Delta \vdash_P [X \mid Xs] : \sigma_2}}{\Gamma, \Delta \vdash_P L = [X \mid Xs] : bool}$$

The examples above show how we can expand our semantics, and the type system to introduce some arithmetic built-ins. We argue that some extension are also possible to allow for more built-ins that are based on different data structures. In order to do so, we would follow a similar process as we did for arithmetic built-ins.

4.5 Discussion

Statically typing programs has several advantages such as ease of debugging, guarantees about properties of programs, and better program documentation. However, static typing cannot be a perfect process. Whenever we try to get more specific and cover the largest number of programs possible, we always lose some information.

For instance, the interpretation of a predicate is seen as a set of tuples that are accepted by the predicate. As soon as we have typed interpretations, some regularity is demanded by the types, and as a consequence some

information is lost. Take as an example a predicate with two arguments, defined as follows:

```
p(1,a).
p(a,1).
```

Any function f that is a typed interpretation for this predicate has, at least, the type $f :: D \times D \rightarrow Bool$, where $D = Int + Atom$. Any smaller domain would make one of the clauses *wrong*. Therefore, we now included tuples such as $(1, 1)$ as part of the domain of this predicate, which probably is not the programmer's intention, since dynamically, it would always lead to type errors.

More information is lost when dealing with tuple-distributive closures of types [8, 20], which we use in order to have decidability. Take the following predicate:

```
p(f(1,a)).
p(f(a,1)).
```

If we have a typed interpretation $f :: D \rightarrow Bool$, where $D = F(Int, Atom) + F(Atom, Int)$, then the actual type inferred would be $\sigma_1 = f(\sigma_2, \sigma_2)$, where $\sigma_2 = int + atom$, since we need type definitions to be tuple distributive. Therefore calls such as $p(f(1, 1))$ would not be considered type errors, even though they are probably intended to be.

This loss of information is inevitable and can be improved by limiting the expressiveness of programs or the language itself, having optional types, or dropping some important operations such as type inference.

This last option is not considered here because decidable type inference is mandatory in our approach. This is the subject of the next chapter.

Chapter 5

Type Inference

We now present a type inference algorithm that automatically infers types for programs and prove that the types inferred are sound, *i.e.*, they can be derived by the type system. The type inference algorithm allows for an automatic type verification without any input from the programmer. It was first published in [3], but some changes have been made since, which we present here, with the new corresponding proofs for soundness.

5.1 Type Inference

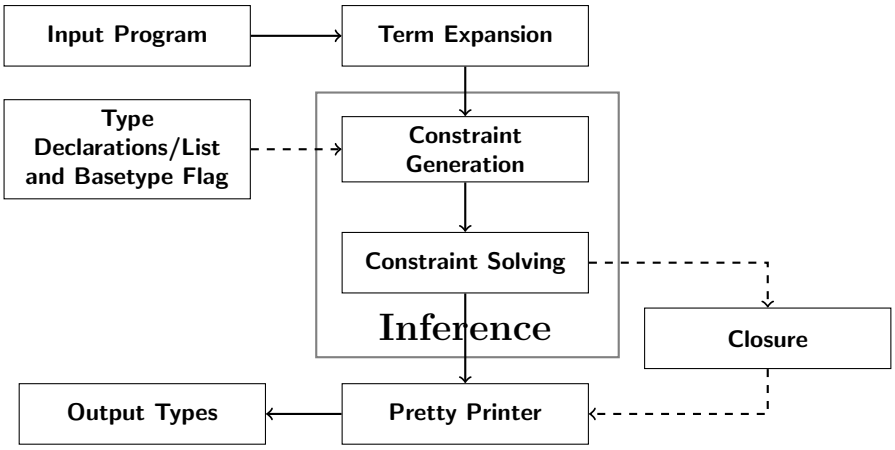


FIGURE 5.1: Type Inference Algorithm Flowchart

We have seen how to define the notion of *well-typed* program using a set of rules which assign types to programs. Here we will present a type inference algorithm which, given an untyped logic program, is able to infer types which make the program well-typed.

Our type inference algorithm is composed of several modules, as described in Figure 5.1. On a first step, when consulting programs, we apply term expansion to transform programs into the internal format that the rest of the algorithm expects. Secondly, we have the type inference phase itself, where constraint generation is performed, and a type constraint solver outputs the inferred types for a given program. To assure that at every step the

type definitions remain deterministic, TDC is performed during type inference, as described in Definition 21. After this, we either directly run a type pretty printer, or go through closure before printing the types. The closure operation will be described in the next chapter. For now, we will assume that after constraint solving, we immediately print our results.

Thus the type inference algorithm is composed of four main parts with some auxiliary steps:

- Term expansion
- Constraint generation
- Constraint solving
- Closure (optional)

5.1.1 Stratification

We assume that the input program of our algorithm is *stratified*. To understand the meaning of stratified programs, let us define the *dependency directed graph* of a program as the graph that has one node representing each predicate in the program and an edge from q to p for each call from a predicate p to a predicate q.

Definition 31 - Stratified Program: A *stratified program* P is such that the dependency directed graph of P has no cycles of size more than one.

This means that our type inference algorithm deals with predicates defined by direct recursion but not with mutual recursion. Note that stratified programs are widely used and characterize a large class of programs which is used in several database and knowledge base systems [76].

We discuss an extension of the algorithm to deal with mutually recursive predicates at the end of this chapter.

5.1.2 Constraints and Constraint Generation

The type inference algorithm starts by generating type constraints from a logic program which are solved by a constraint solver in a second stage of the algorithm. There are two different kinds of type constraints: equality constraints and subtyping constraints. An equality constraint is of the form $\tau_1 \doteq \tau_2$ and a subtyping constraint is of the form $\tau_1 \leq \tau_2$. Ultimately, we want to determine if a set of constraints C can be instantiated affirmatively using some substitution S, that substitutes types for type variables. For this we need to consider a notion of *constraint satisfaction* $S \models C$, in a first order theory with equality [77] and the extra axioms in definition 27 for subtyping.

Definition 32 - Constraint satisfaction: Let $=$ mean syntactic type equality, $\sqsubseteq$ the subtyping relation defined in definition 27, and Δ be a set of type definitions. $S \models_\Delta C$ is defined as follows:

1. $S \models_\Delta \tau_1 \doteq \tau_2$ if and only if $S(\tau_1) = S(\tau_2)$;
2. $S \models_\Delta \tau_1 \leq \tau_2$ if and only if $S(\tau_1) \sqsubseteq_\Delta S(\tau_2)$;
3. $S \models_\Delta C$ if and only if $S \models_\Delta c$ for each constraint $c \in C$.

The constraint generation step of the algorithm will output two sets of constraints, Eq (a set of equality constraints) and $Ineq$ (a set of subtyping constraints), that need to be solved during type inference.

Let us first present an auxiliary function to combine contexts. Contexts can be obtained from the disjunction, or conjunction, of other contexts. Previously, we have defined an auxiliary function $\oplus$ that results in the tuple distributive closure of summing the types for each variable. We will now define another auxiliary function $\otimes$ defined as follows:

Definition 33 - Product of Contexts: Let $\Gamma_1, \ldots, \Gamma_n$ be contexts, and $\Delta_1, \ldots, \Delta_n$ be sets of type definitions defining the type symbols in each Γ_i, respectively, such that no to Δs define the same type symbol. Let V be the set of variables that occur in more than one context.
$\otimes\big((\Gamma_1, \ldots, \Gamma_n), (\Delta_1, \ldots, \Delta_n)\big) = (\Gamma, \Delta, Eq)$, where:
$\Gamma(X) = \sigma\prime$, where $\sigma\prime$ is a fresh type symbol, for all $X \in V$, and $\Gamma(X) = \Gamma_i(X)$, for all $X \notin V \wedge X \in domain(\Gamma_i)$;
$\Delta(\sigma) = \alpha$, where α is a fresh type variable, for all type symbols $\sigma \notin \Delta_1 \cup \cdots \cup \Delta_n$, and $\Delta(\sigma) = \Delta_i(\sigma)$, otherwise;
$Eq = \{\alpha \doteq \Gamma_{i_1}(X), \ldots, \alpha \doteq \Gamma_{i_k}(X)\}$, for all fresh α, for each Γ_{i_j}, such that $X \in def(\Gamma_{i_j})$.

We can interpret the product of contexts operation as a way of making sure the types for each variable are the same in every context. That may not be possible.

Example 41: Let
$\Gamma_1 = \{X : \sigma_1, Y : \sigma_2\}$, $\Delta_1 = \{\sigma_1 = int,\ \sigma_2 = \alpha\}$,
$\Gamma_2 = \{X : \sigma_3, Y : \sigma_4\}$, and $\Delta_2 = \{\sigma_3 = float,\ \sigma_4 = float\}$.
Then:
$\otimes\big((\Gamma_1, \Gamma_2), (\Delta_1, \Delta_2)\big) = (\Gamma_3, \Delta_3, Eq)$, where
$\Gamma_3 = \{X : \sigma_5, Y : \sigma_6\}$,
$\Delta_3 = \{\sigma_5 = \beta, \sigma_6 = \gamma\}$,
$Eq = \{\beta \doteq int, \beta \doteq float, \gamma \doteq \alpha, \gamma \doteq float\}$).

As we can see in the previous example, the type for variable X in the resulting context is σ_5, defined as $\sigma_5 = \beta$. For β, we have $\beta \doteq int$ and $\beta \doteq float$. There is no substitution that satisfies these constraints, since we consider basic types are disjoint. However, if we just consider the variable Y, the constraints generated $\{\gamma \doteq \alpha, \gamma \doteq float\}$ can be satisfied, in particular by the substitution $[\gamma \mapsto float, \alpha \mapsto float]$.

We are now ready to present the constraint generation algorithm. Let P be a term, an atom, a query, a sequence of queries, or a clause. $generate(P)$ is a function that outputs a tuple of the form $(\tau, \Gamma, Eq, Ineq, \Delta)$, where τ is a type, Γ is an context for variables, Eq is a set of equality constraints, $Ineq$ is a set of subtyping constraints, and Δ is a set of type definitions. The function *generate*, which generates the initial type constraints, is defined case by case from the program syntax. Its definition follows:

$generate(P) =$

- $generate(X) = (\alpha, \{X : \sigma\}, \emptyset, \emptyset, \{\sigma = \alpha\})$, X is a variable,
  where α is a fresh type variable and σ is a fresh type symbol.

- $generate(c) = (basetype(c), \emptyset, \emptyset, \emptyset, \emptyset)$, c is a constant.

- $generate(f(t_1, \ldots, t_n)) = (f(\tau_1, \ldots, \tau_n), \Gamma, Eq, \emptyset, \Delta)$, f is a function symbol,
  where $generate(t_i) = (\tau_i, \Gamma_i, Eq_i, \emptyset, \Delta_i)$,
  $(\Gamma, \Delta, Eq\prime) = \otimes\big((\Gamma_1, \ldots, \Gamma_n), (\Delta_1, \ldots, \Delta_n)\big)$,
  and $Eq = Eq_1 \cup \ldots \cup Eq_n \cup Eq\prime$.

- $generate(t_1 = t_2) = (bool, \Gamma, Eq, \emptyset, \Delta)$
  where $generate(ti) = (\tau_i, \Gamma_i, Eq_i, \emptyset, \Delta_i)$,
  $(\Gamma, \Delta, Eq\prime) = \otimes\big((\Gamma_1, \Gamma_2), (\Delta_1, \Delta_2)\big)$, and
  $Eq = Eq_1 \cup Eq_2 \cup \{\tau_1 \doteq \tau_2\} \cup Eq\prime$.

- $generate(p(X_1, \ldots, X_n)) = (bool, (\{X_1 : \sigma_1, \ldots, X_n : \sigma_n\}, \emptyset, \{\sigma_1 \leq \tau_1, \ldots, \sigma_n \leq \tau_n\}, \Delta\prime)$, p is a predicate symbol,
  where $generate(p(Y_1, \ldots, Y_n) : -body) = (bool, \Gamma, Eq, Ineq, \Delta)$,
  $\{Y_1 : \tau_1, \ldots Y_n : \tau_n\} \in \Gamma$
  $\Delta\prime = \Delta \cup \{\sigma_i = \alpha_i\}$, and σ_i and α_i are all fresh.

- $generate(c_1, \ldots, c_n) = (bool, \Gamma, Eq, Ineq_1 \cup \ldots \cup Ineq_n, \Delta)$, a query,
  where $generate(c_i) = (bool, \Gamma_i, Eq_i, Ineq_i, \Delta_i)$,

$(\Gamma, \Delta, Eq\prime) = \otimes\big((\Gamma_1, \ldots, \Gamma_n), (\Delta_1, \ldots, \Delta_n)\big)$, and
$Eq = Eq_1 \cup \ldots \cup Eq_n \cup Eq\prime$.

- $generate(p(X_1, \ldots, X_n) : -b_1; \ldots; b_n.) = (bool, \Gamma, Eq, Ineq, \Delta)$, a non-recursive clause,
  where $generate(b_i) = (bool, \Gamma_i, Eq_i, Ineq_i, \Delta_i)$,
  $Eq = Eq_1 \cup \ldots \cup Eq_n$,
  $Ineq = Ineq_1 \cup \ldots \cup Ineq_n$, and
  $(\Gamma, \Delta) = \oplus\big((\Gamma_1, \ldots, \Gamma_n), (\Delta_1, \ldots, \Delta_n)\big)$.

- $generate(p(X_1, \ldots, X_n) : -body) = (bool, \Gamma, Eq, Ineq\prime, \Delta)$, a recursive clause,
  where $generate(p(X_1, \ldots, X_n) : -body\prime) = (bool, \Gamma, Eq, Ineq, \Delta)$, such that $body\prime$ is $body$ after removing all recursive calls,
  and $Ineq\prime = Ineq \cup \{\overrightarrow{\sigma_1} \leq \overrightarrow{\tau}, \ldots, \overrightarrow{\sigma_k} \leq \overrightarrow{\tau}, \overrightarrow{\tau} \leq \overrightarrow{\sigma_1}, \ldots, \overrightarrow{\tau} \leq \overrightarrow{\sigma_k}\}$, such that τ are the types for the variables in the head of the clause in Γ and σ_i are the types for the variables in each recursive call.

The function *basetype* used in the constant case corresponds to the function *type* in the type system, but it is assumed to only output base types for constants. For the complex case, we assume that the type for every function symbol or arity n is $\alpha_1 \times \cdots \times \alpha_n \to f(\alpha_1, \ldots, \alpha_n)$. In the next chapter, we will discuss the modifications necessary to the algorithm in order to have a different interpretation of function symbols.

We can see that inside a single query, we perform the $\otimes$ operation to the contexts from each call and unification, to make sure the types for each variable inside a query are the same. However, for different queries, we perform the $\oplus$ operation, since the types need not be the same, and instead we need to consider the union of all possibilities.

Let us illustrate constraint generation for a simple example bellow.

Example 42: Consider the following predicate:

```
list(X) :- X = []; X = [Y|YS], list(Ys).
```

the output of applying the generate function to the predicate is:
$generate(list(X) : -X = [\,]; X = [Y|Ys], list(Ys)) =$
$\{bool, \{X : \sigma_1, Y : \sigma_2, Ys : \sigma_3\}, \{\alpha \doteq [\,], \beta \doteq [\delta \mid \epsilon]\}, \{\sigma_3 \leq \sigma_1, \sigma_1 \leq \sigma_3\}, \{\sigma_1 = \alpha + \beta, \sigma_2 = \delta, \sigma_3 = \epsilon\}\}$
The set $\{\sigma_3 \leq \sigma_1, \sigma_1 \leq \sigma_3\}$ comes from the recursive call to the predicate, while $\alpha \doteq [\,]$ comes from $X = [\,]$, and $\beta \doteq [\delta \mid \epsilon]$ comes from $X = [Y|Ys]$. The definition $\sigma_1 = \alpha + \beta$ comes from the application of the $\oplus$ operation.

Now that we have presented the algorithm that generates constraints for each predicate definition, we will present the algorithm that solves the constraints, in order to obtain a substitution for type variables, such that when applied to the set of type definitions gives us the types for the predicate.

5.1.3 Constraint Solving

Let Eq be a set of equality constraints, $Ineq$ be a set of subtyping constraints, and Δ a set of type definitions. Function $solve(Eq, Ineq, \Delta)$ is a rewriting algorithm that solves the constraints, outputting a pair of a substitution and a new set of type definitions. Note that the rewriting rules in the following definitions of the solver algorithm are assumed to be ordered.

Definition 34 - Solved Form: A set of equality constraints Eq is in *solved form* if:

- all constraints are of the form $\alpha_i \doteq \tau_i$;
- there are no two constraints with the same α_i on the left hand side;
- no type variables on the left-hand side of the equations occur on the right-hand side of equations.

A set of equality constraints in solved form can be interpreted as a substitution, where each constraint $\alpha_i \doteq \tau_i$ corresponds to a substitution for the type variable α_i, $[\alpha_i \mapsto \tau_i]$.

A *configuration* is either the term *fail* (representing failure), a pair of a substitution and a set of type definitions (representing the end of the algorithm), or a triple of a set of equality constraints Eq, a set of subtyping constraints $Ineq$, and a set of type definitions Δ. The following rewriting algorithm consists of the transformation rules on configurations.

$solve(Eq, Ineq, \Delta) =$

1. $(\{\tau \doteq \tau\} \cup Eq, Ineq, \Delta) \rightarrow (Eq, Ineq, \Delta)$
2. $(\{\alpha \doteq \tau\} \cup Eq, Ineq, \Delta) \rightarrow (\{\alpha \doteq \tau\} \cup Eq[\alpha \mapsto \tau], Ineq[\alpha \mapsto \tau], \overline{\Delta[\alpha \mapsto \tau]})$, if type variable α occurs in Eq, $Ineq$, or Δ
3. $(\{\tau = \alpha\} \cup Eq, Ineq, \Delta) \rightarrow (\{\alpha \doteq \tau\} \cup Eq, Ineq, \Delta)$, where α is a type variable and τ is not a type variable
4. $(\{f(\tau_1, \ldots, \tau_n) \doteq f(\tau\prime_1, \ldots, \tau\prime_n)\} \cup Eq, Ineq, \Delta \rightarrow (\{\tau_1 \doteq \tau\prime_1, \ldots, \tau_n \doteq \tau\prime_n\} \cup Eq, Ineq, \Delta)$

5. $(\{f(\tau_1, \ldots, \tau_n) \doteq g(\tau\prime_1, \ldots, \tau\prime_m)\} \cup Eq, Ineq, \Delta) \rightarrow fail$

6. $(Eq, \{\tau \leq \tau\} \cup Ineq, \Delta) \rightarrow (Eq, Ineq, \Delta)$

7. $(Eq, \{f(\tau_1, \ldots, \tau_n) \leq f(\tau\prime_1, \ldots, \tau\prime_n)\} \cup Ineq, \Delta) \rightarrow (Eq, \{\tau_1 \leq \tau\prime_1, \ldots, \tau_n \leq \tau\prime_n\} \cup Ineq, \Delta)$

8. $(Eq, \{\alpha \leq \tau_1, \alpha \leq \tau_2\} \cup Ineq, \Delta) \rightarrow (Eq \cup Eq\prime, \{\alpha \leq \tau\} \cup Ineq, \Delta\prime)$, where α is a type variable, and $intersect(\tau_1, \tau_2, \Delta, I) = (\tau\prime, Eq\prime, \Delta\prime)$

9. $(Eq, \{\alpha \leq \tau\} \cup Ineq, \Delta) \rightarrow (Eq \cup \{\alpha = \tau\}, Ineq, \Delta)$, where α is a type variable and no other constraints exist with α on the left-hand side

10. $(Eq, \{\tau_1 + \ldots + \tau_n \leq \tau\} \cup Ineq, \Delta) \rightarrow (Eq, \{\tau_1 \leq \tau, \ldots, \tau_n \leq \tau\} \cup Ineq, \Delta)$

11. $(Eq, \{\sigma \leq \tau\} \cup Ineq, \Delta) \rightarrow (Eq, Ineq, \Delta)$, if (σ,τ) are on the store of pairs of types that have already been compared

12. $(Eq, \{\sigma \leq \tau\} \cup Ineq, \Delta) \rightarrow (Eq, \{\Delta(\sigma) \leq \tau\} \cup Ineq, \Delta)$, where σ is a type symbol defined in Δ. Also add (σ, τ) to the store of pairs of types that have been compared

13. $(Eq, \{\tau_1 \leq \alpha, \ldots \tau_n \leq \alpha\} \cup Ineq, \Delta) \rightarrow (Eq[\alpha \mapsto \sigma], Ineq[\alpha \mapsto \sigma], \overline{\Delta \cup \{\sigma = \tau_1 + \cdots + \tau_n\}})$, where σ is a fresh type symbol

14. $(Eq, \{\tau \leq \tau_1 + \ldots + \tau_n\} \cup Ineq, \Delta) \rightarrow (Eq, \{\tau \leq \tau_i\} \cup Ineq, \Delta)$, where τ_i is one of the summands

15. $(Eq, \{\tau \leq \sigma\} \cup Ineq, \Delta) \rightarrow (Eq, Ineq, \Delta)$, if (σ,τ) are on the store of pairs of types that have already been compared

16. $(Eq, \{\tau \leq \sigma\} \cup Ineq, \Delta) \rightarrow (Eq, \{\tau \leq \Delta(\sigma)\} \cup Ineq, \Delta)$, where σ is a type symbol defined in Δ. Also add (σ, τ) to the store of pairs of types that have been compared

17. $(Eq, \emptyset, \Delta) \rightarrow (Eq, \Delta)$

18. otherwise $\rightarrow$ fail.

Note that an occur check is required in steps 2, 9, and 13. This rewriting algorithm is based on the one described in [77] for equality constraints, and an original one for the subtyping constraints. Also note that a store is used. This is to ensure termination, since we are dealing with possibly recursive types. The pairs of types that are introduced into the store, correspond to

assumptions we are making that can later be used as facts, as happens in Definition 27, when unfolding type symbols.

We will now show an example of the execution of the algorithm on the output of the constraint generation algorithm, showed in example 42.

Example 43: Following Example 42, we can apply *solve* to the tuple $(Eq, Ineq, \Delta)$, corresponding to $(\{X : \sigma_1, Y : \sigma_2, Ys : \sigma_3\}, \{\alpha \doteq [\,], \beta \doteq [\delta \mid \epsilon]\}, \{\sigma_3 \leq \sigma_1, \sigma_1 \leq \sigma_3\}, \{\sigma_1 = \alpha + \beta, \sigma_2 = \delta, \sigma_3 = \epsilon\})$:
$(\{\alpha \doteq [\,], \beta \doteq [\delta \mid \epsilon]\}, \{\sigma_3 \leq \sigma_1, \sigma_1 \leq \sigma_3\}, \{\sigma_1 = \alpha + \beta, \sigma_2 = \delta, \sigma_3 = \epsilon\}) \rightarrow_2$

$(\{\alpha \doteq [\,], \beta \doteq [\delta \mid \epsilon]\}, \{\sigma_3 \leq \sigma_1, \sigma_1 \leq \sigma_3\}, \{\sigma_1 = [\,] + \beta, \sigma_2 = \delta, \sigma_3 = \epsilon\}) \rightarrow_2$

$(\{\alpha \doteq [\,], \beta \doteq [\delta \mid \epsilon]\}, \{\sigma_3 \leq \sigma_1, \sigma_1 \leq \sigma_3\}, \{\sigma_1 = [\,] + [\delta \mid \epsilon], \sigma_2 = \delta, \sigma_3 = \epsilon\}) \rightarrow_{12}$

$(\{\alpha \doteq [\,], \beta \doteq [\delta \mid \epsilon]\}, \{\epsilon \leq \sigma_1, \sigma_1 \leq \sigma_3\}, \{\sigma_1 = [\,] + [\delta \mid \epsilon], \sigma_2 = \delta, \sigma_3 = \epsilon\}) \rightarrow_9$

$(\{\alpha \doteq [\,], \beta \doteq [\delta \mid \epsilon], \epsilon \doteq \sigma_1\}, \{\sigma_1 \leq \sigma_3\}, \{\sigma_1 = [\,] + [\delta \mid \epsilon], \sigma_2 = \delta, \sigma_3 = \epsilon\}) \rightarrow_2$

$(\{\alpha \doteq [\,], \beta \doteq [\delta \mid \sigma_1], \epsilon \doteq \sigma_1\}, \{\sigma_1 \leq \sigma_3\}, \{\sigma_1 = [\,] + [\delta \mid \sigma_1], \sigma_2 = \delta, \sigma_3 = \sigma_1\}) \rightarrow_s$

$(\{\alpha \doteq [\,], \beta \doteq [\delta \mid \sigma_1], \epsilon \doteq \sigma_1\}, \{\sigma_1 \leq \sigma_1\}, \{\sigma_1 = [\,] + [\delta \mid \sigma_1], \sigma_2 = \delta\}) \rightarrow_6$

$(\{\alpha \doteq [\,], \beta \doteq [\delta \mid \sigma_1], \epsilon \doteq \sigma_1\}, \emptyset, \{\sigma_1 = [\,] + [\delta \mid \sigma_1], \sigma_2 = \delta\})$

Note that the resulting set of constraints only contains constraints in solved form, that can be seen as a substitution. Step $\rightarrow_s$, stands for the simplification step. Therefore, the resulting context Γ is $\{X : \sigma_1, Y : \sigma_2, Ys : \sigma_1\}$.

We can see that the algorithm uses intersection of types on case 8. This intersection algorithm tries to find the type that corresponds to the intersection of several types. Below, we present the algorithm. Note that intersect of type variables is not defined in general. We use unification for the intersection of variables.

Type intersection represented by $intersect(\tau_1, \tau_2, \Delta, I) = (\tau, Eq\prime, \Delta\prime)$, is calculated as follows:

- if both τ_1 and τ_2 are different type variables, then $\tau = \tau_2, \Delta\prime = \Delta, Eq\prime = \{\tau_1 \doteq \tau_2\}$.
- if $\tau_1 = \tau_2$, then $\tau = \tau_1, \Delta\prime = \Delta, Eq\prime = \emptyset$.

- if $(\tau_1, \tau_2, \tau_3) \in I$, then $\tau = \tau_3, \Delta\prime = \Delta, Eq\prime = \emptyset$.
- if τ_1 is a type variable, then $\tau = \tau_2, \Delta\prime = \Delta, Eq\prime = \{\tau_1 \doteq \tau_2\}$.
- if τ_2 is a type variable, then $\tau = \tau_1, \Delta\prime = \Delta, Eq\prime = \{\tau_2 \doteq \tau_1\}$.
- if $\tau_1 = \sigma_1, \tau_2 = \sigma_2$, and $(\widetilde{\tau}, Eq, \Delta_2) = cpi(\widetilde{\tau_1}, \widetilde{\tau_2}, \Delta, I \cup \{(\sigma_1, \sigma_2, \sigma_3)\})$, then $\tau = \sigma_3, \Delta\prime = \overline{\Delta_2 \cup \{\sigma_3 = \widetilde{\tau}\}}, Eq\prime = Eq$, where $\sigma_1 = \widetilde{\tau_1}, \sigma_2 = \widetilde{\tau_2} \in \Delta$ and σ_3 is fresh.
- if $\tau_1 = \sigma_1, \tau_2 = f(t_1, \ldots, t_n)$, and given $(\widetilde{\tau}, Eq, \Delta_2) = cpi(\widetilde{\tau}, \tau_2, \Delta, I \cup \{(\sigma_1, \tau_2, \sigma_3)\})$, then $\tau = \sigma_3, \Delta\prime = \overline{\Delta_2 \cup \{\sigma_3 = \widetilde{\tau}\}}, Eq\prime = Eq$, where $\sigma_1 = \widetilde{\tau_1} \in \Delta$ and σ_3 is fresh. Same for $\tau_2 = \sigma_1$ and $\tau_1 = f(t_1, \ldots, t_n)$.
- if $\tau_1 = f(\tau_1, \ldots, \tau_n), \tau_2 = f(\tau\prime_1, \ldots, \tau\prime_n)$, and given $\forall i.(\tau\prime\prime_i, Eq_i, \Delta_i) = intersect(\tau_i, \tau\prime_i, \Delta, I)$, then $\tau = f(\tau\prime\prime_1, \ldots, \tau\prime\prime_n), \Delta\prime = \Delta_1 \cup \ldots \cup \Delta_n$, $Eq\prime = Eq_1 \cup \ldots \cup Eq_n$.
- otherwise fail.

$cpi(\widetilde{\tau_1}, \widetilde{\tau_2}, \Delta, I)$ is a function that applies $intersect(\tau, \tau\prime, \Delta, I)$ to every pair of types $\tau, \tau\prime$, such that $\tau \in \widetilde{\tau_1}$ and $\tau\prime \in \widetilde{\tau_2}$, and gathers all results as the output.

Example 44: Let $\Delta = \{\sigma_1 = f(\alpha, \alpha), \sigma_2 = f(int, float)\}$. Then:
$intersect(\sigma_1, \sigma_2, \Delta, \emptyset)$

$cpi(f(\alpha, \alpha), f(int, float), \Delta, I = \{(\sigma_1, \sigma_2, \sigma_3)\})$

$intersect(f(\alpha, \alpha), f(int, float), \Delta, I)$

$intersect(\alpha, int, \Delta, I) = (int, \Delta, \{\alpha \doteq int\})$

$intersect(\alpha, float, \Delta, I) = (float, \Delta, \{\alpha \doteq float\})$

$= (f(int, float), \Delta, Eq = \{\alpha \doteq int, \alpha \doteq float\})$

$= (f(int, float), \Delta, Eq)$

$= (\sigma_3, \overline{\Delta \cup \{\sigma_3 = f(int, float)\}}, Eq)$

Now note that the set of generated constraints, when solved, will fail. Therefore the intersection of σ_1 and σ_2 is empty.

This intersection algorithm is based on the one presented in [6], with a few minor changes. The difference is that our types can be type variables, which could not happen in Zobel's algorithm, since intersection was only calculated between ground types. To deal with this extension, in our algorithm type variables are treated as Zobel's *any* type, except that for type variables, we also unify them with whichever type they are being intersected with. Termination and correctness of type intersection for a tuple distributive version of Zobel's algorithm was proved previously in [8] and replacing the *any* type with type variables maintains the same properties, because our

use of type intersection considers types where type variables occur only once, thus they can be safely replaced by Zobel's *any* type. Note that we deal with type variables which occur more than once with calls to type unification.

5.1.4 Decidability

One important property of the type inference algorithm is that its execution halts for any input, and if it halts with success, then the resulting set of equalities is in solved form.

The following theorem shows that the constraint solver terminates at every input set of constraints.

Theorem 10 - Termination: *solve* always terminates, and when *solve* terminates, it either fails or the output is a pair of a substitution and a new set of type definitions.

Proof: We will define the following metrics for *solve*:

- NPC: number of possible comparisons between types, that have not been made yet.
- NVRS: number of variables on the right-hand side of constraints.
- NSC: number of symbols in the constraints.
- NI: number of subtyping constraints.

We will prove termination by showing that NI reduces to zero. Termination of *solve* is proven by a measure function that maps the constraint set to a tuple (NPC,NVRS,NSC,NI). The following table shows that each step decreases the tuple w.r.t. the lexicographical order of the tuple.

	NPC	NVRS	NSC	NI
1.	$=$	$\leq$	$<$	
2.	$=$	$<$		
3.	$=$	$<$		
4.	$=$	$=$	$<$	
5.	0	0	0	0
6.	$=$	$\leq$	$<$	
7.	$=$	$=$	$<$	
8.	$=$	$\leq$	$\leq$	$<$
9.	$=$	$=$	$=$	$<$
10.	$=$	$=$	$<$	
11.	$=$	$\leq$	$<$	
12.	$<$			
13.	$=$	$<$		
14.	$=$	$\leq$	$<$	
15.	$=$	$=$	$<$	
16.	$<$			
17.	$=$	$=$	$=$	$<$
18.	0	0	0	0

□

The proof for this theorem follows a usual termination proof approach, where we show that a carefully chosen metric decreases at every step.

To guarantee that the output set of equality constraints is in solved form, in order to be interpreted as a substitution, we also prove the lemma below.

Lemma 10: If $solve(Eq, Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, then S is in solved form.

Proof: Suppose S was not in solved form. Then, either:

1. there is more than one constraint with the same variable on left-hand side,
2. a variable on the left-hand side of a constraint occurs somewhere on the right-hand side of a constraint,
3. some constraint has something that is not a variable on the left-hand side.

In this case, either we can apply rule 2, for the first two cases, or rule 3, 4, or 5, for the last case. Either way, the algorithm, would not have finished executing and output S. □

By proving the following lemma, we prove that the output set of equality constraints is in solved form and therefore can be interpreted as a substitution.

5.1.5 Soundness

Here we prove that the type inference algorithm is sound, in the sense that inferred types are derivable in the type system, which defines well-typed programs. For this we need the following auxiliary definitions and lemmas which are used in the proofs of the main theorems.

The following lemmas state properties of the constraint satisfaction relation $\models$, the $\otimes$ operation, and the type intersection operation.

Lemma 11 - Distribution of Constraints: If we have S such that $S \models_\Delta C \cup C\prime$, then $S \models_\Delta C$ and $S \models_\Delta C\prime$.

Proof: If $S \models_\Delta C \cup C\prime$, then for all constraints $c_i \in C \cup C\prime$, $S(c_i)$ is true, meaning if c_i is an equality constraint $t_1 \doteq t_2$, then $S(t_1) = S(t_2)$, and if c_i is a subtyping constraint $s_1 \leq s_2$, then $S(s_1) \sqsubseteq_\Delta S(s_2)$. In particular, for all constraints $c\prime_i \in C$, and for all constraints $c\prime\prime_i \in C\prime$, the same holds. Therefore $S \models_\Delta C$ and $S \models_\Delta C\prime$. □

This lemma guarantees that any substitution that models some constraints, also models any subset of that set of constraints.

The following lemma relates syntactic equality with the concept of equivalence.

Lemma 12 - Syntactic Equality and Equivalence: Suppose $\tau_1 = \tau_2$, then $\tau_1 \equiv_\Delta \tau_2$, for any Δ.

Proof: Since $\tau_1 = \tau_2$ then τ_1 and τ_2 are equal syntactically. Let $\tau_1 = \tau$ and $\tau_2 = \tau$. For any Δ, $\mathbf{T}[\![\tau_1]\!]_\Delta = \mathbf{T}[\![\tau]\!]_\Delta$ and $\mathbf{T}[\![\tau_2]\!]_\Delta = \mathbf{T}[\![\tau]\!]_\Delta$. Therefore $\mathbf{T}[\![\tau_1]\!]_\Delta = \mathbf{T}[\![\tau_2]\!]_\Delta$. So we can conclude that for any S, $\mathbf{T}[\![S(\tau_1)]\!]_\Delta = \mathbf{T}[\![S(\tau_2)]\!]_\Delta$, which means $\tau_1 \equiv_\Delta \tau_2$. □

This allows us to prove the following property.

Lemma 13 - $\otimes$ Property: Let $(\Gamma, \Delta, Eq) = \otimes((\Gamma_1, \ldots, \Gamma_n), (\Delta_1, \ldots, \Delta_n))$, $S \models_\Delta Eq$, and $\forall i.\Gamma_i, \overline{S(\Delta_i)} \vdash M_i : \tau_i$, then $\forall i.\Gamma, \overline{S(\Delta)} \vdash M_i : \tau\prime_i$, where $\tau\prime_i \equiv_\Delta \tau_i$.

Proof: If there are no common variables between two different contexts then this is trivially true, since $\Gamma = \Gamma_1 \cup \ldots \cup \Gamma_n$ and $\Delta = \Delta_1 \cup \ldots \cup \Delta_n$, and we can freely add irrelevant information to the rules in the type system.

Let us assume there is some common variable X between two contexts Γ_i and Γ_j. Then, there will be constraints $\alpha = \tau_1$ and $\alpha = \tau_2$ in Eq, where $X : \sigma_1 \in \Gamma_i$ and $X : \sigma_2 \in \Gamma_j$, $(\sigma_1 = \tau_1) \in \Delta_i$ and $(\sigma_2 = \tau_2) \in \Delta_j$, while $X : \sigma \in \Gamma$ and $(\sigma = \alpha) \in \Delta$.

Since $S \models_\Delta Eq$, by lemma 11, it also models the constraints above, which means that $S(\alpha) = S(\tau_1)$ and $S(\alpha) = S(\tau_2)$, which implies, from Lemma 12, $S(\alpha) \equiv_\Delta S(\tau_1) \equiv_\Delta \tau_2$. But since, by definition, $S(\sigma_1) \equiv_\Delta S(\tau_1)$, $S(\sigma_2) \equiv_\Delta S(\tau_2)$, and $S(\sigma) \equiv_\Delta S(\tau)$, we know that $S(\sigma) \equiv_\Delta S(\sigma_1) \equiv_\Delta S(\sigma_2)$. Therefore, the type for M in derivation $\Gamma, \overline{S(\Delta)} \vdash M_i : \tau\prime_i$ is such that $\Gamma_i, \overline{S(\Delta_i)} \vdash M_i : \tau_i$, and $\tau\prime_i \equiv \Delta\tau_i$.

We can follow a similar argument for all other variables that occur in more than one context.

Therefore the types for all variables in a context Γ_i using definitions $\overline{S(\Delta_i)}$ will be the same using the context Γ and definitions $\overline{S(\Delta)}$, and if we have $\Gamma_i, \overline{S(\Delta_i)} \vdash M_i : S(\tau_i)$ we can use $\Gamma, \overline{S(\Delta)} \vdash M_i : S(\tau_i)$, for all i. □

This property guarantees that we can extend contexts and sets of type definitions with respect to the $\otimes$ operation and get equivalent results.

Lemma 14: If $intersect(\tau_1, \tau_2, I, \Delta) = (\tau, Eq, \Delta\prime)$ and $S \models_{\Delta\prime} Eq$, then $S(\tau) \sqsubseteq_{\Delta\prime} S(\tau_1)$, and $S(\tau) \sqsubseteq_{\Delta\prime} S(\tau_2)$.

Proof: The proof follows by induction on the number of steps until the *intersect* function finishes.

- Suppose τ_1 and τ_2 are different type variables. Then $(\tau, Eq, \Delta\prime) = (\tau_2, \{\tau_1 = \tau_2\}, \Delta)$. For any S, such that $S \models_\Delta \tau_1 \doteq \tau_2$, then $S(\tau_1) = S(\tau_2)$. By Lemma 12, we know that $S(\tau_1) \equiv_\Delta S(\tau_2)$. Therefore, $S(\tau_2) \sqsubseteq_\Delta S(\tau_1)$ from the definition of $\equiv_{\Delta\prime}$, and $S(\tau_2) \sqsubseteq_\Delta S(\tau_2)$, trivially.

- Suppose τ_1 and τ_2 are equal. Then $(\tau, Eq, \Delta\prime) = (\tau_1, \emptyset, \Delta)$. Any S models the empty set of constraints. But since $\tau_1 = \tau_2$, we know that for any S, $S(\tau_1) = S(\tau_2)$. By Lemma 12, we know that $S(\tau_1) \equiv_\Delta S(\tau_2)$. Therefore, $S(\tau_2) \sqsubseteq_\Delta S(\tau_1)$ from the definition of $\equiv_\Delta$, and $S(\tau_2) \sqsubseteq_\Delta S(\tau_2)$, trivially.

- Suppose the tuple (τ_1, τ_2, τ_3) is in I. Then we are assuming that $intersect(\tau_1, \tau_2, I \setminus \{(\tau_1, \tau_2, \tau_3)\}, \Delta) = (\tau_3, \emptyset, \Delta)$. By the induction hypothesis, if this is true, then by the induction hypothesis, we know that for any S, $S(\tau_3) \sqsubseteq_\Delta S(\tau_1)$ and $S(\tau_3) \sqsubseteq_\Delta S(\tau_2)$.

- Suppose τ_1 is a type variable and τ_2 is not. Then $(\tau, Eq, \Delta\prime) = (\tau_2, \{\tau_1 \doteq \tau_2\}, \Delta)$. If some S is much that $S \models_\Delta \tau_1 \doteq \tau_2$, then $S(\tau_1) = S(\tau_2)$. By Lemma 12, we know that $S(\tau_1) \equiv_\Delta S(\tau_2)$. Therefore, $S(\tau_2) \sqsubseteq_\Delta S(\tau_1)$ from the definition of $\equiv_\Delta$, and $S(\tau_2) \sqsubseteq_\Delta S(\tau_2)$, trivially.

- Same as the previous case, replace τ_1 by τ_2 and vice-versa.

- Suppose both τ_1 and τ_2 are type symbols. Then $(\tau, Eq, \Delta\prime) = (\sigma_3, Eq, \overline{\Delta_2 \cup \{\sigma_3 = \widetilde{\tau}\}})$, where $cpi(\widetilde{\tau_1}, \widetilde{\tau_2}, \Delta, I \cup \{(\tau_1, \tau_2, \sigma_3)\}) = (\sigma_3, Eq, \Delta_2\})$. By the induction hypothesis, since *cpi* corresponds to multiple applications of the *intersect* function, we assume that $S(\widetilde{\tau}) \sqsubseteq_{\Delta_2} S(\widetilde{\tau_1})$ and $S(\widetilde{\tau}) \sqsubseteq_{\Delta_2} S(\widetilde{\tau_2})$, for all S such that $S \models_{\Delta_2} Eq$. Since: 1) $S(\widetilde{\tau}) \equiv_{\Delta\prime} S(\sigma_3)$, by definition; 2) $S(\widetilde{\tau}) \sqsubseteq_{\Delta\prime} S(\widetilde{\tau_1})$ and $S(\widetilde{\tau}) \sqsubseteq_{\Delta\prime} S(\widetilde{\tau_2})$ is still true, since σ_3 is a fresh type symbol that does not occur in τ_1 nor in τ_2; and 3) any S such that $S \models_{\Delta_2} Eq$ is such that $S \models_{\Delta\prime} Eq$, since σ_3 is a fresh type symbol that does not occur in Eq, then $S(\sigma_3) \sqsubseteq_{\Delta\prime} S(\widetilde{\tau_1})$ and $S(\sigma_3) \sqsubseteq_{\Delta\prime} S(\widetilde{\tau_2})$ for any S such that $S \models_{\Delta\prime} Eq$.

- Suppose τ_1 is a type symbol and $\tau_2 = f(\tau_1\prime, \ldots, \tau_n\prime)$. Then $(\tau, Eq, \Delta\prime) = (\sigma_3, Eq, \overline{\Delta_2 \cup \{\sigma_3 = \widetilde{\tau}\}})$, where $cpi(\widetilde{\tau_1}, \tau_2, \Delta, I \cup \{(\tau_1, \tau_2, \tau_3)\}) = (\sigma_3, Eq, \Delta_2)$. By the induction hypothesis, we know that for any S such that $S \models_{\Delta_2} Eq$, $S(\widetilde{\tau}) \sqsubseteq_{\Delta_2} S(\widetilde{\tau_1})$ and $S(\widetilde{\tau}) \sqsubseteq_{\Delta_2} S(\tau_2)$. Following a similar argument to the previous case, we prove that for any S, such that $S \models_{\Delta\prime} Eq$, we have that $S(\sigma_3) \sqsubseteq_{\Delta\prime} S(\tau_1)$ and $S(\sigma_3) \sqsubseteq_{\Delta\prime} S(\tau_2)$. The proof still holds for τ_2 being a type symbol and τ_1 a complex term.

- Suppose $\tau_1 = f(\tau\prime_1, \ldots, \tau\prime_n)$ and $\tau_2 = f(\tau\prime\prime_1, \ldots, \tau\prime\prime_n)$. Then we have $(\tau, Eq, \Delta\prime) = (f(\tau\prime\prime\prime_1, \ldots, \tau\prime\prime\prime_n), Eq_1 \cup \cdots \cup Eq_n, \Delta_1 \cup \cdots \cup \Delta_n\})$, where $intersect(\tau_i\prime, \tau_i\prime\prime, \Delta, I) = (\tau_i\prime\prime\prime, Eq_i, \Delta_i)$. By the induction hypothesis, we know that any S such that $S \models_{\Delta_i} Eq_i$, $S(\tau_i\prime\prime\prime) \sqsubseteq_{\Delta_i} S(\tau_i\prime)$ and $S(\tau_i\prime\prime\prime) \sqsubseteq_{\Delta_i} S(\tau_i\prime\prime)$. In each of the previous statements we can replace Δ_i with $\Delta\prime$, since all new definitions in each Δ_i are defining fresh type symbols that do not occur in $\tau\prime_j$, $\tau\prime\prime_j$ nor Eq_j, for $j \neq i$. Then, since $S \models_{\Delta\prime} Eq_i$, then $S \models_{\Delta\prime} Eq_1, \ldots, Eq_n$. If $S(\tau\prime\prime\prime) \sqsubseteq_{\Delta\prime} S(\tau_i\prime)$, and $S(\tau\prime\prime\prime) \sqsubseteq_{\Delta\prime} S(\tau_i\prime\prime)$, we hope it is clear to realize that $f(\tau\prime\prime\prime_1, \ldots, \tau\prime\prime\prime_n) \sqsubseteq_{\Delta\prime} f(\tau\prime_1, \ldots, \tau\prime_n)$ and $f(\tau\prime\prime\prime_1, \ldots, \tau\prime\prime\prime_n) \sqsubseteq_{\Delta\prime} f(\tau\prime\prime_1, \ldots, \tau\prime\prime_n)$.

- The last case does not apply, since we do not get a tuple $(\tau, Eq, \Delta\prime)$ as a result.

□

A set of equality constraints can be interpreted as a substitution. The following lemma states that such substitution models the constraints themselves.

Lemma 15 - Self-Satisfiability: If Eq is a set of equality constraints in solved form, then for any Δ, $Eq \models_\Delta Eq$.

Proof: If Eq is in solved form, then every equality in Eq is of the form $\alpha \doteq \tau$, such that no α that occurs on the left-hand side of any equality occurs in any right-hand side of any equality. Therefore, we can interpret Eq as a substitution S, such that for every equality $\alpha \doteq \tau$ we have a substitution $\alpha \mapsto \tau$. Suppose we apply this substitution to the set of equalities Eq, then for each constraint $\alpha \doteq \tau$ we would have $S(\alpha) = S(\tau)$. But $S(\tau) = \tau$, since no variable that has a substitution in S occurs in any right-hand side of any equality. Since $S(\alpha) = \tau$, then for any Δ, we get $\tau = \tau$, so $Eq \models_\Delta Eq$. We prove the lemma. □

Now we have a theorem for the soundness of constraint generation which states that if one applies a substitution which satisfies the generated constraints to the type obtained by the constraint generation function, we get a well-typed program.

Theorem 11 - Soundness of Constraint Generation: For a program P, given a clause, a query, or term M, if $generate(M) = (\tau, \Gamma, Eq, Ineq, \Delta)$, then for any $S \models_\Delta Eq, Ineq$, and assuming *basetype* and *type* give the same types for constants and function symbols, we have $\Gamma, \overline{S(\Delta)} \vdash_P M : \tau\prime$, where $\tau\prime \equiv_{\overline{S(\Delta)}} S(\tau)$.

Proof: The proof will follow by induction on M.

- M is a variable.
  $generate(X) = (\alpha, \{X : \sigma\}, \emptyset, \emptyset, \{\sigma = \alpha\})$. Then, for any S, we can derive $\{X : \sigma\}, \overline{S(\{\sigma = \alpha\})} \vdash_P M : \sigma$ in the type system with one application of rule VAR, and, trivially, $\sigma \equiv_{\overline{S(\{\sigma=\alpha\})}} S(\alpha)$.

- M is a constant.
  $generate(c) = (basetype(c), \emptyset, \emptyset, \emptyset, \emptyset)$. Since $basetype(c) = type(c)$, which is a base type $\tau\prime$, then $S(\tau\prime) = \tau\prime$. Since the only instance $\tau \preceq \tau\prime$ is $\tau\prime$ itself, then we can use rule CST to derive $\emptyset, \emptyset \vdash_P c : \tau\prime$, and $\tau\prime \equiv_\Delta \tau\prime$, trivially.

- M is a complex term.
  $generate(f(t_1, \ldots, t_n)) = (f(\tau_1, \ldots, \tau_n), \Gamma, Eq, Ineq, \Delta)$, where for all i, $generate(t_i) = (\tau_i, \Gamma_i, Eq_i, Ineq_i, \Delta_i)$. We know that $basetype(f) =$

$type(f) = \alpha_1 \times \ldots \times \alpha_n \rightarrow f(\alpha_1, \ldots, \alpha_n)$. Then we know that $\tau_1 \times \ldots \tau_n \rightarrow f(\tau_1, \ldots, \tau_n) \preceq \alpha_1 \times \cdots \times \alpha_n \rightarrow f(\alpha_1, \ldots, \alpha_n)$. For any S, such that $S \models_\Delta Eq$, then, by lemma 11, we know that $\forall i. S \models_\Delta Eq_i$, and $S \models_\Delta Eq\prime$. Since $S \models_\Delta Eq_i$, by the induction hypothesis we know that $\Gamma_i, \overline{S(\Delta_i)} \vdash_P t_i : \tau\prime_i$, where $\tau\prime_i \equiv_{\Delta_i} S(\tau_i)$. Also, since $S \models_\Delta Eq\prime$, then, by lemma 13, we know that we can replace Γ_i by Γ and Δ_i by Δ in the previous rules. Therefore $\Gamma, S(\Delta) \vdash_P t_i : \tau\prime\prime_i$, where $\tau\prime\prime_i \equiv_\Delta \tau\prime_i \equiv_\Delta \tau_i$. Therefore we can derive in the system, using rule CPL, $\Gamma, \overline{S(\Delta)} \vdash_P f(t_1, \ldots, t_n) : f(\tau\prime\prime_1, \ldots, \tau\prime\prime_n)$, where $\tau\prime\prime_i \equiv_\Delta \tau_i$.

- M is an equality.
  $generate(t_1 = t_2) = (bool, \Gamma, Eq, \emptyset, \Delta)$, where for all i, $generate(t_i) = (\tau_i, \Gamma_i, Eq_i, \emptyset, \Delta_i)$, $(\Gamma, D, Eq\prime) = \otimes((\Gamma_1, \Gamma_2), (\Delta_1, \Delta_2))$, and $Eq = Eq_1 \cup Eq_2 \cup \{\tau_1 \doteq \tau_2\} \cup Eq\prime$. For any S, such that $S \models Eq$, then, by lemma 11, we know that $\forall i. S \models_\Delta Eq_i$, $S \models_\Delta \tau_1 \doteq \tau_2$, and $S \models_\Delta Eq\prime$. Since $S \models_\Delta Eq_i$, by the induction hypothesis we know that $\Gamma_i, \overline{S(\Delta_i)} \vdash_P t_i : \tau\prime_i$, where $\tau\prime_i \equiv_\Delta \tau_i$. Also, since $S \models_\Delta Eq\prime$, then, by lemma 13, we know that we can replace Γ_i by Γ and Δ_i by Δ in the previous rules. Therefore $\Gamma, \overline{S(\Delta)} \vdash_P t_i : \tau\prime\prime_i$, where $\tau\prime\prime_i \equiv_\Delta \tau\prime_i \equiv_\Delta \tau_i$, and since $S \models_\Delta \tau_1 \doteq \tau_2$, then $S(\tau_1) = S(\tau_2)$, and therefore $S(\tau_1) \equiv_\Delta S(\tau_2)$. Finally, we can apply UNF rule of the type system to obtain $\Gamma, \overline{S(\Delta)} \vdash_P t_1 = t_2 : bool$.

- M is a call.
  $generate(p(X_1, \ldots, X_n)) = (bool, \{X_1 : \sigma_1, \ldots, X_n : \sigma_n\}, Eq, Ineq \cup \{\alpha_1 \leq \tau_1, \ldots, \alpha_n \leq \tau_n\}, \Delta\prime)$, where we have $generate(p(Y_1, \ldots, Y_n) :- body.) = (bool, \Gamma, Eq, Ineq, \Delta)$, $\{Y_1 : \tau_1, \ldots, Y_n : \tau_n\} \in \Gamma$, and $\Delta\prime = \Delta \cup \{\sigma_1 = \alpha_1, \ldots, \sigma_n = \alpha_n\}$. For any S such that $S \models_{\Delta\prime} Eq, Ineq, \{\alpha_1 \leq \tau_1, \ldots, \alpha_n \leq \tau_n\}$, then, by lemma 11, we know that $S \models_{\Delta\prime} Eq, Ineq$ and $S \models_{\Delta\prime} \{\alpha_1 \leq \tau_1, \ldots, \alpha_n \leq \tau_n\}$. Since, $S \models_{\Delta\prime} Eq, Ineq$, by the induction hypothesis, $\Gamma, S(\Delta) \vdash_P p(Y_1, \ldots, Y_n) :- body. : bool$. Also, since $S \models \{\alpha_1 \leq \tau_1, \ldots, \alpha_n \leq \tau_n\}$, we know that $\forall i. S(\sigma_i) \sqsubseteq_{\Delta\prime} S(\tau_i)$. Therefore, we can derive by the rule CLL that $\Gamma \cup \{X_1 : \tau_1, \ldots, X_n : \tau_n\}, \overline{S(\Delta \cup \{\sigma_1 = \alpha_1, \ldots, \sigma_n = \alpha_n\})} \vdash_P p(X_1, \ldots, X_n) : bool$.

- M is a query.
  $generate(c_1, \ldots, c_n) = (bool, \Gamma, Eq, Ineq, \Delta)$, where $generate(c_i) = (bool, \Gamma_i, Eq_i, Ineq_i, \Delta_i)$, $(\Gamma, \Delta, Eq\prime) = \otimes((\Gamma_1, \ldots, \Gamma_n), (\Delta_1, \ldots, \Delta_n))$, $Eq = Eq_1 \cup \ldots \cup Eq_n \cup Eq\prime$, and $Ineq = Ineq_1 \cup \ldots \cup Ineq_n$. For any S such that $S \models_\Delta Eq, Ineq$, then, by lemma 11, $S \models_\Delta Eq_i$ and $S \models_\Delta Ineq_i$ for all i. As a result, $\Gamma_i, \overline{S(\Delta_i)} \vdash_P c_i : bool$ for all i. Since $S \models_\Delta Eq\prime$, by lemma 13, we can replace all Γ_i and Δ_i by Γ and Δ.

Therefore we have $\Gamma, \overline{S(\Delta)} \vdash_P c_i : bool$ for all i. We can then use rule CON of the type system to obtain $\Gamma, \overline{S(\Delta)} \vdash_P c_1, \ldots, c_n : bool$.

- M is a clause.
  $generate(p(X_1, \ldots, x_n) : -body.) = (bool, \Gamma, Eq, Ineq, \Delta)$, where we have $generate(b_i) = (bool, \Gamma_i, Eq_i, Ineq_i, \Delta_i)$, $(\Gamma, \Delta) = \oplus((\Gamma_1 \cup \ldots \cup \Gamma_n), (\Delta_1 \cup \ldots \cup \Delta_n))$, $Eq = Eq_1 \cup \ldots \cup Eq_n$, and $Ineq = Ineq_1 \cup \ldots \cup Ineq_n$. For any S such that $S \models_\Delta Eq, Ineq$, then, by lemma 11, $S \models_\Delta Eq_i, Ineq_i$ for all i. By the induction hypothesis, $\Gamma_i, \overline{S(\Delta_i)} \vdash_P b_i : bool$. Therefore we can then use rule CLS of the type system to obtain $\Gamma, \overline{S(\Delta)} \vdash_P p(X_1, \ldots, X_n) : -b_1; \ldots, b_n : bool$.

- M is a recursive clause.
  $generate(p(X_1, \ldots, X_n) : -body) = (bool, \Gamma, Eq, Ineq\prime, \Delta)$, where we have $generate(p(X_1, \ldots, X_n) : -body\prime) = (bool, \Gamma, Eq, Ineq, \Delta)$, such that $body\prime$ is $body$ after removing all recursive calls, and $Ineq\prime = Ineq \cup \{\overrightarrow{\sigma_1} \leq \overrightarrow{\tau}, \ldots, \overrightarrow{\sigma_k} \leq \overrightarrow{\tau}\}$, such that τ are the types for the variables in the head of the clause in Γ and σ_i are the types for the variables in each recursive call.
  For any S such that $S \models_\Delta Eq, Ineq\prime$, then by lemma 11 $S \models_\Delta Eq, Ineq$. By the induction hypothesis, $\Gamma, S(\Delta) \vdash_P p(X_1, \ldots, X_n) : -body\prime. : bool$. Since $S \models_\Delta \overrightarrow{\sigma_i} \leq \overrightarrow{\tau_i}, \overrightarrow{\tau} \leq \overrightarrow{\sigma_i}$, then $\overrightarrow{\sigma_i} \sqsubseteq_\Delta \overrightarrow{\tau_i}$ and $\overrightarrow{\sigma_i} \sqsubseteq_\Delta \overrightarrow{\tau_i}$, which means $\overrightarrow{\sigma_i} \equiv_\Delta \overrightarrow{\tau_i}$. Therefore, by rule RCLS in the type system, we obtain $\Gamma, S(\Delta) \vdash_P p(X_1, \ldots, X_n) : -body. : bool$.

□

We also proved the soundness of constraint solving, which basically shows that the solved form returned by our constraint solver for a set of constrains C satisfies C.

Theorem 12 - Soundness of Constraint Solving: Let Eq be a set of equality constraints, $Ineq$ a set of subtyping constraints, and Δ a set of type definitions. If $solve(Eq, Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$ then $S \models_\Delta Eq, Ineq$.

Proof: The proof will follow by induction of the number of steps until the algorithm is finished.

- Case 1. By the induction hypothesis, $(Eq, Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, such that $S \models_\Delta Eq, Ineq$. The same S also models $t \doteq t$, since $S(\tau) = S(\tau)$ for any S. Therefore $(\{\tau \doteq \tau\} \cup Eq, Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, such that $S \models_\Delta (\tau \doteq \tau), Eq, Ineq$.

- Case 2. By the induction hypothesis, $(\{\alpha \doteq \tau\} \cup Eq[\alpha \rightarrow \tau], Ineq[\alpha \rightarrow \tau], \overline{\Delta[\alpha \rightarrow \tau]}) \rightarrow^* (S, \Delta)$, such that $S \models_\Delta (\alpha \doteq t), Eq[\alpha \rightarrow \tau], Ineq$.

By the lemma 11 $S \models_\Delta \alpha \doteq \tau$, which means that, $S(\alpha) = S(\tau)$. Because of that, $S \models_\Delta (\alpha = \tau), Eq, Ineq$, because for every occurrence of α in Eq, it will be replaced by $S(\alpha) = S(\tau)$, so having α in place of τ in Eq or in $Ineq$ will not change anything.

- Case 3. By the induction hypothesis, $(\{\alpha \doteq \tau\} \cup, Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, such that $S \models_\Delta (\alpha \doteq \tau), Eq, Ineq$. By the lemma 11, $S \models_\Delta \alpha \doteq \tau$, which means that $S(\alpha) = S(\tau)$. Therefore $S \models_\Delta \tau \doteq \alpha$, and $S \models_\Delta \tau \doteq \alpha, Eq, Ineq$.

- Case 4. By the induction hypothesis, $(\{\tau_1 \doteq \tau_1\prime, \ldots, \tau_n \doteq \tau_n\prime\} \cup Eq, Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, where $S \models_\Delta \tau_1 \doteq \tau_1\prime, \ldots, \tau_n \doteq \tau_n\prime, Eq, Ineq$. By the lemma 11, $S \models_\Delta \tau_1 \doteq tau_1\prime, \ldots, \tau_n \doteq \tau_n\prime$, which means that $S(\tau_1) = S(\tau_1\prime), \ldots, S(\tau_n) = S(\tau_n\prime)$. Therefore $S(f(\tau_1, \ldots, \tau_n)) = S(f(\tau_1\prime, \ldots, \tau_n\prime))$, so $S \models_\Delta (f(\tau_1, \ldots, \tau_n) \doteq f(\tau_1\prime, \ldots, \tau_n\prime)), Eq, Ineq$.

- Case 5. The proof does not apply, since the output is not a pair of a substitution and a set of type definitions. and trivially $\tau \equiv_\Delta \tau$

- Case 6. By the induction hypothesis, $(Eq, Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, such that $S \models_\Delta Eq, Ineq$. The same S also models $t \leq t$, since $S(t) = S(t)$ for any S and for any $S(t)$, $S(t) \sqsubseteq_\Delta S(t)$, by Reflexivity. Therefore $(Eq, \{t \leq t\} \cup Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, such that $S \models_\Delta (t \leq t), Eq, Ineq$.

- Case 7. By the induction hypothesis, $(Eq, \{\tau_1 \leq \tau_1\prime, \ldots, \tau_n \leq \tau_n\prime\} \cup Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, such that $S \models_\Delta \tau_1 \leq \tau_1\prime, \ldots, \tau_n \leq \tau_n\prime, Eq, Ineq$. By the lemma 11, $S \models_\Delta \tau_1 \leq \tau_1\prime, \ldots, \tau_n \leq \tau_n\prime$, which means that $S(\tau_1) \sqsubseteq_\Delta S(\tau_1\prime), \ldots, S(\tau_n) \sqsubseteq_\Delta S(\tau_n\prime)$. Therefore, $S(f(\tau_1, \ldots, \tau_n)) \sqsubseteq_\Delta S(f(\tau_1\prime, \ldots, \tau_n\prime))$, and $S \models_\Delta (f(\tau_1, \ldots, \tau_n) \leq f(\tau_1\prime, \ldots, \tau_n\prime)), Eq, Ineq$.

- Case 8. By the induction hypothesis, $(Eq \cup Eq\prime, \{\alpha \leq \tau\} \cup Ineq, \Delta\prime) \rightarrow^* (S, \Delta\prime\prime)$, such that $S \models_\Delta Eq \cup Eq\prime, \{\alpha \leq \tau\} \cup Ineq$. By the lemma 11, $S \models_\Delta \alpha \leq t$, so $S(\alpha) \sqsubseteq_\Delta S(t)$. By lemma 14, we know that since $intersect(\tau_1, \tau_2, \emptyset, \Delta) = (\tau, Eq, \Delta\prime)$, then $\forall i. S(\tau) \sqsubseteq_\Delta S(\tau_i)$, by lemma 14 and the fact that $S \models_\Delta Eq$. This means that $\forall i. S(\alpha) \sqsubseteq_\Delta S(t_i)$. Therefore, $S \models_\Delta Eq, (\alpha \leq t_1), (\alpha \leq t_2), Ineq$.

- Case 9. By the induction hypothesis, $(\{\alpha \doteq \tau\} \cup Eq, Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, such that $S \models_\Delta (\alpha \doteq \tau), Eq, Ineq$. By the lemma 11, $S \models_\Delta \alpha \doteq \tau$, which means $S(\alpha) = S(\tau)$. As a consequence, $S(\alpha) \sqsubseteq_\Delta S(\tau)$. Therefore, $S \models_\Delta \alpha \leq \tau$, so $S \models_\Delta Eq, (\alpha \leq \tau), Ineq$.

- Case 10. By the induction hypothesis, $(Eq, \{\tau_1 \leq \tau, \ldots, \tau_n \leq \tau\} \cup Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, such that $S \models_\Delta Eq, (\tau_1 \leq \tau), \ldots, (\tau_n \leq \tau), Ineq$. By the lemma 11, $S \models_\Delta (\tau_1 \leq \tau), \ldots, (\tau_n \leq \tau)$, which means that $\forall i. S(\tau_i) \sqsubseteq_\Delta S(\tau)$. Then by the definition of $\sqsubseteq_\Delta$ we know that $S(\tau_1 + \ldots + \tau_n) \sqsubseteq_\Delta S(\tau)$. Therefore $S \models_\Delta Eq, (\tau_1 + \ldots + \tau_n \leq \tau), Ineq$.

- Case 11. By the induction hypothesis, $(Eq, Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, such that $S \models_\Delta Eq, Ineq$. Since σ and τ have been compared before, we can assume $\sigma \sqsubseteq_\Delta \tau$, since that is what is intended by the store. Therefore, it is true that $S \models_\Delta Eq, (\sigma \leq \tau), Ineq$.

- Case 12. By the induction hypothesis, $(Eq, \{\Delta(\sigma) \leq \tau\} \cup Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, such that $S \models_\Delta Eq, (\Delta(\sigma) \leq \tau), Ineq$. By the lemma 11, $S \models_\Delta \Delta(\sigma) \leq \tau$. Therefore, from the definition of the subtyping relation, using the left unfolding rule, we know that $S \models_\Delta \sigma \leq \tau$. Therefore $S \models_\Delta Eq, (\sigma \leq \tau), Ineq$.

- Case 13. By the induction hypothesis, $(\{\alpha = \tau_1 + \ldots + \tau_n\} \cup Eq, Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, such that $S \models_\Delta (\alpha = \tau_1 + \ldots + \tau_n), Eq, Ineq$. By the lemma 11, $S \models_\Delta \alpha = \tau_1 + \ldots + \tau_n$, which means $S(\alpha) = S(\tau_1 + \ldots + \tau_n)$. If they are equal, then $S(\tau_1 + \ldots + \tau_n) \sqsubseteq_\Delta S(\alpha)$, therefore it is obvious that, $\forall i. S(\tau_1) \sqsubseteq_\Delta S(\alpha)$. As a consequence, $S \models_\Delta (\tau_1 \leq \alpha), \ldots, (\tau_n \leq \alpha)$. Therefore $S \models_\Delta Eq, (\tau_1 \leq \alpha), \ldots, (\tau_n \leq \alpha), Ineq$.

- Case 14. By the induction hypothesis, $(Eq, \{\tau \leq \tau_i\} \cup Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, such that $S \models Eq, (\tau \leq \tau_i), Ineq$. By the lemma 11, $S \models_\Delta \tau \leq \tau_i$, which means $S(\tau) \sqsubseteq S(\tau_i)$. Now, by the definition of $\sqsubseteq_\Delta$, we have that $S(\tau) \sqsubseteq_\Delta S(\tau_1 + \cdots + \tau_i + \ldots \tau_n)$, which means $S \models_\Delta t \leq \tau_1 + \ldots + \tau_n$. Therefore $S \models_\Delta Eq, (\tau \leq \tau_1 + \ldots + \tau_n), Ineq$.

- Case 15. By the induction hypothesis, $(Eq, Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, such that $S \models_\Delta Eq, Ineq$. Since τ and σ have been compared before, we can assume $\tau \sqsubseteq_\Delta \sigma$, since that is what is intended by the store. Therefore, it is true that $S \models_\Delta Eq, (\tau \leq \sigma), Ineq$.

- Case 16. By the induction hypothesis, $(Eq, \{\tau \leq \sigma_{11} \leq \sigma_{13}\}, \{\sigma_4 = \alpha_4, \sigma_5 = \alpha_5, \sigma_7 = \alpha_7, \sigma_9 = \alpha_9, \sigma_{11} = \alpha_{11}, \sigma_{12} = \alpha_1 + \alpha_3, \sigma_{13} = \alpha_2 + \alpha_6, \sigma = int + float\})$

- Case 17. By Proposition 15 and Lemma 10, we know that Eq is in solved form, and, since it is in solved form, it can be interpreted as a substitution, such that $Eq \models_\Delta Eq$, so $Eq \models_\Delta Eq \cup \emptyset$.

- Case 18. The proof does not apply, since the output is not a pair of a substitution and a set of type definitions.

□

Finally, using the last two theorems we prove the soundness of the type inference algorithm. The soundness theorem states that if one applies the substitution corresponding to the solved form returned by the solver to the type obtained by the constraint generation function, we get a well-typed program.

Theorem 13 - Soundness of Type Inference: Given M, if $generate(M) = (\tau, \Gamma, Eq, Ineq, \Delta)$ and $solve(Eq, Ineq, \Delta) \rightarrow^* (S, \Delta\prime)$, then $\Gamma, S(\Delta\prime) \vdash_P M : S(\tau)$.

Proof: The proof follows from using theorems 11 and 12 directly. □

5.1.6 Arithmetic

We extend the type inference algorithm to deal with some of the traditional arithmetic built-ins, as described in Section 4.4. We extend our constraint generation algorithm to include the following cases:

- $generate(binOp(t_1, t_2)) = (\sigma, \Gamma, Eq, Ineq, \Delta \cup \{\sigma = int + float\})$
  where $generate(t_i) = (\tau_i, \Gamma_i, Eq_i, Ineq_i, \Delta_i)$,
  $(\Gamma, \Delta, Eq\prime) = \otimes\big((\Gamma_1, \Gamma_2), (\Delta_1, \Delta_2)\big)$,
  $Eq = Eq_1 \cup Eq_2 \cup Eq\prime$, and
  $Ineq = Ineq_1 \cup Ineq_2 \cup \{\tau_1 \leq int + float, \tau_2 \leq int + float\}$.

- $generate(unOp\ (t)) = (\tau, \Gamma, Eq, Ineq \cup \{\tau \sqsubseteq int + float\}, \Delta)$
  where $generate(t) = (\tau, \Gamma, Eq, Ineq, \Delta)$.

- $generate(is(t_1, t_2)) = (bool, \Gamma, Eq, Ineq, \Delta)$
  where $generate(t_i) = (\tau_i, \Gamma_i, Eq_i, Ineq_i, \Delta_i)$,
  $(\Gamma, \Delta, Eq\prime) = \otimes\big((\Gamma_1, \Gamma_2), (\Delta_1, \Delta_2)\big)$,
  $Eq = Eq_1 \cup Eq_2 \cup Eq\prime$, and
  $Ineq = Ineq_1 \cup Ineq_2 \cup \{\tau_1 \leq int + float, \tau_2 \leq int + float\}$.

- $generate(relOp(t_1, t_2)) = (bool, \Gamma, Eq, Ineq, \Delta)$
  where $generate(t_i) = (\tau_i, \Gamma_i, Eq_i, Ineq_i, \Delta_i)$,
  $(\Gamma, \Delta, Eq\prime) = \otimes\big((\Gamma_1, \Gamma_2), (\Delta_1, \Delta_2)\big)$,
  $Eq = Eq_1 \cup Eq_2 \cup Eq\prime$, and
  $Ineq = Ineq_1 \cup Ineq_2 \cup \{\tau_1 \leq int + float, \tau_2 \leq int + float\}$.

We will now illustrate with an example what the results for the constraint generation would be.

Example 45: Let *len* be a predicate that calculates the length of a list, defined as follows:

```
len([],0).
len([X|Xs],N) :- len(Xs,N1), N is N1 + 1.
```

The normalized predicate definition is:

```
len(L,S) :- L = [], S = 0;
       L = [X | Xs], S = N, N is M, M = N1 + 1, len(Xs,N1).
```

The exact trace of *generate* for the predicate *len* as defined previously is as follows:

$generate(len(L,S) : -L = [\,], S = 0; L = [X|Xs], S = N, N\ is\ M, M = N1 + 1, len(Xs, N1).) =$

$generate(len(L,S) : -L = [\,], S = 0; L = [X|Xs], S = N, N\ is\ M, M = N1 + 1.) =$

$generate(L = [\,], S = 0) =$

$generate(L = [\,]) = (bool, \{L : \sigma_1\}, \{\alpha_1 \doteq [\,]\}, \emptyset, \{\sigma_1 = \alpha_1\})$

$generate(S = 0) = (bool, \{S : \sigma_2\}, \{\alpha_2 \doteq int\}, \emptyset, \{\sigma_2 = \alpha_2\})$

$(bool, \{L : \sigma_1, S : \sigma_2\}, \{\alpha_1 = [\,], \alpha_2 = int\}, \emptyset, \{\sigma_1 = \alpha_1, \sigma_2 = \alpha_2\})$

$generate(L = [X|Xs], S = N, N\ is\ M, M = N1 + 1) =$

$generate(L = [X|Xs]) = (bool, \{L : \sigma_3, X : \sigma_4, Xs : \sigma_5\}, \{\alpha_3 \doteq [\alpha_4 \mid \alpha_5]\}, \emptyset, \{\sigma_3 = \alpha_3, \sigma_4 = \alpha_4, \sigma_5 = \alpha_5\})$

$generate(S = N) = (bool, \{S : \sigma_6\, N : \sigma_7\}, \{\alpha_6 \doteq \alpha_7\}, \emptyset, \{\sigma_6 = \alpha_6, \sigma_7 = \alpha_7\})$

$generate(N\ is\ M) = (bool, \{N : \sigma_8, M : \sigma_9\}, \emptyset, \{\alpha_8 \leq int + float, \alpha_9 \leq int + float\}, \{\sigma_8 = \alpha_8, \sigma_9 = \alpha_9\})$

$generate(M = N1 + 1) = (bool, \{M : \sigma_{10}, N1 : \sigma_{11}\}, \{\alpha 10 \doteq \sigma\}, \{\alpha_{11} \leq int + float\}, \{\sigma_{10} = \alpha_{10}, \sigma_{11} = \alpha_{11}, \sigma = int + float\})$

$(bool, \{L : \sigma_3, X : \sigma_4, Xs : \sigma_5, S : \sigma_6, N : \sigma_7, M : \sigma_9, N1 : \sigma_{11}\}, \{\alpha_3 \doteq [\alpha_4 \mid \alpha_5], \alpha_6 \doteq \alpha_7, \alpha 10 \doteq \sigma, \alpha_7 \doteq \alpha_8, \alpha_9 \doteq \alpha_{10}\}, \{\alpha_8 \leq int + float, \alpha_9 \leq int + float, \alpha_{11} \leq int + float\}, \{\sigma_3 = \alpha_3, \sigma_4 = \alpha_4, \sigma_5 = \alpha_5, \sigma_6 = \alpha_6, \sigma_7 = \alpha_7, \sigma_9 = \alpha_9, \sigma_{11} = \alpha_{11}, \sigma = int + float\})$

$(bool, \{L : \sigma_{12}, X : \sigma_4, Xs : \sigma_5, S : \sigma_{13}, N : \sigma_7, M : \sigma_9, N1 : \sigma_{11}\}, \{\alpha_1 \doteq [\,], \alpha_2 \doteq int, \alpha_3 \doteq [\alpha_4 \mid \alpha_5], \alpha_6 \doteq \alpha_7, \alpha 10 \doteq \sigma, \alpha_7 \doteq \alpha_8, \alpha_9 \doteq \alpha_{10}\}, \{\alpha_8 \leq int + float, \alpha_9 \leq int + float, \alpha_{11} \leq int + float\}, \{\sigma_4 = \alpha_4, \sigma_5 = \alpha_5, \sigma_7 = \alpha_7, \sigma_9 = \alpha_9, \sigma_{11} = \alpha_{11}, \sigma_{12} = \alpha_1 + \alpha_3, \sigma_{12} = \alpha_2 + \alpha_6, \sigma = int + float\})$

$(bool, \{L : \sigma_{12}, X : \sigma_4, Xs : \sigma_5, S : \sigma_{13}, N : \sigma_7, M : \sigma_9, N1 : \sigma_{11}\}, \{\alpha_1 \doteq [\,], \alpha_2 \doteq int, \alpha_3 \doteq [\alpha_4 \mid \alpha_5], \alpha_6 \doteq \alpha_7, \alpha 10 \doteq \sigma, \alpha_7 \doteq \alpha_8, \alpha_9 \doteq \alpha_{10}\}, \{\alpha_8 \leq int + float, \alpha_9 \leq int + float, \alpha_{11} \leq int + float, \sigma_5 \leq \sigma_{12}, \sigma_{12} \leq \sigma_5, \sigma_{13} \leq \sigma_{11}, \sigma_{11} \leq \sigma_{13}\}, \{\sigma_4 = \alpha_4, \sigma_5 = \alpha_5, \sigma_7 = \alpha_7, \sigma_9 = \alpha_9, \sigma_{11} = \alpha_{11}, \sigma_{12} = \alpha_1 + \alpha_3, \sigma_{13} = \alpha_2 + \alpha_6, \sigma = int + float\})$

Although the presentation becomes quite unreadable due to the large number of variables and type definitions, the resulting sets of equations can then be introduced into the solve algorithm.

For the sake of readability, we simply present some intermediate steps:

$$solve(\{\alpha_1 \doteq [\,], \alpha_2 \doteq int, \alpha_3 \doteq [\alpha_4 \mid \alpha_5], \alpha_6 \doteq \alpha_7, \alpha_{10} \doteq \sigma, \alpha_7 \doteq \alpha_8, \alpha_9 \doteq \alpha_{10}\}, \{\alpha_8 \leq int + float, \alpha_9 \leq int + float, \alpha_{11} \leq int + float, \sigma_5 \leq \sigma_1, \sigma_1 \leq \sigma_5, \sigma_2 \leq \sigma_{11}, \sigma_{11} \leq \sigma_2\}, \{\sigma_4 = \alpha_4, \sigma_5 = \alpha_5, \sigma_7 = \alpha_7, \sigma_9 = \alpha_9, \sigma_{11} = \alpha_{11}, \sigma_{12} = \alpha_1 + \alpha_3, \sigma_{13} = \alpha_2 + \alpha_6\})) \rightarrow^*$$

$$solve(\{\alpha_1 \doteq [\,], \alpha_2 \doteq int, \alpha_3 \doteq [\alpha_4 \mid \alpha_5], \alpha_6 \doteq \alpha_8, \alpha_{10} \doteq \sigma, \alpha_7 \doteq \alpha_8, \alpha_9 \doteq \sigma\}, \{\alpha_8 \leq int + float, \sigma \leq int + float, \alpha_{11} \leq int + float, \sigma_5 \leq \sigma_1, \sigma_1 \leq \sigma_5, \sigma_2 \leq \sigma_{11}, \sigma_{11} \leq \sigma_2\}, \{\sigma_4 = \alpha_4, \sigma_5 = \alpha_5, \sigma_7 = \alpha_8, \sigma_9 = int + float, \sigma_{11} = \alpha_{11}, \sigma_{12} = [\,] + [\alpha_4 \mid \alpha_5], \sigma_{13} = int + \alpha_8\})) \rightarrow^*$$

$$solve(\{\alpha_1 \doteq [\,], \alpha_2 \doteq int, \alpha_3 \doteq [\alpha_4 \mid \alpha_5], \alpha_6 \doteq \alpha_8, \alpha_{10} \doteq \sigma, \alpha_7 \doteq \alpha_8, \alpha_9 \doteq \sigma\}, \emptyset, \{\sigma_4 = \alpha_4, \sigma_{12} = [\,] + [\alpha_4 \mid \sigma_{12}], \sigma_{13} = int + float\}))$$

The interesting types, corresponding to the argument of *len* predicate, are σ_{12} and σ_{13}, respectively. Therefore the type inferred for the predicate is $len :: \sigma_{12} \times \sigma_{13} \rightarrow bool$, where $\sigma_{12} = [\,] + [\alpha_4 \mid \sigma_{12}]$ and $\sigma_{13} = int + float$.

In the next chapter we discuss the introduction of data structures declared by the programmer and explain the changes necessary to the type inference algorithm in order to obtain reasonable results.

5.1.7 Discussion

In this thesis, we recognize the limitations of static typing as discussed in Section 4.5. The limitations discussed there are necessary in order to have decidable type inference. In this chapter, we presented a type inference algorithm that outputs types that may be more general than the actual types of the interpretation function intended by the programmer. This interpretation may itself have more tuples accepted (either *true* or $false$) than intended as well.

As discussed in Section 4.5, we are inferring over approximations of programs. As a result, the types inferred for a program may be very general and uninformative, in fact even correspond to the entire Herbrand universe, in some cases. In order to infer types that are closer to the programmer's intention, in the next chapter we present an algorithm that *closes* the types we get from type inference, based on some fundamental principles, such that, for some predicates, the resulting types are closer to the programmer's intention.

We also allow for the declaration of algebraic data types, that can be used during type inference, in order to improve the result obtained from it.

One other limitation of the work presented in the previous two chapters is that we deal with recursive predicates, but not with mutually recursive predicates. First of all, we argue that the class of programs we already cover, excluding mutually recursive predicates, is still significant. Secondly, if we consider every strongly connected component (SCC) of the dependency graph of the program to be a mutually recursive definition of several predicates, we could extend this work in two ways:

- We could define the semantics for each SCC; add a rule in the type system for SCCs; and define an extra case for type inference where we infer the types for each SCC.

- In alternative, we could compile logic programs to an intermediate language where mutually recursive definitions are defined in a single syntactic element, such as a *letrec* in functional programming languages, and then do the steps described previously, *i.e.*, define the semantics for that syntactic element; and add a rule in the type system for it; define an extra case for type inference of types of that syntactic element.

Chapter 6

Closed Types

In the previous chapter, we described a type inference algorithm to infer types for an untyped program, and proved that the types inferred can be derived in the type system presented in Chapter 4.

However, as discussed in Section 4.5, the types resulting from type inference are often too general to be useful or informative. This is not due to limitation in the type inference algorithm itself, but by virtue of how logic programs are written, and how logic variables are used.

In this chapter, we present a class of types called *closed types*, that correspond to a subset of the types described by our type language. Types that are inferred from our type inference algorithm can be closed, but normally they are not, in which case we will call them *open types*. Closed types are much more informative and often closer to the programmer's intention [25]. The restriction on the type language in order to obtain closed types is the same that functional logic programming uses for the types allowed.

We also present an algorithm that given some open types inferred by the type inference algorithm, returns closed types.

In addition to that, we present an optional process for declaring data types, in order to use them during inference. We show the effects of declaring simple data types, such as lists or trees, on the resulting types returned by the algorithm. We force the use of closure when data types are declared so that the inferred types are clearer.

The content of this chapter was partially presented in [1] and in [3].

6.1 Motivation

Types are sets of objects that share some characteristic and are used to either document a program, help in the development of programs, or describe some properties of programs. Therefore, having types that correspond to the whole Herbrand Universe is not very helpful. Unfortunately, as said in the previous section, types inferred by our type inference algorithm sometimes are exactly like that. They are often crude, uninformative, and not the intended type for the program.

This problem is known since the early beginning of research on types for logic programming. Zobel [6] said about 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*" [7].

Of course one may say that the definition of intended interpretation is not known in advance, but we argue that the intended interpretation of a program is the interpretation any programmer would have when programming the same specification in a typed programming language. This view is implicit in previous work by Naish [78] and explicit in type systems which make it mandatory to explicitly declare type definitions of program functors [11,12]. Let us take as an example the *append* predicate as defined in several libraries.

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

The type inferred by our type inference algorithm for (a normalized version of) this predicate is $\sigma_1 \times \sigma_2 \times \sigma_3 \rightarrow bool$, where:

$\sigma_1 = [\,] + [\alpha \mid \sigma_1]$,
$\sigma_2 = \beta$,
$\sigma_3 = \beta + [\alpha \mid \sigma_3]$.

This type accepts more terms than the "intended" interpretation of *append* which is relating three lists as arguments, for example calls such as $append([\,], 1, 1)$ are accepted by the predicate. If we take a look at the types in [25], specifications for programs always describe some behavior desired by the programmer and some structured set of terms for which that behavior should be satisfied.

In fact, Naish argues that types need to be part of the specification, in order for the specification to be correct and complete. An example of a specification presented in [25] for *append* is shown below:

> *Specification: append(A,B,C) is true iff C is the list B concatenated onto the list A.*

Obviously if this is the specification we want for our predicate, then the type inferred above is not the one "intended", but instead it would be $\sigma_1 \times \sigma_1 \times \sigma_1 \rightarrow bool$, where:

$\sigma_1 = [\,] + [\alpha \mid \sigma_1]$.

Note that this type does not over-approximate the predicate model, but, somehow, it corresponds to the most general accepted intended interpretation of the predicate *append* as a program which appends two lists of elements of some type α. This is also a well-typing of *append* by the most strict systems [11, 12] based on an Hindley-Milner style of type system definition.

The motivation behind *closed types* is to have a restriction on the type language such that types inferred for predicates are closer to the programmer's intention, without a requirement for any information given by the programmer. In the next section, we explain the fundamental principles on which we base our definition for closed types.

6.2 Principles

Having union types as part of your type syntax, as in this work, can lead to types that have useless information, for instance, if we have the type $\sigma = [\,] + [int \mid \sigma] + \alpha$, even though we can see some structure that might be interpreted as a list, any term can be typed with this type, due to the presence of type variable α as a summand on the definition.

The concept of algebraic data types is relevant here, where all summands in a sum must be disjoint. This is in fact a restriction in functional programming languages, such as in data declarations in Haskell. So we introduce the following principle:

Principle 1: Types should be strictly smaller than the set of all possible terms.

Example 46: Let p be defined as:

```
p(1).
p(a).
p(X).
```

The type for the argument of p would be inferred as $\sigma = int + atom + \alpha$. This is probably not the intended type for p.

A second motivation stems from deductive databases [79, 80]. Consider the following Datalog program:

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

e(1).
```

This program is not allowed in Datalog 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.

In fact, we can have the following program:

```
id(X,X).
```

Which is the identity predicate, or equality predicate, and we can reason about what is so different about the two.

The types for i would be $\sigma_1 \times \sigma_2 \to bool$, where $\sigma_1 = int$, and $\sigma_2 = \beta$, while the types for id would be $\sigma_3 \times \sigma_3 \to bool$, where $\sigma_3 = \gamma$. The difference is that there is some restriction on what is accepted by id. If we think about the type for the predicate id we have $\forall\gamma.\gamma \times \gamma \to bool$, so now the semantics of this type are all functions that accepted pairs of objects of the same type. Therefore there is some type restriction.

In order to distinguish from the cases like id and i, we introduce the following principle:

Principle 2: All type variables should be constrained.

Example 47: Let us consider the following definition for predicate p:

```
p(1, X).
```

The type we get for the second argument is $\sigma_2 = \alpha$, where α is a type variable that appears only once in the types for the predicate. This makes the type for the second argument open to be anything.

Even if we are willing to accept this principle, there are still some difficulties. Do we want to exclude the predicates that do not follow these principles? Or could it be that there is a type for the predicate that follows the principles, and the general way that logic programs are written on meant we could not infer it? If so, is there an operation we can perform to transform the open types into closed types, such that the principles are followed?

We believe some operation exists and it must be based on the following principle:

Principle 3: Types are based on self-contained definitions.

This means that all the information we need to *close* the types resulting from type inference is in the type themselves.

In the rest of this chapter we present the definition of *closed types* and the *closure operation*, that given some set of types that are possibly open, transforms them into closed types, if possible. We also present data type declarations and a comparison of the results we get on several predicates when performing type inference without any extra step, when using closure, and when using data type declarations for relevant data structures.

6.3 Closed Types

Closed types are, intuitively, types that are constrained in some way. If the definition for a type symbol is open, then it may type any term, including terms that use function symbols not present in the program. This is exactly what we want to avoid with closed types. There are two main issues that motivate closed types: unconstrained type variables and closed composite types. We will hereon name *composite types* to union types that have more than one summand.

Definition 35 - Unconstrained Type Variable: A type variable α is unconstrained with respect to a set of type definitions Δ, which we denote by $unconstrained(\alpha, \Delta)$, if and only if it occurs exactly once in Δ.

These type variables are inferred for logic variables in the program that are used only once.

Example 48: Let $\Delta_1 = \{\sigma_1 = \alpha, \sigma_2 = \beta\}$. In this case, both α and β are unconstrained type variables.
Let $\Delta_2 = \{\sigma_3 = [\,] + [\gamma \mid \sigma_3]\}$. In this case, γ is an unconstrained type variable.

As we can see in the previous example, if there is an unconstrained type variable it does not mean that the type is open, as σ_3 is actually not an open type. In fact, unconstrained type variables are only a problem in certain cases, so we need to distinguish those cases in the definition of closed types.

Definition 36 - Closed Composite Type: A composite type $\widetilde{\tau}$ is a closed composite type, notation $closedComposite(\widetilde{\tau})$, if and only if it has no type variables as summands.

Example 49: Let $\Delta = \{\sigma_1 = int, \sigma_2 = int + float, \sigma_3 = f(int) + \alpha\}$. In this case, σ_1 is not a composite type, σ_2 is a closed composite type, and σ_3

is an open composite type.

The definition for closed types uses these two previous auxiliary definitions.

Definition 37 - Closed Types: A type symbol σ is closed with respect to a set of type definitions Δ, notation $closed(\sigma, \Delta)$, if and only if the following holds:

$$closed(\sigma, \Delta) = \begin{cases} closedComposite(\Delta(\sigma)) & \text{if } \Delta(\sigma) \text{ is a composite type} \\ \neg unconstrained(\Delta(\sigma), T) & \text{if } \Delta(\sigma) \text{ is a type variable} \\ True & \text{otherwise} \end{cases}$$

Informally, a closed type is either a closed composite type or, if it is a type term, then that type term cannot be an unconstrained type variable. The first case of this definition avoids open data types while the second case avoids unconstrained type variables which could be instantiated by the whole Herbrand universe.

Example 50: $\sigma_1 = \alpha + f(\beta)$ is not a closed type with respect to any set of type definitions Δ, since the right-hand side of the definition is an open composite type.
$\tau_2 = int + f(\alpha)$ is a closed type with respect to any set Δ, since it does not have variables as *summands*.

6.4 Closure Operation

Using our type inference algorithm which infers regular (open) types for a given logic program, we now define a closure operation which, given a set of regular types, closes them, if possible. This operation follows Principle 3, *i.e.*, types are based on self-contained definitions.

We first define the *proper type domain* of a type symbol σ with respect to a set of type definitions Δ as the set of non-variable summands in the definition for σ itself and in the type definitions of all the type symbols that share at least one type constructor with σ.

A precise definition of the proper type domain of a type τ with respect to set of type definitions Δ, notation $properType(\tau, \Delta)$, is that it is the set of types computed by the function in pseudo-code, in Figure 6.1.

We then define the *proper variable domain* of a type variable α with respect to a set of type definitions Δ as the union of all proper domains for the types whose definition includes α as a summand.

```
function properType(σ, Δ)
P = the set of non-variable summands of σ;
for each type function symbol f ∈ σ do
    for σ′ ∈ T do
        if f ∈ σ′ then
            s = set of non-variable summands of σ′;
            P = P ∪ S;
        end
    end
end
    return P
end function
```

FIGURE 6.1: Proper Type Domain function

A precise definition of the proper variable domain of a type variable α with respect to a set of type definitions Δ, notation $properVar(\alpha, \Delta)$, is that it is the set of types computed by the function in pseudo-code, in Figure 6.2.

```
function properVar(α, Δ)
V = ∅;
for each σ ∈ Δ do
    if α ∈ Δ(σ) then
        P = properType(σ, Δ);
        D is the set of type terms in P that contain α;
        V = V ∪ P \ D;
    end
end
    return V
end function
```

FIGURE 6.2: Proper Variable Domain function

Note that the definitions for the proper type domain and the proper variable domain are based on principles 2 and 3, since they are self-contained in the sense that the information used is defined in the lexical components of the program and closure will forbid unconstrained type variables.

Using these algorithms we can now define a closure operation given a set of type definitions T, notation $closure(T)$, as the set of types computed by the function, in pseudo-code, in Figure 6.3.

The auxiliary function *makeSum* transforms a set of type terms into a union type of those type terms. Also note that if the proper variable domain of some type variable in a given set of type definitions is empty, this indicates that we have no information to use for the closure. In these cases the variable is either just "ignored" if the type symbol where it occurs has some other type terms or the algorithm halts with failure because there is no way to obtain a closed type for this predicate.

```
function closure(Δ, Γ)
while some σ ∈ Δ is open do
    for each α that occurs only once in Δ do
        if σ = α ∈ Δ then
            fail
        else
            we have (σ = α + X) in Δ; Δ = Δ[α ↦ X]‾;
        end
    end
    for each α ∈ Δ(σ), for some σ do
        P = properVar(α, Δ);
        if P = ∅ then
            if σ = α ∈ Δ then
                fail
            else
                we have (σ = α + X) in Δ; Δ = Δ[α ↦ X]‾;
            end
        else
            S = makeSum(P);
            Δ = Δ[α ↦ S]‾;
        end
    end
end
    return Δ
end function
```

FIGURE 6.3: Closure operation

Informally, the algorithm to compute the closure of a set of type definitions T consists of the following steps:

- get all open types $\sigma \in \Delta$ with respect to $\Delta \setminus \{\sigma\}$;
- for each variable α that makes these types open, get their proper variable domain;
- substitute every occurrence of those variables as summands in the definitions in Δ by their proper variable domain, and simplify.

Theorem 14 - Closure Termination: Given a set of type definitions Δ, $closure(\Delta)$ always terminates.

A draft of the proof for this theorem is to notice that every variable is substituted by a union type composed of type terms constructed from the symbols present in Δ. Since the set of symbols is finite, so are the possible union types. Since at every step we reduce the number of type variables by at least one, we eventually either exhaust all possible union types, eliminate every type variable, or halt with failure. Therefore the algorithm terminates.

Example 51: Let the predicate *gcd* calculate the greatest common divisor between two integers and be defined as follows:

```
gcd(X,0,X).
gcd(X,J,K) :- R is (X mod J), gcd(J,R,K).
```

If we assume that the type inference algorithm attributes *int* to all variables involved in an *is* goal, then the types inferred for this predicate is $\sigma_1 \times \sigma_2 \times \sigma_1 \rightarrow bool$, where:

$\sigma_1 = \alpha + int$
$\sigma_2 = int$
$\sigma_3 = \alpha + int$

Applying the closure to the set $\Delta = \{\sigma_1 = \alpha + int, \sigma_2 = int, \sigma_3 = \alpha + int\}$, we get the set of closed types $\{\sigma_1 = int, \sigma_2 = int, \sigma_3 = int\}$. Following execution step-by-step:

- types σ_1 and σ_3 will be detected as open;
- the proper variable domain of α will be calculated;
- the proper type domain of σ_1 and σ_3 will be calculated;
- the result for the proper variable domain will be obtained (int);
- after substituting the variable for the proper domain, the resulting set of type definitions will be $\{\sigma_1 = int, \sigma_2 = int, \sigma_3 = int\}$.

A final note about closed types has to do with their relation to untyped programs. Note that closed types filter the set of admissible queries to a program as the ones which are typed by the closed type. This has an immediate consequence which is that we may have a ground query for which the answer in an untyped version is "yes" and the answer in the same program but typed by a closed type is *wrong*. The example on Section 6.1 for the append predicate shows this: the query $append([], 1, 1)$ has answer "yes" in the untyped version of append and *wrong* in the version typed by the closed type in the example.

The opposite does not happen, *i.e.*, if the answer to a query to a predicate typed by a closed type is "yes", then the answer to the same query to the untyped version of the predicate still is "yes". This holds because closed types are instances of an (open) type for the predicate which we assume over-approximates the program semantics.

6.4.1 Soundness

One important property of the closure operation is that the resulting types still type the program, in the sense that we can derive the types in the type system. We will prove this property by showing types resulting from closure are instances of the original types, and then, by Lemma 7, have a derivation in the type system.

Lemma 16 - Closed Types as Instances: Let Δ be a set of type definitions. If $closure(\Delta) = \Delta\prime$, then there is a substitution S such that $\overline{S(\Delta)} = \Delta\prime$.

Proof: Suppose there is some type variable α that occurs only once in Δ. Then either we have $(\sigma = \alpha) \in \Delta$ and we fail, so the theorem does not apply, or we get $\Delta = \overline{\Delta[\alpha \mapsto X]}$. In this case $S = [\alpha \mapsto X]$ is the substitution applied to Δ such that $\overline{S(\Delta)} = \Delta\prime$. Suppose instead we have there is some type variable α that occurs more than once, but its proper domain is empty. Then, either we have $(\sigma = \alpha) \in \Delta$ we fail, so the theorem does not apply, or we get $\Delta = \overline{\Delta[\alpha \mapsto X]}$. In this case $S = [\alpha \mapsto X]$ is the substitution applied to Δ such that $\overline{S(\Delta)} = \Delta\prime$. Suppose there is a type variable that occurs more than once, but its proper domain is P, corresponding to the union type $\widetilde{\tau}$. Then in the end we get $\Delta\prime = \overline{(S(\Delta))}$, where $S = [\alpha \mapsto \widetilde{\tau}]$. Now note that the algorithm is iterative and one of the previous options applies at every step, until no type is open. Also note that none of the steps introduce any new type variable. Since the algorithm always terminates, in the end, let the substitutions $S_1, \ldots, S_n$ be the substitutions applied at every step. $S = S_1 \circ \ldots \circ S_n$ will be such that $closure(\Delta) = \Delta\prime$ and $S(\Delta) = \Delta\prime$. □

Theorem 15 - Closure Soundness: Let Δ be a set of type definitions, Γ be a context, P a program, M a clause, a query, or a term, and τ a type. Suppose we have $closure(\Delta) = \Delta\prime$, and there is a derivation $\Gamma, \Delta \vdash_P M : \tau$. Then, there is a derivation $\Gamma, \Delta\prime \vdash_P M : \tau\prime$ and $\tau \preceq \tau\prime$.

Proof: We know that if $closure(\Delta) = \Delta\prime$, then, there is a substitution S, such that $\overline{S(\Delta)} = \Delta\prime$, by lemma 16. Also, by lemma 7, we know that there is a derivation $\Gamma, \overline{S(\Delta)} \vdash_P M : S(\tau)$. Since $S(\tau) \preceq \tau$, and $\overline{S(\Delta)} = \Delta\prime$, then we can replace it in the derivation to get $\Gamma, \Delta\prime \vdash_P M : \tau\prime$, where $\tau\prime \preceq \tau$, which is what we wanted to prove. □

6.5 Examples

In this section we show some practical examples of applications for closed types, where they become useful, unlike their open counterpart. We also present some results for type inference for some predicates using the closure operation post-inference.

6.5.1 Bug Detection

Closed types, being less permissive that open types, may detect more bugs during program development. This is a pragmatic motivation for the use of closed types. Let us now present an illustrating example.

Example 52: Let *max* be a predicate that finds the largest number of a list.

```
max([ ], Max, Max).
max([H|L], Max0, Max) :- Max0 < H, max(L, H, Max).
max([H|L], Max0, Max) :- max(L, Max0, Max).
```

The type inferred by our type inference algorithm is $\sigma_1 \times \sigma_2 \times \sigma_3 \rightarrow bool$, where:

$\sigma_1 = [\,] + [\sigma_4|\sigma_1]$
$\sigma_2 = \alpha + int + float + \beta$
$\sigma_3 = \alpha + \gamma + \eta$
$\sigma_4 = int + float + \delta$

Clearly, types σ_2, σ_3, and σ_4 are open. The closed types returned by our closure operation are the following:

$\sigma_1 = [\,] + [\sigma_2|\sigma_1]$
$\sigma_2 = int + float$

with the type for the predicate $\sigma_1 \times \sigma_2 \times \sigma_2 \rightarrow bool$.

One common bug when defining this predicate can be defining the first clause as:

```
max([ ], Max, M).
```

In this case closure fails, since σ_3 would be defined as a union type of three type variables that occur nowhere else, so they have an empty proper variable domain, and therefore that type becomes empty and thus the closure fails.

This is an example of how closed types, being less permissive, avoid the debugging process at run-time, by catching more bugs at compile time.

6.5.2 Datatype-centric Programming

Programs in functional programming languages (such as Haskell and ML) and in imperative and object-oriented languages (such as C and Java) are often datatype-centric, in the sense that they are based on and make an intensive use of algebraic data types. Usually it is considered that the same happens in Prolog, using terms as a notation for data type definitions. We argue that this is not always the case due to the use of unconstrained logical variables. In these pathological cases the types we would get for programs are open types. Closed types play an important role on the use of a truly datatype-centric style of programming in logic programming. Let us consider the following example:

Example 53: Let *flatten* be the standard Prolog 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 type inferred would be $\sigma_1 \times \sigma_2 \rightarrow bool$, where:

$\sigma_1 = [\,] + [\sigma_1 \mid \sigma_1] + \alpha$
$\sigma_2 = [\,] + [\alpha \mid \sigma_2]$

Note that the type for the first argument is open. The problem here is that in the implicit data type definition of the first argument of the predicate, includes single elements of lists (here processed by the third clause of the predicate definition) and those do not have an associated constructor which distinguishes them from any other terms, such as lists. Moreover, closure of these types would wield results that are not the ones intended:

$\sigma_1 = [\,] + [\sigma_1 \mid \sigma_1]$
$\sigma_2 = [\,] + [\sigma_1 \mid \sigma_2]$

The proper variable domain of α is $\{[\,], [\sigma_1 \mid \sigma_1]\}$, which is σ_1. Note that this does not include the term $[\alpha \mid \sigma_2]$, since α occurs in it.

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 type $\sigma_3 \times \sigma_4 \rightarrow bool$ is inferred for the predicate, where:

$\sigma_1 = [\,] + [\sigma_1 \mid \sigma_1] + elem(\alpha)$
$\sigma_2 = [\,] + [elem(\alpha) \mid \sigma_2]$

Note that the *elem* functor in this predicate definition is playing the role of a type constructor which identifies single elements in a data type definition for nested lists (such as in a data declaration in Haskell). The resulting types are now closed.

We are not advocating this second style of programming in Prolog, but we argue that if one wants to have safer programs, in the sense that bugs and errors are easier to catch at compile time, either we declare types, such as in Curry or Mercury, or we infer types with an extra closure operation after type inference which will make those types truly denote datatype definitions. If this is the case, then a more datatype-centric style of programming, with functors as type constructors in every case of an implicit datatype definition will avoid the extra closure operation in many cases, making the initial inferred types already closed.

6.6 Data Type Declarations

We avoid type declarations, and by type declarations we mean the programmer declaring the type for every function symbol (including constants), and predicate symbol, because from our experience as programmers and as part of the logic programming community, the benefits of having types are not enough to compensate for the extra work.

In fact, in the functional programming community, the extra work required for having types is considerably decreased by automatic type inference and the good results that are wielded from it. The main difference is that programmers declare the algebraic data types that will be used in the program and then type signatures for functions are inferred automatically.

We argue that allowing for the introduction of data type definitions in logic programming in a similar way would require much less extra work from the programmer and still yield very interesting results. We decided to add the possibility of declaring data types, through the introduction of data type definitions of the form:

$$:-\texttt{type}\ type\_symbol(type\_vars) = type\_term_1 + \ldots + type\_term_n.$$

One example of such a declaration for trees is:

```
:- type tree(X) = empty + node(X, tree(X), tree(X)).
```

One restriction of such declarations is the assumption that if there is a data type declared, then every constant of function symbol that starts a summand in the definition is unique to that data type. This greatly improves the results from type inference. Note that there is a similar restriction on data declarations in functional programming languages.

We basically assume that every data type declaration corresponds to an *intended* new domain in the semantics, and every term that can be built from the data type definition belongs to that domain. If the data type is polymorphic, then there is a domain for every instance.

For these type declarations to be correctly inferred we need to make a few changes in the algorithm. First of all, we will interpret a data type declaration as in fact being a declaration for the base types of the constants and function symbols used in it.

So continuing the example above, the data type declaration can be interpreted as:
$basetype(empty) = tree(\alpha)$, and $basetype(node) = \alpha \times tree(\alpha) \times tree(\alpha) \to tree(\alpha)$.

So on the constraint generation algorithm, in fact the constant case remains the same, with the minor change that a constant can be typed by a polymorphic type. But we change the complex term cases as follows:

- $generate(f(t_1, \ldots, t_n)) = (\sigma, \Gamma, Eq, \emptyset, \Delta)$, f is a function symbol,
  where $basetype(f) = \tau\prime_1 \times \ldots \times \tau\prime_n \to \sigma$,
  $generate(t_i) = (\tau_i, \Gamma_i, Eq_i, \emptyset, \Delta_i)$,
  $(\Gamma, \Delta, Eq\prime) = \otimes\big((\Gamma_1, \ldots, \Gamma_n), (\Delta_1, \ldots, \Delta_n)\big)$,
  and $Eq = Eq_1 \cup \ldots \cup Eq_n \cup Eq\prime \cup \{\tau_1 \doteq \tau\prime_1, \ldots, \tau_n \doteq \tau\prime_n\}$.

And in the constraint solving algorithm, the names of data types are treated as if they were function symbols, so we assume the alphabets are disjoint.

The proof of soundness for the constraint generation algorithm still holds. We are assuming $type(f) = basetype(f)$, and for any S such that $S \models_\Delta \{\tau_1 \doteq \tau\prime_1, \ldots, \tau_n \doteq \tau\prime_n\}$, then $S(\tau\prime_i) = S(\tau_i) \preceq \tau_i$, for all $i = 1, \ldots, n$. Therefore we know that $S(\tau\prime_1) \times \cdots \times S(\tau\prime_n) \to S(\sigma) \preceq \tau_1 \times \cdots \times \tau_n \to \sigma$. Since $S \models_\Delta Eq_i$, by the induction hypothesis we know that $\Gamma_i, S(\Delta_i) \vdash_P t_i : \tau\prime\prime_i$, where $\tau\prime\prime_i \equiv_\Delta \tau_i$. Since $S \models_\Delta Eq\prime$, then we can replace Γ_i and Δ_i by Γ and Δ, respectively. Therefore we can use rule CPL to derive that $\Gamma, S(\Delta) \vdash_P f(t_1, \ldots, t_n) : S(\sigma)$.

6.7 Type Inference Revisited

In this section we show the results of type inference for some example predicates. We also show the results of closing the inferred types and the results yielded by using data type definitions for the relevant data structures. Note that declaring data type definitions implies using closure.

For simplicity of presentation we will replace the type symbols σ_i for the type of predicate arguments by the name of the predicate and the argument position. If any other types are presented, they will be represented by some lower case letter. Type variables with be represented by upper case letters.

Example 54: Let *append* be defined as usual. The type inferred for *append* is $append1 \times append2 \times append3 \rightarrow bool$, where:

```
append1 = [~] + [A~|~append1],
append2 = B,
append3 = B + [A~|~append3].
```

The types inferred if we used the data type declaration for lists $list(X) = [\,] + [X \mid list(X)]$, without using closure:

```
append1 = list(t),
append2 = B,
append3 = B + list(A),
t = A + G.
```

`t` results from the fact that lists are used both in the first and second clause in the first argument of the predicate. Also, the second argument never uses any function symbol that would indicate is it a list. But the type inferred for *append* using the same data type definition and closure is:

```
append1 = list(append1),
append2 = list(append2),
append3 = list(append3).
```

In the implementation of the type inference algorithm, we have some flags that can be turned on or off.

- *closure* (default: *off*) - when this flag is turned on, the closure operation is applied as a post-processing step in the algorithm;
- *list* (default: *off*) - this flag adds the data type declaration for polymorphic lists to the program when turned on, and also turns on the closure flag.

We will now show the results of the type inference for some programs.

Example 55: Let us consider the predicate *concat*, which flattens a list of lists, where *app* is the *append* predicate:

```
concat(X1,X2) :- X1 = [], X2 = [];
      X1 = [X|Xs], X2 = List, concat(Xs,NXs), app(X,NXs,List).

app(A,B,C) :- A=[], B=D, C=D;
      app(E,F,G), E=H, F=I, G=J, A=[K|H], B=I, C=[K|J].
```

The types inferred with all the flags *off* correspond to types inferred in previous type inference algorithms which view types as an approximation of the success set of the program:

```
concat :: concat1 x concat2 -> bool
concat1 = [] + [ t | concat1 ]
concat2 = C + [] + [ B | concat2 ]
t = [] + [ B | t ]

app :: app1 x app2 x app3 -> bool
app1 = [] + [ A | app1 ]
app2 = B
app3 = B + [ A | app3 ]
```

Now the types inferred when turning on the closure flag are:

```
concat :: concat1 x concat2 -> bool
concat1 = [] + [ concat2 | concat1 ]
concat2 = [] + [ B | concat2 ]

app :: app1 x app2 x app3 -> bool
app1 = [] + [ A | app1 ]
app2 = [] + [ A | app2 ]
app3 = [] + [ A | app3 ]
```

Note that these types are not inferred by any previous type inference algorithm for logic programming so far, and they are a step towards the automatic inference of types for programs used in a specific context, more precisely, a context which corresponds to how it would be used in a programming language with data type declarations, such as Curry [81] or Haskell.

We also present the types resulting from type inference when declaring a data type definition for lists (which is the same as turning the list flag on).

```
concat :: concat1 x concat2 -> bool
concat1 = list(list(B))
```

```
concat2 = list(B)

app :: app1 x app2 x app3 -> bool
app1 = list(A)
app2 = list(A)
app3 = list(A)

list(X) = [] + [X | list(X)]
```

Example 56: Let *rev* be the reverse list predicate, defined using the *append* definition used in the previous example:

```
rev(A, B) :- A = [], B = [] ;
        rev(C, D), app(D, E, F), E = [G], A = [G|C], B = F.
```

The types inferred by the algorithm are:

```
rev :: rev1 x rev2 -> bool
rev1 = [] + [ A | rev1 ]
rev2 = [] + [ t | rev2 ]
t = B + A
```

If the closure flag is turned on, we perform closure on these types. The resulting types are:

```
rev :: rev1 x rev2 -> bool
rev1 = [] + [ A | rev1 ]
rev2 = [] + [ A | rev2 ]
```

If we turn on the list flag, which declares the data type for polymorphic lists, the type inference algorithm outputs the same types that would be inferred in Curry or Haskell with pre-defined built-in lists:

```
rev :: rev1 x rev2
rev1 = list(A)
rev2 = list(A)

list(X) = [] + [X | list(X)]
```

We now show an example of the minimum of a tree to illustrate how different data structures can be dealt with. We are assuming the extension to the algorithm described in Section 4.4, where the type for all variables used in arithmetic expression or predicates is $int + float$.

Example 57: Let *tree_minimum* be the predicate defined as follows:

```
tree_min(A,B) :- A = empty, B = 0 ;
        A = node(C,D,E), tree_min(D,F), tree_min(E,G),
        Y = [C,F,G], minimum(Y,X), X = B.

minimum(A,B) :- A = [I], B = I;
        A = [X|Xs], minimum(Xs,C), X =< C, B = C ;
        A = [Y|Ys], minimum(Ys,D), D =< Y, B = D.
```

The inferred types by the type inference algorithm are:

```
tree_min :: tree_min1 x tree_min2 -> bool
tree_min1 = atom + node(tree_min2, tree_min1, tree_min1)
tree_min2 = A + int + float

minimum :: minimum1 x minimum2 -> bool
minimum1 = [ minimum2 | t ]
minimum2 = A + int + float
t2 = [] + [ minimum2 | t2 ]
```

By applying the closure operation on these types, we get the following:

```
tree_min :: tree_min1 x tree_min2 -> bool
tree_min1 = atom + node(tree_min2, tree_min1, tree_min1)
tree_min2 = int + float

minimum :: minimum1 x minimum2 -> bool
minimum1 = [ minimum2 | t ]
minimum2 = int + float
t2 = [] + [ minimum2 | t2 ]
```

These types, although closed and closer to the programmer's intention, still somehow are not that intuitive.

If we now add a predefined declaration of a tree data type and turn on the list flag, the algorithm outputs:

```
tree_minimum :: tree_minimum1 x tree_minimum2 -> bool
tree_minimum1 = tree(tree_minimum2)
tree_minimum2 = int + float

minimum :: minimum1 x minimum2 -> bool
minimum1 = list(minimum2)
minimum2 = int + float

tree(X) = empty + node(X, tree(X), tree(X))
list(Y) = [] + [ Y | list(Y) ]
```

These would be the types declared in most functional programming languages, and the types intended by the programmer.

6.8 Discussion

In this chapter, we presented *one* sound way to close open types. This is not the only way that we could do it. It is a heuristic, based on some principles, and the results we got are promising. They are also close to what we set out to do which was to get to the types that would be declared by programmers, without any need for type annotations for each predicate. The intended types for some predicates are obtained by our algorithm, but some others are not.

In order not to be too restrictive and fail a lot of type inference applications due to closure of open types not being possible, we argue that this step should be an optional step, or be used to provide recommendations to the programmer. As such, programs would still compile if type inference succeeded, but a warning is generated by the algorithm that the types are open and cannot be closed, or that a closure is possible and the returned types are presented.

The results we got when declaring data type definitions for the data structures in some programs are often even better than using closure, and type definitions can get us even closer to the programmer's intended meaning for programs. The amount of extra work is irrelevant since the data declarations only occur once in a program, but can be used a large number of times, and even re-used from program to program. One could even argue that some structures, such as lists, are so frequent that they could be introduced as a polymorphic base type, in a way, and be inferred as such automatically.

Concluding, closed types, here defined by the simple principle that types should be defined by (and only by) the set of ground symbols existing in the program, give us a simple way of approaching the output of automatic type inference to the actual programmer's intention.

Chapter 7

Conclusions and Further Work

In this thesis, we presented a type discipline for logic programming that both dynamically and statically defines well-typed programs and detects type errors.

For dynamic typing, we provided a typed unification algorithm based on a three-valued logic. Given such an algorithm, we also described a typed operational semantics called TSLD-resolution that follows a similar approach to SLD-resolution, and calculates the derivation trees for programs and queries. We also defined a typed declarative semantics and proved that TSLD-resolution is correct with respect to that semantics.

For static typing, we defined a more complex typed declarative semantics that allows for different interpretations of function symbols. We also define a type language that describes regular types and defined their semantics. Then, we defined a type system that described exactly which programs are well-typed. Finally, we proved that the type system is sound, *i.e.*, if there is a derivation for a certain predicate with a type, then the predicate belongs to the semantics of that type.

We then presented a type inference algorithm that, given a program, infers types for the predicates in that program. We proved the correctness of the algorithm: types inferred are proved to have a derivation in the type system. The algorithm is based on type constraint generation followed by constraint solving.

Lastly, we presented the definition of closed types, following some fundamental principles. We presented an algorithm that given open types calculates closed types that are an instance of the open ones. We also described how we could change the type inference algorithm in order to include data type definitions. We then showed results of both the closure operation and data type definitions on some example programs.

We will now reflect on the results we obtained, and discuss possible future lines of work.

7.1 Dynamic Typing

TSLD-resolution can be used to dynamically detect type errors. The operational semantics uses a typed unification algorithm. Therefore, this algorithm may influence the results obtained by the semantics. Our typed unification algorithm only really deals with constants, but lacks the capability of identifying lists, or any other recursively defined structure. Supposing that we have such an algorithm, we could in fact dynamically detect type errors in a larger class of programs.

One major limitation of dynamic type checking is the necessity of running the program. Unfortunately, logic programming worsens this limitation due to the use of backtracking for finding several answers. Moreover, in TSLD-resolution, unlike in SLD-resolution, $false$ results do not stop execution, so the execution of simple programs is less efficient in the typed semantics.

A certain compromise could be met, where we detect some, but not all, type errors. This compromise can be seen as a step towards having type information as part of the logic programming semantics, or as an argument for the use of static typing, where the issues with execution and efficiency do not occur.

7.2 Static Typing and Type Inference

Having static typing has the major advantage of being able to detect type errors at compile-type, easing program development. The major disadvantage is the fact that introducing types in a programming language always introduces some loss of information.

We argue this loss is not that relevant when compared to the benefits that come from having a type system.

The type inference algorithm infers types correctly, but types can be overly-broad. This mostly comes from the programming style. This limitation is compensated by the lack of extra work required from the programmers in order to still have type verification in their programs.

7.3 Closed Types and Data Type Declaration

The results we got from the implementation of this work are types that are much closer to the programmer's intention than previous results from other type inference algorithms.

Usually, type declaration is recommended, which involves the declaration of type signatures for each predicate in a program. In our approach, we can have no extra information and use closure, or just declarations of data types, and we get similar results. We also show some examples of how static typing, and in particular the closure operation, can be used for bug detection.

One major limitation of the closure operation is that it will fail sometimes. However it can be used as a recommendation system, or as an optional step in a real life application.

7.4 Further Developments

In the future, we would like to apply our type inference algorithm to several larger examples that use more complex, yet widely used, data structures, such as tries or red and black trees.

We would also like to provide a detailed comparison of the results obtained by our type inference algorithm and other type inference algorithms. This line of work could hopefully lead to the definition of a general framework for type inference in logic programming, where one could input different options in order to obtain a type inference algorithm as described by several different authors.

We would also like to extend TSLD, or in specific the typed unification algorithm, to specially interpreted functions. In this case, the algorithm would have to perform some kind of type checking on the terms it is trying to unify.

Finally, we would like to extend the type system to include refinement types as defined by Freeman and Pfenning in [82]. Refinement types include extra information about types, for instance we can have lists with a length smaller than n. One big success application using refinement types was Liquid Haskell [83], where the inference of refinement types was possible.

7.5 Final Comments

Untyped Prolog has the advantage of being very flexible and allowing for a larger class of programs, when compared to typed logic programming languages. However, sometimes those programs can be very hard to debug and understand, while usually typed logic programming programs have a longer development time, but are much more reliable.

In this thesis, we present a solution that is halfway between untyped Prolog and the languages that have mandatory type declaration. We try to have the best of both worlds: retain some of the flexibility of untyped Prolog while using type information in an optional, but sound, way.

Bibliography

[1] Barbosa, J., Florido, M. & Santos Costa, V. Closed types for logic programming. In *25th Int. Workshop on Functional and Logic Programming (WFLP 2017)* (2017).

[2] Barbosa, J., Florido, M. & Santos Costa, V. A three-valued semantics for typed logic programming. In *Proceedings 35th International Conference on Logic Programming (Technical Communications), ICLP 2019 Technical Communications, Las Cruces, NM, USA, September 20-25, 2019*, vol. 306 of *EPTCS*, 36–51 (2019).

[3] Barbosa, J., Florido, M. & Santos Costa, V. Data type inference for logic programming. In De Angelis, E. & Vanhoof, W. (eds.) *Logic-Based Program Synthesis and Transformation*, 16–37 (Springer International Publishing, Cham, 2022).

[4] Barbosa, J., Florido, M. & Santos Costa, V. Typed SLD-Resolution: Dynamic typing for logic programming. In Villanueva, A. (ed.) *Logic-Based Program Synthesis and Transformation*, 123–141 (Springer International Publishing, Cham, 2022).

[5] Körner, P. *et al.* Fifty years of prolog and beyond. *Theory and Practice of Logic Programming* 1–83 (2022).

[6] Zobel, J. Derivation of polymorphic types for Prolog programs. In *Logic Programming, Proceedings of the Fourth International Conference, Melbourne, 1987* (1987).

[7] Dart, P. W. & Zobel, J. A regular type language for logic programs. In Pfenning, F. (ed.) *Types in Logic Programming*, 157–187 (The MIT Press, 1992).

[8] Lu, L. On Dart-Zobel algorithm for testing regular type inclusion. *SIGPLAN Not.* **36**, 81–85 (2001).

[9] Frühwirth, T. W., Shapiro, E. Y., Vardi, M. Y. & Yardeni, E. Logic programs as types for logic programs. In *Proc. of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Netherlands,1991*, 300–309 (1991).

[10] Yardeni, E., Frühwirth, T. W. & Shapiro, E. Polymorphically typed logic programs. In Pfenning, F. (ed.) *Types in Logic Programming*, 63–90 (The MIT Press, 1992).

[11] Mycroft, A. & O'Keefe, R. A. A polymorphic type system for Prolog. *Artif. Intell.* **23**, 295–307 (1984).

[12] Lakshman, T. L. & Reddy, U. S. 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* (1991).

[13] Schrijvers, T., Costa, V. S., Wielemaker, J. & Demoen, B. Towards typed Prolog. In *Logic Programming, 24th International Conference, ICLP 2008, Udine, Italy, Proceedings*, 693–697 (2008).

[14] Drabent, W., Małuszyński, J. & Pietrzak, P. Using parametric set constraints for locating errors in CLP programs. *Theory and Practice of Logic Programming* **2**, 549–610 (2002).

[15] Schrijvers, T., Costa, V. S., Wielemaker, J. & Demoen, B. Towards Typed Prolog. In de la Banda, M. G. & Pontelli, E. (eds.) *Proceedings ICLP*, vol. 5366 of *Lecture Notes in Computer Science*, 693–697 (Springer, 2008).

[16] Hanus, M. Multiparadigm languages. In Gonzalez, T. F., Diaz-Herrera, J. & Tucker, A. (eds.) *Computing Handbook, Third Edition: Computer Science and Software Engineering* (CRC Press, 2014).

[17] Bruynooghe, M. & Janssens, G. An instance of abstract interpretation integrating type and mode inferencing. In *Fifth International Conference and Symposium, Washington, 1988*, 669–683 (1988).

[18] Gallagher, J. P. Analysis of logic programs using regular tree languages. In Vidal, G. (ed.) *Logic-Based Program Synthesis and Transformation*, 1–3 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2012).

[19] Lu, L. Polymorphic type analysis in logic programs by abstract interpretation. *The Journal of Logic Programming* **36**, 1–54 (1998). URL https://www.sciencedirect.com/science/article/pii/S0743106697100103.

[20] Mishra, P. Towards a theory of types in Prolog. In *Proceedings of the 1984 International Symposium on Logic Programming, Atlantic City, New Jersey, USA, February 6-9, 1984*, 289–298 (IEEE-CS, 1984).

[21] Hermenegildo, M. V. *et al.* An overview of the ciao system. In *Rule-Based Reasoning, Programming, and Applications - 5th International Symposium, RuleML 2011 - Spain, 2011*, 2 (2011).

[22] Fruwirth, T. Type inference by program transformation and partial evaluation. In *Proceedings. 1988 International Conference on Computer Languages*, 347–354 (1988).

[23] Yardeni, E., Fruehwirth, T. & Shapiro, E. Polymorphically typed logic programs. In *Types in Logic Programming*, 63–90 (MIT Press, 1991).

[24] Naish, L. Specification=program+types. In Nori, K. V. (ed.) *Foundations of Software Technology and Theoretical Computer Science*, 326–339 (Springer Berlin Heidelberg, Berlin, Heidelberg, 1987).

[25] Naish, L. Types and the intended meaning of logic programs. In *Types in Logic Programming*, 189–216 (The MIT Press, 1992).

[26] Schrijvers, T., Bruynooghe, M. & Gallagher, J. P. From monomorphic to polymorphic well-typings and beyond. In *Logic-Based Program Synthesis and Transformation, 18th International Symposium, LOPSTR 2008, Spain, July 17-18, 2008*, 152–167 (2008).

[27] Bruynooghe, M., Gallagher, J. & Van Humbeeck, W. Inference of well-typings for logic programs with application to termination analysis. In Hankin, C. & Siveroni, I. (eds.) *Static Analysis*, 35–51 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2005).

[28] Hermenegildo, M., Puebla, G., Bueno, F. & Garcia, P. L. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming* **58**, 115–140 (2005).

[29] Puebla, G., Bueno, F. & Hermenegildo, M. V. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, no. 1817 in LNCS, 273–292 (Springer-Verlag, 2000). URL https://cliplab.org/papers/assrt-theoret-framework-lopstr99_bitmap.pdf.

[30] Bueno, F. *et al.* On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd Int'l. Workshop on Automated Debugging–AADEBUG'97*, 155–170 (U. of Linköping Press, Linköping, Sweden, 1997). URL https://cliplab.org/papers/assert-lang-ws_bitmap.pdf.

[31] Hermenegildo, M. V., Puebla, G. & Bueno, F. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In Apt, K. R., Marek, V. W., Truszczynski, M. & Warren, D. S. (eds.) *The Logic Programming Paradigm: a 25–Year Perspective*, 161–192 (Springer, 1999).

[32] Lopez-Garcia, P., Bueno, F. & Hermenegildo, M. V. Automatic Inference of Determinacy and Mutual Exclusion for Logic Programs Using Mode and Type Information. *New Generation Computing* **28**, 117–206 (2010).

[33] Puebla, G., Bueno, F. & Hermenegildo, M. V. An Assertion Language for Constraint Logic Programs. In Deransart, P., Hermenegildo, M. V. & Małuszynski, J. (eds.) *Analysis and Visualization Tools for Constraint Programming*, vol. 1870 of *Lecture Notes in Computer Science*, 23–61 (Springer, 2000).

[34] Deransart, P., Ed-Dbali, A. & Cervoni, L. *Prolog - the standard: reference manual* (Springer, 1996).

[35] Martelli, A. & Montanari, U. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.* **4**, 258–282 (1982).

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

[37] Colmerauer, A., Kanoui, H., Pasero, R. & Roussel, P. Un système de communication homme-machine en français. Rapport préliminaire de fin de contrat IRIA. Tech. Rep., Faculté des Sciences de Luminy, Université Aix-Marseille II (1973).

[38] Van Emden, M. H. & Kowalski, R. A. The semantics of predicate logic as a programming language. *J. ACM* **23**, 733–742 (1976).

[39] Körner, P. *et al.* Fifty years of Prolog and beyond. *Theory and Practice of Logic Programming* 1–83 (2022).

[40] Apt, K. R. *From Logic Programming to Prolog* (Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996).

[41] Lloyd, J. W. *Foundations of Logic Programming* (Springer-Verlag, Berlin, Heidelberg, 1984).

[42] Kowalski, R. Algorithm = Logic + Control. *Communications of the ACM* **22**, 424–436 (1979).

[43] Robinson, J. A. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM* **12**, 23–41 (1965).

[44] Debray, S. K. & Mishra, P. Denotational and operational semantics for Prolog. *The Journal of Logic Programming* **5**, 61 – 91 (1988).

[45] Yardeni, E. & Shapiro, E. A type system for logic programs. *The Journal of Logic Programming* **10**, 125–153 (1991). URL https://www.sciencedirect.com/science/article/pii/074310669180002U.

[46] Gallagher, J. P. & Henriksen, K. S. Abstract Domains Based on Regular Types. In *International Conference on Logic Programming*, 27–42 (Springer, 2004).

[47] Solomon, M. Type definitions with parameters. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, 31–38 (Association for Computing Machinery, New York, NY, USA, 1978). URL https://doi.org/10.1145/512760.512765.

[48] Smaus, J.-G., Hill, P. M. & King, A. Mode analysis domains for typed logic programs. In Bossi, A. (ed.) *Logic-Based Program Synthesis and Transformation*, 82–101 (Springer Berlin Heidelberg, Berlin, Heidelberg, 2000).

[49] Somogyi, Z., Henderson, F. & Conway, T. C. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. *Journal of Logic Programming* **29**, 17–64 (1996).

[50] Hanus, M. & Krone, J. A Typeful Integration of SQL into Curry. In Schwarz, S. & Voigtländer, J. (eds.) *Proceedings WLP 2015/2016 and WFLP 2016*, vol. 234 of *EPTCS*, 104—119 (CoRR, 2017).

[51] Gallagher, J. P. & de Waal, D. A. Fast and precise regular approximations of logic programs. In *Logic Programming, International Conference on Logic Programming, Italy, 1994*, 599–613 (1994).

[52] Naish, L. A three-valued semantics for logic programmers. *Theory and Practice of Logic Programming (TPLP)* **6**, 509–538 (2006).

[53] Naish, L. A three-valued semantic for horn clause programs. In *Australasian Computer Science Conference*, 174 (IEEE Computer Society, Los Alamitos, CA, USA, 2000). URL https://doi.ieeecomputersociety.org/10.1109/ACSC.2000.824399.

[54] Barbuti, R. & Giacobazzi, R. A bottom-up polymorphic type inference in logic programming. *Sci. Comput. Program.* **19**, 281–313 (1992).

[55] Hadjichristodoulou, S. 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, 451–457 (2012).

[56] Henglein, F. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.* **15**, 253–289 (1993).

[57] Schrijvers, T. & Bruynooghe, M. Polymorphic algebraic data type reconstruction. In *Proc. 8th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '06, 85–96 (ACM, New York, NY, USA, 2006).

[58] Hermenegildo, M. V. *et al.* An Overview of Ciao and its Design Philosophy. *Theory and Practice of Logic Programming* **12**, 219–252 (2012). URL http://arxiv.org/abs/1102.5497.

[59] Vaucheret, C. & Bueno, F. More precise yet efficient type inference for logic programs. In *Proceedings of the 9th International Symposium on Static Analysis*, SAS '02, 102–116 (Springer-Verlag, 2002).

[60] Pietrzak, P., Correas, J., Puebla, G. & Hermenegildo, M. V. A practical type analysis for verification of modular Prolog programs. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, 61–70 (2008).

[61] Mycroft, A. & O'Keefe, R. A. A polymorphic type system for Prolog. *Artificial intelligence* **23**, 295–307 (1984).

[62] van Emden, M. & Kowalski, R. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM* **23**, 733–742 (1976).

[63] Pereira, L. M. Rational debugging in logic programming. In *Proceedings of the Third International Conference on Logic Programming*, 203–210 (Springer-Verlag, London, UK, UK, 1986).

[64] Kleene, S. C. On notation for ordinal numbers. *Journal Symbolic Logic* **3** (1938).

[65] Beall, J. Off-topic: A new interpretation of weak-kleene logic. *The Australasian Journal of Logic* **13** (2016).

[66] Saglam, H. & Gallagher, J. Approximating Constraint Logic Programs Using Polymorphic Types and Regular Descriptions. Technical Report CSTR-95-17, Dep. of Computer Science, U. of Bristol, Bristol BS8 1TR (1995).

[67] Bochvar, D. & Bergmann, M. On a three-valued logical calculus and its application to the analysis of the paradoxes of the classical extended functional calculus. *History and Philosophy of Logic* **2**, 87–112 (1981).

[68] Herbrand, J. *Recherches sur la théorie de la démonstration* (Numdam, 1930). URL http://eudml.org/doc/192791.

[69] Wadler, P. & Findler, R. B. Well-typed programs can't be blamed. In Castagna, G. (ed.) *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, York, UK, March 22-29, 2009. Proceedings*, vol. 5502 of *Lecture Notes in Computer Science*, 1–16 (Springer, 2009).

[70] van Emden, M. H. & Kowalski, R. A. The semantics of predicate logic as a programming language. *J. ACM* **23**, 733–742 (1976).

[71] Van Roy, P. L. *Can Logic Programming Execute as Fast as Imperative Programming?* Ph.D. thesis, EECS Department, University of California, Berkeley (1990). URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/1990/5686.html.

[72] Pyo, C. & Reddy, U. S. Inference of polymorphic types for logic programs. In *Logic Programming, Proceedings of the North American Conference 1989, USA, 1989. 2 Volumes*, 1115–1132 (1989).

[73] Florido, M. & Damas, L. Types as theories. In *Proc. of post-conference workshop on Proofs and Types, Joint International Conference and Symposium on Logic Programming* (1992).

[74] Hanus, M. Polymorphic high-order programming in Prolog. In Levi, G. & Martelli, M. (eds.) *Logic Programming, Proceedings of the Sixth International Conference, Lisbon, Portugal, June 19-23, 1989*, 382–397 (MIT Press, 1989).

[75] Kfoury, A. J., Tiuryn, J. & Urzyczyn, P. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.* **15**, 290–311 (1993).

[76] Ullman, J. D. *Principles of Database and Knowledge-base Systems* (Computer Science Press, Inc., 1988).

[77] Maher, M. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *Proceedings Third Annual Symposium on Logic in Computer Science*, 348,349,350,351,352,353,354,355,356,357 (IEEE Computer Society, Los Alamitos, CA, USA, 1988). URL https://doi.ieeecomputersociety.org/10.1109/LICS.1988.5132.

[78] Naish, L. Types and the intended meaning of logic programs. In Pfenning, F. (ed.) *Types in Logic Programming*, 189–216 (The MIT Press, 1992).

[79] Ullman, J. D. & Zaniolo, C. Deductive databases: Achievements and future directions. *SIGMOD Rec.* **19**, 75–82 (1990). URL https://doi.org/10.1145/122058.122067.

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

[81] Hanus, M. Functional logic programming: From theory to Curry. In *Programming Logics - Essays in Memory of Harald Ganzinger*, 123–168 (2013).

[82] Freeman, T. & Pfenning, F. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, 268–277 (ACM, New York, NY, USA, 1991). URL http://doi.acm.org/10.1145/113445.113468.

[83] Vazou, N., Seidel, E. L., Jhala, R., Vytiniotis, D. & Peyton-Jones, S. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, 269–282 (Association for Computing Machinery, New York, NY, USA, 2014). URL https://doi.org/10.1145/2628136.2628161.