Polymorphic Directional Types for Logic Programming

Paweł Rychlikowski         Tomasz Truderung

Ph.D. Thesis

supervisor: Prof. Leszek Pacholski

Instytut Informatyki
Uniwersytet Wrocławski
ul. Przesmyckiego 20
PL-51–151 Wrocław

January 11, 2000
# Contents

1 Introduction 1

I The Basic System 7

2 The Type System 9
2.1 Basic Definitions ................................................. 9
2.2 Types of Terms .................................................. 10
   2.2.1 The Language of Types ..................................... 10
   2.2.2 Signatures ................................................. 11
   2.2.3 Equality Axioms ........................................... 13
   2.2.4 Term Typing ............................................... 14
2.3 Properties of Types ............................................. 15
   2.3.1 The Equality and the Inequality Relation ............... 15
   2.3.2 Type Substitutions ...................................... 17
   2.3.3 Environments ............................................ 19
2.4 Types of Predicates ........................................... 20
   2.4.1 Introduction .............................................. 20
   2.4.2 The Predicate Typing Rules ............................... 21
2.5 Technical Lemmas ............................................. 25
   2.5.1 Environments ............................................ 25
   2.5.2 Consequence Operator ................................... 27
2.6 Intersection Lemma ........................................... 28
2.7 Derivations .................................................. 32

3 Semantics 39
3.1 Semantics of Types ........................................... 39
3.2 Soundness of Equality Axioms ................................. 40
3.3 Soundness of Term Typing Rules ............................... 41
3.4 Soundness of Consequence Operator Rules ................... 41
3.5 Soundness Theorems ......................................... 41

4 Type Checking 47
4.1 Checking Type Formulas ....................................... 47
   4.1.1 Algorithm ................................................ 48
   4.1.2 Discriminative Types ................................... 56
   4.1.3 Comparing Ground Types ................................ 62
4.2 Auxiliary Functions ......................................... 66
   4.2.1 Best substitutions. Function BS ....................... 67
   4.2.2 Best Types. Function BT ............................... 70
4.3 The Type Checking Algorithm ................................ 71
Contents

4.4 Complexity of the Type Checking Algorithm .......................... 77
4.5 Hardness of Type Checking ........................................... 82
  4.5.1 co-NP Hardness for τ-closed Families of Types ............... 83
  4.5.2 \( \Sigma^p_2 \) Hardness for Arbitrary Types ..................... 85

5 Reconstruction of Guarantees ........................................ 89
  5.1 The Algorithm .................................................. 89
  5.1.1 Complexity of Algorithm 5.1 ................................ 96
  5.2 Hardness of the Reconstruction of Guarantees ................. 97

II The System with Pruning ............................................ 107

6 The Main Type ..................................................... 109
  6.1 Introduction .................................................. 109
  6.2 System P ....................................................... 109
    6.2.1 Axioms and Rules of System P ............................... 109
    6.2.2 Signatures in the System P ................................ 110
    6.2.3 Properties of the System ................................ 110
  6.3 Pruning ...................................................... 110
  6.4 The Proof of the Pruning Theorem .............................. 112
  6.5 Existence of the Main Type .................................. 123

7 Type Reconstruction ............................................... 129
  7.1 Strongly Typeable Predicates .................................. 130
    7.1.1 Algorithm ................................................ 131
  7.2 Properties of Type Reconstruction Algorithm .................. 133

8 Extensions .......................................................... 139
  8.1 Error Prevention .............................................. 139
  8.2 Connections between Variables ............................... 141
    8.2.1 Types of Terms ........................................... 142
    8.2.2 Axioms and Rules ........................................ 142
    8.2.3 Properties of System C .................................. 144
    8.2.4 Semantics ................................................ 144
    8.2.5 Pruning Theorem ......................................... 144

9 Conclusions .......................................................... 145

III Appendixes ......................................................... 147

A Proofs ............................................................... 149
  A.1 Proofs for Chapter 2 ........................................... 149
  A.2 Proofs for Chapter 3 ........................................... 157
  A.3 Proofs for Chapter 4 ........................................... 166
  A.4 Proofs for Chapter 6 ........................................... 183
    A.4.1 Properties of System P ................................... 183
    A.4.2 Semantics ................................................ 183
    A.4.3 Type Checking and Guarantees ............................ 183
  A.5 Proofs for Chapter 7 ........................................... 186
  A.6 Proofs for Chapter 8 ........................................... 189
    A.6.1 Properties of System C ................................... 189
    A.6.2 Semantics of System C ................................... 189
A.6.3 The Pruning Theorem for System C .......................... 191

B Axioms and rules .................................................. 197
   B.1 Rules in System B ............................................ 197
   B.2 Rules in the System P ....................................... 200
Chapter 1

Introduction

Logic programming languages have been created as untyped languages. Therefore they give a lot of freedom to write programs and construct terms. But there is a price to pay for it. Many errors, which could be detected during compilation time, remain in the code. Debugging is very difficult. To remedy this, several type systems for logic languages have been designed. See e.g. [19] for a survey.

Type systems can be classified as prescriptive or descriptive [21]. In the descriptive approach a type serves as an approximation of the success set of a predicate, i.e. the set of terms for which the predicate is true. Types can be automatically inferred from a program. A programmer need not provide any type declarations. However, the types so obtained may not exactly specify the programmers intentions, and some facts in the success set of a predicate can be considered as senseless. In the prescriptive approach some subsets of the universe are declared as domains associated with types. A programmer must provide some type declarations which express the intended meaning of a program. Type inference algorithm can complete the missing part of information on types. The main purpose of such systems is detection of errors. Some languages (for instance AProlog [17]) based on this approach use typed version of unification which changes the semantics of programs.

Both approaches described above, although obviously useful, do not convey any information concerning an important aspect of predicates: relationship between input and output of a predicate.

A given argument position of a Prolog predicate has no fixed input or output mode. For instance, when the predicate append is used to concatenate two lists, the first and the second argument have input mode, while the third argument has the output mode. When the same predicate is used to split a list given in the third argument, the first and the second argument have output mode. So, it is clear that the mode of a predicate depends on the way the predicate is being used. Some type systems distinguish input and output argument positions by fixing modes of predicates. Of course it restricts the usage of a predicate, i.e. if append is used to concatenate lists in one part of a program, it cannot be used for splitting in other parts.

An interesting combination of modes and types in the notion of a directional type has been proposed by U.S. Reddy in [20]. He used Girard’s linear logic to build a system in which the input of a predicate can be in a subtle way mixed with its output. In his system the types are polymorphic. However, as in simple the mode systems, the direction of data-flow in a predicate is fixed.

Modes describe only one, directional aspect of predicates in Prolog. The other aspect has to do with the fact, that unlike in functional or imperative languages, the same variable before and after execution of a program, although represents still the
same object, can carry much more information, e.g. a free variable $X$ after execution of the goal $\text{member}(X, \{1, 2, 3\})$ becomes an integer.

The directional type system of Aiken and Lakshman [1], which as a starting point
had the idea of mode analysis presented by Brossard et al. in [6], is a successful
attempt to describe both logic and directional features of predicates. A directional
type of a predicate is defined as a pair $(\tau_1, \ldots, \tau_n), (\sigma_1, \ldots, \sigma_n)$ of
tuples, written as $(\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)$, in which
$(\tau_1, \ldots, \tau_n)$ are called input types, and $(\sigma_1, \ldots, \sigma_n)$ output
types of the predicate. Semantically, a predicate $p$ has a type $(\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)$ if the following is true: whenever, for $i \in \{1, \ldots, n\}$,
a term $t_i$ has type $\tau_i$, then after a successful derivation of the goal $p(t_1, \ldots, t_n)$, term $t_i$
gets type $\sigma_i$. It is assumed in this system that subgoals are resolved in left-to-right
order (LD resolution), which is consistent with logic programming languages used in
practice.

Formally, Aiken and Lakshman define well-typedness of program as follows. Consider
a program with a directional type $(\tau_i \rightarrow \sigma_i)$, for each predicate $p_i$. The clause
$p_0(t_0) \vdash p_1(t_1), \ldots, p_n(t_n)$ is well-typed if
\[
\forall 1 \leq j \leq n \left( \models \ i_0 : \tau_0 \land \bigwedge_{1 \leq i < j} t_i : \sigma_i \Rightarrow t_j : \tau_j \right), \quad \text{and}
\models \ i_0 : \tau_0 \land \bigwedge_{1 \leq i \leq n} t_i : \sigma_i \Rightarrow t_0 : \sigma_0.
\]
A program is well-typed if every clause is. A query $p_1(t_1), \ldots, p_n(t_n)$ is well-typed
if $\forall 1 \leq j \leq n \models \bigwedge_{1 \leq k \leq j} t_k : \sigma_k \Rightarrow t_j : \tau_j$. It is known [5] that LD-resolution is type
consistent for well-typed programs, i.e. for a well-typed program with a directional
type $\tau_i \rightarrow \sigma_i$, for each predicate $p_i$, and a well-typed query $p_1(t_1), \ldots, p_n(t_n)$, for
any answer substitution $\theta$, it is the case that $\theta(t_i) : \sigma_i$, for each $i \in \{1, \ldots, n\}$.

Types $\tau_i, \sigma_i$ are expressed by ground set expressions, so directional types in this
system are not polymorphic. Let us also notice that the definition assumes that there
is exactly one directional type for a predicate.

For instance, predicate append has type
\[(\text{list}(\top), \text{list}(\top), \top) \rightarrow (\text{list}(\top), \text{list}(\top), \text{list}(\top)),\]
where $\top$ denotes the set of all terms. This type can be applied when append is used to
concatenate two lists. When append is used to split a list, a more appropriate type
is $(\top, \top, \text{list}(\top)) \rightarrow (\text{list}(\top), \text{list}(\top), \text{list}(\top))$. Furthermore, $\top$ in these types can be
replaced by any ground type, e.g. int. Set expressions allow to express more accurate
types. For example, append has also type which corresponds to concatenation of lists of
odd length. Usefulness of such types is however not obvious, since if a predicate
has one non-polymorphic type, then this type should by sufficiently general to be
useful in various contexts.

Aiken and Lakshman have given a procedure which maps the problem of checking
that a program is well-typed to a decision problem concerning set constraints. In
the general case the reduction is sound but not complete. For the discriminative
types the reduction is also complete. The algorithm based on this reduction works
in NEXPTIME. They have also shown that the problem whether a given program is
well-typed by a directional type is hard for EXPTIME.

The complexity of type verification and inference in the Aiken–Lakshman system
was studied by Charatonik and Podelski [8] who gave an algorithm for inferring
directional types. Since each predicate can have many directional types, and it is not
clear which type should be inferred, Charatonik and Podelski assume that a program
comes with a query which allows to fix just one “right” directional type. Furthermore,
Charatonik [7] showed that directional type checking is EXPTIME-complete.
J. Boye and J. Mahszyński in [3] and [4] designed a system where directional types are merged with polymorphic types of terms. They assign an input or output direction to every argument position of a predicate. For instance a predicate `append` in this system can have type

$$\forall \alpha. (\bot \alpha, \top \alpha, \top \alpha)$$

or a type

$$\forall \alpha. (\top \alpha, \bot \alpha, \bot \alpha)$$

where \(\bot\) denote an input argument position, and \(\top\) denotes an output argument position. They studied several verification conditions with special attention to well-typedness introduced by Bonsard et al. in [6]. They also considered a new criterion called S-well-typedness which uses the techniques developed in the field of attribute grammars (see Deransart, Mahszyński [10]). S-well typedness do not take into consideration a computation rule (i.e. a strategy of searching a SLD-tree), which is useful when reasoning about types of predicates with open structures. Authors describe type checking and limited interactive type reconstruction of non-polymorphic types.

In the system presented in this thesis types describe all possible ways using a predicate. To a large extent we follow Aiken and Lakshman [1], but our input and output types are not described by set expressions. To describe them we follow the notion of a type taken from [16] and [24]. Types of terms are polymorphic. They are constructed from variables and type constructors, we also allow union and intersection of types. This makes it possible to express relations between arguments in a way that would be impossible in any of the existing directional type systems. We also have a subtyping relation. Subtyping and type-checking are described by a system of derivation rules. We have a type checking algorithm working in EXPTIME.

In our system, in contrast to the system of Aiken and Lakshman, the type “list of odd length” cannot be expressed. On the other hand, however, polymorphic types express in fact a family of ground types. For instance,

$$(\text{list}(\alpha), \text{list}(\beta), \top) \rightarrow (\text{list}(\alpha), \text{list}(\beta), \text{list}(\alpha \cup \beta)).$$

is a type of `append`. If we substitute \(\alpha\) by \text{int}, and \(\beta\) by \text{real}, and if we use the fact that \text{int} \text{ \leq} \text{real}, then we obtain a ground type

$$(\text{list(int)}, \text{list(real)}, \top) \rightarrow (\text{list(int)}, \text{list(real)}, \text{list(real)}).$$

Another valid type of `append` is

$$(\text{list}(\alpha), \top, \text{list}(\gamma)) \rightarrow (\text{list}(\alpha \cap \gamma), \text{list}(\gamma), \text{list}(\gamma))$$

which can be applied when `append` is used to subtracting lists.

In many languages with polymorphic types there is a notion of the most general type. If a function (predicate) has the most general type \(t\), then all types of this function can be obtained from \(t\) by substitution. In directional type systems the most general type of a predicate does not exist. In our system, however, a set of all proper types of a predicate can be described in a compact way using the notion of a main type that we introduce here. We also introduce a new notion of pruning, which besides substitution and subtyping can be used to generate types from a given type. For instance, the all given above types of `append` can be generated from the main type of `append`.

We provide an algorithm, which for a given predicate computes its main type. Another important feature of our system is incrementality: after adding a new predicate it is not necessary to reconstruct types of all predicates that have already been typed.
Chapter 1. Introduction

An important application of our system can be to eliminate runtime errors due to an improper usage of a predicate. We can declare which types of predicates could be allowed, and then the type checker will report when a predicate is used in a way that we want to avoid.

Now, we summarize the most important properties of our system:

- we use directional types,
- types describe both declarative and procedural properties of predicates, we consider LD resolution used in e.g. Prolog,
- the system does not introduce any change to the standard semantics of logic programs,
- types are polymorphic,
- there is a subtyping relation on types,
- user provides signatures of term constructors which allows to determine the type of a term,
- types can be used to detect errors,
- there is a notion of a main type,
- an algorithms for type checking and reconstruction of the main type are provided.

The paper is organized as follows. In Part I we describe the basic version of our type system. In Chapter 2 we define the language of types, we present axioms and rules of the system. We also prove the basic properties of the system. We define a notion of derivation i.e. a method which from a correct type of a predicate generates other types. Derivation is based on parametric polymorphism and subtyping.

In Chapter 3 we present a semantics of the type system, and prove that types provable in our system describe both procedural and declarative properties of programs.

In Chapter 4 we present a type checking algorithm, and give lower and upper bounds on the complexity of the type checking problem. In Chapter 5 we define the problem of partial type reconstruction called 'reconstruction of guarantees', and study complexity of this problem.

In Part II we describe a slightly weaker version of our system. An important feature of this version, called System P, is the existence of the main type for an important class of predicates. In Chapter 6 we define System P by a modification of System B. We provide a new method of derivation types, called pruning. We define the notion of main type. From the main type of a predicate one can derive other types of this predicate. We describe the class of programs for which the main type exists.

In Chapter 7 we define a class of strongly typeable predicates, and give an algorithm which finds the main type for these predicates. We analyze the complexity of this algorithm.

In Chapter 8 we present two useful extensions of our system. First, we describe how our types can be useful for detecting errors. Second, we extend the language of types, and the set of inference rules in such a way that the system can prove more accurate types.

This paper has been extended by two appendixes. In the first we collect many technical and routine proofs that have been omitted in the main body of the thesis. These proofs can be skipped by almost every reader since they will not improve the understanding of the systems we have presented. However, we have added them in the appendix for the sake of completeness.

In the second appendix we give all the axioms and rules of systems B and P.
Part I

The Basic System
Chapter 2

The Type System

In this chapter we introduce the basic version of our type system, called *System B*. We define the polymorphic language of types, and present axioms and rules of the system.

2.1 Basic Definitions

In this section we give a few standard definitions and fix the notation.

We assume the existence of three disjoint sets: $V$ – the set of program variables (denoted by $X, Y, \ldots$), $F$ – the set of term constructors (e.g. $\cdot \cdot \cdot$, denoted by $f, g, \ldots$), $P$ – the set of predicate symbols (denoted by $p, q, \ldots$). Every element of $F$ and $P$ has a fixed *arity* (number of arguments).

**Definition 2.1.** The set $T$ of *terms* is the least set satisfying the following conditions:

(i) $V \subseteq T$.

(ii) if $t_i \in T$ for $i \in \{1, \ldots, n\}$, $f \in F$ and $f$ has arity $n$ then $f(t_1, \ldots, t_n) \in T$.

A term constructor of arity 0 is said be a constant.

**Definition 2.2.** The set $A$ of *atoms* is defined by

$$A = \{ p(t_1, \ldots, t_n) \mid p \in P, \ p \text{ has arity } n, \ t_1, \ldots, t_n \in T \}.$$ 

**Definition 2.3.** A *clause* is a tuple $(a_0, \ldots, a_n)$ of atoms, written as

$$a_0 \leftarrow a_1, \ldots, a_n.$$ 

$a_0$ is called the *head* of the clause, and $a_1, \ldots, a_n$ is called the *body*. If $a_0 = p(t_1, \ldots, t_k)$ then $p$ is said to be the *head predicate symbol* of the clause $a_0 \leftarrow a_1, \ldots, a_n$.

**Definition 2.4.** A *program* is a finite set of clauses.

**Definition 2.5.** If $x$ is a term or an atom, then $\text{var}(x)$ denotes the set of variables occurring in $x$. A term $t$ is *ground* if $\text{var}(t) = \emptyset$.

**Definition 2.6.** A substitution $\theta$ is a finite set of the form $\{ X_1/t_1, \ldots, X_n/t_n \}$, where each $X_i$ is a distinct program variable, and each $t_n$ is a term. The substitution $\theta$
Chapter 2. The Type System

defines a function from terms to terms as follows:

\[ \theta(X_i) = t_i \]
\[ \theta(Y) = Y \quad \text{for } Y \notin \{X_1, \ldots, X_n\} \]
\[ \theta(f(t_1, \ldots, t_n)) = f(\theta(t_1), \ldots, \theta(t_n)). \]

The set \( \{X_1, \ldots, X_n\} \) is called the domain of \( \theta \) and is denoted by \( \text{dom}(\theta) \). The set \( \{t_1, \ldots, t_n\} \), denoted by \( \text{range}(\theta) \), is called the range of \( \theta \).

The restriction of \( \theta \) to \( V \), denoted by \( \theta|_V \), is defined as follows

\[ \theta|_V = \{X_i/t_i \mid X_i/t_i \in \theta \text{ and } X_i \in V\}. \]

For substitutions \( \theta \) and \( \theta' \) we define the composition \( \theta \circ \theta' \):

\[ \theta \circ \theta' = \{X/t \mid t = \theta(\theta'(X)), \ t \neq X \}. \]

It is easy to see that \( (\theta \circ \theta')(t) = \theta(\theta'(t)) \), so definition of composition of substitutions is compatible with definition of compositions of functions.

\[ \Box \]

2.2 Types of Terms

2.2.1 The Language of Types

Now, we define the language of our type system. We assume the existence of two disjoint sets: \( V_T \) – the set of type variables (denoted by \( \alpha, \beta, \ldots \)), \( F_T \) – the set of type constructors (i.e. \( \text{list}(...) \), \( \text{tree}(...) \), \( \text{prod}(...) \), denoted by \( F, G, \ldots \)). We assume that \( V_T \) and \( F_T \) are disjoint with \( V, F, P \). Every element of \( F_T \) has a fixed arity (number of arguments).

**Definition 2.7.** The set of types is the least set \( T \) satisfying the following conditions:

(i) \( \top, \bot \in T \)

(ii) \( \alpha \in T \) \quad \text{where } \alpha \in V_T \)

(iii) \( F(\tau_1, \ldots, \tau_n) \in T \) \quad \text{where } \tau_i \in T, \ F \in F_T, \ F \text{ has arity } n \)

(iv) \( \tau \cap \sigma, \tau \cup \sigma \in T \) \quad \text{where } \tau, \sigma \in T

\[ \Box \]

The intuitions behind the notion of type is based on \( \top \) is the type of all terms, \( \bot \) denotes the type without any element, (iii) defines structural polymorphic types (e.g. list or tree) if \( n \geq 1 \) or atomic types (e.g. real) if \( n = 0 \).

**Example 2.1.** Assume that \( F_T \) contains type constructors \( \text{int} \) and \( \text{real} \) of arity 0, \( \text{list} \) of arity 1, and \( \text{prod} \) of arity 2. Then we have following types:

\( \alpha, \ \text{int}, \ \text{list}(\alpha), \ \text{list}(\bot), \ \text{prod}(\alpha, \alpha \cap \beta), \ \text{prod}(\text{real}, \text{list}(\text{int})). \)

Types with type variables (e.g. \( \text{list}(\alpha) \)) are called polymorphic, while types without variables are called ground.

\[ \Box \]

**Definition 2.8.** A type formula has one of the forms \( (\tau = \sigma), (t : \tau), \text{false} \), where \( \tau, \sigma \) are types, and \( t \) is a term.

\( \tau \leq \sigma \) is a shorthand for \( \tau \cap \sigma = \tau \) (the symbol \( \leq \) expresses the subtyping relation).

\[ \Box \]
In section 2.2.3 we present axioms which allow to prove formulas of the form \((\tau = \sigma)\) (and thus \(\tau \leq \sigma\)). A formula of the form \(t : \tau\) means “the term \(t\) has the type \(\tau\).” Rules for proving such formulas are presented in section 2.2.4.

**Definition 2.9.** An environment \(\Gamma\) is a finite set of formulas of the form \(\text{false} \lor (X : \tau)\), where \(X\) is a variable and \(\tau\) is a type. For each variable \(X\) there is at most one element of the form \((X : \tau)\) in \(\Gamma\). An environment \(\Gamma\) is ground if all types occurring in \(\Gamma\) are ground. In particular, \(\text{false}\) is ground.

In this chapter we introduce a proof system which allow to prove facts of the form \(\Gamma \vdash \varphi\), where \(\Gamma\) is an environment and \(\varphi\) is a type formula. We write \(\vdash \varphi\) instead of \(\varnothing \vdash \varphi\).

**Definition 2.10.** For a type \(F(\tau_1, \ldots, \tau_n)\), head\((F(\tau_1, \ldots, \tau_n)) = F\).

**Definition 2.11.** \(\text{var}(\tau)\) denotes the set of type variables occurring in \(\tau\). Similarly, \(\text{var}(\tau_1, \ldots, \tau_n)\) denotes the set of types variables occurring in types \(\tau_1, \ldots, \tau_n\).

The notion of substitution on types is similar to the notion of substitution of terms.

**Definition 2.12.** A type substitution \(\theta\) is a finite set of the form

\[
\{\alpha_1/\tau_1, \ldots, \alpha_n/\tau_n\},
\]

where each \(\alpha_i\) is a distinct type variable and each \(\tau_i\) is a type. The substitution \(\theta\) defines a function from types to types as follows:

\[
\begin{align*}
\theta(\top) &= \top \\
\theta(\bot) &= \bot \\
\theta(\alpha_i) &= \tau_i \\
\theta(\beta) &= \beta \quad (\beta \in V_T, \beta \not\in \alpha_1, \ldots, \alpha_n) \\
\theta(F(\tau_1, \ldots, \tau_n)) &= F(\theta(\tau_1), \ldots, \theta(\tau_n)) \\
\theta(\tau \cup \sigma) &= \theta(\tau) \cup \theta(\sigma) \\
\theta(\tau \cap \sigma) &= \theta(\tau) \cap \theta(\sigma)
\end{align*}
\]

The notions of the domain of \(\theta\), the range of \(\theta\), the restriction of \(\theta\) on \(V\) \((V \subseteq V_T)\), and the composition of two substitutions are defined exactly as the same notions for term substitution.

A type substitution \(\theta\) is ground if \(\text{range}(\theta)\) is a set of ground types (types without type variables).

### 2.2.2 Signatures

So far, we know how to construct correct types, but we do not know how to relate types to terms. We are going to express the relation between types and terms using notions of a signature.

**Definition 2.13.** A signature of arity \(n\) is a tuple \(\langle \tau_1, \ldots, \tau_n, \tau \rangle\) of types, written as \(\tau_1, \ldots, \tau_n \rightarrow \tau\), where

(i) \(\tau\) is a type of the form \(F(x_1, \ldots, x_k)\) where, for each \(i \in \{1, \ldots, k\}\), \(x_i\) is either \(\bot\), or a distinct type variable,

(ii) for each \(i \in \{1, \ldots, n\}\), \(\tau_i\) is either a type variable or has the form \(G(\alpha_1, \ldots, \alpha_l)\), where \(\alpha_1, \ldots, \alpha_l\) are distinct type variables from \(\text{var}(\tau)\).
Chapter 2. The Type System

(iii) \( \text{var}(\tau_1, \ldots, \tau_n) = \text{var}(\tau) \).

If a signature has arity 0 then we write \( \tau \) instead of \( (\rightarrow \tau) \). A signature \( \tau_1, \ldots, \tau_n \rightarrow \tau \) is atomic if \( \tau \) is an atomic type.

We assign to each term constructor \( f \in F \) of arity \( n \) one or more signatures of arity \( n \). We write \( f : s \) if the signature \( s \) is assigned to \( f \). We assume that

(a) all signatures assigned to \( f \) have the same left-hand sides,
(b) if a signature \( s \) assigned to \( f \) is not atomic then it is the only signature for \( f \).

The signatures assigned to term constructors determine types of terms. The intended meaning of \( f : \tau_1 \cdots \tau_n \rightarrow \tau \) is: “if terms \( t_1, \ldots, t_n \) have types \( \tau_1, \ldots, \tau_n \) respectively, then \( f(t_1, \ldots, t_n) \) has type \( \tau \).”

Point (b) states that we can assign more than one signature to a term constructor \( f \) if \( f \) gives an atomic type. It means that a term \( f \) can have more than one atomic type.

Example 2.2. Assume following signatures for the standard types:

\[
\begin{align*}
[\cdot] & : \alpha \ast \text{list}(\alpha) \rightarrow \text{list}(\alpha) & 0, 1, 2 : \text{int} \\
[ ] & : \text{list}(\bot) & 0, 1, 2 : \text{real} \\
\text{pair} & : \alpha \ast \beta \rightarrow \text{prod}(\alpha, \beta) & 5.2 : \text{real}
\end{align*}
\]

Note, that two signatures have been assigned to term constructors \( 0,1 \) and \( 2 \). It is possible, since both int and real are atomic, so (b) of Definition 2.13 is satisfied, and the left hand sides of both signatures are the same (empty), and thus (a) is satisfied too.

These signatures admit a claim that \( [\cdot] \) has type \( \text{list}(\bot) \), \( 1 \) has type \( \text{real} \), \( 1 \) has type \( \text{int} \), \( \text{pair}(1,1) \) has type \( \text{prod}(\text{int}, \text{int}) \).

Example 2.3. The following signatures are incorrect:

\[
\begin{align*}
F(\alpha, G(\beta)) & \quad \text{(does not satisfy (i) of Definition 2.13)}, \\
F(\bot) \rightarrow F(\bot) & \quad \text{(does not satisfy (iii))}, \\
F(\alpha) \rightarrow G(\alpha, \beta) & \quad \text{(does not satisfy (iii))}.
\end{align*}
\]

The following assignments are incorrect:

\[
\begin{align*}
a : F & \rightarrow G \quad \text{and} \quad a : G \rightarrow G & \quad \text{(does not satisfy \([\cdot]\))}, \\
g : G(\alpha) & \rightarrow F(\alpha) \quad \text{and} \quad g : G(\alpha) \rightarrow G(\alpha) & \quad \text{(does not satisfy \([\cdot]\))}.
\end{align*}
\]

\[\]  

Definition 2.14. For an atomic type \( \tau \) let

\[\text{Constr}(\tau) = \{ f \in F \mid \exists \tau_1, \ldots, \tau_n \text{ such that } f : \tau_1 \cdots \tau_n \rightarrow \tau \}.\]

\[\]  

Definition 2.15. Let \text{and} and \text{or} be partial function defined for (some) pairs of atomic types, such that

\[
\text{and}(\tau, \tau') = \begin{cases} 
\bot & \text{if } \text{Constr}(\tau) \cap \text{Constr}(\tau') = \emptyset \\
\sigma & \text{if } \text{Constr}(\sigma) = \text{Constr}(\tau) \cap \text{Constr}(\tau') \\
\text{undefined} & \text{if such } \sigma \text{ does not exist}
\end{cases}
\]

\[
\text{or}(\tau, \tau') = \begin{cases} 
\sigma & \text{if } \text{Constr}(\sigma) = \text{Constr}(\tau) \cup \text{Constr}(\tau') \\
\text{undefined} & \text{if such } \sigma \text{ does not exist}
\end{cases}
\]

\[10\]
We assume that signatures are assigned to term constructors in a such way that \textbf{and} and \textbf{or} are always defined.

This definition allows to compute the intersection and union of atomic types taking into consideration only the set of constructors of these types.

**Example 2.4.** For atomic types \texttt{int} and \texttt{real} defined above we have:
\[
\begin{align*}
\text{Constr}(\texttt{int}) &= \{1, 2, 3\} \\
\text{Constr}(\texttt{real}) &= \{1, 2, 3, 5, 2\}.
\end{align*}
\]
\[
\text{Constr}(\texttt{int}) \cap \text{Constr}(\texttt{real}) = \text{Constr}(\texttt{int}), \text{ and thus } \textbf{and}(\texttt{int}, \texttt{real}) = \texttt{int}.
\]

Note that Constr(\(\tau\)) is \textbf{not} the set of terms of the type \(\tau\). It is illustrated by the following example.

**Example 2.5.** Assume following signatures for types \(a\) and \(b\).
\[
\begin{align*}
0 : a & \quad 0 : b \\
s : a \rightarrow a & \quad s : a \rightarrow b
\end{align*}
\]
The signatures assigned to 0 and \(s\) have the same left side (empty and \(a\) respectively), and all the signatures above are atomic, thus \((a)\) and \((b)\) is fulfilled. The set of terms of type \(a\) is \(\{0, s(0), s(s(0)), \ldots\}\), and the set of terms of type \(b\) is \(\{0, r(0), r(r(0)), s(0), r(s(0)), r(r(s(0))), \ldots\}\).

Now, because Constr\((a)\) = \(\{0, s\}\) and Constr\((b)\) = \(\{0, s, r\}\), thus Constr\((a)\cap\text{Constr}(b)\) = Constr\((a)\). Therefore \textbf{and}(\(a, b\)) = \(a\) can be effectively computed.

We will show in Chapter 3 that if \textbf{and}(\(\tau, \tau'\)) = \(\tau''\) then \([\tau] \cap [\tau'] = [\tau'']\), where \([\tau]\) represents the set of terms of the type \(\tau\). Similarly, if \textbf{or}(\(\tau, \tau'\)) = \(\tau''\) then \([\tau] \cup [\tau'] = [\tau'']\).

### 2.2.3 Equality Axioms

In this section we give a set of axioms defining the equality relation. First we equip our system with the standard equality axioms adapted to our type language:

\[
\begin{align*}
\vdash \tau = \tau \\
\vdash \tau_1 = \tau_1', \ldots, \tau_n = \tau_n' & \quad \vdash F(\tau_1, \ldots, \tau_n) = F(\tau_1', \ldots, \tau_n') \\
\quad \text{for any type constructor } F \text{ of arity } n \\
\vdash \tau_1 = \tau_1', \quad \vdash \tau_2 = \tau_2' & \quad \vdash \tau_1 \cap \tau_2 = \tau_1' \cap \tau_2' \\
\vdash \tau_1 = \tau_1', \quad \vdash \tau_2 = \tau_2' & \quad \vdash \tau_1 \cup \tau_2 = \tau_1' \cup \tau_2' \\
\vdash \tau_1 = \tau_1', \quad \vdash \tau_2 = \tau_2' & \quad \vdash \tau_1 = \tau_2 \quad \vdash \tau_1' = \tau_2'
\end{align*}
\]

The axioms above allow to prove all standard properties of equality, like \textit{reflexivity}, \textit{symmetry} and \textit{transitivity} (see Lemma 2.16).
Chapter 2. The Type System

The following axioms (Ax6) – (Ax12) describe the well known universal properties of distributive lattices.

(Ax6) \( \vdash \tau \cup \tau = \tau \)

(Ax7) \( \vdash \tau \cup \sigma = \sigma \cup \tau \)

(Ax8) \( \vdash \tau \cup (\sigma \cup \xi) = (\tau \cup \sigma) \cup \xi \)

(Ax9) \( \vdash \tau \cup (\tau \cap \sigma) = \tau \)

(Ax10) \( \vdash \tau \cup (\sigma \cap \xi) = (\tau \cup \sigma) \cap (\tau \cup \xi) \)

(Ax11) \( \vdash \tau \cup \top = \top \)

(Ax12) \( \vdash \tau \cap \bot = \bot \)

Axioms (Ax13) – (Ax16) are specific to our types. In particular (Ax14) states, that two polymorphic types with different head symbols are disjoint.

(Ax13) \( \vdash F(\tau_1, \ldots, \tau_n) \cap F(\tau'_1, \ldots, \tau'_m) = F(\tau_1 \cap \tau'_1, \ldots, \tau_n \cap \tau'_m) \)

for any type constructor \( F \) of arity \( n \)


(Ax14) \( \vdash F(\tau_1, \ldots, \tau_n) \cap G(\tau'_1, \ldots, \tau'_m) = \bot \) if \( F \neq G \) and \( F \) or \( G \) is not atomic.

For atomic types \( \tau \) and \( \sigma \) (like \( \text{int} \) or \( \text{real} \)) we also have

(Ax15) \( \vdash \tau \cup \sigma = \text{or}(\tau, \sigma) \)

(Ax16) \( \vdash \tau \cap \sigma = \text{and}(\tau, \sigma) \)

2.2.4 Term Typing

In Figure 2.1 we give term typing rules. They allow to prove facts of the form \( t : \tau \) (term \( t \) has type \( \tau \)). In these rules \( \theta \) denotes a type substitution. In rule (\( T_6 \)) we allow \( n \) to be 0. If \( n = 0 \) then (\( T_6 \)) has no assumptions.

Rules (\( T_3 \)) and (\( T_6 \)) are known from other type systems. Rule (\( T_4 \)) allows to use subtyping in order to check a type of a term. Rule (\( T_5 \)) can be used when a given type is the conjunction of types. Let us notice that there is no appropriate rule for type which is the disjunction of types, since the fact \( \Gamma \vdash t : \tau_1 \cup \cdots \cup \tau_n \) can be obtained using rule (\( T_4 \)) from \( \Gamma \vdash t : \tau_k \) for some \( k \in \{1, \ldots, n\} \).

Example 2.6. Consider the signatures from Example 2.2. Let \( \Gamma = \{X : \text{list}(\text{real})\} \).

Then we can use term typing rules to prove that

\[ \Gamma \vdash [1 \mid x], \Box : \text{list}(\text{list}(\text{real})) \]

To illustrate how the rules work we give a detailed proof of this statement.

(1) \( \Gamma \vdash 1 : \text{real} \), because 1 has signature \( \text{real} \), and thus

\[ \Gamma \vdash 1 : \text{real} \]

is an instance of rule (\( T_6 \)) (with the empty set of assumptions and with any type substitution \( \theta \)).

(2) \( \Gamma \vdash x : \text{list}(\text{real}) \), since it is an instance of rule (\( T_3 \)).
2.3 Properties of Types

2.3.1 The Equality and the Inequality Relation

In this section we prove some basic properties of the equality and the inequality relation.

Lemma 2.16. The equality relation is reflexive, symmetric and transitive, i.e.

(a) $\vdash \tau = \tau$,
(b) $\vdash \tau = \tau'$ then $\vdash \tau' = \tau$,
(c) $\vdash \tau = \tau'$ and $\vdash \tau' = \tau''$ then $\vdash \tau = \tau''$.

Proof. (a) is given by (Ax1). Assume that $\vdash \tau = \tau'$. Then

$$\vdash \tau = \tau', \vdash \tau = \tau, \vdash \tau = \tau \Rightarrow \vdash \tau' = \tau.$$
is an instance of rule (Ax5). \( \vdash \tau = \tau' \) is the assumption, whereas \( \vdash \tau = \tau \) is given by (Ax1). That shows (b). Now, assume that \( \vdash \tau = \tau' \) and \( \vdash \tau = \tau'' \).

\[
\begin{align*}
\vdash \tau = \tau, \ &\vdash \tau' = \tau' \Rightarrow \\
\vdash \tau = \tau''
\end{align*}
\]

is an instance of rule (Ax5). That shows (c). \( \square \)

**Lemma 2.17.** For any types \( \tau, \tau' \) and \( \tau'' \) we have

\[
\begin{align*}
\text{if } \quad &\vdash \tau = \tau' \quad \text{then } \quad \vdash \tau \cap \tau'' = \tau' \cap \tau''; \\
\text{if } \quad &\vdash \tau = \tau'' \quad \text{then } \quad \vdash \tau \cup \tau' = \tau' \cup \tau'';
\end{align*}
\]

**Proof.** Assume that \( \vdash \tau = \tau' \). By (Ax1), we have \( \vdash \tau'' = \tau'' \). Now,

\[
\begin{align*}
\vdash \tau = \tau', \ &\vdash \tau'' = \tau'' \Rightarrow \\
\vdash \tau \cap \tau'' = \tau' \cap \tau''
\end{align*}
\]

is an instance of rule (Ax3), what proves \( \vdash \tau \cap \tau'' = \tau' \cap \tau'' \). In the very similar way, using (Ax4), we can show that \( \vdash \tau \cup \tau'' = \tau' \cup \tau'' \). \( \square \)

**Lemma 2.18.** The subtyping relation is reflexive, transitive and antisymmetric, i.e.

(a) \( \vdash \tau \leq \tau \),
(b) \( \vdash \tau \leq \tau' \) and \( \vdash \tau' \leq \tau'' \) then \( \vdash \tau \leq \tau'' \),
(c) \( \vdash \tau \leq \tau' \) and \( \vdash \tau' \leq \tau \) then \( \vdash \tau = \tau' \).

**Proof.** Axiom (Ax6) has the form \( \vdash \tau = \tau \cap \tau \), which means that \( \tau \leq \tau \), thus (a) holds.

Now, we will show (b). Let \( \vdash \tau \leq \sigma \) and \( \vdash \sigma \leq \delta \). Thus \( \vdash \tau \cap \sigma = \tau \) and \( \vdash \sigma \cap \delta = \sigma \). Hence,

\[
\begin{align*}
\vdash \tau = \tau \cap \sigma \\
\vdash \tau \cap \sigma = \tau \cap (\sigma \cap \delta) \quad &\text{by (Ax3)} \\
\vdash \tau \cap (\sigma \cap \delta) = (\tau \cap \sigma) \cap \delta \quad &\text{by (Ax8)} \\
\vdash (\tau \cap \sigma) \cap \delta = \tau \cap \delta \quad &\text{by (Ax3)}
\end{align*}
\]

which, by transitivity of equality relation, gives \( \vdash \tau \leq \delta \).

Finally let us show (c). Suppose that \( \vdash \tau \leq \sigma \) and \( \vdash \sigma \leq \tau \). It means that \( \vdash \tau \cap \sigma = \tau \) and \( \vdash \tau \cap \sigma = \sigma \). Therefore, Lemma 2.16 implies \( \vdash \tau = \sigma \). \( \square \)

**Remark 2.19.** In the rest of this paper we will tacitly use the following facts:

- \( \cup \) and \( \cap \) are associative and commutative. In particular, we shall omit parentheses in expressions like \( (\tau \cup \tau') \cup \tau'' \).
- \( = \) is reflexive, symmetric and transitive. In particular, we will write

\[
\vdash \tau_1 = \tau_2 = \cdots = \tau_n
\]

which means that \( \vdash \tau_i = \tau_j \) for any \( i, j \in \{1, \ldots, n\} \).
- \( \leq \) is reflexive, antisymmetric and transitive. For instance, we will write

\[
\vdash \tau_1 = \tau_2 \leq \tau_3 \leq \tau_4
\]

to express \( \vdash \tau_1 = \tau_2, \ \vdash \tau_1 \leq \tau_3, \ \vdash \tau_1 \leq \tau_4, \ \vdash \tau_2 \leq \tau_4 \), etc.

We have defined \( \vdash \tau \leq \sigma \) as a shorthand for \( \vdash \tau \cap \sigma = \tau \). The next lemma provides a similar relation between \( \leq \) and \( \cup \).
Lemma 2.20. \( \vdash \tau \leq \sigma \) iff \( \vdash \tau \cup \sigma = \sigma \).

Proof. Since \( \vdash \tau \leq \sigma \) is a shorthand for \( \vdash \tau \cap \sigma = \tau \), it suffices to show that \( \vdash \tau \cup \sigma = \sigma \)
is equivalent to \( \vdash \tau \cap \sigma = \tau \).

Suppose that \( \vdash \sigma = \tau \cup \sigma \). Then, by Lemma 2.17, \( \vdash \sigma \cap \tau = (\tau \cup \sigma) \cap \tau \) and hence, by (Ax9), we have \( \vdash \sigma \cap \tau = \tau \).

On the other hand, suppose that \( \vdash \tau = \tau \cap \sigma \). Then, by Lemma 2.17, \( \vdash \tau \cup \sigma = (\tau \cap \sigma) \cup \sigma \). Using (Ax9) we obtain \( \vdash \tau \cup \sigma = \sigma \).

Now, we present a fact dual to axiom (Ax13).

Lemma 2.21. We have

\[ \vdash F(\tau_1, \ldots, \tau_n) \cup F(\tau_1', \ldots, \tau_n') \leq F(\tau_1 \cup \tau_1', \ldots, \tau_n \cup \tau_n'). \]

Proof. If we show that \( L \cap R = L \), where \( L \) is the left-hand and \( R \) is the right-hand side of the inequality, lemma will be proved.

\[ \vdash L \cap R = (F(\tau_1, \ldots, \tau_n) \cup F(\tau_1', \ldots, \tau_n')) \cap F(\tau_1 \cup \tau_1', \ldots, \tau_n \cup \tau_n') \]
\[ = (F(\tau_1, \ldots, \tau_n) \cap F(\tau_1 \cup \tau_1', \ldots, \tau_n \cup \tau_n')) \cup (F(\tau_1', \ldots, \tau_n') \cap F(\tau_1 \cup \tau_1', \ldots, \tau_n \cup \tau_n')) \]
\[ = F(\tau_1 \cap (\tau_1 \cup \tau_1'), \ldots, \tau_n \cap (\tau_n \cup \tau_n')) \cup F(\tau_1', \ldots, \tau_n') \]
\[ \vdash L \cap R = F(\tau_1, \ldots, \tau_n) \cup F(\tau_1', \ldots, \tau_n') = L \]

The statement:

\[ F(\tau_1, \ldots, \tau_n) \cup F(\tau_1', \ldots, \tau_n') \geq F(\tau_1 \cup \tau_1', \ldots, \tau_n \cup \tau_n') \]
does not hold and cannot be proved in our system. In fact, \([\bot, 1] \) is a term of the type \( \text{list}(/\bot/) \cup \text{int} \) but it is not of the type \( \text{list}(/\bot/) \cup \text{list}(/\text{int}) \).

Lemma 2.22. If \( \vdash \tau_i \leq \tau_i' \) for \( i \in \{1, \ldots, n\} \) then

\[ \vdash F(\tau_1, \ldots, \tau_n) \leq F(\tau_1', \ldots, \tau_n'). \]

Proof. We have \( \vdash \tau_i = \tau_i \cap \tau_i' \) (\( i \in \{1, \ldots, n\} \)). So, by (Ax13)

\[ \vdash F(\tau_1, \ldots, \tau_n) \cap F(\tau_1', \ldots, \tau_n') = F(\tau_1 \cap \tau_1', \ldots, \tau_n \cap \tau_n') = F(\tau_1, \ldots, \tau_n) \]
which proves the lemma.

2.3.2 Type Substitutions

Now, we shall prove some properties of substitutions which will be used to establish correctness of the system. We also define an order on substitutions.

Lemma 2.23. For any type substitution \( \theta \), if \( \vdash \tau = \sigma \) then \( \vdash \theta(\tau) = \theta(\sigma) \).

Proof. \( \vdash \tau = \sigma \) means that there exists a proof \( Pr \) of \( \vdash \tau = \sigma \). We prove the lemma by induction on the structure of \( Pr \).

- Assume that \( \vdash \tau = \tau \) was obtained using rule (Ax1). To prove \( \vdash \theta(\tau) = \theta(\tau) \) we use the same rule.
Chapter 2. The Type System

- Assume that \( \vdash F(\tau_1, \ldots, \tau_n) = F(\tau'_1, \ldots, \tau'_n) \) was obtained from \( \vdash \tau_i = \tau'_i \), for \( i \in \{1, \ldots, n\} \), using (Ax2). By the inductive hypothesis, \( \vdash \tau_1 = \theta \tau'_1 \) and \( \vdash \tau_2 = \theta \tau'_2 \), hence we can use the same rule to obtain proof of
  \[
  \vdash F(\theta \tau_1, \ldots, \theta \tau_n) = F(\theta \tau'_1, \ldots, \theta \tau'_n)
  \]
  which, by definition of type substitution, is equivalent to
  \[
  \vdash \theta(F(\tau_1, \ldots, \tau_n)) = \theta(F(\tau'_1, \ldots, \tau'_n)).
  \]

- Assume that \( \vdash \tau_1 \cap \tau_2 = \tau'_1 \cap \tau'_2 \) was obtained from \( \vdash \tau_1 = \tau'_1 \), \( \vdash \tau_2 = \tau'_2 \) using (Ax3). By the inductive hypothesis, \( \vdash \tau_1 = \theta \tau'_1 \) and \( \vdash \tau_2 = \theta \tau'_2 \), hence we can use the same rule to obtain a proof of
  \[
  \vdash \tau_1 \cap \tau_2 = \theta \tau'_1 \cap \theta \tau'_2
  \]
  which, by definition of type substitution, is equivalent to
  \[
  \vdash \theta(\tau_1 \cap \tau_2) = \theta(\tau'_1 \cap \tau'_2).
  \]

- In the case of rule (Ax4) the proof is very similar.

- Assume that \( \vdash \tau'_1 = \tau'_2 \) was obtained from \( \vdash \tau_1 = \tau'_1 \), \( \vdash \tau_2 = \tau'_2 \) and \( \vdash \tau_1 = \tau_2 \) using (Ax5). By the inductive hypothesis, \( \vdash \tau_1 = \theta \tau'_1 \), \( \vdash \tau_2 = \theta \tau'_2 \) and \( \vdash \tau_2 = \theta \tau_2 \). We use the same rule to prove \( \vdash \tau'_1 = \tau'_2 \).

- Assume that \( \vdash \tau \cup \tau = \tau \) was obtained using (Ax6). The same axiom gives \( \vdash \tau \cup \tau = \theta \tau \) which is equivalent to \( \vdash \theta(\tau \cup \tau) = \theta \tau \).

In the case of axioms (Ax7)–(Ax13) proofs are very similar and simple, so we omit them.

- Assume that \( \vdash F(\tau_1, \ldots, \tau_n) \cap G(\tau'_1, \ldots, \tau'_m) = \perp \) was obtained using (Ax14), where \( F \neq G \) and \( F \) or \( G \) is not atomic. We use the same rule obtaining \( \vdash F(\theta \tau_1, \ldots, \theta \tau_n) \cap G(\theta \tau'_1, \ldots, \theta \tau'_m) = \perp \) what is equivalent to \( \vdash \theta(F(\tau_1, \ldots, \tau_n)) \cap \theta(G(\tau'_1, \ldots, \tau'_m)) = \theta(\perp) \).

Since all types occurring in the rules (Ax15) and (Ax16) are atomic, they do not change after applying a substitution \( (\theta \tau = \tau \text{ and } \theta \sigma = \sigma) \). Thus \( \vdash \theta \tau = \theta \sigma \).

**Corollary 2.24.** For any type substitution \( \theta \), for any types \( \tau \) and \( \sigma \)
\[
\vdash \tau \leq \sigma \quad \Rightarrow \quad \vdash \theta(\tau) \leq \theta(\sigma)
\]

Indeed, \( \vdash \tau \leq \sigma \) means that \( \vdash \tau \cap \sigma = \tau \). Applying Lemma 2.23 we obtain \( \vdash \theta(\tau) = \theta(\tau \cap \sigma) = \theta(\tau) \cap \theta(\sigma) \), what gives \( \vdash \theta(\tau) \leq \theta(\sigma) \).

**Definition 2.25.** Let \( \theta \) and \( \theta' \) be type substitutions.

\[
\theta \leq \theta' \quad \text{iff} \quad \text{for each type variable } \alpha, \ \vdash \theta(\alpha) \leq \theta'(\alpha).
\]

**Lemma 2.26.** Let \( \tau \) be a type, let \( \theta, \theta' \) be type substitutions. Then
\[
\theta \leq \theta' \quad \Rightarrow \quad \vdash \theta(\tau) \leq \theta'(\tau).
\]

**Proof.** We proceed by induction on the structure of \( \tau \).
2.3. Properties of Types

• \( \tau = \top \). Then \( \theta(\tau) = \theta'(\tau) = \top \), and hence \( \vdash \theta(\tau) \leq \theta'(\tau) \). When \( \tau = \bot \) the proof is similar.

• \( \tau = \alpha \ (\alpha \in V_T) \). Then, by Definition 2.25, \( \vdash \theta(\alpha) \leq \theta'(\alpha) \).

• \( \tau = F(\tau_1, \ldots, \tau_n) \). By the inductive hypothesis, \( \theta(\tau_1) \leq \theta'(\tau_1) \), for each \( i \in \{1, \ldots, n\} \). This implies, by Lemma 2.22, that
  \[
  \vdash F(\theta(\tau_1), \ldots, \theta(\tau_n)) \leq F(\theta'(\tau_1), \ldots, \theta'(\tau_n))
  \]
what, by definition of type substitution, is equivalent to
  \[
  \vdash \theta(F(\tau_1, \ldots, \tau_n)) \leq \theta'(F(\tau_1, \ldots, \tau_n)).
  \]

• \( \tau = \tau_1 \cap \tau_2 \). By the inductive hypothesis, \( \vdash \theta(\tau_1) \leq \theta'(\tau_1) \) and \( \vdash \theta(\tau_2) \leq \theta'(\tau_2) \), what means that \( \vdash \theta(\tau_1) \cap \theta'(\tau_1) = \theta(\tau_1) \) and \( \vdash \theta(\tau_2) \cap \theta'(\tau_2) = \theta(\tau_2) \). Thus
  \[
  \vdash \theta(\tau_1 \cap \tau_2) = \theta(\tau_1) \cap \theta(\tau_2)
  \]
  \[
  = (\theta(\tau_1) \cap \theta'(\tau_1)) \cap (\theta(\tau_2) \cap \theta'(\tau_2))
  \]
  \[
  = (\theta(\tau_1) \cap \theta(\tau_2)) \cap (\theta'(\tau_1) \cap \theta'(\tau_2))
  \]
  \[
  = \theta(\tau_1 \cap \tau_2) \cap \theta'(\tau_1 \cap \tau_2)
  \]
which means that \( \theta(\tau_1 \cap \tau_2) \leq \theta'(\tau_1 \cap \tau_2) \).

If \( \tau = \tau_1 \cup \tau_2 \) the proof is similar. \( \square \)

**Corollary 2.27.** From Corollary 2.24 and Lemma 2.26 it follows that

if \( \theta \leq \theta' \) and \( \vdash \tau \leq \sigma \) then \( \vdash \theta(\tau) \leq \theta'(\sigma) \).

Indeed, since \( \vdash \tau \leq \sigma \) thus, by corollary 2.24, \( \vdash \theta(\tau) \leq \theta(\sigma) \). On the other hand, \( \theta \leq \theta' \), hence, by Lemma 2.26, \( \vdash \theta(\sigma) \leq \theta'(\sigma) \). Finally, transitivity of \( \leq \) gives \( \vdash \theta(\tau) \leq \theta'(\sigma) \).

2.3.3 Environments

Recall that, for each variable \( X \), there is in environment at most one element of the form \((X : \tau)\) (see Definition 2.9). Therefore an environment can be seen as a function from variables to types, which is formalized in the following definition.

**Definition 2.28.** For an environment \( \Gamma \), and a program variable \( X \), let

\[
\Gamma(X) = \begin{cases} 
\bot & \text{false } \in \Gamma \\
\tau & \text{false } \notin \Gamma \text{ and } (X : \tau) \in \Gamma \\
\top & \text{otherwise}
\end{cases}
\]

\( \square \)

**Lemma 2.29.** For a variable \( X \) and an environment \( \Gamma \),
(a) \( \Gamma \vdash X : \Gamma(X) \),
(b) if \( \Gamma \vdash (X : \tau') \) then \( \Gamma(X) \leq \tau' \).

The proof is moved to Appendix A.1.

**Definition 2.30.** Let \( \Gamma_1 \) and \( \Gamma_2 \) be environments. \( \Gamma_1 \leq \Gamma_2 \) iff

for each term \( t \), and type \( \tau \), \( \Gamma_2 \vdash t : \tau \) then \( \Gamma_1 \vdash t : \tau \).
Lemma 2.31. The relation $\leq$ defined on environments is transitive and reflexive.

Proof. It is clear that $\leq$ is reflexive. Now, assume that $\Gamma_1 \leq \Gamma_2$ and $\Gamma_2 \leq \Gamma_3$. It means that, for each $t$ and $\tau$, $(\Gamma_2 \vdash t : \tau) \Rightarrow (\Gamma_1 \vdash t : \tau)$ and $(\Gamma_3 \vdash t : \tau) \Rightarrow (\Gamma_2 \vdash t : \tau)$. Then obviously, for each $t$ and $\tau$, $(\Gamma_3 \vdash t : \tau) \Rightarrow (\Gamma_1 \vdash t : \tau)$ and hence $\Gamma_1 \leq \Gamma_3$. $\square$

Definition 2.32. Let $\Gamma_1$ and $\Gamma_2$ be environments. Then

$$\Gamma_1 \sim \Gamma_2 \text{ iff } \Gamma_1 \leq \Gamma_2 \text{ and } \Gamma_2 \leq \Gamma_1.$$ 

Corollary 2.33. From Lemma 2.31 it follows that the relation $\sim$ is reflexive, symmetric and transitive.

Lemma 2.34. Let $\Gamma_1$ and $\Gamma_2$ be environments. Then

$$\Gamma_1 \leq \Gamma_2 \text{ iff for each } X \in V, \vdash \Gamma_1(X) \leq \Gamma_2(X).$$

The proof is moved to Appendix A.1.

Corollary 2.35. $\Gamma_1 \sim \Gamma_2$ if and only if, for each variable $X, \vdash \Gamma_1(X) = \Gamma_2(X)$.

Corollary 2.36. If false $\in \Gamma$, then, for each environment $\Gamma', \Gamma \leq \Gamma'$.

Indeed, for each variable $X \in V$, we have $\Gamma(X) = \bot$, and $\vdash \bot \leq \tau$ for each type $\tau$.

2.4 Types of Predicates

2.4.1 Introduction

Definition 2.37. If $\tau_i$ and $\sigma_i$ (for $1 \leq i \leq n$) are types, then $(\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)$ is a directional type.

The left-hand side of such a type is called an input type or an assumption, whereas the right-hand side is called an output type or a guarantee. The intuitive meaning of directional types is as follows: if a predicate $p$ has a type $(\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)$, a term $t_i$ has type $\tau_i$, for $i \in \{1, \ldots, n\}$, and $p(t_1, \ldots, t_n)$ holds, then $t_i$ has type $\sigma_i$, for $i \in \{1, \ldots, n\}$. Sometimes we write $\tau \rightarrow \sigma$ instead of $(\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)$.

In the next section we introduce rules which allow to prove that a predicate has a directional type. Now, let us consider a few informal examples.

Example 2.7. Consider the following predicates and their types.

\[
\begin{align*}
q(X) & : \text{-X=2.} \\
q(X) & : \text{-X=3.14.} \\
r(X,Y,pair(X,Y)) & : r : (\top, \top, \text{prod}(\alpha, \beta)) \rightarrow (\alpha, \beta, \text{prod}(\alpha, \beta))
\end{align*}
\]

The type of the predicate $q$ means that $X$, after successful call of $q(X)$, becomes a real number. The type of the predicate $r$ means: “if we know that the third argument has the type $\text{prod}(\tau, \sigma)$, then after the successful call of $r$ the first argument is of the type $\tau$, while the second is of the type $\sigma$.” $\square$

Example 2.8. A predicate may have many types. Let us consider the program:

\[
\begin{align*}
p(\Omega). \\
\text{unif}(X,X).
\end{align*}
\]
2.4. Types of Predicates

append([], X, X).
append([X|Xs], Ys, [X|Zs]) :- append(Xs, Ys, Zs).

Figure 2.2: The predicate append

Predicate p has the type $\top \rightarrow \text{list}(\bot)$, but, of course, it has also the weaker types $\top \rightarrow \text{list}($real$)$ and $\top \rightarrow \top$. The predicate unif, which unifies its arguments, has the type $(\alpha, \beta) \rightarrow (\alpha \cap \beta, \alpha \cap \beta)$, but it has also the type $(\alpha, \top) \rightarrow (\alpha, \alpha)$.

Example 2.9. Now consider the predicate append (figure 2.2). It has the following types:

\begin{align*}
(2.1) & \quad (\top, \top, \text{list}(\gamma)) \rightarrow (\text{list}(\gamma), \text{list}(\gamma), \text{list}(\gamma)) \\
(2.2) & \quad (\text{list}(\alpha), \text{list}(\beta), \top) \rightarrow (\text{list}(\alpha), \text{list}(\beta), \text{list}(\alpha \cup \beta)) \\
(2.3) & \quad (\text{list}(\alpha), \text{list}(\beta), \text{list}(\gamma)) \rightarrow (\text{list}(\alpha \cap \gamma), \text{list}(\beta \cap \gamma), \text{list}(\gamma \cap (\alpha \cup \beta))) \\
(2.4) & \quad (\top, \text{list}(\beta), \text{list}(\gamma)) \rightarrow (\text{list}(\gamma), \text{list}(\beta \cap \gamma), \text{list}(\gamma)) \\
(2.5) & \quad (\text{list}(\alpha), \top, \text{list}(\gamma)) \rightarrow (\text{list}(\alpha \cap \gamma), \text{list}(\gamma), \text{list}(\gamma))
\end{align*}

Let us observe that different types correspond to different ways the predicate is used. We have: (2.1) corresponds to splitting a list into two parts, (2.2) to appending lists, (2.3) to checking whether two lists joined together are equal to the third one, (2.4) to deleting the suffix of the list, (2.5) to deleting the prefix of the list.

Example 2.9 shows that the existence of many types for one predicate can be very useful. It describes an important feature of logic programming languages: a variety of possible applications of a predicate.

2.4.2 The Predicate Typing Rules

The purpose of this section is to define well-typedness of a program. In order to do it we introduce relation $\Rightarrow$, called the consequence operator, relation InferFromAtoms, and relation ClauseHasType.

The Consequence Operator

Definition 2.38. We define function simpl from types to types:

$$
simpl(\sigma) = \begin{cases} 
\top & \text{if } (\sigma = \top \cup \tau) \text{ or } (\sigma = \top \cap \tau) \\
\tau & \text{if } (\sigma = \tau \cup \top) \text{ or } (\sigma = \tau \cap \top) \\
\sigma & \text{otherwise.}
\end{cases}
$$

As we can see, simpl transforms a type into a simpler form. By (Ax11), $\vdash \sigma = \simpl(\sigma)$. 

**Definition 2.39.** For environments $\Gamma_1$ and $\Gamma_2$

\[
\begin{align*}
\Gamma_1 \cap \Gamma_2 &= \begin{cases}
\{\text{false}\} & \text{if false } \in \Gamma_1 \cup \Gamma_2 \\
\{(X : \tau) \mid \tau = \text{simpl}(\Gamma_1(X) \cap \Gamma_2(X)), \tau \neq \top \} & \text{otherwise}
\end{cases} \\
\Gamma_1 \cup \Gamma_2 &= \begin{cases}
\{\text{false}\} & \text{if false } \in \Gamma_1 \cap \Gamma_2 \\
\{(X : \tau) \mid \tau = \text{simpl}(\Gamma_1(X) \cup \Gamma_2(X)), \tau \neq \top \} & \text{otherwise}
\end{cases}
\end{align*}
\]

**Example 2.10.** Let $\Gamma_1 = \{(X : \text{int}), (Y : \top)\}$, and $\Gamma_2 = \{(X : \text{real}), (Z : \text{real})\}$. Then

\[
\begin{align*}
\Gamma_1 \cap \Gamma_2 &= \{(X : \text{int} \cap \text{real}), (Z : \text{real})\}, \quad \text{and} \\
\Gamma_1 \cup \Gamma_2 &= \{(X : \text{int} \cup \text{real})\}.
\end{align*}
\]

In particular, there is no element of the form $(Y : \tau)$ in $\Gamma_1 \cap \Gamma_2$ since $\text{simpl}(\Gamma_1(Y) \cap \Gamma_2(Y)) = \text{simpl}(\top \cap \top) = \top$. □

**Definition 2.40.** Let $\tau$ be a type. $\mathbb{N}(\tau)$ is the type obtained from $\tau$ by substituting each occurrence of $\bot$ by a distinct variable not belonging to $\text{var}(\tau)$. □

**Example 2.11.**

\begin{align*}
(2.6) \quad \mathbb{N}(\text{list}(\bot)) &= \text{list}(\alpha) \\
&= \mathbb{N}(F(\alpha_1, \bot, G(\bot, \alpha_2))) = F(\alpha_1, \alpha_3, G(\alpha_4, \alpha_2)).
\end{align*}

□

We shall apply $\mathbb{N}$ to the right-hand sides of signatures in order to obtain a form without $\bot$. Such a form is in some situations more convenient.

In Figure 2.3 we present rules which allow to deduce some new information from facts of the form $(t : \tau)$ and accumulate them in an environment. In order to do that, we introduce a new three argument relation $\Rightarrow$. If $\Gamma$ and $\Gamma'$ are environments, $\varphi$ has the form $(t_1 : \tau_1, \ldots, t_n : \tau_n)$ then $\Gamma, \varphi \Rightarrow \Gamma'$ means that $\Gamma$ and $\varphi$ imply $\Gamma'$. The consequence rules are constructed in such a way that the choice of a rule to be applied is determined by $t$ and $\tau$. Therefore, we have $(K_5)$ and $(K_6)$ instead of a more general rule $\Gamma, (t : \tau) \Rightarrow \Gamma$.

**Example 2.12.** Rule $(K_9)$ allows to prove that $\varnothing, (X : \text{int}) \Rightarrow \varnothing \cap \{X : \text{int}\}$ which, by the definition of $\cap$, is equivalent to $\varnothing, (X : \text{int}) \Rightarrow \{X : \text{int}\}$. □

**Example 2.13.** If we assume that $\text{int} \leq \text{real}$, rules at Figure 2.3 allow to prove that

\begin{align*}
(2.7) \quad \{X : \text{real}\}, (X : \text{int}) \Rightarrow \Gamma', \quad \text{where } \Gamma' \sim \{X : \text{int}, Y : \text{list(int)}\}
\end{align*}

In fact, since $[1]$ has signature $\alpha \ast \text{list}(\alpha) \rightarrow \text{list}(\alpha)$,

\[
\{X : \text{real}\}, (X : \text{int}) \Rightarrow \Gamma_1, \quad \{X : \text{real}\}, (Y : \text{list(int)}) \Rightarrow \Gamma_2
\]

\[
\{X : \text{real}\}, (X : \text{int}) \Rightarrow \Gamma_1 \cap \Gamma_2
\]

is an instance of rule $(K_2)$, for $\theta = \{\alpha / \text{int}\}$. It is easy to check that the assumptions hold for $\Gamma_1 = \{X : \text{real} \cap \text{int}\}$, and $\Gamma_2 = \{X : \text{real}, Y : \text{list(int)}\}$. Thus $\Gamma' = \Gamma_1 \cap \Gamma_2 = \{X : \text{real} \cap \text{int} \cap \text{real}, Y : \text{list(int)}\}$ which gives (2.7). □
2.4. Types of Predicates

### Figure 2.3: Consequence rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Premise</th>
<th>Conclusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(K_1)$</td>
<td>$\Gamma, (t_i : \tau_i) \Rightarrow \Gamma_i \quad (1 \leq i \leq n)$</td>
<td>$\Gamma, (t_1 : \tau_1, \ldots, t_n : \tau_n) \Rightarrow \Gamma_1 \cap \cdots \cap \Gamma_n$</td>
</tr>
<tr>
<td>$(K_2)$</td>
<td>$\Gamma, (t_i : \theta(\tau_i)) \Rightarrow \Gamma_i \quad (1 \leq i \leq n)$</td>
<td>$\Gamma, (f(t_1, \ldots, t_n) : \theta(\theta(\tau))) \Rightarrow \Gamma_1 \cap \cdots \cap \Gamma_n$ if $f : \tau_1 \cdots \tau_n \rightarrow \tau$</td>
</tr>
<tr>
<td>$(K_3)$</td>
<td></td>
<td>$\Gamma, (f(t_1, \ldots, t_n) : \tau) \Rightarrow {\text{false}}$ if there is no signature $\tau_1 \cdots \tau_n \rightarrow \tau_0$ assigned to $f$ such that $\text{head}(\tau) = \text{head}(\tau_0)$</td>
</tr>
<tr>
<td>$(K_4)$</td>
<td></td>
<td>$\Gamma, (t : \perp) \Rightarrow {\text{false}}$</td>
</tr>
<tr>
<td>$(K_5)$</td>
<td></td>
<td>$\Gamma, (t : \top) \Rightarrow \Gamma$</td>
</tr>
<tr>
<td>$(K_6)$</td>
<td>$\Gamma, (f(t_1, \ldots, t_n) : \alpha) \Rightarrow \Gamma$</td>
<td></td>
</tr>
<tr>
<td>$(K_7)$</td>
<td>$\Gamma, (t : \tau_i) \Rightarrow \Gamma_i \quad (1 \leq i \leq n)$</td>
<td>$\Gamma, (t : \tau_1 \cap \cdots \cap \tau_n) \Rightarrow \Gamma_1 \cap \cdots \cap \Gamma_n$</td>
</tr>
<tr>
<td>$(K_8)$</td>
<td>$\Gamma, (t : \tau_i) \Rightarrow \Gamma_i \quad (1 \leq i \leq n)$</td>
<td>$\Gamma, (t : \tau_1 \cup \cdots \cup \tau_n) \Rightarrow \Gamma_1 \cup \cdots \cup \Gamma_n$</td>
</tr>
<tr>
<td>$(K_9)$</td>
<td></td>
<td>$\Gamma, (X : \tau) \Rightarrow \Gamma \cap {X : \tau}$ if $\tau \neq \perp$ and $\tau \neq \top$</td>
</tr>
</tbody>
</table>

### Directional Type of a Program

**Definition 2.41.** A *directional type of a program* is a set

$$\mathcal{T} \subseteq \{(p : \tau \rightarrow \sigma) \mid p \text{ is a predicate name, } \tau \rightarrow \sigma \text{ is a directional type}\}.$$  

Note that there can be many types for one predicate in a directional type of a program.

In Figure 2.4 we give rules for two new relations: InferFromAtoms and Clause-HasType. These relations will be used in the definition of well-typedness. The relation

$$\text{InferFromAtoms}(\mathcal{T}, \Gamma, \langle a_1, \ldots, a_n \rangle, \Gamma')$$

is defined for a directional type $\mathcal{T}$ of a program, environments $\Gamma, \Gamma'$ and a sequence $\langle a_1, \ldots, a_n \rangle$ of atoms. ( ) in the rule $(P_1)$ represents the empty sequence. The intended meaning of $\text{InferFromAtoms}(\mathcal{T}, \Gamma, \langle a_1, \ldots, a_n \rangle, \Gamma')$ is “if predicates have types described by $\mathcal{T}$, variables have types described by $\Gamma$, then, after execution of the sequence $\langle a_1, \ldots, a_n \rangle$ of atoms, variables get types described by $\Gamma'$.”

**Example 2.14.** Assume that $\mathcal{T}$ contains

$$\text{append} : (\mathcal{T}, \mathcal{T}, \text{list}(\text{int})) \rightarrow (\text{list}(\text{int}), \text{list}(\text{int}), \text{list}(\text{int})).$$

Let $\Gamma = \{L : \text{list(\text{int})}\}$. Then

$$\vdash \text{InferFromAtoms}(\mathcal{T}, \Gamma, \text{append}(A, B, L), \Gamma')$$
\[(P_1)\]
\[\vdash \text{InferFromAtoms}(T, \Gamma, \langle \rangle, \Gamma)\]
\[\Gamma \vdash t : \theta(\tau), \quad \Gamma, (t_1 : \theta(\sigma_1), \ldots , t_n : \theta(\sigma_n)) \Rightarrow \Gamma' ,\]
\[\vdash \text{InferFromAtoms}(T, \Gamma', (a_2, \ldots , a_k), \Gamma'' )\]
\[\vdash \text{InferFromAtoms}(T, \Gamma, \langle p(t_1, \ldots , t_n), a_2, \ldots , a_k \rangle, \Gamma'' )\]
\[\text{if } (p : (\tau_1, \ldots , \tau_n) \rightarrow (\sigma_1, \ldots , \sigma_n)) \in T,\]
\[(P_3)\]
\[\varnothing, (t_1 : \tau_1, \ldots , t_n : \tau_n) \Rightarrow \Gamma_1 ,\]
\[\vdash \text{InferFromAtoms}(T, \Gamma_1, B, \Gamma_2) ,\quad \Gamma_2 \vdash t_1 : \sigma_1, \ldots , \Gamma_2 \vdash t_n : \sigma_n\]
\[\vdash \text{ClauseHasType}(T, p(t_1, \ldots , t_n) : B, (\tau_1, \ldots , \tau_n) \rightarrow (\sigma_1, \ldots , \sigma_n))\]

Figure 2.4: Clause typing rules

holds for \(\Gamma' = \{ A : \text{list}(\text{int}), B : \text{list}(\text{int}), L : \text{list}(\text{int}) \} \). In fact, if \(L\) is a list of integers, then, after executing \(\text{append}(A, B, L)\), \(A\) and \(B\) also become lists of integers.

The relation \(\text{ClauseHasType}(T, C, \tau \rightarrow \sigma)\) is defined for a directional type \(T\) of a program, a clause \(C\), and a directional type \(\tau \rightarrow \sigma\). \(B\) in the rule \((P_3)\) denotes the body of a clause, i.e. a sequence of atoms. The intended meaning of \(\text{ClauseHasType}(T, C, \tau \rightarrow \sigma)\) is “if the clause \(C\) is executed with an argument of the type \(\tau\), then after execution the argument gets the type \(\sigma\).” We can treat rule \((P_3)\) as a description of an algorithm working in three steps. First, information about variables is inferred from the fact that argument has type \(\tau\). Second, \(\text{InferFromAtoms}\) is performed to obtain types of variables after executing the body of the clause. And finally, guarantees are checked, i.e. it is proved that argument of the predicate has type \(\sigma\).

**Example 2.15.** Let \(T = \{ \text{append} : w \}\) where
\[w = (\text{list}(\alpha), \text{list}(\beta), \text{list}(\gamma)) \rightarrow (\text{list}(\alpha \cap \gamma), \text{list}(\beta \cap \gamma), \text{list}(\gamma \cap (\alpha \cup \beta))).\]
Let \(C = (\text{append} (\text{X} \text{[} A \text{]} , B , \text{X} \text{[} C \text{]} ) : - \text{ append}(A, B, C) \). One can show that \(\vdash \text{ClauseHasType}(T, C, w)\) holds. Indeed, by \((P_3)\), it suffices to show that there exists \(\Gamma_1, \Gamma_2\) such that
\[(2.8)\]
\[\varnothing, ([X[A] : \text{list}(\alpha), B : \text{list}(\beta), [X[C] : \text{list}(\gamma)]) \Rightarrow \Gamma_1 ,\]
\[(2.9)\]
\[\vdash \text{InferFromAtoms}(T, \Gamma_1, (\text{append}(A, B, C)), \Gamma_2)\]
\[(2.10)\]
\[\Gamma_2 \vdash [X[A] : \text{list}(\alpha \cap \gamma),\]
\[(2.11)\]
\[\Gamma_2 \vdash B : \text{list}(\beta \cap \gamma),\]
\[(2.12)\]
\[\Gamma_2 \vdash [X[C] : \text{list}(\gamma \cap (\alpha \cup \beta)).\]

It is easy to check that \((2.8)\) holds for
\[\Gamma_1 \sim \{ X : \alpha \cap \gamma, A : \text{list}(\alpha), B : \text{list}(\beta), C : \text{list}(\gamma) \} .\]
Moreover, one can show that \((2.9)\) holds for
\[\Gamma_2 \sim \{ X : \alpha \cap \gamma, A : \text{list}(\alpha \cap \gamma), B : \text{list}(\beta \cap \gamma), C : \text{list}(\gamma \cap (\alpha \cup \beta)) \} .\]
Hence, one can check that \((2.10)-(2.12)\) holds.

Now, we give the definition of **well-typedness** of a program with respect to a directional type of a program.

22
Definition 2.42. Let $P$ be a program consisting of clauses $\{C_i\}_{i=1}^n$. Let $h(C_i)$ be the main head symbol of $C_i$. $P$ is well-typed with respect to a directional type $\mathcal{T}$ iff

$$\vdash \text{ClauseHasType}(\mathcal{T}, C_i, \tau \rightarrow \sigma).$$

for each $i \in \{1, \ldots, n\}$, and for each directional type $(\tau \rightarrow \sigma)$ such that $(h(C_i) : \tau \rightarrow \sigma) \in \mathcal{T}$. \hfill \Box

Example 2.16. Let us consider a program $P$ of the form

\begin{align*}
P(1), \\
P(2), \\
q(X, Y) &\colon p(X), X=Y.
\end{align*}

Let $\mathcal{T} = \{(p : \top \rightarrow \text{int}), (q : (\top, \top) \rightarrow (\top, \text{int})), (q : (\top, \top) \rightarrow (\text{int}, \text{int}))\}$. To prove that $P$ is well-typed with respect to $\mathcal{T}$ it suffices to show that

\begin{align*}
\vdash \text{ClauseHasType}(\mathcal{T}, P(1), \top \rightarrow \text{int}), \\
\vdash \text{ClauseHasType}(\mathcal{T}, P(2), \top \rightarrow \text{int}), \\
\vdash \text{ClauseHasType}(\mathcal{T}, q(X, Y) \colon p(X), X=Y), (\top, \top) \rightarrow (\top, \text{int})), \\
\vdash \text{ClauseHasType}(\mathcal{T}, q(X, Y) \colon p(X), X=Y), (\top, \top) \rightarrow (\text{int}, \text{int})).
\end{align*}

\hfill \Box

If a program $P$, and a directional type $\mathcal{T}$ of a program are fixed, and if $P$ is well-typed with respect to $\mathcal{T}$, then we write that a predicate $p$ of $P$ has a type $(\tau \rightarrow \sigma)$ if $(p : \tau \rightarrow \sigma) \in \mathcal{T}$.

2.5 Technical Lemmas

In this section we give a few definitions and provide some facts useful in the proofs given in the next sections.

2.5.1 Environments

Lemma 2.43. For environments $\Gamma$ and $\Gamma'$ the following conditions holds for each variable $X$:

(a) $(\Gamma \cap \Gamma')(X) = \Gamma(X) \cap \Gamma'(X)$,
(b) $(\Gamma \cup \Gamma')(X) = \Gamma(X) \cup \Gamma'(X)$.

The proof is moved to Appendix A.1.

Lemma 2.44. Assume that $\Gamma$, $\Gamma'$ and $\Gamma''$ are environments. Then

(a) $\Gamma \sim \Gamma$,
(b) $\Gamma \cap \Gamma \sim \Gamma$,
(c) $\Gamma \cup \Gamma \sim \Gamma$,
(d) $\Gamma \cap \Gamma' \sim (\Gamma \cap \Gamma') \cap \Gamma'$,
(e) $\Gamma \cup \Gamma' \sim (\Gamma \cup \Gamma') \cup \Gamma'$,
(f) $\Gamma \cap (\Gamma \cap \Gamma') \sim \Gamma$,
(g) $\Gamma \cup (\Gamma \cap \Gamma') \sim \Gamma$,
(h) $\Gamma \cap (\Gamma' \cap \Gamma'') \sim (\Gamma \cap \Gamma') \cap (\Gamma \cap \Gamma'')$,
(i) $\Gamma \cup (\Gamma' \cap \Gamma'') \sim (\Gamma \cup \Gamma') \cup (\Gamma \cap \Gamma'')$,
(j) $\Gamma \cap (\Gamma' \cup \Gamma'') \sim (\Gamma \cap \Gamma') \cup (\Gamma \cap \Gamma'')$,
(k) $\Gamma \cup (\Gamma' \cup \Gamma'') \sim (\Gamma \cup \Gamma') \cap (\Gamma \cup \Gamma'')$. 

23
(l) \( \Gamma_1 \sim \Gamma'_1 \), and \( \Gamma_2 \sim \Gamma'_2 \) then \( \Gamma_1 \sqcap \Gamma_2 \sim \Gamma'_1 \sqcap \Gamma'_2 \).
(m) \( \Gamma_1 \sim \Gamma'_1 \), and \( \Gamma_2 \sim \Gamma'_2 \) then \( \Gamma_1 \sqcup \Gamma_2 \sim \Gamma'_1 \sqcup \Gamma'_2 \).

**Proof.** Proofs in all cases are simple consequences of Lemma 2.43 and properties of types. So, we present only the proof of (k). We want to show that \( \Gamma \sqcap (\Gamma' \sqcap \Gamma'') \sim (\Gamma \sqcap \Gamma') \sqcap (\Gamma \sqcap \Gamma'') \). By Corollary 2.35, it suffices to show that, for each variable \( X \),
\[
\vdash L = (\Gamma \sqcup (\Gamma' \sqcap \Gamma''))(X) = ((\Gamma \sqcup \Gamma') \sqcap (\Gamma \sqcup \Gamma''))(X) = R
\]
which is true because
\[
\begin{align*}
\vdash L &= (\Gamma \sqcup (\Gamma' \sqcap \Gamma''))(X) \\
&= \Gamma(X) \cup (\Gamma'(X) \cap \Gamma''(X)) & \text{by Lemma 2.43} \\
&= (\Gamma(X) \cup \Gamma'(X)) \cap (\Gamma(X) \cup \Gamma''(X)) & \text{(Ax10)} \\
&= ((\Gamma \sqcup \Gamma') \cap (\Gamma \sqcup \Gamma''))(X) & \text{by Lemma 2.43} \\
&= R.
\end{align*}
\]

**Corollary 2.45.** Points (l) and (m) of the previous lemma, imply that
if \( \Gamma_1 \sim \Gamma'_1, \ldots, \Gamma_n \sim \Gamma'_n \) then \( (\Gamma_1 \sqcap \cdots \sqcap \Gamma_n) \sim (\Gamma'_1 \sqcap \cdots \sqcap \Gamma'_n) \),
if \( \Gamma_1 \sim \Gamma'_1, \ldots, \Gamma_n \sim \Gamma'_n \) then \( (\Gamma_1 \sqcup \cdots \sqcup \Gamma_n) \sim (\Gamma'_1 \sqcup \cdots \sqcup \Gamma'_n) \).

**Lemma 2.46.** Let \( \Gamma_1 \) and \( \Gamma_2 \) are environments. Then the following statements are equivalent
1. \( \Gamma_1 \leq \Gamma_2 \),
2. \( \Gamma_1 \sqcap \Gamma_2 \sim \Gamma_1 \),
3. \( \Gamma_1 \sqcup \Gamma_2 \sim \Gamma_2 \).

**Proof.** (1) \( \Rightarrow \) (2) For each variable \( X \), by Lemma 2.43, we have
\[
\vdash (\Gamma_1 \sqcap \Gamma_2)(X) = \Gamma_1(X) \cap \Gamma_2(X).
\]
From (1) and lemma (2.34) it follows that, \( \Gamma_1(X) \leq \Gamma_2(X) \), and therefore \( \Gamma_1(X) \cap \Gamma_2(X) = \Gamma_1(X) \), and thus \( \vdash (\Gamma_1 \sqcap \Gamma_2)(X) = \Gamma_1(X) \) which, by corollary 2.35, gives (2).

(2) \( \Rightarrow \) (1) For each variable \( X \), by Lemma 2.43, we have
\[
\vdash \Gamma_1(X) \cap \Gamma_2(X) = (\Gamma_1 \sqcap \Gamma_2)(X).
\]
From (2) and corollary 2.35 it follows that \( \vdash (\Gamma_1 \sqcap \Gamma_2)(X) = \Gamma_1(X) \), for each \( X \). Thus \( \vdash \Gamma_1(X) \cap \Gamma_2(X) = \Gamma_1(X) \) which proves that \( \Gamma_1(X) \leq \Gamma_2(X) \). Hence, by lemma (2.34), we have (1).

The proofs for (1) \( \Rightarrow \) (3) and (3) \( \Rightarrow \) (1) are similar.

**Corollary 2.47.** From Lemma 2.46 it follows that
if \( \Gamma_1 \leq \Gamma'_1, \ldots, \Gamma_n \leq \Gamma'_n \) then \( (\Gamma_1 \sqcap \cdots \sqcap \Gamma_n) \leq (\Gamma'_1 \sqcap \cdots \sqcap \Gamma'_n) \),
if \( \Gamma_1 \leq \Gamma'_1, \ldots, \Gamma_n \leq \Gamma'_n \) then \( (\Gamma_1 \sqcup \cdots \sqcup \Gamma_n) \leq (\Gamma'_1 \sqcup \cdots \sqcup \Gamma'_n) \).

Indeed, suppose \( \Gamma_i \leq \Gamma'_i \), for \( i \in \{1, \ldots, n\} \). By Lemma 2.46, \( \Gamma_i \sqcap \Gamma'_i \sim \Gamma_i \). Hence, by Lemma 2.44,
\[
(\Gamma_1 \sqcap \cdots \sqcap \Gamma_n) \sqcap (\Gamma'_1 \sqcap \cdots \sqcap \Gamma'_n) \sim (\Gamma_1 \sqcap \Gamma'_1) \sqcap \cdots \sqcap (\Gamma_n \sqcap \Gamma'_n)
\]
which, by Lemma 2.46, gives \( (\Gamma_1 \sqcap \cdots \sqcap \Gamma_n) \leq (\Gamma'_1 \sqcap \cdots \sqcap \Gamma'_n) \). In the same way we can prove the second implication.
2.5.2 Consequence Operator

Lemma 2.48. For any environment $\Gamma$, any term $t$ and any type $\tau$ there exists exactly one environment $\Gamma'$ such that $\Gamma, (t : \tau) \Rightarrow \Gamma'$.

Proof. It is easy to see that, for each $t$ and $\tau$, exactly one of the rules $(K_1)$, $(K_2)$ can be used.

Lemma 2.48 states that we can treat the $\Rightarrow$ relation as a function which for the first and the second arguments ($\Gamma$ and $(t : \tau)$) gives the third argument ($\Gamma'$).

Lemma 2.49. If $\Gamma, \varphi \Rightarrow \Gamma'$ then $\Gamma' \leq \Gamma$.

Proof. We proceed by induction on the structure of the proof of $\Gamma, \varphi \Rightarrow \Gamma'$. The proofs in the cases of $(K_1)$, $(K_2)$ and $(K_7)$ are simple and similar. Therefore we present only the proof for $(K_2)$. The cases of $(K_3)$, $(K_4)$, $(K_5)$ and $(K_6)$ are trivial and we omit proofs for them.

• Assume that $\Gamma, (f(t_1, \ldots, t_n) : \theta(\bar{t}(\tau))) \Rightarrow \Gamma_1 \sqcup \cdots \sqcup \Gamma_n$ was obtained from

\[
\Gamma, (t : \theta(\tau_1)) \Rightarrow \Gamma_1, \ldots, \Gamma, (t : \theta(\tau_n)) \Rightarrow \Gamma_n
\]

using $(K_2)$, where $f : \tau_1 \ast \cdots \ast \tau_n \Rightarrow \tau$. We want to show that

\[(2.13) \quad \Gamma_1 \sqcup \cdots \sqcup \Gamma_n \leq \Gamma.\]

By the inductive hypothesis, we have $\Gamma_i \leq \Gamma$, for $i \in \{1, \ldots, n\}$. By Lemma 2.46, we have $\Gamma_i \sqcup \Gamma \sim \Gamma_i$. Hence we use Lemma 2.44 to obtain

\[
\Gamma \sqcup (\Gamma_1 \sqcup \cdots \sqcup \Gamma_n) \sim ((\Gamma \sqcup \Gamma_1) \sqcup \cdots \sqcup (\Gamma \sqcup \Gamma_n)) \sim (\Gamma_1 \sqcup \cdots \sqcup \Gamma_n).
\]

It, by Lemma 2.46, gives (2.13).

• Assume that $\Gamma, (t : \tau_1 \cup \cdots \cup \tau_n) \Rightarrow \Gamma_1 \sqcup \cdots \sqcup \Gamma_n$ was obtained from

\[
\Gamma, (t : \tau_1) \Rightarrow \Gamma_1, \ldots, \Gamma, (t : \tau_n) \Rightarrow \Gamma_n
\]

using $(K_8)$. We want to show that

\[(2.14) \quad \Gamma_1 \sqcup \cdots \sqcup \Gamma_n \leq \Gamma.\]

By the inductive hypothesis, we have $\Gamma_i \leq \Gamma$, for $i \in \{1, \ldots, n\}$. By Lemma 2.46, we have $\Gamma_i \sqcup \Gamma \sim \Gamma_i$. Hence we use Lemma 2.44 to obtain

\[
\Gamma \sqcup (\Gamma_1 \sqcup \cdots \sqcup \Gamma_n) \sim ((\Gamma \sqcup \Gamma_1) \sqcup \cdots \sqcup (\Gamma \sqcup \Gamma_n)) \sim (\Gamma_1 \sqcup \cdots \sqcup \Gamma) \sim \Gamma.
\]

It, by Lemma 2.46, gives (2.14).

Lemma 2.50. If $\vdash \tau_1 = \tau_2$, $\Gamma_1 \sim \Gamma_2$, $\Gamma_1, (t : \tau_1) \Rightarrow \Gamma_1'$ and $\Gamma_2, (t : \tau_2) \Rightarrow \Gamma_2'$, then $\Gamma_1' \sim \Gamma_2'$.

The proof of this lemma is moved to Appendix A.1.

Lemma 2.51. If $\Gamma_1 \leq \Gamma_2$, $\Gamma_1, \varphi \Rightarrow \Gamma_1'$ and $\Gamma_2, \varphi \Rightarrow \Gamma_2'$, then $\Gamma_1' \leq \Gamma_2'$.

The proof is moved to Appendix A.1.
2.6 Intersection Lemma

Our system has the following property: if a predicate has a type \( \tau \rightarrow \sigma \) and a type \( \tau' \rightarrow \sigma' \) then it has also the type \( \tau \cap \tau' \rightarrow \sigma \cap \sigma' \). We formalize this property below.

**Definition 2.52.** A directional type \( T \) of a program is non-conflicting if, for each pair \((p_1 : \tau_1 \rightarrow \sigma_1), (p_2 : \tau_2 \rightarrow \sigma_2)\) of distinct elements of \( T \), we have \( \text{var}(\tau_1, \sigma_1) \cap \text{var}(\tau_2, \sigma_2) = \emptyset \).

**Lemma 2.53 (Intersection Lemma).** Assume that a program \( P \) is well typed with respect to \( T \), \( T \) is non-conflicting, and

\[
T' = T \cup \{(p : \tau_1 \cap \tau_2 \rightarrow \sigma_1 \cap \sigma_2) \mid (p : \tau_1 \rightarrow \sigma_1) \in T, (p : \tau_2 \rightarrow \sigma_2) \in T\}.
\]

Then the program \( P \) is also well typed with respect to \( T' \).

There are two reasons why Intersection Lemma is important. First, it is one of the methods which, for directional types of a program (of a predicate), gives another directional type. Second, the type-checking algorithm applied to a directional type \( T \) of a program can deterministically choose types for atoms (see rule \((P_2)\)) only if \( T \) is closed under intersection. By Intersection Lemma, this requirement is easy to satisfy, since for every family of types we can generate its closure with respect to intersection.

Intersection Lemma assumes that a directional type of a program is non-conflicting. Now, we show that this requirement is not inconvenient, since, for every directional type of a program, we can find an equivalent non-conflicting type.

**Lemma 2.54 (Renaming Lemma).** Assume that a program \( P \) is well typed with respect to \( T = \{p^i : \tau^i \rightarrow \sigma^i\}_{i=1}^n \), and, for \( i \in \{1, \ldots, n\} \), \( \theta_i \) is a renaming substitution, i.e. \( \theta_i = (\alpha_i^1 / \beta_1, \ldots, \alpha_i^j / \beta_m) \) where \( \beta_1, \ldots, \beta_m \) are distinct variables. Then the program \( P \) is also well typed with respect to

\[
T' = \{p^i : \theta_i(\tau^i) \rightarrow \theta_i(\sigma^i)\}_{i=1}^n.
\]

**Proof.** Suppose that \( C \) is a clause of the program \( P \) with the head predicate symbol \( p \), and \( (p : \tau \rightarrow \sigma) \in T' \). In order to prove the lemma we should show that

\[ \vdash \text{ClauseHasType}(T', C, \tau \rightarrow \sigma). \]

From definition of \( T' \) it follows that \( \tau = \theta_j(\tau^j) \), \( \sigma = \theta_j(\sigma^j) \) and \( p = p^j \), for some \( j \in \{1, \ldots, n\} \). Since \( (p^j : \tau^j \rightarrow \sigma^j) \in T \), and \( P \) is well typed with respect to \( T \), there exists a proof \((*)\) of

\[ \vdash \text{ClauseHasType}(T, C, \tau \rightarrow \sigma). \]

Let us substitute each occurrence of \( T \) in this proof by \( T' \). Moreover, let us take each instance

\[
a_i = p^i(t_1, \ldots, t_n), \quad (p^i : (\tau^i_1, \ldots, \tau^i_n) \rightarrow (\sigma^i_1, \ldots, \sigma^i_n)) \in T,
\]

\[
\Gamma \vdash t_i : \theta_i(\tau^i_i), \quad \Gamma, (t_1 : \theta_i(\tau^i_1), \ldots, t_n : \theta_i(\tau^i_n)) \Rightarrow \Gamma'.
\]

\[ \vdash \text{InferFromAtoms}(T, \Gamma', (a_2, \ldots, a_k), \Gamma'') \]

\[ \vdash \text{InferFromAtoms}(T, \Gamma, (a_1, \ldots, a_k), \Gamma'') \]

26
of rule \((P_2)\) in \((*)\), and substitute it by
\[
\begin{align*}
\nu_1 &= p^j(t_1, \ldots, t_n), \\
(p^j : (\theta_i(\tau^j_1), \ldots, \theta_i(\tau^j_n)) \rightarrow (\theta_i(\sigma^j_1), \ldots, \theta_i(\sigma^j_n))) &\in T', \\
\Gamma \vdash t_i : (\theta \circ \theta^{-1}_i)(\theta_i(\tau^j_1)), \\
\Gamma, (t_1 : (\theta \circ \theta^{-1}_i)(\theta_i(\sigma^j_1)), \ldots, t_n : (\theta \circ \theta^{-1}_i)(\theta_i(\sigma^j_n))) &\Rightarrow \Gamma' \\
\vdash \text{InferFromAtoms}(T', \Gamma', \langle a_2, \ldots, a_k \rangle, \Gamma'') \\
\end{align*}
\]
\[
\vdash \text{InferFromAtoms}(T', \Gamma, \langle a_1, \ldots, a_k \rangle, \Gamma'')
\]

One can see that we have obtained a correct tree of the proof of
\[
\vdash \text{ClauseHasType}(T', C, \tau \rightarrow \sigma).
\]

This is because
\[
(p^j : (\theta_i(\tau^j_1), \ldots, \theta_i(\tau^j_n)) \rightarrow (\theta_i(\sigma^j_1), \ldots, \theta_i(\sigma^j_n))) \in T',
\]
from definition of \(T'\). Moreover \((\theta \circ \theta^{-1}_i)(\theta_i(\tau^j_1)) = \theta(\tau^j_1)\) and \((\theta \circ \theta^{-1}_i)(\theta_i(\sigma^j_1)) = \theta(\sigma^j_1)\),
and thus the other assumptions of the obtained rules are equal to assumptions in \((*)\).

\[\Box\]

Corollary 2.55. Assume that we have a directional type \(T\) of a program. In order to obtain a non-conflicting equivalent type \(T'\) we apply the Lemma 2.54 with a family \(\{\theta_1, \ldots, \theta_n\}\) of renaming substitutions such that \(\text{range}(\theta_i) \cap \text{range}(\theta_j) = \emptyset\), for \(i \neq j\).

The rest of the section presents the proof of intersection lemma.

The Proof of Intersection Lemma

Lemma 2.56. Consider a program \(P\) with a clause \(C\). Assume that \(T\) and \(T'\) are directional types of a program such that \(T \subseteq T'\).

\[\text{if } \vdash \text{ClauseHasType}(T, C, \tau \rightarrow \sigma) \text{ then } \vdash \text{ClauseHasType}(T', C, \tau \rightarrow \sigma).\]

Proof. Suppose that \(\vdash \text{ClauseHasType}(T, C, \tau \rightarrow \sigma)\), i.e. there exists the tree of a
proof with the root \(\vdash \text{ClauseHasType}(T, C, \tau \rightarrow \sigma)\). Let us substitute each occurrence of \(T\) in this tree by \(T'\). The obtained tree has the root
\[
\vdash \text{ClauseHasType}(T', C, \tau \rightarrow \sigma),
\]
and it is a correct tree of a proof. This is because, in the original proof, \(T\) is used only
in conditions of the form \((p : \tau \rightarrow \sigma) \in T\) in the rule \((P_2)\). So, if such a conditions is satisfied, all the more \((p : \tau \rightarrow \sigma) \in T'\) (we have assumed \(T \subseteq T'\)).

\[\Box\]

Lemma 2.57. Suppose that
\[
\begin{align*}
(2.15) & \quad \Gamma \leq \Gamma_1, \\
(2.16) & \quad \Gamma \leq \Gamma_2, \\
(2.17) & \quad \Gamma_1, (t : \tau_1) \Rightarrow \Gamma'_1, \\
(2.18) & \quad \Gamma_2, (t : \tau_2) \Rightarrow \Gamma'_2, \\
(2.19) & \quad \Gamma, (t : \tau_1 \cap \tau_2) \Rightarrow \Gamma'.
\end{align*}
\]

Then \(\Gamma' \leq \Gamma'_1\) and \(\Gamma' \leq \Gamma'_2\).
Chapter 2. The Type System

Proof. (2.19) can be proved only using \((K_7)\) from

\[
\begin{align*}
\Gamma, (t : r_1) &\Rightarrow \Gamma'' \\
\Gamma, (t : r_2) &\Rightarrow \Gamma'' \\
\Gamma' &= \Gamma'' \cap \Gamma'''.
\end{align*}
\]

By Lemma 2.51, from (2.15), (2.17) and (2.20) it follows that \(\Gamma'' \leq \Gamma_1\). Similarly
(2.16), (2.18) and (2.21) imply \(\Gamma''' \leq \Gamma_2\). Thus, by corollary 2.47,

\[
\Gamma' = \Gamma'' \cap \Gamma''' \leq \Gamma_1' \cap \Gamma_2' .
\]

Furthermore, by Lemma 2.44,

\[
(\Gamma_1' \cap \Gamma_2') \cap \Gamma_1 \sim (\Gamma_1' \cap \Gamma_1') \cap \Gamma_2 \sim \Gamma_1' \cap \Gamma_2'
\]

which means, by Lemma 2.46, that \(\Gamma_1' \cap \Gamma_2' \leq \Gamma_1'\). In the same way we show that \(\Gamma_1' \cap \Gamma_2' \leq \Gamma_2'\). Thus, by transitivity of \(\leq\), \(\Gamma'' \leq \Gamma_1'\) and \(\Gamma'' \leq \Gamma_2'\).

Lemma 2.58. Assume that \(\Gamma_1 \leq \Gamma_2\) and \(\vdash \tau_1 \leq \tau_2\). Let \(\Gamma_1, (t : \tau_1) \Rightarrow \Gamma_1'\) and \(\Gamma_2, (t : \tau_2) \Rightarrow \Gamma_2'\). Then \(\Gamma_1' \leq \Gamma_2'\).

Proof. Since \(\vdash \tau_1 \leq \tau_2\), thus \(\vdash \tau_1 \cap \tau_2 = \tau_1\). We have assumed that \(\Gamma_1, (t : \tau_1) \Rightarrow \Gamma_1'\).
Let \(\Gamma_1, (t : \tau_1 \cap \tau_2) \Rightarrow \Gamma_1''\). By Lemma 2.50, \(\Gamma_1' \sim \Gamma_1''\). By Lemma 2.57, \(\Gamma_1'' \leq \Gamma_2''\). Since \(\Gamma_1' \sim \Gamma_1''\), by definition of \(\sim\), \(\Gamma_1' \leq \Gamma_2''\). We know that \(\leq\) is transitive, so \(\Gamma_1' \leq \Gamma_2''\).

Lemma 2.59. Suppose that \(T\) is a non-conflicting directional type of a program and

\[T' = T \cup \{ (p : \tau_1 \cap \tau_2 \rightarrow \sigma_1 \cap \sigma_2) \mid (p : \tau_1 \rightarrow \sigma_1) \in T, (p : \tau_2 \rightarrow \sigma_2) \in T \} . \]

Assume also that

\[
\begin{align*}
\Gamma &\leq \Gamma_1, \quad \Gamma \leq \Gamma_2, \\
\vdash \text{InferFromAtoms}(T, \Gamma_1, A, \Gamma_1''), \\
\vdash \text{InferFromAtoms}(T, \Gamma_2, A, \Gamma_2'').
\end{align*}
\]

Then there exists \(\Gamma''\) such that

\[
\vdash \text{InferFromAtoms}(T', \Gamma, A, \Gamma'''), \quad \Gamma''' \leq \Gamma_1'' \quad \text{and} \quad \Gamma''' \leq \Gamma_2'' .
\]

Proof. We shall construct \(\Gamma'''\) and the proof of \(\vdash \text{InferFromAtoms}(T', \Gamma, A, \Gamma''')\) by

induction on the length of \(A\).

If \(A = \langle \rangle\), we use \((P_1)\). We have:

\[
\begin{align*}
\vdash \text{InferFromAtoms}(T, \Gamma_1, \langle \rangle, \Gamma_1), \\
\vdash \text{InferFromAtoms}(T, \Gamma_2, \langle \rangle, \Gamma_2)
\end{align*}
\]

and we use the same rule obtaining

\[
\vdash \text{InferFromAtoms}(T', \Gamma, \langle \rangle, \Gamma) .
\]

By the assumptions, \(\Gamma \leq \Gamma_1\), and \(\Gamma \leq \Gamma_2\).
Now suppose that \(A \neq \langle \rangle\). Then \(\vdash \text{InferFromAtoms}(T, \Gamma_1, \langle a_1, \ldots, a_k \rangle, \Gamma_1'')\) was obtained using \((P_2)\) from

\[
\begin{align*}
\Gamma_1 &\vdash t_1 : \theta_1(\tau_1^i), \\
\Gamma_1, (t_1 : \theta_1(\sigma_1^i), \ldots, t_n : \theta_1(\sigma_n^i)) &\Rightarrow \Gamma_1', \\
\vdash \text{InferFromAtoms}(T, \Gamma_1', \langle a_2, \ldots, a_k \rangle, \Gamma_1''').
\end{align*}
\]

28
where \( a_1 = p(t_1, \ldots, t_n) \), and

\[(2.26) \quad (p : (\tau_1^1, \ldots, \tau_n^1) \to (\sigma_1^1, \ldots, \sigma_n^1)) \in T.\]

Similarly, \( \vdash \text{InferFromAtoms}(T, \Gamma_2, \langle a_1, \ldots, a_k \rangle, \Gamma_2') \) was obtained using \((P_2)\) from

\[(2.27) \quad \Gamma_2 \vdash t_i : \theta_2(\tau_2^i),\]
\[(2.28) \quad \Gamma_2, (t_1 : \theta_2(\sigma_2^1), \ldots, t_n : \theta_2(\sigma_n^1)) \Rightarrow \Gamma_2',\]
\[(2.29) \quad \vdash \text{InferFromAtoms}(T, \Gamma_2', \langle a_2, \ldots, a_k \rangle, \Gamma_2').\]

where

\[(2.30) \quad (p : (\tau_1^2, \ldots, \tau_n^2) \to (\sigma_1^2, \ldots, \sigma_n^2)) \in T.\]

By the definition of \( T' \), \((2.26)\) and \((2.30)\), we have

\[(2.31) \quad (p : (\tau_1^1 \cap \tau_1^2, \ldots, \tau_n^1 \cap \tau_n^2) \to (\sigma_1^1 \cap \sigma_1^2, \ldots, \sigma_n^1 \cap \sigma_n^2)) \in T'.\]

Since \( T \) is not conflicting,

\[\text{var}(\tau_1^1, \ldots, \tau_n^1, \sigma_1^1, \ldots, \sigma_n^1) \cap \text{var}(\tau_2^1, \ldots, \tau_2^2, \sigma_2^1, \ldots, \sigma_2^2) = \emptyset\]

Let

\[\theta'_1 = \theta_1|_{\text{var}(\tau_1^1, \ldots, \tau_1^2, \sigma_1^1, \ldots, \sigma_2^1)} \quad \text{and} \quad \theta'_2 = \theta_2|_{\text{var}(\tau_2^1, \ldots, \tau_2^2, \sigma_2^1, \ldots, \sigma_2^2)}.\]

One can see that \( \text{dom}(\theta'_1) \cap \text{dom}(\theta'_2) = \emptyset \). Let \( \theta = \theta'_1 \circ \theta'_2 \). Then for all \( i \in \{1, \ldots, n\} \), \( \theta_1(\tau_i^1) = \theta(\tau_i^1) \), \( \theta_1(\sigma_i^1) = \theta(\sigma_i^1) \) and \( \theta_2(\tau_i^2) = \theta(\tau_i^2) \), \( \theta_2(\sigma_i^2) = \theta(\sigma_i^2) \). We have assumed that \( \Gamma \leq \Gamma_1 \) and \( \Gamma \leq \Gamma_2 \), hence, by definition of the \( \leq \) relation on environments, \((2.23)\) and \((2.27)\) imply

\[\Gamma \vdash t_i : \theta_1(\tau_i^1) \quad \text{and} \quad \Gamma \vdash t_i : \theta_2(\tau_i^2)\]

which is equivalent to

\[\Gamma \vdash t_i : \theta(\tau_i^1) \quad \text{and} \quad \Gamma \vdash t_i : \theta(\tau_i^2)\]

Thus, by \((T_5)\), we have \( \Gamma \vdash t_i : \theta(\tau_i^1 \cap \tau_i^2) \) which is equivalent to

\[(2.32) \quad \Gamma \vdash t_i : \theta(\tau_i^1 \cap \tau_i^2).\]

Now, let

\[(2.33) \quad \Gamma, (t_1 : \theta(\sigma_1^1 \cap \sigma_2^1), \ldots, t_n : \theta(\sigma_n^1 \cap \sigma_n^2)) \Rightarrow \Gamma'.\]

Since \( \theta(\sigma_1^1 \cap \sigma_2^1) = \theta(\sigma_1^1 \cap \sigma_2^1) \) and since \( \Gamma \leq \Gamma_1 \) and \( \Gamma \leq \Gamma_2 \), by \((2.24)\), \((2.28)\), \((2.33)\) and Lemma 2.57, we have \( \Gamma' \leq \Gamma'_1 \) and \( \Gamma' \leq \Gamma'_2 \). Now, by the inductive hypothesis, there exists \( \Gamma'' \) such that \( \Gamma'' \leq \Gamma'_1 \) and \( \Gamma'' \leq \Gamma'_2 \) and

\[\vdash \text{InferFromAtoms}(T', \Gamma', \langle a_2, \ldots, a_k \rangle, \Gamma'').\]

which together with \((2.31)\), \((2.32)\) and \((2.33)\), by \((P_2)\), gives

\[\vdash \text{InferFromAtoms}(T', \Gamma, \langle a_1, \ldots, a_k \rangle, \Gamma'').\]

Since we have shown that \( \Gamma'' \leq \Gamma''_1 \) and \( \Gamma'' \leq \Gamma''_2 \), the proof is completed. \( \square \)
Chapter 2. The Type System

The Proof of Intersection Lemma. According to the definition of well-typedness (Definition 2.42), we should show that if $C$ is a clause of the program $P$ with the head predicate symbol $p$ and $(p : \tau \rightarrow \sigma) \in T'$ then

\begin{equation}
(2.34) \quad \vdash \text{ClauseHasType}(T', C, \tau \rightarrow \sigma).
\end{equation}

If $(p : \tau \rightarrow \sigma) \in T$ then (2.34) holds obviously by Lemma 2.56, since $P$ is well-typed with respect to $T$. So, assume that $(p : \tau \rightarrow \sigma) \notin T$. Then it has the form $(p : \tau_1 \cap \tau_2 \rightarrow \sigma_1 \cap \sigma_2)$ where $(p : \tau_1 \rightarrow \sigma_1) \in T$ and $(p : \tau_2 \rightarrow \sigma_2) \in T$. We should show that

\begin{equation}
(2.35) \quad \vdash \text{ClauseHasType}(T', C, (\tau_1 \cap \tau_2 \rightarrow \sigma_1 \cap \sigma_2)).
\end{equation}

We will construct the proof of this fact using proofs of

\begin{equation}
(2.36) \quad \vdash \text{ClauseHasType}(T, C, \tau_1 \rightarrow \sigma_1)
\end{equation}

and

\begin{equation}
(2.37) \quad \vdash \text{ClauseHasType}(T, C, \tau_2 \rightarrow \sigma_2)
\end{equation}

which exist since both $(p : \tau_1 \rightarrow \sigma_1)$ and $(p : \tau_2 \rightarrow \sigma_2)$ belong to $T$, and $P$ is well typed with respect to $T$. (2.36) was obtained using $(P_3)$ from

\begin{equation}
(2.38) \quad \varnothing, (t_1 : \tau_1^1, \ldots, t_n : \tau_1^n) \Rightarrow \Gamma_1,
\end{equation}

\begin{equation}
(2.39) \quad \vdash \text{InferFromAtoms}(T, \Gamma_1, A, \Gamma'_1),
\end{equation}

\begin{equation}
(2.40) \quad \Gamma'_1 \vdash t_1 : \sigma_1^1, \ldots, \Gamma'_1 \vdash t_n : \sigma_1^n.
\end{equation}

Similarly, (2.37) was obtained using $(P_3)$ from

\begin{equation}
(2.41) \quad \varnothing, (t_1 : \tau_2^1, \ldots, t_n : \tau_2^n) \Rightarrow \Gamma_2,
\end{equation}

\begin{equation}
(2.42) \quad \vdash \text{InferFromAtoms}(T, \Gamma_2, A, \Gamma'_2),
\end{equation}

\begin{equation}
(2.43) \quad \Gamma'_2 \vdash t_1 : \sigma_2^1, \ldots, \Gamma'_2 \vdash t_n : \sigma_2^n.
\end{equation}

Let

\begin{equation}
(2.44) \quad \varnothing, (t_1 : \tau_1^1 \cap \tau_2^1, \ldots, t_n : \tau_1^n \cap \tau_2^n) \Rightarrow \Gamma,
\end{equation}

By Lemma 2.57, (2.38), (2.41) and (2.44) we have $\Gamma \leq \Gamma_1$ and $\Gamma \leq \Gamma_2$. Now, by Lemma 2.59, (2.39) and (2.42), there exists $\Gamma'$ such that

\begin{equation}
(2.45) \quad \vdash \text{InferFromAtoms}(T', \Gamma, A, \Gamma')
\end{equation}

and $\Gamma' \leq \Gamma'_1$ and $\Gamma' \leq \Gamma'_2$. By Definition 2.30, (2.40), and (2.43), we have $\Gamma' \vdash t_i : \sigma_i^1$ and $\Gamma' \vdash t_i : \sigma_i^2$, for $i \in \{1, \ldots, n\}$. Thus, by $(T_3)$,

\begin{equation}
(2.46) \quad \Gamma' \vdash t_i : \sigma_i^1 \cap \sigma_i^2
\end{equation}

By $(P_3)$, (2.44), (2.45), and (2.46) completes the proof of (2.35).

\[\square\]

2.7 Derivations

Types in our system are polymorphic. It means that a directional type can be understood as a description of many ground directional types. One of the methods which, for a directional type of a predicate, gives another directional type of this predicate is substitution. The second method is subtyping. In this section we define the notion of derivation which subsumes both mentioned above methods. The main result presented here is Derivation Theorem.
2.7. Derivations

**Definition 2.60.** Let \( \tau_1 \rightarrow \sigma_1, \tau_2 \rightarrow \sigma_2 \) be directional types. \((\tau_1 \rightarrow \sigma_1) \rightsquigarrow (\tau_2 \rightarrow \sigma_2)\) iff there exists a substitution \( \theta \) such that \( \vdash \theta(\tau_1) \geq \tau_2 \) and \( \vdash \theta(\sigma_1) \leq \sigma_2 \).

If \((\tau_1 \rightarrow \sigma_1) \rightsquigarrow (\tau_2 \rightarrow \sigma_2)\) then we write that \((\tau_2 \rightarrow \sigma_2)\) is derived from \((\tau_1 \rightarrow \sigma_1)\). \(\square\)

It is easy to see that \((\tau_1 \rightarrow \sigma_1) \rightsquigarrow (\tau_2 \rightarrow \sigma_2)\) holds if \((\tau_2 \rightarrow \sigma_2)\) can be obtained from \((\tau_1 \rightarrow \sigma_1)\) by weakening and substitution.

**Lemma 2.61.** The relation \( \rightsquigarrow \) is transitive and reflexive.

**Proof.** Let \( \tau \rightarrow \sigma \) be a directional type, and let \( \theta \) be the empty substitution (the identity). Then \( \vdash \theta(\tau) = \tau \geq \tau \) and \( \vdash \theta(\sigma) = \sigma \leq \sigma \), and hence \((\tau \rightarrow \sigma) \rightsquigarrow (\tau \rightarrow \sigma)\). Thus \( \rightsquigarrow \) is reflexive.

Now, suppose that \((\tau_1 \rightarrow \sigma_1) \rightsquigarrow (\tau_2 \rightarrow \sigma_2)\) and \((\tau_2 \rightarrow \sigma_2) \rightsquigarrow (\tau_3 \rightarrow \sigma_3)\). From definition of the \( \rightsquigarrow \) relation there exist \( \theta_1 \) and \( \theta_2 \) such that

\[
\begin{align*}
(2.47) & \quad \vdash \theta_1(\tau_1) \geq \tau_2, \\
(2.48) & \quad \vdash \theta_1(\sigma_1) \leq \sigma_2, \\
(2.49) & \quad \vdash \theta_2(\tau_2) \geq \tau_3, \\
(2.50) & \quad \vdash \theta_2(\sigma_2) \leq \sigma_3.
\end{align*}
\]

Let \( \theta = \theta_2 \circ \theta_1 \).

\[
\vdash \theta(\tau_1) = \theta_2(\theta_1(\tau_1)) \\
\geq \theta_2(\tau_2) \quad \text{by (2.47) and corollary 2.24} \\
\geq \tau_3 \quad \text{by (2.49)}.
\]

Similarly

\[
\vdash \theta(\sigma_1) = \theta_2(\theta_1(\sigma_1)) \\
\leq \theta_2(\sigma_2) \quad \text{by (2.48) and corollary 2.24} \\
\leq \sigma_3 \quad \text{by (2.50)}.
\]

It gives \((\tau_1 \rightarrow \sigma_1) \rightsquigarrow (\tau_3 \rightarrow \sigma_3)\), hence the \( \rightsquigarrow \) relation is transitive. \(\square\)

**Theorem 2.1 (Derivation Theorem).** Assume that a program \( P \) is well typed with respect to \( T \) and

\[
T' = \{(p : \tau' \rightarrow \sigma') \mid (p : \tau \rightarrow \sigma) \in T, (\tau \rightarrow \sigma) \rightsquigarrow (\tau' \rightarrow \sigma')\}
\]

Then the program \( P \) is also well typed with respect to \( T' \).

Note that, since \( \rightsquigarrow \) is reflexive, we have \( T \subseteq T' \). Informally, Theorem 2.1 states that if a predicate \( p \) has a type \( (\tau \rightarrow \sigma) \) then \( p \) has also the type \( (\tau' \rightarrow \sigma') \), for each type \( (\tau' \rightarrow \sigma') \) such that \((\tau \rightarrow \sigma) \rightsquigarrow (\tau' \rightarrow \sigma')\). The rest of this section presents the proof of this theorem.

**The Proof of Derivation Theorem**

**Lemma 2.62.** Assume that a program \( P \) is well typed with respect to \( T \) and

\[
T' = \{(p : \tau' \rightarrow \sigma') \mid (p : \tau \rightarrow \sigma) \in T, \tau \geq \tau', \sigma \leq \sigma'\}
\]

Then \( P \) is also well typed with respect to \( T' \).
Chapter 2. The Type System

Proof. First, we will show that

\[(2.51) \text{ if } \Gamma_1 \geq \Gamma_2 \text{ and } \vdash \text{InferFromAtoms}(\mathcal{T}, \Gamma_1, \langle a_1, \ldots, a_k \rangle, \Gamma'_2) \]
\[
\text{then there exists } \Gamma'_2 \text{ such that }
\vdash \text{InferFromAtoms}(\mathcal{T}, \Gamma_2, \langle a_1, \ldots, a_k \rangle, \Gamma'_2) \text{ and } \Gamma'_1 \geq \Gamma'.'
\]

We shall proceed by induction on the length of the sequence \(\langle a_1, \ldots, a_k \rangle\).

- \(k = 0\).
  \[\vdash \text{InferFromAtoms}(\mathcal{T}, \Gamma_1, \langle \rangle, \Gamma'_1)\] can be proved only using rule \(P_1\). Hence \(\Gamma_1 = \Gamma'_1\). Let \(\Gamma'_2 = \Gamma_2\). We use the same rule to prove \(\vdash \text{InferFromAtoms}(\mathcal{T}, \Gamma_2, \langle \rangle, \Gamma'_2)\).

Obviously, \(\Gamma'_2 = \Gamma_2 \leq \Gamma_1 = \Gamma'_1\).

- \(k > 0\).
  \[\vdash \text{InferFromAtoms}(\mathcal{T}, \Gamma_1, \langle a_1, \ldots, a_k \rangle, \Gamma'_1)\] can be proved only using the \(P_2\) from

\[(2.52) \quad a_1 = p(t_1, \ldots, t_n),\]
\[(2.53) \quad (p : (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)) \in \mathcal{T},\]
\[(2.54) \quad \Gamma_1 \vdash t_i : \theta(\tau_i),\]
\[(2.55) \quad \Gamma_1, (t_1 : \theta(\sigma_1), \ldots, t_n : \theta(\sigma_n)) \Rightarrow \Gamma'_1\]
\[(2.56) \quad \vdash \text{InferFromAtoms}(\mathcal{T}, \Gamma'_1, \langle a_2, \ldots, a_k \rangle, \Gamma'_1)\).

Since \(\Gamma_2 \leq \Gamma_1\) and (2.54) holds, by Definition 2.30,

\[(2.57) \quad \Gamma_2 \vdash t_i : \theta(\tau_i).\]

By Lemma 2.48, there exists \(\Gamma''_2\) such that

\[(2.58) \quad \Gamma_2, (t_1 : \theta(\sigma_1), \ldots, t_n : \theta(\sigma_n)) \Rightarrow \Gamma''_2\]

By Lemma 2.51, \(\Gamma''_2 \leq \Gamma''_1\). Now, by the inductive hypothesis, there exists \(\Gamma'_2\) such that

\[\vdash \text{InferFromAtoms}(\mathcal{T}, \Gamma''_2, \langle a_2, \ldots, a_k \rangle, \Gamma'_2) \text{ and } \Gamma'_1 \geq \Gamma'_2.\]

This fact together with (2.52), (2.53), (2.57) and (2.58) allow to use rule \(P_2\) to obtain

\[\vdash \text{InferFromAtoms}(\mathcal{T}, \Gamma_2, \langle a_1, \ldots, a_k \rangle, \Gamma'_2)\]

which closes this part of the proof.

Now we shall prove the lemma. We want to show that \(P\) is well typed with respect to \(\mathcal{T}'\). This holds only if, for each clause \(C\) of the program \(P\) with the head predicate symbol \(p\), if \((p : (\tau'_1, \ldots, \tau'_n) \rightarrow (\sigma'_1, \ldots, \sigma'_n)) \in \mathcal{T}'\) then

\[\vdash \text{ClauseHasType}(\mathcal{T}', C, (\tau'_1, \ldots, \tau'_n) \rightarrow (\sigma'_1, \ldots, \sigma'_n)).\]

Since \(\mathcal{T} \subseteq \mathcal{T}'\), by Lemma 2.56, it suffices to show that

\[(2.59) \quad \vdash \text{ClauseHasType}(\mathcal{T}, C, (\tau'_1, \ldots, \tau'_n) \rightarrow (\sigma'_1, \ldots, \sigma'_n)).\]

Let \(C = (p(t_1, \ldots, t_n) \vdash B)\). By the definition of \(\mathcal{T}'\), there exist \(\tau_1, \ldots, \tau_n\) and \(\sigma_1, \ldots, \sigma_n\) such that \((p : (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)) \in \mathcal{T}\) and, for \(i \in \{1, \ldots, n\},\)

\[(2.60) \quad \tau_i \geq \tau'_i, \quad \text{and}\]
\[(2.61) \quad \sigma_i \leq \sigma'_i.\]

32
Since $P$ is well typed with respect to $T$ thus
\[ \vdash \text{ClauseHasType}(T, C, (\tau_1, \ldots, \tau_n) \to (\sigma_1, \ldots, \sigma_n)). \]

This can be obtained only using \((P_3)\) if
\[ \begin{align*}
& (2.62) \quad \emptyset, (t_1 : \tau_1, \ldots, t_n : \tau_n) \Rightarrow \Gamma_1, \\
& (2.63) \quad \vdash \text{InferFromAtoms}(T, \Gamma_1, B, \Gamma_2), \\
& (2.64) \quad \Gamma_2 \vdash t_1 : \sigma_1, \ldots, \Gamma_2 \vdash t_n : \sigma_n.
\end{align*} \]

Let
\[ (2.65) \quad \emptyset, (t_1 : \tau'_1, \ldots, t_n : \tau'_n) \Rightarrow \Gamma'_1. \]

From \((2.60), (2.62), \) and Lemma 2.58 it follows that $\Gamma'_1 \leq \Gamma_1$. Now, by \((2.51), \) there exists $\Gamma'_2$ such that
\[ (2.66) \quad \vdash \text{InferFromAtoms}(T, \Gamma'_1, B, \Gamma'_2) \]
and $\Gamma'_2 \leq \Gamma_2$. This and \((2.64)\) imply that $\Gamma'_2 \vdash t_1 : \sigma_1, \ldots, \Gamma'_2 \vdash t_n : \sigma_n$. Since $\sigma_i \leq \sigma'_i$, by \((T_3)\), we have
\[ (2.67) \quad \Gamma'_2 \vdash t_1 : \sigma'_1, \ldots, \Gamma'_2 \vdash t_n : \sigma'_n. \]

Now, \((2.65), (2.66)\) and \((2.67)\) allow to use rule \((P_3)\) to obtain the proof of \((2.59)\). \(\Box\)

**Definition 2.63.** Let $\theta$ be a type substitution, and $\Gamma = \{X_1 : \tau_1, \ldots, X_n : \tau_n\}$. Then
\[ \theta(\Gamma) = \{X_1 : \theta(\tau_1), \ldots, X_n : \theta(\tau_n)\}. \]

**Lemma 2.64.** If $\Gamma \vdash t : \tau$ then, for any type substitution $\theta$,
\[ \theta(\Gamma) \vdash t : \theta(\tau). \]

The proof proceeds by easy induction of the structure of the proof of $\Gamma \vdash t : \tau$.

We move it to Appendix A.1.

**Lemma 2.65.** Assume that $\Gamma, (t : \tau) \Rightarrow \Gamma'$ and $\theta(\Gamma), (t : \theta(\tau)) \Rightarrow \Gamma''$. Then $\Gamma'' \leq \theta(\Gamma')$.

**Proof.** The proof proceeds by induction on the structure of the proof of $\Gamma, (t : \tau) \Rightarrow \Gamma'$.

Proofs for $(K_1)$, $(K_2)$, $(K_3)$ are similar, so we present only one of them — the proof for $(K_2)$. We also omit the obvious proof for $(K_3)$.

- Assume that $f : \tau_1 \ast \cdots \ast \tau_n \rightarrow \tau$. Assume also that
\[ \Gamma, (f(t_1, \ldots, t_n) : \nu(\mathcal{N}(\tau))) \Rightarrow \Gamma_1 \cap \cdots \cap \Gamma_n \]
was obtained from $\Gamma, (t_i : \nu(\tau_i)) \Rightarrow \Gamma_i$, for $i \in \{1, \ldots, n\}$, using $(K_2)$. By the inductive hypothesis, there exist environments $\Gamma'_1, \ldots, \Gamma'_n$ such that
\[ \theta(\Gamma_i), (t_i : \theta(\nu(\tau_i))) \Rightarrow \Gamma''_i \quad \text{and} \quad \vdash \Gamma''_i \leq \theta(\Gamma_i). \]

The rule $(K_2)$ proves that
\[ \theta(\Gamma), (f(t_1, \ldots, t_n) : \theta(\nu(\mathcal{N}(\tau)))) \Rightarrow \Gamma''_1 \cap \cdots \cap \Gamma''_n. \]

Furthermore, since $\vdash \Gamma''_i \leq \theta(\Gamma_i)$, by Corollary 2.47, we have
\[ \vdash \Gamma''_1 \cap \cdots \cap \Gamma''_n \leq \theta(\Gamma_1) \cap \cdots \cap \theta(\Gamma_n) = \theta(\Gamma_1 \cap \cdots \cap \Gamma_n) \]

33
• Assume that $\Gamma, (f(t_1, \ldots, t_n) : \tau) \Rightarrow \{ \text{false} \}$ was obtained using $(K_3)$, since there is no signature assigned to $f$ with the same head as $\tau$. Since $\text{head of } \theta(\tau)$ is equal to head of $\tau$, there is no signature assigned to $f$ with the same head as $\theta(\tau)$. Therefore we can use the same rule to prove

$$\theta(\Gamma), (f(t_1, \ldots, t_n) : \theta(\tau)) \Rightarrow \{ \text{false} \}$$

what closes the proof, since $\{ \text{false} \} \leq \{ \text{false} \}$.

• Assume that $\Gamma, (t : \bot) \Rightarrow \{ \text{false} \}$ was obtained using $(K_4)$. Since $\theta(\bot) = \bot$, we can use the same rule to prove $\theta(\Gamma), (t : \theta(\bot)) \Rightarrow \{ \text{false} \}$.

• Assume that $\Gamma, (f(t_1, \ldots, t_n) : \alpha) \Rightarrow \Gamma$ was obtained using $(K_6)$. Let

$$\theta(\Gamma), (f(t_1, \ldots, t_n) : \theta(\alpha)) \Rightarrow \Gamma''.$$  

By Lemma 2.49, we have $\Gamma'' \leq \theta(\Gamma)$.

• Assume that $\Gamma, (t : \tau_1 \cup \ldots \cup \tau_n) \Rightarrow \Gamma_1 \cup \ldots \cup \Gamma_n$ was obtained from $\Gamma, (t : \tau_i) \Rightarrow \Gamma_i$, for each $i \in \{1, \ldots, n\}$, using $(K_8)$. By the inductive hypothesis, there exist environments $\Gamma''_1, \ldots, \Gamma''_n$ such that

$$\theta(\Gamma), (t : \theta(\tau_i)) \Rightarrow \Gamma''_i$$  

and $\vdash \Gamma''_i \leq \theta(\Gamma_i)$.

The rule $(K_8)$ proves that

$$\theta(\Gamma), (t : \theta(\tau_1) \cup \ldots \cup \theta(\tau_n)) \Rightarrow \Gamma''_1 \cup \ldots \cup \Gamma''_n$$

which is equivalent to

$$\theta(\Gamma), (t : \theta(\tau_1 \cup \ldots \cup \tau_n)) \Rightarrow \Gamma''_1 \cup \ldots \cup \Gamma''_n.$$  

Since $\vdash \Gamma''_i \leq \theta(\Gamma_i)$, by corollary 2.47, we have

$$\Gamma''_1 \cup \ldots \cup \Gamma''_n \leq \theta(\Gamma_1) \cup \ldots \cup \theta(\Gamma_n) = \theta(\Gamma_1 \cup \ldots \cup \Gamma_n).$$

• Assume that $\Gamma, (X : \tau) \Rightarrow \Gamma \cap \{ X : \tau \}$ was obtained using $(K_9)$. We use the same rule to obtain

$$\theta(\Gamma), (X : \theta(\tau)) \Rightarrow \theta(\Gamma) \cap \{ X : \theta(\tau) \}.$$  

To complete the proof we want to show that

$$L = \theta(\Gamma) \cap \{ X : \theta(\tau) \} \leq \theta(\Gamma \cap \{ X : \tau \}) = R.$$  

This holds since, from Definition 2.39 it follows that $L = R$.

\[\square\]

**Corollary 2.66.** Assume that

(a) $\Gamma_2 \leq \theta(\Gamma_1)$,
(b) $\Gamma_1, (t : \tau) \Rightarrow \Gamma'_1$,
(c) $\Gamma_2, (t : \theta(\tau)) \Rightarrow \Gamma'_2$.

Then $\Gamma'_2 \leq \theta(\Gamma'_1)$.

**Proof.** Let $\theta(\Gamma_1), (t : \theta(\tau)) \Rightarrow \Gamma''$. By Lemma 2.65, $\Gamma'' \leq \theta(\Gamma'_1)$. By Lemma 2.51, $\Gamma'_2 \leq \Gamma''$. So, since the $\leq$ relation is transitive, $\Gamma'_2 \leq \theta(\Gamma'_1)$.

\[\square\]
Lemma 2.67. Assume that, for environments \( \Gamma_1, \Gamma'_1 \) and \( \Gamma_2 \) and for a sequence \( A \) of atoms, we have

\[ \vdash \text{InferFromAtoms}(T, \Gamma_1, A, \Gamma'_1) \quad \text{and} \quad \Gamma_2 \leq \theta(\Gamma_1). \]

Then there exists \( \Gamma'_2 \) such that

\[ \vdash \text{InferFromAtoms}(T, \Gamma_2, A, \Gamma'_2) \quad \text{and} \quad \Gamma'_2 \leq \theta(\Gamma'_1). \]

Proof. We proceed by induction on the length of \( A \).

- \( A = \langle \rangle \). \( \vdash \text{InferFromAtoms}(T, \Gamma_1, A, \Gamma'_1) \) can be obtained only using rule \( (P_1) \), thus \( \Gamma'_1 = \Gamma_1 \). Let \( \Gamma'_2 = \Gamma_2 \). We use the same rule to prove \( \vdash \text{InferFromAtoms}(T, \Gamma_2, \langle \rangle, \Gamma'_2) \).

  Since we have assumed that \( \Gamma_2 \leq \theta(\Gamma_1) \), the proof is completed.

- \( A = \langle a_1, \ldots, a_k \rangle \), \( k \geq 1 \).

  \( \vdash \text{InferFromAtoms}(T, \Gamma_1, A, \Gamma'_1) \) can be obtained only using rule \( (P_2) \) if

  \begin{align*}
  & (2.68) \quad a_1 = p(t_1, \ldots, t_n) \\
  & (2.69) \quad (p : (\tau_1, \ldots, \tau_n) \Rightarrow (\sigma_1, \ldots, \sigma_n)) \in T, \\
  & \Gamma_1 \vdash t_i : \nu(\tau_i) \\
  & \Gamma_1, (t_1 : \nu(\sigma_1), \ldots, t_n : \nu(\sigma_n)) \Rightarrow \Gamma'_1 \\
  & \vdash \text{InferFromAtoms}(T, \Gamma'_1, \langle a_2, \ldots, a_k \rangle, \Gamma'_1). 
  \end{align*}

By Lemma 2.64, \( \theta(\Gamma_1) \vdash t_i : \theta(\nu(\tau_i)) \). Since \( \Gamma_2 \leq \theta(\Gamma_1) \), we have

\[ (2.70) \quad \Gamma_2 \vdash t_i : \theta(\nu(\tau_i)). \]

Now, let

\[ (2.71) \quad \Gamma_2, (t_1 : \theta(\nu(\sigma_1)), \ldots, t_n : \theta(\nu(\sigma_n))) \Rightarrow \Gamma'_2. \]

We know from corollary 2.66 that \( \Gamma''_2 = \theta(\Gamma''_1) \). Moreover, by the inductive hypothesis there exists \( \Gamma'_2 \) such that

\[ (2.72) \quad \vdash \text{InferFromAtoms}(T, \Gamma'_2, \langle a_2, \ldots, a_k \rangle, \Gamma'_2) \quad \text{and} \quad \Gamma'_2 \leq \theta(\Gamma'_1). \]

Since, for \( i \in \{1, \ldots, n\}, \theta(\nu(\sigma_n)) = \theta(\sigma_n), \) we can use (2.68), (2.69), (2.70) (2.71), (2.72), and rule \((P_2)\) to obtain

\[ \vdash \text{InferFromAtoms}(T, \Gamma_2, A, \Gamma'_2) \]

Since we have showed that \( \Gamma'_2 \leq \theta(\Gamma'_1) \), this closes the proof.

\[ \square \]

Lemma 2.68. Assume that

\( \vdash \text{ClauseHasType}(T, C, (\tau_1, \ldots, \tau_n) \Rightarrow (\sigma_1, \ldots, \sigma_n)). \)

Then, for any type substitution \( \theta \),

\( \vdash \text{ClauseHasType}(T, C, (\theta(\tau_1), \ldots, \theta(\tau_n)) \Rightarrow (\theta(\sigma_1), \ldots, \theta(\sigma_n))). \)

Proof. \( \vdash \text{ClauseHasType}(T, C, (\tau_1, \ldots, \tau_n) \Rightarrow (\sigma_1, \ldots, \sigma_n)) \) can be obtained only using rule \( (P_3) \) if

\[ \emptyset, (t_1 : \tau_1, \ldots, t_n : \tau_n) \Rightarrow \Gamma_1, \]

\( \vdash \text{InferFromAtoms}(T, \Gamma_1, B, \Gamma_2) \)

\( \Gamma_2 \vdash t_1 : \sigma_1, \ldots, \Gamma_2 \vdash t_n : \sigma_n \)
where $C = (p(t_1, \ldots, t_n) \rightarrow B)$. Let
\[(2.73) \quad \emptyset, (t_1 : \theta(\tau_1), \ldots, t_n : \theta(\tau_n)) \Rightarrow \Gamma'_1.
\]
By Lemma 2.65, $\Gamma'_1 \leq \theta(\Gamma_1)$. By Lemma 2.67, there exists $\Gamma'_2$ such that
\[(2.74) \quad \vdash \text{InferFromAtoms}(\mathcal{T}, \Gamma'_1, B, \Gamma'_2) \quad \text{and} \quad \Gamma'_2 \leq \theta(\Gamma_2).
\]
Since $\Gamma_2 \vdash t_i : \sigma_i$, by Lemma 2.64, $\theta(\Gamma_2) \vdash t_i : \theta(\sigma_i)$. Hence, since we have showed that $\Gamma'_2 \leq \theta(\Gamma_2)$,
\[(2.75) \quad \Gamma'_2 \vdash t_i : \theta(\sigma_i), \quad \text{for } i \in \{1, \ldots, n\}.
\]
(2.73), (2.74) and (2.75) allow us to use the rule $(P_3)$ to prove
\[\vdash \text{ClauseHasType}(\mathcal{T}, C, (\theta(\tau_1), \ldots, \theta(\tau_n)) \rightarrow (\theta(\sigma_1), \ldots, \theta(\sigma_n))).\]

\[\square\]

**Lemma 2.69.** Assume that a program $P$ is well typed with respect to $\mathcal{T}$ and
\[\mathcal{T}' = \{(p : \theta(\tau) \rightarrow \theta(\sigma)) \mid (p : \tau \rightarrow \sigma) \in \mathcal{T}, \theta \text{ is a type substitution}\}.
\]
Then $P$ is also well typed with respect to $\mathcal{T}'$.

**Proof.** According to the definition of well-typedness, we should show that if $C$ is a clause of the program $P$ with the head predicate symbol $p$ and $(p : \tau' \rightarrow \sigma') \in \mathcal{T}'$ then
\[(2.76) \quad \vdash \text{ClauseHasType}(\mathcal{T}', C, \tau' \rightarrow \sigma').
\]
So, suppose that $(p : \tau' \rightarrow \sigma') \in \mathcal{T}'$. Thus we have $\tau' = \theta(\tau)$ and $\sigma' = \theta(\sigma)$, for some type substitution $\theta$, and for some $\tau, \sigma$ such that $(p : \tau \rightarrow \sigma) \in \mathcal{T}$.

Note that $\mathcal{T} \subseteq \mathcal{T}'$. That is because, in definition of $\mathcal{T}'$ we use, among other substitutions, the empty substitution (identity). Hence, by Lemma 2.56, it suffices to show that
\[(2.77) \quad \vdash \text{ClauseHasType}(\mathcal{T}, C, (\theta(\tau) \rightarrow \theta(\sigma))).
\]
We have assumed that $P$ is well typed with respect to $\mathcal{T}$. Thus
\[\vdash \text{ClauseHasType}(\mathcal{T}, C, \tau \rightarrow \sigma).
\]
Hence, by Lemma 2.68, (2.77) holds, which closes the proof. \[\square\]

**Proof of the derivation lemma.** Assume that $P$ is well typed with respect to $\mathcal{T}$. Let
\[\mathcal{T}'' = \{(p : \theta(\tau) \rightarrow \theta(\sigma)) \mid (p : \tau \rightarrow \sigma) \in \mathcal{T}\}.
\]
By Lemma 2.69, $P$ is well typed with respect to $\mathcal{T}''$. Moreover
\[\mathcal{T}' = \{(p : \tau' \rightarrow \sigma') \mid (p : \tau \rightarrow \sigma) \in \mathcal{T}, (\tau \rightarrow \sigma) \rightarrow (\tau' \rightarrow \sigma')\}
\[= \{(p : \tau' \rightarrow \sigma') \mid (p : \tau \rightarrow \sigma) \in \mathcal{T}, \theta(\tau) \geq \tau', \theta(\sigma) \leq \sigma', \text{for some } \theta\}
\[= \{(p : \tau' \rightarrow \sigma') \mid (p : \tau \rightarrow \sigma) \in \mathcal{T}'' \tau \geq \tau', \sigma \leq \sigma'\}.
\]
By Lemma 2.62, the program $P$ is well typed with respect to $\mathcal{T}'$. \[\square\]
Chapter 3

Semantics

In this chapter we define an interpretation of types and formulas of the type language. These formulas will be interpreted using some family of models based on the Herbrand universe. We shall show that the proof system defined by axioms and rules of our system is sound. In the last section of this chapter we prove Soundness Theorems which show that directional types provable in our system are correct and describe declarative and procedural properties of logic programs.

3.1 Semantics of Types

Intuitions behind our notion of a type is set based. A type is a description of a subset of the Herbrand universe. We shall describe the meaning of types by the semantic function $[[\ ]]$ from the set of ground types into the powerset of the Herbrand Universe $H$.

Let $\Delta$ be the set of type interpretations, i.e. the set of functions from types into the powerset of $H$ ($\Delta = \{ \delta \mid \delta : T \to 2^H \}$). We define an order on $\Delta$. Let $\delta, \delta' \in \Delta$.

$$\delta \leq \delta' \iff \forall \tau \in T \quad \delta(\tau) \subseteq \delta'(\tau).$$

**Definition 3.1.** Let $[[\ ]] \in \Delta$ be the least function such that

\begin{align}
(3.1) \quad &[[\bot]] = \emptyset, \quad [[\top]] = H \\
(3.2) \quad &[[F(\tau_1, \ldots, \tau_n)]] = \\
&\{ f(t_1, \ldots, t_k) \in H \mid f : \sigma_1 \cdots \sigma_k \to F(y_1, \ldots, y_n), \text{there exists } \theta \text{ such that, for each } i \in \{1, \ldots, n\}, \text{either } y_i = \bot \text{ or } \theta(y_i) = \tau_i, \text{ and } t_i \in \theta(\sigma_i) \} \\
(3.3) \quad &[[\tau \cup \sigma]] = [[\tau]] \cup [[\sigma]], \quad [[\tau \cap \sigma]] = [[\tau]] \cap [[\sigma]]
\end{align}

This definition is correct, i.e. there exists a function which satisfies conditions (3.1)–(3.3)

37
Chapter 3. Semantics

Proof. First, let us show that \( \langle \Delta, \subseteq \rangle \) is a complete lattice. For any \( X \subseteq \Delta \), let

\[
\delta_X(\tau) = \bigcup_{\epsilon \in X} \epsilon(\tau).
\]

It is easy to see that \( \delta_X \) is the supremum of \( X \).

Now, let \( R \) be the function which for a type interpretation \( \delta \) gives a type interpretation \( R_\delta \) defined as follows

(3.4) \[ R_\delta(\bot) = \emptyset, \]
(3.5) \[ R_\delta(\top) = H \]

(3.6) \[ R_\delta(F(\tau_1, \ldots, \tau_n)) = \{ f(t_1, \ldots, t_k) \in H \mid f : \sigma_1 \ast \cdots \ast \sigma_k \rightarrow F(y_1, \ldots, y_n), \]

there exists \( \theta \) such that, for each \( i \in \{1, \ldots, n\} \),

either \( y_i = \bot \) or \( \theta(y_i) = \tau_i \), and \( t_i \in \delta(\sigma_i) \} \]

(3.7) \[ R_\delta(\tau \cup \sigma) = R_\delta(\tau) \cup R_\delta(\sigma) \]
(3.8) \[ R_\delta(\tau \cap \sigma) = R_\delta(\tau) \cap R_\delta(\sigma). \]

It is easy to see that \([\cdot]\), if it exists, is the least fixpoint of \( R \). By the Fixpoint Theorem (Knaster and Tarski [23]), \( R \) has the least fixpoint provided it is monotonic. Thus, to complete the proof of correctness of definition 3.1 it suffices to show that \( R \) is monotonic. This can be easily checked by induction on the structure of types. \( \square \)

We have introduced symbol \( \leq \) as a shorthand and we intend to use \( \leq \) to express set inclusion. So, since \( \tau \leq \sigma \) denotes \( \tau \cap \sigma = \tau \), we expect that

\[
[\tau \cap \sigma] = [\tau] \iff [\tau] \subseteq [\sigma]
\]

Indeed, by (3.3), \([\tau \cap \sigma] = [\tau] \cap [\sigma]\), so

\[
[\tau] \cap [\sigma] = [\tau] \iff [\tau] \subseteq [\sigma]
\]

is obviously true.

3.2 Soundness of Equality Axioms

In this section we give a lemma which states that the proof system given by equality axioms is sound.

Lemma 3.2. Assume that \( \tau, \sigma \) are ground types. If \( \vdash \tau = \sigma \) then \([\tau] = [\sigma] \).

The proof is moved to Appendix A.2.

Corollary 3.3. Assume that \( \tau, \sigma \) are types. If \( \vdash \tau = \sigma \) then, for each ground type substitution \( \theta \), we have \([\theta(\tau)] = [\theta(\sigma)]\).

This corollary is a simple consequence of the previous lemma and Lemma 2.23.
3.3 Soundness of Term Typing Rules

So far, we have defined an interpretation of ground types. This can serve as a base on which models of our system can be built.

**Definition 3.4.** A valuation is a function $v : V \rightarrow H$, where $V$ is the set of program variables, and $H$ is the Herbrand Universe, i.e. the set of ground terms. In the natural way we extend $v$ to a function $v : T \rightarrow H$, where $T$ denotes the set of all (not only ground) terms.

**Definition 3.5.** For a variable valuation $v$, $M_v$ is a structure with universe $H$, with interpretation of term variables as ground terms fixed by $v$, and $\models$ denotes second order satisfaction relation over $H$ defined as follows.

(i) $M_v \models (\tau = \sigma)$ iff $[\tau] = [\sigma]$.

(ii) $M_v \models (t : \tau)$ iff $v(t) \in [\tau]$.

(iii) $M_v \models \text{false}$ does not holds.

In the remaining part of this chapter we consider only such structures. All of them share interpretation of types, and differ only in variable mapping. We will write $\models \varphi$ if $M_v \models \varphi$ does not depend on variable valuation (it is the case of type formulas of the form $\tau = \sigma$). Moreover $\Gamma \models \varphi$ means that, for any model $M_v$ which satisfies $\Gamma$ (i.e., satisfies each sentence in $\Gamma$), $M_v \models \varphi$.

Now, we present the main result of this section.

**Lemma 3.6.** Let $\Gamma$ be a ground environment, $\tau$ a ground type, and $t$ a term. If $\Gamma \vdash t : \tau$ then $\Gamma \models t : \tau$.

The proof is moved to Appendix A.2. We can easily generalize this lemma to work with all (not only ground) environments and types.

**Corollary 3.7.** Let $\Gamma$ be an environment, $\tau$ be a type, and $t$ be a term. If $\Gamma \vdash t : \tau$ then, for any ground type substitution $\theta$, we have $\theta(\Gamma) \models t : \theta(\tau)$.

This corollary is a simple consequence of Lemma 3.6 and Lemma 2.64.

3.4 Soundness of Consequence Operator Rules

We define the meaning of the consequence operator $\Rightarrow$ as follows. $M_v \models (\Gamma, \varphi \Rightarrow \Gamma')$ iff

$$\text{if } M_v \models \Gamma \text{ and } M_v \models \varphi \text{ then } M_v \models \Gamma'.$$

**Lemma 3.8.** Let $\Gamma, \Gamma'$ be a ground environments, $\varphi$ a ground formula. If $(\Gamma, \varphi \Rightarrow \Gamma')$ is provable in our system then we have $\models (\Gamma, \varphi \Rightarrow \Gamma')$.

The proof of this lemma is given in Appendix A.2.

3.5 Soundness Theorems

In this section we show that in our system only correct types of programs can be proved. We formulate this fact in two different ways. Theorem 3.1 gives a declarative meaning, and Theorem 3.2 gives a procedural meaning to this fact. We need some classical definitions and well-known facts related to semantics of logic programs. Recall that $H$ is the set of ground terms.
Chapter 3. Semantics

Definition 3.9. (see [2]) Herbrand base is the set of all elements \( a \) of the form \( p(t_1, \ldots, t_n) \), where \( p \) is a predicate symbol, and \( t_1, \ldots, t_n \) are elements of \( H \).

A Herbrand interpretation is a subset of Herbrand base.

A Herbrand model for a program \( P \) is a Herbrand interpretation which is a model of \( P \). We denote by \( M_P \) the least (according to subset relation) Herbrand model of \( P \).

Now, we introduce the immediate consequence operator \( T_P \) mapping Herbrand interpretations to Herbrand interpretations.

Definition 3.10. For a program \( P \) and a Herbrand interpretation \( I \), we put

\[
a \in T_P(I) \quad \text{iff} \quad \text{for some ground substitution } v \text{ and a clause } a_0 \vdash a_1, \ldots, a_n \text{ of } P \text{ we have } a = v(a_0) \text{ and } v(a_1), \ldots, v(a_n) \in I.
\]

We assume that \( P \) is a fixed program. We will write \( T \) instead of \( T_P \). Now we define iterations of operator \( T \).

Definition 3.11.

\[
T^0(I) = I, \quad T^{n+1}(I) = T(T^n I), \quad T^\omega(I) = \bigcup_{n<\omega} T^n I.
\]

We write \( T^n \) instead of \( T^n(\emptyset) \), and \( T^\omega \) instead of \( T^\omega(\emptyset) \).

It is known (see [11]) that \( T^\omega \) is the least Herbrand model of \( P \).

By the success set of a program \( P \) we denote the set of all ground atoms \( a \) such that \( P \cup \{ \vdash a \} \) has an SLD-refutation. The following lemma is given in [2]:

Lemma 3.12. Consider a program \( P \) and a ground atom \( a \). Then the following are equivalent:

(a) \( a \) is in the success set of \( P \).
(b) \( a \) is in the \( M_P \).
(c) \( a \in T^\omega \).
(d) \( a \) is a semantic consequence of \( P \), i.e., each model\(^1\) satisfying \( P \), satisfies also \( a \).

Now, we are ready to give the first, declarative, version of the Soundness Theorem.

Theorem 3.1 (Soundness Theorem 1). Assume that a program \( P \) is well-typed with respect to \( T \). Then for any predicate \( p \) of the program \( P \), for any ground terms \( t_1, \ldots, t_n \), and any ground type substitution \( \theta \),

\[
\text{if } (p : (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)) \in T, \quad t_i \in [\theta(\tau_i)], \text{ for each } i \in \{1, \ldots, n\}, \text{ and } p(t_1, \ldots, t_n) \in M_P
\]

then \( t_i \in [\theta(\sigma_i)] \), for \( i \in \{1, \ldots, n\} \).

Proof. By Lemma 3.12, \( M_P \) is equal to \( T^\omega \). Therefore, by the definition of \( T^\omega \),

\[
\text{if } (p : (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)) \in T, \quad t_i \in [\theta(\tau_i)], \text{ for each } i \in \{1, \ldots, n\}, \text{ and } p(t_1, \ldots, t_n) \in T^m
\]

then \( t_i \in [\theta(\sigma_i)] \), for \( i \in \{1, \ldots, n\} \).

We will show (3.10) by induction on \( m \).

\(^1\)We consider here all models, not only models from the family introduced in section 3.3.
3.5. Soundness Theorems

Suppose that \( m = 0 \). Then \( T \upharpoonright m = \emptyset \), so \( p(t) \in T \upharpoonright m \) does not hold. Since the left-hand side of the implication (3.10) is false, the whole implication is true.

Suppose that \( m > 0 \). By the inductive hypothesis, for any predicate \( p \) of the program \( P \), for any ground terms \( t_1, \ldots, t_n \), and any ground type substitution \( \theta \), we have

\[
\begin{align*}
\text{if } & (p : (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)) \in \mathcal{T}, \ t_i \in [\theta(\tau_i)], \text{ for each } i \in \{1, \ldots, n\}, \\
\text{then } & t_i \in [\theta(\sigma_i)], \text{ for } i \in \{1, \ldots, n\}.
\end{align*}
\]

(3.11)

Assume that

\[
\begin{align*}
(p : (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)) \in \mathcal{T} \\
i \in \{1, \ldots, n\} \\
p(t_1, \ldots, t_n) \in \mathcal{T} \upharpoonright (m - 1)
\end{align*}
\]

(3.12)

Then

\[
t_i \in [\theta(\tau_i)] \\
p(t_1, \ldots, t_n) = v(a_0) \quad \text{and} \quad v(a_1), \ldots, v(a_n) \in \mathcal{T} \upharpoonright (m - 1).
\]

(3.13)

(3.14)

We want to show that

\[
t_i \in [\theta(\sigma_i)], \quad \text{for } i \in \{1, \ldots, n\}.
\]

(3.15)

By (3.14), we have \( p(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{T} \upharpoonright (m - 1)) \), so, by the definition of \( \mathcal{T} \), for some ground substitution \( v \), for some clause \( a_0 \leftarrow a_1, \ldots, a_n \) of \( P \), we have

\[
\begin{align*}
p(t_1, \ldots, t_n) = v(a_0) & \quad \text{and} \quad v(a_1), \ldots, v(a_n) \in \mathcal{T} \upharpoonright (m - 1).
\end{align*}
\]

(3.16)

This implies that \( a_0 = p(s_1, \ldots, s_n) \), for some terms \( s_i \) such that, for \( i \in \{1, \ldots, n\} 

v(s_i) = t_i.

Since \( (p : (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)) \in \mathcal{T} \), by the definition of a well typed program (Definition (2.42)), we have

\[
\vdash \text{ClauseHasType}(\mathcal{T}, (a_0 \leftarrow a_1, \ldots, a_n), (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)).
\]

By Lemma 2.68

\[
\vdash \text{ClauseHasType}(\mathcal{T}, (a_0 \leftarrow a_1, \ldots, a_n), \\
(\theta(\tau_1), \ldots, \theta(\tau_n)) \rightarrow (\theta(\sigma_1), \ldots, \theta(\sigma_n)),
\]

which can be obtain only be \( (P_3) \) if there exists \( \Gamma_1, \Gamma_2 \) such that

\[
\begin{align*}
\emptyset, (s_1 : \theta(\tau_1), \ldots, s_n : \theta(\tau_n)) & \Rightarrow \Gamma_1, \\
\vdash \text{InferFromAtoms}(\mathcal{T}, \Gamma_1, \langle a_1, \ldots, a_n \rangle, \Gamma_2), \\
\Gamma_2 \vdash s_1 : \theta(\sigma_1), \ldots, \Gamma_2 \vdash s_n : \theta(\sigma_n)
\end{align*}
\]

(3.17)

(3.18)

(3.19)

Since \( t_i = v(s_i) \), (3.13) gives \( v(s_i) \in [\theta(\tau_i)] \) which means that \( M_v \models s_i : \theta(\tau_i) \). Of course \( M_v \models \emptyset \), so, (3.17), and Lemma 3.8 imply that \( M_v \models \Gamma_1 \).

We can claim that

\[
M_v \models \Gamma \land \vdash \text{InferFromAtoms}(\mathcal{T}, \Gamma, \langle a_1, \ldots, a_n \rangle, \Gamma''') \quad \Rightarrow \quad M_v \models \Gamma''
\]

(3.20)

which implies \( M_v \models \Gamma_2 \), and by (3.19). Hence, by Lemma 3.6, \( M_v \models s_i : \theta(\sigma_i) \), for \( i \in \{1, \ldots, n\} \). Thus \( v(s_i) \in [\theta(\sigma_i)] \). Since \( v(s_i) = t_i \), we have \( t_i \in [\theta(\sigma_i)] \). Hence (3.15) holds which finishes the proof. Thus it remains to prove (3.20).
Chapter 3. Semantics

We proceed by induction on the length of the sequence $B = \langle a_1, \ldots, a_k \rangle$. For the empty sequence, $(P_1)$ is the only rule we can use. So, $\Gamma'' = \Gamma$, and (3.20) is obviously true.

Now suppose that $B = \langle a_1, \ldots, a_k \rangle$ and $k > 0$. Assume that

$$M_v \models \Gamma \land \vdash \text{InferFromAtoms}(T, \Gamma, B, \Gamma'')$$

We are going to show that $M_v \models \Gamma'' \vdash \text{InferFromAtoms}(T, \Gamma, B, \Gamma')$ is obtained using $(P_2)$ from the following facts:

(3.21) $\Gamma \vdash t_i : \theta(\tau_i)$,
(3.22) $\Gamma, (t_1 : \theta(\sigma_1), \ldots, t_n : \theta(\sigma_n)) \Rightarrow \Gamma'$,
(3.23) $\vdash \text{InferFromAtoms}(T, \Gamma', \langle a_2, \ldots, a_k \rangle, \Gamma'').$

where $a_1 = p(t_1, \ldots, t_n)$, and

(3.24) $(p : (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)) \in T,$

Since we have assumed that $M_v \models \Gamma$, (3.21) implies that $M_v \models t_i : \theta(\tau_i)$ which means that $v(t_i) \in [\theta(\tau_i)]$. By (3.16)

$$p(v(t_1), \ldots, v(t_n)) = v(a_1) \in T \uparrow (m - 1).$$

Since (3.24) holds, we can use (3.11) to obtain $[v(t_i) : [\theta(\sigma_i)]]$ which is equivalent to

$$M_v \models t_i : \theta(\sigma_i) \quad \text{for each } i \in \{1, \ldots, n\}.$$

Since, $M_v \models \Gamma$ and (3.22), by Lemma 3.8, we have $M_v \models \Gamma'$. Now, by the inductive hypothesis, $M_v \models \Gamma''$, which completes the proof. $\square$

Consider a program $P$ with a goal $G$. We say that $G$ evaluates successfully with the computed answer substitution $\eta$ if there exists an SLD-refutation of $P \cup \{G\}$ with the computed answer substitution $\eta$.

Recall that, if $\phi$ is a formula of our type language, then $\models \phi$ means that $M_v \models \phi$ for any ground substitution $v$. Thus $\models t : \tau$ iff, for any ground substitution $v$, $v(t) \in [\tau]$.

**Theorem 3.2 (Soundness Theorem 2).** Assume that a program $P$ is well-typed with respect to $T$, $p$ is one of its predicates, $(p : \tau \rightarrow \sigma) \in T$, and $\theta$ is any ground type substitution. If $\models t : \theta(\tau)$, and $p(t)$ evaluates successfully with computed answer substitution $\eta$, then $\models \eta(t) : \theta(\sigma)$.

To prove this theorem we shall use the classical lemma below (see [2]).

**Lemma 3.13.** Let $P$ be a program, and let $N = (\vdash a_1, \ldots, a_n)$ be a goal. Suppose that there exists an SLD-refutation of $P \cup \{N\}$ with the computed answer substitution $\eta$. Then $\eta(a_1, \ldots, a_n)$ is a semantic consequence of $P$.

Note that if $\eta(a_1, \ldots, a_n)$ is a semantic consequence of $P$ then, for any substitution $\eta'$, $\eta'(\eta(a_1, \ldots, a_n))$ is also a semantic consequence of $P$. The reason is that, if $\eta(a_1, \ldots, a_n)$ contains variables $X_1, \ldots, X_n$, then the meaning of $\eta(a_1, \ldots, a_n)$ is $\forall X_1, \ldots, \forall X_n \eta(a_1, \ldots, a_n)$.

**Proof of Soundness Theorem 2.** Assume that $\models t : \theta(\tau)$, and $p(t)$ evaluates successfully with computed answer substitution $\eta$. By Lemma 3.13, $\eta(p(t)) = p(\eta(t))$ is a semantic consequence of $P$. 42
3.5. Soundness Theorems

Let η′ be any ground substitution. Then \( p(\eta'(\eta(t))) \) is also a semantic consequence of \( P \). Thus from Lemma 3.12 it follows that \( p(\eta'(\eta(t))) \in M_P \). Now, \( \models t : \theta(\tau) \) implies that \( \eta'(\eta(t)) \in [\theta(\tau)] \). Hence, by the assumptions of the theorem, and by Theorem 3.1, \( \eta'(\eta(t)) \in [\theta(\sigma)] \), for any ground substitution \( \eta' \), which implies \( \models \eta(t) : \theta(\sigma) \). □

The converse of Theorem 3.1 does not hold. Let \( P \) be a program with a predicate \( p \). Let \( (p : \top \to \bot) \in T \). Consider two sentences.

1. A program \( P \) is well typed with respect to \( T \).
2. \( p(X) \) does not evaluate successfully (does not have an SLD-refutation).

From Theorem 3.1 we know that (1) implies (2) (it is because \( \bot \) = \( \emptyset \)). However, equivalence of (1) and (2) is impossible since the type checking in our system is decidable (thus (1) is decidable), while (2) is undecidable.
Chapter 3. Semantics
Chapter 4

Type Checking

In this chapter we analyze the problem of checking whether a program is well typed with respect to a directional type $T$, i.e. the problem of verifying if for each clause $C$ with the main predicate symbol $p$, and for each type $\tau \rightarrow \sigma$ such that $(p : \tau \rightarrow \sigma) \in T$ we have

$$\vdash \text{ClauseHasType}(T, C, \tau \rightarrow \sigma)$$

The rules for System B are almost syntax directed i.e. given $T, C, \tau \rightarrow \sigma$ almost determine a proof of (4.1). However, some questions should be more precisely analyzed:

1. How to check formulas of the form $\vdash \tau_1 \leq \tau_2$ (in rule $(T_4)$)?
2. Which type of the predicate should be taken while proving InferFromAtoms (in rule $(P_2)$)?
3. How to find substitutions used in the rules $(T_6), (K_2), (P_5)$?

We shall consider these questions in this chapter. This chapter is organized as follows. In Section 4.1 we investigate the problem of checking type formulas. In Section 4.2 we give two auxiliary functions $BS$ and $BT$ used by a type checking algorithm. $BS$ computes substitutions used in the proof of (4.1), $BT$ computes the least type of a term in a given environment. Section 4.3 provides a type checking algorithm, whose complexity is analyzed in Section 4.4. We close this chapter with the Section 4.5 in which we show the $\Sigma^P_2$ hardness of the type checking problem.

4.1 Checking Type Formulas

In this section we study the problem of checking validity of type formulas of the form $\vdash \tau_1 \leq \tau_2$. The section is organized as follows: in the Subsection 4.1.1 we define the notion of a normal form and give an algorithm which checks type equations transforming them in normal forms. This algorithm works in EXPTIME. In Subsection 4.1.2 we define a class of discriminative types and state the connection between all discriminative types and ground discriminative types. In Subsection 4.1.3 we analyze ground type formulas, show an algorithm which in polynomial time checks validity of discriminative type formulas. Then, using this algorithm we show that the problem of checking validity of type formulas is in $\text{co-NP}$.

**Remark 4.1.** To make the presentation more clear we treat atomic types as a syntactic sugar according to the following schema. Let $a$ be an atomic type, let $\text{Constr}(a) = \{s_1, \ldots, s_n\}$. We can replace every occurrence of $a$ in a checked formula
by

\[ F^{s_1}(\bot) \cup \cdots \cup F^{s_n}(\bot) \]

where each \( F^{s_i} \) is an unary type constructor, specific to a term constructor \( s_i \). Therefore in this chapter we will not consider atomic types.

### 4.1.1 Algorithm

Before we present an algorithm which checks whether \( \vdash \tau \leq \sigma \), we develop some tools. We will need two normal forms of types.

#### Normal Forms

The notion of normal forms is similar to notion of disjunctive and conjunctive normal forms of Boolean formulas.

**Definition 4.2.** A type \( \tau \) is in \textit{L-normal form} if it either belongs to \{\top, \bot\} or the following condition holds:
1. \( \tau \) has the form \( \tau_1 \cup \cdots \cup \tau_n \) (for \( n \geq 1 \)),
2. \( \vdash \tau_i \leq \tau_j \) does not hold, for each \( i, j \in \{1, \ldots, n\} \) such that \( i \neq j \),
3. Each \( \tau_i \) has the form \( \alpha_1^i \cap \cdots \cap \alpha_k^i \), or the form \( \alpha_1^i \cap \cdots \cap \alpha_k^i \cap F_i(\sigma_1^i, \ldots, \sigma_{m_i}^i) \),
   where \( \{\alpha_1^i, \ldots, \alpha_k^i\} \) are distinct variables, and each \( \sigma^i \) is in L-normal form.

**Definition 4.3.** The type \( \tau \) is in \textit{R-normal form} if it is either \( \top \) or \( \bot \) or the following conditions hold:
1. \( \tau \) has the form \( \tau_1 \cap \cdots \cap \tau_n \) (for \( n \geq 1 \)),
2. \( \vdash \tau_i \leq \tau_j \) does not hold for each \( i, j \in \{1, \ldots, n\} \) such that \( i \neq j \),
3. Each \( \tau_i \) has the form \( \alpha_1^i \cup \cdots \cup \alpha_k^i \), or the form
   \[ \alpha_1^i \cup \cdots \cup \alpha_k^i \cup \bigcup_{j=1}^{m_i} F_j^i(\sigma_{j,1}^i, \ldots, \sigma_{j,m_j}^i) \]
   where \( \{\alpha_1^i, \ldots, \alpha_k^i\} \) are distinct variables, and each \( \sigma^i \) is in R-normal form.
4. For each \( i \in \{1, \ldots, n\} \) and for each \( k, l \in \{1, \ldots, m_i\} \) such that \( k \neq l \) we have that
   \[ \vdash F_k^i(\sigma_{k,1}^i, \ldots, \sigma_{k,m_k}^i) \leq F_l^i(\sigma_{l,1}^i, \ldots, \sigma_{l,m_l}^i) \]
   does not hold.

Every type can be transformed to both normal forms. To prove it we define the operation \( f_{\text{ac}} \) which changes the arrangements of brackets and then sorts items in unions and products occurring in types. This operation can be seen as transforming to some kind of a pre-normal form.

**Definition 4.4.** A function \( f_{\text{ac}} \) is a function which for a type \( \tau \) returns a type in which arrangements of brackets in products is changed in such a way that every product has the form

\[
(4.2) \quad \tau_{k_1} \cap (\tau_{k_2} \cap (\ldots \cap \tau_{k_n} \ldots))
\]

where each \( \tau_{k_i} \) is not a product. In the remaining part of this chapter we will write (4.2) as \( \tau_{k_1} \cap \cdots \cap \tau_{k_n} \). The same changes are done for unions.
4.1. Checking Type Formulas

A function $f_a$ is a function which for a type $\tau$ returns a type. It changes the order of items in every product $\tau_1 \cap \cdots \cap \tau_k$ in such a way that in the result $\bot$ is before $\top$, $\bot$ is before unions (types with $\cup$ in the outer position), unions are before type variables, type variables are before types with type constructor.

Similarly, for each union $\tau_1 \cup \cdots \cup \tau_k$ the order is changed in such a way that in the result $\bot$ is before $\top$, $\bot$ is before products, products are before type variables, type variables are before types with type constructor.

We put $f_{ac} = f_a \circ f_a \qed$

Remark 4.5. Axioms (Ax8), (Ax7) imply $\vdash \tau = f_{ac}(\tau)$

Definition 4.6. $N_L$ is a function from types to types. We define

$$N_L(\tau) = N'_L(f_{ac}(\tau))$$

where $N'_L$ is defined as follows

(4.3) \[ N'_L(y) = y \text{ if } y \text{ is } \top, \bot, \text{ or a type variable} \]

(4.4) \[ N'_L(\tau_1 \cap \tau_2) = N_L(\tau_1) \cup N_L(\tau_2) \]

(4.5) \[ N'_L(\bot \cap \tau) = \bot \]

(4.6) \[ N'_L(\top \cap \tau) = N_L(\tau) \]

(4.7) \[ N'_L(\alpha \cap \tau) = \alpha \cap N_L(\tau) \]

(4.8) \[ N'_L((\tau_2 \cup \tau_3) \cap \tau_1) = N_L(\tau_2 \cap \tau_1) \cup N_L(\tau_3 \cap \tau_1) \]

(4.9) \[ N'_L(\bigcap_{i=1}^m F_i(\tau_1^i, \ldots, \tau_n^i)) = \top \text{ if } F_i \neq F_j \text{ for some } i, j \]

(4.10) \[ N'_L(\bigcap_{i=1}^m F(\tau_1^i, \ldots, \tau_n^i)) = F(N_L(\bigcap_{i=1}^m \tau_1^i), \ldots, N_L(\bigcap_{i=1}^m \tau_n^i)) \]

where $\alpha$ is a type variable. \qed

Remark 4.7. Let us notice that types returned by $f_{ac}$ are of the form to which one case of Definition 4.6 applies. In fact a type $\tau$ which is a result of $f_{ac}$ can be

1. $\bot, \top$ or a variable — (4.3) applies,
2. a union — (4.4) applies,
3. $\tau = \tau_1 \cap \cdots \cap \tau_k$. In this case if $\tau_1 = \bot$ then (4.5) applies, if $\tau_1 = \top$ (4.6) applies, if $\tau_1$ is a type variable then (4.7) applies, if $\tau_1$ is a union (4.8) applies, otherwise all $\tau_1$ are types with type constructors in the outer position and either (4.9) or (4.10) applies.

Definition 4.8. The function $N_R$ takes and returns types. We define

$$N_R(\tau) = N'_R(f_{ac}(\tau)),$$

where

$N'_R(y) = y$ if $y$ is $\top, \bot, \text{ or a type variable}$

$N'_R(\tau_1 \cap \tau_2) = N_R(\tau_1) \cap N_R(\tau_2)$

$N'_R(\alpha \cup \tau) = \alpha \cup N_R(\tau)$

$N'_R(\top \cap \tau) = \top$

$N'_R(\bot \cup \tau) = N_R(\tau)$

$N'_R((\tau_1 \cup \tau_2) \cup \tau_3) = N_R(\tau_1 \cup \tau_2) \cup N_R(\tau_2 \cup \tau_3)$

$N'_R(\bigcap_{i=1}^m F_i(\tau_1^i, \ldots, \tau_n^i)) = \bigcup_{i=1}^m F(N_R(\tau_1^i), \ldots, N_R(\tau_n^i))$
where $\alpha$ is a variable.

Now, we state some properties of functions $N_L$ and $N_R$.

**Lemma 4.9.** For each type $\tau$, $\vdash \tau = N_L(\tau)$

*Proof.* In Appendix.

**Corollary 4.10.** If $\vdash \tau_1 = \tau_2$ then $\vdash N_L(\tau_1) = N_L(\tau_2)$

*Proof.* By Lemma 4.9 we have that $\vdash \tau_1 = N_L(\tau_1)$ and $\vdash \tau_2 = N_L(\tau_2)$. So, the conclusion follows from (Ax5).

We state the properties of $N_R$ without any proof.

**Lemma 4.11.** For any type $\tau$ $\vdash \tau = N_R(\tau)$. Moreover, if $\vdash \tau_1 = \tau_2$ then $\vdash N_R(\tau_1) = N_R(\tau_2)$

**Remark 4.12.** If $\vdash \tau_1 = \tau_2$ then $\tau_1, \tau_2$ have L-normal forms which are equal modulo AC.

**Definition 4.13.** A type $\tau$ is $L$-reduced if $\tau = N_L(\tau)$. A type $\tau$ is $R$-reduced if $\tau = N_R(\tau)$.

In algorithms there is no need to use normal forms — algorithm can operate on L-reduced, (R-reduced) types. However, normal forms are useful in some proofs.

**Remark 4.14.** The L-normal form can be obtained by applying $N_L$ and then deleting redundant elements, i.e. repeated variables in subexpressions of the form $\alpha_1 \cap \cdots \cap \alpha_k$ and the types $\tau_i$ occurring in unions $\tau_1 \cup \cdots \cup \tau_m$, for which $\vdash \tau_i \leq \tau_j$, $j \in \{1, \ldots, m\}, j \neq k$.

When it does not cause any misunderstandings, we write $F(\tau)$ instead of $F(\tau_1, \ldots, \tau_n)$, and $F(\tau_1 \cup \tau_2)$ instead of $F(\tau_1^1 \cup \tau_1^2, \ldots, \tau_2^1 \cup \tau_2^2)$, and similarly $F(\tau_1 \cap \tau_2)$ instead of $F(\tau_1^1 \cap \tau_1^2, \ldots, \tau_2^1 \cap \tau_2^2)$.

Before we present the algorithm, we describe some technical properties of the relation $\leq$.

**Lemma 4.15.** If

$$\vdash \bigcap_{i=1}^{m} \alpha_i \cap F(\tau) = \bigcap_{i=1}^{m_1} \alpha_i^1 \cap F(\tau_1) \cup \cdots \cup \bigcap_{i=1}^{m_n} \alpha_i^m \cap F(\tau_n)$$

Then for some $i$ we have

$$\vdash \bigcap_{j=1}^{m} \alpha_j \cap F(\tau) = \bigcap_{j=1}^{m_i} \alpha_j^i \cap F(\tau_i)$$

*Proof.* We will use the L-normal form. The L-normal form of the left-hand side of (4.11) is $\bigcap_{j=1}^{m'} \beta_j \cap F(\tau')$ where $\tau'$ is the normal form of $\tau$ and $\{\beta_1, \ldots, \beta_{m'}\} = \{\alpha_1, \ldots, \alpha_m\}$, and all variables $\beta_j$ are distinct. The equation (4.11) holds if the normal forms of left-hand side and right-hand side of (4.11) are the same modulo AC. Transforming $\bigcap_{i=1}^{m_1} \alpha_i^1 \cap F(\tau_1) \cup \cdots \cup \bigcap_{i=1}^{m_n} \alpha_i^m \cap F(\tau_n)$ to the normal form can be done in two steps. First we compute the normal form $\tau'_i$ of each $\tau_i$. We obtain

$$\bigcap_{i=1}^{m_1} \alpha_i^1 \cap F(\tau'_1) \cup \cdots \cup \bigcap_{i=1}^{m_n} \alpha_i^m \cap F(\tau'_n)$$

48
Then we delete all redundant products (i.e. products which are in relation $\leq$ to some other), and all repeating variables in $\bigcap_{i=1}^{k} \alpha_i$, for each $k$. Since normal forms of both sides of (4.11) are equal modulo AC, only one product can remain and this gives the thesis.

\[ \vdash \bigcap_{i=1}^{k} \alpha_i \leq \bigcup_{i=1}^{m} \beta_i \cup \bigcup_{i=1}^{n} F(\tau_i) \cup \bigcup_{i=1}^{p} G_i(\sigma_i) \]

if and only if

\[ \vdash \bigcap_{i=1}^{k} \alpha_i \leq \bigcup_{i=1}^{m} \beta_i \text{ or } \vdash \tau \leq \tau_i \text{ for some } i \in \{1, \ldots, n\} \]

Proof. In appendix.

We will use this lemma to decompose the types while checking whether relation $\leq$ holds.

Definition 4.17. The formula $\tau_1 \leq \tau_2$ is in the normal form if $\tau_1$ is in L-normal form and $\tau_2$ is in R-normal form. It is in the LR-reduced form if $\tau_1$ is L-reduced and $\tau_2$ is R-reduced.

Definition 4.18. The formula $\tau_1 \leq \tau_2$ is a basic type formula if it is of the form

\[ \vdash \alpha_1 \cap \cdots \cap \alpha_n \leq \beta_1 \cup \cdots \cup \beta_k \]

where $\alpha_i$, $\beta_j$ are type variables.

Lemma 4.19. The basic type formula $\vdash \alpha_1 \cap \cdots \cap \alpha_n \leq \beta_1 \cup \cdots \cup \beta_k$ is provable iff the following condition is fulfilled:

\[ \{\alpha_1, \ldots, \alpha_n\} \cap \{\beta_1, \ldots, \beta_n\} \neq \emptyset \]

Proof.

\[ \vdash \alpha_1 \cap \cdots \cap \alpha_n \leq \beta_1 \cup \cdots \cup \beta_k \]

holds iff

\[ \vdash \alpha_1 \cap \cdots \cap \alpha_n \cap (\beta_1 \cup \cdots \cup \beta_k) = \alpha_1 \cap \cdots \cap \alpha_n \]  \[ \vdash (\alpha_1 \cap \cdots \cap \alpha_n \cap \beta_1) \cup \cdots \cup (\alpha_1 \cap \cdots \cap \alpha_n \cap \beta_k) = \alpha_1 \cap \cdots \cap \alpha_n \]  \[ \vdash (\alpha_1 \cap \cdots \cap \alpha_n \cap \beta_1) \cup \cdots \cup (\alpha_1 \cap \cdots \cap \alpha_n \cap \beta_k) = \alpha_1 \cap \cdots \cap \alpha_n \]  \[ \vdash (\alpha_1 \cap \cdots \cap \alpha_n \cap \beta_1) \cup \cdots \cup (\alpha_1 \cap \cdots \cap \alpha_n \cap \beta_k) = \alpha_1 \cap \cdots \cap \alpha_n \]

Equation (4.15) is equivalent to

(4.16) $\vdash (\alpha_1 \cap \cdots \cap \alpha_n \cap \beta_1) \cup \cdots \cup (\alpha_1 \cap \cdots \cap \alpha_n \cap \beta_k) = \alpha_1 \cap \cdots \cap \alpha_n$.

Without any loss of generality we can assume that all $\alpha_i$ are distinct variables. The right-hand side of (4.16) is in L-normal form. Moreover, (4.16) is fulfilled iff the normal form of its left-hand side is equal to $\alpha_1 \cap \cdots \cap \alpha_n$ modulo AC. Suppose a contrario that for all $\beta_i$ we have $\beta_i \notin \{\alpha_1, \ldots, \alpha_n\}$. Then the left-hand side of (4.16) is in the L-normal form, which is different from $\alpha_1 \cap \cdots \cap \alpha_n$ and (4.16) does not hold. So, there exists $\beta_i \in \{\alpha_1, \ldots, \alpha_n\}$, so $\{\alpha_1, \ldots, \alpha_n\} \cap \{\beta_1, \ldots, \beta_n\} \neq \emptyset$.
Chapter 4. Type Checking

We will show how to decompose the formulas in LR-reduced form into the positive Boolean formula with basic type formulas in the leaves.

**Definition 4.20.** Let \( \tau_1 \leq \tau_2 \) be in LR-reduced form. Then the value of the function \( SF(\tau_1 \leq \tau_2) \) is a Boolean formula built from basic type formulas. It is given by the following inductive definition.

\[
(4.17) \quad SF(\tau_1 \cup \cdots \cup \tau_n \leq \sigma_1 \cap \cdots \cap \sigma_m) = \bigwedge_{i \in \{1, \ldots, n\}, \quad j \in \{1, \ldots, m\}} SF(\tau_i \leq \sigma_j)
\]

\[
(4.18) \quad SF\left( \bigcap_{i=1}^{k} \alpha_i \cap F(\tau) \leq \bigcup_{i=1}^{m} \beta_i \cup \bigcup_{i=1}^{n} F(\tau_i) \cup \bigcup_{i=1}^{p} G_i(\sigma_i) \right) = \left( \vdash \bigcap_{i=1}^{k} \alpha_i \leq \bigcup_{i=1}^{m} \beta_i \right) \lor \bigvee_{i=1}^{n} SF(\tau \leq \tau_i)
\]

In this definition we use the convention which says that union of zero elements is equal to \( \bot \) and the disjunction of zero elements is false. \( \square \)

Consider an example.

**Example 4.1.** Let \( \tau_1 \leq \tau_2 \) be a formula \( \alpha \cap \beta \cup \beta \cap F(\gamma) \leq (\beta \cup F(\gamma)) \cap \gamma \). In this case \( SF(\tau_1 \leq \tau_2) \) is an conjunction of four formulas:

1. \( SF(\alpha \cap \beta \leq \beta \cup F(\gamma)) \)
2. \( SF(\alpha \cap \beta \leq \gamma) \)
3. \( SF(\beta \cap F(\gamma) \leq \beta \cup F(\gamma)) \)
4. \( SF(\beta \cap F(\gamma) \leq \gamma) \)

Finally, we obtain

\[
SF(\tau_1 \leq \tau_2) = (\vdash \alpha \cap \beta \leq \beta) \\
\quad \land (\vdash \alpha \cap \beta \leq \gamma) \\
\quad \land (\vdash \beta \leq \beta \lor \gamma \leq \gamma) \\
\quad \land (\vdash \beta \leq \gamma)
\]

\( \square \)

To prove the properties of \( SF \) we need a few lemmas.

**Lemma 4.21.** Let \( \tau_1, \tau_2, \tau_3, \tau_4 \) are types. Suppose that \( \vdash \tau_1 \leq \tau_2 \) and \( \vdash \tau_3 \leq \tau_4 \). Then

\[
\vdash \tau_1 \cap \tau_3 \leq \tau_2 \cap \tau_4
\]

and

\[
\vdash \tau_1 \cup \tau_3 \leq \tau_2 \cup \tau_4
\]

**Proof.** We prove the thesis only for '\( \cap \)'. We have that

\[
\vdash \tau_1 = \tau_1 \cap \tau_2 \\
\vdash \tau_3 = \tau_3 \cap \tau_4
\]

50
By (Ax3) we obtain
\[ \vdash \tau_1 \cap \tau_3 = (\tau_1 \cap \tau_2) \cap (\tau_3 \cap \tau_4) = (\tau_1 \cap \tau_3) \cap (\tau_2 \cap \tau_4) \]
which, by definition of \( \leq \) is equivalent to
\[ \vdash \tau_1 \cap \tau_3 \leq \tau_2 \cap \tau_4 \]
\[ \Box \]

**Lemma 4.22.** For types \( \tau_1, \tau_2, \tau_3 \) we have
\[ \vdash \tau_1 \leq \tau_2 \cap \tau_3 \text{ if and only if } \vdash \tau_1 \leq \tau_2 \text{ and } \vdash \tau_1 \leq \tau_3 \]

**Proof.** Assume that \( \vdash \tau_1 \leq \tau_2 \cap \tau_3 \). Moreover, since \( \vdash \tau_2 \cap \tau_3 \cap \tau_3 = \tau_2 \cap \tau_3 \) we have that \( \vdash \tau_2 \cap \tau_3 \leq \tau_2 \). So, by the transitivity of the \( \leq \) we obtain \( \vdash \tau_1 \leq \tau_2 \).

Similarly, we can prove that \( \vdash \tau_1 \leq \tau_2 \cap \tau_3 \) implies \( \vdash \tau_1 \leq \tau_3 \). From the other hand assume that \( \vdash \tau_1 \leq \tau_2 \) and \( \vdash \tau_1 \leq \tau_3 \). By Lemma 4.21 we obtain \( \vdash \tau_1 = \tau_1 \cap \tau_1 \leq \tau_2 \cap \tau_3 \).
\[ \Box \]

**Lemma 4.23.** Let \( \tau_1, \tau_2, \tau_3 \) be arbitrary types. Then
\[ \vdash \tau_1 \cup \tau_2 \leq \tau_3 \text{ if and only if } \vdash \tau_1 \leq \tau_3 \text{ and } \vdash \tau_2 \leq \tau_3 \]

**Proof.** Assume that \( \vdash \tau_1 \cup \tau_2 \leq \tau_3 \). Moreover, since \( \vdash \tau_1 \cup \tau_1 \cup \tau_2 = \tau_1 \cup \tau_2 \), we have by Lemma 2.20 that \( \vdash \tau_1 \leq \tau_1 \cup \tau_2 \). So, by the transitivity of the \( \leq \) we obtain \( \vdash \tau_1 \leq \tau_3 \).

Similarly we get \( \vdash \tau_1 \cup \tau_2 \leq \tau_3 \Rightarrow \tau_2 \leq \tau_3 \). To prove the second direction we use Lemma 4.21.
\[ \Box \]

Using Lemma 4.22 and Lemma 4.23 we can easily prove three corollaries below.

**Corollary 4.24.** \( \vdash \bigcup_{i=1}^{n} \tau_i \leq \tau \) if and only \( \vdash \tau_i \leq \tau \) for each \( i \in \{1, \ldots, n\} \).

**Corollary 4.25.** \( \vdash \tau \leq \bigcap_{i=1}^{n} \tau_i \) if and only \( \vdash \tau \leq \tau_i \) for each \( i \in \{1, \ldots, n\} \)

**Corollary 4.26.** \( \vdash \bigcup_{i=1}^{m} \tau_i \leq \bigcap_{i=1}^{m} \sigma_i \Leftrightarrow \vdash \tau_i \leq \sigma_i \) for each \( i \in \{1, \ldots, n\}, j \in \{1, \ldots, m\} \)

The next lemma says that \( SF \) can be used in type checking, since it preserves validity of type formulas.

**Lemma 4.27.** Let \( \tau_1 \leq \tau_2 \) be the type equation in normal form. We have:
\[ \vdash \tau_1 \leq \tau_2 \text{ iff } SF(\tau_1 \leq \tau_2) \text{ is valid} \]

**Proof.** In Appendix.
\[ \Box \]

**Lemma 4.28.** The algorithm 4.1 checks whether \( \tau_1 \leq \tau_2 \) holds.

**Proof.** This lemma is a simple corollary from the Lemma 4.27.
\[ \Box \]

**Remark 4.29.** In this thesis we present several algorithms. They are written in an imperative pseudo-code. The code contains ‘standard’ program keywords, as for, if, then, else, let etc. We assume that the function stops and returns a value after a keyword return. Moreover, the structure of the code is given by tab positions. For instance:
Chapter 4. Type Checking

function LessSF(ζ₁, ζ₂)
ζ₁, ζ₂ — types to be compared
let τ₁ = N_L(ζ₁)
let τ₂ = N_R(ζ₂)
let Φ = SF(τ₁ ≤ τ₂)
if Φ holds then return ‘Yes’
else return ‘No’

Algorithm 4.1: Comparing types using SF

function Less(ζ₁, ζ₂)
ζ₁, ζ₂ — types to be compared
let τ₁ = N_L(ζ₁)
let τ₂ = N_R(ζ₂)
let Pᵢ be such that τ₁ = ∪ᵢ₌₁ⁿ Pᵢ
let Sᵢ be such that τ₂ = ∩ᵢ₌₁ⁿ Sᵢ
for i ∈ {1, ⋯, m}, j ∈ {1, ⋯, n} do
  let Pᵢ = ∩ₖ₌₁ᵖ αᵢ,k ∩ Fᵢ(τᵢ)
  let Sᵢ = ∪ₖ₌₁₁ β₉,k ∪ ∪ₖ₌₁ⁿ F₉(τ₉,k) ∪ ∪ₖ₌₁ⁿ G₉,k(σ₉,k)
  if {αᵢ₁, ⋯, αᵢₚ} ∩ {β₉₁, ⋯, β₉ₙ} = ∅ then
    for k = 1 to r do
      if Less(τᵢ', τ₉,k) then continue loop for next i, j
    return ‘Yes’
return ‘No’

Algorithm 4.2: Comparing types (without SF)

if A then
  Iₐ
  if A then I₉
  else I₉

is interpreted as Pascal program

if A then
  begin
    Iₐ;
    if B then Ib;
  end
else I₉;

It is not necessary to compute the whole SF(τ₁ ≤ τ₂). In most cases the logical value of it can be known earlier. Algorithm 4.2 is a variant of the algorithm 4.1 which checks only necessary conditions. It is based on the same idea as SF but it does not use the function SF.

We define the size of the type:
\textbf{Definition 4.30.} For a term \( \tau \) its size denoted by \( |\tau| \) is defined in the following way:
\[
|\alpha| = 1 \\
|F(\tau_1, \ldots, \tau_n)| = 1 + \sum_{i=1}^{n} |\tau_i| \\
|\tau_1 \cup \tau_2| = 1 + |\tau_1| + |\tau_2| \\
|\tau_1 \cap \tau_2| = 1 + |\tau_1| + |\tau_2|
\]
Moreover \( |\tau_1| \leq |\tau_2| = |\tau_1| + |\tau_2| + 1 \).

\textbf{Lemma 4.31.} If \( N \) is the size of a formula \( \tau_1 \leq \tau_2 \), where \( \tau_1 \) is in L-reduced form, and \( \tau_2 \) is in R-reduced form, then algorithm 4.2 answers ‘Yes’ if and only if \( \vdash \tau_1 \leq \tau_2 \). Moreover, it gives an answer after at most \( O(N^2) \) steps.

\textbf{Proof.} The algorithm 4.2 gives the same answer as the algorithm 4.1 which is correct by Lemma 4.28.

Let \( L \) be the size of \( \tau_1 \), and let \( R \) be the size of \( \tau_2 \). These sizes are equal to sizes of arguments since \( N_L \) is an identity on L-reduced types and \( N_R \) is an identity on R-reduced types. We show that the algorithm gives the answer after \( LR \) steps which, of course, is \( O(N^2) \). We use induction on the length of the formula in the LR-reduced form.

\begin{itemize}
  \item We check the equation of the form:
  \[
  (4.19) \quad \tau_1 \cup \cdots \cup \tau_n \leq \sigma_1 \cap \cdots \cap \sigma_m
  \]
  Let \( L_i = |\tau_i| \) and let \( R_i = |\sigma_i| \). Induction hypothesis says that each \( \tau_i \leq \sigma_j \) can be checked in \( L_i R_j \) steps. Let \( S \) be the number of steps needed to check (4.19). We have that
  \[
  (4.20) \quad S \leq \sum_{i \leq n, j \leq m} L_i R_j = (L_1 + \cdots + L_n)(R_1 + \cdots + R_m) \leq LR
  \]
  \item We check the equation of the form:
  \[
  (4.21) \quad \bigcap_{i=1}^{k} \alpha_i \cap F(\tau) \leq \bigcup_{i=1}^{m} \beta_i \cup \bigcup_{i=1}^{n} F(\tau_i) \cup \bigcup_{i=1}^{p} G(\sigma_i)
  \]
  Let \( L \) be the size of the left-hand side of (4.21) and \( R \) be the size of the right-hand side of it. Let \( L' = |\tau| \) and let \( R_i = |\tau_i| \). Let \( S \) be the number of steps needed to check (4.21). From the induction hypothesis we have that \( \tau \leq \tau_1 \) can be checked in \( LR_i \) steps. The condition
  \[
  (4.22) \quad \bigcap_{i=1}^{k} \alpha_i \leq \bigcup_{i=1}^{m} \beta_i
  \]
  by Lemma 4.19 can be checked in time \( km \). We have
  \[
  (4.23) \quad S \leq km + \sum_{i \in \{1, \ldots, n\}} L'R_i \leq (k + L')(m + \sum_{i \in \{1, \ldots, n\}} R_i) \leq LR
  \]
  as claimed.
\end{itemize}

Transforming type to the normal form can cause the exponential increase of the length of the formula.
4.2 The L-normal form of the type

\[(\alpha_1^0 \cup \alpha_1^1) \cap \cdots \cap (\alpha_n^0 \cup \alpha_n^1)\]

is a union of all types of the form \(\alpha_i^j \cap \cdots \cap \alpha_n^j\), where \(i_j \in \{0, 1\}\), for \(j \in \{1, \ldots, n\}\), so it has \(2^n\) elements.

**Lemma 4.32.** Assume that \(\tau\) has size \(n\). Then the size of \(N_L(\tau)\) is less than \(2^n\). Moreover, the size of \(N_R(\tau)\) is less than \(2^n\).

**Proof.** In Appendix.

The next lemma is a corollary to the Lemmas 4.31 and 4.32.

**Lemma 4.33.** The algorithm 4.2 works in EXPTIME. It has a complexity \(O(2^{N^2})\)

The algorithms 4.1 and 4.2 do not work in PSPACE since the normal forms can have exponential size. Now, we develop the tools to prove that there is an algorithm which, for some types, works in co-NP.

### 4.1.2 Discriminative Types

Now, we define and analyze a useful class of discriminative types. The notion of discriminative types is strongly related to the notion of tuple distributive types (see e.g., [26]) which are accepted as a good approximation of the success set of a predicate. It is also related to set based approximation. The following definition is adapted from [27].

**Definition 4.34.** Let \(S\) be the set of ground terms. The **tuple distributive closure** of \(S\), written \(TDC(S)\), is recursively defined as

\[
TDC(S) = \{c : c \text{ is a constant, } c \in S\}
\]

\[
\cup \{f(t_1, \ldots, t_n) : t_i \in TDC(f_i^{-1}(S)), i \in \{1, \ldots, n\}\}
\]

where \(f\) is a term constructor, and \(f_i^{-1}(S)\) is defined as

\[
\{t_i : f(t_1, \ldots, t_i, \ldots, t_n) \in S\}
\]

We present some basic intuitions connected with the notion of tuple distributive closure.

**Example 4.3.** Let \(S = \{f(a, b), f(c, d)\}\), then

\[
TDC(S) = \{f(a, b), f(c, d), f(a, d), f(c, b)\}
\]

**Example 4.4.** Let \(B\) be the set of all Prolog lists containing only constant \(b\), such as e.g. \([b, b, b, b]\), let \(C\) be the set of all prolog lists containing only constant \(c\), such as e.g. \([c, c, c]\). Then tuple distributive closure of the set \(B \cup C\) is the set of all lists containing constants \(b, c\), such as \([c, c, b, c, b]\).

Now, we define the function \(A\), a discriminative approximation of types. To do this we first define function \(A'\) operating on types transformed to R-normal form.
4.1. Checking Type Formulas

**Definition 4.35.** Let $\tau$ be a type in R-normal form. We define the function $A'$:

$$A'(\tau_1 \cap \cdots \cap \tau_n) = A'(\tau_1) \cap \cdots \cap A'(\tau_n)$$

$$A'(\bigcup_{i=1}^{m} \beta_i \cup \bigcup_{i=1}^{p} \bigcup_{j=1}^{q_i} F_i(\tau_{i1}, \ldots, \tau_{in})) = \bigcup_{i=1}^{m} \beta_i \cup \bigcup_{i=1}^{p} \bigcup_{j=1}^{q_i} F_i(A'(\bigcup_{i=1}^{m} \beta_i), \ldots, A'(\bigcup_{i=1}^{m} \beta_i))$$

**Definition 4.36.** Let $\tau$ be a type, and let $\tau'$ be its R-normal form. We define a **discriminative approximation** of the type $\tau$ as $A(\tau) = A'(\tau')$.

Informally speaking $A(\tau)$ can be obtained from $\tau$ by transforming $\tau$ to R-normal form and then by repeatedly transforming each subexpression $F(\tau_1, \ldots, \tau_n) \cup F(\sigma_1, \ldots, \sigma_n)$ into $F(\tau_1 \cup \sigma_1, \ldots, \tau_n \cup \sigma_n)$.

**Example 4.5.** Let $b$ has the only signature $b : B$, and let $c$ has the only signature $c : C$. We have $A(\text{list}(B) \cup \text{list}(C)) = \text{list}(B \cup C)$. So, we have

$$[[A(\text{list}(B) \cup \text{list}(C))]] = [[\text{list}(B \cup C)]] = \text{TDC}([[\text{list}(B)] \cup [[\text{list}(C)]])$$

where $[\cdot]$ is an interpretation function defined in chapter 3.

We state the following property of TDC without any proof:

**Remark 4.37.** If $\tau$ is a ground type whose R-normal form does not contain $\bot$ then $\text{TDC([\tau])} = [\tau]$.

Now, we give an example which shows that there are situations when the type inferred by System B is already the discriminative approximation of $\tau$.

**Example 4.6.** Assume that $\emptyset, ([X : \text{list}(\alpha) \cup \text{list}(\beta)]) \Rightarrow \Gamma$. Then $\Gamma \sim \{X : \alpha \cup \beta\}$. The best provable type of a term $[X]$ in environment $\Gamma$ is $\text{list}(\alpha \cup \beta)$, a discriminative approximation of $\text{list}(\alpha) \cup \text{list}(\beta)$.

The following two remarks are immediate consequences of definition of discriminative approximation:

**Remark 4.38.** For each type $\tau$ we have $A(\tau) = A(A(\tau))$

**Remark 4.39.** For each types $\tau_1, \tau_2$ we have that if $\vdash \tau_1 = \tau_2$ then $\vdash A(\tau_1) = A(\tau_2)$.

**Definition 4.40.** The type $\tau$ is **discriminative** if $\vdash \tau = A(\tau)$

**Example 4.7.** Type $\alpha \cup F(\beta \cup \gamma)$ is discriminative. Type $F(\alpha) \cup (\alpha \cap F(\beta))$ is not. In fact

$$A(F(\alpha) \cup (\alpha \cap F(\beta))) = (F(\alpha) \cup \alpha) \cap F(\alpha \cup \beta)$$

which is not equal to $F(\alpha) \cup \alpha \cap F(\beta)$.

Now, we show how to reduce the problem of checking discriminative formula $\tau_1 \leq \tau_2$ to checking the collection of ground formulas. First we state some lemmas and define some tools.

**Definition 4.41.** A type substitution is called $(\top, \bot)$-substitution if its range is $\{\bot, \top\}$.
In the rest of this section we will use only the $(\top, \bot)$-substitutions. To underline the restriction of the range we use for such substitutions the letter $\nu$ instead of $\theta$.

**Lemma 4.4.2.** Let $\tau_1, \tau_2$ be types built from variables and operators $\cap, \cup$. Then $\vdash \tau_1 \leq \tau_2$ if and only if, for each $(\top, \bot)$-substitution $\nu$, we have $\vdash \nu(\tau_1) \leq \nu(\tau_2)$.

**Proof.** First, using the equality axioms, we can transform $\tau_1, \tau_2$ to the appropriate normal forms obtaining:

\begin{align*}
\tau_1 &= (\alpha_1^1 \cap \cdots \cap \alpha_{k_1}^1) \cup \cdots \cup (\alpha_1^m \cap \cdots \cap \alpha_{k_m}^m) \\
\tau_2 &= (\beta_1^1 \cup \cdots \cup \beta_{k_1}^1) \cap \cdots \cap (\beta_1^m \cup \cdots \cup \beta_{k_m}^m)
\end{align*}

where each $\alpha_i^j$ and each $\beta_i^j$ is a variable. The condition $\tau_1 \leq \tau_2$, according to Corollary 4.26, is equivalent to

\begin{equation}
(\forall i \in \{1, \ldots, n\})(\forall j \in \{1, \ldots, m\}) (\alpha_i^1 \cap \cdots \cap \alpha_{k_i}^i) \leq (\beta_i^1 \cup \cdots \cup \beta_{k_i}^i)
\end{equation}

We want to show that, if for each $(\top, \bot)$-substitution $\nu$, (4.27) holds then $\vdash (\alpha_i^1 \cap \cdots \cap \alpha_{k_i}^i) \leq (\beta_i^1 \cup \cdots \cup \beta_{k_i}^i)$ is a provable formula. So, let us suppose that, for $(\top, \bot)$-substitution $\nu$, (4.27) holds. Let us take any $i' \in \{1, \ldots, n\}, j' \in \{1, \ldots, m\}$. Suppose a contrario that all variables $\alpha_i^j$ are different from $\beta_i^j$. Then, by Lemma 4.19 the formula $(\alpha_i^1 \cap \cdots \cap \alpha_{k_i}^i) \leq (\beta_i^1 \cup \cdots \cup \beta_{k_i}^i)$ is not provable. Let $\nu'$ be the substitution defined as follows:

$$\nu'(\gamma) = \begin{cases} 
\top & \text{if } \gamma \in \{\alpha_1^{i'}, \ldots, \alpha_n^{i'}\} \\
\bot & \text{if } \gamma \in \{\beta_1^{j'}, \ldots, \beta_n^{j'}\}
\end{cases}$$

Then in (4.27) for $i = i', j = j'$ we have

$$\vdash \nu'(\alpha_i^{i'} \cap \cdots \cap \alpha_{k_i}^{i'}) \leq (\beta_i^{j'} \cup \cdots \cup \beta_{k_i}^{j'})$$

After applying $\nu'$ we obtain $\vdash \top \leq \bot$. This is a contradiction according to Lemma 3.2. So, there is a pair of equal variables and $\vdash \tau_1 \leq \tau_2$.

To prove the other direction suppose that $\vdash \tau_1 \leq \tau_2$. By Lemma 2.23 we get that for each substitution $\nu$ we have that $\vdash \nu(\tau_1) \leq \nu(\tau_2)$. \qed

As we have seen one can check formulas without type constructors by checking their values for all $(\top, \bot)$-substitutions.

For the types with type constructors it is not the case. Consider the examples:

**Example 4.8.** It is not the case that $\vdash \beta \cap F(\alpha) \leq F(\beta)$. However for each $(\top, \bot)$-substitution $\nu$ we have $\nu(\beta \cap F(\alpha)) \leq \nu(F(\beta))$.

<table>
<thead>
<tr>
<th>$\nu$</th>
<th>$\nu(\beta \cap F(\alpha))$</th>
<th>$\nu(F(\beta))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot, \bot$</td>
<td>$\bot$</td>
<td>$F(\bot)$</td>
</tr>
<tr>
<td>$\bot, \top$</td>
<td>$F(\bot)$</td>
<td>$F(\bot)$</td>
</tr>
<tr>
<td>$\top, \bot$</td>
<td>$\bot$</td>
<td>$F(\bot)$</td>
</tr>
<tr>
<td>$\top, \top$</td>
<td>$F(\top)$</td>
<td>$F(\top)$</td>
</tr>
</tbody>
</table>

\[\square\]

**Example 4.9.** It is not the case that $\vdash F(\alpha \cup \beta) \leq F(\alpha) \cup F(\beta)$. But again for all $(\top, \bot)$-substitution $\nu$ this equation is fulfilled:

<table>
<thead>
<tr>
<th>$\nu$</th>
<th>$\nu(F(\alpha \cup \beta))$</th>
<th>$\nu(F(\alpha) \cup F(\beta))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bot, \bot$</td>
<td>$F(\bot)$</td>
<td>$F(\bot)$</td>
</tr>
<tr>
<td>$\bot, \top$</td>
<td>$F(\bot)$</td>
<td>$F(\bot)$</td>
</tr>
<tr>
<td>$\top, \bot$</td>
<td>$F(\top)$</td>
<td>$F(\top)$</td>
</tr>
<tr>
<td>$\top, \top$</td>
<td>$F(\top)$</td>
<td>$F(\top)$</td>
</tr>
</tbody>
</table>

56
Notice that the type $F(\alpha) \cup F(\beta)$ is not discriminative.

Now, we define some notions.

**Definition 4.43.** The depth of an occurrence of a variable $\alpha$ in a type $\tau$ is the number of function symbols on the path from the root of $\tau$ to this occurrence of $\alpha$.

The function $l$ replaces each occurrence of each variable $\alpha$ in $\tau$ by the variable $\alpha^k$ where $k$ is the depth of this occurrence. Formally:

**Definition 4.44.** Let $k$ be a natural number. Then $l_k$, a function from types to types is defined as follows:

\[
l_k(\alpha) = \alpha^k
\]

\[
l_k(\tau_1 \cup \tau_2) = l_k(\tau_1) \cup l_k(\tau_2)
\]

\[
l_k(\tau_1 \cap \tau_2) = l_k(\tau_1) \cap l_k(\tau_2)
\]

\[
l_k(F(\tau_1, \ldots, \tau_n)) = F(l_{k+1}(\tau_1), \ldots, l_{k+1}(\tau_n))
\]

We use $l(\tau)$ as an abbreviation of $l_0(\tau)$. We can also apply $l_k$ to type formulas according to the following definition:

\[
l_k(\tau \leq \sigma) = (l_k(\tau) \leq l_k(\sigma))
\]

**Example 4.10.** We have:

\[
l(\beta \cup G(\alpha \cap F(\beta \cup \alpha))) = \beta^0 \cup G(\alpha^2 \cap F(\beta^2 \cup \alpha^2))
\]

The following are easy:

**Remark 4.45.** If there is a subterm $\bigcap_{i=1}^n \alpha_i \cap F(\sigma)$ of a term $\tau$ then in $l_k(\tau)$ there is a subterm $\bigcap_{i=1}^n \alpha_i^{k'} \cap F(l_{k+1}(\sigma))$ and $\{\alpha_1, \ldots, \alpha_n\} \cap \text{var}(l_{k+1}(\tau)) = \emptyset$, where $k' \geq k$.

The function $l$ will be used to stratify variables. After applying it to $\bigcap \alpha_i \cap F(\tau)$ we obtain different variables in both `layers', i.e. in $\bigcap \alpha_i$ and in $F(\tau)$.

**Lemma 4.46.** Let $\tau, \sigma$ be types. Let $k$ be any integer. If the basic type formula $\vdash \bigcap_{i=1}^m \alpha_i^{a_i} \leq \bigcup_{i=1}^n \beta_i^{b_i}$ is a subformula of $SF(l_k(\tau \leq \sigma))$ then for each $i, j$ we have $a_i = b_j$.

**Proof.** In Appendix.

Now, we define the height of the type:

**Definition 4.47.** For a type $\tau$ its height denoted by $h(\tau)$ is defined as follows:

\[
h(\alpha) = 0
\]

\[
h(\alpha) = 0
\]

\[
h(\tau_1 \cup \tau_2) = \max(h(\tau_1), h(\tau_2))
\]

\[
h(\tau_1 \cap \tau_2) = \max(h(\tau_1), h(\tau_2))
\]
where $\alpha$ is an atomic type. Moreover, let $h(\tau_2 \leq \tau_2) = \max(h(\tau_2), h(\tau_2))$.

Now, we state the main result of this section.

**Lemma 4.48.** Let $\tau_1, \tau_2$ be discriminative types. Let $k \geq 0$. The formula $l_k(\tau_1 \leq \tau_2)$ is true for any $(\top, \bot)$-substitution if and only if

$$\vdash \tau_1 \leq \tau_2$$

**Proof.** Let

$$E_1 = (\tau_1 \leq \tau_2)$$
$$E_2 = l_k(\tau_1 \leq \tau_2)$$

The if part.

Assume $\vdash \tau_1 \leq \tau_2$. We will show that

$$(4.28) \quad SF(E_1) \Rightarrow SF(E_2)$$

Let us consider a basic type formula

$$(4.29) \quad \alpha_1 \sqcap \cdots \sqcap \alpha_n \leq \beta_1 \sqcup \cdots \sqcup \beta_k$$

in $SF(E_1)$. Suppose, that it is provable. So, by Lemma 4.19, there are $i, j$ s.t. $\alpha_i = \beta_j$ for some $i, j$. By Lemma 4.46 in $SF(E_2)$ on the same position we have

$$(4.30) \quad \alpha_i^m \sqcap \cdots \sqcap \alpha_n^m \leq \beta_i^m \sqcup \cdots \sqcup \beta_k^m$$

for some integer $m$.

Both $SF(E_1), SF(E_2)$ are Boolean formulas which have the same shape but differ in their leaves and we have just proved that any leaf from $SF(E_1)$ implies a corresponding leaf in $SF(E_2)$. So, since both $SF(E_1), SF(E_2)$ are positive, we can conclude that $SF(E_2)$ holds. So, we can apply the Lemma 4.27 two times to obtain:

$$(4.31) \quad (\vdash E_1) \Rightarrow (SF(E_1)) \Rightarrow (SF(E_2)) \Rightarrow (\vdash E_2)$$

which, by Lemma 2.23, implies $(\forall \nu) \vdash \nu(E_2)$, i.e. $(\forall \nu) \vdash \nu(l_k(\tau_1 \leq \tau_2))$.

The only if part.

The proof proceeds by induction on the height of the formula. The basic case (height 0) is covered by Lemma 4.42. Now, suppose that the thesis holds for each formula of the height $H', H' < H$, where $H = h(E_2)$. We are going to show that it also holds for $H$.

Suppose that $E_2$ is true for any $(\top, \bot)$-substitution. $E_2$ is in normal form. Let $\Pi_1, \ldots, \Pi_p$ and $\Sigma_1, \ldots, \Sigma_s$ be such that

$$(4.32) \quad E_2 = l_k(\Pi_1 \sqcup \cdots \sqcup \Pi_p \leq \Sigma_1 \sqcap \cdots \sqcap \Sigma_s)$$

Let $\nu$ be a $(\top, \bot)$-substitution. So, we have

$$\nu(E_2) \equiv \nu(l_k(\Pi_1) \sqcup \cdots \sqcup l_k(\Pi_p) \leq l_k(\Sigma_1) \sqcap \cdots \sqcap l_k(\Sigma_s))$$

and by Corollary 4.26

$$(4.33) \quad \bigwedge_{i,j} \vdash \nu(l_k(\Pi_i) \leq l_k(\Sigma_j))$$

58
4.1. Checking Type Formulas

Let $E_{i,j} = \Pi_i \leq \Sigma_j$. If $\vdash \Pi_i = \bot$ or $\vdash \Sigma_j = \top$ then the thesis easily follows. So, suppose that the formula $E_{i,j}$ has either the form

$$\bigwedge_{i=1}^{q} \alpha_i \land F(\tau) \leq \bigcup_{i=1}^{r} \beta_i \cup G_1(\sigma_1) \cup \cdots \cup G_n(\sigma_n)$$

where $G_i \neq F$, or the form

$$\bigwedge_{i=1}^{q} \alpha_i \leq \bigcup_{i=1}^{r} \beta_i \cup G_1(\sigma_1) \cup \cdots \cup G_n(\sigma_n)$$

$$\bigwedge_{i=1}^{q} \alpha_i \land F(\tau) \leq \bigcup_{i=1}^{r} \beta_i \cup F(\tau') \cup G_1(\sigma_1) \cup \cdots \cup G_n(\sigma_n)$$

where, since the type in the right-hand side of (4.36) is discriminative, the types $F, G_i$ are distinct. We consider the case (4.36) (other are simpler and similar).

Now, we show that

$$\forall \nu. \vdash F(l_k(E_{i,j})) \implies \vdash E_{i,j}$$

Suppose that

$$\forall \nu. \vdash l_k(E_{i,j}))$$

By Lemma 4.16 we know that $E_{i,j}$ is equivalent to

$$\vdash \bigwedge_{i=1}^{q} \alpha_i \leq \bigcup_{i=1}^{r} \beta_i \text{ or } \vdash \tau \leq \tau'$$

Suppose, a contrario, that (4.39) does not hold. The negation of (4.39) is

$$\neg \vdash \bigwedge_{i=1}^{q} \alpha_i \leq \bigcup_{i=1}^{r} \beta_i \text{ and } \neg \vdash \tau \leq \tau'$$

We define the substitution $\nu_0$ which makes $l_k(E_{i,j})$ false. Since $\vdash \bigwedge_{i=1}^{q} \alpha_i \leq \bigcup_{i=1}^{r} \beta_i$ does not hold, $\vdash \bigwedge_{i=1}^{q} \alpha_i \leq \bigcup_{i=1}^{r} \beta_i$ does not hold, and $\{\alpha_1^k, \ldots, \alpha_q^k\} \cap \{\beta_1^k, \ldots, \beta_r^k\} = \emptyset$. We define

$$\nu_{\alpha,\beta}(\gamma) = \begin{cases} 1, & \text{if } \gamma \in \{\alpha_1^k, \ldots, \alpha_q^k\} \\ \bot, & \text{if } \gamma \in \{\beta_1^k, \ldots, \beta_r^k\} \end{cases}$$

Clearly we have that $h(\tau \leq \tau') \prec H$, so we can apply the induction hypothesis to obtain that

$$\vdash \tau \leq \tau' \text{ if } (\forall \nu) \nu(l_{k+1}(\tau \leq \tau'))$$

So, because $\vdash \tau \leq \tau'$ is not true there is a substitution $\nu'$ for which $\nu'(l_{k+1}(\tau \leq \tau'))$ is not provable.

By Remark 4.45 the substitutions $\nu'$ and $\nu_{\alpha,\beta}$ have different domains so we can join them to obtain $\nu = \nu_{\alpha,\beta} \circ \nu'$ for which $\vdash F(l_k(E_{i,j}))$ is false. But by (4.38) $\vdash l_k(E_{i,j})$ should be true for all substitutions, contradiction. So, we have established (4.37).
Chapter 4. Type Checking

Simple rules

\[
\begin{align*}
\top \cup \tau & \Rightarrow \top \\
\top \cap \tau & \Rightarrow \tau \\
\bot \cap \tau & \Rightarrow \bot \\
\bot \cup \tau & \Rightarrow \tau
\end{align*}
\]

(4.42) \[ F(\tau_1, \ldots, \tau_n) \cap F(\sigma_1, \ldots, \sigma_n) \Rightarrow F(\tau_1 \cap \sigma_1, \ldots, \tau_n \cap \sigma_n) \]

(4.43) \[ F(\tau_1, \ldots, \tau_n) \cap G(\sigma_1, \ldots, \sigma_n) \Rightarrow \bot \text{ if } F \neq G \]

**Distributive rule:** \( (\tau_1 \cup \cdots \cup \tau_n) \cap (\sigma_1 \cup \cdots \cup \sigma_m) \Rightarrow \bigcup_{i \leq n, j \leq m} \tau_i \cap \sigma_j \)

**Discriminative rule:**

\[ F(\tau_1, \ldots, \tau_n) \cup F(\sigma_1, \ldots, \sigma_n) \Rightarrow \bigcup F(\tau_1 \cup \sigma_1, \ldots, \tau_n \cup \sigma_n) \]

Figure 4.1: The ground types reduction rules

Now, by (4.37) the thesis of the lemma follows by the following sequence of implications:

\[
(\forall \nu) \vdash \nu(E_2) \Rightarrow (\forall \nu) \vdash \nu(l_k(\Pi_1 \cup \cdots \cup \Pi_p \leq \Sigma_1 \cap \cdots \cap \Sigma_s))
\]

\[
\Rightarrow (\forall \nu)(\forall i, j) \vdash \nu(l_k(\Pi_i \leq \Sigma_j))
\]

\[
\Rightarrow (\forall \nu)(\forall i, j) \vdash \nu(l_k(E_{i,j}))
\]

\[
\Rightarrow (\forall i, j)(\forall \nu) \vdash \nu(l_k(E_{i,j}) \quad (*)
\]

\[
\Rightarrow (\forall i, j) \vdash E_{i,j}
\]

\[
\Rightarrow E_1
\]

(\begin{itemize}
\item in the line (\*) we use (4.37).
\end{itemize})

4.1.3 Comparing Ground Types

In this section we analyze the complexity of checking validity of ground formulas. This is an important issue since, as we have seen in the previous section, checking discriminative type equations can be done by checking some ground type formulas. We will show that ground discriminative formulas can be tested in polynomial time, and that all discriminative type formulas can be tested in co-\text{NP}.

In the figure 4.1 we present some reduction rules which simplifies the ground types.

**Definition 4.49.** A ground type is simple if it does not contain operators \( \cap, \cup \).

**Definition 4.50.** By \( \Rightarrow_{\cup} \) we denote the rewriting system given by **simple rules**, **distributive rule** and **discriminative rule**.

We need the monotonicity of the operator of discriminative approximation \( A \). First we prove the auxiliary result:

60
Lemma 4.51. Let $A'$ be any function mapping types to types. If
\[ \vdash A'(\tau_1 \cap \tau_2) = A'((\tau_1) \cap A'(\tau_2)) \]
and
\[ \vdash \tau = \sigma \implies \vdash A'(\tau) = A'(\sigma) \]
then for each types $\sigma_1, \sigma_2$ we have
\[ \vdash \sigma_1 \leq \sigma_2 \implies \vdash A'((\sigma_1) \leq A'(\sigma_2)) \]

Proof. In Appendix.

Corollary 4.52. $A$ is monotonic i.e. for each types $\tau_1, \tau_2$ we have
\[ \vdash \tau_1 \leq \tau_2 \implies \vdash A(\tau_1) \leq A(\tau_2) \]

Proof. From definition of $A$ we can conclude that $A(\tau_1 \cap \tau_2) = A(\tau_1) \cap A(\tau_2)$. So, the thesis follows by Remark 4.39 and Lemma 4.51.

Now, we show that, informally speaking, when we increase a subterm of a type we also increase this type.

Lemma 4.53. Let $\tau$ be a type. Let $\tau'$ be obtained in the following way: the subterm $\sigma$ in $\tau$ is replaced by the type $\sigma'$ such that $\vdash \sigma \leq \sigma'$. Then we have
\[ \vdash \tau \leq \tau' \]

Proof. Let $\theta = \{\alpha/\sigma\}$, let $\theta' = \{\alpha/\sigma'\}$. Let $\tau_0$ be such that $\tau = \theta(\tau_0)$ and $\tau' = \theta'(\tau')$. Since $\theta \leq \theta'$ by Lemma 2.26 we obtain $\theta(\tau_0) \leq \theta'(\tau_0)$ and the thesis follows.

Lemma 4.54. For each type $\tau$ we have that $\tau \leq A(\tau)$

Proof. While computing $A(\tau)$ we may increase some subterms of the R-normal of $\tau$ and this, by Lemma 4.53 increases the type $\tau$.

Now, we show that the rewriting rule $\Rightarrow_{\cup}$ can be used to compute $A(\tau)$, i.e. $A(\tau)$ is the normal form with respect to $\Rightarrow_{\cup}$.

Remark 4.55. Let $\tau_0$ be a type. Let $\tau_i \Rightarrow_{\cup} \tau_{i+1}$. Then the sequence of $\tau_i$ is finite.

Lemma 4.56. Let $\tau_0$ be a type. Let $\tau_i \Rightarrow_{\cup} \tau_{i+1}$, for $i \in \{0, \ldots, k\}$ for some $k$. Then
for each $i$ we have
\[ \vdash \tau_i \leq A(\tau_0) \]

Proof. In appendix.

Lemma 4.57. Let $\tau_0$ be a type. Let $\tau_i \Rightarrow_{\cup} \tau_{i+1}$. Then for each $i$ we have $\tau_i \leq \tau_{i+1}$.

Proof. There is only one rule (discriminative rule) which changes the value of the type. This rule increases some subterm of $\tau$ which, by Lemma 4.53, increases the value of $\tau$.

Lemma 4.58. Let $\tau_0$ be a ground type. Let $\tau_i \Rightarrow_{\cup} \tau_{i+1}$. Let $\tau_{\omega}$ be such $\tau_k$ for which no further reductions are possible. Then we have that $\vdash \tau_{\omega} = A(\tau_{\omega})$. 
(F_{k_1}(\tau_{k_1}) \cup \cdots \cup F_{k_m}(\tau_{k_m})) \cap (F_{j_1}(\sigma_{j_1}) \cup \cdots \cup F_{j_n}(\sigma_{j_n})) \Rightarrow \_ \\
\Rightarrow_\_ F_{l_1}(\tau_{l_1} \cap \sigma_{l_1}) \cup \cdots \cup F_{l_p}(\tau_{l_p} \cap \sigma_{l_p})}

where \( \{k_1, \ldots, k_m\} \cap \{j_1, \ldots, j_n\} = \{l_1, \ldots, l_p\} \) and \( k_p \neq k_q \) and \( j_p \neq j_q \) for \( p \neq q \)

Figure 4.2: Distributive rule for discriminative types

Proof. \( \tau_\omega \) is in R-reduced (also in L-reduced) form, since all \( \cap \) have been eliminated. Since discriminative rule does not apply to \( \tau_\omega \), there are no subexpressions of the form \( F(\tau_1) \cup F(\tau_2) \) in \( \tau_\omega \). For such types \( A \) is idempotent, i.e. \( \vdash \tau_\omega = A(\tau_\omega) \) and the thesis follows.

Lemma 4.59. Let \( \tau_0 \) be a ground type. Assume that \( \tau_i \Rightarrow_\cup \tau_{i+1} \). Let \( \tau_\omega \) be such \( \tau_k \) for which no further reductions are possible. Then

\[
(4.48) \quad \vdash \tau_\omega = A(\tau_0)
\]

Proof. From Lemma 4.56 we know that

\[
(4.49) \quad \vdash \tau_\omega \leq A(\tau_0)
\]

By Lemma 4.57 we have that \( \tau_0 \leq \tau_\omega \). So, since \( A \) is monotonic by Lemma 4.58 \( \vdash A(\tau_0) \leq A(\tau_\omega) \) which by Lemma 4.58 gives \( \vdash A(\tau_0) \leq \tau_\omega \) Now, by (4.49) the thesis follows.

From Lemma 4.59 we conclude that the order of reductions is irrelevant. So, we can use \( \Rightarrow_\cup \) to compute for the type \( \tau \) a ground type \( \tau' \) without \( \cap \) such that \( \tau' = A(\tau) \). Moreover, we can define any strategy of choosing the rules.

In Algorithm 4.3 which computes the discriminative approximation of \( \tau \) without operators \( \cap \) we do not want to use the normal distributive rule, because it can cause the exponential growth of the type. Instead we use its variant proposed in Figure 4.2.

This rule is correct because when we apply the distributive rule to the left-hand side of it and then use the fact that \( F_{k_p}(\tau) \cap F_{j_v}(\sigma) = \bot \) if \( k_p \neq j_v \) we obtain the right-hand side of this rule.

Moreover, it does not increase the number of function symbols in a term to which it applies.

Now, let us consider Algorithm 4.3 which uses this rule.

Lemma 4.60. Let \( \tau \) be a ground type. Algorithm 4.3 finds in polynomial time a type \( \tau' \) without \( \cap \) such that

\[
\vdash \tau' = A(\tau)
\]

Proof. It is easy to see that \( \tau_i \Rightarrow_\cup \tau_{i+1} \). Let \( \tau_k \) be the output type of Algorithm 4.3. We will show that there is no \( \tau' \) such that \( \tau_k \Rightarrow_\cup \tau' \). Suppose a contrario that there is such \( \tau' \). It can be obtained only by a distributive rule (the full version). Other rules are excluded since no rules of Algorithm 4.3 applies to \( \tau_k \). So, there is in \( \tau_k \) a subterm of the form:

\[
(\tau^1 \cup \cdots \cup \tau^n) \cap (\sigma^1 \cup \cdots \cup \sigma^m)
\]

Each \( \tau^i, \sigma^j \) are different from \( \top \) and \( \bot \), since otherwise one of simple rules could be applied. Moreover, in both unions, the term constructors are distinct, since otherwise
the discriminative rule would be applicable. But to such subterm the new version of the discriminative rule applies. Contradiction.

So, since \( \tau_k = \tau_0 \) from Lemma 4.59 it follows that \( \tau_k \) is a type without \( \cap \) and \( \tau_k = A(\tau_0) \).

Now, we analyze the complexity of this algorithm. Let \( L \) be the length of the formula. The rule (4.42) decreases the number of function symbols in \( \tau \) so it can be applied \( O(L) \) times. Let \( K \) be the maximal arity of the function symbol in \( \tau \). Let \( S \) be the maximal number of items in the union. Every application of the rule in Figure 4.2 adds \( O(KS) \) symbols \( \cap \), so, during execution of algorithm 4.3 \( O(NKS) \) such symbols can be added. Similarly the simple rule \( F(\ldots) \cap F(\ldots) \) decreases the number of function symbols and introduces \( O(N) \) operators \( \cap \).

All other rules reduce the number of operators \( \cap, \cup \), so, they can be applied \( O(N + KNS + N^2) \) times, which is \( O(N^3) \).

Lemma 4.61. Let \( \tau_1, \tau_2 \) be types. Suppose that \( \tau_2 \) is discriminative. Then

\[
\vdash \tau_1 \leq \tau_2 \text{ iff } \vdash A(\tau_1) \leq A(\tau_2)
\]

Proof. Since \( A \) is monotonic we have: \( \vdash \tau_1 \leq \tau_2 \Rightarrow \vdash A(\tau_1) \leq A(\tau_2) \). Now, suppose that \( \vdash A(\tau_1) \leq A(\tau_2) \). The type \( \tau_2 \) is discriminative, so \( \vdash A(\tau_1) \leq \tau_2 \). On the other hand by Lemma 4.54 we have \( \vdash \tau_1 \leq A(\tau_1) \), and, by transitivity of \( \leq \) we get \( \vdash \tau_1 \leq \tau_2 \).

Now, we present the main results of this section.

Lemma 4.62. Let \( \tau_1, \tau_2 \) be constant discriminative types. The formula \( \tau_1 \leq \tau_2 \) can be checked in polynomial time.

Proof. Using Algorithm 4.3 we can in polynomial time find the types \( \tau'_1, \tau'_2 \) without \( \cap \) such that

\[
\vdash \tau'_1 = A(\tau_1) = \tau_1
\]

\[
\vdash \tau'_2 = A(\tau_2) = \tau_2
\]

Comparison of such types can be done by Algorithm 4.2, since these types are in adequate reduced forms (they do not have \( \cap \)). By Lemma 4.31 this algorithm works in quadratic time.
Chapter 4. Type Checking

Theorem 4.1. Let \( \tau_1, \tau_2 \) be types. The problem whether

\[
\vdash \tau_1 \leq \tau_2
\]

for a discriminative type \( \tau_2 \) is in co-NP.

Proof. We describe how to check in co-NP the validity of (4.51). By Lemma 4.48 the formula (4.51) is provable if for every (\( \top, \bot \))-substitution \( \nu \) we have

\[
\vdash \nu(l(\tau_1 \leq \tau_2))
\]

Let \( E = l(\tau_1 \leq \tau_2) \). We can compute \( E \) in linear time. We claim that we can nondeterministically guess (\( \bot, \top \))-substitution \( \nu \) for which \( \nu(E) \) is not provable and then check it in polynomial time. Let \( \nu \) be a substitution and let

\[
\nu(E) = (\tau'_1 \leq \tau'_2)
\]

Then \( \tau'_2 = \nu(l(\tau_2)) \). Since \( \tau_2 \) is discriminative \( \tau'_2 \) is also discriminative.

Because of \( A(\tau'_2) = \tau'_2 \) by Lemma 4.61 we have that \( \vdash \tau'_1 \leq \tau'_2 \) is equivalent to

\[
\vdash A(\tau'_1) \leq \tau'_2
\]

which according to Lemma 4.62 can be checked in polynomial time. \( \square \)

4.2 Auxiliary Functions

Definition 4.63. Type \( \tau \) is an assumption type if it contains only variables, \( \top \) and type constructors. So in assumption type there are no \( \bot \), operators \( \land \), \( \lor \) and atomic types. \( \square \)

In this section we define and analyze two functions used in type checking. First one is called \( BS \). For a type \( \tau \) and an assumption type \( \sigma \) it computes the least substitution \( \theta \) such that \( \vdash \tau \leq \theta(\sigma) \). It is used in a type checking, i.e. in a searching for the proofs, whenever in the rules \( (T_0), (K_2), \) and \( (P_2) \) a substitution occurs.

The second function is called \( BT \), it computes the best possible type of the given term in the given environment. For every term we can find its best provable type. In the worst case it is the type \( \top \) derived by the rule \( (T_1) \).

The best substitution does not always exist. For instance there is no substitution \( \theta \) such that \( \alpha \leq \theta(F(\beta)) \). To make the function \( BS \) total we use some technical trick. In the result of \( BS \) we will mark the places which are wrong. We make it with a new constant \( \top^* \).

We embed the equality system to a system with the constant \( \top^* \).

Definition 4.64. The relation \( \vdash^* \) is defined by axioms (Ax1), ..., (Ax10), and (Ax12), ..., (Ax16), and three additional axioms:

\[
\begin{align*}
\vdash^* \top^* \cap \tau & = \tau \cap \top^* = \tau \\
\vdash^* \top^* \cup \tau & = \tau \cup \top^* = \top^* \\
\vdash^* F(\top^*, \ldots, \top^*) & = \top^*
\end{align*}
\]

So, we add the new constant \( \top^* \) which is greater than all other types, and has the special property expressed by (4.57).
Remark 4.65. Notice that for types \( \tau_1, \tau_2 \) not containing \( \top^* \), if \( \vdash \tau_1 = \tau_2 \) then \( \vdash \tau_1 = \tau_2 \).

We define some operations on substitutions:

**Definition 4.66.** Let \( \theta_1, \theta_2 \) be substitutions. For any substitution \( \theta \) we define:

\[
\theta^\top(\alpha) = \begin{cases} 
\theta(\alpha), & \text{if } \alpha \in \text{domain}(\theta) \\
\bot, & \text{otherwise}
\end{cases}
\]

Then, for each \( \alpha \), we put

\[
(\theta_1 \sqcup \theta_2)(\alpha) = \theta_1^\top(\alpha) \sqcup \theta_2^\top(\alpha)
\]

and

\[
(\theta_1 \sqcap \theta_2)(\alpha) = \theta_1^\top(\alpha) \sqcap \theta_2^\top(\alpha)
\]

Moreover, we define \( \emptyset(\alpha) = \bot \). Note that for each \( \alpha \) \( \emptyset(\alpha) = \emptyset^\top(\alpha) \).

By axioms (Ax7), (Ax8), (Ax10) it follows that \( \sqcap, \sqcup \) are associative, commutative and distributive.

We write \( \vdash^* \theta_1 = \theta_2 \) as an abbreviation of \( (\forall \alpha) \vdash^* \theta_1(\alpha) = \theta_2(\alpha) \). Similarly \( \vdash^* \theta_1 \leq \theta_2 \) means \( (\forall \alpha) \vdash^* \theta_1(\alpha) \leq \theta_2(\alpha) \).

Now, we are ready to define function \( BS \).

### 4.2.1 Best substitutions. Function \( BS \).

In this subsection we present the definition of \( BS \). Moreover, we prove the main properties of \( BS \).

**Definition 4.67.** \( BS \) is defined inductively for any type \( \tau \), and any assumption type \( \sigma \) not containing \( \top \) as follows:

\[
BS(\bot, \sigma) = \emptyset \\
BS(\tau, \alpha) = \{\alpha/\tau\} \sqcup \emptyset \\
BS(\tau_1 \sqcup \tau_2, \sigma) = BS(\tau_1, \sigma) \sqcup BS(\tau_2, \sigma) \\
BS(\tau_1 \sqcap \tau_2, \sigma) = BS(\tau_1, \sigma) \sqcap BS(\tau_2, \sigma) \\
BS(F(\tau_1, \ldots, \tau_n), F(\sigma_1, \ldots, \sigma_n)) = \bigcup_{i=1}^{n} BS(\tau_i, \sigma_i)
\]

If more than one case is applicable, we use the first which is applicable. If none of the cases above is applicable we put:

\[
BS(\tau, \sigma) = \emptyset \sqcup \{\alpha/\top^* \mid \alpha \in \text{var}(\sigma)\}
\]

**Remark 4.68.** In definition of \( BS \) we do not allow \( \top \) in \( \sigma \). It is not a serious restriction since we can replace \( \top \) by a new variable and obtain the substitution which differ only in the value of \( \text{var}(\sigma) \). For instance:

\[
BS(F(\tau_1, \tau_2), F(\top, \sigma)) = BS(F(\tau_1, \tau_2), F(\text{var}(\sigma), \sigma))
\]

This restriction allow to define \( BS \) as a total function.
Remark 4.69. By an easy induction on the structure of a computation of $BS(\tau, \sigma)$ we can show that for a variable $\beta \not\in \text{var}(\sigma)$ we have $BS(\tau, \sigma)(\beta) = \bot$. This fact implies two facts below:

\[
(\text{BS}(\tau_1, \sigma_1) \cap \cdots \cap \text{BS}(\tau_n, \sigma_n))(\alpha) = \bigcap_{i=1}^{n} (\text{BS}(\tau_1, \sigma_1)(\alpha))
\]

\[
(\text{BS}(\tau_1, \sigma_1) \cup \cdots \cup \text{BS}(\tau_n, \sigma_n))(\alpha) = \bigcup_{i=1}^{n} (\text{BS}(\tau_1, \sigma_1)(\alpha))
\]

We have the following simple lemma:

Lemma 4.70. Let $\tau_1, \ldots, \tau_n$ be types, $\rho_1, \ldots, \rho_n, \sigma$ be assumption types. Then

\[
\bigcup_{i=1}^{n} \text{BS}(\tau_i, \rho_i)(\sigma) \leq (\text{BS}(\tau_1, \rho_1) \cup \cdots \cup \text{BS}(\tau_n, \rho_n))(\sigma)
\]

Proof. In Appendix.

Example 4.11. Let $\tau_1 = F(\alpha, G(\beta))$, $\tau_2 = F(G(\gamma), \delta)$, $\sigma = F(G(\alpha'), G(\beta'))$. Then there is no substitution $\theta$ s.t. $\vdash \tau_1 \leq \theta(\sigma)$, and there is no substitution $\theta$ s.t. $\vdash \tau_2 \leq \theta(\sigma)$. However,

\[
\vdash^* \theta_1 = BS(\tau_1, \sigma) = \{\alpha'/\top^*, \beta'/\beta\}
\]

\[
\vdash^* \theta_2 = BS(\tau_2, \sigma) = \{\alpha'/\alpha, \beta'/\top^*\}
\]

\[
\vdash^* \theta_3 = BS(\tau_1 \cap \tau_2, \sigma) = \{\alpha'/\gamma, \beta'/\beta\}
\]

Example 4.12.

$BS(F(\bot, G(\alpha)), F(\gamma, G(\beta))) = \{\beta/\alpha\}$

Now, we state the lemma which shows that for equivalent types $BS$ returns equivalent substitutions.

Lemma 4.71. Assume that $\vdash \tau_1 = \tau_2$. Then, for any assumption type $\sigma$ without $\top$, and for any variable $\alpha \in \text{var}(\sigma)$

\[
\vdash^* BS(\tau_1, \sigma)(\alpha) = BS(\tau_2, \sigma)(\alpha)
\]

Proof. In Appendix.

Now, we are going to prove that $BS$ has the desired properties.

Lemma 4.72. Let $\tau, \sigma$ be types. Then

\[
\vdash^* \tau \leq BS(\tau, \sigma)(\sigma)
\]

Proof. The proof proceeds by induction on the structure of the computation of $BS(\tau, \sigma)$. Consider the following cases:

$\bullet$ $\tau = \bot$. Obvious.
4.2. Auxiliary Functions

- $\sigma = \alpha$ (i.e. $\sigma$ is a type variable). Then
  $$\vdash^* \tau \leq (G/\tau) \cup 0(\alpha) = BS(\tau, \sigma)(\sigma)$$

- $\tau = \tau_1 \cup \tau_2$. By induction hypothesis we have that
  $$\tau_1 \leq BS(\tau_1, \sigma)\sigma$$
  $$\tau_2 \leq BS(\tau_2, \sigma)\sigma$$
  So, by Lemma 4.70 we obtain
  $$\vdash^* \tau_1 \cup \tau_2 \leq BS(\tau_1, \sigma)\sigma \cup BS(\tau_2, \sigma) \leq (BS(\tau_1, \sigma) \cup BS(\tau_2, \sigma))\sigma = BS(\tau_1 \cup \tau_2, \sigma)\sigma$$

- $\tau = \tau_1 \cap \tau_2$. The proof is similar as above.
- $\tau = F(\tau_1, \ldots, \tau_n), \sigma = F(\sigma_1, \ldots, \sigma_n)$. Let $\theta_i = BS(\tau_i, \sigma_i)$, and $\theta = BS(F(\tau_1, \ldots, \tau_n), F(\sigma_1, \ldots, \sigma_n)).$
  Then $\theta = \bigcup_{i=1}^n \theta_i$. By the induction hypothesis we have that for each $i \in \{1, \ldots, n\}$
  $$\tau_i \leq \theta_i(\sigma_i)$$
  By Remark 4.69 we conclude that
  \begin{equation}
  (4.59) \quad \theta_i \leq \theta.
  \end{equation}
  for each $i \in \{1, \ldots, n\}$. Therefore
  $$\tau = F(\tau_1, \ldots, \tau_n)$$
  $$\leq F(\theta_1(\sigma_1), \ldots, \theta_n(\sigma_n))$$
  by inductive hypothesis and Lemma 2.21
  $$\leq F(\theta(\sigma_1), \ldots, \theta(\sigma_n))$$
  by (4.59) and Lemma 2.21 again
  $$= \theta(F(\sigma_1, \ldots, \sigma_n))$$
  $$= BS(\tau, \sigma)(F(\sigma_1, \ldots, \sigma_n))$$

- None of the cases above. Then we have
  $$\vdash^* BS(\tau, \sigma)(\sigma) = (\{\alpha/\tau^* | \alpha \in \var(\sigma)\} \cup 0)(\sigma) = \top^*$$
  which, by $\vdash^* \tau \leq \top^*$ gives the thesis.

\[\Box\]

**Lemma 4.73.** Assume that $\theta = BS(\tau, \sigma)$. Then for each $\theta'$ such that $\tau \leq \theta'(\sigma)$ we have $\vdash^* \theta \leq \theta'$.

**Proof.** In Appendix. \[\Box\]

**Lemma 4.74.** If there is a substitution $\theta_{ok}$ without $\top^*$, such that $\vdash^* \tau \leq \theta_{ok}(\sigma)$ then $BS(\tau, \sigma)$ does not contain $\top^*$.

In other words $BS$ always find a proper substitution if it exists, and $\top^*$ cannot appear in the result.

**Proof.** Let $\theta = BS(\tau, \sigma)$. By Lemma 4.73 $\vdash^* \theta \leq \theta_{ok}$. Suppose a contrario that $\top^*$ belongs to the range of $\theta$. By the definition of $BS$ we have that in that case for some variable $\alpha$ we have that $\theta(\alpha) = \top^*$. It is, however, impossible since $\vdash^* \theta(\alpha) \leq \theta_{ok}(\alpha) < \top^*$.

\[\Box\]
Chapter 4. Type Checking

As we have seen \( T^* \) can be treated as a marker of errors. If there are no errors then the constant \( T^* \) does not occur in a substitution returned by \( BS \).

The domain of \( BS \) can be easily extended to cover assumptions containing \( \cap \). The additional case in the definition of \( BS \) is

\[
BS(\tau, \sigma_1 \cap \sigma_2) = BS(\tau, \sigma_1) \cup BS(\tau, \sigma_2).
\]

However, we will not consider such assumptions, so, we do not analyze this construction.

### 4.2.2 Best Types. Function \( BT \).

In this section we define the function \( BT \). The function \( BT \) has two arguments: an environment \( \Gamma \) and program term \( t \). It returns a type. We show that the function \( BT(\Gamma, t) \) computes the best provable type of the term \( t \) in the environment \( \Gamma \).

**Definition 4.75.** The function \( BT \) is defined as follows:

\[
BT(\Gamma, t) = \bot \text{ if } \text{false} \in \Gamma
\]

\[
BT(\Gamma, X) = \Gamma(X)
\]

\[
BT(\Gamma, f(t_1, \ldots, t_n)) = \left( \bigcap_{i=1}^{n} BS(BT(\Gamma, t_i), \sigma_i) \right)(\sigma)
\]

where \( f : \sigma_1 \ast \cdots \ast \sigma_n \rightarrow \sigma \), and for all \( i \in \{1, \ldots, n\}, BS(\tau_i, \sigma_i) \) has no \( T^* \) in its range. If more than one case is applicable we apply the first one. If none of above rules is applicable then:

\[
BT(\Gamma, t) = \top
\]

**Lemma 4.76.** If \( \tau = BT(\Gamma, t) \) then \( \Gamma \vdash t : \tau \).

**Proof.** The proof proceeds by induction on the structure of a computation of \( BS(\Gamma, t) \).

- \( \text{false} \in \Gamma \). Then \( BT(\Gamma, t) = \bot \), and by rule \((T_2)\) we have \( \Gamma \vdash t : \bot \).
- \( t \) is a variable. We have \( BT(\Gamma, X) = \Gamma(X) \), and \( \Gamma \vdash X : \Gamma(X) \).
- \( t = f(t_1, \ldots, t_n) \). Let \( f \) has a signature \( f : \sigma_1, \ldots, \sigma_n \rightarrow \sigma \). Let \( \tau_i = BT(\Gamma, t_i) \).

By the inductive hypothesis we have

\[
(4.60) \quad \Gamma \vdash t_i : \tau_i
\]

Let \( \theta_i = BS(\tau_i, \sigma_i) \) and let \( \theta = \bigcup_{i=1}^{n} \theta_i \). Since \( \theta \geq \theta_i \), Lemma 4.72 implies that \( \tau_i \leq \theta(\sigma_i) \) which by \((4.60)\), and the rule \((T_4)\) gives that

\[
(4.61) \quad \Gamma \vdash t_i : \theta(\sigma_i)
\]

and then, by rule \((T_6)\) it follows that

\[
(4.62) \quad \Gamma \vdash f(t_1, \ldots, t_n) : \theta(\sigma)
\]

Since \( \theta(\sigma) = BT(\Gamma, t) \) the thesis has been established.

- None of the cases above. Then \( BT(\Gamma, t) = \top \), and \( \Gamma \vdash t : \top \).

**Lemma 4.77.** If \( \Gamma \vdash t : \tau \) then \( \vdash BT(\Gamma, t) \leq \tau \).

**Proof.** In Appendix.
4.3 The Type Checking Algorithm

In this section we present an algorithm which checks whether a program P is well typed with respect to a directional type \( T \). It requires checking whether each clause of the program has a type which is assigned to its main head symbol.

To make the presentation more clear, we will use only atoms of the form \( q(t) \). The generalization to atoms \( q(t_1, \ldots, t_n) \) is straightforward.

The algorithm, which checks whether a clause has a type contains two parts. The first part computes the least guarantee which can be proved for this clause, while the second compares this guarantee with the one which we want to prove. Now, we give more details. We define a notion of a result type:

**Definition 4.78.** A result type of a clause \( C = (p(t) \leftarrow q_1(t_1), \ldots, q_n(t_n)) \), a vector of directional types

\[
((\tau_1 \rightarrow \sigma_1), \ldots, (\tau_n \rightarrow \sigma_n)),
\]

and assumptions \( \tau \) is the least type \( \sigma \) such that

\[ \vdash \text{ClauseHasType}(T, C, \tau \rightarrow \sigma) \]

where

\[ T = \{ q_1 : (\tau_1 \rightarrow \sigma_1), \ldots, q_n : (\tau_n \rightarrow \sigma_n) \} \]

and for the atom \( q_i \), while proving \( \text{InferFromAtoms} \), the type \( \tau_i \rightarrow \sigma_i \) is taken for the atom \( q_i \). If such a type does not exist the result type is a special constant \( \text{error} \).

In a similar way we define result environment.

**Definition 4.79.** A result environment of a clause

\[ C = (p(t) \leftarrow q_1(t_1), \ldots, q_n(t_n)), \]

a vector of directional types

\[
((\tau_1 \rightarrow \sigma_1), \ldots, (\tau_n \rightarrow \sigma_n)),
\]

and assumptions \( \tau \) is the least environment \( \Gamma \) such that

\[ \vdash \text{InferFromAtoms}(T, \Gamma_0, \langle q_1(t_1), \ldots, q_n(t_n) \rangle, \Gamma) \]

where \( \emptyset, t : \tau \Rightarrow \Gamma_0 \) and

\[ T = \{ q_1 : (\tau_1 \rightarrow \sigma_1), \ldots, q_n : (\tau_n \rightarrow \sigma_n) \} \]

and for the atom \( q_i \), while proving \( \text{InferFromAtoms} \), the type \( \tau_i \rightarrow \sigma_i \) is taken for the atom \( q_i \). If such an environment does not exists the result environment is a special constant \( \text{error} \).

Algorithm 4.4 (function \( \text{ResultType} \)) computes the result type. To proof it we need the following lemmas.

**Lemma 4.80.** Let \( \Gamma_1, \Gamma_2 \) be environments. Let \( t \) be a program term. Then if \( \Gamma_1 \leq \Gamma_2 \) then

\[ \mathcal{BT}(\Gamma_1, t) \leq \mathcal{BT}(\Gamma_2, t) \]

69
function ResultType(C, τ, V)
    C — a clause
    τ — assumption
    V — vector of directional types
let C = p(t) ← q₁(t₁), ..., qₙ(tₙ)
let V = ((τ₁ → σ₁), ..., (τₙ → σₙ))
let Γ₀ be such that ∅, t : τ ⇒ Γ₀
for i = 1 to n do
    let ρᵢ be BT(Γᵢ₋₁, tᵢ)
    let θᵢ be BS(ρᵢ, τᵢ)
    if θᵢ contains ⊤ then return error
    let Γᵢ be such that Γᵢ₋₁, tᵢ : θᵢ(σᵢ) ⇒ Γᵢ
return BT(Γₙ, t)

Algorithm 4.4: Computing of the result type

Proof. Assume that

\( (4.63) \quad Γ₁ ≤ Γ₂ \)

By Lemma 4.76 we have \( Γ₂ ⊢ t : BT(Γ₂, t) \). So, by (4.63) and by the definition of the relation \( ≤ \) on environments we have \( Γ₁ ⊢ BT(Γ₂, t) \). This, by Lemma 4.77, gives

\( BT(Γ₁, t) ≤ BT(Γ₂, t) \)

Now, we show the correctness of Algorithm 4.4.

Lemma 4.81. Let \( C = (p(t) ← q₁(t₁), ..., qₙ(tₙ)) \) be a clause, \( V = ((τ₁ → σ₁), ..., (τₙ → σₙ)) \) be a vector of directional types, and let \( τ \) be a type. The result type for \( C, V, τ \) is equal to the value returned by Algorithm 4.4.

Proof. Suppose that

\( (4.64) \quad \vdash \text{ClauseHasType}(T, C, τ → σ) \)

for some \( σ \). We will use the fact that, as we have seen in Section 4.2 the functions \( BS, BT \) return the best possible values. In Algorithm 4.4 we have a sequence of environments \( Γ₀, Γ₁, ..., Γₙ \). In the proof of (4.64) in the System B we also have a sequence of environments. Let us denote them by \( Γ'_₀, ..., Γ'_ₙ \). We show by induction on \( i \) that for each \( i \in \{0, ..., n\} \) we have

\( (4.65) \quad Γᵢ ≤ Γ'_ᵢ \)

We have \( Γ₀ = Γ'_₀ \). Suppose that (4.65) holds for \( i - 1 \). In the proof of (4.64) use the rule \((P₂)\) so there a proof of \( Γ'_ᵢ₋₁ ⊢ tᵢ : θᵢ(τᵢ) \). Because of the lemma 4.77 and 4.80 we have

\( (4.66) \quad θᵢ(τᵢ) ≥ BT(Γ'_ᵢ₋₁, tᵢ) ≥ BT(Γᵢ₋₁, tᵢ) = ρᵢ \)

By (4.66) and Lemma 4.73 we have that

\( (4.67) \quad θᵢ ≥ θᵢ = BS(ρᵢ, τᵢ) \)
4.3. The Type Checking Algorithm

function CorrectClause(T, C, τ → σ)
    T — type of the program
    C — clause to be tested
    τ → σ — type to be tested
guess the vector V of the types for q_i (⋆)
let ρ be the ResultType(C, V, τ)
if ρ ≤ σ then return ‘Yes’
else return ‘No’

Algorithm 4.5: Checking whether clause C has a given type

Since

\[ \Gamma_{i-1}, t : \theta(\sigma) \Rightarrow \Gamma_i \]
\[ \Gamma'_{i-1}, t : \theta'(\sigma) \Rightarrow \Gamma_i \]

by Lemma 2.58, (4.65) and (4.67) we can conclude that \( \Gamma'_{i+1} \geq \Gamma_{i+1} \).

So, we have that \( \Gamma''_n \geq \Gamma_n \). Moreover \( \Gamma'_n \vdash t : \sigma \). So, by Lemmas 4.77 and 4.80 we have

\[ \sigma \geq BT(\Gamma'_n, t) \geq BT(\Gamma_n, t) \]

We have established that the returned value is less than any provable guarantee of C. It remains to show that if Algorithm 4.4 returns a value, then one can prove that this value is a guarantee for this clause.

Now, suppose that the Algorithm 4.4 returns a type.

We show how to construct the proof of (4.64) using the correctness of the functions \( BT, BS \). We show that for every atom \( q_i(t_i) \) we can apply the rule \( (P_2) \). We have that

\[ \theta_i = BS(\rho_i, \tau^i) \]

By Lemma 4.72 we have

\[ \rho_i \leq \theta_i(\tau_i) \]

From that, by Lemma 4.76, and rule \( (T_4) \) we have

\[ \Gamma_{i-1} \vdash t_i : \theta_i(\tau_i) \]

and, while proving InferFromAtoms, the atom \( q_i \) can be analyzed. So, there is a proof of (4.64), for \( \sigma \) equal to \( BT(\Gamma_n, t) \).

Algorithm 4.5 checks whether a clause has given type. It uses the function ResultType. It is a nondeterministic algorithm which guesses the types for the predicates in the body of the clause to be checked.

Lemma 4.82. Algorithm 4.5 (CorrectClause) answers ‘Yes’ if and only if

\[ \vdash \text{ClauseHasType}(T, C, \tau \rightarrow \sigma) \]

Proof. This lemma is a simple corollary to Lemma 4.81.

71
function ResultType2($C, \tau, T$)
    $C$ — a clause
    $\tau$ — assumption
    $T$ — program type
    let $C = p(t) \leftarrow q_1(t_1), \ldots, q_n(t_n)$
    let $\Gamma_0$ be such that $\emptyset, t : \tau \Rightarrow \Gamma_0$
    for $i = 0$ to $n-1$ do
        let $\rho_i$ be $BT(\Gamma_i, t_i)$
        choose type $\tau^i \rightarrow \sigma^i$ for $q_i$ from $T$
        let $\theta_i$ be $BS(\rho_i, \tau_i)$
        if $\theta_i$ contains $T^*$ then return error
        let $\Gamma_{i+1}$ be such that $\Gamma_i, t : \theta(\sigma') \Rightarrow \Gamma_{i+1}$
    return $BT(\Gamma_n, t)$

Algorithm 4.6: Computing the result type for $\cap$ closed $T$.

Now we shortly discuss the line marked with the star ($\star$). In general this is a nondeterministic guess, so deterministic program should test all the possibilities. It gives us single exponential mark-up since the possibilities (i.e. types) are given by an user.

However, there is a very useful family of types for which it is always possible to find the most appropriate type of the predicate.

Definition 4.83. A directional type $T$ of a program $P$ is $\cap$-closed if for each predicate $p$ in the program $P$, for every pair of directional types $p : \tau_1 \rightarrow \sigma_1$ and $p : \tau_2 \rightarrow \sigma_2$ in $T$ there is a type $\tau_1 \cap \tau_2 \rightarrow \sigma_1 \cap \sigma_2$ in $T$. □

If a directional type is $\cap$-closed, then there is always a perfect choice: if there are $n$ incomparable types which can be applied, namely

$$(\tau_1 \rightarrow \sigma_1), \ldots, (\tau_n \rightarrow \sigma_n)$$

then we can choose the type $\bigcap_{i=1}^{n} \tau_i \rightarrow \bigcap_{i=1}^{n} \sigma_i$. We recall that according to Lemma 2.53 (Intersection Lemma) we can always assume that the family of types is $\cap$-closed.

The algorithm 4.6 is a variant of the function ResultType which works for $\cap$-closed types. It does not require the vector of directional type (i.e. the order in which they are used), it is enough to know the set of all types — as we have shown above it suffices to choose the type.

To analyze Algorithm 4.6 we need some lemmas.

Lemma 4.84. If $\rho \leq \rho'$ then $\vdash BS(\rho, \tau) \leq BS(\rho', \tau)$

Proof. Let

$$\theta = BS(\rho, \tau)$$
$$\theta' = BS(\rho', \tau)$$

By Lemma 4.72 we have

$$\rho \leq \rho' \leq \theta'(\tau)$$

Now, by Lemma 4.73 we can conclude that if $\rho \leq \theta'(\tau)$ then $\theta \leq \theta'$.
Lemma 4.85. Let \( \rho \) be a type. Let \( \tau_1, \tau_2 \) be assumption types. Assume that \( \tau_1 \) has distinct variables, and \( \tau_2 \) has also distinct variables. Moreover, assume that \( \tau_1 \leq \tau_2 \).
Let
\[
\theta_1 = BS(\rho, \tau_1) \\
\theta_2 = BS(\rho, \tau_2)
\]
Suppose that \( \theta_2 \) does not contain \( \top^* \). Then
\[
\vdash \theta_1(\alpha) = \theta_2(\alpha), \text{ for each } \alpha \in \text{var}(\tau_2)
\]
Proof. Sketch. One can show that \( \tau_2 \) can be obtained from \( \tau_1 \) by replacing some subtrees of \( \tau_1 \) by \( \top \). So, for each variable \( \alpha \in \text{var}(\tau_2) \), paths from the root to this variable are the same in \( \tau_1 \) and \( \tau_2 \). The value of \( BS \) for a variable \( \alpha \) depends on the type \( \rho \), and the path from the root to variable \( \alpha \), which implies the thesis.

Lemma 4.86. Let \( C = (p(t) \leftarrow q_1(t_1), \ldots, q_n(t_n)) \) be a clause. Let
\[
V = ((\tau_1 \rightarrow \sigma_1), \ldots, (\tau_n \rightarrow \sigma_n)), \\
V' = ((\tau'_1 \rightarrow \sigma'_1), \ldots, (\tau'_n \rightarrow \sigma'_n))
\]
are the vectors of directional types, and let \( \tau \) be a type. Finally let
\[
\tau_1, \ldots, \tau_n, \tau_1', \ldots, \tau_n'
\]
are assumption types with distinct variables in leaves, type constructors and \( \top \). Assume that for \( i \in \{1, \ldots, n\} \)
\[
\tau_i \leq \tau'_i \text{ and } \sigma_i \leq \sigma'_i
\]
Moreover assume that ResultType\((C, V, \tau)\) and ResultType\((C, V', \tau)\) are different than error. If \( \tau_i \leq \tau'_i \) for each \( i \in \{1, \ldots, n\} \) then
\[
\vdash \text{ResultType}(C, V, \tau) \leq \text{ResultType}(C, V', \tau)
\]
Proof. Let \( (\Gamma_i) \) denote the sequence of environments obtained by computing the ResultType for \( V \), let \( (\Gamma'_i) \) denote the sequence of environments obtained by computing the ResultType for \( V' \).
We show that \( \Gamma_i \leq \Gamma'_i \) for each \( i \in \{1, \ldots, n\} \). The proof proceeds by induction on \( i \). The basic case is trivial since \( \Gamma_0 = \Gamma'_0 \). Now, suppose that \( \Gamma_i \leq \Gamma'_i \)
By Lemma 4.80 we have that
\[
(4.68) \quad \rho_i \leq \rho'_i
\]
Let
\[
\theta_i = BS(\rho_i, \tau_i) \\
\theta'_i = BS(\rho'_i, \tau'_i)
\]
Because both ResultType\((C, V, \tau)\) and ResultType\((C, V', \tau)\) are different than error, \( \theta'_0, \theta'_i \) do not contain \( \top^* \). From that, by Lemma 4.85 we obtain
\[
(4.69) \quad \theta'_i(\alpha) = \theta'_i(\alpha), \text{ for each } \alpha \in \text{var}(\tau'_i)
\]
Moreover, by (4.68) and Lemma 4.84 we have

\[(4.70) \quad \theta_i(\tau_i) \leq \theta_i'(\tau_i)\]

By this and (4.69), since \(\text{var}(\tau_i) \supseteq \text{var}(\tau'_i)\) we obtain

\[(4.71) \quad \theta_i(\alpha) = \theta_i'(\alpha), \text{ for each } \alpha \in \text{var}(\tau'_i)\]

Since \(\sigma_i \leq \sigma'_i\) we have \(\theta_i(\sigma_i) \leq \theta_i'(\sigma'_i)\) and by (4.71) we obtain \(\theta_i(\sigma_i) \leq \theta_i'(\sigma'_i)\).

Now, using the Lemma 2.58 we obtain

\[(4.72) \quad \Gamma_{i+1} \leq \Gamma'_{i+1}\]

So, \(\Gamma_n \leq \Gamma'_n\) and by Lemma 4.80 we obtain the thesis. \(\square\)

Suppose that all types from a \(\cap\)-closed family have assumptions which are assumption types. Consider two vectors of types:

\[
V = ((\tau_1 \rightarrow \sigma_1), \ldots, (\tau_n \rightarrow \sigma_n)),
\]

\[
W = ((\tau'_1 \rightarrow \sigma'_1), \ldots, (\tau'_n \rightarrow \sigma'_n))
\]

Let

\[
Z = ((\tau_1 \cap \tau'_1 \rightarrow \sigma_1 \cap \sigma'_1), \ldots, (\tau_n \cap \tau'_n \rightarrow \sigma_n \cap \sigma'_n))
\]

Then by Lemma 4.86 applied to vectors \(V\) and \(Z\) we get that for \(Z\) we obtain smaller value of ResultType.

So, if we take the product of type which can be applied, we obtain better ResultType. We conclude these considerations with a following corollary:

**Lemma 4.87.** Algorithm 4.6 is correct, i.e. it computes the result type.

We state a lemma which will be useful in proving properties of the type reconstruction guarantees:

**Lemma 4.88.** Let \(C\) be a clause, let \(\tau_0\) be a type, let \(T, T'\) be directional types, s.t. for each \(q : (\tau \rightarrow \sigma) \in T'\) there is \(q : (\tau \rightarrow \sigma) \in T\) such that \(\sigma \leq \sigma'\). Suppose, moreover, that ResultType\(2(C, \tau_0, T)\) do not return error. Then we have that ResultType\(2(C, \tau_0, T)\) is different from error and

\[
\text{ResultType}_{2}(C, \tau_0, T) \leq \text{ResultType}_{2}(C, \tau_0, T')
\]

**Proof.** Proof is similar to proofs of Lemma 4.81 and Lemma 4.86 and will not be presented. \(\square\)

It remains to show how to use the procedure which checks the type of a clause to check the type of a program. Algorithm 4.7 for a program \(P\) and a directional type \(T\) checks whether a program \(P\) is well typed with respect to \(T\). It is very simple and do not need to be formally analyzed.

The next section is devoted to the complexity of type checking. Let us notice the simple fact. The type checker which uses rules \((K_1)\text{-}(K_9)\) can not work in PSPACE. In fact, while checking a prove of InferFromAtoms large types can be obtained. Below we give example of a program for which type checking produces types of the exponential size.

**Example 4.13.** Assume that \(f : \alpha \ast \beta \rightarrow F(\alpha, \beta)\). Consider the following program:
4.4. Complexity of the Type Checking Algorithm

\textbf{Algorithm 4.7: The type checking algorithm}

\begin{verbatim}
function CorrectProgram(P, T)
P — program
T — a directional type of a program
for each clause C = p(t) ← B ∈ P do
  for each (p : τ → σ) ∈ T do
    if not CorrectClause(T, P, τ → σ) then return 'No'
return 'Yes'
\end{verbatim}

\textit{q}(X, f(X, X)).

\textit{p}(X, Y) :- \textit{q}(X, V_1), \textit{q}(V_2, V_3), \ldots, \textit{q}(V(n-1), V_n), \%(*)
\textit{q}(Z_n, V_n), q(Z(n-1), Z_n), \ldots, \textit{q}(Z_1, Z_2), \textit{q}(Y, Z_1).

and the type:

\[ p : (\alpha, \beta) \rightarrow (\alpha \cap \beta, \beta) \]

Let

\[ T_0 = \alpha \]
\[ T_{i+1} = F(T_i, T_i) \]

When the proof of InferFromAtoms comes to the point \((*)\) then \(T_n\) is a type of \(\forall n\). Clearly \(T_n\) is the full binary tree of the height \(n\) and size \(2^n\). \(\square\)

4.4 Complexity of the Type Checking Algorithm

In this section we show that for discriminative types type checking can be done in \(\text{EXPTIME}\). We shortly discuss the reasons for such complexity and say why, in our opinion, it does not abort the practical usage of our system.

First we bound the size of types which are obtained during type checking. It is convenient to define a new measure of sizes of types.

\textbf{Definition 4.89.} Let \(s\) be a function from types to integers defined as follows:

\[ s(\alpha) = 1 \]
\[ s(a) = 1 \]
\[ s(\top) = s(\bot) = 1 \]
\[ s(\tau_1 \cup \tau_2) = s(\tau_1) + s(\tau_2) \]
\[ s(\tau_1 \cap \tau_2) = s(\tau_1) + s(\tau_2) \]

\[ s(F(\tau_1, \ldots, \tau_n)) = 1 + \sum_{i=1}^{n} s(\tau_i) \]

where \(a\) is atomic type. \(\square\)

The definition of \(s(\tau)\) is similar to the definition of \(|\tau|\). However, while computing \(s(\tau)\) we do not consider the space used by \(\cap, \cup\) operators. Since these operators are interlaced with other symbols we have:

\textbf{Corollary 4.90.} For each type \(\tau\), \(|\tau| \leq 2s(\tau)\)
We can extend the definition of $s$ in order to make it work with substitutions and environments.

**Definition 4.91.** Function $s$ on type substitutions is defined by:

$$s(\theta) = \sum_{\alpha \in \text{dom}(\theta)} s(\theta(\alpha))$$

Function $s$ on environments is defined by:

$$s(\Gamma) = \begin{cases} 
0, & \text{if } \Gamma = \{false\} \\
\sum_{X: \tau \in \Gamma} s(\Gamma(X)) & \text{otherwise}
\end{cases}$$

The next lemmas describe some simple properties of $s$.

**Lemma 4.92.** Let $\Gamma_1, \Gamma_2$ be environments. We have

$$s(\Gamma_1 \sqcup \Gamma_2) \leq s(\Gamma_1) + s(\Gamma_2)$$

*Proof. In appendix.*

**Lemma 4.93.** Let $\Gamma_1, \Gamma_2$ be environments. We have

$$s(\Gamma_1 \cap \Gamma_2) \leq s(\Gamma_1) + s(\Gamma_2)$$

*Proof. In appendix.*

From Lemmas 4.92, 4.93 we get immediately two corollaries:

**Corollary 4.94.** $s(\Gamma_1 \sqcup \cdots \sqcup \Gamma_n) \leq \sum_{i=1}^n s(\Gamma_i)$

**Corollary 4.95.** $s(\Gamma_1 \cap \cdots \cap \Gamma_n) \leq \sum_{i=1}^n s(\Gamma_i)$

Corresponding properties of substitutions can be proved in a similar way. So, we state the next lemma without any proof.

**Lemma 4.96.** Let $\theta_1, \ldots, \theta_n$ be substitutions. Then

$$s(\theta_1 \sqcup \cdots \sqcup \theta_n) \leq \sum_{i=1}^n s(\theta_i)$$

and

$$s(\theta_1 \cap \cdots \cap \theta_n) \leq \sum_{i=1}^n s(\theta_i)$$

The lemma below says that the relation $\Rightarrow$ in some sense does not depend on the initial environment (its first argument). This fact is very useful in estimating the size of the final environment (third argument).

**Lemma 4.97.** Let $t$ be a term, $\tau$ be a type and $\Gamma$ be an environment. If $\Gamma, t : \tau \Rightarrow \Gamma'$ then $\emptyset, t : \tau \Rightarrow \Gamma' \emptyset$, and $\Gamma' \emptyset = \Gamma \cap \Gamma' \emptyset$

*Proof. In Appendix.*

**Lemma 4.98.** If $\emptyset, t : \tau \Rightarrow \Gamma'$ then $s(\Gamma') \leq s(\tau) \cdot |t|$. 

76
4.4. Complexity of the Type Checking Algorithm

Proof. In appendix. □

**Lemma 4.99.** If $\Gamma, t : \tau \Rightarrow \Gamma'$ then $s(\Gamma') \leq s(\Gamma) + s(\tau) \cdot |t|$

**Proof.** By Lemma 4.97 we conclude that

\[
\emptyset, t : \tau \Rightarrow \Gamma^0 \text{ and } \Gamma' = \Gamma \cap \Gamma^0
\]

By Lemma 4.98 and (4.73) we have that $s(\Gamma^0) \leq s(\tau)|t|$. This together with the second part of (4.73) and Lemma 4.93 implies $s(\Gamma') \leq s(\Gamma) + s(\Gamma^0) \leq s(\Gamma) + s(\tau)|t|$ □

Now, we state lemmas which describes the relation between sizes of results and of arguments for functions $BS$ and $BT$.

**Lemma 4.100.** Suppose that $\top^*$ do not appear in the range of $BS(\tau, \sigma)$. Then

\[
s(\text{BS}(\tau, \sigma)) \leq s(\tau)
\]

**Proof.** In Appendix. □

**Lemma 4.101.** Assume that $\tau = BT(\Gamma, t)$. Then

\[
s(\tau) \leq \max(1, s(\Gamma)) \cdot |t|
\]

**Proof.** In Appendix. □

The next lemma collects results of three previous lemmas.

**Lemma 4.102.** Assume that $\Gamma$ is an environment the entry point of the loop in ResultType (Algorithm 4.4). Assume that $\Gamma'$ is the environment at the exit point of the loop in ResultType. Let $q(\tau)$ be the current predicate and $\tau \rightarrow \sigma$ be its type. Then

\[
s(\Gamma') \leq \max(1, s(\Gamma)) \cdot (1 + |t|^2 \cdot s(\sigma))
\]

**Proof.** Let

\[
\rho = BT(\Gamma, t)
\]

By Lemma 4.101 we get

\[
s(\rho) \leq |t| \cdot \max(1, s(\Gamma))
\]

Let $\theta = BS(\rho, \tau)$. By Lemma 4.100 and (4.78) we get

\[
s(\theta) \leq s(\rho) \leq |t| \cdot \max(1, s(\Gamma))
\]

Since $s(\theta(\sigma)) \leq \max(1, s(\theta)) \cdot s(\sigma)$ we have

\[
s(\theta(\sigma)) \leq \max(1, |t| \cdot \max(1, s(\Gamma))) \cdot s(\sigma)
\]

\[
= |t| \cdot \max(1, s(\Gamma)) \cdot s(\sigma)
\]

Since $\Gamma, t : \theta(\sigma) \Rightarrow \Gamma'$ from lemma 4.99 we have

\[
s(\Gamma') \leq s(\Gamma) + |t| \cdot (1, s(\Gamma)) \cdot |t| \cdot s(\sigma)
\]

\[
\leq \max(1, s(\Gamma))(1 + |t|^2 \cdot s(\sigma))
\]

which finishes the proof. □
Lemma 4.103. The size of the environment $\Gamma$ obtained during run of ResultType is $O(N(3+\varepsilon)N)$, where $\varepsilon > 0$.

Proof. By Corollary 4.90 it does not matter whether we use $s(\cdot)$ or $|\cdot|$. So, by Lemma 4.102, since $\sigma$ and $|t|$ are $O(N)$, we obtain the thesis. $\square$

The similar considerations lead to obtaining the bounds for the height of types.

We use the function $h(\tau)$ (see definition 4.47). We can adapt this definition to manage substitutions and environments.

Definition 4.104. The function $h$ for type substitution

\[(4.80) \quad h(\theta) = \max\{h(\theta(\alpha)) | \alpha \in \text{dom}(\theta)\} \]

Definition 4.105. The function $h$ for environment

\[(4.81) \quad h(\Gamma) = \max\{h(\tau) | (X : \tau) \in \Gamma\} \]

Now, we analyze how behave the height of the terms in environments from Algorithm 4.4. So we have to examine changes of the height done by the relations $\Rightarrow, BS, BT$.

Lemma 4.106. If $\emptyset, t : \tau \Rightarrow \Gamma'$ then

\[(4.82) \quad h(\Gamma') \leq h(\tau) \]

Proof. In Appendix. $\square$

Lemma 4.107. If $\Gamma, t : \tau \Rightarrow \Gamma'$ then $h(\Gamma') \leq h(\Gamma) + h(\tau)$

Proof. By Lemma 4.97 we conclude that

\[(4.83) \quad \emptyset, t : \tau \Rightarrow \Gamma^0 \text{ and } \Gamma' = \Gamma \cap \Gamma^0 \]

By Lemma 4.106 and (4.83) we have that $h(\Gamma^0) \leq h(\tau)$. From that, from the second part of (4.83) we have $h(\Gamma') \leq \max(h(\Gamma), h(\Gamma^0)) \leq \max(h(\Gamma), h(\tau))$ $\square$

Now, we state the lemma which describes the height of BS.

Lemma 4.108. Suppose that there are no $\top^*$ in the range of $BS(\tau, \sigma)$. Then $h(BS(\tau, \sigma)) \leq h(\tau)$

Proof. In Appendix. $\square$

Lemma 4.109. Suppose that $\tau = BT(\Gamma, t)$. Then $h(\tau) \leq h(\Gamma) + h(t)$.

Proof. In Appendix. $\square$

Lemma 4.110. Assume that $\Gamma$ is an environment at the entry point of the loop in ResultType (Algorithm 4.4). Assume that $\Gamma'$ is the environment at the exit point of the loop in ResultType. Let $q(t)$ be the current predicate and $\tau \rightarrow \sigma$ be its type. Then

\[(4.84) \quad h(\Gamma') \leq h(\Gamma) + h(t) + h(\sigma) \]

78
4.4. Complexity of the Type Checking Algorithm

Proof. Let

\[ \rho = \text{BT}(\Gamma, t) \]

By Lemma 4.109 we get

\[ h(\rho) \leq h(\Gamma) + h(t) \]

Let \( \theta = \text{BS}(\rho, \tau) \). By Lemma 4.108 and (4.86) we get

\[ h(\theta) \leq h(\rho) \leq h(\Gamma) + h(t) \]

Because \( h(\theta(\sigma)) \leq h(\theta) + h(\sigma) \) we have

\[
h(\theta(\sigma)) \leq h(\theta) + h(\sigma) \\
\leq h(\Gamma) + h(t) + h(\sigma)
\]

\[ \square \]

Lemma 4.111. The height of the environment \( \Gamma \) obtained during type checking is \( O(N^2) \) where \( N \) is the size of the type checking task.

Proof. When typing the clause the initial environment \( \Gamma_0 \) will have height \( O(N) \) since \( \emptyset, t : \tau \Rightarrow \Gamma_0 \) and height of the \( \tau \) is \( O(N) \).

When we infer from atoms then every atom \( q(t) \) can, according to Lemma 4.110, make the environment deeper by \( h(\sigma) + h(t) \), both of them have size \( O(N) \), since both of them are parts of the input data. Because only \( O(N) \) such changes are possible and

\[ O(N) + O(N) * O(N) = O(N^2) \]

we obtain the thesis of the lemma. \[ \square \]

Theorem 4.2. For discriminative types Algorithm 4.5 works in \( \text{EXPTIME} \).

Proof. By Lemma 4.103 the result type \( (\rho) \) has a single exponential size. Since all operations \( (\text{BS}, \text{BT}, \Rightarrow) \) work in linear time, \( \rho \) can be obtained in a single exponential time. Checking whether result type is less then the desired guarantee \( (\sigma) \) can be done as follows:

1. We compute \( l_0(\rho \leq \sigma) \). It can be done in time linear in the size of \( \rho \).
2. We iterate over all \( (\top, \bot)\)-substitutions of \( \text{var}(l_0(\rho \leq \sigma)) \). By Lemma 4.111 there are \( O(N \cdot N^2) \) variables in \( l_0(\rho \leq \sigma) \). So the number of substitutions is single exponential.
3. For ground discriminative types relation \( \leq \) can be checked, according to Lemma 4.62 in polynomial time.
4. If for all substitutions the answer is 'Yes' then the algorithm answers 'Yes', otherwise it answers 'No'.

\[ \square \]

We shortly analyze this result. There are two sources of the large number of steps: the number of type variables, and the sizes of the types obtained during execution of ResultType. We have presented in section 4.3 an Example 4.13 which has showed an exponential blow-up of environment. In that example a polymorphic type of arity 2 was used. Now, we present example which shows that even after further restrictions on types such an exponential blow-up may occur.

Example 4.14. Let \( f : \alpha \rightarrow F(\alpha) \) and \( g : \alpha \rightarrow g(\alpha) \). Consider the program
and the type \( p : (\alpha, T) \rightarrow (\alpha, F(\alpha) \cup G(\alpha)) \). When the algorithm ResultType computes the sequence of environments, a current environment will have exponential size at point \((*)\). Even if we restrict number of type constructors to 1, the similar situations can happen when we use, for instance, a predicate of the type

\[
(\alpha, \beta, T, T) \rightarrow (\alpha, \beta, F(\alpha) \cup F(\beta)), F(F(\alpha)) \cup F(\beta))
\]

\[
\square
\]

However, we believe that in programs that appear in practice such situations are unusual. For instance, in programs with unary type constructors (such as list, tree), if assumptions are simple, then the number of type variables is equal to the arity of a predicate. It is a small integer, not connected with a size of a whole program. The shapes of types obtained during type checking of real programs are also fairly small. It is because every element of an environment describes a type of a program variable and programmers rather do not use complicated types: it is hard to imagine an useful program in which, for instance, the type tree(tree(list(tree(int)))) is used.

### 4.5 Hardness of Type Checking

**Definition 4.112.** Type checking problem (TCP) is the problem of checking whether or not a program \( P \) is well-typed with respect to a directional type \( T \).

In this section we show the hardness of the type checking problem.

**Definition 4.113.** Let \( \{\alpha_i\}_{i \in I} \) and \( \{\alpha'_i\}_{i \in I} \) be two disjoint sets of type variables indexed with the same finite sets \( I \). The top-bottom substitution \( \nu \) over these sets is **good**, if for all \( i \in I \) we have that

\[
\nu(\alpha_i) \neq \nu(\alpha'_i)
\]

\[
\square
\]

We define the conversions between ground positive Boolean formulas and ground types without type constructors.

**Definition 4.114.** For a given ground Boolean formula \( \Psi \) in which negation is in front of constants we define \( T(\Psi) \) by the following inductive equation:

\[
T(\Psi_1 \lor \Psi_2) = T(\Psi_1) \cup T(\Psi_2)
\]

\[
T(\Psi_1 \land \Psi_2) = T(\Psi_1) \cap T(\Psi_2)
\]

\[
T(1) = \top
\]

\[
T(0) = \bot
\]

\[
T(-1) = \bot
\]

\[
T(-0) = \top
\]

\[
\square
\]
4.5. Hardness of Type Checking

**Definition 4.115.** For a given ground type \( \Phi \) without type constructors we define \( F(\phi) \) by the following inductive equation:

\[
F(\Phi_1 \cup \Phi_2) = F(\Phi_1) \lor F(\Phi_2) \\
F(\Phi_1 \cap \Phi_2) = F(\Phi_1) \land F(\Phi_2) \\
F(\top) = 1 \\
F(\bot) = 0
\]

Let us notice the simple fact:

**Remark 4.116.** Let \( \Psi \) be a ground type without type constructors. Let \( \Phi \) be a ground Boolean formula in which negation appears only in front of constants. Then

1. \( \vdash T(\Phi) = \top \) if and only if \( \models \Phi \)
2. \( \vdash \Psi = \top \) if and only if \( \models F(\Psi) \)

where \( \models \Phi \) means that \( \Phi \) is valid.

The remaining part of this section is divided into two subsections. In first we show that type checking programs with \( \cap \)-closed families of types is \( \text{co-NP} \) hard. In the second we show that if we allow arbitrary types then type checking becomes \( \Sigma^P_2 \)-hard.

### 4.5.1 \( \text{co-NP} \) Hardness for \( \cap \)-closed Families of Types

**Definition 4.117.** TAUT is the problem of checking whether or not a propositional Boolean formula is a tautology.

The proof of the next lemma can be found e.g. in [18].

**Lemma 4.118.** TAUT is \( \text{co-NP} \) complete.

We show, by reduction of TAUT, that TCP is \( \text{co-NP} \) hard.

**Theorem 4.3.** TCP for \( \cap \)-closed families of types is \( \text{co-NP} \) hard.

**Proof.** For a given Boolean formula \( \Phi \) we construct a program \( P \) and a directional type \( T \) in such a way that \( \Phi \) is a tautology if and only if \( P \) is well typed with respect to \( T \). This, by Lemma 4.118 gives \( \text{co-NP} \) hardness of type checking (TCP).

Assume that \( \Phi \) contains variables \( \alpha_1, \ldots, \alpha_n \). We can assume, without loss of generality, that the negation appears only in front of variables.

Let \( P \) be the following program:

\[
\begin{align*}
\text{or}(X,Y,X) & . \\
\text{or}(X,Y). & \\
\text{and}(X,X, \ldots, X). & \text{'and' has arity } n+1
\end{align*}
\]

\[
\begin{align*}
p(X_1,X_1', \ldots, X_n,X_n', Z) & : - & \text{(*)} \\
\text{or}(X,X',Z_1), \ldots, \text{or}(X_n,X_n',Z_n), & \\
\text{and}(Z_1, \ldots, Z_n,Z). & 
\end{align*}
\]

Let \( T \) be the following directional type:

\[
\begin{align*}
\text{or} : (\alpha, \beta, \top) & \rightarrow (\alpha, \beta, \top) \\
\text{and} : (\alpha_1, \ldots, \alpha_n, \top) & \rightarrow (\alpha_1, \ldots, \alpha_n, \alpha_1 \cap \cdots \cap \alpha_n)
\end{align*}
\]

\]
Chapter 4. Type Checking

\[ p : (\alpha_1, \alpha'_1, \ldots, \alpha_n, \alpha'_n) \rightarrow \]  
\[ \rightarrow (\alpha_1, \alpha'_1, \ldots, \alpha_n, \alpha'_n, \Psi \cup (\alpha_1 \cap \alpha'_1) \cup \cdots \cup (\alpha_n \cap \alpha'_n)) \]  

where \( \Psi \) is obtained from \( \Phi \) by replacing each \( \neg \alpha \) by \( \alpha' \), replacing each \( \vee \) by \( \cup \) and

replacing each \( \wedge \) by \( \cap \).

Let \( \tau \rightarrow \sigma \) be the abbreviation for (4.89).

It is easy to show that part of \( P \) consisting of predicates and \( \alpha \) is well typed with respect to \( T \). So, well typedness of \( P \) with respect to \( T \) depends on the fact whether or not

\[ \vdash \text{ClauseHasType}(T, (*), \tau \rightarrow \sigma) \]  

Let \( V \) be the following vector of directional type:

\[ V_i = (\alpha, \beta, T) \rightarrow (\alpha, \beta, \alpha \cup \beta) \]  
\[ V_{n+1} = (\alpha_1, \ldots, \alpha_n, T) \rightarrow (\alpha_1, \ldots, \alpha_n, \alpha_1 \cap \cdots \cap \alpha_n) \]  

One can check that

\[ \text{ResultType}((*), V, (\alpha_1, \alpha'_1, \ldots, \alpha_n, \alpha'_n, T)) = (\alpha_1, \alpha'_1, \ldots, \alpha_n, \alpha'_n, \tau') \]  

where \( \tau' = (\alpha_1 \cup \alpha'_1) \cap \cdots \cap (\alpha_n \cup \alpha'_n) \).

By Lemma 4.81 and (4.91) we have that (4.90) holds if and only the following formula is provable:

\[ \vdash (\alpha_1 \cup \alpha'_1) \cap \cdots \cap (\alpha_n \cup \alpha'_n) \leq \Psi \cup (\alpha_1 \cap \alpha'_1) \cup \cdots \cup (\alpha_n \cap \alpha'_n) \]  

Now, we will show that (4.92) is provable if and only if \( \Phi \) is a tautology.

Suppose that (4.92) is provable. By Lemma 4.42 we have that (4.92) holds for every \( \{ \top, \bot \} \)-substitution. Let us take any valuation \( \eta \). Let \( \nu \) be defined by

\[ \nu(\alpha_k) = T(\eta(\alpha_k)) \]  
\[ \nu(\alpha'_k) = T(\neg \eta(\alpha_k)) \]  

for each \( i \in \{1, \ldots, n\} \).

It is easy to check that \( \nu \) is a good substitution and therefore \( \vdash \nu(\alpha_1 \cap \alpha'_1) = \bot \), and

\[ \vdash \nu(\alpha_1 \cup \alpha'_1) = \top \]  

for each \( i \in \{1, \ldots, n\} \).

So, after applying \( \nu \), (4.92) is equivalent to \( \vdash \top \leq \bot \cup \nu(\Psi) \) which is equivalent to

\[ \vdash \top = \nu(\Psi). \]  

Since \( \nu(\Psi) = T(\eta(\Phi)) \) by (4.93) and Remark 4.116 we obtain that \( \vdash \eta(\Phi) \). It holds for any valuation \( \eta \), so, \( \Phi \) is a tautology.

On the other hand suppose that \( \Phi \) is a tautology. By Lemma 4.42 to show that (4.92) holds it suffices to show that (4.92) holds for any \( \{ \top, \bot \} \)-substitution. Let \( \nu \) be a \( \{ \top, \bot \} \)-substitution. We claim that (4.92) holds for \( \nu \). Consider the cases:

1. \( \nu \) is good. Let \( \eta \) be defined by

\[ \eta(\alpha) = F(\nu(\alpha)), \quad \text{for each} \quad i \in \{1, \ldots, n\} \]

After applying \( \nu \) to (4.92) we obtain a formula equivalent to

\[ \vdash \top = \nu(\Psi) \]  

Since \( \nu \) is good, and \( \eta(\Phi) \) holds we can obtain, by Remark 4.116, that (4.94) and (4.92) hold for \( \nu \) which was to be shown.
2. \( \nu \) is not good one. Then, for some \( j \), we have either
   (a) \( \nu(\alpha_j) = \nu(\alpha_j') = T \), or
   (b) \( \nu(\alpha_j) = \nu(\alpha_j') = \bot \).

In the case (a) we have \( \nu(\alpha_j \cap \alpha_j') = T \), so the right-hand side of (4.92) is equivalent to \( T \), which makes (4.92) valid. In case (b) the left-hand side of (4.92) is equivalent to \( \bot \), so (4.92) is valid.

So, for every \( \nu \) the formula (4.92) is satisfied, which establishes our claim. \( \square \)

Theorem 4.3 implies a simple corollary:

**Corollary 4.119.** The problem of checking whether \( \vdash \tau_1 \leq \tau_2 \) is co-NP hard.

### 4.5.2 \( \Sigma^P_2 \) Hardness for Arbitrary Types

We show hardness of type checking (TCP) by reduction of validation problem of special kind of quantified Boolean formula to TCP.

**Definition 4.120.** The QBF (Quantified Boolean Formulas) is the least set satisfying following conditions:

- the variable \( x \) is in QBF,
- if \( E_1, E_2 \) are in QBF then \( \neg E_1, E_1 \lor E_2 \) and \( E_1 \land E_2 \) are in QBF,
- if \( x \) is a variable, \( E \) is in QBF then \( (\forall x)E \) and \( (\exists x)E \) are in QBF.

The value of a formula in QBF without free occurrences of variables is obtained by replacing every \( (\forall x)E \) by \( E[x := 0] \land E[x := 1] \) and every \( (\exists x)E \) by \( E[x := 0] \lor E[x := 1] \).

**Definition 4.121.** By \( \exists \forall \) QBF formula we denote the subset of QBF consisting of formulas of the form \( \exists x_1 \ldots \exists x_n \forall y_1 \ldots \forall y_m E \) where \( E \) is a Boolean formula with variables \( x_1, \ldots, x_n, y_1, \ldots, y_m \). By \( QSAT_2 \) we denote the problem of checking whether \( \exists \forall \) QBF formula is valid.

The polynomial hierarchy was introduced in [22]. The proof of the next lemma can be found in [25].

**Lemma 4.122.** \( QSAT_2 \) is \( \Sigma^P_2 \) hard.

With this lemma we can prove the following one:

**Theorem 4.4.** TCP for arbitrary types is \( \Sigma^P_2 \)-hard.

**Proof.** The proof is by reduction of \( QSAT_2 \) to TCP. Let \( F = \exists \forall \Phi \) be a \( \exists \forall \) QBF formula.

The main idea is similar to the idea from the proof of Theorem 4.3. Let \( P \) be the following program:

```prolog
select(X, X, X).

or(X, Y, X).
or(X, Y, Y).

and(X, X, ..., X). % 'and' has arity n+1
p(X1, X1', ..., Xn, Xn', Y1, Y1', ..., Ym, Ym', Z):- %(+)
or(X, X', Z1), ..., or(Xn, Xn', Zn),
```

83
and $(Z_1, \ldots, Z_n, Z)$.

\text{select}(Z, Y_1, Y_1'), \ldots, \text{select}(Z, Y_m, Y_m').

Let

$$A = \alpha_1, \alpha'_1, \ldots, \alpha_n, \alpha'_n, \beta_1, \beta'_1, \ldots, \beta_m, \beta'_m.$$ 

Let $\Psi$ be a type obtained from $\Phi$ by replacing each $\neg \alpha$ by $\alpha'$, replacing each $\neg \beta$ by $\beta'$, replacing each $\lor$ by $\land$ and replacing each $\land$ by $\land$.

Let $T$ be a directional type:

\begin{align*}
(4.95) & \quad \text{or} : (\alpha, \beta, \top) \rightarrow (\alpha, \beta, \alpha \lor \beta) \\
(4.96) & \quad \text{and} : (\alpha_1, \ldots, \alpha_n, \top) \rightarrow (\alpha_1, \ldots, \alpha_n, \alpha_1 \land \cdots \land \alpha_n) \\
(4.97) & \quad \text{select} : (\alpha, \beta, \gamma) \rightarrow (\alpha \land \gamma, \beta, \gamma) \\
(4.98) & \quad \text{select} : (\alpha, \beta, \gamma) \rightarrow (\alpha \land \beta, \gamma)
\end{align*}

\begin{align*}
(4.99) & \quad p : (A, \top) \rightarrow (A, \Psi \cup (\alpha_1 \land \alpha'_1) \cup \cdots \cup (\alpha_n \land \alpha'_n) \cup (\beta_1 \land \beta'_1) \cup \cdots \cup (\beta_m \land \beta'_m))
\end{align*}

Note that $T$ is not $\cap$-closed, since it does not contain the type $\text{select} : (\alpha, \beta, \gamma) \rightarrow (\alpha \land \beta \land \gamma, \beta, \gamma)$. Let $\tau \rightarrow \sigma$ be the abbreviation for the type of $p$ in $T$.

It is easy to show that part of $P$ consisting of predicates and or, select is well typed with respect to $T$. So, well typedness of $P$ with respect to $T$ depends on the fact whether or not

\begin{align*}
(4.100) & \quad \vdash \text{ClauseHasType}(T, (+), \tau \rightarrow \sigma)
\end{align*}

We claim that, in our system the clause $(+)$ has type (4.99) if and only if the formula $F$ is valid.

So, suppose that $F$ is valid. Let $\pi$ be a vector of values of existentially quantified variables ($\beta$) for which $F$ is valid. We define

$$\delta_i = \begin{cases} 
\beta_i & \text{ if } x_i = 1 \\
\beta'_i & \text{ if } x_i = 0
\end{cases}$$

Let $V$ be the following vector of directional type:

\begin{align*}
\forall i \in \{1, \ldots, n\} \\
V_{n+1} = \begin{cases} 
\text{type (4.97)} & \text{if } x_i = 0 \\
\text{type (4.98)} & \text{if } x_i = 1
\end{cases} (i \in \{1, \ldots, m\})
\end{align*}

One can check that

\begin{align*}
\text{ResultType}((+), V, (A, \top)) = (A, \tau')
\end{align*}

where $\tau' = (\alpha_1 \cup \alpha'_1) \cap \cdots \cap (\alpha_n \cup \alpha'_n) \cap \delta_1 \cap \cdots \cap \delta_m$.

By Lemma 4.81 we have that clause $(+)$ has type (4.99) if and only the following formula is provable:

\begin{align*}
(4.102) & \quad (\alpha_1 \cup \alpha'_1) \cap \cdots \cap (\alpha_n \cup \alpha'_n) \cap \delta_1 \cap \cdots \cap \delta_m \leq \\
& \quad \leq \Psi \cup (\alpha_1 \land \alpha'_1) \cup \cdots \cup (\alpha_n \land \alpha'_n) \cup (\beta_1 \land \beta'_1) \cup \cdots \cup (\beta_m \land \beta'_m)
\end{align*}

By Lemma 4.42 it suffices to show that (4.102) is provable for all $(\top, \bot)$-substitution. So, let us take any $(\top, \bot)$-substitution $\nu$. We want to show that (4.102) is satisfied for $\nu$. Let us consider two cases:
4.5. Hardness of Type Checking

- $\nu$ is a good substitution. Let $\eta$ be defined by

\[
\eta(\alpha_i) = F(\nu(\alpha_i)), \text{ for each } i \in \{1, \ldots, n\} \\
\eta(\beta_i) = F(\nu(\alpha_i)), \text{ for each } i \in \{1, \ldots, n\}
\]

Consider two cases.
- First, let us assume that, for all $i$, $\nu(\beta_i) = T(x_i)$. Then $\vdash \eta(\Phi)$, and by Remark 4.116 we have that $\vdash \nu(\Psi) = \top$. So, (4.102) after applying $\nu$ will be provable.
- On the other hand, assume that for some $j$, $\nu(\beta_j) \neq T(x_j)$. Then, since $\nu$ is good,

\[
(4.103) \quad \nu(\beta_j) = \neg x_j \text{ and } \nu(\beta_j') = x_j
\]

By the definition of $\delta_j$ we have

\[
\nu(\delta_j) = \begin{cases} 
\nu(\beta_j) & \text{if } x_j = 1 \\
\nu(\beta_j') & \text{if } x_j = 0
\end{cases}
\]

Since $\nu$ is a good substitution, by (4.103), we have:

\[
\nu(\delta_j) = \begin{cases} 
\neg x_j & \text{if } x_j = \top \\
x_j & \text{if } x_j = \bot = \bot
\end{cases}
\]

So, after applying $\nu$, the left side of 4.102 will be equivalent to $\bot$ and (4.102) becomes provable for substitution $\nu$.

- $\nu$ is not a good substitution. Then we have either $\nu(\alpha_k) = \nu(\alpha_k')$ for some $k \in \{1, \ldots, n\}$ or $\nu(\beta_k) = \nu(\beta_k')$ for some $k \in \{1, \ldots, n\}$. If the first case holds the proof runs exactly the same way as in Theorem 4.4. If $\nu(\beta_k) = \nu(\beta_k')$ we have to consider two possibilities: (a) $\nu(\beta_j) = \nu(\beta_j') = \top$ or (b) $\nu(\beta_j) = \nu(\beta_j') = \bot$.

(a) is again as in the previously mentioned proof. If (b) holds then $\delta_j$, which is either $\beta_j$ or $\beta_j'$, is equal to $\bot$ and the left side of the 4.102 is also equivalent to $\bot$, and 4.102 is provable for $\nu$.

Now, suppose that $p$ has a type (4.99). So, there must be a vector $V$ of types for which $\text{ResultType}(\langle + \rangle, C, \langle A, \top \rangle)$ returns a type $\tau_r = \langle A, \tau' \rangle$ such that

\[
(4.104) \quad \tau' \leq \Psi \cup \langle \alpha_1 \cap \alpha_1' \rangle \cup \cdots \cup \langle \alpha_n \cap \alpha_n' \rangle \cup \langle \beta_1 \cap \beta_1' \rangle \cup \cdots \cup \langle \beta_n \rangle \cap \langle \beta_m' \rangle
\]

Let us define a vector of variables:
1. $\delta_i = \beta_i$ if $V_{i+n+1}$ is equal to a type (4.97)
2. $\delta_i = \beta_i'$ if $V_{i+n+1}$ is equal to a type (4.98)

One can check that

\[
(4.105) \quad \tau' = \langle \alpha_1 \cup \alpha_1' \rangle \cap \cdots \cap \langle \alpha_n \cup \alpha_n' \rangle \cap \delta_1 \cap \cdots \cap \delta_m
\]

From this vector we take the values of existentially quantified variables of the formula $F$. Let us take any valuation $\eta$ for which

\[
(4.106) \quad \eta(\beta_i) = \begin{cases} 
1 & \text{if } \delta_i = \beta_i \\
0 & \text{if } \delta_i = \beta_i'
\end{cases}
\]

We want to show that $\eta(\Phi)$ is valid. Let us define a substitution $\nu$ such that $\nu(\gamma) = T(\eta\gamma)$. Applying $\nu$ to (4.104) we obtain the formula equivalent to $\top = \nu(\Psi)$ and since $\nu(\Psi) = T(\eta(\Phi))$ by Remark 4.116 we obtain that $\eta(\Phi)$ is valid. This implies that $F$ is valid.
Chapter 5

Reconstruction of Guarantees

In the previous chapter we focused on the task of type checking. This chapter begins discussion of type reconstruction.

The objective of type reconstruction is to assign types to predicates in (partially) untyped program. We expect that types assigned by the algorithm to a predicate describe a program well, i.e. (1) are correct which in our system means provable, (2) give as much information about the program as possible. In our system there are two natural questions connected with type reconstruction: reconstruction of guarantees (an output type) when an input type is given and reconstruction of entire directional type.

In this chapter we analyze the first variant of a type reconstruction — finding a guarantees of a predicate when assumptions are known. We consider discriminative types. Moreover, we expect that a user provides information about the shape of a guarantee. Given this data the algorithm finds the best discriminative guarantees of a given shape.

We will consider \( \cap \)-closed families of types, since for other types, in general, best guarantees do not exist.

5.1 The Algorithm

**Definition 5.1.** For a type \( \tau \) we define \( \text{shape}(\tau) \) as follows:

\[
\begin{align*}
\text{shape}(\top) &= \text{shape}(\bot) = \text{shape}(\alpha) = \text{shape}(\alpha) = \top \\
\text{shape}(\tau_1 \cap \tau_2) &= \text{shape}(\tau_1) \cap \text{shape}(\tau_2) \\
\text{shape}(\tau_1 \cup \tau_2) &= \\
&\begin{cases} 
\text{shape}(\tau_1) & \text{if } \vdash \text{shape}(\tau_2) = \top \\
\text{shape}(\tau_2) & \text{if } \vdash \text{shape}(\tau_1) = \top \\
\text{shape}(\tau_1) \cup \text{shape}(\tau_2) & \text{otherwise}
\end{cases}
\end{align*}
\]

**Example 5.1.** We have

\[
\vdash \text{shape}(F(\bot, G(\alpha \cup \beta)) = F(\top, G(\top \cup \top)) = F(\top, G(\top))
\]

We have also that

\[
\text{shape(list(real))) \cup tree(\beta \cap list(\bot))) = \text{list(\top) \cup tree(list(\top)))}
\]

\[\square\]
**Definition 5.2.** We say that type is a simple shape if it contains only type constructors and constant \( \top \).

In this section we present an algorithm which, for given assumption type, computes the best discriminative (see Definition 4.40) guarantees of a shape which is compatible with the given shape. Consider an example.

**Example 5.2.** If a programmer wants to use the predicate `append` to concatenate 2 lists, he declares the assumptions

\[(\text{list}(\alpha), \text{list}(\beta), \top)\]

and requires that the third argument should be a list, i.e. the guarantee should have the shape

\[(\text{list}(\top), \text{list}(\top), \text{list}(\top))\]

In this case the algorithm outputs:

\[
\text{append} : (\text{list}(\alpha), \text{list}(\beta), \top) \rightarrow (\text{list}(\alpha), \text{list}(\beta), \text{list}(\alpha \cup \beta))
\]

The algorithm uses two operations on types. The first one is the discriminative approximation \( A \), (see Definition 4.36), the second is defined below.

**Definition 5.3.** The operator \( A_s \) takes a simple shape \( \sigma \) as a first argument and transforms a type \( \tau \) which is its second argument according to \( \sigma \).

\[
\begin{align*}
A_s(\sigma, \tau \cup \tau_2) &= A_s(\sigma, \tau_1) \cup A_s(\sigma, \tau_2) \\
A_s(\sigma, \tau_1 \cap \tau_2) &= A_s(\sigma, \tau_1) \cap A_s(\sigma, \tau_2) \\
A_s(\top, \alpha) &= \alpha \\
A_s(\top, \alpha) &= a_s \\
A_s(\sigma, \bot) &= \bot \\
A_s(F(\sigma_1, \ldots, \sigma_n), \alpha) &= \top \\
A_s(F(\sigma_1, \ldots, \sigma_n), a) &= a_s \\
A_s(F(\sigma_1, \ldots, \sigma_n), F(\tau_1, \ldots, \tau_n)) &= F(A_s(\sigma_1, \tau_1), \ldots, A_s(\sigma_n, \tau_n)) \\
A_s(F(\sigma_1, \ldots, \sigma_n), G(\tau_1, \ldots, \tau_n)) &= G(\tau_1, \ldots, \tau_n) \\
A_s(\top, \bot) &= \bot \\
A_s(\top, G(\tau_1, \ldots, \tau_n)) &= G(\tau_1, \tau_2, \ldots, \tau_n)
\end{align*}
\]

where \( \alpha \) is an atomic type, \( G, F \) are any distinct type constructors.

**Definition 5.4.** A type \( \tau \) is compatible with a shape \( \sigma \) if \( A_s(\sigma, \tau) = \top \)

Consider an example:

**Example 5.3.** We have

\[
\begin{align*}
(5.1) \quad A_s(F(\top), F(\alpha \cap \beta \cup G(\beta, F(\bot)))) &= F(\alpha \cap \beta \cup G(\top, \top)) \\
(5.2) \quad A_s(F(\top), F(\alpha) \cup F(\beta) \cup G(\gamma)) &= F(\alpha) \cup F(\beta) \cup G(\top) \\
(5.3) \quad A_s(F(\top), F(\alpha) \cup F(\beta \cap \text{int})) &= F(\alpha) \cup F(\beta \cap \text{int})
\end{align*}
\]

Operator \( A_s \) ‘takes out’ from the type such its fragments which are compatible with the given shape. The parts which do not match the shape are simplified (with
a waste of information), i.e. replaced by types with the same head, but with a very shallow structure. In line (5.1) we simplify \( G(\beta, F(\bot)) \) which is beyond the requested shape, in line (5.2) we change \( G(\gamma) \) which do not match the requested shape. If shapes of both arguments are equal, this operation returns its second argument unchanged, as in the (5.3).

Now, we are going to study properties of \( A_s \). To prove the monotonicity of \( A_s \) we need the following lemma.

**Lemma 5.5.** Let \( \sigma \) be any simple shape. Suppose that \( \vdash \tau_1 = \tau_2 \). Then \( \vdash A_s(\sigma, \tau_1) = A_s(\sigma, \tau_2) \).

**Proof.** We show the thesis by induction on the structure of a proof of \( \vdash \tau_1 = \tau_2 \).

(Ax1) Obvious. Types \( \tau_1 \) and \( \tau_2 \) are the same, so \( A_s(\sigma, \tau_1) \) and \( A_s(\sigma, \tau_2) \) are also the same.

(Ax2) We have \( \vdash F(\tau_1, \ldots, \tau_n) = F(\tau'_1, \ldots, \tau'_n) \) with assumptions \( \vdash \tau_i = \tau'_i \) for \( i \in \{1, \ldots, n\} \). First, suppose that \( \sigma = \top \) or \( \sigma = G(\sigma_1, \ldots, \sigma_n) \). Then,
\[
\vdash A_s(\sigma, F(\tau_1, \ldots, \tau_n)) = F(\top, \ldots, \top) = A_s(\sigma, F(\tau'_1, \ldots, \tau'_n))
\]
which establishes the thesis.

Now, suppose that \( \sigma = F(\sigma_1, \ldots, \sigma_n) \). By the induction hypothesis we have that
\[
\vdash A_s(\sigma_i, \tau_1) = A_s(\sigma_i, \tau'_1).
\]
Moreover,
\[
A_s(F(\sigma_1, \ldots, \sigma_n), F(\tau_1, \ldots, \tau_n)) = F(A_s(\sigma_1, \tau_1), \ldots, A_s(\sigma_n, \tau_n)),
\]
\[
A_s(F(\sigma_1, \ldots, \sigma_n), F(\tau'_1, \ldots, \tau'_n)) = F(A_s(\sigma_1, \tau'_1), \ldots, A_s(\sigma_n, \tau'_n)).
\]

From that, by induction hypothesis, and (Ax2), we obtain the thesis.

(Ax3) We have \( \vdash \tau_1 \cap \tau_2 = \tau'_1 \cap \tau'_2 \) with assumptions \( \vdash \tau_1 = \tau'_1 \) and \( \vdash \tau_2 = \tau'_2 \). By the induction hypothesis we have
\[
\vdash A_s(\sigma, \tau_1) = A_s(\sigma, \tau'_1)
\]
\[
\vdash A_s(\sigma, \tau_2) = A_s(\sigma, \tau'_2)
\]
Since,
\[
A_s(\sigma, \tau_1 \cap \tau_2) = A_s(\sigma, \tau_1) \cap A_s(\sigma, \tau_2)
\]
\[
A_s(\sigma, \tau'_1 \cap \tau'_2) = A_s(\sigma, \tau'_1) \cap A_s(\sigma, \tau'_2)
\]
by induction hypothesis and (Ax3) we obtain the thesis.

(Ax4) For \( \vdash \tau_1 \cup \tau_2 = \tau'_1 \cup \tau'_2 \) the proof is very similar.

(Ax5) We have assumptions \( \tau_1 = \tau'_1 \), \( \tau_2 = \tau'_2 \) and \( \tau_1 = \tau_2 \) with the conclusion \( \tau'_1 = \tau'_2 \). Inductive hypothesis says that for each \( \sigma \)
\[
\vdash A_s(\sigma, \tau_1) = A_s(\sigma, \tau'_1)
\]
\[
\vdash A_s(\sigma, \tau_2) = A_s(\sigma, \tau'_2)
\]
\[
\vdash A_s(\sigma, \tau_1) = A_s(\sigma, \tau_2)
\]
The thesis follows directly from it and from (Ax4).

For (Ax7)–(Ax10) the proofs are similar.
(Ax11) We have $\tau \cup \top = \top$.

By the definition of $A_\sigma$, and axioms (Ax5), (Ax11) we obtain

$$\vdash A_\sigma(\sigma, \tau \cup \top) = A_\sigma(\sigma, \tau) \cup A_\sigma(\sigma, \top) = A_\sigma(\sigma) \cup \top = \top = A_\sigma(\sigma)$$

For $\tau \cap \top = \tau$ the proof is similar.

(Ax12) The proof is similar to the case above.

(Ax13) We have $\vdash F(\tau_1, \ldots, \tau_n) \cap F(\tau'_1, \ldots, \tau'_n) = F(\tau_1 \cap \tau'_1, \ldots, \tau_n \cap \tau'_n)$.

Suppose that $\sigma \neq F(\sigma_1, \ldots, \sigma_n)$. Then we have

$$\vdash A_\sigma(\sigma, F(\tau_1, \ldots, \tau_n) \cap F(\tau'_1, \ldots, \tau'_n)) = A_\sigma(\sigma, F(\tau_1, \ldots, \tau_n)) \cup A_\sigma(\sigma, F(\tau'_1, \ldots, \tau'_n))$$

$$= F(\top, \ldots, \top) \cap F(\top, \ldots, \top) = F(\top, \ldots, \top)$$

Now, suppose that $\sigma = F(\sigma_1, \ldots, \sigma_n)$. We have

$$\vdash A_\sigma(\sigma, F(\tau_1, \ldots, \tau_n) \cap F(\tau'_1, \ldots, \tau'_n))$$

$$= A_\sigma(\sigma, \sigma_1, \ldots, \sigma_n), F(\tau_1, \ldots, \tau_n) \cap A(\sigma_1, \ldots, \sigma_n), F(\tau'_1, \ldots, \tau'_n))$$

$$= F(\sigma, \tau_1, \ldots, \tau_n) \cap F(\sigma, \tau'_1, \ldots, \tau'_n))$$

$$= F(\sigma, \tau_1, \ldots, \tau_n) \cap F(\sigma, \tau'_1, \ldots, \tau'_n))$$

(Ax14) We have $\vdash F(\tau_1, \ldots, \tau_n) \cap G(\tau'_1, \ldots, \tau'_n) = \bot$. We consider three cases:

$\sigma = F(\sigma_1, \ldots, \sigma_n)$

$\sigma = G(\sigma_1, \ldots, \sigma_m)$

$\sigma = H(\sigma_1, \ldots, \sigma_k) \text{ or } \sigma = \top$

where $H$ is a type constructor different from $F$ and $G$.

In all these cases proofs are similar to the proof of case

$$\sigma \neq F(\sigma_1, \ldots, \sigma_n)$$

for axiom (Ax13).

(Ax15) We have $\vdash \tau_1 \cup \tau_2 = \text{or}(\tau_1, \tau_2)$, for atomic types $\tau_1$ and $\tau_2$.

Suppose that $\sigma \neq \top$. Then

$$\vdash A_\sigma(\sigma, \tau_1 \cup \tau_2) = A_\sigma(\sigma, \tau_1) \cup A_\sigma(\sigma, \tau_2)$$

$$= \tau_1 \cup \tau_2 = \text{or}(\tau_1, \tau_2)$$

On the other hand suppose that $\sigma = \top$. Then we have

$$\vdash A_\sigma(\top, \tau_1 \cup \tau_2) = A_\sigma(\sigma, \tau_1) \cup A_\sigma(\sigma, \tau_2)$$

$$= \tau_1 \cup \tau_2 = \text{or}(\tau_1, \tau_2)$$

(Ax16) The proof is as above.
function Guarantee($Q, T, P, T, \Xi$)
  $Q$ — a program
  $T$ — directional type for $Q$
  $P$ — set of untyped predicates
  $T$ — function which for $p$ returns its assumption
  $\Xi$ — function which for $p$ returns a desired shape of its guarantees
  where $p$ is a predicate from $P$
  let $T_R = \{(p : T(p) \rightarrow \bot) : p \in R\}$
  repeat
    choose $p \in R$
    take a clause $C = p(t) \vdash q_1(t_1), \ldots, q_n(t_n)$ from definition of $p$
    take $\tau \rightarrow \varrho$ such that $(p : \tau \rightarrow \varrho) \in T_R$
    let $\varrho' = $ ResultType2($C, T \cup T_R, \tau$)
    if $\varrho' = \text{error}$ then return type error
    else
      let $\varrho''$ be $A_s(\Xi(p), \varrho \cup \varrho')$
      replace $(p : \tau \rightarrow \varrho)$ in $T_R$ by $(p : \tau \rightarrow \varrho'')$
  until the guarantees cannot be changed by any choice of the clause
  return $T_R$

Algorithm 5.1: Guarantee reconstruction algorithm

Let us state a simple corollary to Lemma 5.5.

**Corollary 5.6.** $A_s$ is monotonic, i.e. for each types $\tau_1, \tau_2$ such that $\vdash \tau_1 \leq \tau_2$ and each simple shape $\sigma$ we have

$$\vdash A_s(\sigma, \tau_1) \leq A_s(\sigma, \tau_2)$$

**Proof.** It follows from Lemma 5.5 and Lemma 4.51 since $A_s(\sigma, \tau_1 \cap \tau_2) = A_s(\sigma, \tau_1) \cap A_s(\sigma, \tau_2)$.

Now, we give Algorithm 5.1 which for a program, directional type of some predicates, assumptions and shapes of guarantees for other predicates reconstructs the missing parts of the directional type.

In our system a type inference is strictly connected to proving well typedness. So we will use the predicate ResultType2, which operates on $\cap$-closed types.

The algorithm begins with very strong types (probably incorrect) and then tries to correct them by weakening. Function $A$, a discriminative approximation, can be performed since we look for discriminative types, $A_s$ can be performed since we look for types with shape compatible with the given one.

The next example describes a situation when a shape of guarantee given by an user does not match the predicate, i.e. the best guarantees have another shape:

**Example 5.4.** A programmer wants to use the predicate `append` to concatenate 2 lists and obtain a tree. He declares the assumptions

$$(\text{list}(\alpha), \text{list}(\beta), \top),$$

and requires that third argument should be a tree, i.e. the guarantee should have the shape

$$(\text{list}(\top), \text{list}(\top), \text{tree}(\top))$$
Chapter 5. Reconstruction of Guarantees

In that case algorithm outputs:

\[ \text{append} : (\text{list}(\alpha), \text{list}(\beta), \top) \to (\text{list}(\alpha), \text{list}(\beta), \text{list}(\top)) \]

since \( A_\ast(\text{tree}(\top), \text{list}(\alpha \cup \beta)) = \text{list}(\top) \)

This example reveals the nice feature of the algorithm — wrongly guessed shapes of guarantees are corrected. This algorithm in many cases can be used as an algorithm which reconstructs guarantees without predefined shapes, and perhaps suggests other assumptions. The dialog with a user could as be follows:

\[
\begin{align*}
Q & : (\text{list}(\alpha), \text{list}(\beta), \top), (\top, \top, \top) \\
A & : (\text{list}(\top), \text{list}(\top), \text{list}(\top)) \\
Q & : (\text{list}(\alpha), \text{list}(\beta), \top), (\text{list}(\top), \text{list}(\top), \text{list}(\top)) \\
A & : (\text{list}(\alpha), \text{list}(\beta), \text{list}(\alpha \cup \beta)) \\
Q & : (\text{list}(\alpha), \text{list}(\beta), \text{list}(\gamma)), (\text{list}(\top), \text{list}(\top), \text{list}(\top)) \\
A & : (\text{list}(\alpha \cap \gamma), \text{list}(\beta \cap \gamma), \text{list}(\gamma \cap (\alpha \cup \beta))
\end{align*}
\]

There are, however, programs for which the shape of a guarantee cannot be found automatically by such a procedure. We consider this problem in the next chapters.

Now, we state some theorems on this algorithm. If in a directional type \( T \) there is only one type for predicate \( p \) we denote this type by \( T(p) \).

**Definition 5.7.** We say that \( T \) is a directional type of a program \( P \) if \( P \) is well-typed with respect to \( T \).

**Definition 5.8.** Let \( Q \) be a program. Let \( T \) be a directional type of \( Q \). Let \( P \) be a program whose clauses contain calls of predicates from \( P \) and \( Q \), such that \( P \cap Q \neq \emptyset \). Let \( T(p) \), for a predicate \( p \) in \( P \) be the assumption for \( p \). Let \( \Xi(p) \), for a predicate \( p \), be the requested shape of guarantees for \( p \). We denote an ordered tuple \( (Q, T, P, T, \Xi) \) by a RG task.

**Lemma 5.9.** Let \( (Q, T, P, T, \Xi) \) be a RG-task. Assume that \( T_P = \text{Guarantee}(Q, T, P, T, \Xi) \). If \( T_P \) is different than error then \( T \cup T_P \) is a directional type for \( P \).

**Proof.** We have to show that for each predicate \( p \in Q \), and for each clause \( C \) defining \( p \) we have

\[ \vdash \text{ClauseHasType}(T \cup T_P, C, T_P(p)) \]

Let \( \tau \to \sigma = T_R(p) \). Since the fixpoint has been reached we have

\[ \vdash \text{ReturnType2}(C, T \cup T_P, \tau) \cup \sigma = \sigma \]

which is equivalent to

\[ \vdash \text{ReturnType2}(C, T \cup T_P, \tau) \leq \sigma. \]

This, by Lemma 4.87, implies

\[ \vdash \text{ClauseHasType}(T \cup T_R, C, T_R(p)) \]

which establishes the thesis.

We have shown that the algorithm finds types for which one can prove type correctness of every clause. It remains to show that computed by the algorithm types are minimal. The next lemma says that the guarantees obtained in the algorithm are the best possible in the class of discriminative types of a shape compatible with a given one. We need some definitions:
5.1. The Algorithm

**Definition 5.10.** Let $P$ be a program. Let $T$ be the function which for a predicate $p$ in $P$ returns assumption for $p$. Let $\Xi(p)$ be a function which for $p$ in $P$ returns a simple shape. We say that directional type $T$ is compatible with $P, T, \Xi$ if
1. for each predicate $p$ in $P$ there is exactly one element $(p : \tau \rightarrow \sigma)$ in $P$ and
2. $\tau = T(p)$, and $\sigma = A(A_s(\Xi(p), \sigma))$.

**Definition 5.11.** Let $T_1, T_2$ are directional types containing only one type for a predicate. We say that $T_1$ has stronger guarantees than $T_2$, if, for each predicate $p$, if $(p : \tau_1 \rightarrow \sigma_1) \in T_1$, and $(p : \tau_2 \rightarrow \sigma_2) \in T_2$ then $\tau_1 = \tau_2$, and $\vdash \sigma_1 \leq \sigma_2$. □

**Theorem 5.1.** Let $\langle Q, T, P, T, \Xi \rangle$ be a RG-task. Let

$$T_P = \text{Guarantee}(Q, T, P, T, \Xi).$$

Let $T'$ be a directional type having exactly one type for each predicate in $P$, such that $T'$ is compatible with $P, T, \Xi$. Moreover, assume that $T \cup T'$ is a directional type of $P \cup Q$. Then $T_P$ is different from $\text{error}$ and $T_P$ has stronger guarantees than $T'$.

**Proof.** Let us define functions $G$ and $G'$ from predicates to types

$$G(p) = \sigma, \text{ if } T_P(p) = \tau \rightarrow \sigma$$

$$G'(p) = \sigma, \text{ if } T'(p) = \tau \rightarrow \sigma$$

It is enough to show that for each $p$ in $P$ we have that $\vdash G(p) \leq G'(p)$.

During the execution of the function Guarantee the directional type $T_P$ is changing. We denote its consecutive values by $T^1_P, T^2_P, \cdots$. We define the functions $G_i$ for predicates in $P$

$$G_i(p) = \sigma, \text{ if } T^i_P(p) = \tau \rightarrow \sigma$$

since $G = G_k$ for some $k$, it suffices to show that for each $i$, and $p$ in $P$

$$\vdash G_i(p) \leq G'(p)$$

(5.4)

We show it by induction on $i$. The basic step is obvious, since for each $p$, $G_i(p)$ is equal to $\perp$. So suppose that (5.4) holds for $i$. We choose a predicate $p$ and one of its clauses $C$. Suppose that it has the assumptions $\tau$. Since $T'$ is a directional type of $P$ we have

(5.5) $\text{ClauseHasType}(T \cup T', C, G'(p))$

By Lemma 4.87 and (5.5) we have

(5.6) $\tau' = \text{ResultType2}(C, \tau, T') \leq G'(p)$

By (5.4) we have that for each $q : (\tau \rightarrow \sigma') \in T'$ there is $q : (\tau \rightarrow \sigma) \in T^i_P$ such that $\sigma \leq \sigma'$. Let

$$\tau = \text{ResultType2}(C, \tau, T \cup T^i_P)$$

Then, by Lemma 4.88, we have that $\tau$ is different from $\text{error}$ and $\tau \leq \tau'$. By (5.6) we have $\tau \leq G'(p)$. So, by (5.4)

$$G_i(p) \cup \tau \leq G'(p),$$
Since $T$ is compatible with $P,T, \Xi$, and by monotonicity of $A$ and $A_s$ we get

$$A(A_s(\Xi(p), G_i(p) \cup \tau_i)) \leq G'(p)$$

which is equivalent to

$$G_{i+1}(p) \leq G'(p)$$

This finishes the proof. \(\Box\)

### 5.1.1 Complexity of Algorithm 5.1

In order to analyze the complexity of Algorithm 5.1 we need some facts about the sequences of types.

**Lemma 5.12.** Let $\alpha_1, \ldots, \alpha_n$ be type variables. The number $N$ of different types of the form $\alpha_1 \cap \cdots \cap \alpha_n$ is equal to $2^n$.

**Lemma 5.13.** Let $\alpha_1, \ldots, \alpha_n$ be type variables. Let $(\tau_i)_{i \in \{1, \ldots, m\}}$ be a sequence of types built from variables $\alpha_i$, $\top$, $\bot$ and operators $\cap, \cup$ such that for all $i < m$ we have

\begin{align*}
\vdash \tau_i & \leq \tau_{i+1} \quad \text{and not} \quad \vdash \tau_1 = \tau_2
\end{align*}

Then

\begin{align*}
\vdash m < 2^n
\end{align*}

**Proof.** Suppose that all $\tau_i$ are in L-normal form. We have that $\tau_i \leq \tau_{i+1}$, which is equivalent to $\tau_{i+1} = \tau_i \cup \tau_{i+1}$. So, since $\tau_{i+1}$ is strictly greater than $\tau_i$, there must be a $\alpha_m \cap \cdots \cap \alpha_{n}$ absent in $\tau_i$. Every such conjunct can be added only once. According to Lemma 5.12 number of such conjuncts is $2^n$ which gives the thesis. \(\Box\)

**Lemma 5.14.** Let $\tau$ be a type which is a Boolean combination of type variables, atomic types and types of the form $F(\top, \ldots, \top)$. Then $\vdash A(\tau) = \tau$.

**Proof.** The R-normal form of $\tau$ has the form $\tau_1 \cap \cdots \cap \tau_k$ where each $\tau_k$ is a union of type variables and types of the form $F^k(\top, \ldots, \top)$, where $F^k$ is a type constructor and in each $\tau_i$ all $F^k$ are distinct. Such types are discriminative. \(\Box\)

### Theorem 5.2.** Algorithm 5.1 finds the answer in EXPTIME.

**Proof.** Sketch. In Algorithm 5.1 there are three sources of complexity: (1) execution of ResultType2 which, due to considerations from chapter 4, works in single exponential time and produces single exponential output, (2) computation of $A$ and $A_s$, (3) reaching the fixpoint.

First we describe how to compute the values of $\tau' = A(A_s(\sigma, \tau))$. It is a simple corollary to definition of $A_s$ that value of $\tau'' = A_s(\sigma, \tau)$ can be computed in linear time. In $\tau'$ variables occur only in Boolean combinations of variables and types of the form $F(\top, \ldots, \top)$, where $F$ is a type constructor.

By Lemma 5.14 such types are discriminative. Therefore, we can compute $\tau'$ in polynomial time using term rewriting system $\rightarrow_\tau$ defined in Section 4.1.3.

\[^1\text{i.e., such types for which one cannot prove that they are equal.}\]
5.2. Hardness of the Reconstruction of Guarantees

Now, we analyze the number of changes done before the fixpoint is reached. For a shape \( \sigma \) we define a set \( T^\sigma \) by the following inductive equations:

\[
\tau \in T^F(\sigma_1, \ldots, \sigma_n) \text{ iff } \tau \in \{ \top, \bot \} \text{ or } \tau = F(\tau_1, \ldots, \tau_n) \cup \bigcup_{i=1}^{m} a_i \cup \bigcup_{i=1}^{k} G_k(\top, \ldots, \top)
\]

where each \( \tau_i \in T^{\sigma_i} \).

\( \tau \in T^\top \) iff \( \tau \) is a Boolean combination of atomic types, \( \top, \bot \) and types of the form \( G(\top, \ldots, \top) \) where \( G \) is any type constructor.

One can show that for each type \( \tau \) such that \( \vdash \tau = A(A_\sigma(\sigma, \tau_0)) \), there is a type \( \tau' \) in \( T^\sigma \) such that \( \vdash \tau = \tau' \). The rewriting rule \( \Rightarrow^\cup \) leads to types in \( T^\sigma \).

Now, we analyze how many times the main loop of a function Guarantee (algorithm 5.1) is executed. We will count only such courses in which a guarantee increases. We estimate the maximal number of increasing changes for types in \( T^\sigma \).

Let \( V \) be the number of variables in assumptions. Let \( M \) be the size of guarantees. Let \( K \) be the number of type constructors. Each leaf can be treated as a Boolean expression over \( K + V \) variables (a combination of variables and types \( F(\top, \ldots, \top) \)). For such expressions we have at most \( 2^{K+V} \) increasing changes. Since there are \( O(M) \) leaves we obtain \( 2^{K+V} O(M) \) changes in the leaves. Moreover, every node can be changed: either \( F(\top, \ldots, \top) \) can be added (in general \( O(M) \cdot O(K) \) changes), or the node can be replaced by \( \top \) — in general \( O(M) \) changes. Gathering it together, we obtain that the number of changes is

\[
2^{K+V} O(M) + O(M) \cdot O(K) + O(M)
\]

which is single exponential. This changes can be interlaced with such runs of the main loop in which no changes are done. For every change there is \( O(N) \) runs of the main loop in which nothing changes. Single exponential number of steps, such that each of them is also single exponential gives single exponential time for the entire execution.

We have the following theorem:

**Theorem 5.3.** Reconstruction of the guarantee is \textsc{ExpTime}-complete.

To prove it we need to show \textsc{ExpTime} hardness of reconstruction of guarantees problem, which is the subject of the next section.

5.2 Hardness of the Reconstruction of Guarantees

**Definition 5.15.** Let \( \langle Q, T, P, T, \Xi \rangle \) be the instance of RG-task, let \( \sigma \) be a type, and finally let \( p \) be a predicate in \( P \).

Assume that \( T_P \) is a solution of \( \langle Q, T, P, T, \Xi \rangle \), i.e. the strongest directional type of \( P \) compatible with \( P, T, \Xi \). Let \( p : \tau_0 \rightarrow \sigma_0 \) be the only type of \( p \) in \( T \).

\( \text{RGP} \) is a problem of checking whether \( \vdash \sigma_0 \leq \sigma \).

In this section we show \textsc{ExpTime} hardness of \text{RGP}. In fact reconstruction of guarantees is hard even if we restrict types of predicates to be \( \cap \)-closed and constructed without type constructors.

In the proof of hardness we use a special kind of extended Boolean formulas.
Definition 5.16. A recursive monotone Boolean formula (RMBF) is an expression of the form $f(x_1, \ldots, x_n) = E$, where $E$ is an expression over the variables $x_1, \ldots, x_n$ defined by the following grammar:

$$E ::= \text{true} \mid \text{false} \mid x_i \ (1 \leq i \leq n) \mid E_1 \lor E_2 \mid E_1 \land E_2 \mid f(E_1, \ldots, E_n)$$

We call an expression $E$ by RMBF-term. Let $B$ denote the Boolean domain $\{\text{true}, \text{false}\}$ with the ordering $\text{false} \preceq \text{true}$. The least fixpoint of RMBF defines a Boolean function, from $B^n$ to $B$.

Definition 5.17. An instance of a RMBF-problem is a pair $(\Psi, a)$ where $\Psi$ is a RMBF, and $a$ is a tuple $(a_1, \ldots, a_n)$. The instance $(\Psi, a)$ is true if and only if $f_0(a_1, \ldots, a_n) = \text{true}$, where $f_0$ is the least fixpoint of $\Psi$.

In this section we consider Boolean formulas with the variables $x_1, \ldots, x_n$ i.e. functions from $B^n$ to $B$.

Definition 5.18. Let $E$ be a RMBF-term. Let $f_0$ be any Boolean formula. By $E[f := f_0]$ we denote Boolean function obtained from $E$ by replacing $f$ with $f_0$.

We need to state the connection between such RMBFs and inequalities of the form $f(x_1, \ldots, x_n) \preceq E$. We have the following lemma.

Lemma 5.19. The least fixpoint of the RMBF

\begin{equation}
(5.9) \quad f(x_1, \ldots, x_n) = E
\end{equation}

is the same as the least solution of the

\begin{equation}
(5.10) \quad f(x_1, \ldots, x_n) \preceq E
\end{equation}

Proof. Sketch. Let $f^{(1)}$ be the least fixpoint of (5.9). Let $f^{(2)}$ be he least solution of (5.10). Let $(f_i)$ be the sequence of the fixpoint approximations given by the equations:

\begin{align}
(5.11) & \quad f_0 = \text{false} \\
(5.12) & \quad f_i = E[f := f_{i-1}]
\end{align}

One can show that for each $i$ we have $f_i \subseteq f_{i+1}$. Moreover, every $f_i$ is monotonic. We show by induction that

\begin{align}
(5.13) & \quad (\forall i)f_i \subseteq f^{(1)} \\
(5.14) & \quad (\forall i)f_i \subseteq f^{(2)}
\end{align}

The thesis is obvious for $i = 0$, since $f_0 = 0$ is the least Boolean function. So, suppose that it holds for $i$. $E$ is a composition of monotonic functions, so it is also monotonic. Because of that and since $f^{(1)}$ is the least fixpoint, by induction hypothesis $f_i \subseteq f^{(1)}$ we have

$$f_{i+1} = E[f := f_i] \subseteq E[f := f^{(0)}] = f^{(1)}$$

Similarly

$$f_{i+1} = E[f := f_i] \subseteq E[f := f^{(2)}] \leq f^{(2)}$$

96
5.2. Hardness of the Reconstruction of Guarantees

Because there is no infinite ascending chain of Boolean function the sequence \( (f_i) \)
becomes constant from some \( k \), i.e. for each \( j > k \) we have that \( f_k = f_j \). Moreover,
\( f_k \) is a solution of both (5.9) and (5.10) which means
\[
(5.15) \quad f^{(1)} \subseteq f_k \\
(5.16) \quad f^{(2)} \subseteq f_k
\]
So, from (5.15), (5.16), (5.13), (5.14),
\[
f^{(1)} = f_k = f^{(2)}
\]
which establishes the lemma. \( \square \)

The following result is due to Hudak and Young [13]:

**Lemma 5.20.** RMBF problem is \( \text{EXPTIME}\)-complete in the length of the instance \( \langle eq, a \rangle \).

We shall construct a program which represents an instance of a RMBF-problem. First we define some auxiliary predicates.

\[\text{loop}(X) : - \text{loop}(X).\]
\[\text{and}(X,X,X).\]
\[\text{or}(X,Y,X).\]
\[\text{or}(X,Y,Y).\]
\[\text{copy}(X,X).\]

These predicates have types:

1. \( \text{loop} : \top \rightarrow \bot \)
2. \( \text{and} : (\alpha, \beta, \top) \rightarrow (\alpha, \beta, \alpha \land \beta) \)
3. \( \text{or} : (\alpha, \beta, \top) \rightarrow (\alpha, \beta, \alpha \lor \beta) \)
4. \( \text{copy} : (\alpha, \top) \rightarrow (\alpha, \alpha) \)

For a RMBF-term we define its translation into the logic goal. This is done by the function \( \mathcal{P} \) which has two arguments. First one is the translated Boolean term \( t \), second is a variable \( X \). \( \mathcal{P} \) returns a logic goal \( G \). The least possible type of the variable \( X \) after performing InferFromAtoms on the goal \( G \) is equivalent to \( t \).

**Definition 5.21.** The function \( \mathcal{P} \) is defined by the following inductive equations:

\[
\mathcal{P}(\text{false}, Z) = \text{loop}(Z) \\
\mathcal{P}(\text{true}, Z) = \emptyset \\
\mathcal{P}(X, Z) = \text{copy}(X, Z) \\
\mathcal{P}(t_1 \land t_2, Z) = \mathcal{P}(t_1, Z_1), \mathcal{P}(t_2, Z_2), \text{and}(Z_1, Z_2, Z) \\
\mathcal{P}(t_1 \lor t_2, Z) = \mathcal{P}(t_1, Z_1), \mathcal{P}(t_2, Z_2), \text{or}(Z_1, Z_2, Z) \\
\mathcal{P}(f(t_1, \ldots, t_n), Z) = \mathcal{P}(t_1, Z_1), \ldots, \mathcal{P}(t_1, Z_1), \text{p}(Z_1, \ldots, Z_2, Z).
\]

where \( Z_1, \ldots, Z_n \) are new distinct variables, \( p \) is the new predicate symbol whose role is to represent \( f \) and \( \emptyset \) means the empty sequence of atoms. \( \square \)
Chapter 5. Reconstruction of Guarantees

Example 5.5. Let \( t \) be a term \( x_1 \lor (x_2 \land f(x_1, x_2)) \). Let \( R \) be a program variable. Then \( \mathcal{P}(t, R) \) is the following goal:

\[
\text{copy}(X_1, Z_1), \text{copy}(X_2, Z_2), \text{copy}(X_1, Z_3), \text{copy}(X_2, Z_4), \\
p(Z_3, Z_4, Z_5)\text{, and}(Z_2, Z_5, Z_6)\text{, or}(Z_1, Z_6, R)
\]

First we prove that this construction meets our intentions. We need to define the connection between logical functions and goals. At first we define a notational convention:

Definition 5.22. A type \( \tau \) and Boolean function (given by Boolean formula) \( f' \) are equivalent (written \( \tau \cong f' \)) iff

\[
\tau[a_i/x_i, \cap/\wedge, \cup/\lor, \top/\text{true, } \bot/\text{false}] \text{ is equivalent to } f'
\]

Example 5.6. \( (a_1 \cup a_2) \cap a_3 \cong x_3 \times x_2 \lor x_1 \times x_3 \), since \( x_3 \times x_2 \lor x_1 \times x_3 \) and \( (x_1 \lor x_2) \land x_3 \) represent the same Boolean function.

In this section we will use the directional type \( T_0 \) which contains types of all auxiliary predicates (and, or, copy, loop).

In this section we consider directional types with only one type for each predicate. So, for every sequence of atoms, and initial environment \( \Gamma' \) there is always the least environment \( \Gamma'' \) such that

\[\vdash \text{InferAtoms}(T, \Gamma, A, \Gamma'')\]

holds. We denote this environment by

\[\Gamma'' = R\mathcal{C}_T(\Gamma, A)\]

Definition 5.23. Let \( G \) be a goal (sequence of atoms). Let \( \tau \) be a type, let

\[T = T_0 \cup \{p : (a_1, \ldots, a_n, \top) \rightarrow (a_1, \ldots, a_n, \tau)\}\]

Let \( \Gamma = R\mathcal{C}_T(\{(X_i : a_i), \ldots, (X_n : a_n)\}, G) \). The goal \( G \) with a selected variable \( Z \) and type \( \tau \) describes the Boolean formula \( b \) with variables \( x_1, \ldots, x_n \) iff

\[BT(\Gamma, Z) \cong b\]

Now, we can state the following lemma.

Lemma 5.24. Let \( E \) be a RMBF-term. Let \( b_f \) be a Boolean formula, (intended to be an interpretation of \( f \)). Assume that \( \tau \cong b_f \). Then \( \mathcal{P}(E, \gamma) \) with a variable \( Y \notin \{X_1, \ldots, X_n\} \) and type \( \tau \) describes \( E[f := b_f] \).

Proof. The proof proceeds by induction on the term \( E \). Let

\[\Gamma_0 = \{X_i : a_i \mid i \in \{1, \ldots, n\}\}\]

Let \( T = T_0 \cup \{p : (a_1, \ldots, a_n, \top) \rightarrow (a_1, \ldots, a_n, \tau)\} \). We consider the several cases:
\section{Hardness of the Reconstruction of Guarantees}

- \( E = \text{false}. \) Then \( \mathcal{P}(\text{false}, Y) = \text{loop}(Y). \) Let
  \[
  \Gamma' = \mathcal{RC}_T(\Gamma_0, \text{loop}(Y))
  \]
  Then \( BT(\Gamma', Y) = \bot \equiv \text{false}. \)
- \( E = \text{true}. \) Then \( \mathcal{P}(\text{true}, Y) = \emptyset. \) Let
  \[
  \Gamma' = \mathcal{RC}_T(\Gamma_0, \emptyset)
  \]
  Then \( \Gamma' = \Gamma \) and since \( Y \notin \{X_1, \ldots, X_n\} \) we get \( BT(\Gamma', Y) = \top \equiv \text{false}. \)
- \( E \) is a variable \( x_i. \) Then \( \mathcal{P}(x_i, Y) = \text{copy}(x_i, Y). \) Let
  \[
  \Gamma' = \mathcal{RC}_T(\Gamma_0, \text{copy}(X_i, Y))
  \]
  Then \( BT(\Gamma', Y) = \alpha_i \equiv x_i. \)
- \( E = E_1 \land E_2. \) Then
  \[
  G = \mathcal{P}(E_1 \land E_2, Y) = \mathcal{P}(E_1, Y_1), \mathcal{P}(E_2, Y_2), \text{and}(Y_1, Y_2, Y)
  \]
  Let
  \[
  \Gamma_1 = \mathcal{RC}_T(\Gamma_0, \mathcal{P}(E_1, Y_1))
  \Gamma_2 = \mathcal{RC}_T(\Gamma_0, \mathcal{P}(E_2, Y_2))
  \Gamma_{12} = \mathcal{RC}_T(\Gamma_0, (\mathcal{P}(E_1, Y_1), \mathcal{P}(E_2, Y_2)))
  \]
  By the induction hypothesis \( \mathcal{P}(E_1, Y_1) \) with \( Y_1 \) and \( \tau \) describes \( E_1 \) and \( \mathcal{P}(E_2, Y_2) \) with \( Y_2 \) and \( \tau \) describes \( E_2. \) So,
  \[
  BT(\Gamma_1, Y_1) \equiv E_1[f := b_f]
  \]
  \[
  BT(\Gamma_2, Y_2) \equiv E_2[f := b_f]
  \]
  Since \( \mathcal{P}(E_1, Y_1) \) does not influence the type of \( Y_2, \) and \( \mathcal{P}(E_2, Y_2) \) does not influence the type of \( Y_1 \) we have
  \[
  \vdash BT(\Gamma_1, Y_1) = BT(\Gamma_{12}, Y_1)
  \]
  \[
  \vdash BT(\Gamma_2, Y_2) = BT(\Gamma_{12}, Y_2)
  \]
  So, if
  \[
  \Gamma_3 = \mathcal{RC}_T(\Gamma_{12}, \text{and}(Y_1, Y_2, Y))
  \]
  then, since and has type \((\alpha, \beta, \top) \to (\alpha, \beta, \alpha \land \beta)\) we get
  \[
  \vdash BT(\Gamma_3, Y) = BT(\Gamma_{12}, Y_1) \cap BT(\Gamma_{12}, Y_2) = BT(\Gamma_1, Y_1) \cap BT(\Gamma_2, Y_2)
  \]
  So, by induction hypothesis we get:
  \[
  BT(\Gamma_3, Y) = BT(\Gamma_1, Y_1) \cap BT(\Gamma_2, Y_2)
  \]
  \[
  \equiv E_1[f := b_f] \land E_2[f := b_f]
  \]
  \[
  = (E_1 \land E_2)[f := b_f]
  \]
  which establishes the proof.
- Formula \( E = E_1 \lor E_2. \) The proof is similar to the case \( E = E_1 \land E_2. \)

99
• Formula $E = f(E_1, \ldots, E_n)$. We have

$$G = \mathcal{P}(f(E_1, \ldots, E_n), \text{Y}) = \mathcal{P}(E_1, \text{Y}_1), \ldots, \mathcal{P}(E_n, \text{Y}_n), p(\text{Y}_1, \ldots, \text{Y}_n, \text{Y})$$

Let for each $i \in \{1, \ldots, m\}$

$$\Gamma_i = \mathcal{R}\mathcal{C}_T(\Gamma_0, \mathcal{P}(E_i, \text{Y}_i))$$

and

$$\Gamma'' = \mathcal{R}\mathcal{C}_T(\Gamma_0, \mathcal{P}(E_1, \text{Y}_1), \ldots, \mathcal{P}(E_m, \text{Y}_m))$$

Induction hypothesis says that, for each $i \in \{1, \ldots, m\}$, we have

$$\vdash BT(\Gamma_i, \text{Y}_i) \cong E_i[f := b_f]$$

Since $\mathcal{P}(E_i, \text{Y}_i)$ does not influence the type of a variable $\text{Y}_j$ for different $i, j$ we have

$$\vdash BT(\Gamma_i, \text{Y}_i) = BT(\Gamma'', \text{Y}_i)$$

for each $i \in \{1, \ldots, m\}$. So, if

$$\Gamma'' = \mathcal{R}\mathcal{C}_T(\Gamma', p(\text{Y}_1, \ldots, \text{Y}_n, \text{Y}))$$

then

$$\vdash BT(\Gamma'', \text{Y}) = \{\beta_1 := BT(\Gamma', \text{Y}_1)\}_{i \in \{1, \ldots, m\}}(\beta_f)$$

$$= \{\beta_1 := BT(\Gamma_i, \text{Y}_i)\}_{i \in \{1, \ldots, n\}}(\beta_f)$$

So, by induction hypothesis we have

$$\vdash BT(\Gamma'', \text{Y}) = \{\beta_1 := BT(\Gamma_i, \text{Y}_i)\}_{i \in \{1, \ldots, m\}}(\beta_f)$$

$$= b_f(E_1[f := b_f], \ldots, E_n[f := b_f]) = f(E_1, \ldots, E_n)[f := b_f]$$

\[ \square \]

**Definition 5.25.** Let $f(x_1, \ldots, x_n) = E$ be the RMBF formula $\Psi$. We define $\text{pred}(\Psi)$ to be the program presented in Figure 5.1. \[ \square \]

Before we state the properties of this construction, we state the connection between ordering on types and Boolean functions.

**Lemma 5.26.** Let $\tau_1, \tau_2$ be types built from variables $\alpha_1, \ldots, \alpha_n$. Moreover, let $\tau_1 \cong f_1$ and $\tau_2 \cong f_2$. Then we have that

$$\vdash \tau_1 \leq \tau_2 \text{ if and only if } f_1 \sqsubseteq f_2$$

We omit the routine proof of this fact. Now, we state a lemma on the function $\text{pred}$.

**Lemma 5.27.** For a given RMBF formula $\Psi = (f(x_1, \ldots, x_n) = E)$ and given assumptions $(\alpha_1, \ldots, \alpha_n, \Gamma)$ the best guarantee of the predicate $p$ in $\text{pred}(\Psi)$ has following properties:
5.2. Hardness of the Reconstruction of Guarantees

\[ \text{loop}(X) := \neg \text{loop}(X). \]

and \((X, X, X). \]

or \((X, Y, X). \]

or \((X, Y, Y). \]

\[ \text{copy}(X, X). \]

\[ p(X_1, \ldots, X_n, Y) := \text{loop}(Y). \]
\[ p(X_1, \ldots, X_n, Y) := \mathcal{P}(E, Y). \]

Figure 5.1: Program representing RMBF formula \( f(x_1, \ldots, x_n) = E \)

1. It is of the form: \((\alpha_1, \ldots, \alpha_n, \tau_0)\) where \(\tau_0\) is a type built from variables \(\alpha_1, \ldots, \alpha_n\), constants \(\bot, \top\) and \(\land, \lor\).

2. Assume that \(f_0\) be a Boolean formula such that \(\tau_0 \supseteq f_0\). Then \(f_0\) is the least fixed point of \(f(x_1, \ldots, x_n) = E\).

Proof. We need to prove two statements.

First clause of the predicate \(p\) enforces the first \(n\) assumptions to stay unchanged, since it has a type: \((\alpha_1, \ldots, \alpha_n, \gamma) \rightarrow (\alpha_1, \ldots, \alpha_n, \bot)\). Moreover, it does not influence the type of the last argument — any type concluded from the second clause will be greater than \(\bot\). Moreover, the best guarantee does not contain function symbol (the only terms in program are variables, types of the auxiliary predicates does not contain type constructors) so it has the desired form.

By Lemma 5.19 it suffices to show that \(f_0\) is the least solution of

\[
(5.18) \quad f(x_1, \ldots, x_n) \supseteq E
\]

Let \(T = T_0 \cup \{p : (\alpha_1, \ldots, \alpha_n, \top) \rightarrow (\alpha_1, \ldots, \alpha_n, \tau_0)\}\). First, we show that \(f_0\) is a solution of \(f(x_1, \ldots, x_n) \supseteq E\). Let

\[
\Gamma_1 = \mathcal{RC}_T(\{X_i : \alpha_i\}, \mathcal{P}(E,Y))
\]

By Lemma 5.24 \(\mathcal{P}(E,Y)\) with \(\tau_0\) and \(Y\) describes \(E[f := f_0]\), which means:

\[
(5.19) \quad \tau_1 = \mathcal{BT}(\Gamma_1, Y) \equiv E[f := f_0]
\]

Since \(\tau_0\) is a guarantee of the predicate \(p\) we have

\[ \vdash \tau_1 \leq \tau_0 \]

Since \(\tau_0 \supseteq f_0\) by Lemma 5.26 and (5.19) we get

\[ E[f := f_0] \subseteq f_0 \]

So \(f_0\) is a solution to (5.18).

Now, we shall prove that \(f_0\) is the least solution. Suppose by contradiction, that there is \(f'\) such that

\[
(5.20) \quad \neg (f' \supseteq f_0) \quad \text{and} \quad f'(x_1, \ldots, x_n) \supseteq E[f := f']
\]

101
\[ \text{is\_int}(X) ::= X = 5. \]
\[ \text{is\_real}(X) ::= X = 3.14. \]
\[ r(Y) ::= \text{val}_1(X_1), \ldots, \text{val}_n(X_n), \]
\[ p(X_1, \ldots, X_n, Y ) . \]
\[ \text{where } \text{val}_i = \begin{cases} \text{is\_int} & \text{if } a_i = 0 \\ \text{is\_real} & \text{if } a_i = 1 \end{cases} \]

Figure 5.2: Program related to an instance of RMBF-problem \( \langle \Psi, (a_1, \ldots, a_n) \rangle \).

Let \( \tau' \) be a type which is equivalent to \( f' \). Let
\[ \Gamma'' = \text{RC}_{\tau'}(\{X_i : \alpha_i\}, P(E, Y)) \]
where \( \tau' = T_0 \cup \{ (p : ((\alpha_1, \ldots, \alpha_n, \top) \to (\alpha_1, \ldots, \alpha_n, \tau')) \} \).

By Lemma 5.24 we have
\[ BT(\Gamma'', Y) \cong E[f := f'] \]
So, since \( \tau' \equiv f' \) by the second part of (5.20), Lemma 5.26, and (5.21) we get
\[ \vdash \tau' \geq BT(\Gamma', Y) \]
so \( (\alpha_1, \ldots, \alpha_n, \top) \to (\alpha_1, \ldots, \alpha_n, \tau') \) is a type of \( p \).

Since \( \neg(f_0 \subseteq f') \) we have \( \neg(\vdash \tau_0 \leq \tau') \). It is, however, impossible, since \( \tau_0 \) is the least guarantee.

In Figure 5.2 we present the program whose type is related to the instance
\[ \langle \Psi, (a_1, \ldots, a_n) \rangle \]
of the RMBF problem.

**Lemma 5.28.** Let \( P \) be the program given in Figure 5.2. Let
\[ \Psi = (f(x_1, \ldots, x_n) = E) \]
be the RMBF formula. Let the task of a guarantee reconstruction is given as follows:

- **assumptions:** \( p : (\alpha_1, \ldots, \alpha_n, \top) \)
- **shapes of guarantees:** \( r : (\top) \)

Let \( \tau \) be a best guarantee for \( r \). Then \( \tau \in \{ \text{int, } \bot \} \) if \( f_0(a_1, \ldots, a_n) = 0 \), and \( \tau \) in \( \{ \text{real, } \top \} \) if \( f_0(a_1, \ldots, a_n) = 1 \) where \( f_0 \) is the least fixpoint of \( \Psi \).

**Proof.** Sketch. By Lemma 5.27 the type of a predicate \( p \), is a solution of \( \Psi \). The variable \( X_i \) before inferring from atom \( p(X_1, \ldots, X_n, Y ) \) in the clause of \( r \) has type reflecting the value of parameters \( a_i \) where the smaller type int corresponds to the logical value 0 and the type real corresponds to the value 1.

Suppose that \( f_0(a_1, \ldots, a_n) = 0 \). By Lemma 5.27 we get that guarantee of \( p \) is an equivalent of \( f_0 \). So, because of isomorphism between Boolean operations on \( \{0, 1\} \) and \( \cap, \cup \)-operations on types \( \text{int, real} \) the type of the variable \( Y \) is \( \text{int, or } \bot \) if \( f = \text{false} \).

And similarly if \( f_0(a_1, \ldots, a_n) = 1 \) we obtain the best possible type for \( Y \) equal to \( \text{real, or } \top \) when \( f = \text{true} \). \( \square \)
Theorem 5.4. \textit{RGP is EXPTIME-hard.}

\textit{Proof.} This theorem is a corollary to Lemma 5.20 and Lemma 5.28. \qed
Part II

The System with Pruning
Chapter 6

The Main Type

6.1 Introduction

A predicate can have many directional types. Polymorphism and subtyping describes natural relations between them. In our system these relations are described by Theorem 2.1 (page 33). Informally, Theorem 2.1 states that if a predicate \( p \) has a type \( (\tau \rightarrow \sigma) \) then \( p \) also has any type which can be obtained from \( (\tau \rightarrow \sigma) \) by substitution and weakening.

In many languages for each function, there exists the principal type, i.e. the type from which any other type of this function can be derived by a substitution (and weakening).

In case of directional types existence of principal types is rather problematic. On page 21 we have given five different types of \texttt{append} . All of them are useful, but there is no type from which all these types can be obtained by weakening and substitution. This creates a serious problem, because some predicates may have exponentially many types (with respect to the length of a predicate).

In this chapter we present a version of our type system, called \textit{System P}. To distinguish proofs in the both systems in confusing situations, we use \( \vdash_B \) and \( \vdash_p \) to denote facts provable in System B and in System P respectively. The main differences between these systems are:

- System P is weaker than System B, i.e. sometimes \( \Gamma \vdash_B \varphi \) holds, while \( \Gamma \vdash_P \varphi \) do not. Hence for some programs in System B we can infer better types than in System P.

- There are in System P some additional restriction on signatures.

- In System P we have a new method of deriving types, called \textit{pruning}.

- There is an important family of predicates which enjoy a property of having the main type, i.e. the type of a predicate from which other types of this predicate can be derived by pruning, substitution and weakening.

We believe that the assumptions we made are not too restrictive. On the other hand, existence of main types can be very useful.
6.2 System P

6.2.1 Axioms and Rules of System P

System P has the same set of equality axioms and rules as System B with the following exceptions. First, axiom (Ax14) is omitted. Second, rule \((K_3)\) is substituted by
\[
\left( K_3^P \right) \quad \Gamma, (f(t_1, \ldots, t_n) : \tau) \Rightarrow \Gamma \quad \text{if there is no signature} \quad \tau_1 \cdot \cdots \cdot \tau_n \rightarrow \tau_0 \quad \text{assigned to} \quad f \quad \text{such that} \quad \text{head}(\tau) = \text{head}(\tau_0)
\]

Axioms and rules of System P are given in appendix B.2.

The purpose of these changes is to gain a new derivation method, called \textit{pruning} which is defined in Section 6.3. The crucial fact in the new system, Pruning Theorem which states that pruning is a valid method of deriving types of predicates from other types, is not true in the system without the changes made above. We also redefine the notion of a signature. The more detailed explanations are given at the beginning of Section 6.4.

6.2.2 Signatures in the System P

The notion of a signature for System P is stronger than in the basic system. Here, we give the new definition of a signature.

**Definition 6.1.** \(\tau_1 \cdot \cdots \cdot \tau_n \rightarrow \tau\) is a \textit{signature of arity} \(n\) (in System P) if the following conditions hold:

(i) \(\tau\) is a type of the form \(F(x_1, \ldots, x_k)\), where for each \(i \in \{1, \ldots, k\}\), \(x_i\) is either \(\bot\) or a distinct type variable,

(ii) for each \(i \in \{1, \ldots, n\}\), \(\tau_i\) is either a type variable or a non-atomic type of the form \(G(\alpha_1, \ldots, \alpha_k)\) where \(\alpha_1, \ldots, \alpha_k\) are distinct variables and \(\{\alpha_1, \ldots, \alpha_k\} = \{x_1, \ldots, x_k\}\),

(iii) \(\text{var}(\tau_1, \ldots, \tau_n) = \text{var}(\tau)\).

As we can see, (i) and (ii) are not changed. Condition (ii) has been changed, and now it means that if there is a type of the form \(G(\alpha_1, \ldots, \alpha_k)\) in the left-hand side of the signature, then it is non-atomic (i.e. \(k > 0\)), and the right-hand side is of the form \(F(\beta_1, \ldots, \beta_k)\), where \(\beta_1, \ldots, \beta_k\) is a permutation of \(\alpha_1, \ldots, \alpha_k\).

**Example 6.1.** For an atomic type \(a\), the signature \(a \rightarrow a\) is correct in the basic system, but it does not fulfill condition (ii) of the definition above (\(a\) is atomic). Signatures

\[
F(\alpha) \cap G(\alpha, \beta) \Rightarrow G(\alpha, \beta) \quad \text{and} \quad G(\alpha) \Rightarrow F(\alpha, \bot)
\]

fulfill definition of signature in the basic system, but they do not fulfill condition (ii) of the definition above (in the first case \(\{\alpha\} \neq \{\alpha, \beta\}\), in the second case \(\{\alpha\} \neq \{\alpha, \bot\}\)).

6.2.3 Properties of the System

All the facts stated in Chapter 2 and Chapter 3 remain true in System P, and the proofs can be easily adopted. Similarly, algorithms and proofs described in Chapter 4 and Chapter 5 can be easily adopted to work with System P. All necessary changes in proofs and algorithms are given in Appendix A.4.
6.3 Pruning

In this section we give a definition of pruning. In System P pruning plays a very important role. It is a basis for an extended definition of a derivation, and further, for the notion of the main type. The main result of this section is Pruning Theorem.

**Definition 6.2.** For a set $A$ of type variables pruning is a function $\nabla_A$ whose domain and range are sets of types. $\nabla_A$ is defined by the following equations:

\[
\begin{align*}
\nabla_A(\tau \cap \sigma) &= \nabla_A(\tau) \cap \nabla_A(\sigma) \\
\nabla_A(\tau \cup \sigma) &= \nabla_A(\tau) \cup \nabla_A(\sigma) \\
\nabla_A(\bot) &= \bot \\
\nabla_A(\alpha) &= \begin{cases} 
\top & \text{if } \alpha \in A \\
\alpha & \text{if } \alpha \notin A 
\end{cases} \\
\nabla_A(F(\tau_1, \ldots, \tau_n)) &= \begin{cases} 
\top & \text{if } \forall i \in \{1, \ldots, n\} \vdash \nabla_A(\tau_i) = \top \\
F(\nabla_A(\tau_1), \ldots, \nabla_A(\tau_n)) & \text{otherwise}
\end{cases}
\end{align*}
\]

Pruning is a simple syntactic operation. In fact, the only non-trivial part of pruning is testing whether $\vdash \tau = \top$. But $\vdash \tau = \top$ holds if and only if one of the following cases holds: (a) $\tau = \top$, (b) $\tau = \tau_1 \cap \tau_2$, $\vdash \tau_1 = \top$, and $\vdash \tau_2 = \top$, (c) $\tau = \tau_1 \cup \tau_2$, and either $\vdash \tau_1 = \top$ or $\vdash \tau_2 = \top$. Therefore the test can be performed in linear time.

**Example 6.2.** Let $\tau = \text{prod}(\text{list}(\alpha), \text{list}(\alpha \cap \beta))$. Then we have

\[
\nabla_{\{\alpha\}}(\tau) = \text{prod}(\top, \text{list}(\beta)) \quad \text{and} \quad \nabla_{\{\alpha, \beta\}}(\tau) = \top.
\]

In the next section we shall prove that, for any set $A$ of type variables, for any types $\tau$ and $\tau'$, if $\vdash \tau = \tau'$ then $\vdash \nabla_A(\tau) = \nabla_A(\tau')$.

**Theorem 6.1 (Pruning Theorem).** Assume that a program $P$ is well typed with respect to $T$ and

\[
T' = T \cup \{(p : \nabla_A(\tau) \rightarrow \nabla_A(\sigma)) \mid (p : \tau \rightarrow \sigma) \in T, A \subseteq V_T\},
\]

then the program $P$ is also well typed with respect to $T'$.

The proof is given in the next section. Informally, the lemma states that, for any set $A$ of type variables, if the predicate $p$ has a type $\tau \rightarrow \sigma$ then it also has the type $\nabla_A(\tau) \rightarrow \nabla_A(\sigma)$.

Now, as we have a new tool that beside substitution and weakening can derive a correct type of a predicate from another correct type, we can extend the definition of a derivation.

**Definition 6.3.** Let $\tau_1 \rightarrow \sigma_1$, $\tau_2 \rightarrow \sigma_2$ be directional types.

\[
(\tau_1 \rightarrow \sigma_1) \leadsto (\tau_2 \rightarrow \sigma_2) \quad \text{iff} \quad \text{there exists a set } A \text{ of type variables such that } \nabla_A(\tau_1) \rightarrow \nabla_A(\tau_2) \leadsto (\tau_2 \rightarrow \sigma_2).
\]

\[\blacksquare\]
Definition 6.4. Let $T$ and $T'$ be directional types of a program. $T$ is $\nabla$-derivable from $T'$ if for each $(p : \tau \rightarrow \sigma) \in T$ there exists $(p : \tau' \rightarrow \sigma') \in T'$ such that $(\tau' \rightarrow \sigma') \models (\tau \rightarrow \sigma)$.

The following lemma is a consequence of Theorem 2.1 and Theorem 6.1.

Lemma 6.5. If $T'$ is $\nabla$-derivable from $T$, and a program $P$ is well typed with respect to $T$ then $P$ is also well typed with respect to $T \cup T'$.

Example 6.3. Let us analyze how pruning works for the predicate append (see Figure 2.2 on page 21). In our system one can prove that append has the type:

$$(\tau) \models (\text{list}(\alpha), \text{list}(\beta), \text{list}(\gamma)) \rightarrow (\text{list}(\alpha \cap \gamma), \text{list}(\beta \cap \gamma), \text{list}(\gamma \cap (\alpha \cup \beta)))$$

Let us denote this type by $\tau$ and prune it as follows:

$$(\tau) \models \nabla_{\{\alpha\}}(\tau) = (\top, \text{list}(\beta), \text{list}(\gamma)) \rightarrow (\text{list}(\gamma), \text{list}(\beta \cap \gamma), \text{list}(\gamma))$$

$$(\tau) \models \nabla_{\{\beta\}}(\tau) = (\text{list}(\alpha), \top, \text{list}(\gamma)) \rightarrow (\text{list}(\alpha \cap \gamma), \text{list}(\gamma), \text{list}(\gamma))$$

$$(\tau) \models \nabla_{\{\alpha, \beta\}}(\tau) = (\top, \top, \text{list}(\gamma)) \rightarrow (\text{list}(\gamma), \text{list}(\gamma), \text{list}(\gamma))$$

$$(\tau) \models \nabla_{\{\alpha, \beta, \gamma\}}(\tau) = (\top, \top, \top) \rightarrow (\top, \top, \top)$$

All the types above are types of append, and all, except of the last trivial one, were discussed in this paper. Therefore, the type (6.1) is a good candidate for the main type of append.

6.4 The Proof of the Pruning Theorem

The Pruning Theorem can be proved only for the system with modifications described in Section 6.2. Now, we explain where these modifications are used. Lemma 6.6 does not hold for the system with axiom (Ax14). Also, if we had rule $(K_3)$ instead of $(K_3^P)$, Lemma 6.18 would not be true. Finally, the new definition of signature is indispensable for the validity of Lemma 6.14, and Lemma 6.19.

Lemma 6.6. If $\vdash \tau = \sigma$ then for any set $A$ of variables, $\vdash \nabla_A(\tau) = \nabla_A(\sigma)$.

Proof. A proof of equality is a tree whose leaves are equality axioms and nodes are equality rules. We shall prove the lemma by induction on the structure of the proof of $\vdash \tau = \sigma$.

- Assume that $\vdash \tau = \tau$ was obtained using (Ax1). We use the same axiom to prove $\vdash \nabla_A(\tau) = \nabla_A(\tau)$.

- Assume that $\vdash F(\tau_1, \ldots, \tau_n) = F(\tau'_1, \ldots, \tau'_n)$ was obtained from $\vdash \tau_i = \tau'_i$, for each $i \in \{1, \ldots, n\}$, using (Ax2). By the inductive hypothesis, for each $i \in \{1, \ldots, n\}$, $\nabla_A(\tau_i) = \nabla_A(\tau'_i)$. If for each $i \in \{1, \ldots, n\}$, $\nabla_A(\tau_i) = \top$ then $\nabla_A(\tau'_i) = \top$. So, we have $\nabla_A(F(\tau_1, \ldots, \tau_n)) = \top$ and $\nabla_A(F(\tau'_1, \ldots, \tau'_n)) = \top$.

Now, suppose that, for some $k \in \{1, \ldots, n\}$, $\vdash \nabla_A(\tau_k) = \top$ does not hold. Then $\vdash \nabla_A(\tau'_k) = \top$ does not hold. So we have

$$\nabla_A(F(\tau_1, \ldots, \tau_n)) = F(\nabla_A(\tau_1), \ldots, \nabla_A(\tau_n))$$

$$= F(\nabla_A(\tau'_1), \ldots, \nabla_A(\tau'_n)) \quad \text{(by (Ax2))}$$

$$= \nabla_A(F(\tau'_1, \ldots, \tau'_n)).$$

110
For (Ax3)–(Ax12) proofs are easy, thus we omit them.

- Assume that \( \vdash F(\tau_1, \ldots, \tau_n) \cap F(\tau'_1, \ldots, \tau'_n) = F(\tau_1 \cap \tau'_1, \ldots, \tau_n \cap \tau'_n) \) was obtained using (Ax13). Let \( R = F(\tau_1, \ldots, \tau'_n) \) and \( L = F(\tau_1, \ldots, \tau_n) \). We should show that \( \vdash \nabla_A(L \cap L') = \nabla_A(R) \). We consider four cases: (1) \( \nabla_A(L) = \top \land \nabla_A(L') = \top \), (2) \( \nabla_A(L) = \top \land \nabla_A(L') \neq \top \), (3) \( \nabla_A(L) \neq \top \land \nabla_A(L') = \top \), and (4) \( \nabla_A(L) \neq \top \land \nabla_A(L') \neq \top \).

(1) Assume that \( \nabla_A(L) = \top \land \nabla_A(L') = \top \). Then \( \nabla_A(L \cap L') = \top \). By definition of pruning, \( \nabla_A(\tau_i) = \top \), and thus \( \nabla_A(\tau_i) \cap \nabla_A(\tau'_i) = \top \). Hence \( \nabla_A(R) = \top \).

(2) Assume that \( \nabla_A(L) = \top \land \nabla_A(L') \neq \top \). Then

\[
\vdash \nabla_A(L \cap L') = \nabla_A(L') = F(\nabla_A(\tau'_1), \ldots, \nabla_A(\tau'_n)).
\]

Moreover, for all \( i \in \{1, \ldots, n\} \), \( \nabla_A(\tau_i) = \top \), and thus \( \nabla_A(\tau_i \cap \tau'_i) = \nabla_A(\tau_i) \cap \nabla_A(\tau'_i) = \top \). Further, there exists \( k \in \{1, \ldots, n\} \) such that \( \nabla_A(\tau'_k) \neq \top \) and thus \( \nabla_A(\tau_k \cap \tau'_k) = \top \). Hence

\[
\vdash \nabla_A(R) = F(\nabla_A(\tau'_1), \ldots, \nabla_A(\tau'_n)) = \nabla_A(L \cap L').
\]

(3) If \( \nabla_A(L) \neq \top \land \nabla_A(L') = \top \) we show that \( \vdash \nabla_A(R) = \nabla_A(L \cap L') \) in the same way as in the previous case.

(4) Assume that \( \nabla_A(L) \neq \top \land \nabla_A(L') \neq \top \). Thus, by definition of pruning, there exists \( k \in \{1, \ldots, n\} \) such that \( \nabla_A(\tau_k) \neq \top \), thereby \( \nabla_A(\tau_k) \cap \nabla_A(\tau'_k) \neq \top \) and \( \nabla_A(\tau_k \cap \tau'_k) \neq \top \). Hence

\[
\vdash \nabla_A(L \cap L') = \nabla_A(L) \cap \nabla_A(L')
\]

\[
= F(\nabla_A(\tau_1), \ldots, \nabla_A(\tau_n)) \cap F(\nabla_A(\tau'_1), \ldots, \nabla_A(\tau'_n))
\]

\[
= F(\nabla_A(\tau_1 \cap \tau'_1), \ldots, \nabla_A(\tau_n \cap \tau'_n)) \quad \text{by (Ax13)}
\]

\[
= F(\nabla_A(\tau_1), \ldots, \nabla_A(\tau_n) \cap \nabla_A(\tau'_n)) = \nabla_A(R).
\]

- We do not consider (Ax14) because System P does not contain it.

- Assume that \( \vdash \tau \cup \sigma = \sigma \cup \tau \) was obtained using (Ax15). Since \( \tau \), \( \sigma \) and \( \sigma \cup \tau \) are atomic, we have \( \nabla_A(\tau \cup \sigma) = \tau \cup \sigma \), and \( \nabla_A(\sigma \cup \tau) = \sigma \cup \tau \), hence the lemma holds obviously.

- For (Ax16) the proof is similar. \( \square \)

**Corollary 6.7.** For any set \( A \) of type variables, if \( \vdash \tau \leq \sigma \) then \( \vdash \nabla_A(\tau) \leq \nabla_A(\sigma) \).

**Proof.** \( \vdash \tau \leq \sigma \) means that \( \vdash \tau \cap \sigma = \tau \). We claim that \( \vdash \nabla_A(\tau) \leq \nabla_A(\sigma) \) i.e. \( \vdash \nabla_A(\tau) \cap \nabla_A(\sigma) = \nabla_A(\tau) \). In fact

\[
\nabla_A(\tau) \cap \nabla_A(\sigma) = \nabla_A(\tau \cap \sigma)
\]

by definition of pruning

\[
= \nabla_A(\tau) \quad \text{by Lemma 6.6}
\]

\( \square \)

**Lemma 6.8.** For any type \( \tau \), and any set \( A \) of type variables

\[
\nabla_A(\tau) = \nabla_A(\nabla_A(\tau)).
\]

**Proof.** The proof proceeds by induction on \( \tau \). Let \( L \) and \( R \) are the left and the right-hand side of equation (6.7) respectively.

\[
\nabla_A(\tau) = \nabla_A(\nabla_A(\tau)).
\]
Suppose that \( \tau = \alpha \). If \( \alpha \in A \) then both \( L \) and \( R \) are \( \top \), if \( \alpha \notin A \) then both \( L \) and \( R \) are equal to \( \alpha \).

If \( \tau \) is an atomic type then both \( L \) and \( R \) are equal to \( \tau \).

Suppose that \( \tau = F(\tau_1, \ldots, \tau_n) \). If \( L = \nabla_A(\tau) = \top \), then obviously \( R = \nabla_A(\top) = \top \).

If \( L = \nabla_A(\tau) \neq \top \), then, by the definition of pruning,
\[
L = F(\nabla_A(\tau_1), \ldots, \nabla_A(\tau_n)),
\]
and for some \( k \in \{1, \ldots, n\} \), \( \nabla_A(\tau_k) \neq \top \). By the inductive hypothesis, \( \nabla_A(\tau_i) = \nabla_A(\nabla_A(\tau_i)) \) for all \( i \in \{1, \ldots, n\} \), thereby \( \nabla_A(\nabla_A(\tau_k)) \neq \top \). Hence, by the definition of pruning,
\[
R = \nabla_A(L) = F(\nabla_A(\nabla_A(\tau_1)), \ldots, \nabla_A(\nabla_A(\tau_n))) = F(\nabla_A(\tau_1), \ldots, \nabla_A(\tau_n)) = L.
\]

Suppose that \( \tau = \tau_1 \cap \tau_2 \). Then
\[
L = \nabla_A(\tau_1 \cap \tau_2) = \nabla_A(\tau_1) \cap \nabla_A(\tau_2) = \nabla_A(\nabla_A(\tau_1) \cap \nabla_A(\tau_2)) = \nabla_A(\nabla_A(\tau_1 \cap \tau_2)) = R.
\]

If \( \tau = \tau_1 \cup \tau_2 \) then the proof goes in the same way as above. \( \square \)

**Definition 6.9.** We define relation \( \prec \) on pairs of types and on pairs of environments:
\[
\tau \prec \tau' \quad \text{iff} \quad \vdash \tau \leq \tau' \leq \nabla_A(\tau)
\]
\[
\Gamma \prec \Gamma' \quad \text{iff} \quad \Gamma \leq \Gamma' \land (\Gamma \vdash t : \tau \quad \Rightarrow \quad \Gamma' \vdash t : \nabla_A(\tau)).
\]

**Lemma 6.10.** For any type \( \tau \), and any \( A \subseteq V \), we have \( \vdash \tau \leq \nabla_A(\tau) \).

**Proof.** We proceed by induction on the structure of \( \tau \).

- If \( \tau = \top \) or \( \tau = \bot \) then \( \nabla_A(\tau) = \tau \), and \( \vdash \tau \leq \nabla_A(\tau) \) obviously holds.

- Assume that \( \tau = \alpha \) (\( \alpha \in V \)). If \( \alpha \in A \) then \( \nabla_A(\alpha) = \top \), so \( \vdash \tau \leq \nabla_A(\tau) \). If \( \alpha \notin A \) then \( \nabla_A(\alpha) = \alpha \), and obviously \( \vdash \tau \leq \nabla_A(\tau) \).

- Let \( \tau = F(\tau_1, \ldots, \tau_n) \). If \( \nabla_A(\tau) = \top \) then the thesis obviously holds. So, assume that \( \nabla_A(\tau) \neq \top \). By the inductive hypothesis, \( \tau_i \leq \nabla_A(\tau_i) \), for \( i \in \{1, \ldots, n\} \).

\[
\vdash F(\tau_1, \ldots, \tau_n) \leq F(\nabla_A(\tau_1), \ldots, \nabla_A(\tau_n)) \leq \nabla_A(F(\tau_1, \ldots, \tau_n)) \quad \text{(by Lemma 2.22)}
\]

- Let \( \tau = \tau_1 \cap \tau_2 \). By the inductive hypothesis \( \tau_1 \leq \nabla_A(\tau_1) \), and \( \tau_2 \leq \nabla_A(\tau_2) \). Hence
\[
\vdash \tau_1 \cap \tau_2 \leq \nabla_A(\tau_1) \cap \nabla_A(\tau_2) = \nabla_A(\tau_1 \cap \tau_2).
\]

- If \( \tau = \tau_1 \cup \tau_2 \) the proof is similar. \( \square \)
\textbf{Corollary 6.11.} Since, for any set $A$ of variables, and any type $\tau$, $\tau \leq \nabla_A(\tau)$ and 
$\nabla_A(\tau) \leq \nabla_A(\tau)$, we have

$$\tau \overset{\sim}{\leq} \nabla_A(\tau).$$

\textbf{Definition 6.12.} Let $\theta$ be a type substitution, let $A$ be a finite set of type variables.
We define a type substitution $\nabla_A(\theta)$ as follows.

$$(\nabla_A(\theta))(\alpha) = \nabla_A(\theta(\alpha)).$$

\[\Box\]

\textbf{Corollary 6.13.} Note that since, for each type $\tau$, $\tau \leq \nabla_A(\tau)$ holds, we have

$$\theta \leq \nabla_A(\theta)$$

(see Definition 2.25). Moreover, by Lemma 2.26, for each type $\tau$,

$$\vdash \theta(\tau) \leq (\nabla_A(\theta)(\tau)).$$

\textbf{Lemma 6.14.} The following statements are equivalent:

1. $\Gamma \overset{\sim}{\leq} \Gamma'$,

2. for each variable $X$, we have $\Gamma(X) \overset{\sim}{\leq} \Gamma'(X)$.

\textbf{Proof.}

1. $\Rightarrow$ 2. Suppose that $\Gamma \overset{\sim}{\leq} \Gamma'$. It means that $\Gamma \leq \Gamma'$, and if $\Gamma \vdash t : \tau$ then 
$\Gamma' \vdash t : \nabla_A(\tau)$. Let $X$ be a program variable, $\tau = \Gamma(X)$, and $\tau' = \Gamma'(X)$. Since $\Gamma \leq \Gamma'$, by Lemma 2.34, we have $\vdash \tau \leq \tau'$. To finish the proof it remains to show

that $\vdash \tau' \leq \nabla_A(\tau)$.

$\Gamma \vdash X : \tau$, thus $\Gamma' \vdash X : \nabla_A(\tau)$. By Lemma 2.29 (b), we have $\vdash \tau' \leq \nabla_A(\tau)$.

2. $\Rightarrow$ 1. Suppose that 2 holds. Since, for each variable $X$, $\vdash \Gamma(X) \leq \Gamma'(X)$, 
by Lemma 2.34, we have $\Gamma \leq \Gamma'$. Now, it suffices to show that if $\Gamma \vdash t : \tau$ then $\vdash t : \nabla_A(\tau)$. We proceed by induction on the structure of the proof of $\Gamma \vdash t : \tau$.

- If $\Gamma \vdash t : \top$ is obtained using $(T_1)$ then, since $\nabla_A(\top) = \top$, by the same rule, we have $\Gamma' \vdash t : \nabla_A(\top)$.

- Assume that false $\in \Gamma$. Assume also that $\Gamma \vdash t : \bot$ was obtained using $(T_2)$. We claim that $\Gamma' \vdash t : \bot$.

$\Gamma'$ is, by the definition of an environment, a finite set of formulas, thus it contains only finitely many variables. So, suppose that $Y$ is a variable not occurring in $\Gamma'$. From (2) we have $\Gamma(Y) \overset{\sim}{\leq} \Gamma'(Y)$, hence

$$\Gamma'(Y) \leq \nabla_A(\Gamma(Y)) = \nabla_A(\bot) = \bot.$$ 

It implies that $\Gamma'(Y) = \bot$. Since $Y$ does not occur in $\Gamma'$, we have false $\in \Gamma'$ and thus $\Gamma' \vdash t : \bot$.

- Assume that $\Gamma \vdash X : \tau$, where $(X : \tau) \in \Gamma$, was obtained using $(T_3)$.

Since $(X : \tau) \in \Gamma$, we can use (2) to obtain $\tau \overset{\sim}{\leq} \tau'$ where $\tau' = \Gamma'(X)$, thus $\vdash \tau' \leq \nabla_A(\tau)$. Since we have $\Gamma' \vdash \tau'$, we can use (2) to obtain $\Gamma' \vdash \nabla_A(\tau)$.

- Assume that $\Gamma \vdash t : \tau$ was obtained from $\vdash \tau' \leq \tau$, and $\Gamma \vdash t : \tau'$ using $(T_3)$. By 
the inductive hypothesis, $\Gamma' \vdash t : \nabla_A(\tau')$. By Corollary 6.7, $\vdash \nabla_A(\tau') \leq \nabla_A(\tau)$, hence, by (2), we have $\Gamma' \vdash t : \nabla_A(\tau)$.

113
Assume that $\Gamma \vdash t : \tau_i \cap \cdots \cap \tau_n$ was obtained from $\Gamma \vdash t : \tau_i$ for all $i \in \{1, \ldots, n\}$, using $(T_5)$. By the inductive hypothesis $\Gamma' \vdash t : \nabla_A(\tau_i)$ for all $i \in \{1, \ldots, n\}$. We have $\nabla_A(\tau_i) \cap \cdots \cap \nabla_A(\tau_n) = \nabla_A(\tau)$, so we can use $(T_5)$ to obtain $\Gamma' \vdash t : \nabla_A(\tau)$.

Assume that $f$ has signature $\tau_1 \ast \cdots \ast \tau_n \Rightarrow \tau$. Assume also that $\Gamma \vdash f(t_1, \ldots, t_n) : \theta(\tau)$ was obtained from $\Gamma \vdash t_i : \theta(\tau_i)$, for $i \in \{1, \ldots, n\}$, using $(T_6)$. Suppose that $\nabla_A(\theta(\tau)) = \top$. Then obviously, by $(T_1)$, $\Gamma' = \top$.

Now, suppose that $\nabla_A(\theta(\tau)) \neq \top$. Then, it is easy to check that, from the definition of signature in System $P$ it follows that $\nabla_A(\theta(\tau)) = (\nabla_A(\theta))(\tau)$, and $\nabla_A(\theta(\tau_i)) = (\nabla_A(\theta))(\tau_i)$. For $i \in \{1, \ldots, n\}$, we have $\Gamma \vdash t_i : \theta(\tau_i)$, thus, by the inductive hypothesis,

$$\Gamma' \vdash t_i : \nabla_A(\theta(\tau_i))$$

which is equivalent to $\Gamma' \vdash t_i : (\nabla_A(\theta))(\tau_i)$. This implies, by $(T_6)$, that $\Gamma' \vdash t : (\nabla_A(\theta))(\tau_i)$ which is equivalent to $\Gamma' \vdash t : \nabla_A(\theta(\tau_i))$.

\begin{lemma} If $\Gamma \vdash \tau'$ and $\Gamma' \vdash \tau''$ then $\Gamma \vdash \tau' \Rightarrow \tau''$. Similarly, if $\tau \vdash \tau'$ and $\tau' \vdash \tau''$ then $\tau \vdash \tau''$.
\end{lemma}

\begin{proof}
1. Suppose that $\Gamma \vdash \tau'$. By the definition of $\vdash$, we have that $\Gamma \vdash t : \nabla_A(\tau)$, and $\Gamma' \vdash t : \nabla_A(\nabla_A(\tau)) = \nabla_A(\tau)$. Now, it remains to show that $\Gamma \leq \Gamma''$ which is obvious, since $\Gamma \leq \Gamma'$ and $\Gamma' \leq \Gamma''$.

2. Suppose now, that $\tau \vdash \tau'$ and $\tau' \vdash \tau''$. $\vdash \tau \leq \tau' \Rightarrow \vdash \tau \leq \tau''$ trivially holds since $\vdash \tau \leq \tau'$ and $\vdash \tau' \leq \tau''$.

Now, we want to show that $\vdash \tau'' \leq \nabla_A(\tau)$. From the fact $\vdash \tau \leq \tau'$ and from the definition of $\vdash$ it follows that $\vdash \tau \leq \tau'$ and $\vdash \tau' \leq \nabla_A(\tau)$. Applying Corollary 6.7 we obtain $\vdash \nabla_A(\tau) \leq \nabla_A(\tau')$ and $\vdash \nabla_A(\tau') \leq \nabla_A(\nabla_A(\tau)) = \nabla_A(\tau)$. This gives $\vdash \nabla_A(\tau) = \nabla_A(\tau')$. Now, $\vdash \tau' \leq \tau''$ implies $\vdash \tau'' \leq \nabla_A(\tau)$, hence $\vdash \tau'' \leq \nabla_A(\tau)$.
\end{proof}

\begin{lemma} Assume that $\Gamma_i \vdash \Gamma_i'$, for $i \in \{1, \ldots, n\}$. Then

$$\Gamma_1 \cap \cdots \cap \Gamma_n \vdash \Gamma'_1 \cap \cdots \cap \Gamma'_n$$

\end{lemma}

\begin{proof}
By Lemma 6.14 it suffices to show that, for each variable $X$,

$$(\Gamma_1 \cap \cdots \cap \Gamma_n)(X) \vdash (\Gamma'_1 \cap \cdots \cap \Gamma'_n)(X).$$

Let $\tau_i = \Gamma_i(X)$ and $\tau'_i = \Gamma'_i(X)$. Let $\tau = \tau_1 \cap \cdots \cap \tau_n$, and $\tau' = \tau'_1 \cap \cdots \tau'_n$. Since $\Gamma_i \vdash \Gamma_i'$, we have

$$\vdash \tau_i \leq \tau'_i \quad \text{and} \quad \vdash \tau'_i \leq \nabla_A(\tau_i) \quad \text{for} \quad i \in \{1, \ldots, n\}.$$ 

Hence

$$\vdash \tau \cap \tau' = (\tau_1 \cap \cdots \cap \tau_n) \cap (\tau'_1 \cap \cdots \cap \tau'_n)$$

$$= (\tau_1 \cap \tau'_1) \cap \cdots \cap (\tau_n \cap \tau'_n) = \tau_1 \cap \cdots \cap \tau_n = \tau.$$ 

It means that $\tau \leq \tau'$. On the other hand

$$\vdash \nabla_A(\tau) \cap \tau' = (\nabla_A(\tau_1) \cap \cdots \cap \nabla_A(\tau_n)) \cap (\tau'_1 \cap \cdots \cap \tau'_n)$$

$$= (\nabla_A(\tau_1) \cap \tau'_1) \cap \cdots \cap (\nabla_A(\tau_n) \cap \tau'_n) = \tau'_1 \cap \cdots \tau'_n = \tau'.$$

114
It means that \( \tau' \leq \nabla_A(\tau) \). Hence \( \tau \xrightarrow{A} \tau' \). By Lemma 2.43

\[
(\Gamma_1 \cap \cdots \cap \Gamma_n)(X) = \Gamma_1(X) \cup \cdots \cup \Gamma_n(X) = \tau
\]

and

\[
(\Gamma'_1 \cap \cdots \cap \Gamma'_n)(X) = \Gamma'_1(X) \cup \cdots \cup \Gamma'_n(X) = \tau'.
\]

Hence (6.8) holds.

\( \square \)

**Corollary 6.17.** If \( \Gamma_i \xrightarrow{A} \Gamma \), for \( i \in \{1, \ldots, n\} \), then \( \Gamma_1 \cap \cdots \cap \Gamma_n \xrightarrow{A} \Gamma \).

In fact, if \( \Gamma'_1 = \cdots = \Gamma'_n = \Gamma \) then, by Lemma 6.16 and Lemma 2.44, we get

\[
\Gamma_1 \cap \cdots \cap \Gamma_n \sim \Gamma'_1 \cap \cdots \cap \Gamma'_n \sim \Gamma \cap \cdots \cap \Gamma \sim \Gamma.
\]

**Lemma 6.18.** If \( \Gamma, (t : \tau) \vdash \top \) and \( \Gamma, (t : \tau) \Rightarrow \Gamma' \), then \( \Gamma' \xrightarrow{A} \Gamma \).

**Proof.** We proceed by induction on the structure of the proof of \( \Gamma, (t : \tau) \Rightarrow \Gamma' \). Proofs in the case of \( (K_1), (K_2) \) and \( (K_7) \) are very similar, thus we present only one of them — \( (K_7) \). For \( (K_3), (K_5) \) and \( (K_6) \) we use the fact that input and output environments are the same. The case of \( (K_4) \) does not apply since \( \vdash A(\bot) = \top \) does not hold.

\( (K_7) \) In this case

\[
\Gamma, (t : \tau_1 \cap \cdots \cap \tau_n) \Rightarrow \Gamma_1 \cap \cdots \cap \Gamma_n
\]

is proved using \( \Gamma_i, (t : \tau_i) \Rightarrow \Gamma_i \), for \( i \in \{1, \ldots, n\} \). We have \( \vdash A(\tau_1 \cap \cdots \cap \tau_n) = \top \) thus \( \vdash A(\tau_i) = \top \), for each \( i \in \{1, \ldots, n\} \), hence, by induction, \( \Gamma_i \xrightarrow{A} \Gamma \), for each \( i \in \{1, \ldots, n\} \). We use Corollary 6.17 to obtain \( \Gamma_1 \cap \cdots \cap \Gamma_n \xrightarrow{A} \Gamma \).

\( (K_8) \) In this case

\[
\Gamma, (t : \tau_1 \cup \cdots \cup \tau_n) \Rightarrow \Gamma_1 \cup \cdots \cup \Gamma_n
\]

is proved using \( \Gamma_i, (t : \tau_i) \Rightarrow \Gamma_i \), for \( i \in \{1, \ldots, n\} \). We have \( \vdash A(\tau_1 \cup \cdots \cup \tau_n) = \top \), thus \( \vdash A(\tau_k) = \top \), for some \( k \in \{1, \ldots, n\} \), hence, by induction, \( \Gamma_k \xrightarrow{A} \Gamma \). We also have \( \Gamma_k \leq \Gamma_1 \cup \cdots \cup \Gamma_n \). Now, suppose that \( \Gamma_1 \cup \cdots \cup \Gamma_n \vdash t : \tau \), for a term \( t \) and a type \( \tau \). Hence \( \Gamma_k \vdash t : \tau \). Since \( \Gamma_k \xrightarrow{A} \Gamma \), we have \( \Gamma \vdash t : \nabla_A(\tau) \).

This proves that \( \Gamma_1 \cup \cdots \cup \Gamma_n \xrightarrow{A} \Gamma \).

\( (K_9) \) In this case we have \( \Gamma_i(X : \tau) \Rightarrow \Gamma' \), where \( \Gamma' = \Gamma \cap \{X : \tau\} \).

By Lemma 6.14, it suffices to show that

for each \( Y \), \( \Gamma' (Y) \xrightarrow{A} \Gamma (Y) \).

Suppose that \( Y \neq X \). Then \( \Gamma' (Y) = \Gamma (Y) \), and hence \( \Gamma' (Y) \xrightarrow{A} \Gamma (Y) \).

Now, suppose that \( Y = X \). Then we have \( \Gamma' (X) = \tau \cap \Gamma (X) \). Thus \( \Gamma' (X) \leq \Gamma (X) \). We have also

\[
\vdash \nabla_A(\tau \cap \Gamma (X)) = \nabla_A(\tau) \cap \nabla_A(\Gamma (X)) = \nabla_A(\tau) \cap \nabla_A(\Gamma (X)).
\]

By Corollary 6.11 we have \( \Gamma (X) \leq \nabla_A(\Gamma (X)) = \nabla_A(\Gamma' (X)) \). It means that \( \Gamma' (X) \xrightarrow{A} \Gamma (X) \).

\( \square \)

115
Lemma 6.19 (Consequence Pruning Lemma). If
\[ \Gamma, (t : \tau) \Rightarrow \Gamma' \] (\#) and \( \Gamma \overset{i}{\Rightarrow} \hat{\Gamma} \)
then there exist \( \hat{\Gamma}' \) such that
\[ \Gamma' \overset{i}{\Rightarrow} \hat{\Gamma}' \] and \( \hat{\Gamma}, (t : \nabla_A(\tau)) \Rightarrow \hat{\Gamma}' \) (\#\#)

Proof. We shall construct \( \hat{\Gamma}' \) as well as the proof of (\#\#) on the basis of the proof of (\#). This construction will be done by recursion on the structure of the proof of (\#). Elements of the proof of (\#\#) will be marked by hat.

- Assume that \( \Gamma, (t_1 : \tau_1, \ldots, t_n : \tau_n) \Rightarrow \Gamma' \), where \( \Gamma' = \Gamma_1 \cap \cdots \cap \Gamma_n \), was obtained from \( \Gamma, (t_i : \tau_i) \Rightarrow \Gamma_i \), for \( i \in \{1, \ldots, n\} \), using \((K_1)\).

By the inductive hypothesis, for each \( i \in \{1, \ldots, n\} \), there exists \( \hat{\Gamma}_i \) such that \( \Gamma_i \overset{i}{\Rightarrow} \hat{\Gamma}_i \) and \( \hat{\Gamma}, (t_i : \nabla_A(\tau_i)) \Rightarrow \hat{\Gamma}_i \). Let \( \hat{\Gamma}' = \hat{\Gamma}_1 \cap \cdots \cap \hat{\Gamma}_n \). We can use rule \((K_1)\) to prove
\[ \hat{\Gamma}, (t_1 : \nabla_A(\tau_1), \ldots, t_n : \nabla_A(\tau_n)) \Rightarrow \hat{\Gamma}' \].

By Lemma 6.16, \( \Gamma' \overset{i}{\Rightarrow} \hat{\Gamma}' \).

- Assume that \( \Gamma, (f(t_1, \ldots, t_n) : \sigma) \Rightarrow \Gamma' \) was obtained from \( \Gamma, (t_i : \theta(\tau_i)) \Rightarrow \Gamma_i \), for \( i \in \{1, \ldots, n\} \), using \((K_2)\), where
\[
(6.9) \quad \Gamma' = \Gamma_1 \cap \cdots \cap \Gamma_n.
\]
\[ \sigma = \theta(\nabla(\tau)) \]
\[ f : \tau_1 \ast \cdots \ast \tau_n \to \tau, \]

Consider two cases: (1) \( \nabla_A(\sigma) \neq \top \), and (2) \( \nabla_A(\sigma) = \top \).

(1) \( \nabla_A(\sigma) \neq \top \).

To construct the proof of (\#\#) we use the signature of \( f \) given in (6.9). According to the features of signatures (Definition 6.1), \( \nabla(\tau) \) has the form \( F(\alpha_1, \ldots, \alpha_k) \). Since \( \theta(\nabla(\tau)) = \sigma \), \( \sigma \) has the form \( F(\sigma_1, \ldots, \sigma_k) \), and \( \theta = \{ \alpha_i / \sigma_i \}_{i=1}^k \). Since \( \nabla_A(\sigma) \neq \top \), thus \( \nabla_A(\sigma) = F(\nabla_A(\sigma_1), \ldots, \nabla_A(\sigma_k)) \).

Let \( \tilde{\theta} = \{ \alpha_i / \nabla_A(\sigma_i) \}_{i=1}^k \). It is easy to see that \( \nabla_A(\sigma) = \tilde{\theta}(\nabla(\tau)) \), and by Definition 6.1, (ii), \( \tilde{\theta}(\tau_i) = \nabla_A(\theta(\tau_i)) \). Hence, by the inductive hypothesis, for each \( i \in \{1, \ldots, n\} \), there exists \( \hat{\Gamma}_i \) such that \( \Gamma_i \overset{i}{\Rightarrow} \hat{\Gamma}_i \), and
\[ \hat{\Gamma}_i, (t_i : \tilde{\theta}(\tau_i)) \Rightarrow \hat{\Gamma}_i. \]

Thus we can use \((K_2)\) to construct the proof of
\[ \hat{\Gamma}, (f(t_1, \ldots, t_n) : \nabla_A(\sigma)) \Rightarrow \hat{\Gamma}' \]
where \( \hat{\Gamma}' = \hat{\Gamma}_1 \cap \cdots \cap \hat{\Gamma}_n \). Since \( \Gamma_i \overset{i}{\Rightarrow} \hat{\Gamma}_i \), by Lemma 6.16, we have \( \Gamma' \overset{i}{\Rightarrow} \hat{\Gamma}' \).

(2) \( \nabla_A(\sigma) = \top \).

It is easy to check that, from Definition 6.1 (ii) it follows that \( \nabla_A(\theta(\tau_i)) = \top \), for \( i \in \{1, \ldots, n\} \). Hence, for each \( i \in \{1, \ldots, n\} \), by Lemma 6.18, we have \( \Gamma_i \overset{i}{\Rightarrow} \hat{\Gamma} \). Since \( \Gamma \overset{i}{\Rightarrow} \hat{\Gamma} \), we can apply Lemma 6.15, to obtain \( \Gamma \overset{i}{\Rightarrow} \hat{\Gamma} \). By corollary 6.17,
\[ \Gamma' = \Gamma_1 \cap \cdots \cap \Gamma_n \overset{i}{\Rightarrow} \hat{\Gamma}. \]

We construct proof of (\#\#) using rule \((K_3)\), so \( \hat{\Gamma}' = \hat{\Gamma} \), and thus \( \Gamma' \overset{i}{\Rightarrow} \hat{\Gamma}' \).
• Assume that $\Gamma, (f(t_1, \ldots, t_n) : \tau) \Rightarrow \Gamma$ was obtained using $(K_3^P)$, with assumption that $f$ does not have a signature with the same head as $\tau$.

Suppose that $\nabla_A(\tau) = \top$. Then we use rule $(K_5)$ to obtain

$$\widehat{\Gamma}, (f(t_1, \ldots, t_n) : \top) \Rightarrow \widehat{\Gamma}.$$

By the assumption of the lemma, $\Gamma \sim^\delta \widehat{\Gamma}$.

Now, suppose that $\nabla_A(\tau) \neq \top$. Then $\nabla_A(\tau) = F(\nabla_A(\tau_1), \ldots, \nabla_A(\tau_k))$, where $\tau = F(\tau_1, \ldots, \tau_k)$. Since $\tau$ and $\nabla_A(\tau)$ have the same head, we can use rule $(K_3^P)$ to obtain $\widehat{\Gamma}, (f(t_1, \ldots, t_n) : \tau) \Rightarrow \widehat{\Gamma}$. By assumption of the lemma, $\Gamma \sim^\delta \widehat{\Gamma}$.

• Assume that $\Gamma, (t : \bot) \Rightarrow \{\text{false}\}$ was obtained using $(K_4)$. Since $C_A(\bot) = \bot$, we use the same rule to obtain $\widehat{\Gamma}, (t : \nabla_A(\bot)) \Rightarrow \{\text{false}\}$. Obviously $\{\text{false}\} \sim^\delta \{\text{false}\}$.

• Assume that $\Gamma, (t : \top) \Rightarrow \Gamma$ was obtained using $(K_5)$. Since $C_A(\top) = \top$, we use the same rule to obtain $\widehat{\Gamma}, (t : \nabla_A(\top)) \Rightarrow \widehat{\Gamma}$. By assumption of the lemma, $\Gamma \sim^\delta \widehat{\Gamma}$.

• Assume that $\Gamma, (f(t_1, \ldots, t_n) : \alpha) = \Gamma$ was obtained using $(K_6)$. If $\alpha \in A$ we use $(K_5)$ to construct the proof of $(*)$, otherwise we use rule $(K_6)$. In both cases $\widehat{\Gamma}' = \widehat{\Gamma}$, and thus $\Gamma \sim^\delta \widehat{\Gamma}'$.

• Assume that $\Gamma, (t_1 \cap \cdots \cap t_n) \Rightarrow \Gamma'$, where $\Gamma' = \Gamma_1 \cap \cdots \cap \Gamma_n$, was obtained from $\Gamma, (t_i : \tau_i) \Rightarrow \Gamma_i$, for $i \in \{1, \ldots, n\}$, using $(K_7)$. By the inductive hypothesis, there exists $\widehat{\Gamma}_i$ such that $\Gamma_i \sim^\delta \widehat{\Gamma}_i$, and $\widehat{\Gamma}_i, (t_i : \nabla_A(\tau_i)) \Rightarrow \widehat{\Gamma}_i$. Let $\widehat{\Gamma}' = \widehat{\Gamma}_1 \cap \cdots \cap \widehat{\Gamma}_n$. We can use rule $(K_7)$ to prove

$$\widehat{\Gamma}_i, (t : \nabla_A(\tau_1) \cap \cdots \cap \nabla_A(\tau_n)) \Rightarrow \widehat{\Gamma}'$$

By the definition of pruning, $\nabla_A(\tau_1 \cap \cdots \cap \tau_n) = \nabla_A(\tau_1) \cap \cdots \cap \nabla_A(\tau_n)$. So we have

$$\widehat{\Gamma}_i, (t : \nabla_A(\tau_1 \cap \cdots \cap \tau_n)) \Rightarrow \widehat{\Gamma}'$$

By Lemma 6.16, $\Gamma' \sim^\delta \widehat{\Gamma}'$.

• Assume that $\Gamma, (t : \sigma) \Rightarrow \Gamma'$, where $\sigma = \tau_1 \cup \cdots \cup \tau_n$, and $\Gamma' = \Gamma_1 \cup \cdots \cup \Gamma_n$, was obtained from $\Gamma, (t_i : \tau_i) \Rightarrow \Gamma_i$, for $i \in \{1, \ldots, n\}$, using $(K_8)$.

By the definition of pruning, $\nabla_A(\sigma) = \nabla_A(\tau_1) \cup \cdots \cup \nabla_A(\tau_n)$. By the inductive hypothesis there exists environments $\widehat{\Gamma}_i$, for $i \in \{1, \ldots, n\}$, such that $\Gamma_i \sim^\delta \widehat{\Gamma}_i$, and $\widehat{\Gamma}_i, (t_i : \nabla_A(\tau_i)) \Rightarrow \widehat{\Gamma}_i$. Let $\widehat{\Gamma}' = \widehat{\Gamma}_1 \cup \cdots \cup \widehat{\Gamma}_n$. We use the same rule to obtain $\widehat{\Gamma}_i, (t : \nabla_A(\sigma)) \Rightarrow \widehat{\Gamma}'$. It remains to show that $\Gamma' \sim^\delta \widehat{\Gamma}'$.

For each $i \in \{1, \ldots, n\}$, for any program variable $X$, let $\delta_i = \Gamma_i(X)$ and $\delta'_i = \widehat{\Gamma}_i(X)$.

By Lemma 6.14 and the definition of $\sim$, we have $\delta_i \leq \delta'_i$, and $\delta'_i \leq \nabla_A(\delta_i)$. Thus $\delta_i \cup \cdots \cup \delta_n \leq \delta'_i \cup \cdots \cup \delta'_n$. Moreover, by Lemma 2.20, $\delta_i \cup \nabla_A(\delta_i) = \nabla_A(\delta_i)$. Now,

$$(\delta'_1 \cup \cdots \cup \delta'_n) \cup (\nabla_A(\delta_1) \cup \cdots \cup \nabla_A(\delta_1)) = (\delta'_1 \cup \nabla_A(\delta_1)) \cup \cdots \cup (\delta'_n \cup \nabla_A(\delta_n)) = \nabla_A(\delta_1) \cup \cdots \cup \nabla_A(\delta_n).$$

Thus

$$\delta'_1 \cup \cdots \cup \delta'_n \leq \nabla_A(\delta_1) \cup \cdots \cup \nabla_A(\delta_n) = \nabla_A(\delta_1 \cup \cdots \cup \delta_n),$$

117
and hence, by the definition of $\sim$, 

$$\delta_1 \cup \cdots \cup \delta_n \sim \tilde{\delta_1} \cup \cdots \cup \tilde{\delta}_n,$$

which, by Lemma 2.43, is equivalent to $\Gamma'(X) \sim \tilde{\Gamma}'(X)$. Since we have showed that for any variable $X$, we can use Lemma 6.14 to obtain $\Gamma' \sim \tilde{\Gamma}'$.

- Assume that $\Gamma, (X : \tau) \Rightarrow \Gamma'$, where $\Gamma' = \Gamma \cap \{X : \tau\}$, was obtained using $(K_9)$. By $(K_9)$ we have $\tilde{\Gamma}, (X : \nabla_A(\tau)) \Rightarrow \tilde{\Gamma}'$ where $\tilde{\Gamma}' = \tilde{\Gamma} \cap \{X : \nabla_A(\tau)\}$. We want to show that $\Gamma' \sim \tilde{\Gamma}'$ which, by Lemma 6.14, is equivalent to showing that, for any program variable $Y$,

$$(6.10) \quad \Gamma'(Y) \sim \tilde{\Gamma}'(Y).$$

If $Y \neq X$ then $\Gamma'(Y) = \Gamma(Y)$ and $\tilde{\Gamma}'(Y) = \tilde{\Gamma}(Y)$. Since we have assumed that $\Gamma \sim \tilde{\Gamma}$, we have $\Gamma(Y) \sim \tilde{\Gamma}(Y)$ and (6.10) holds.

If $Y = X$, we have $\Gamma'(X) = \tau \cap \Gamma(X)$, and $\tilde{\Gamma}'(X) = \nabla_A(\tau) \cap \tilde{\Gamma}(X)$. Since we have assumed that $\Gamma \sim \tilde{\Gamma}$, we have $\Gamma(X) \sim \tilde{\Gamma}(X)$ which implies $\vdash \tilde{\Gamma}(X) \leq \nabla_A(\Gamma(X))$. Hence we have

$$\vdash \tilde{\Gamma}'(X) = \nabla_A(\tau) \cap \tilde{\Gamma}(X) \leq \nabla_A(\tau) \cap \nabla_A(\Gamma(X)) = \nabla_A(\tau \cap \Gamma(X)) = \nabla_A(\Gamma').$$

Thus $\tilde{\Gamma}'(X) \leq \nabla_A(\Gamma')$. On the other hand, $\Gamma(X) \sim \tilde{\Gamma}(X)$ implies that $\Gamma(X) \leq \tilde{\Gamma}(X)$, and thus, by Corollary 6.11,

$$\vdash \Gamma'(X) = \tau \cap \Gamma(X) \leq \nabla_A(\tau) \cap \tilde{\Gamma}(X) = \tilde{\Gamma}'(X).$$

This gives (6.10).

Lemma 6.20. If $\theta$ is a type substitution, $\tau$ is a type, and

$$B = \{\alpha \in \text{dom}(\theta) \mid \nabla_A(\theta(\alpha)) = \top\}$$

then

$$\vdash L = \nabla_A(\theta(\tau)) = (\nabla_A(\theta))(\nabla_B(\tau)) = R.$$
6.4. The Proof of the Pruning Theorem

c) \(\tau = F(\tau_1, \ldots, \tau_n)\).
Suppose that \(L = \top\). So
\[
L = \nabla_A(\theta(F(\tau_1, \ldots, \tau_n))) = \nabla_A(F(\theta \tau_1, \ldots, \theta \tau_n)) = \top.
\]
Thus, for each \(i \in \{1, \ldots, n\}\), we have \(\nabla_A(\theta \tau_i) = \top\). By the inductive hypothesis,
\(\nabla_A(\theta \tau_i) = (\nabla_A \theta)(\nabla_B \tau_i)\), and hence \((\nabla_A \theta)(\nabla_B \tau_i) = \top\). It is possible only if \(\nabla_B \tau_i = \top\). Hence \(\nabla_B(F(\tau_1, \ldots, \tau_n)) = \top\), and thus
\[
R = (\nabla_A \theta)(\nabla_B(F(\tau_1, \ldots, \tau_n))) = (\nabla_A \theta)(\top) = \top = L.
\]
Now suppose that \(L \neq \top\). We have:
\[
L = F(\nabla_A \theta \tau_1, \ldots, \nabla_A \theta \tau_n)
= F((\nabla_A \theta)(\nabla_B \tau_1), \ldots, (\nabla_A \theta)(\nabla_B \tau_n)) \quad \text{(by the inductive hypothesis)}
= (\nabla_A \theta)(F(\nabla_B \tau_1, \ldots, \nabla_B \tau_n))
\]
Since \(L \neq \top\), for some \(k \in \{1, \ldots, n\}\) we have \(\nabla_A(\theta \tau_k) \neq \top\) and thus, by the inductive hypothesis, \((\nabla_A \theta)(\nabla_B \tau_k) \neq \top\). It is possible only if \(\nabla_B \tau_k \neq \top\). Hence
\[
L = (\nabla_A(\theta)(F(\nabla_B \tau_1, \ldots, \nabla_B \tau_n))) = (\nabla_A \theta)(\nabla_B F(\tau_1, \ldots, \tau_n)) = R.
\]
d) \(\tau = \tau_1 \cap \tau_2\).
\[
L = \nabla_A(\theta(\tau_1 \cap \tau_2))
= \nabla_A(\theta(\tau_1)) \cap \nabla_A(\theta(\tau_2))
= (\nabla_A \theta)(\nabla_B(\tau_1)) \cap (\nabla_A \theta)(\nabla_B(\tau_1)) \quad \text{(by the inductive hypothesis)}
= (\nabla_A \theta)(\nabla_B(\tau_1) \cap \nabla_B(\tau_2))
= (\nabla_A \theta)(\nabla_B(\tau_1 \cap \tau_2)) = R.
\]
e) If \(\tau = \tau_1 \cup \tau_2\) the proof is similar.

Now, we are able to achieve the purpose of this section: to prove the Pruning Theorem.

Proof of the Pruning Theorem  By the definition of well-typedness (Definition 2.42), we should show that if \(C\) is a clause of the program \(P\) with the head predicate symbol \(p\) and \((p : \tau' \rightarrow \sigma') \in T'\) then
\[
\vdash \text{ClauseHasType}(T', C, \tau' \rightarrow \sigma').
\]
If \((p : \tau' \rightarrow \sigma') \in T\) then (6.11) easily follows from Lemma 2.56. So, suppose that \((p : \tau' \rightarrow \sigma') \notin T\). Thus it has the form
\[
p : (\nabla_A(\tau_1), \ldots, \nabla_A(\tau_n)) \rightarrow (\nabla_A(\sigma_1), \ldots, \nabla_A(\sigma_n))
\]
where \((p : (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)) \in T\) and \(A \subseteq V_T\). We should show that
\[
\vdash \text{ClauseHasType}(T', C, (\nabla_A(\tau_1), \ldots, \nabla_A(\tau_n)) \rightarrow (\nabla_A(\sigma_1), \ldots, \nabla_A(\sigma_n))).
\]
We will construct the proof of this fact using the proof of
\[
\vdash \text{ClauseHasType}(T, C, (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n))
\]
119
which exists because \((p: (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)) \in \mathcal{T}\) and \(P\) is well typed with respect to \(\mathcal{T}\). Let \(C = p(t_1, \ldots, t_n) \vdash B\). (6.13) is proved using rule \((P_3)\), and thus we have

\[
\begin{align*}
(6.14) & \quad \varnothing, (t_1: \tau_1, \ldots, t_n: \tau_n) \Rightarrow \Gamma_1, \\
(6.15) & \quad \vdash \text{InferFromAtoms}(\mathcal{T}, \Gamma_1, B, \Gamma_2), \\
(6.16) & \quad \Gamma_2 \vdash t_i: \sigma_1, \ldots, \Gamma_2 \vdash t_n: \sigma_n
\end{align*}
\]

Let

\[
(6.17) \quad \varnothing, (t_1: \nabla_A(\tau_1), \ldots, t_n: \nabla_A(\tau_n)) \Rightarrow \hat{\Gamma}_1.
\]

By Lemma 6.19, we have \(\Gamma_1 \prec \hat{\Gamma}_1\). If we assume that there exists \(\hat{\Gamma}_2\) such that

\[
\vdash \text{InferFromAtoms}(\mathcal{T}', \hat{\Gamma}_1, B, \hat{\Gamma}_2) \quad \text{and} \quad \Gamma_2 \prec \hat{\Gamma}_2
\]

then, by Definition 6.9, \(\hat{\Gamma}_2 \vdash t_i: \nabla_A(\sigma_i)\) for \(i \in \{1, \ldots, n\}\), which finishes the proof. Therefore, it remains to show that if

\[
(6.18) \quad \Gamma_1 \prec \hat{\Gamma}_1 \quad \text{and} \quad \vdash \text{InferFromAtoms}(\mathcal{T}, \Gamma_1, B, \Gamma_2)
\]

then there exists \(\hat{\Gamma}_2\) such that

\[
\Gamma_2 \prec \hat{\Gamma}_2 \quad \text{and} \quad \vdash \text{InferFromAtoms}(\mathcal{T}', \hat{\Gamma}_1, B, \hat{\Gamma}_2),
\]

We proceed by induction on the length of \(B\).

- \(B = \{\}\). This case is trivial. Rule \((P_1)\) is used, thus \(\Gamma_2 = \Gamma_1\), and \(\hat{\Gamma}_2 = \hat{\Gamma}_1\). Since \(\Gamma_1 \prec \hat{\Gamma}_1\), we have \(\Gamma_2 \prec \hat{\Gamma}_2\).
- \(B = \{a_1, \ldots, a_k\}, k > 0\). Let \(a_1 = p(t_1, \ldots, t_n)\). Assume that (6.18) holds. Then \(\vdash \text{InferFromAtoms}(\mathcal{T}, \Gamma_1, B, \Gamma_2)\) is proved using rule \((P_2)\) from

\[
\begin{align*}
(6.19) & \quad \Gamma_1 \vdash t_i: \theta(\tau_i), \\
(6.20) & \quad \Gamma_1, (t_1: \theta(\sigma_1), \ldots, t_n: \theta(\sigma_n)) \Rightarrow \Gamma', \\
& \quad \vdash \text{InferFromAtoms}(\mathcal{T}, \Gamma', \Gamma_1, B, a_1, \Gamma_2)
\end{align*}
\]

where \((p: (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)) \in \mathcal{T}\). Let

\[
B = \{\alpha \in \text{dom}(\theta) \mid \nabla_A(\theta(\alpha)) = \top\}.
\]

Let \(\hat{\theta} = \nabla_A(\theta), \hat{\tau}_i = \nabla_B(\tau_i),\) and \(\hat{\sigma}_i = \nabla_B(\sigma_i)\). Then, by Lemma 6.20,

\[
\begin{align*}
(6.21) & \quad \nabla_A(\theta(\tau_i)) = (\nabla_A(\theta))(\nabla_B(\tau_i)) = \hat{\theta}(\hat{\tau}_i), \\
(6.22) & \quad \nabla_A(\theta(\sigma_i)) = (\nabla_A(\theta))(\nabla_B(\sigma_i)) = \hat{\theta}(\hat{\sigma}_i)
\end{align*}
\]

By the definition of \(\mathcal{T}'\), we have also

\[
(6.23) \quad (p: (\hat{\tau}_1, \ldots, \hat{\tau}_n) \rightarrow (\hat{\sigma}_1, \ldots, \hat{\sigma}_n)) \in \mathcal{T}'.
\]

By (6.19), and Definition 6.9, \(\hat{\Gamma}_1 \vdash t_i: \nabla_A(\theta(\tau_i))\), and hence by (6.21) we obtain

\[
(6.24) \quad \hat{\Gamma}_1 \vdash t_i: \hat{\theta}(\hat{\tau}_i).
\]

From (6.20) and (6.22), by Lemma 6.19, it follows that there exists \(\hat{\Gamma}'\) such that

\[
(6.25) \quad \hat{\Gamma}_1, (t_1: \hat{\theta}(\hat{\tau}_1), \ldots, t_n: \hat{\theta}(\hat{\tau}_n)) \Rightarrow \hat{\Gamma}'.
\]
and $\Gamma' \sim \hat{\Gamma}'$. Finally, by induction, there exists $\hat{\Gamma}_2$ such that

$$
(6.26) \quad \vdash \text{InferFromAtoms}(T', \hat{\Gamma}', (a_2, \ldots, a_k), \hat{\Gamma}_2),
$$

and $\Gamma_2 \sim \hat{\Gamma}_2$. Now, (6.23), (6.24), (6.25) and (6.26) allow to use $(P_2)$ to obtain

$$
\vdash \text{InferFromAtoms}(T', \hat{\Gamma}_1, B, \hat{\Gamma}_2).
$$

Since we have shown that $\Gamma_2 \sim \hat{\Gamma}_2$, the proof is completed.

\section{6.5 Existence of the Main Type}

In this section we give the definition of the \textit{main type} of a program. We describe also family of programs for which the main types exists.

\begin{definition}
Let $T$ and $T'$ be directional types of a program. $T'$ is a \textit{generator} of $T$ if, for each $(p : \tau \rightarrow \sigma) \in T$, there exists $(p : \tau' \rightarrow \sigma') \in T'$ such that $(\tau' \rightarrow \sigma') \Rightarrow (\tau \rightarrow \sigma)$. \hfill \qed
\end{definition}

Note that in the definition above we use ordinary derivation, not $\nabla$-derivation.

\begin{definition}
A directional type $\tau \rightarrow \sigma$ is \textit{proper} if

(i) $\nabla_\sigma(\sigma) = \sigma$,

(ii) $\nabla_\tau(\tau) = \top$, where $V = \text{var}(\tau)$.

A directional type $T$ of a program is \textit{proper} if all types occurring in $T$ are proper. \hfill \qed
\end{definition}

\begin{definition}
For a given program $P$ let

$$
\mathcal{X} = \{ T \mid T \text{ is a proper directional type such that } P \text{ is well typed with respect to } T \}.
$$

We define the \textit{maximal proper directional type of the program} $P$ as follows

$$
T_{\text{max}}^P = \bigcup_{T \in \mathcal{X}} T.
$$

\hfill \qed
\end{definition}

Note that $P$ is well typed with respect to $T_{\text{max}}^P$.

\begin{definition}
A directional type $T$ of a program $P$ is the \textit{main type} of $P$ if

(i) for each predicate $p$ of the program $P$, $T$ contains exactly one element of the form $(p : \tau \rightarrow \sigma)$,

(ii) $T_{\text{max}}^P$ is $\nabla$-derivable from $T$.

For a fixed program $P$, $\tau \rightarrow \sigma$ is the \textit{main type} of a predicate $p$ if $(p : \tau \rightarrow \sigma)$ belongs to the main type of $P$. \hfill \qed

Note, that some improper types may not be $\nabla$-derivable from the main type of a program.

\begin{theorem}[Main Type Theorem]
A program $P$ has the main type if and only if there exists a finite generator of $T_{\text{max}}^P$.
\end{theorem}
Proof. ($\Leftarrow$) Consider a program $P$. Assume that $T_{\text{max}}^P$ has a finite generator $G$. Without loss of generality we can assume that, for $a, b \in G$, if $a \neq b$ then $\text{var}(a) \cap \text{var}(b) = \emptyset$. For a predicate $p$ of the program $P$ let

$$G_p = \{(\tau \rightarrow \sigma) \mid (p : \tau \rightarrow \sigma) \in G\}.$$ 

Since $G$ is finite, $G_p$ can be written as $\{\tau_1^p \rightarrow \sigma_1^p, \ldots, \tau_{n_p}^p \rightarrow \sigma_{n_p}^p\}$. Let $\mathcal{M} = \{(p : \tau_1^p \cap \cdots \cap \tau_{n_p}^p \rightarrow \sigma_1^p \cap \cdots \cap \sigma_{n_p}^p) \mid p \text{ is a predicate of } P\}$. We will show that $\mathcal{M}$ is the main type of the program $P$. Obviously, $\mathcal{M}$ satisfies condition (i) of Definition 6.24. Now, we shall check that $\mathcal{M}$ satisfies also condition (ii).

Let $(p : \tau \rightarrow \sigma) \in T_{\text{max}}^P$. Since $G$ is a generator of $T_{\text{max}}^P$, there exist $\tau', \sigma'$ such that $(p : \tau' \rightarrow \sigma') \in G$ and $(\tau' \rightarrow \sigma') \rightarrow (\tau \rightarrow \sigma)$. Since $(p : \tau' \rightarrow \sigma') \in G$, we have $\tau' = \tau_k^p$ and $\sigma' = \sigma_k^p$, for some $k \in \{1, \ldots, n_p\}$. Let $A = \text{var}(\tau_1^p, \ldots, \tau_{n_p}^p, \sigma_1^p, \ldots, \sigma_{n_p}^p) \setminus \text{var}(\tau_k^p, \sigma_k^p)$. Since, for each $i \in \{1, \ldots, n\}$, $\tau_i^p \rightarrow \sigma_i^p$ is proper, and, for each $i, j$ such that $i \neq j$, $\text{var}(\tau_i^p, \sigma_i^p) \cap \text{var}(\tau_j^p, \sigma_j^p) = \emptyset$, we have

$$\nabla_A(\tau_i^p) = \top \quad \text{if } i \neq k$$

and

$$\nabla_A(\sigma_k^p) = \sigma_k^p$$

Moreover, by Lemma 6.10, $\nabla_A(\tau_k^p) \geq \tau_k^p$. Hence

$$\vdash \nabla_A(\tau_1^p \cap \cdots \cap \tau_{n_p}^p) = \nabla_A(\tau_1^p) \cap \cdots \cap \nabla_A(\tau_{n_p}^p) = \nabla_A(\tau_k^p) \geq \tau_k^p = \tau',$$

and

$$\vdash \nabla_A(\sigma_1^p \cap \cdots \cap \sigma_{n_p}^p) = \nabla_A(\sigma_1^p) \cap \cdots \cap \nabla_A(\sigma_{n_p}^p) \leq \nabla_A(\sigma_k^p) = \sigma_k^p = \sigma'.$$

Since $\vdash \nabla_A(\tau_1^p \cap \cdots \cap \tau_{n_p}^p) \geq \tau'$ and $\vdash \nabla_A(\sigma_1^p \cap \cdots \cap \sigma_{n_p}^p) \leq \sigma'$, by Definition 2.60,

$$(\nabla_A(\tau_1^p \cap \cdots \cap \tau_{n_p}^p) \rightarrow \nabla_A(\sigma_1^p \cap \cdots \cap \sigma_{n_p}^p)) \rightarrow (\tau' \rightarrow \sigma').$$

Since $(\tau' \rightarrow \sigma') \rightarrow (\tau \rightarrow \sigma)$, and $\rightarrow$ is transitive, we have

$$(\nabla_A(\tau_1^p \cap \cdots \cap \tau_{n_p}^p) \rightarrow \nabla_A(\sigma_1^p \cap \cdots \cap \sigma_{n_p}^p)) \rightarrow (\tau \rightarrow \sigma).$$

Thus, by Definition 6.4,

$$(\tau_1^p \cap \cdots \cap \tau_{n_p}^p) \nabla (\tau \rightarrow \sigma).$$

Since $(p : (\tau_1^p \cap \cdots \cap \tau_{n_p}^p) \rightarrow (\sigma_1^p \cap \cdots \cap \sigma_{n_p}^p)) \in \mathcal{M}$, we have shown that $T_{\text{max}}^P$ is $\nabla$-derivable from $\mathcal{M}$.

($\Rightarrow$) Assume that the program $P$ has the main type $\mathcal{M}$. Let

$$G = \{(p : \nabla_A(\tau) \rightarrow \nabla_A(\sigma)) \mid (p : \tau \rightarrow \sigma) \in \mathcal{M}, A \subseteq \text{var}(\tau, \sigma)\}.$$ 

First, let us notice that $G$ is finite. Assume that $(p : \tau \rightarrow \sigma) \in T_{\text{max}}^P$. Since $T_{\text{max}}^P$ is $\nabla$-derivable from $\mathcal{M}$, there exists $\tau', \sigma'$, and $B \subseteq V$ such that $(p : \tau' \rightarrow \sigma') \in \mathcal{M}$ and

$$(\nabla_B(\tau') \rightarrow \nabla_B(\sigma')) \rightarrow (\tau \rightarrow \sigma).$$

Let $C = B \cap \text{var}(\tau', \sigma')$. It is easy to show that $\nabla_C(\tau') = \nabla_B(\tau')$ and $\nabla_C(\sigma') = \nabla_B(\sigma')$, which gives $(\nabla_C(\tau') \rightarrow \nabla_C(\sigma')) \rightarrow (\tau \rightarrow \sigma)$. From the definition of $G$ we know that $(p : \nabla_C(\tau') \rightarrow \nabla_C(\sigma')) \in G$ which shows that $G$ is a generator of $T_{\text{max}}^P$. \qed
6.5. Existence of the Main Type

\[
\text{multiAppend}(X_1, Y_1, Z_1, \ldots, X_n, Y_n, Z_n) :- \\
\text{append}(X_1, Y_1, Z_1), \ldots, \text{append}(X_n, Y_n, Z_n).
\]

Figure 6.1: The multiAppend predicate

The proof of the \((\Rightarrow)\) part of the Main Type Theorem shows that the main type \(\mathcal{M}\) of a given program \(P\) describes finite generators of \(T^P_{\text{max}}\) of the exponential size with respect to the size of \(\mathcal{M}\). Usefulness of the main type theorem illustrates the following example.

**Example 6.4.** Consider the program consisting of the predicate \(\text{append}\) (figure 2.2) and the predicate \(\text{multiAppend}\) (figure 6.1). This program has finite generators, but the size of the smallest of them is exponential with respect to \(n\). It means that we have to store exponentially many types if we want to derive all possible types of that program without pruning. More precisely, we have to remember \(6^n\) types for \(\text{append}\) and \(6^n\) types \(\text{multiAppend}\).

However, since the generator is finite, Theorem 6.2 states that the program has the main type. This type consists of two types: the type

\[
(6.27)\quad (\text{list}(\alpha) \cap \alpha', \text{list}(\beta) \cap \beta', \text{list}(\gamma) \cap \gamma') \rightarrow \\
(\text{list}(\alpha \cap \gamma) \cap \alpha', \text{list}(\beta \cap \gamma) \cap \beta', \text{list}(\gamma \cap (\alpha \cup \beta)) \cap \gamma')
\]

for \(\text{append}\), and the type

\[
(\text{list}(\alpha_1) \cap \alpha_1', \text{list}(\beta_1) \cap \beta_1', \text{list}(\gamma_1) \cap \gamma_1', \\
\vdots \\
\text{list}(\alpha_n) \cap \alpha_n', \text{list}(\beta_n) \cap \beta_n', \text{list}(\gamma_n) \cap \gamma_n') \rightarrow \\
(\text{list}(\alpha_1 \cap \gamma_1) \cap \alpha_1', \text{list}(\beta_1 \cap \gamma_1) \cap \beta_1', \text{list}(\gamma_1 \cap (\alpha_1 \cup \beta_1)) \cap \gamma_1') \\
\vdots \\
\text{list}(\alpha_n \cap \gamma_n) \cap \alpha_n', \text{list}(\beta_n \cap \gamma_n) \cap \beta_n', \text{list}(\gamma_n \cap (\alpha_n \cup \beta_n)) \cap \gamma_n')
\]

for \(\text{multiAppend}\). As we can see, these types are small, i.e. are of the size proportional to the number of arguments of the predicates. From these types we can obtain all proper types of these predicates by using pruning, weakening and substitutions.

Let us notice that type (6.27) can be represented as the intersection of

\[
(6.28)\quad (\text{list}(\alpha), \text{list}(\beta), \text{list}(\gamma)) \rightarrow (\text{list}(\alpha \cap \gamma), \text{list}(\beta \cap \gamma), \text{list}(\gamma \cap (\alpha \cup \beta)))
\]

and

\[
(6.29)\quad (\alpha', \beta', \gamma') \rightarrow (\alpha', \beta', \gamma').
\]

Type (6.29) is a tautological type, i.e. type whose left and right-hand sides are equal. Such types are valid for any predicate of adequate arity, thus they do not provide any interesting information about a predicate. Therefore, we need not to keep them. So, we can simplify the main type of \(\text{append}\) to the type (6.28) which is more compact. In the similar way we can simplify the main type of \(\text{multiAppend}\). \(\square\)

Theorem 6.2 requires some comments.

1. Some programs do not have a finite generator, and thus they do not have the main type. Let us consider the following example. The binary predicate \(=\) is a shorthand for predicate \(\text{unif}\) (see example 2.8), and has the type \((\alpha, \beta) \rightarrow (\alpha \cap \beta, \alpha \cap \beta)\).
Chapter 6. The Main Type

\[ p(X,Y) : \text{if } X=Y, \text{ otherwise } \]
\[ p(X,Y) : \text{if } X=Y, \text{ otherwise } \]

The predicate \( p \) has the following types:

\[
(\alpha, \beta) \rightarrow (\alpha \cap \beta, \alpha \cap \beta \cap \text{list}(\alpha \cap \beta))
\]
\[
(\alpha, \beta) \rightarrow (\alpha \cap \beta, \alpha \cap \beta \cap \text{list}(\alpha \cap \beta))
\]
\[
(\alpha, \beta) \rightarrow (\alpha \cap \beta, \alpha \cap \beta \cap \text{list}(\alpha \cap \beta \cap \text{list}(\alpha \cap \beta)))
\]
\[
\vdots
\]

and does not have either a finite generator nor the main type. In fact, this predicate is completely useless. It is hard to find any useful predicate which does not have a finite generator. One of the possible reason of this situation is presented in the next section where we define some class of predicates. If a predicate belongs to this class, it has a finite generator (and the main type). As it turns out, all analyzed by us predicates from the standard Prolog library, except flatten, belong to this class.

Furthermore, we strongly believe that the existence of the main type in our system is closely related to the existence of the most general type in the Mycroft-O’Keefe type system [16]. We think that if the type reconstruction algorithm for the Mycroft-O’Keefe type system [14] finds the most general type for a program, then this program has the main type which can be found by the type reconstruction algorithm given in Chapter 7. We do not have, however, a complete proof of this fact yet.

2. The main type of a predicate, if exists, describes all proper types of the predicate. However some types which do not satisfy property (i) of Definition 6.22 cannot be obtained from the main type. For example, (6.27) is the main type of the predicate \( \text{append} \). Types (6.2) – (6.6) can be obtained from (6.27), but \( \text{append} \) has also the type

\[
(\alpha, \beta, \gamma) \rightarrow (\alpha \cap \text{list}(\beta), \beta, \gamma),
\]

which is not proper (it does not fulfill property (i) of the Definition 6.22), and it cannot be obtain from the main type. More precisely, types obtained by substitution from

\[
(\alpha, \beta, \gamma) \rightarrow (\alpha \cap \text{list}(\beta), \beta, \gamma),
\]

cannot be obtained from the main type. However, from the main type one can obtain types from that family without \( \text{list}(\top) \).

There is, however, a simple idea which helps to deal with improper types. Most of improper types which cannot be derived from the main type, are intersections of types derived from the main type, and a type with assumptions of the form \( (\top, \ldots, \top) \), like (6.30). Recall that, by Lemma 2.53, intersection of valid types of a predicate \( p \) is a valid type of \( p \). So, all the types of \( \text{append} \) can be obtained by taking the intersection of (6.30) and a type derived from the main type of \( \text{append} \).

3. Condition (ii) of Definition 6.22 is not a serious restriction. One can prove that if a predicate \( p \) has a type \( (\tau \rightarrow \sigma) \) which does not satisfies this property then \( p \) has also a type \( (\tau' \rightarrow \sigma') \) satisfying this property such that \( (\tau \rightarrow \sigma) \) can be obtain from \( (\tau' \rightarrow \sigma') \) by substitution. It follows from the definition of signature in System P. Let us consider an example:

\[
p(1, 1).
\]
\[
p(X, X).
\]

124
Predicate $p$ has types $((\text{int}, \text{int}) \rightarrow (\text{int}, \text{int}))$, $((\text{int}, \top) \rightarrow (\text{int}, \text{int}))$, and $((\top, \text{int}) \rightarrow (\text{int}, \text{int}))$. These types do not satisfy condition (ii) of Definition 6.22. However, they can be obtained from the following type of $p$.

$$(\alpha, \beta) \rightarrow (\alpha \cap (\beta \cup \text{int}), \beta \cap (\alpha \cup \text{int})).$$
Chapter 7

Type Reconstruction

In this chapter we study the problem of type reconstruction. The main task of this chapter is to give a type reconstruction algorithm. First we define the notion of a cluster.

Definition 7.1. Let \( P \) be a program. Let \( p, q \) be predicates from \( P \). We say that the predicate \( p \) directly depends on the predicate \( q \) in program \( P \) (written \( p \leadsto_p q \)) iff there is a clause in \( P \) with head \( p \) and with \( q \) in the body. We say that predicates \( p \) and \( q \) depend on each other in \( P \) (written \( q \leadsto_p p \)) iff

\[
q \leadsto_p^* p \text{ and } p \leadsto_p^* q
\]

where \( \leadsto_p^* \) is a transitive and reflexive closure of \( \leadsto_p \). We omit \( P \) if it is clear from context.

Remark 7.2. The relation \( \equiv \) is an equivalence relation.

Definition 7.3. A cluster in a program \( P \) is an equivalence class of relation \( \equiv_p \).

In this section we assume that the reconstruction task is given in a similar way as the problem of reconstruction of guarantees.

Definition 7.4. Let \( Q \) be a program. Let \( T \) be a directional type of \( Q \). Assume that \( T \) is represented by main types. Let \( P \) be a program whose clauses contain calls of predicates from \( P \) and \( Q \). Assume that predicates from \( P \) do not occur in \( Q \), and form a cluster. A type reconstruction task, or TR-task, is a tuple \( \langle Q, T, P \rangle \).

We will use names \( q_i \) for predicates in \( Q \) (with known types) and names \( p_i \) for predicates in \( P \).

In this section we use the modified version of ResultType which benefits the existence of a main type. If the function \( BS \) returns a substitution with constants \( T^* \) in the range then the function ResultType discussed in 4 returns type error. In this chapter in such situations we use the pruned type of analyzed predicate. The set of variables which is an argument of pruning is the set of variables, to which \( BS \) assigns \( T^* \).

We shall give an algorithm which for a TR-task \( \langle Q, T, P \rangle \) finds the main type \( T' \) of \( P \).

Definition 7.5. The assumption \( \tau = (\tau_1, \ldots, \tau_m) \) is a simple assumption if each \( \tau_i \) is an assumption type with distinct variables and without \( \tau \).
Definition 7.6. The guarantee \( \tau = (\tau_1, \ldots, \tau_m) \) is a simple guarantee if in each \( \tau_i \) operators \( \cap, \cup \) are applied only to types without polymorphic type constructors. \( \square \)

Example 7.1. The guarantee \( (F(\alpha \cap \beta, G(\text{real} \cup \gamma))) \) is a simple guarantee. \( (F(\top, \top) \cap G(\bot)) \) and \( (G(\alpha \cup G(\beta))) \) are not simple guarantees. \( \square \)

We define a notion of ordinary types.

Definition 7.7. Let \( (Q, T, P) \) be a RT-task. Assume that \( p_1, \ldots, p_n \) are all predicates in \( P \). We say that \( P \) has an ordinary type \( T' = \{(p_1 : \tau_1 \to \sigma_1), \ldots, (p_n : \tau_n \to \sigma_n)\} \) if for each \( i \in \{1, \ldots, n\} \):
1. \( \tau_i \) is a simple assumptions, \( \sigma_i \) is simple guarantee,
2. \( \text{shape} (\tau_i) = \text{shape} (\sigma_i) \),
3. \( T' \cup T \) is a directional type of \( Q \cup P \).

For ordinary types we denote by \( As(p_i, T) \) an assumption of the only directional type of \( p_i \) in \( T \).

7.1 Strongly Typeable Predicates

In this section we define a notion of strongly typeable predicates. This notion formalizes the idea that (1) types should have simple shapes, (2) predicates should be used according to this shape.

We need some definitions

Definition 7.8. Let \( \rho \) be a type and \( \tau \) be an assumption type. Assume without loss of generality that \( \text{var} (\rho) \cap \text{var} (\tau) = \emptyset \). We say that type \( \rho \) is safe for assumption \( \tau \) if the \( \text{L-normal form} \) of \( \tau \cap \rho \) does not contain subexpressions of the form \( F(\xi_1, \ldots, \xi_m) \cap G(\xi_1, \ldots, \xi_n) \) or \( \alpha \cap F(\xi_1, \ldots, \xi_n) \), where \( \alpha \in \text{var} (\rho) \). \( \square \)

Example 7.2. Let \( \tau = F(\alpha, F(\beta, \gamma)) \). The type \( F(\alpha', \beta') \) is not safe for \( \tau \), since
\[ \vdash \tau \cap \rho = F(\alpha \cap \alpha', \beta' \cap F(\beta, \gamma)) \]

Definition 7.9. A clause \( C_2 \) is an extension of a clause
\[ C_1 = p(t) :: a_1, \ldots, a_n \]
if \( C_2 = p(t) \leftarrow a_1, \ldots, a_n, (a_1 + \cdots + a_n)^* \), where \( (a_1 + \cdots + a_n)^* \) denotes a finite sequence of atoms from \( \{a_1, \ldots, a_n\} \). \( \square \)

Definition 7.10. Let \( C \) be a clause with the predicate symbol \( p \), and let \( T \) be a directional type. Assume that \( \tau \) is an assumption from the main type of \( p \) in \( T \).

A clause \( C \) is saturated for a directional type \( T \) if all its extensions have equivalent (wrt. to \( \sim \)) result environments for the assumption \( \tau \) and the directional type \( T \). \( \square \)

We define a useful class of strongly typeable programs. For this class we are able to give a type reconstruction algorithm. We believe that type reconstruction algorithm does not exist for all types.

Definition 7.11. Let \( (Q, T, P) \) be a RT-task. We say that \( P \) is strongly typeable if \( P \) has an ordinary main type which satisfies the following additional conditions:
7.1. Strongly Typeable Predicates

(i) recursive calls (i.e. calls of \( p_k \), for \( k \in \{1, \ldots, n\} \)) are allowed only if the type of an argument has exactly the same shape as the assumptions of a predicate being called,

(ii) each clause \( C \) in \( P \) has saturated extension, denoted by \( e(C) \),

(iii) for each clause \( C \) in \( P \) the result environment, for \( e(C) \), \( T \) does not contain expressions of the form \( G(\sigma') \cap F(\tau') \), or \( \alpha \cap F(\tau') \).

(iv) for each clause \( C \) in \( P \) every term \( t \) from \( C \), can be typed without using the rule \((T_1)\) in the result environment for \( e(C) \) and \( T \).

(v) Let \( C \) be a clause. Let \( q(t_1, \ldots, t_m) \) be an atom in the body of \( C \). Let \( (\tau_1, \ldots, \tau_m) \rightarrow (\sigma_1, \ldots, \sigma_m) \) be the main type of \( q \) in \( T \). We demand that wile computing the result environment, for \( e(C) \) and analyzing an atom \( q(t_1, \ldots, t_m) \), for current environment \( \Gamma \), we have that \( BT(\Gamma, t_i) \) is safe for \( \tau_i \), for each \( i \in \{1, \ldots, n\} \).

So, for strongly typeable predicates we can freely extend their clauses and expect that in the result environments such types as \( G(\sigma') \cap F(\tau') \) will not occur.

The condition (v) ensures that predicates that have already been assigned a type are used according to shapes of their assumptions.

We believe that violation of any of the conditions (i)-(v) of Definition 7.11 is often connected with an error and is a sign of a bad (meaning non-typeable) programming style.

**Example 7.3.** Below we give some predicates which are not strongly typable.

\[
\begin{align*}
p(\emptyset). \\
p(X) &::= p([X]). \\
q(X) &::= X=[X]. \\
r(X) &::= X=[1], \ X=\text{prod}(1,3). \\
s(X,Y) &::= X=[Y][1]. \\
t(X) &::= \text{append}(\text{prod}(X,X),[1],[1,2]).
\end{align*}
\]

These predicates do not satisfy the following conditions of Definition 7.11: predicate \( p \) violates condition (i), predicate \( q \) — conditions (ii),(iii), predicate \( r \) — condition (iii) and predicate \( s \) — condition (iv), predicate \( t \) — condition (v).

7.1.1 Algorithm

The process of searching the main type can be split into two parts. The first part consists of computing assumptions and the second one consists of computing guarantees of the same shape as assumptions. The second part is performed by using Algorithm 5.1, described in Section 5.1.

We transform the problem of finding assumptions into the problem of solving a set \( S \) of term equations. These equations can be solved by a term unification algorithm.

We define two operations on program atoms which for an atom \( a \) give a set \( S(a) \) of equations connected with this atom. The set \( S \) is an union of sets \( S(a) \).

**Remark 7.12.** The solution \( \Theta \) of term equations can be considered as an environment \( \Gamma = \{ X : \Theta(X) \mid X \in \text{var}(P) \} \), where \( P \) is a program.
**Definition 7.13.** We define a function $\mathcal{E}_1$ which for a term and a variable returns the set of equations:

$$\mathcal{E}_1(X, T) = \{X = T\}$$

$$\mathcal{E}_1(f(t_1, \ldots, t_n), T) =$$

$$\{T = \theta(\mathcal{N}(\sigma)), T_1 = \theta(\sigma_1), \ldots, T_n = \theta(\sigma_n)\} \cup \bigcup_{i=1}^n \mathcal{E}_1(t_i, T_i)$$

where all $T_i$ are new distinct variables, $f$ has a signature $\sigma_1 \ast \cdots \ast \sigma_n \rightarrow \sigma$, $\theta$ is a substitution which replaces signature variables by new ones. $\mathcal{N}$ is a function defined in Definition 2.40.

The function $\mathcal{E}_1$ returns a set of equations. A solution of these equations gives an environment (see Remark 7.12) in which term $t$ can be typed without rule $(T_1)$.

Consider an example:

**Example 7.4.**

$$\mathcal{E}_1([X|Y], T) = \{T = \text{list}(\alpha), T_1 = \alpha, T_2 = \text{list}(\alpha)\} \cup \mathcal{E}_1(X, T_1) \cup \mathcal{E}_1(Y, T_2)$$

$$= \{T = \text{list}(\alpha), T_1 = \alpha, T_2 = \text{list}(\alpha), T_1 = X, T_2 = Y\}$$

Below is a simple example for which the function $\mathcal{N}$ is important.

**Example 7.5.** We have that $[\ ]$ has a signature list(⊥).

$$\mathcal{E}_1([\ ], T) = \{T = \text{list}(\alpha')\}$$

where $\alpha'$ was returned by $\mathcal{N}$.  

**Definition 7.14.** Let $\tau \rightarrow \sigma$ be a type such that $\tau = (\tau_1, \ldots, \tau_n)$, $\sigma = (\sigma_1, \ldots, \sigma_n)$ and $\sigma_1, \ldots, \sigma_n$ are in L-normal form. A function $\mathcal{E}_2$ for $\tau \rightarrow \sigma$ and a tuple of variables $Y_1, \ldots, Y_n$ returns the set of equations:

$$\mathcal{E}_2(\tau \rightarrow \sigma, (Y_1, \ldots, Y_n)) = \theta(\mathcal{E}_1) \cup \theta(\mathcal{E}_2)$$

where

$$E_1 = \{(Y_i = \tau_i) \mid i \in \{1, \ldots, n\}\}$$

$$E_2 = \{\alpha_1 = \alpha_2, \alpha_2 = \alpha_3, \ldots, \alpha_{k-1} = \alpha_k \mid \text{for each } \alpha_1 \cap \cdots \cap \alpha_k \text{ in } \sigma\}$$

and $\theta$ is a renaming substitution replacing the variables in $\text{var}(\tau \rightarrow \sigma)$ by new variables.

**Example 7.6.** Consider the predicate $\text{unif}$ with type $(\alpha, \beta) \rightarrow (\alpha \cap \beta, \alpha \cap \beta)$. Suppose, that in the program $P$ there is an atom $\text{unif}(X, Y)$. We have

$$\mathcal{E}_2(X, Y, (\alpha, \beta) \rightarrow (\alpha \cap \beta, \alpha \cap \beta)) = \{X = \alpha', Y = \beta'\} \cup \{\alpha' = \beta', \alpha' = \beta'\}$$
append(X,Y,Y) := X=[].
append(X,Y,Z) := X=[V|Xs], Z = [V|Zs],
append(Xs, Y, Zs).

Figure 7.1: Predicate append in a constraint style.

**Definition 7.15.** A program is written in the constraint style if function symbols occur only inside atoms of the form:

\[ X = t \]

where \( t \) is a program term and \( X \) is a variable.

**Definition 7.16.** Program variables have non-confusing names if the following conditions hold:

1. Head variables for the clauses defining one predicate have the same names.
2. Non head variables from two different clauses have different names.

Every program can be easily rewritten to a constraint style with non-confusing names (see e.g. Figure 7.1).

We give Algorithm 7.1 which reconstructs types. It uses the operations \( E_1, E_2 \) to compute a set of term equations. Then these equations are solved, and solution (if exists) determines the shape of assumptions for each untyped predicate. For such assumptions we find the guarantee. We use a function Guarantee' which is defined very similarly to the function Guarantee (Algorithm 5.1). The only difference is that Guarantee starts with computed guarantees equal to \( \bot \), while Guarantee's starts with guarantees equal to its additional argument.

**Example 7.7.** In Figure 7.2 we show the set of equations given by Algorithm 7.1 for the predicate append (see fig. 7.1). The solution of these equations gives the shapes

\( \text{list}(T), \text{list}(\bot), \text{list}(\bot) \)


### 7.2 Properties of Type Reconstruction Algorithm

In this section we prove that Algorithm 7.1 applied to strongly typeable predicates finds their main type. Moreover, we analyze complexity of this algorithm.

We will need a characterization of the main type for strongly typeable predicates.

**Definition 7.17.** Let \( R = \langle Q, T, P \rangle \) be a RT-task. Assume that \( P \) consists of predicates \( p_1, \ldots, p_n \). Let \( \tau_1, \ldots, \tau_n \) be a vector of types. Assume that \( \sigma_1, \ldots, \sigma_n \) are the best provable guarantees for predicates \( p_1, \ldots, p_n \), and for assumptions \( \tau_1, \ldots, \tau_n \). The assumptions \( \tau_1, \ldots, \tau_n \) are sufficient for \( D \) if for any substitution \( \theta \) the best provable guarantees for \( \theta(\tau_1), \ldots, \theta(\tau_n) \) are equal to \( \theta(\sigma_1), \ldots, \theta(\sigma_n) \).

In other words the assumption is sufficient if stronger assumption do not give better guarantee.

**Example 7.8.** Not all assumptions are sufficient. The rule \( (K_9) \) can cause an assumption to be not sufficient. For instance the clause
function FindType(Q, T, P)
    Q — a typed program
    T — a directional type of Q
    P — an untyped program with a cluster of untyped predicates

let (p₁, ..., pₙ) be heads of a predicates in P

let S = ∅
    for each clause C = (pᵢ(A¹₁, ..., Aⁿᵢ) ← a₁, ..., aₘ) ∈ P do
        for i = 1 to mᵢ do
            if aᵢ is of the form Y = t then let S = S ∪ E₁(t, Y)
            if aᵢ is of the form q(Y₁, ..., Yₖ) where q ∈ Q then
                let τᵢ → σᵢ be the main type of q
                S = S ∪ E₂(τᵢ → σᵢ, (Y₁, ..., Yₖ))
            if aᵢ is of the form p(Y₁, ..., Yₖ) where p ∈ P then
                S = S ∪ {Yᵢ = Aᵢ} ∈ {1, ..., nᵢ}, where Aᵢ are head variables of p
        if S is unsatisfiable then return ‘type error’
    else
        let Θ = mgu(S)
        for i = 1 to n do
            let Aᵢ = (A¹ᵢ, ..., Aⁿᵢ)
            let φᵢ = shape(Θ(Aᵢ))
            let τᵢ be φᵢ in which every ⊤ is replaced by a new variable
            let θᵢ = {a := ⊥ | a ∈ var(τᵢ)}
            let σᵢ = θᵢ(τᵢ)
        return Guarantee((P, T, (τ₁, ..., τₙ), (φ₁, ..., φₙ), (σ₁, ..., σₙ))]

Algorithm 7.1: Type reconstruction algorithm

p(X, Y) :- X = [Y].

for assumptions (α, β) has best guarantee equal to (α ∩ list(β), β) but for assumptions
(list(α), β) the best guarantee is (list(α ∩ β), α ∩ β), which cannot be obtained by
applying a substitution to (α ∩ list(β), β). Types, for which $BT$ returns ⊤ also can
cause an assumption to be not sufficient. Consider the clause:

p(X, Y) :- X=[1|Y].

It has for assumptions (α, β) the result type equal to (α, β) but for assumptions
(list(α), list(β)) we obtain the better guarantees

(list(α ∩ (β ∪ int)), list(α ∩ β))

Definition 7.18. A full assumption τ is a type in which all variables are distinct. It
has either the form α or the form α ∩ F(τ₁, ..., τₙ) where each τᵢ is a full assumption.

We have the following lemma:

Lemma 7.19. Let $R = (Q, T, P)$ be a RT-task where $P$ is strongly typeable. Let
p₁, ..., pₙ be the predicates from P. Let τ₁, ..., τₙ be a full sufficient assumptions
for p₁, ..., pₙ. Let σ₁, ..., σₙ be the best guarantees of p₁, ..., pₙ for assumptions
7.2. Properties of Type Reconstruction Algorithm

% First clause
X = list(A1), Y=Z,

% Second clause
% atom X=[V|Vz]
X = list(A2),
T1 = A1, V=T1,
T2 = list(A2), Xs = list(A2),
% atom Y=[V|Vz]
Z = list(A3),
T3 = A1, V=T1,
T4 = list(A3), Vz = list(A3),
% atom append(Xs,Y,Zs)
X = Xs
Y = Y
Z = Zs

Figure 7.2: The set $S$ for the predicate append

$\tau_1, \ldots, \tau_n$. Let

$$T' = \{(p_1 : \tau_1 \rightarrow \sigma_1), \ldots, (p_n : \tau_n \rightarrow \sigma_n)\}$$

Then $T \cup T'$ is the main type of $P \cup Q$.

Proof. Let $T'' = \{(p_1 : \tau'_1 \rightarrow \sigma'_1), \ldots, (p_n : \tau'_n \rightarrow \sigma'_n)\}$ be the main type of $P$. Such type exists since $P$ as strongly typeable has a main ordinary type. Without any loss of generality we can assume that variables in $T'$ and in the type $T$ are different.

One can show that since each $\tau_i$ is a full assumption we have that there exist $\theta$ such that for each $i \in \{1, \ldots, n\}$

$$\vdash \tau_i \cap \tau'_i = \theta(\tau_i)$$

We have that for each $i$

$$\tau_i \rightarrow \sigma_i \sim \theta(\tau_i) \rightarrow \theta(\sigma_i)$$

$$\vdash \tau_i \cap \tau'_i \rightarrow \theta(\sigma_i)$$

$$\vdash \tau_i \cap \tau'_i \rightarrow \sigma_i \cap \sigma'_i \quad \text{since } T' \text{ is sufficient}$$

$$\vdash \tau'_i \rightarrow \sigma'_i \quad \text{after pruning variables from var}(\tau_i)$$

So, the main type is derivable from type $T'$. Types $T'$ and $T''$ are equal up to renaming and $T'$ is a main type of $P$.

Now, we state some auxiliary facts connected with the notion of sufficient assumptions.

**Lemma 7.20.** Let $\tau$ be a type, let $\sigma$ be an assumption type, and let $\theta$ be a substitution. Suppose that $\tau^*$ (see Section 4.2) does not appear in the types in the range of $BS(\tau, \sigma)$. Then we have

$$\vdash BS(\theta(\tau), \sigma) = \theta \circ BS(\tau, \sigma).$$

Proof. In Appendix.

133
Lemma 7.21. Let $t$ be a term, and $\Gamma$ be an environment. Let $\theta$ be a substitution. Moreover, suppose that for any subterm $t'$ of $t$ we have that
\begin{equation}
\text{if } BT(\Gamma, t') = \top \text{ then } BT(\theta(\Gamma), t') = \top
\end{equation}
Then $\theta(BT(\Gamma, t)) = BT(\theta(\Gamma), t)$.

Proof. In Appendix.

Lemma 7.22. If in the proof of $\Gamma, t : \tau \Rightarrow \Gamma'$, we have never used the rule (K₂), then for each substitution $\theta$
\begin{equation}
\theta(\Gamma), t : \theta(\tau) \Rightarrow \theta(\Gamma')
\end{equation}

Proof. We omit the proof which is very similar to the proof of Lemma 2.65.

Now, we analyze Algorithm 7.1. First we show that the types returned by this algorithm are good.

Lemma 7.23. Let $R = ((Q, T, P))$ be a RT-task. Suppose that for $R$ Algorithm 7.1 returns $T'$. Then $T'$ is a main type of $P$.

Proof. The proof is rather tedious and routine. The more detailed version of it reader can find in Appendix. Here we present the main idea. Let $S$ be the set of equations obtained in algorithm. Let $\Theta$ be the most general unifier of $S$. One can show that if $\Theta$ gives assumptions $\tau$ as was described in Algorithm 7.1 then, during execution of the reconstruction of guarantees algorithm, shapes of best types of all variables are compatible with shapes given by $\Theta$. Moreover, one can show that obtained assumptions are sufficient, since for every execution of ResultType we use the BS, $BT$ and $\Rightarrow$ in such a way that assumptions of Lemmas 7.20, 7.21, 7.22 are fulfilled. Then, the thesis follows by Lemma 7.19 and Theorem 5.1.

Theorem 7.1. Let $R = ((Q, T, P))$ be a RT-task. Suppose that $P$ is strongly typeable. Then Algorithm 7.1 finds a main type.

Proof. By Lemma 7.23 it is sufficient to prove that the algorithm finds a type (do not returns type error]. We sketch the proof that the set of equations $S$ in Algorithm 7.1 has a solution. Since in System $P$ the best guarantees always exist it is sufficient to establish the thesis.

$P$ is strongly typeable. We can assign to every program variable in $P$ its shape in the following way. For every clause $C$ we take its saturated extension $e(C)$. Let $\Gamma(C)$ be the result environment for $e(C)$ in $T$. We put $\Theta(X) = \text{shape}(\Gamma(C)(X))$. One can show that since $P$ is strongly typeable $\Theta$ is well defined.

We show how to enlarge the domain of $\Theta$ to all variables from $S$ and obtain the substitution $\Theta'$ for which $S$ is fulfilled. We define an environment:
\[ \Gamma = \{ X_i : \Theta(X_i) \mid X_i \text{ is a variable from program } P \}\]

First, notice that if $E(a)$ denotes the set of equations in $S$ connected with an atom $a$, then we have
\begin{equation}
\text{var}(E(a)) \cap \text{var}(E(a')) \subseteq \text{var}(P) \text{ for different atoms } a, a'
\end{equation}

So, for every atom in $P$ we can compute $\Theta'$ separately and then join them. In order to simplify this considerations we omit the renaming substitutions in operations $E_1, E_2$.

We consider three groups of equations in $S$. 

134
1. We take equations connected with atom \(( Y = t )\). Since \( p_i \) is strongly typeable we have that \( \Theta(Y) = \text{shape}(BT(\Gamma, t)) \). It suffices to prove the following statement:

   If \( \Theta(Y) = \text{shape}(BT(\Gamma, t)) \) then we can widen the domain of \( \Theta \) and obtain \( \Theta' \) for which \( \Theta'(E_1(Y, t)) \) is satisfiable'

   We prove it by induction on the structure of \( t \). If \( t \) is a variable then the thesis trivially holds. So, suppose that \( t = f(t_1, \ldots , t_n) \). Let \( f \) has a signature \( f : \sigma_1 \cdots \sigma_n \to \sigma \). We have, by definition

   \[
   E_1(Y, t) = \{ T = \sigma , T_1 = \sigma_1 , \ldots , T_n = \sigma_n \} \cup \bigcup_{i} E_1(t_i , T_i)
   \]

   Since \( p_i \) is strongly typeable we have that

   \[ \Theta(Y) = \text{shape}(BT(\Gamma, f(t_1, \ldots , t_n))) = \text{shape}(\theta(\sigma)) \]

   Let \( \Theta_0 \) be defined in a following way:

   \[
   \begin{align*}
   \Theta_0(X) &= \Theta(X) \text{ if } X \in \text{var}(P) \\
   \Theta_0(\alpha) &= \theta(\alpha) \text{ if } X \in \text{var}(\sigma) \\
   \Theta_0(T_i) &= \theta(\sigma_i) \text{ for } i \in \{ 1, \ldots , n \}
   \end{align*}
   \]

   It is easy to see that for \( \Theta_0 \) equations \( \{ T = \sigma , T_1 = \sigma_1 , \ldots , T_n = \sigma_n \} \) are satisfied.

   Since \( \Theta_0(T_i) = \text{shape}(BT(\Gamma, t_i)) \) we can apply the induction hypothesis and obtain for each \( i \in \{ 1, \ldots , n \} \) a substitution \( \Theta_i \) fulfilling \( E_1(t_i , T_i) \). By (7.4) we can join this substitutions and obtain the substitution

   \[ \Theta' = \Theta_0 \circ \Theta_1 \circ \cdots \circ \Theta_n \]

   for which \( E_1(Y, f(t_1, \ldots , t_n)) \) is valid.

2. The equation connected with the call \( q(Y_1, \ldots , Y_n) \). Suppose that the atom \( q(Y_1, \ldots , Y_n) \) is typed with the type \( \tau \to \sigma \). ResultEnvironment cannot produce ugly expressions \( \alpha \cap F(p_1, \ldots , p_n) \) and \( G(\zeta_1, \ldots , \zeta_n) \cap F(p_1, \ldots , p_n) \), since they do not disappear during run of ResultEnvironment. This violates conditions of strongly typedness. Therefore one can show that there is a substitution \( \theta \) for variables in \( \text{var}(\tau) \) such that \( E_2(t_1, \ldots , t_n, \tau \to \sigma) \) is valid for \( \Theta' = \theta \circ \Theta \).

3. The equation connected with the recursive call \( p_i(X_1, \ldots , X_n) \). Because recursive calls have the same shapes we can obtain \( \Theta' \) by unifying variables from \( \Theta(X_i) \) and \( \Theta(A_i) \) where \( A_1, \ldots , A_n \) are head variables for \( A \).

Now, let us analyze the complexity of algorithm 7.1. We have the following theorem:

**Theorem 7.2.** The algorithm 7.1 works in 2-EXPTIME.

**Proof.** Sketch. First we analyze the size of obtained assumptions. Let \( h(C) \) for a clause be the sum

\[
    h(C) = \sum_{t \text{ in clause } C} h(t)
\]

where \( h(t) \) denotes the height of the term \( t \). Let \( n(C) \) be the number of atoms in the body of \( C \).

We have assumptions for all previously typed predicates (predicates in a \( Q \)). Let

\[
    HQ = \max\{ h(\tau_q) \mid \tau_q \text{ is an assumption of } q \in Q \}
\]

135
Chapter 7. Type Reconstruction

One can show that if $\tau$ is an assumption returned by the algorithm then

$$
    h(\tau) \leq \max\{h(C) \mid C \text{ is a clause with the head predicate symbol } p\} \\
    + H Q \cdot \max\{n(C)\mid C \text{ is a clause with the head predicate symbol } p\}
$$

So, the size of assumptions is single exponential. Since assumptions $\tau_q$ are given by an user they are of size $O(N)$. When assumptions are found, algorithm starts computing guarantees for single exponential size of data which, by Theorem 5.2 gives double exponential complexity.

As was said at the beginning of this chapter, algorithm of type reconstruction can be used to reconstruct a directional type of an entire program, not only of one cluster. In this case clusters are analyzed in the order determined by $\gg$, i.e. in every step such cluster is analyzed whose clauses contain only calls of $p_1, \ldots, p_n$ and calls of the predicates of known type.

If we run the algorithm for the whole program, cluster by cluster, we obtain the double exponential size of assumptions and the whole algorithm will work in $\text{3-EXPTIME}$. It is illustrated by the following example.

**Example 7.9.** Consider the program:

- $q_0(X, \text{pair}(X, X))$.
- $q_1(X, Y) : \neg q_0(X, Z), q_0(Z, Y)$.
- $\ldots$
- $q_n(X, Y) : \neg q_{n-1}(X, Z), q_{n-1}(Z, Y)$.

The predicate $q_n$ has type of the height $2^n$ with $2^{2^n}$ type variables. This size is connected with the fact that our system attempts to describe types of a very big terms from the success set of this program.
Chapter 8

Extensions

8.1 Error Prevention

Directional types express dependence between input and output of a predicate. Therefore, we can use directional types to obtain information about types that terms may get during execution of a program. For example, after executing \texttt{member(X, [1, 2])} variable \texttt{X} is of the type \texttt{int}, which can be proved in our system. Information about types of variables can be useful in detecting errors.

**Definition 8.1.** A program \( P \) is \textit{typeable} if there exists a directional type \( T \) of a program such that, for each predicate \( p \) of \( P \), \( T \) contains at least one element of the form \( (p : \tau \to \sigma) \), and \( P \) is well-typed with respect to \( T \).

**Definition 8.2.** Let \( P \) be a program which is well-typed with respect to \( T \), and let \( p \) be a predicate not belonging to \( P \). The predicate \( p \) is \textit{typeable in the context of \( P \) and \( T \)} iff there is an extension \( T' \) of \( T \) containing at least one type for \( p \) such that \( P \cup \{p\} \) is well-typed with respect to \( T' \).

A goal \( \langle : a_1, \ldots, a_n \rangle \) is \textit{typeable in the context of \( P \) and \( T \)} iff \( \langle p : a_1, \ldots, a_n \rangle \) is typeable in the context of \( P \) and \( T \), where \( p \) is a 0-ary predicate symbol not occurring in \( P \).

We omit the phrase "in the context of \( P \) and \( T \)" whenever it does not lead to a confusion.

**Example 8.1.** Consider the following program

\[
q ::= 1 < X.
\]

We assume that arguments of the predefined predicate \(<\) should be of the type \texttt{int}, since otherwise a run-time error will occur. Therefore, we demand that \(<\) has the only type \((\text{int}, \text{int}) \to (\text{int}, \text{int})\). This type cannot be used when analyzing the predicate \( q \), since \( X \) is not of the type \texttt{int}. Hence \( q \) is not typeable.

**Example 8.2.** One can prove that program

\[
r(X,Y) ::= \text{unif}(X,Y), X < Y.
\]

is well typed with respect to

\[
\{(r : (\text{int}, \text{int}) \to (\text{int}, \text{int})), (r : (\top, \text{int}) \to (\text{int}, \text{int})), (r : (\text{int}, \top) \to (\text{int}, \text{int}))\}.
\]
Thus the following goals are typeable
\[\text{:- } r(1, 2).\]
\[\text{:- } r(X, 5).\]
\[\text{:- } r(2, Y).\]

while the goal
\[\text{:- } r(X, Y).\]

is not.

The examples above show how directional type of a program can restrict possibility of applying a predicate to a given term. Such restrictions could be useful because: (1) real programs cause run time errors when wrong input terms are invoked, (2) sometimes applying a predicate to given arguments do not cause any error, but programmer recognizes such application as senseless. For example rather senseless goal
\[\text{append([], 1, 1)}\]
evaluates successfully.

The main type of a program is a compact description of a family of types. It does not, however, restrict possibility of using predicates. In order to add such restrictions to the main types we define pattern of types.

**Definition 8.3.** A pair
\[Q = (\tau \rightarrow \sigma, (a_1 \leq b_1, \ldots, a_n \leq b_2))\]
is a pattern of types if \(\tau\) is a directional type, and, for each \(i \in \{1, \ldots, n\}\), \(b_i\) is an atomic types, and \(a_i\) is a type constructed using only operators \(\cap, \cup\) and variables belonging to \(\text{var}(\tau)\).

For a pattern \(Q = (\tau \rightarrow \sigma, (a_1 \leq b_1, \ldots, a_n \leq b_2))\) we define the set \(G(Q)\) of types generated from \(Q\):
\[G(Q) = \{\theta(\nabla_A(\tau)) \rightarrow \theta(\nabla_A(\sigma)) \mid \theta\text{ is a type substitution, } A \subseteq V, \theta(\nabla_A(a_i)) \leq \theta(\nabla_A(b_i)), \text{ for } i \in \{1, \ldots, n\}\} \]

**Definition 8.4.** A set \(S\) of the form
\[\{(p : Q) \mid p\text{ is a predicate name, } Q\text{ is a pattern of types}\}\]
is called a pattern of types of a program.

We define \(G(S)\) as follows.
\[G(S) = \{(p : \tau \rightarrow \sigma) \mid (p : Q) \in S, (\tau \rightarrow \sigma) \in G(Q)\}\]
The program \(P\) is well-typed with respect to \(S\) iff it is well-typed with respect to \(G(A)\).

**Definition 8.5.** \(Q\) is the most general pattern of a program \(P\) iff \(P\) is well-typed with respect to \(Q\), and, for any pattern \(Q'\) such that \(P\) is well-typed with respect to \(Q'\), \(G(Q) \supseteq G(Q')\).

The idea of deal with pattern of types is as follows. The user can provide pattern of types for some predicates. The conditions in these patterns express intended meaning of a predicate. Moreover, for some standard predicates (like arithmetic predicates) pattern of types are provided. Now, for the whole program, the system reconstructs...

138
the most general pattern. There is an algorithm which reconstruct the most general
general pattern of types working in the exponential time.

Without any loss of generality we assume that directional types used in pattern
do not contain atomic types. Thus instead of

\[(\text{int, int}) \rightarrow (\text{int, int})\]

we write

\[(\alpha, \beta) \rightarrow (\alpha, \beta), \quad \alpha \leq \text{int}, \beta \leq \text{int} .\]

**Example 8.3.** Assume that

\[(\alpha, \beta) \rightarrow (\alpha, \beta), \quad \alpha \leq \text{int}, \beta \leq \text{int} .\]

is the pattern of types of predicate \(<\). The condition described by this pattern can
be expressed in the equivalent form \(\alpha \cup \beta \leq \text{int}\) as well. Consider the program of
Example 8.2. The system reconstructs the following pattern for \(r\):

\[(\alpha, \beta) \rightarrow (\alpha \cap \beta, \alpha \cap \beta), \quad \alpha \cap \beta \leq \text{int}\]

Note that an ordering of atoms in a clause is meaningful. If we change the ordering
of atoms we can obtain different patterns. Consider a predicate \(r'\):

\[ r'(X,Y) :- X < Y, \ \text{unif}(X,Y). \]

This predicate have the same main type as \(r\), but its pattern of types have the form

\[(\alpha, \beta) \rightarrow (\alpha \cap \beta, \alpha \cap \beta), \quad \alpha \leq \text{int}, \beta \leq \text{int} \]

Thus, the goal \(:- r(2,X)\) is typeable, while the goal \(:- r'(2,X)\) is not . \(\square\)

## 8.2 Connections between Variables

Unification plays a very important role in logic programming languages. A type
system can make use of the special meaning of unification. When variables \(X, Y\) of
types \(\tau, \sigma\) respectively are unified, their types are constrained to \(\tau \cap \sigma\) which can be
proved in our system. Then, if the type of \(X\) will be further constrained to type \(\rho\),
the type of \(Y\) will be constrained to \(\rho\) as well. So far this feature is not reflected in
our type systems. Consider the program:

\[ p(X,Y) :- X = Y, \ X = 1. \]

We can prove that \(p\) has the type \((\tau, \tau) \rightarrow (\text{int, int})\), but we cannot prove that it
has the type \((\text{int, int}) \rightarrow (\text{int, int})\), though both types are semantically correct. The
similar situation is in directional type system of Aiken and Lakshman. That is
because they represent types by regular sets of trees. These sets cannot express the
type \(\{ (t,t) \mid t \text{ is any ground term} \}\), which is necessary to prove that \(p\) has the type
\((\tau, \tau) \rightarrow (\text{int, int})\).

The example above describes very simple connection between variables. Such
connections can be much more subtle. In this section we present a stronger version of
our type system, called System C, which take into consideration connections between
variables. In System C stronger types can be proved. For instance both types for
predicate \(p\) mentioned above are provable.

The algorithms of type checking and type reconstruction can be easily adapted
to work in system C. However, the complexity of these algorithms will probably be
higher.
8.2.1 Types of Terms
First, we extend the language of types.

Definition 8.6 (types in System C). The set of types is the least set $T$ satisfying the following conditions:

(i) $\top, \bot \in T$
(ii) $\alpha \in T$ where $\alpha \in V_T$
(iii) $F(\tau_1, \ldots, \tau_n) \in T$ where $\tau_i \in T, F \in F_T$, $F$ has arity $n$
(iv) $\tau \cap \sigma, \tau \cup \sigma \in T$ where $\tau, \sigma \in T$
(v) $f^{-1}_k(\tau) \in T$ where $\tau \in T, f \in K$ has arity $n$ and $1 \leq k \leq n$
(vi) $\alpha_X \in T$ where $X \in V$

As we can see, points (i)-(iv) are exactly the same as in Definition 2.7. The intuitive meaning of a type of the form $\alpha_X$ is “actual type of variable $X$”. Now, let us consider point (v). If $\tau$ is a type, and pair is a term constructor with signature $\alpha \ast \beta \rightarrow \text{prod}(\alpha, \beta)$, then both $\text{pair}^{-1}_1(\tau)$ and $\text{pair}^{-1}_2(\tau)$ are correctly constructed types. Moreover, we expect that $\text{pair}^{-1}_1(\text{prod}(\tau, \sigma))$ should be equal to $\tau$, and $\text{pair}^{-2}_1(\text{prod}(\tau, \sigma))$ should equal to $\sigma$.

8.2.2 Axioms and Rules
We need some new equality axioms to deal with new types.

\[(\text{Ax}17)\] $\vdash f^{-1}_k(\tau) \cup f^{-1}_k(\tau') = f^{-1}_k(\tau \cup \tau')$

\[(\text{Ax}18)\] $\vdash f^{-1}_k(\theta(\tau)) \leq \theta(\tau_k)$ if $f: \tau_1 \ast \cdots \ast \tau_n \rightarrow \tau$

If ‘pair’ has a signature $\alpha \ast \beta \rightarrow \text{prod}(\alpha, \beta)$ then, for $\theta = \{\alpha/\tau, \beta/\sigma\}$, axiom (Ax18) proves that $\vdash \text{pair}^{-1}_1(\text{prod}(\tau, \sigma)) \leq \tau$. The inequality in (Ax18) can not be replaced with equality. Indeed, $\text{prod}(\bot, \text{int})$ is the empty set, thus we can expect that $\text{pair}^{-1}_1(\text{prod}(\bot, \text{int}))$ will be the empty set which is a proper subset of int.

Furthermore, in the system without pruning we have additional axioms:

\[(\text{Ax}19)\] $\vdash f^{-1}_k(\sigma) = \bot$ for each $f, \sigma$ such that $f : \tau_1 \ast \cdots \ast \tau_n \rightarrow \tau$, $\tau$ is a polymorphic type, and $\text{head}(\sigma) \neq \text{head}(\tau)$

\[(\text{Ax}20)\] $\vdash f^{-1}_k(\sigma) = \bot$ for each $f, \sigma$ such that $f : \tau_1 \ast \cdots \ast \tau_n \rightarrow \tau$, $\tau$ is an atomic type, and $\text{and}(\tau, \sigma) = \bot$

We give also an additional rule for term typing.

\[(T_{\gamma})\] $\Gamma \vdash X : \alpha_X$

It express that variable $X$ always has the type $\alpha_X$. Finally, we replace rule $(K_9)$ by $(K'_{9})$, and add a new rule.

\[(K'_{9})\] $\Gamma, (X : \tau) \Rightarrow \Gamma[\alpha_X := \alpha_X \cap \tau]$
where $\Gamma[\alpha_X := \sigma]$ denotes the environment obtained from $\Gamma$ by substituting $\sigma$ for $\alpha_X$.

\[ (K_{10}) \quad \Gamma, (t_i : f^{-1}_i(\tau)) \Rightarrow \Gamma_i \quad (1 \leq i \leq n) \]
\[ \Gamma, (f(t_1, \ldots, t_n) : \tau) \Rightarrow \Gamma_1 \cap \cdots \cap \Gamma_n \]
where $\tau$ has either the form $\alpha_X$ or $g_k^{-1}(\sigma)$

We also replace rule $(P_5)$ by $(P^\alpha_5)$

\[ (P^\alpha_5) \quad \Gamma_0 = \{ X : \alpha_X \mid X \in \text{var}(t_1, \ldots, t_n, B) \}, \]
\[ \Gamma_0, (t_1 : \tau_1, \ldots, t_n : \tau_n) \Rightarrow \Gamma_1, \]
\[ \vdash \text{InferFromAtoms}(T, \Gamma_1, B, \Gamma_2), \quad \Gamma_2 \vdash t_1 : \sigma_1, \ldots, \Gamma_2 \vdash t_n : \sigma_n \]

**Example 8.4.** Recall that unification ($\approx$) has type $(\alpha, \beta) \rightarrow (\alpha \cap \beta, \alpha \cap \beta)$. Consider the program.

\[ p(X, Y) : - X=Y, \; X=1. \]

We are going to check that this program is well-typed with respect to

\[ T = \{ \approx : (\alpha, \beta) \rightarrow (\alpha \cap \beta, \alpha \cap \beta), \; (p : (\top, \top) \rightarrow (\text{int, int})) \}. \]

It suffices to show that

\[ (8.2) \quad \vdash \text{ClauseHasType}(T, (p(X, Y) : - X=Y, \; X=1), ((\top, \top) \rightarrow (\text{int, int}))). \]

To illustrate how the new rules work we give a detailed proof of this statement. By $(P^\alpha_5)$, $(8.2)$ holds if

\[ \Gamma_0 = \{ X : \alpha_X, Y : \alpha_Y \}, \]
\[ \Gamma_0, (X : \top, Y : \top) \Rightarrow \Gamma_1, \]
\[ (8.3) \quad \vdash \text{InferFromAtoms}(T, \Gamma_1, (X=Y, \; X=1), \Gamma_4), \]
\[ (8.4) \quad \Gamma_4 \vdash X : \text{int}, \; \Gamma_4 \vdash Y : \text{int} \]

One can see that, by $(K_3)$, we have $\Gamma_1 = \Gamma_0 = \{ X : \alpha_X, Y : \alpha_Y \}$. To get $(8.3)$ we use rule $(P_2)$ with directional type $(\alpha, \beta) \rightarrow (\alpha \cap \beta, \alpha \cap \beta)$ for $\approx$, and substitution $\theta = \{ \alpha/\alpha_X, \alpha/\alpha_Y \}$. Thus $(8.3)$ holds if

\[ (8.5) \quad \Gamma_1 \vdash X : \alpha_X, \; \Gamma_1 \vdash Y : \alpha_Y \]
\[ (8.6) \quad \Gamma_1, (X : \alpha_X \cap \alpha_Y, Y : \alpha_X \cap \alpha_Y) \Rightarrow \Gamma_2 \]
\[ (8.7) \quad \vdash \text{InferFromAtoms}(T, \Gamma_2, (X=Y), \Gamma_4) \]

(8.5) obviously holds. One can check that $(8.6)$ holds for $\Gamma_2 = \{ X : \alpha_X \cap \alpha_X \cap \alpha_Y, Y : \alpha_Y \cap \alpha_X \cap \alpha_Y \} \sim \{ X : \alpha_X \cap \alpha_Y, Y : \alpha_X \cap \alpha_Y \}$. To get $(8.7)$ we use rule $(P_2)$ with directional type $(\alpha, \beta) \rightarrow (\alpha \cap \beta, \alpha \cap \beta)$ for $\approx$, and substitution $\theta = \{ \alpha/\top, \alpha/\text{int} \}$. Thus $(8.7)$ holds if

\[ (8.8) \quad \Gamma_2 \vdash X : \top, \; \Gamma_2 \vdash 1 : \text{int} \]
\[ (8.9) \quad \Gamma_2, (X : \top \cap \text{int}, 1 : \top \cap \text{int}) \Rightarrow \Gamma_3 \]
\[ (8.10) \quad \vdash \text{InferFromAtoms}(T, \Gamma_3, (\top), \Gamma_4) \]

(8.8) holds obviously. By $(K_1)$, $\Gamma_3 = \Gamma^3_3 \cap \Gamma^3_3$, where $\Gamma_2, (X : \top \cap \text{int}) \Rightarrow \Gamma^3_3$, and
\[ \Gamma_2, (1 : \top \cap \text{int}) \Rightarrow \Gamma^3_3. \]

One can check that $\Gamma^3_3 \sim \Gamma^3_3$, and thus $\Gamma^3_3 \sim \Gamma^3_3$. To compute
Γ₂⁻¹ use rule \((K^2)\) which substitute each occurrence of \(α_X\) in \(Γ₂\) by \(α_X \cap (\top \cap \text{int})\). Recall that \(Γ₂ \sim \{X : α_X \cap α_Y, Y : α_X \cap α_Y\}\). Hence
\[
Γ₃ \sim Γ₃⁻¹ \sim \{X : α_X \cap α_Y \cap \text{int}, Y : α_X \cap α_Y \cap \text{int}\}.
\]
By \((P₁)\), (8.10) holds for \(Γ₄ = Γ₃\). Since we have showed that (8.8)–(8.10) hold, (8.7) holds, and thus (8.3) holds. Since \(Γ₂ \sim \{X : α_X \cap α_Y \cap \text{int}, Y : α_X \cap α_Y \cap \text{int}\}\), (8.4) holds, which completes the proof.

8.2.3 Properties of System C

**Lemma 8.7.** We can prove in System C that
\[
\vdash f_k⁻¹(τ) \cap f_k⁻¹(τ') \geq f_k⁻¹(τ \cap τ')
\]

**Proof.**
\[
\vdash (f_k⁻¹(τ) \cap f_k⁻¹(τ')) \cup f_k⁻¹(τ \cap τ')
= (f_k⁻¹(τ) \cup f_k⁻¹(τ')) \cap (f_k⁻¹(τ') \cup f_k⁻¹(τ \cap τ')) \quad \text{by (Ax10)}
= f_k⁻¹(τ \cup (τ \cap τ')) \cap f_k⁻¹(τ' \cup (τ \cap τ')) \quad \text{by (Ax17)}
= f_k⁻¹(τ) \cap f_k⁻¹(τ') \quad \text{by (Ax9)}
\]

which by Lemma (2.20) finishes the proof.

All the facts stated in Chapter 2 remain true in System C. Most proofs are valid without any changes. Some modifications are presented in Appendix A.6.1.

8.2.4 Semantics

Now, we extend the definition of the semantic function \([\cdot]_v\). The interpretation of \(α_X\) is not independent of the structure, thus we will write \([τ]_v\) to denote the interpretation of \(τ\) in \(M_v\).

\[
\text{(8.11)} \quad [α_X]_v = \{v(X)\}
\]

\[
\text{(8.12)} \quad [f_k⁻¹(τ)]_v = \{t \mid ∃t₁ \ldots t_n (f(t₁, \ldots, t_n) ∈ [τ]_v \wedge t = t_k)\}
\]

Semantics of all the others types is unchanged with the exception that we write \([\cdot]_v\) instead of \([\cdot]\). The proof of the correctness of the new definition can be adopted. A supplement to the proofs of soundness of System C is given in Appendix A.6.2.

8.2.5 Pruning Theorem

First, we define pruning for types of the new form. For any set \(A\) of variables we have
\[
∇_A(α_X) = α_X
\]
\[
∇_A(f_k⁻¹(τ)) = \begin{cases} 
\top & \text{if } \vdash ∇_A(τ) = \top \\
\text{otherwise} & \end{cases}
\]

The Pruning Theorem remains true for System C without rules (Ax19) and (Ax20), however the proofs have to be completed. Necessary completions are given in Appendix A.6.3.
Chapter 9

Conclusions

The type language presented in Chapter 2 together with the type inference rules form a basic type system, called System B. Polymorphic directional types well express declarative and procedural properties of logic programs. The Soundness Theorem of Chapter 3 gives a proof that if we prove in System B that a predicate $p$ has a directional type $(\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)$, and $(\tau'_1, \ldots, \tau'_n) \rightarrow (\sigma'_1, \ldots, \sigma'_n)$ is a ground instance of $(\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)$, then any terms $t_1, \ldots, t_n$ of types $\tau'_1, \ldots, \tau'_n$, respectively, after execution of $p(t_1, \ldots, t_n)$ have types $\sigma'_1, \ldots, \sigma'_n$, respectively. Such a notion of polymorphic directional types is a natural generalization of the notion of non-polymorphic directional types presented by A. Aiken and T. K. Lakshman in [1].

We present an algorithm which checks whether or not the program is well typed with respect to a given directional type. This algorithm for discriminative types works in EXPTIME. Moreover, it can be applied to a larger class of programs, and we believe that its complexity will remain single exponential. We present also an algorithm which for a given program and assumptions reconstructs a guarantee. It can be seen as a reconstruction of types when a goal (logical query) is given. This algorithm for discriminative types also works in EXPTIME. We have given lower bounds for problems of type checking and reconstruction of guarantees. We showed $\Sigma^p_2$ hardness of type checking and EXPTIME completeness of reconstruction of guarantees.

The pair of a type checker and a program which reconstructs guarantee can serve a basis of a program verification and analysis system.

There is a price to pay for expressiveness of the system of directional types. A predicate can have many useful directional types, which complicates the problem of type reconstruction. It is not clear how to automatically identify interesting directional types of a predicate. Reconstruction of all directional types of a predicate also is not a good solution, since a predicate can have exponentially many directional types. The ordinary notion of most general types (from functional languages or from the type system of Mycroft-O’Keefe) can not be applied, since the standard notion of derivation, based on polymorphism (and subtyping) does not suffice. Therefore, we have proposed in Chapter 6 a version of our system, called System P with a new operation of pruning. The extended notion of derivation based on pruning, substitution and subtyping allows to define the notion of the main type. The main type of a predicate describes the family of all proper types of this predicate. Although some of improper types of a predicate cannot be obtained from the main type, we have provided an additional method which allows to get them. The existence of main types is very useful. For the system with main types, parts of a program (library) can be typed independently. There is no need to know how predicates will be used, there is no need to decide which types are interesting.

All algorithms developed for system B have their versions for System P. We also
provide in Chapter 7 a type reconstruction algorithm. We define a class of strongly
typeable programs. We believe that programs written in a good style belong to this
class. For strongly typeable programs, the type reconstruction algorithm finds the
main type.

We believe that there is a strong relation between our type reconstruction algo-
rithm and types reconstruction algorithm for the Mycroft-O’Keefe type system. We
conjecture that if the type reconstruction algorithm for the Mycroft-O’Keefe type
system finds the most general type for a program, then this program has the main
type which can be found by our algorithm.

In Chapter 8 we have presented two extensions of our type system. First, we added
a simple error-prevention mechanism based on types given by the type checking and
the type reconstruction algorithms.

Second, we have described a version of our system, called System C. In this system
we use information about connections between variables to get better types. As a
result, for some predicates, ground instances of types inferred in System C can be
stronger than types inferred in the system of Aiken and Lakshman.

We believe that the type system presented in this thesis can be very useful in
practice. This system does not need any changes in standard SLD resolution mecha-
nism. A user should provide only information about signatures of term constructors,
he does not need to provide any information about predicate types. The practice of
programming in functional languages shows that providing signatures is not inconve-
nient, especially when compared with benefits which are given by types. And finally,
our system is not too restrictive. A programmer accustomed to work with untyped
languages will not feel very constrained by our system.

Future Works An obvious task is to implement our system. We believe that large
complexity of algorithms described in the thesis has only theoretical meaning, and
for real programs the type checking and type reconstruction algorithms should work
sufficiently fast to be useful in practice.

The other objective is to complete the proof of the conjecture stated above, that
if the type reconstruction algorithm for the Mycroft-O’Keefe type system finds the
most general type for a program, then this program has the main type which can be
found by the reconstruction algorithm described in chapter 7.

We are also going to generalize the type reconstruction algorithm to work not
only with strongly typeable programs.
Part III

Appendixes
Appendix A

Proofs

A.1 Proofs for Chapter 2

Lemma 2.29. For a variable $X$ and an environment $\Gamma$,
(a) $\Gamma \vdash X : \Gamma(X)$,
(b) if $\Gamma \vdash X : \tau'$ then $\vdash \Gamma(X) \leq \tau'$.

Proof. Let $\tau = \Gamma(X)$. Assume that $\text{false} \in \Gamma$. Then $\tau = \bot$ and both (a) and (b) hold. Now, assume that $\text{false} \notin \Gamma$. Then $(X : \tau) \in \Gamma$. We can use rule $(T_3)$ to prove (a). In order to prove (b) assume that $\Gamma \vdash X : \tau'$. We shall show that $\vdash \tau \leq \tau'$ by induction on the structure of the proof of $\Gamma \vdash X : \tau'$.

- Assume that $\Gamma \vdash X : \top$ was obtained using $(T_1)$. Then obviously $\tau \leq \top$ (it is true for each type $\tau$).
- $(T_2)$ cannot be used since $\text{false} \notin \Gamma$.
- Now, assume that $\Gamma \cup \{X : \tau'\} \vdash X : \tau'$ was obtained using $(T_3)$. Then, by the definition of environment, $\tau' = \tau$, and therefore $\vdash \tau \leq \tau'$.
- Assume that $\Gamma \vdash X : \tau'$ was obtained from $\Gamma \vdash X : \tau''$ and $\vdash \tau'' \leq \tau'$ using $(T_4)$. By the inductive hypothesis, $\vdash \tau \leq \tau''$ and because $\leq$ is transitive (Lemma 2.18), $\vdash \tau \leq \tau'$.
- Assume that $\Gamma \vdash X : \tau_1 \cap \ldots \cap \tau_n$ was obtained from $\Gamma \vdash X : \tau_i$ for $i \in \{1, \ldots, n\}$ using $(T_5)$. By the inductive hypothesis, $\vdash \tau \leq \tau_i$. It means that $\vdash \tau \cap \tau_i = \tau$. Hence

$$\vdash (\tau_1 \cap \ldots \cap \tau_n) \cap \tau = (\tau_1 \cap \tau) \cap \ldots \cap (\tau_n \cap \tau) = \tau \cap \ldots \cap \tau = \tau$$

which shows that $\vdash \tau \leq \tau_1 \cap \ldots \cap \tau_n$.
- $(T_6)$ cannot be used because $X \neq f(t_1, \ldots , t_n)$.

\[ \square \]

Lemma 2.34. Let $\Gamma_1$ and $\Gamma_2$ are environments. Then

$\Gamma_1 \leq \Gamma_2$ iff for each $X \in V$, $\vdash \Gamma_1(X) \leq \Gamma_2(X)$.

Proof. ($\Rightarrow$) First, we will show that if $\Gamma_1 \leq \Gamma_2$ then, for each $X \in V$, $\vdash \Gamma_1(X) \leq \Gamma_2(X)$. Assume that $\Gamma_1 \leq \Gamma_2$ holds. Let $X \in V$. $\tau_1 = \Gamma_1(X)$ and $\tau_2 = \Gamma_2(X)$. From Lemma 2.29 it follows that $\Gamma_2 \vdash X : \tau_2$. Since $\Gamma_1 \leq \Gamma_2$, we have $\Gamma_1 \vdash X : \tau_2$. By Lemma 2.29, $\tau_1 \leq \tau_2$ which finishes the proof that for any variable $X$, $\vdash \Gamma_1(X) \leq \Gamma_2(X)$.

$\Gamma_1 \leq \Gamma_2$.
Appendix A. Proofs

(⇐) Now, we will show that if \( \vdash \Gamma_1(X) \leq \Gamma_2(X) \), for each \( X \in \mathcal{V}_T \), then \( \Gamma_1 \leq \Gamma_2 \). It suffices to show that \( \Gamma_2 \vdash t : \tau \) implies \( \Gamma_1 \vdash t : \tau \). We shall proceed by induction on the structure of the proof of \( \Gamma_2 \vdash t : \tau \).

- Assume that \( \Gamma_2 \vdash t : \top \) was obtained using (\( T_1 \)). We use the same rule to prove \( \Gamma_1 \vdash t : \top \).
- Assume that \( \Gamma_2 \vdash t : \bot \), where \( \text{false} \in \Gamma_2 \), was obtained using (\( T_2 \)). Since an environment is a finite set of formulas, thus there exists a variable \( Y \) not occurring in \( \Gamma_1 \). By Definition 2.28, \( \Gamma_2(Y) = \bot \), and hence \( \Gamma_1(Y) = \bot \). Since \( (Y : \bot) \notin \Gamma_1 \), we have \( \text{false} \in \Gamma_1 \).
- Assume that \( \Gamma_2 \vdash X : \tau \) was obtained using (\( T_3 \)), where \( (X : \tau) \in \Gamma_2 \). Then \( \Gamma_2(X) = \tau \). We use rule (\( T_3 \)) to obtain \( \Gamma_1 \vdash X : \Gamma_1(X) \) and, because \( \vdash \Gamma_1(X) \leq \Gamma_2(X) = \tau \), we can use rule (\( T_3 \)), to obtain \( \Gamma_1 \vdash X : \tau \).
- Assume that \( \Gamma_2 \vdash t : \tau \) was obtained from \( \Gamma_2 \vdash t : \tau' \) and \( \tau' \leq \tau \) using (\( T_4 \)). By the inductive hypothesis, \( \Gamma_1 \vdash t : \tau' \), so we use the same rule to prove that \( \Gamma_1 \vdash t : \tau \).
- Assume that \( \Gamma_2 \vdash t : \tau_1 \cap \cdots \cap \tau_n \) was obtained from \( \Gamma_2 \vdash t : \tau_i \), for \( i \in \{1, \ldots, n\} \), using (\( T_5 \)). By the inductive hypothesis, \( \Gamma_1 \vdash t : \tau_i \), for \( i \in \{1, \ldots, n\} \), and thus we can use the same rule to prove \( \Gamma_1 \vdash t : \tau_1 \cap \cdots \cap \tau_n \).
- Assume that \( f : \tau_1 \ast \cdots \ast \tau_n \rightarrow \tau \), and \( \Gamma_2 \vdash f(t_1, \ldots, t_n) : \theta(\tau) \) was obtained from \( \Gamma_2 \vdash t_i : \theta(\tau_i) \), for \( i \in \{1, \ldots, n\} \), using (\( T_6 \)). By the inductive hypothesis, \( \Gamma_1 \vdash t_i : \theta(\tau_i) \), for \( i \in \{1, \ldots, n\} \), so we can use the same rule to prove \( \Gamma_1 \vdash f(t_1, \ldots, t_n) : \theta(\tau) \).

\( \square \)

Lemma 2.43. For environments \( \Gamma \) and \( \Gamma' \) the following conditions holds for each variable \( X \):

- (a) \( (\Gamma \cap \Gamma')(X) = \Gamma(X) \cap \Gamma'(X) \).
- (b) \( (\Gamma \cup \Gamma')(X) = \Gamma(X) \cup \Gamma'(X) \).

Proof. We consider four cases:

- \( \text{false} \in \Gamma \) and \( \text{false} \in \Gamma' \). Then \( \Gamma \cup \Gamma' = \Gamma \cap \Gamma' = \{\text{false}\} \) and thus

\[
\begin{align*}
\Gamma(X) &= \bot, & \Gamma'(X) &= \bot, \\
(\Gamma \cup \Gamma')(X) &= \bot, & (\Gamma \cap \Gamma')(X) &= \bot \\
\end{align*}
\]

and finally

\[
\Gamma(X) \cup \Gamma'(X) = \bot \\
\Gamma(X) \cap \Gamma'(X) = \bot
\]

which shows (a) and (b).

- \( \text{false} \in \Gamma \) and \( \text{false} \notin \Gamma' \). Then \( \Gamma \cap \Gamma' = \{\text{false}\} \). We have \( \Gamma(X) = \bot, (\Gamma \cap \Gamma')(X) = \bot \), and \( \Gamma(X) \cap \Gamma'(X) = \bot \) which shows (a).

Now, \( \Gamma \cup \Gamma' = \Gamma' \), so \( (\Gamma \cup \Gamma')(X) = \Gamma'(X) \) and

\[
\Gamma(X) \cup \Gamma'(X) = \bot \cup \Gamma'(X) = \Gamma'(X)
\]

which shows (b).

- When \( \text{false} \notin \Gamma \) and \( \text{false} \notin \Gamma' \) the proof goes very similarly as in the previous case.
• false \notin \Gamma \text{ and false } \notin \Gamma'.

Then
\[
\begin{align*}
\Gamma \cap \Gamma' &= \{(X : \tau) \mid \tau = \Gamma(X) \cap \Gamma'(X) \land \tau \neq \top\}, \\
\Gamma \cup \Gamma' &= \{(X : \tau) \mid \tau = \Gamma(X) \cup \Gamma'(X) \land \tau \neq \top\}
\end{align*}
\]

which, by Definition 2.28, gives the thesis.

\hspace{1cm}\square

**Lemma 2.50.** If \( \vdash \tau_1 = \tau_2, \Gamma_1 \sim \Gamma_2, \Gamma_1, (t : \tau_1) \Rightarrow \Gamma'_1 \text{ and } \Gamma_2, (t : \tau_2) \Rightarrow \Gamma'_2, \text{ then } \Gamma'_1 \sim \Gamma'_2. \)

**Proof.** We proceed by induction on the structure of the term \( t. \) Let \((*)\) denote the proof of \( \Gamma_1, (t : \tau_1) \Rightarrow \Gamma'_1, \) and \((**)\) denote the proof of \( \Gamma_2, (t : \tau_2) \Rightarrow \Gamma'_2. \) Suppose that \( t = X \) is a variable. Then rule \((K_9)\) must be used in \((*)\) and \((**)\). So we have
\[
\begin{align*}
\Gamma'_1 &= \Gamma_1 \cap \{X : \tau_1\}, \\
\Gamma'_2 &= \Gamma_2 \cap \{X : \tau_2\}.
\end{align*}
\]

We have \( \Gamma'_1(X) = \tau_1 \cap \Gamma_1(X) \) and \( \Gamma'_2(X) = \tau_2 \cap \Gamma_2(X). \) Now, \( \Gamma_1 \sim \Gamma_2 \) implies that \( \vdash \tau_1 = \tau_2. \) Hence, by \((\text{Ax3}),\)
\[
\vdash \tau_1 \cap \Gamma_1(X) = \tau_2 \cap \Gamma_2(X).
\]

Thus \( \vdash \Gamma'_1(X) = \Gamma'_2(X). \) Since, for any variable \( Y \neq X, \Gamma'_1(Y) = \Gamma_1(Y) \) \( \Gamma'_2(Y) = \Gamma_2(Y), \) and \( \Gamma_1 \sim \Gamma_2, \) then \( \vdash \Gamma'_1(Y) = \Gamma'_2(Y) \) and therefore \( \Gamma'_1 \sim \Gamma'_2. \)

Now, suppose that \( t = f(t_1, \ldots, t_n). \) The inductive hypothesis states that for each \( i \in \{1, \ldots, n\}, \) for any environments \( \Delta_1 \) and \( \Delta_2, \) for any types \( \sigma_1 \) and \( \sigma_2, \)
\[
\begin{align*}
&\text{(A.1)} \quad \begin{cases} \\
\vdash \sigma_1 = \sigma_2 \\
\Delta_1 \sim \Delta_2 \\
\Delta_1, (t_i : \sigma_1) \Rightarrow \Delta'_1 \\
\Delta_2, (t_i : \sigma_2) \Rightarrow \Delta'_2
\end{cases} \\
&\quad \implies \Delta'_1 \sim \Delta'_2.
\end{align*}
\]

We shall proceed by induction on the structure of the proof of \( \vdash \tau_1 = \tau_2. \) Cases are denoted by axioms which are used in the proof of \( \vdash \tau_1 = \tau_2. \)

\( \text{(Ax1)} \) Obvious (\( \tau_1 \) and \( \tau_2 \) are exactly the same).

\( \text{(Ax2)} \) We have \( \vdash F(\tau_1, \ldots, \tau_k) = F(\tau'_1, \ldots, \tau'_k) \) with assumptions \( \vdash \tau_i = \tau'_i \) for \( i \in \{1, \ldots, k\}. \) To prove that
\[
\Gamma_1, (t : F(\tau_1, \ldots, \tau_k)) \Rightarrow \Gamma_1 \quad \text{and} \quad \Gamma_2, (t : F(\tau'_1, \ldots, \tau'_k)) \Rightarrow \Gamma_2
\]
either rule \((K_2)\) or \((K_3)\) is used. Rule \((K_2)\) is used iff to \( f \) is assigned a signature of the form \( \tau_1 \ast \cdots \ast \tau_n \rightarrow F(\tau_1, \ldots, \tau_k). \) In this case in the proof of \( \Gamma_1, (f(t_1, \ldots, t_n) : \tau_1) \Rightarrow \Gamma'_1 \) we have
\[
\begin{align*}
&f : \tau_1 \ast \cdots \ast \tau_n \rightarrow F(\tau_1, \ldots, \tau_k) \\
&\Gamma_1, (t_i : \theta_1(\tau_1)) \Rightarrow \Gamma'_1 \quad \text{and} \\
&\Gamma'_1 = \Gamma_1 \cap \cdots \cap \Gamma_1.
\end{align*}
\]

Similarly, in the proof of \( \Gamma_2, (f(t_1, \ldots, t_n) : \tau_2) \Rightarrow \Gamma'_2 \) we have
\[
\begin{align*}
&f : \tau_1 \ast \cdots \ast \tau_n \rightarrow F(\tau_1, \ldots, \tau_k) \\
&\Gamma_2, (t_i : \theta_2(\tau_1)) \Rightarrow \Gamma'_2 \quad \text{and} \\
&\Gamma'_2 = \Gamma_2 \cap \cdots \cap \Gamma_2.
\end{align*}
\]
Appendix A. Proofs

Now, we can use A.1 to obtain $\Gamma_1^i \sim \Gamma_2^i$ for $i \in \{1, \ldots, n\}$ and hence, by Corollary 2.45,

$$\Gamma_1' = (\Gamma_1^1 \cap \cdots \cap \Gamma_1^n) \sim (\Gamma_2^1 \cap \cdots \cap \Gamma_2^n) = \Gamma_2'.$$

Now, assume that rule $(K_2)$ cannot be used since there is no signature with the head $F$ assigned to $f$. In this case rule $(K_3)$ is used, and both $\Gamma_1'$ and $\Gamma_2'$ are equal to $\{false\}$, thus $\Gamma_1' \sim \Gamma_2'$.

(Ax3) We have $\vdash \tau_1 \cap \tau_2 = \tau_1' \cap \tau_2'$ with assumptions $\vdash \tau_1 = \tau_1'$ and $\vdash \tau_2 = \tau_2'$. In this case in (⋆) and (⋆⋆) rule $(K_7)$ is used. So, in (⋆) we have

$$\begin{align*}
\Gamma_1', (t : \tau_1) &\Rightarrow \Gamma_1', \\
\Gamma_1', (t : \tau_2) &\Rightarrow \Gamma_1' \quad \text{and} \\
\Gamma_1' &\Rightarrow \Gamma_1' \cap \Gamma_2'.
\end{align*}$$

In (⋆⋆) we have we have

$$\begin{align*}
\Gamma_2', (t : \tau_1') &\Rightarrow \Gamma_2', \\
\Gamma_2', (t : \tau_2') &\Rightarrow \Gamma_2' \quad \text{and} \\
\Gamma_2' &\Rightarrow \Gamma_2' \cap \Gamma_2'.
\end{align*}$$

By the inductive hypothesis, $\Gamma_1' \sim \Gamma_1^2$ and $\Gamma_2' \sim \Gamma_2^2$, and thus, by Corollary 2.45,

$$\Gamma_1' = (\Gamma_1^1 \cap \Gamma_2^2) \sim (\Gamma_1^2 \cap \Gamma_2^2) = \Gamma_2'.$$

(Ax4) For $\vdash \tau_1 \cap \tau_2 = \tau_1' \cap \tau_2'$ the proof goes very similarly: we just substitute $\cap$ by $\cup$ and $\cap$ by $\sqcap$.

(Ax5) We have assumptions $\vdash \tau_1 = \tau_1', \vdash \tau_2 = \tau_2'$ and $\vdash \tau_3 = \tau_2$ with a conclusion $\vdash \tau_1' = \tau_2'$. Let

$$\begin{align*}
\Gamma_1', (t : \tau_1') &\Rightarrow \Gamma_1', \\
\Gamma_2', (t : \tau_2') &\Rightarrow \Gamma_2'.
\end{align*}$$

We want to show that $\Gamma_1' \sim \Gamma_2'$. Let

$$\begin{align*}
\Gamma_1', (t : \tau_1) &\Rightarrow \Gamma_1''', \\
\Gamma_2', (t : \tau_2) &\Rightarrow \Gamma_2'''.
\end{align*}$$

By the inductive hypothesis, we have:

$$\Gamma_1''' \sim \Gamma_1'', \quad \Gamma_2''' \sim \Gamma_2'' \quad \text{and} \quad \Gamma_1''' \sim \Gamma_2'''.$$

Since relation $\sim$ is transitive and symmetric (Corollary 2.33), we have $\Gamma_1' \sim \Gamma_2'$.

(Ax6) The axiom has the form $\vdash \tau \cup \tau = \tau$.

Consider the proofs of $\Gamma_1', (t : \tau \cup \tau) \Rightarrow \Gamma_1'$ and $\Gamma_2', (t : \tau) \Rightarrow \Gamma_2'$. In the proof of $\Gamma_1', (t : \tau \cup \tau) \Rightarrow \Gamma_1'$ we have

$$\begin{align*}
\Gamma_1', (t : \tau) &\Rightarrow \Gamma_1^1 \quad \text{and} \\
\Gamma_1', (t : \tau) &\Rightarrow \Gamma_1^2
\end{align*}$$

and $\Gamma_1' = \Gamma_1^1 \cup \Gamma_1^2$. But, as we can see, $\Gamma_1' = \Gamma_1^2 = \Gamma_2'$ and thus, by Lemma 2.44,

$$\Gamma_1' = \Gamma_1^1 \cup \Gamma_1^2 \sim \Gamma_2'.
$$

For $\vdash \tau \cap \tau = \tau$ the proofs is similar.
For (Ax7)–(Ax10) the proofs are similar.

(Ax11) The axiom has the form \( \tau \cup \top = \top \).

We consider the proofs of \( \Gamma_1, (t : \tau \cup \top) \Rightarrow \Gamma'_1 \) and \( \Gamma_2, (t : \top) \Rightarrow \Gamma'_2 \). In the first proof we have

\[
\Gamma_1, (t : \tau) \Rightarrow \Gamma'_1 \quad \text{and} \quad \Gamma_1, (t : \top) \Rightarrow \Gamma'_2,
\]

and \( \Gamma'_1 = \Gamma'_1 \cup \Gamma'_2 \). It is easy to see that \( \Gamma'_1 = \Gamma_1 \). Lemma 2.49 gives \( \Gamma'_1 \leq \Gamma_1 \), thus by Lemma 2.46, \( \Gamma'_1 \cup \Gamma_1 \sim \Gamma_1 \). Hence

\[
\Gamma'_1 = \Gamma'_1 \cup \Gamma'_2 = \Gamma'_1 \cup \Gamma_1 \sim \Gamma_1.
\]

On the other hand, \( \Gamma'_2 = \Gamma_2 \). Since we have assumed that \( \Gamma_1 \sim \Gamma_2 \), we have \( \Gamma'_1 \sim \Gamma'_2 \).

For \( \vdash \tau \cap \top = \tau \) and for (Ax12) the proofs goes similarly.

(Ax13) The axiom has the form \( \vdash F(\tau_1, \ldots, \tau_n) \cap F(\tau'_1, \ldots, \tau'_n) = F(\tau_1 \cap \tau'_1, \ldots, \tau_n \cap \tau'_n) \). We have proofs of

\[
(\text{A.2}) \quad \Gamma_1, (f(t_1, \ldots, t_k) : F(\tau_1, \ldots, \tau_n) \cap F(\tau'_1, \ldots, \tau'_n)) \Rightarrow \Gamma'_1
\]

and

\[
(\text{A.3}) \quad \Gamma_2, (f(t_1, \ldots, t_k) : F(\tau_1 \cap \tau'_1, \ldots, \tau_n \cap \tau'_n)) \Rightarrow \Gamma'_2
\]

First, suppose that the functor \( f \) does not have a signature with the head \( F \). Then rule \( (K_3) \) is used in the proof of (A.3), so \( \Gamma'_2 = \{\text{false}\} \). In the proof of (A.2) we have

\[
(\text{A.4}) \quad \Gamma_1, (f(t_1, \ldots, t_k) : F(\tau_1, \ldots, \tau_n)) \Rightarrow \Gamma'_1
\]

\[
(\text{A.5}) \quad \Gamma_1, (f(t_1, \ldots, t_k) : F(\tau'_1, \ldots, \tau'_n)) \Rightarrow \Gamma'_2
\]

and

\[
(\text{A.6}) \quad \Gamma'_1 = \Gamma'_1 \cap \Gamma'_2.
\]

But in the proofs of (A.4) and (A.5) rule \( (K_3) \) must be used, so \( \Gamma'_1 = \Gamma'_2 = \{\text{false}\} \). Hence \( \Gamma'_1 \sim \Gamma'_2 \).

Now suppose that the functor \( f \) has a signature with the head \( F \) and thus rule \( (K_2) \) is used in the proof of (A.3). So we have

\[
f : \sigma_1 \ast \cdots \ast \sigma_k \rightarrow \sigma
\]

\[
\Gamma_2, (t_i : \theta_2(\sigma_i)) \Rightarrow \Gamma'_2
\]

\[
\Gamma'_2 = \Gamma'_1 \cap \cdots \cap \Gamma'_2.
\]

In the proof of (A.2) we have

\[
(\text{A.7}) \quad \Gamma_1, (f(t_1, \ldots, t_k) : F(\tau_1, \ldots, \tau_n)) \Rightarrow \Gamma'_1
\]

\[
(\text{A.8}) \quad \Gamma_1, (f(t_1, \ldots, t_k) : F(\tau'_1, \ldots, \tau'_n)) \Rightarrow \Gamma'_2
\]

and

\[
(\text{A.9}) \quad \Gamma'_1 = \Gamma'_1 \cap \Gamma'_2.
\]
Appendix A. Proofs

In the proof of (A.7) rule (K₂) must be used and we have

\( \Gamma_1, (t_1 : \theta_1^1(\sigma_i)) \Rightarrow \Gamma_1^{1,i} \)
\( \Gamma_1 = \Gamma_1^{1,1} \cap \ldots \cap \Gamma_1^{1,n} \).

Similarly, for (A.8) we have

\( \Gamma_1, (t_1 : \theta_1^2(\sigma_i)) \Rightarrow \Gamma_1^{2,i} \)
\( \Gamma_2 = \Gamma_2^{1,1} \cap \ldots \cap \Gamma_2^{1,n} \).

It is easy to check that \( \vdash \theta_2(\sigma_i) = \theta_1^1(\sigma_i) \cap \theta_1^2(\sigma_i) \). Let
\( \Gamma_2, (t_1 : \theta_1^1(\sigma_i) \cap \theta_1^2(\sigma_i)) \Rightarrow \Delta_i \).

Thus \( \Delta_i = \Delta_i^{1} \cap \Delta_i^{2} \), where
\( \Gamma_2, (t_1 : \theta_1^1(\sigma_i)) \Rightarrow \Delta_i^{1} \)
\( \Gamma_2, (t_1 : \theta_1^2(\sigma_i)) \Rightarrow \Delta_i^{2} \).

We can use (A.1) to obtain \( \Gamma_1^{1,i} \sim \Delta_i^{1} \), \( \Gamma_1^{2,i} \sim \Delta_i^{2} \). Moreover, we have
\( \Gamma_i' = \Gamma_i^{1} \cap \Gamma_i^{2} = \Gamma_i^{1,1} \cap \ldots \cap \Gamma_i^{1,n} \cap \Gamma_i^{2,1} \cap \ldots \cap \Gamma_i^{2,n} \).

By (A.1), and since \( \vdash \theta_2(\sigma_i) = \theta_1^1(\sigma_i) \cap \theta_1^2(\sigma_i) \), we have \( \Delta_i \sim \Gamma_i^{2} \), and thus
\( \Gamma_2 = \Gamma_i^{1} \cap \ldots \cap \Gamma_i^{2,n} \)
\( \sim \Delta_i^{1} \cap \ldots \cap \Delta_i^{n} \) by corollary 2.45
\( = \Delta_i^{1} \cap \Delta_i^{2} \cap \ldots \cap \Delta_i^{n} \cap \Delta_i^{2} \)
\( = \Delta_i^{1} \cap \ldots \cap \Delta_i^{n} \cap \Delta_i^{2} \cap \ldots \cap \Delta_i^{n} \).

Since, \( \Delta_i^{1} \sim \Gamma_1^{1,i} \) and \( \Delta_i^{2} \sim \Gamma_2^{1,i} \), we have
\( (\Delta_i^{1} \cap \ldots \cap \Delta_i^{n} \cap \Delta_i^{2} \cap \ldots \cap \Delta_i^{n}) \)
\( \sim (\Gamma_1^{1,1} \cap \ldots \cap \Gamma_1^{1,n} \cap \Gamma_1^{2,1} \cap \ldots \cap \Gamma_1^{2,n}) = \Gamma_i' \)
and thus \( \Gamma_i' \sim \Gamma_2' \).

(Ax14) We have \( \vdash F(\tau_1, \ldots, \tau_n) \cap G(\tau'_1, \ldots, \tau'_m) = \bot \), \( F \neq G \) and \( F \) or \( G \) is not atomic. We have proofs of
\( \Gamma_1, (f(t_1, \ldots, t_k) : F(\tau_1, \ldots, \tau_n) \cap G(\tau'_1, \ldots, \tau'_m)) \Rightarrow \Gamma_1' \)
and
\( \Gamma_2, (f(t_1, \ldots, t_k) : \bot) \Rightarrow \Gamma_2' \).

In the second one rule (K₃) must be used, so \( \Gamma_2' = \{\text{false}\} \). In the first proof rule (K₄) must be used and we have
\( \Gamma_1, (f(t_1, \ldots, t_k) : F(\tau_1, \ldots, \tau_n)) \Rightarrow \Gamma_1' \)
\( \Gamma_1, (f(t_1, \ldots, t_k) : G(\tau'_1, \ldots, \tau'_m)) \Rightarrow \Gamma_1'' \)

But, since \( F \) or \( G \) are not atomic, the functor \( f \) cannot have two signatures, one with head \( F \) and second with head \( G \). Thus either \( \Gamma_1' = \{\text{false}\} \) or \( \Gamma_1'' = \{\text{false}\} \) (the rule \( (K_3) \) is used). In the both cases \( \Gamma_1' = \Gamma_1'' \) and \( \{\text{false}\} \). Hence \( \Gamma_1' = \Gamma_2' \), and of course \( \Gamma_1' \sim \Gamma_2' \).
(Ax15) We have \( \vdash \tau \cup \sigma = \text{or}(\tau, \sigma) \) for atomic types \( \tau \) and \( \sigma \).

Let \( \rho = \text{or}(\tau, \sigma) \) (\( \rho \) is an atomic type too). We consider the proofs of \( \Gamma_1, (t : \tau \cup \sigma) \Rightarrow \Gamma'_1 \) and \( \Gamma_2, (t : \rho) \Rightarrow \Gamma'_2 \). In the first proof we have

\[
\Gamma_1, (t : \tau) \Rightarrow \Gamma'_1, \quad \Gamma_1, (t : \sigma) \Rightarrow \Gamma'_1 \quad \text{and} \quad \Gamma'_1 = \Gamma'_1 \cup \Gamma'_2
\]

We have assumed that \( t \) is not a variable, so it has the form \( t = f(t_1, \ldots, t_n) \).

Suppose that to prove \( \Gamma_2, (t : \rho) \Rightarrow \Gamma'_2 \) is used rule \((K_3)\). It means that there is no signature \( \tau_1 \ast \cdots \ast \tau_n \to \rho \) assigned to \( f \) and \( \Gamma'_2 = \{\text{false}\} \). By definition of \( \text{or} \), there is no signature \( \tau_1 \ast \cdots \ast \tau_n \to \tau \) assigned to \( f \) and there is no signature \( \tau_1 \ast \cdots \ast \tau_n \to \sigma \) assigned to \( f \). Thus \( \Gamma'_1 = \{\text{false}\} \) and \( \Gamma'_1 = \{\text{false}\} \), and hence \( \Gamma'_1 = \{\text{false}\} \).

Now, suppose that to prove \( \Gamma_2, (t : \rho) \Rightarrow \Gamma'_2 \) is used rule \((K_2)\). It means that \( f : \tau_1 \ast \cdots \ast \tau_n \to \rho \) and

\[
\Gamma_2, (t_i : \tau_i) \Rightarrow \Gamma'_2, \\
\Gamma'_2 = \Gamma'_2 \cap \cdots \cap \Gamma^n_2.
\]

By definition of \( \text{or} \), either \( f : \tau_1 \ast \cdots \ast \tau_n \to \tau \) or \( f : \tau_1 \ast \cdots \ast \tau_n \to \sigma \). Suppose that it is the first case. Then \( \Gamma_1, (t : \tau) \Rightarrow \Gamma'_1 \) is proved using rule \((K_2)\) with

\[
\Gamma_1, (t_i : \tau_i) \Rightarrow \Gamma'_i, \\
\Gamma'_1 = \Gamma'_1 \cap \cdots \cap \Gamma^n_i.
\]

Since \( \Gamma_1 \sim \Gamma_2 \), by (A.1), we have \( \Gamma'_1 \sim \Gamma'_i \) for \( i \in \{1, \ldots, n\} \). Thus, by corollary 2.45,

\[
\Gamma'_1 = \Gamma'_1 \cap \cdots \cap \Gamma^n_i = \Gamma'_1 \cap \cdots \cap \Gamma^n_2 = \Gamma'_2.
\]

If \( f \) also the signature \( \tau_1 \ast \cdots \ast \tau_n \to \sigma \) then we can prove in the similar way that \( \Gamma^2_1 \sim \Gamma'_2 \), so

\[
\Gamma'_1 = \Gamma'_1 \cup \Gamma^2_1 \sim \Gamma'_2 \cup \Gamma^2_2 \sim \Gamma'_2.
\]

If \( f : \tau_1 \ast \cdots \ast \tau_n \to \sigma \) does not hold then \( \Gamma^2_1 = \{\text{false}\} \), and thus \( \Gamma'_1 = \Gamma'_1 \cup \Gamma^2_1 = \Gamma'_1 \sim \Gamma'_2 \). In the same way we show that \( \Gamma'_1 \sim \Gamma'_2 \) if \( f : \tau_1 \ast \cdots \ast \tau_n \to \tau \) does not hold.

(Ax16) The proof is similar as in the previous case.

\[\square\]

**Lemma 2.51.** If \( \Gamma_1 \leq \Gamma_2, \quad \Gamma_1, \varphi \Rightarrow \Gamma'_1 \) and \( \Gamma_2, \varphi \Rightarrow \Gamma'_2 \) then \( \Gamma'_1 \leq \Gamma'_2 \).

**Proof.** First, let use notice that proofs of \( \Gamma_1, \varphi \Rightarrow \Gamma'_1 \) and \( \Gamma_2, \varphi \Rightarrow \Gamma'_2 \) have the same structure, i.e. use exactly the same rules. The choice of rule depends only on the structure of \( \varphi \), not on the environment. Hence, we can use induction on the structure of the proofs. Let the proof of \( \Gamma_1, \varphi \Rightarrow \Gamma'_1 \) be denoted by \((*)\) and the proof of \( \Gamma_2, \varphi \Rightarrow \Gamma'_2 \) be denoted by \((**)*\).

The proofs in the cases of rules \((K_1)\), \((K_2)\) and \((K_7)\) are very simple and similar. Therefore, we present only one of them: \((K_2)\). Moreover we omit simple proofs for \((K_3)\), \((K_4)\), \((K_5)\), and \((K_6)\).

\((K_2)\) In \((*)\) and \((**)*\) we have

\[
\Gamma, (t_1 : \theta(t_1)) \Rightarrow \Gamma_1, \ldots, \\
\Gamma', (t_i : \theta(t_i)) \Rightarrow \Gamma'_1, \ldots
\]

\[
\Gamma', (t_1 : \theta(t_1)) \Rightarrow \Gamma'_1, \ldots, \\
\Gamma', (t_i : \theta(t_i)) \Rightarrow \Gamma'_n
\]

We want to show that \( \Gamma_1 \cap \cdots \cap \Gamma_n \leq \Gamma'_1 \cap \cdots \cap \Gamma'_n \). We do it using the inductive hypothesis to obtain \( \Gamma_i \leq \Gamma'_i \), for \( i \in \{1, \ldots, n\} \), and using corollary 2.47.
Appendix A. Proofs

(K8) In (**) we have
\[ \Gamma_i(t : \tau_i) \Rightarrow \Gamma_{1}, \ldots , \Gamma_i(t : \tau_n) \Rightarrow \Gamma_n \]
\[ \Gamma'_i(t : \tau_i) \Rightarrow \Gamma'_{1}, \ldots , \Gamma'_i(t : \tau_n) \Rightarrow \Gamma'_n \]

We want to show that \( \Gamma_{1} \cup \cdots \cup \Gamma_{n} \leq \Gamma'_{1} \cup \cdots \cup \Gamma'_{n} \). We do it using the inductive hypothesis to obtain \( \Gamma_i \leq \Gamma'_i \) for \( i \in \{ 1 , \ldots , n \} \), and using corollary 2.47.

(K9) In (**) we have
\[ \Gamma_1(X : \tau) \Rightarrow \Gamma'_1(X : \tau) \Rightarrow \Gamma'_2(X : \tau) \]
where \( \Gamma'_1 = \Gamma_1 \cap \{ X : \tau \} \) and \( \Gamma'_2 = \Gamma_2 \cap \{ X : \tau \} \). By Lemma 2.34, we have \( \Gamma_1(X) \leq \Gamma_2(X) \). Hence, by Lemma 2.43,
\[ \vdash \Gamma'_1(X) = \tau \cap \Gamma_1(X) \leq \tau \cap \Gamma_2(X) = \Gamma'_2(X). \]
For \( Y \neq X \), \( \Gamma'_1(Y) = \Gamma_1(Y) \) and \( \Gamma'_2(Y) = \Gamma_2(Y) \), and hence \( \Gamma'_1(Y) \leq \Gamma'_2(Y) \).
Thus we have \( \Gamma'_1 \leq \Gamma'_2 \).

Lemma 2.64. If \( \Gamma \vdash t : \tau \) then, for any type substitution \( \theta \),
\[ \theta(\Gamma) \vdash t : \theta(\tau). \]

Proof. The proof proceeds by induction on the structure of the proof of \( \Gamma \vdash t : \tau \).

(T1) We have \( \Gamma \vdash t : \top \). Since \( \theta(\top) = \top \), we use the same rule to prove \( \theta(\Gamma) \vdash t : \top \).

(T2) We have \( \Gamma \vdash \{ \text{false} \} \vdash \bot \). Since \( \theta(\Gamma \cup \{ \text{false} \}) = \theta(\Gamma) \cup \{ \text{false} \} \), and \( \theta(\bot) = \bot \), we can use the same rule to prove
\[ \theta(\Gamma \cup \{ \text{false} \}) \vdash t : \bot. \]

(T3) We have \( \Gamma \vdash \{ X : \tau \} \vdash X : \tau \). We use the same rule to prove
\[ \theta(\Gamma \cup \{ X : \tau \}) = \theta(\Gamma) \cup \{ X : \theta(\tau) \} \vdash X : \theta(\tau). \]

(T4) We have \( \Gamma \vdash t : \tau \) with assumptions \( \Gamma \vdash t : \tau' \) and \( \bot \vdash \tau' \leq \tau \). By the inductive hypothesis, \( \theta(\Gamma) \vdash t : \theta(\tau') \). Moreover, by corollary 2.24, \( \vdash \theta(\tau') \leq \theta(\tau) \). Hence we can use the same rule to obtain
\[ \theta(\Gamma) \vdash t : \theta(\tau). \]

(T5) We have \( \Gamma \vdash t : \tau_1 \cap \cdots \cap \tau_n \) with assumptions \( \Gamma \vdash t : \tau_i \), for \( i \in \{ 1, \ldots , n \} \). By induction \( \theta(\Gamma) \vdash t : \theta(\tau_i) \) for \( i \in \{ 1, \ldots , n \} \). Moreover,
\[ \theta(\tau_1 \cap \cdots \cap \tau_n) = \theta(\tau_1) \cap \cdots \cap \theta(\tau_n). \]
So we can use the same rule to obtain
\[ \theta(\Gamma) \vdash t : \theta(\tau_1 \cap \cdots \cap \tau_n). \]

(T6) We have \( \Gamma \vdash f(t_1, \ldots , t_n) : \nu(\tau) \) with assumptions \( f : \tau_1 \star \cdots \star \tau_n \rightarrow \tau \), \( \Gamma \vdash t_i : \nu(\tau_i) \), for \( i \in \{ 1, \ldots , n \} \). By the inductive hypothesis, for \( i \in \{ 1, \ldots , n \} \), we have \( \theta(\Gamma) \vdash t_i : \theta(\nu(\tau_i)) \) which is equivalent to \( \theta(\Gamma) \vdash t_i : (\theta \circ \nu)(\tau_i) \). Hence we can use the same rule to prove
\[ \theta(\Gamma) \vdash f(t_1, \ldots , t_n) : (\theta \circ \nu)(\tau) \]
and thereby
\[ \theta(\Gamma) \vdash f(t_1, \ldots , t_n) : \theta(\nu(\tau)). \]

\( \square \)

154
A.2 Proofs for Chapter 3

Lemma 3.2. Assume that $\tau, \sigma$ are ground types. If $\vdash \tau = \sigma$ then $[\tau] = [\sigma]$.

The proof is by induction on the structure of the proof of $\vdash \varphi = \psi$. We split the proof into proofs of soundness of axioms. If an axiom has the form $\vdash \varphi = \psi$, we claim that it is correct if $[\varphi] = [\psi]$. Likewise, we claim that a rule of the form

\[
\frac{\vdash \varphi_1 = \psi_1, \ldots, \vdash \varphi_n = \psi_n}{\vdash \varphi = \psi}
\]

is correct if $[\varphi_1] = [\psi_1] \land \ldots \land [\varphi_n] = [\psi_n]$ implies $[\varphi] = [\psi]$.

Soundness of (Ax1): Obviously, for any type $\tau$, we have $[\tau] = [\tau]$.

Soundness of (Ax2): We want to show that

\[
[[\tau_1]] = [[\tau_1']], \ldots, [[\tau_n]] = [[\tau_n']]
\]

implies $[[F(\tau_1, \ldots, \tau_n)]] = [[F(\tau_1', \ldots, \tau_n')]]$.

We shall show equivalent statement:

\[
[[t]] = [[t']]\]

implies that, for any term $t$,

\[
t \in [[F(\tau_1, \ldots, \tau_n)]] \iff t \in [[F(\tau_1', \ldots, \tau_n')]].
\]

We proceed by induction on the depth of $t$. Suppose that $\text{depth}(t) = 0$. Then $t$ is a 0-ary functor $c$. So, by (3.2), and by the definition of signature, we have $c \in [[F(\tau_1, \ldots, \tau_n)]]$ (for $n \geq 0$) iff $c$ has signature $F(\perp, \ldots, \perp)$. This holds iff $c \in [[F(\tau_1', \ldots, \tau_n')]]$.

Now suppose that $\text{depth}(t) > 0$, so $t = f(t_1, \ldots, t_k)$. Suppose that $t \in [[F(\tau_1, \ldots, \tau_n)]]$. We want to show that $t \in [[F(\tau_1', \ldots, \tau_n')]]$. By (3.2), $t \in [[F(\tau_1, \ldots, \tau_n)]]$ if and only if

\[
f : \sigma_1 * \cdots * \sigma_k \to F(y_1, \ldots, y_n)
\]

and, for some type substitution $\theta$,

\[
y_i = \perp \lor \theta(y_i) = \tau_i, \quad \text{and} \quad t_i \in [[[\theta(\sigma_i)]]].
\]

We know that $y_i$ is either $\perp$ or a variable, thus we can define a substitution $\theta'$ as follows.

\[
\theta'(\alpha) = \begin{cases} 
\tau_i' & \text{if } \alpha = y_i \text{ for some } i \in \{1, \ldots, n\} \\
\alpha & \text{otherwise.}
\end{cases}
\]

One can check that

(A.10) \quad $y_i = \perp \lor \theta'(y_i) = \tau_i'$.

Now we want to show that $t_i \in [[[\theta'(\sigma_i)]]]$. We know that $\sigma$ is either a variable in $\text{var}(y_1, \ldots, y_n)$, or has the form $G(\beta_1, \ldots, \beta_m)$, where $\{\beta_1, \ldots, \beta_n\} \subseteq \text{var}(y_1, \ldots, y_n)$.

First, suppose that $\sigma_i$ is a variable, i.e. $\sigma_i \in \text{var}(y_1, \ldots, y_n)$. Then $\theta(\sigma_i) = \tau_j$, and $\theta'(\sigma_i) = \tau_j'$. Since $t_i \in [[[\theta(\sigma_i)]]]$, we have $t_i \in [[[\tau_j]]]$. We have assumed that $[[\tau_j]] = [[\tau_j']]$, hence $t_i \in [[[\tau_j']]]$, and thus $t_i \in [[[\theta'(\sigma_i)]]]$.

Now, suppose that $\sigma_i$ is of the form $G(\beta_1, \ldots, \beta_m)$, where $\{\beta_1, \ldots, \beta_n\} \subseteq \text{var}(y_1, \ldots, y_n)$. Then

\[
\theta(\sigma_i) = G(\tau_{k_1}, \ldots, \tau_{k_m}) \quad \text{and} \quad \theta'(\sigma_i) = G(\tau'_{k_1}, \ldots, \tau'_{k_m}).
\]
where \( \{ \tau_1, \ldots, \tau_m \} \subseteq \{ \tau_1, \ldots, \tau_n \} \), and \( \{ \gamma_1', \ldots, \gamma_m' \} \subseteq \{ \gamma_1', \ldots, \gamma_n' \} \). We have \( t_i \in [G(\tau_1, \ldots, \tau_m)] \) and we have assumed that \( \tau_{k_i} = [\gamma_i'] \) for \( i \in \{1, \ldots, n\} \). Thus, since the depth of \( t_i \) is less than the depth of \( t \), we can use the inductive hypothesis to obtain
\[
t_i \in [G(\gamma_1', \ldots, \gamma_m')] = [\theta(\sigma_i)].
\]
Thus, we have showed that \( t_i \in [\theta(\sigma_i)] \) which, together with (A.10) and (3.2) gives
\[
t \in [F(\tau_1, \ldots, \tau_n)].
\]
In a very similar way one can show that \( t \in [F(\tau_1', \ldots, \tau_n')] \) implies \( t \in [F(\tau_1, \ldots, \tau_n)] \).

**Soundness of (A4):** Suppose that
\[
[t_1] = [\tau_1] \quad \text{and} \quad [t_2] = [\tau_2].
\]
We want to show that
\[
[t_1 \cap t_2] = [\tau_1 \cap \tau_2].
\]
Indeed, from the definition of \([\cdot]\) we have
\[
[t_1 \cap t_2] = [t_1] \cap [t_2] = [\tau_1] \cap [\tau_2] = [\tau_1 \cap \tau_2].
\]

**Soundness of (A4):** The proof is similar to the proof above.

**Soundness of (A6)–(A12):** First of all, we can notice, that meaning of types are sets, meaning of symbols \( \cup \) and \( \cap \) (defined by (3.3)) is set intersection and union, and meaning of \( \top \) and \( \bot \) is \( H \) and \( \emptyset \) respectively. So, proving soundness of (A6)–(A12), which express basic properties of sets, is very simple. For this reason we present only a proof for (A9).

We claim that \([\tau \cup (\tau \cap \sigma)] = [\tau]\). Indeed
\[
[\tau \cup (\tau \cap \sigma)] = [\tau] \cup ([\tau] \cap [\sigma]) \quad \text{by definition of } [\cdot]
\]
\[
= [\tau] \quad \text{(a property of sets)}
\]
In a very similar way one can show that \( [\tau \cap (\tau \cup \sigma)] = [\tau] \).

**Soundness of (A13):** We have to show that
\[
(A.11) \quad [F(\tau_1, \ldots, \tau_n) \cap F(\tau_1', \ldots, \tau_n')] = [F(\tau_1 \cap \tau_1', \ldots, \tau_n \cap \tau_n')].
\]
Let
\[
L = [F(\tau_1, \ldots, \tau_n)] \cap [F(\tau_1', \ldots, \tau_n')] \quad \text{and}
\]
\[
R = [F(\tau_1 \cap \tau_1', \ldots, \tau_n \cap \tau_n')].
\]
(A.11) holds if and only if, for any term \( t, t \in L \iff t \in R \). We proceed by induction on the depth of \( t \).

Let \( t \in R \). We are going to show that \( t \in L \). We have \( t \in [F(\tau_1 \cap \tau_1', \ldots, \tau_n \cap \tau_n')] \).

By (3.2), \( t = f(t_1, \ldots, t_k) \),
\[
f : \sigma_1 \ast \cdots \ast \sigma_k \to F(y_1, \ldots, y_n),
\]
\[
\exists \theta((y_i = \bot \lor \theta(y_i) = \tau_i \cap \tau_i') \land t_i \in [\theta(\sigma_i)]).
\]

156
Let us define $\theta_1$ and $\theta_2$ as follows:

$$\theta_1(\alpha) = \begin{cases} \tau_i & \text{if } \alpha = y_i \text{ for some } i \\ \alpha & \text{otherwise} \end{cases} \quad \theta_2(\alpha) = \begin{cases} \tau'_i & \text{if } \alpha = y_i \text{ for some } i \\ \alpha & \text{otherwise} \end{cases}$$

Let us notice that if $y_i$ is not $\perp$, it is a variable, and thus

(A.12) $\alpha \in \text{var}(y_1, \ldots, y_n) \implies \theta(\alpha) = \theta_1(\alpha) \cap \theta_2(\alpha)$.

By the definition of $\theta_1$ and $\theta_2$ we have:

(A.13) $y_i = \perp \lor \theta_1(y_i) = \tau_i$, and $y_i = \perp \lor \theta_2(y_i) = \tau'_i$.

If $\text{depth}(t) = 0$ then $f$ has arity 0, and (A.13) and (3.2) give $t \in L$. If $\text{depth}(t) > 0$ we should also prove that $t_i \in [[\theta_1(\sigma_i)]]$, and $t_i \in [[\theta_2(\sigma_i)]]$. There are two cases: (a) $\sigma_i$ is a variable, and (b) $\sigma_i = G(\beta_1, \ldots, \beta_m)$.

(a) $\sigma_i$ is a variable. Then, by the definition of signature, $\sigma_i \in \text{var}(y_1, \ldots, y_n)$, thus, by (A.12), $\theta(\sigma_i) = \theta_1(\sigma_i) \cap \theta_2(\sigma_i)$. Since $t_i \in [[\theta(\sigma_i)]] = [[\theta_1(\sigma_i) \cap \theta_2(\sigma_i)]]$, by (3.3), we get $t_i \in [[\theta_1(\sigma_i)]]$ and $t_i \in [[\theta_2(\sigma_i)]]$.

(b) $\sigma_i = G(\beta_1, \ldots, \beta_j)$, where $\beta_j \in \text{var}(y_1, \ldots, y_n)$. By (A.12), we have

$$t_i \in [[\theta(\sigma_i)]] = [G(\theta(\beta_1), \ldots, \theta(\beta_j))] = [G(\theta_1(\beta_1) \cap \theta_2(\beta_1), \ldots, \theta_1(\beta_j) \cap \theta_2(\beta_j))].$$

so by the inductive hypothesis

$$t_i \in [G(\theta_1(\beta_1), \ldots, \theta_1(\beta_j))] \cap [G(\theta_2(\beta_1), \ldots, \theta_2(\beta_j))].$$

i.e.

$$t_i \in [G(\theta_1(\beta_1), \ldots, \theta_1(\beta_j))] = [\theta_1(\sigma_i)] \quad \text{and} \quad t_i \in [G(\theta_2(\beta_1), \ldots, \theta_2(\beta_j))] = [\theta_2(\sigma_i)].$$

So, we have showed that $t \in L$.

Now, assume that $t \in L$. Then $t \in [[F(\tau_1, \ldots, \tau_n)]]$ and $t \in [[F(\tau'_1, \ldots, \tau'_n)]]$. By (3.2), $t = f(t_1, \ldots, t_k)$ and

$$f : \sigma_1 \ast \ldots \ast \sigma_k \to F(y_1, \ldots, y_n),$$

$$\exists \theta_1 ((y_i = \perp \lor \theta_1(y_i) = \tau_i) \land t_i \in [[\theta_1(\sigma_i)]]),$$

$$\exists \theta_2 ((y_i = \perp \lor \theta_2(y_i) = \tau'_i) \land t_i \in [[\theta_2(\sigma_i)]]).$$

Let us define substitution $\theta$ as follows:

$$\theta(\alpha) = \begin{cases} \tau_i \cap \tau'_i & \text{if } \alpha = y_i \text{ for some } i \\ \alpha & \text{otherwise} \end{cases}$$

Let us notice that

(A.14) $\alpha \in \text{var}(y_1, \ldots, y_n) \implies \theta(\alpha) = \theta_1(\alpha) \cap \theta_2(\alpha)$.

From definition of $\theta$ we also have:

(A.15) $y_i = \perp \lor \theta(y_i) = \tau_i \cap \tau'_i$

If $\text{depth}(t) = 0$ then $f$ has arity 0, so (A.15) and (3.2) gives $t \in R$. If $\text{depth}(t) > 0$ ($t = f(t_1, \ldots, t_n)$) we should also prove that $t_i \in [[\theta(\sigma_i)]]$. There are two possible cases:
Appendix A. Proofs

a) \( \sigma_i \) is a variable. Then, by definition of signature, \( \sigma_i \in \var(y_1, \ldots, y_n) \), and thus, by (A.14), \( \theta(\sigma_i) = \theta_1(\sigma_i) \cap \theta_2(\sigma_i) \). Since \( t_i \in [\theta_1(\sigma_i)] \) and \( t_i \in [\theta_2(\sigma_i)] \) thus, by (3.3), \( t_i \in [\theta_1(\sigma_i) \cap \theta_2(\sigma_i)] = [\theta(\sigma_i)] \).

b) \( \sigma_i = G(\beta_1, \ldots, \beta_m) \), where \( \beta_i \in \var(y_1, \ldots, y_n) \). By (A.14), we have

\[
t_i \in [\theta_1(\sigma_i)] \cap [\theta_2(\sigma_i)] = [G(\theta_1(\beta_1), \ldots, \theta_1(\beta_j))] \cap [G(\theta_2(\beta_1), \ldots, \theta_2(\beta_j))]
\]

Moreover, \( \text{depth}(t_i) < \text{depth}(t) \), so by the inductive hypothesis,

\[
t_i \in [G(\theta_1(\beta_1) \cap \theta_2(\beta_1), \ldots, \theta_1(\beta_j) \cap \theta_2(\beta_j))] = [\theta(\sigma_i)]
\]

This completes the proof.

**Soundness of (Ax14):** We want to show that if \( F \neq G \) and at least one of them is not atomic, then

\[
[F(\tau_1, \ldots, \tau_n) \cap G(\tau'_1, \ldots, \tau'_m)] = [\bot].
\]

We can transform this equation into equivalent form (by (3.3) and (3.1)):

(A.16) \[
[F(\tau_1, \ldots, \tau_n)] \cap [G(\tau'_1, \ldots, \tau'_m)] = \emptyset
\]

Suppose that \( t \in [F(\tau_1, \ldots, \tau_n)] \). By (3.2), \( t \) must be equal to \( f(t_1, \ldots, t_k) \) and \( f \) have signature \( \sigma_1 \ast \cdots \ast \sigma_k \to F(y_1, \ldots, y_n) \). Now, we shall consider two possible cases:

a) **\( F \) is not atomic.**

According to definition of the signature (section 2.2.2), there is no other signature of \( f \), and thus \( f \) cannot have a signature \( \sigma'_1 \ast \cdots \ast \sigma'_j \to G(y_1, \ldots, y'_m) \). It proves that \( t \notin [G(\tau'_1, \ldots, \tau'_m)] \).

b) **\( F \) is atomic and thus \( G \) is not atomic.**

Signature of \( f \) is atomic, so, according to definition of signature, each other signature of \( f \) must also be atomic. If \( t \) were in \([G(\tau'_1, \ldots, \tau'_m)]\), some signature of \( f \) would have the form \( \sigma'_1 \ast \cdots \ast \sigma'_j \to G(y_1, \ldots, y'_m) \), for \( m > 0 \), which contradicts with assumption that each signature of \( f \) is atomic. Hence, \( t \notin [G(\tau'_1, \ldots, \tau'_m)] \).

So, we have showed that if \( t \notin [F(\tau_1, \ldots, \tau_n)] \) then \( t \notin [G(\tau'_1, \ldots, \tau'_m)] \). In a very similar way we show that if \( t \notin [G(\tau'_1, \ldots, \tau'_m)] \) then \( t \notin [F(\tau_1, \ldots, \tau_n)] \), which proves (A.16).

**Soundness of (Ax15):** Assume that \( \tau \) and \( \tau' \) are atomic types, \( \text{or}(\tau, \tau') \) is defined. We want to show that

\[
[\tau \cup \tau'] = [\text{or}(\tau, \tau')].
\]

By (3.3) we can transform this equation into equivalent form:

\[
L = [\tau] \cup [\tau'] = [\text{or}(\tau, \tau')] = R.
\]

First, notice that since \( \text{or}(\tau, \tau') \) is defined, there exists an atomic type \( \sigma \) such that

(A.17) \[
\sigma = \text{or}(\tau, \tau') \quad \text{and} \quad \text{Constr}(\sigma) = \text{Constr}(\tau) \cup \text{Constr}(\tau').
\]

Now, assume that \( t \in L \) and thus either \( t \in [\tau] \) or \( t \in [\tau'] \). Consider the first case. By (3.2):

(A.18) \[
t = f(t_1, \ldots, t_n), \quad f : \xi_1 \ast \cdots \ast \xi_k \to \tau, \quad t_i \in \xi_i
\]

158
It is clear that \( f \in \text{Constr}(\tau) \), thus, by (A.17), \( f \in \text{Constr}(\sigma) \). Since, all signatures of \( f \) must have the same left sides, we have \( f : \xi_1 \cdots \xi_k \rightarrow \sigma \). By (A.18) we have \( t_i \in \xi_i \). This implies that

\[
t = f(t_1, \ldots, t_n) \in [\sigma] = [\text{or}(\tau, \tau')]
\]

In the second case the proof goes in a very similar way. So, we have proved that \( t \in R \).

Now assume that \( t \in R = [\sigma] \). It holds if

(A.19) \[
t = f(t_1, \ldots, t_n), \quad f : \xi_1 \cdots \xi_k \rightarrow \sigma, \quad t_i \in \xi_i
\]

So, \( f \in \text{Constr}(\sigma) \) and thus, by (A.17), either \( f \in \text{Constr}(\tau) \) or \( f \in \text{Constr}(\tau') \).

Because of that and since left sides of signatures of \( f \) must be the same, we have

\[
f : \xi_1, \ldots, \xi_n \rightarrow \tau \quad \text{or} \quad f : \xi_1, \ldots, \xi_n \rightarrow \tau'
\]

By (A.19) we have \( t_i \in \xi_i \), so

\[
f(t_1, \ldots, t_n) \in [\tau] \quad \text{or} \quad f(t_1, \ldots, t_n) \in [\tau']
\]

Hence \( t = f(t_1, \ldots, t_n) \in [\tau] \cup [\tau'] \).

**Soundness of (Ax16):** Assume that \( \tau \) and \( \tau' \) are atomic types, \( \text{and}(\tau, \tau') \) is defined. We want to show that

\[
[\tau \cap \tau'] = [\text{and}(\tau, \tau')]
\]

By (3.3) we can transform this equation into equivalent form:

\[
L = [\tau] \cap [\tau'] = [\text{and}(\tau, \tau')] = R.
\]

Since \( \text{and}(\tau, \tau') \) is defined, there exists \( \sigma \) such that

(A.20) \[
\sigma = \text{and}(\tau, \tau') \land (\text{Constr}(\sigma) = \text{Constr}(\tau) \cap \text{Constr}(\tau') \lor \sigma = \bot).
\]

Now, assume that \( t \in L \) and thus \( t \in [\tau] \) and \( t \in [\tau'] \). By (3.2):

(A.21) \[
t = f(t_1, \ldots, t_n), \quad f : \xi_1 \cdots \xi_n \rightarrow \tau, \quad t_i \in \xi_i, \quad f : \xi_1 \cdots \xi_n \rightarrow \tau'
\]

It holds since left sides of signatures of \( f \) must be the same. We will now consider two cases:

a) \( \sigma = \bot \). From (A.20) and definition of \( \text{and} \) we have \( \text{Constr}(\tau) \cap \text{Constr}(\tau') = \varnothing \). But \( f \in \text{Constr}(\tau) \) and \( f \in \text{Constr}(\tau') \). So we have contradiction.

b) \( \sigma \neq \bot \), and thus, by (A.20) \( \sigma \) must be an atomic type and \( \text{Constr}(\sigma) = \text{Constr}(\tau) \cap \text{Constr}(\tau') \) holds. From (A.21) it follows that \( f \in \text{Constr}(\tau) \) and \( f \in \text{Constr}(\tau') \). Hence \( f \in \text{Constr}(\sigma) \). Since all signatures of \( f \) must have equal left sides, we have

\[
f : \xi_1 \cdots \xi_k \rightarrow \sigma.
\]

By (A.21) we have \( t_i \in \xi_i \). This implies that

\[
t = f(t_1, \ldots, t_n) \in [\sigma] = [\text{and}(\tau, \tau')]
\]

So we have proved that \( t \in R \).
Appendix A. Proofs

Now assume that \( t \in R = [\sigma] \), so \( \sigma \neq \bot \). It holds if
\[
\text{(A.22)} \quad t = f(t_1,\ldots,t_n), \quad f : \xi_1 \cdots \xi_n \to \sigma, \quad t_i \in \xi_i
\]
so, \( f \in \text{Constr}(\sigma) \) and thus, by (A.20) and since \( \sigma \neq \bot \), \( f \in \text{Constr}(\tau') \). Because of that and because left sides of signatures of \( f \) must be equal, we have
\[
f : \xi_1,\ldots,\xi_n \to \tau \quad \text{and} \quad f : \xi_1,\ldots,\xi_n \to \tau'
\]
From (A.22) it follows that \( t_i \in \xi_i \), so
\[
f(t_1,\ldots,t_n) \in [\tau] \quad \text{and} \quad f(t_1,\ldots,t_n) \in [\tau']
\]
Hence \( t = f(t_1,\ldots,t_n) \in [\tau] \cap [\tau'] \), which means that \( t \in L \).

**Lemma 3.6.** Let \( \Gamma \) be a ground environment, \( \tau \) be a ground type, and \( t \) be a term. If \( \Gamma \vdash t : \tau \) then \( \Gamma \vdash t : \tau \).

We proceed by induction on the structure of the proof of \( \Gamma \vdash t : \tau \), i.e. we prove soundness of each term typing rule. We claim that rule of the form
\[
\frac{\Gamma \vdash \varphi_1,\ldots,\Gamma \vdash \varphi_n}{\Gamma \vdash \varphi}
\]
is sound iff \( \Gamma \models \varphi_1,\ldots,\Gamma \models \varphi_n \) implies \( \Gamma \models \varphi \).

**Soundness of (T1):** \( \Gamma \models t : \top \) obviously holds, since in every model \( M_v \), by definition of variables mapping, \( v(t) \in H = [\top] \).

**Soundness of (T2):** \( \Gamma \cup \{ \text{false} \} \models t : \bot \) holds, since there is no model which satisfies false.

**Soundness of (T3):** \( \Gamma \cup \{ X : \tau \} \models X : \tau \) obviously satisfies \( X : \tau \).

**Soundness of (T4):** Let us assume that \( \models \tau' \leq \tau \) and \( \Gamma \models t : \tau' \). We should show that \( \Gamma \models t : \tau \).

For each model \( M_v \) satisfying \( \Gamma \) we have \( M_v \models t : \tau' \) which means that \( v(t) \in [\tau'] \). Moreover, \( \models \tau' \leq \tau \) means that \( [\tau'] \subseteq [\tau] \). Thus we have \( v(t) \in [\tau] \) which is equivalent to \( M_v \models t : \tau \). It gives \( \Gamma \models t : \tau \).

**Soundness of (T5):** We should show that if \( \Gamma \models t : \tau_1 \land \cdots \land \Gamma \models t : \tau_n \) then \( \Gamma \models t : \tau_1 \cap \cdots \cap \tau_n \).

So, assume that \( \Gamma \models t : \tau_1 \land \cdots \land \Gamma \models t : \tau_n \). Assume also that \( M_v \) is a model satisfying \( \Gamma \). So we have \( v(t) \in [\tau_1] \land \cdots \land v(t) \in [\tau_n] \) and hence \( v(t) \in [\tau_1] \cap \cdots \cap [\tau_n] \). Thus, we have \( \Gamma \models t : \tau_1 \cap \cdots \cap \tau_n \).

**Soundness of (T6):** Assume that \( M_v \) is a model such that \( M_v \models \Gamma, \ f : \sigma_1,\ldots,\sigma_n \to \sigma \) and \( v(t_i) \in [\theta(\sigma_i)] \) for \( i \in \{ 1,\ldots,n \} \). Furthermore, let \( \sigma = F(y_1,\ldots,y_n) \). To prove Soundness of the rule we should show that \( v(f(t_1,\ldots,t_n)) \in [\theta(\sigma)] \).

From our assumption and from (3.2) it follows that
\[
v(f(t_1,\ldots,t_n)) \in [F(\theta(y_1),\ldots,\theta(y_n))]\).
By the definition of variable mapping and substitution, this is equivalent to
\[ v(f(t_1, \ldots, t_n)) \in \llbracket \theta(F(y_1, \ldots, y_n)) \rrbracket = \llbracket \theta(\sigma) \rrbracket \]
which closes the proof.

**Lemma 3.8.** Let \( \Gamma, \Gamma' \) be a ground environments, \( \varphi \) a ground formula. If \( (\Gamma, \varphi \Rightarrow \Gamma') \) is provable in our system then we have \( \models (\Gamma, \varphi \Rightarrow \Gamma') \).

We shall proceed by induction on the structure of the proof of \( (\Gamma, \varphi \Rightarrow \Gamma') \). In order to do that we will prove soundness of consequence operator rules.

**Correctness of \( (K_1) \)** Assume that, for \( i \in \{1, \ldots, n\} \),
\[ M_v \models \Gamma \land M_v \models (t : \tau_i) \implies M_v \models \Gamma_i \]
We want to show that
\[ M_v \models \Gamma \land M_v \models (t_1 : \tau_1) \land \cdots \land M_v \models (t_n : \tau_n) \implies M_v \models \Gamma_1 \land \cdots \land \Gamma_n. \]
So assume that
\[ M_v \models \Gamma \land M_v \models (t_1 : \tau_1) \land \cdots \land M_v \models (t_n : \tau_n) \]
We should show that
\[ M_v \models \Gamma_1 \land \cdots \land \Gamma_n, \]
which holds if, for each variable \( X \),
\[ v(X) \in \llbracket (\Gamma_1 \land \cdots \land \Gamma_n)(X) \rrbracket. \]
(A.25) implies that, for each \( i \in \{1, \ldots, n\} \),
\[ M_v \models \Gamma \land M_v \models (t_i : \tau_i) \]
and thus, by (A.23), \( M_v \models \Gamma_i \). Hence, for each \( X \), \( v(X) \in \llbracket \Gamma_i(X) \rrbracket \), and thus
\[ v(X) \in \llbracket \Gamma_1(X) \rrbracket \land \cdots \land \llbracket \Gamma_n(X) \rrbracket = \llbracket \Gamma_1(X) \rrbracket \land \cdots \land \llbracket \Gamma_n(X) \rrbracket \]
Since, by Lemma 2.43,
\[ \models \Gamma_1(X) \land \cdots \land \Gamma_n(X) = (\Gamma_1 \land \cdots \land \Gamma_n)(X), \]
by Lemma 3.2, we have
\[ v(X) \in \llbracket (\Gamma_1 \land \cdots \land \Gamma_n)(X) \rrbracket. \]
which closes the proof.

**Correctness of \( (K_2) \)** Assume that
\[ f : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau, \]
(A.26) \[ \begin{array}{c}
M_v \models \Gamma \land M_v \models (t_i : \theta(\tau_i)) \\
\implies M_v \models \Gamma_i \quad \text{for } i \in \{1, \ldots, n\}
\end{array} \]
(A.27)
Appendix A. Proofs

We want to show that

\[ M_v \models \Gamma \land M_v \models (f(t_1, \ldots, t_n) : \theta(\tau)) \implies M_v \models \Gamma_1 \land \cdots \land \Gamma_n. \]

So assume that

(A.28) \[ M_v \models \Gamma \land M_v \models (f(t_1, \ldots, t_n) : \theta(\tau)) \]

We should show that

(A.29) \[ M_v \models \Gamma_1 \land \cdots \land \Gamma_n. \]

From (A.28) we have

\[ v(f(t_1, \ldots, t_n)) \in [\theta(\tau)] \]

where \( \tau \) has the form \( F(y_1, \ldots, y_k) \). This is equivalent to

\[ f(v(t_1), \ldots, v(t_n)) \in [\theta(\mathcal{N}(F(y_1, \ldots, y_k))] \]

By (3.2), it holds only if

\[ v(t_i) \in [\theta(\tau_i)] \quad \text{for } i \in \{1, \ldots, n\}. \]

Thus we have, for each \( i \in \{1, \ldots, n\} \), \( M_v \models (t_i : \theta(\tau_i)) \), so, by (A.27), \( M_v \models \Gamma_i \). Therefore, for each variable \( X \)

\[ v(X) \in [\Gamma_i(X)]. \]

and thus

\[ v(X) \in [\Gamma_1(X)] \land \cdots \land [\Gamma_n(X)] \]

\[ = [\Gamma_1(X) \land \cdots \land \Gamma_n(X)] \]

which, by Lemma 2.43 and Lemma 3.2, is equivalent to

\[ v(X) \in [(\Gamma_1 \land \cdots \land \Gamma_n)(X)]. \]

Hence (A.29) holds.

Correctness of \((K_3)\) Let us assume that there is no signature \( \tau_1 \cdots \tau_n \rightarrow \tau_0 \) assigned to \( f \) such that \( \text{head}(\tau) = \text{head}(\tau_0) \). We want to show that

(A.30) \[ M_v \models \Gamma \land M_v \models (f(t_1, \ldots, t_n) : \tau) \implies M_v \models \{\text{false}\}. \]

\( M_v \models (f(t_1, \ldots, t_n) : \tau) \) means that \( v(f(t_1, \ldots, t_n)) \in [\tau] \), which is equivalent to

(A.31) \[ f(v(t_1), \ldots, v(t_n)) \in [\tau] \]

But, we have assumed that there is no signature assigned to \( f \) with the same head as \( \tau \), thus, by (3.2), (A.31) is impossible. Thus, since the left-hand side of the implication (A.30) is false, the whole implication is true.

Correctness of \((K_4)\) We want to show that

\[ M_v \models \Gamma \land M_v \models (t : \bot) \implies M_v \models \{\text{false}\}. \]

\( M_v \models (t : \bot) \) holds iff \( v(t) \in \emptyset \) which cannot be true, thus the implication above is true.

162
Correctness of \((K_5)\)  We want to show
\[
M_v \models \Gamma \land M_v \models (t : \top) \implies M_v \models \Gamma
\]
which is obviously true.

Correctness of \((K_6)\)  This rule cannot be used since we consider only types without variables.

Correctness of \((K_7)\)  Assume that
\[
M_v \models \Gamma \land M_v \models (t : \tau_i) \implies M_v \models \Gamma_i
\]
for \(i \in \{1, \ldots, n\}\). We want to show that
\[
M_v \models \Gamma \land M_v \models (t : \tau_1 \cap \ldots \cap \tau_n) \implies M_v \models \Gamma_1 \cap \ldots \cap \Gamma_n.
\]
So assume that
\[
M_v \models \Gamma \land M_v \models (t : \tau_1 \cap \ldots \cap \tau_n).
\]
We should show that
\[
M_v \models \Gamma_1 \cap \ldots \cap \Gamma_n.
\]
It suffices to show that for each variable \(X\),
\[
v(X) \in \llbracket (\Gamma_1 \cap \ldots \cap \Gamma_n)(X) \rrbracket.
\]
\[(A.33)\] implies that
\[
v(t) \in \llbracket \tau_1 \cap \ldots \cap \tau_n \rrbracket = \llbracket \tau_1 \rrbracket \cap \ldots \cap \llbracket \tau_n \rrbracket.
\]
Hence, for each \(i \in \{1, \ldots, n\}\), \(v(t) \in \llbracket \tau_i \rrbracket\) and thus
\[
M_v \models (t : \tau_i).
\]
So, by \((A.32)\), we have \(M_v \models \Gamma_i\). Hence for each \(X\), \(v(X) \in \llbracket \Gamma_i(X) \rrbracket\), and thus
\[
v(X) \in \llbracket \Gamma_1(X) \rrbracket \cap \ldots \cap \llbracket \Gamma_n(X) \rrbracket = \llbracket \Gamma_1(X) \cap \ldots \cap \Gamma_n(X) \rrbracket.
\]
Since, by Lemma 2.43,
\[
\vdash \Gamma_1(X) \cap \ldots \cap \Gamma_n(X) = (\Gamma_1 \cap \ldots \cap \Gamma_n)(X),
\]
by Lemma 3.2, we have \((A.34)\) which closes the proof.

Correctness of \((K_8)\)  The proof goes in the very similar way as for \((K_7)\)
Appendix A. Proofs

Correctness of $(K_0)$ We want to show that

\[ M_v \models \Gamma \land M_v \models (X : \tau) \implies M_v \models \Gamma \cap \{X : \tau\} \]

Let $\Gamma' = \Gamma \cap \{X : \tau\}$. Assume that

\[ (A.35) \quad M_v \models \Gamma \land M_v \models (X : \tau) \]

In order to prove that $M_v \models \Gamma'$ it suffices to show that, for each $Y$, 

\[ (A.36) \quad v(Y) \in [\Gamma'(Y)] \]

First, suppose that $Y \neq X$. So, $\Gamma'(Y) = \Gamma(Y)$. Since $v(Y) \in [\Gamma(Y)]$ follows from $(A.35)$, we have also $(A.36)$.

Now suppose that $X = Y$. From $(A.35)$ it follows that

\[ v(X) \in [\Gamma(X)] \quad \text{and} \quad v(X) \in [\tau] \]

Hence 

\[ v(X) \in [\Gamma(X)] \cap [\tau] = [\tau \cap \Gamma(X)] = \Gamma'(X) \]

which, since $Y = X$, gives $(A.36)$.

A.3 Proofs for Chapter 4

Lemma 4.9. For each type $\tau$, $\vdash \tau = N_L(\tau)$

Proof. The proof proceeds by induction of the structure of computation of $N_L$. We consider only two cases, $f_{ac}(\tau) = (\tau_1 \cup \tau_2) \cap \tau_3$, for other cases proof are similar.

- case (4.4). We have 

\[ \vdash N_L(\tau) = N'_L(\tau_1 \cup \tau_2) \]

\[ = N_L(\tau_1) \cup N_L(\tau_2) \]

\[ = \tau_1 \cup \tau_2 \]

by inductive hypothesis and (Ax3)

So, by remark 4.5, we have $\vdash N_L(\tau) = \tau_1 \cup \tau_2 = f_{ac}(\tau) = \tau$

- case (4.8) We have

\[ \vdash N_L(\tau) = N'_L((\tau_1 \cup \tau_2) \cap \tau_3) \]

\[ = N_L(\tau_1 \cap \tau_3) \cup N_L(\tau_2 \cap \tau_3) \]

\[ = (\tau_1 \cap \tau_3) \cup (\tau_2 \cap \tau_3) \]

by inductive hypothesis and (Ax4)

\[ = (\tau_1 \cap \tau_3) \cup (\tau_1 \cap \tau_3) \]

So, by remark 4.5, we have $\vdash N_L(\tau) = (\tau_1 \cup \tau_2) \cap \tau_3 = f_{ac}(\tau) = \tau$

\[ \square \]

Lemma 4.16. For any variables $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_m$, for any types $\tau, \tau_1, \ldots, \tau_n, \sigma_1, \ldots, \sigma_p$, and for any type constructors $F, G_1, \ldots, G_p$, such that $F \neq G_i$ for $i \in \{1, \ldots, p\}$ we have

\[ \vdash \bigcap_{i=1}^k \alpha_i \cap F(\tau) \leq \bigcup_{i=1}^m \beta_i \cup \bigcup_{i=1}^n F(\tau_i) \cup \bigcup_{i=1}^p G_i(\sigma_i) \]

164
if and only if

\[ \vdash \bigcap_{i=1}^{k} \alpha_i \leq \bigcup_{i=1}^{m} \beta_i \text{ or } (\exists i) \vdash \tau \leq \tau_i \]

Proof. The formula

\[ \vdash \bigcap_{i=1}^{k} \alpha_i \cap F(\tau) \leq \bigcup_{i=1}^{m} \beta_i \cup \bigcup_{i=1}^{n} F(\tau_i) \cup \bigcup_{i=1}^{p} G_i(\sigma_i) \]

holds if and only if

(A.37) \[ \vdash \bigcap_{i=1}^{k} \alpha_i \cap F(\tau) = \bigcap_{i=1}^{k} \alpha_i \cap \bigcap_{j=1}^{m} \bigcup_{i=1}^{n} \bigcup_{j=1}^{k} \alpha_i \cap F(\tau \cap \tau_j) \]

Since \[ \vdash F(\tau) \cap G(\sigma_i) = \bot \] (A.37) is equivalent to

\[ \vdash \bigcap_{i=1}^{k} \alpha_i \cap F(\tau) = \bigcap_{i=1}^{k} \alpha_i \cap \beta_j \cap F(\tau) \]

or there exists \( j \) s.t.

(A.38) \[ \vdash \bigcap_{i=1}^{k} \alpha_i \cap F(\tau) = \bigcap_{i=1}^{k} \alpha_i \cap \beta_j \cap F(\tau) \]

or 2.

(A.39) \[ \vdash \bigcap_{i=1}^{k} \alpha_i \cap F(\tau) = \bigcap_{i=1}^{k} \alpha_i \cap F(\tau \cap \tau_j) \]

The alternative of (A.38) and (A.39), since \[ \vdash \bigcap_{i=1}^{k} \alpha_i = \bigcap_{i=1}^{k} \alpha_i \cap \beta_j \Leftrightarrow (\exists i)\alpha_i = \beta_j, \]

is equivalent to

\[ \vdash \bigcap_{i=1}^{k} \alpha_i \leq \bigcup_{i=1}^{m} \beta_i \text{ or } (\exists i) \vdash \tau \leq \tau_i \]

which finishes the proof.

\[ \square \]

Lemma 4.27. Let \( \tau_1 \leq \tau_2 \) be the type equation in normal form. We have:

\[ \vdash \tau_1 \leq \tau_2 \text{ iff } SF(\tau_1 \leq \tau_2) \text{ is valid} \]

Proof. The proof proceeds by induction on the definition of \( SF \). Consider two cases depending whether the case (4.17) or (4.18) of definition of \( SF \) has been applied.

• case (4.17). We have

\[ \vdash \tau_1 \cup \cdots \cup \tau_m \leq \sigma_1 \cap \cdots \cap \sigma_m \]

by Lemma 4.26 is equivalent to

\[ (\forall i, j) \vdash \tau_i \leq \sigma_j \]

This, by the induction hypothesis is equivalent to

(A.40) \[ (\forall i, j)SF(\tau_i \leq \sigma_j) \]
Appendix A. Proofs

Finally, (A.40) is equivalent to

$$SF(\tau_1 \cup \cdots \cup \tau_n \leq \sigma_1 \cap \cdots \cap \sigma_m)$$

We have

$$\vdash \bigcap_{i=1}^{k} \alpha_i \cap F(\tau) \leq \bigcup_{i=1}^{m} \beta_i \cup \bigcup_{i=1}^{n} F(\tau_i) \cup \bigcup_{i=1}^{p} G_i(\sigma_i)$$

by Lemma 4.16 is equivalent to

$$\vdash \bigcap_{i=1}^{k} \alpha_i \leq \bigcup_{i=1}^{m} \beta_i \text{ or } (\exists i \in \{1, \ldots, n\}) \vdash \tau \leq \tau_i$$

This, by the inductive hypothesis is equivalent to

(A.41) $$\vdash \bigcap_{i=1}^{k} \alpha_i \leq \bigcup_{i=1}^{m} \beta_i \text{ or } (\exists i)SF(\tau \leq \tau_i)$$

Finally, (A.41) is equivalent to

$$SF\left(\bigcap_{i=1}^{k} \alpha_i \cap F(\tau) \leq \bigcup_{i=1}^{m} \beta_i \cup \bigcup_{i=1}^{n} F(\tau_i) \cup \bigcup_{i=1}^{p} G_i(\sigma_i)\right)$$

□

Lemma 4.32. Assume that $\tau$ has size $n$. Then the size of the $N_L(\tau)$ is less than $2^n$. Moreover, the size of the $N_L(\tau)$ is less than $2^n$.

Proof. Consider the function $N_L$. We proceed by induction on the size of $\tau$. Since, $|\tau| = |f_{ac}(\tau)|$ we can assume that $\tau = f_{ac}(\tau)$. The proof is routine and we consider only cases $\tau = \alpha$, $\tau = \tau_1 \cup \tau_2$, $\tau = \tau_1 \cap (\tau_2 \cup \tau_3)$, and $\tau = \bigcap_{i=1}^{n} F(\tau_1, \ldots, \tau_n)$.

- $\tau = \alpha$. We have $|N_L(\alpha)| = |\alpha| = 1 \leq 2^1$.
- $\tau = \tau_1 \cup \tau_2$. We have $N_L(\tau_1 \cup \tau_2) = N_L(\tau_1) \cup N_L(\tau_2)$. From this we get

$$|N_L(\tau_1 \cup \tau_2)| = 1 + |N_L(\tau_1)| + |N_L(\tau_2)|$$

$$\leq 1 + 2^{r_1} + 2^{r_2}$$

$$\leq 1 + 2^{\max(|\tau_1|,|\tau_2|)+1}$$

$$\leq 2^{r_1+|\tau_2|+1}$$

$$= 2^{r_1+|\tau_2|} = 2^{|\tau|}$$

- $\tau = \tau_1 \cap (\tau_2 \cup \tau_3)$. We have $N_L(\tau_1 \cap (\tau_2 \cup \tau_3)) = N_L(\tau_1 \cap \tau_2) \cup N_L(\tau_1 \cap \tau_3)$. So

$$|N_L(\tau_1 \cap (\tau_2 \cup \tau_3))| = 1 + |N_L(\tau_1 \cap \tau_2)| + |N_L(\tau_1 \cap \tau_3)|$$

$$\leq 1 + 2^{r_1+|\tau_2|+1} + 2^{r_1+|\tau_3|+1}$$

$$\leq 1 + 2^{1+|\tau_1|}(2^{r_2} + 2^{r_3})$$

$$\leq 1 + 2^{1+|\tau_1|}(2^{\max(|\tau_2|,|\tau_3|)+1})$$

$$= 1 + 2^{r_1+|\tau_2|+\max(|\tau_2|,|\tau_3|)+2}$$

$$\leq 2^{r_1+|\tau_2|+|\tau_3|+2} = 2^{|\tau| \cap (\tau_2 \cup \tau_3)|}$$

166
• The case $\tau = \bigcap_{i=1}^{m} F(\tau_1^i, \ldots, \tau_n^i)$. We have:

$$|N_L(\bigcap_{i=1}^{m} F(\tau_1^i, \ldots, \tau_n^i))| = |F(N_L(\bigcap_{i=1}^{m} \tau_1^i), \ldots, N_L(\bigcap_{i=1}^{m} \tau_n^i))|$$

$$= 1 + \sum_{j=1}^{n} |(N_L(\bigcap_{i=1}^{m} \tau_j^i))|$$

$$\leq 1 + \sum_{j=1}^{n} 2^{m-1} \cdot \sum_{i=1}^{m} |\tau_j^i| \leq 1 + n \cdot \max_{j} (2^{m-1} + \sum_{i=1}^{m} |\tau_j^i|)$$

$$= 1 + n \cdot 2^{m-1} \max_{j} (2 \sum_{i=1}^{m} |\tau_j^i|)$$

$$\leq 1 + 2^{m-1} \cdot \max_{j} (2 \sum_{i=1}^{m} |\tau_j^i| \cdot 2 \sum_{i \neq k} \sum_{i=1}^{m} |\tau_j^i|)$$

$$\leq 1 + 2^{m-1} \cdot 2 \sum_{i=1}^{m} \sum_{i=1}^{m} |\tau_j^i|$$

$$\leq 1 + 2^{m-1} + \sum_{i=1}^{m} \sum_{i=1}^{m} |\tau_j^i| = 1 + 2 \big( \bigcap_{i=1}^{m} F(\tau_1^i, \ldots, \tau_n^i) \big)$$

where $k_0$ is such that $\sum_{i=1}^{m} |\tau_j^i|$ reaches the maximum of $\sum_{i=1}^{m} |\tau_j^i|$. For the function $N_R$ the proof is similar.

\[\square\]

**Lemma 4.46.** Let $\tau, \sigma$ be types. Let $k$ be any integer. If the basic type formula $\vdash \bigcap_{i=1}^{m} \alpha_i \leq \bigcup_{i=1}^{n} \beta_i$ is a subformula of $SF(l_k(\tau \leq \sigma))$ then for each $i, j$ we have $\alpha_i = b_j$.

**Proof.** The proof proceeds by induction on the definition of $SF$. Consider two cases depending whether the case (4.17) or (4.18) of definition of $SF$ has been applied.

• case (4.17). We have:

$$SF(l_k(\tau_1 \cup \cdots \cup \tau_n \leq \sigma_1 \cap \cdots \cap \sigma_m))$$

is equivalent to

$$SF(l_k(\tau_1 \cup \cdots \cup \tau_n) \leq l_k(\sigma_1 \cap \cdots \cap \sigma_m))$$

which, by definition of $l_k$ is equivalent to

$$SF(l_k(\tau_1) \cup \cdots \cup l_k(\tau_n) \leq l_k(\sigma_1) \cap \cdots \cap l_k(\sigma_m))$$

This is equivalent to

$$\bigwedge_{i,j} SF(l_k(\tau_i \leq \sigma_j))$$

By the inductive hypothesis we have that all basic formulas in each $SF(l_k(\tau_i \leq \sigma_j))$ have equal indexes, so all basic formulas in $SF(l_k(\tau_1 \cup \cdots \cup \tau_n \leq \sigma_1 \cap \cdots \cap \sigma_m))$ have also equal indexes.

• case (4.18). We have:

$$SF(l_k(\bigcap_{i=1}^{r} \alpha_i \cap F(\tau) \leq \bigcup_{i=1}^{m} \beta_i \cup \bigcup_{i=1}^{n} F(\tau_i) \cup \bigcup_{i=1}^{p} G_i(\sigma_i)))$$

(A.42)
Appendix A. Proofs

It is equivalent to

\[
SF(l_k \bigcap_{i=1}^{r} \alpha_i \cap F(\tau)) \leq l_k \bigcup_{i=1}^{m} \beta_i \cup \bigcup_{i=1}^{n} F(\tau_i) \cup \bigcup_{i=1}^{p} G_i(\sigma_i))
\]

This, by definition of \(l_k\) gives

\[
SF(\bigcap_{i=1}^{r} \alpha_i^k \cap F(\tau)) \leq \bigcup_{i=1}^{m} \beta_i^k \cup \bigcup_{i=1}^{n} l_k(F(\tau_i)) \cup \bigcup_{i=1}^{p} l_k(G_i(\sigma_i))
\]

and

\[
SF(\bigcap_{i=1}^{r} \alpha_i^k \cap F(l_{k+1}(\tau)) \leq \bigcup_{i=1}^{m} \beta_i^k \cup \bigcup_{i=1}^{n} F(l_{k+1}(\tau_i)) \cup \bigcup_{i=1}^{p} G_i(l_{k+1}(\sigma_i))
\]

By definition of SF the last formula is equivalent to

\[
\bigcap_{i=1}^{r} \alpha_i^k \leq \bigcup_{i=1}^{m} \beta_i^k \text{ or } \bigvee_i SF(l_{k+1}(\tau)) \leq l_{k+1}(\tau_i)
\]

and, finally

\[
\bigcap_{i=1}^{r} \alpha_i^k \leq \bigcup_{i=1}^{m} \beta_i^k \text{ or } \bigvee_i \sigma_i \leq l_{k+1}(\tau_i)
\]

The inductive hypothesis implies that all basic formulas in \(SF(l_{k+1}(\tau \leq \tau_i))\) have desired form. Of course the same property has also \(\bigcap_{i=1}^{r} \alpha_i^k \leq \bigcup_{i=1}^{m} \beta_i^k\), so each basic type formula in (A.42) have variables with equal indexes.

\[\Box\]

Lemma 4.51. Let \(A'\) be any function mapping types to types. If

(A.43) \(\vdash A'(\tau_1 \cap \tau_2) = A'(\tau_1) \cap A'(\tau_2)\)

and

(A.44) \(\vdash \tau = \sigma \implies \vdash A'(\tau) = A'(\sigma)\)

then for any types \(\sigma_1, \sigma_2\) we have

(A.45) \(\vdash \sigma_1 \leq \sigma_2 \implies \vdash A'(\sigma_1) \leq A'(\sigma_2)\)

Proof. Assume that \(\vdash \sigma_1 \leq \sigma_2\). This implies \(\vdash \sigma_1 = \sigma_1 \cap \sigma_2\), and, by (A.43) \(A'(\sigma_1) = A'(\sigma_1 \cap \sigma_2)\). Now, the thesis follows by (A.44).

\[\Box\]

Lemma 4.56. Let \(\tau_0\) be a type. Let \(\tau_i \Rightarrow \tau_{i+1}\), for \(i \in \{0, \ldots, k\}\) for some \(k\). Then for each \(i\) we have

(A.46) \(\vdash \tau_i \leq A(\tau_0)\)

Proof. First we prove that

(A.47) \(\vdash \tau_i \leq A(\tau_0) \Rightarrow \vdash \tau_{i+1} \leq A(\tau_0)\)

Suppose that

\(\vdash \tau_i \leq A(\tau_0)\)

Consider two possibilities.

168
1. \( \vdash \tau_{i+1} = \tau_i \). A simple or a distributive rule has been used. So, the thesis is obvious.

2. The discriminative rule has been used. Let \( F(\sigma_1) \cup F(\sigma_2) \) be the part of \( \tau_i \) which was changed by the \( \Rightarrow_{u} \). Let \( \sigma'_1, \sigma'_2 \) are the R-reduced forms of \( \sigma_1, \sigma_2 \). Let \( \tau'_i \) be the R-reduced form of \( \tau_i \), and similarly let \( \tau'_{i+1} \) be the R-reduced form of \( \tau_{i+1} \). On the analogous positions in \( \tau'_i, \tau'_{i+1} \) and \( A(\tau_i) \) we have \( F(\sigma'_1) \cup F(\sigma'_2) \), \( \tau'_i \) and \( F(\sigma_1 \cup \sigma_2) \) in \( \tau'_{i+1} \), and a type which is greater than \( F(\sigma_1 \cup \sigma_2) \) in \( A(\tau_i) \). So, by Lemma 4.53 we have that \( \vdash A(\tau_i) \leq \tau_{i+1} \). So, we have

\[
\vdash \tau_i \leq A(\tau_0) \Rightarrow A(\tau_i) \leq A(A(\tau_0)) \\
\Rightarrow \vdash A(\tau_i) \leq A(\tau_0) \\
\Rightarrow \vdash \tau_{i+1} \leq A(\tau_0)
\]

which finishes the proof of (A.47). From (A.47) and Lemma 4.54 for \( i = 0 \) comes by simple induction the thesis of the lemma.

\[\Box\]

**Lemma 4.70.** Let \( \tau_1, \ldots, \tau_n \) be types, \( \rho_1, \ldots, \rho_n, \sigma \) be assumption types. Then

\[
\bigcup_{i=1}^{n} BS(\tau_i, \rho_i)(\sigma) \leq (BS(\tau_1, \rho_1) \sqcup \cdots \sqcup BS(\tau_n, \rho_n))(\sigma)
\]

**Proof.** The proof proceeds by induction on the structure of \( \sigma \). If \( \sigma \) is a variable then the thesis follows directly by Remark 4.69. So, suppose that \( \sigma = F(\sigma_1, \ldots, \sigma_n) \), and the thesis holds for each \( \sigma_i \). Let

\[
\theta_i = BS(\tau_i, \rho_i)
\]

for \( i \in \{1, \ldots, n\} \). We have

\[
\left( \bigcup_{i=1}^{n} \theta_i \right)(\sigma) = \\
= \left( \bigcup_{i=1}^{n} \theta_i \right)(F(\sigma_1, \ldots, \sigma_n)) \\
= F(\left( \bigcup_{i=1}^{n} \theta_i \right)(\sigma_1), \ldots, \left( \bigcup_{i=1}^{n} \theta_i \right)(\sigma_n)) \quad \text{by ind. hyp. and Lem. 2.21} \\
\geq F(\bigcup_{i=1}^{n} \theta_i(\sigma_1), \ldots, \theta_i(\sigma_n)) \\
\geq \bigcup_{i=1}^{n} F(\theta_i(\sigma_1), \ldots, \theta_i(\sigma_n)) \\
= \bigcup_{i=1}^{n} \theta_i(F(\sigma_1, \ldots, \sigma_n)) = \bigcup_{i=1}^{n} \theta_i(\sigma)
\]

\[\Box\]

**Lemma 4.71.** Assume that \( \vdash \tau_1 = \tau_2 \). Then, for any assumption type \( \sigma \) without \( \top \), and for any variable \( \alpha \in \text{var}(\sigma) \)

\[
\vdash^* BS(\tau_1, \sigma)(\alpha) = BS(\tau_2, \sigma)(\alpha)
\]

169
Proof. We show the thesis by induction on the structure of the proof of $\vdash \tau_1 = \tau_2$. The proof is routine and we show only cases (Ax1), (Ax2), (Ax12).

(Ax1) Obvious. Types $\tau_1$ and $\tau_2$ are equal, so $BS(\tau_1, \sigma)(\alpha)$ and $BS(\tau_2, \sigma)(\alpha)$ are also equal.

(Ax2) We have $\vdash F(\tau_1, \ldots, \tau_n) = F(\tau'_1, \ldots, \tau'_k)$ with assumptions $\vdash \tau_i = \tau'_i$ for $i \in \{1, \ldots, n\}$. First assume that $\sigma$ is a type variable $\alpha$. Then we have

$$\vdash^* BS(F(\tau_1, \ldots, \tau_n), \alpha)(\alpha) = \{\alpha := F(\tau_1, \ldots, \tau_n)\}(\alpha)$$

$$= F(\tau_1, \ldots, \tau_n) = F(\tau'_1, \ldots, \tau'_k) = BS(F(\tau'_1, \ldots, \tau'_k), \alpha)(\alpha)$$

Now, assume that $\sigma = F(\sigma_1, \ldots, \sigma_n)$. Then we have:

$$\vdash^* BS(F(\tau_1, \ldots, \tau_n), \sigma)$$

$$= \left(\bigcup_{i=1}^{n} BS(\tau_i, \sigma_i)\right)(\alpha)$$

$$= \bigcup_{i=1}^{n} BS(\tau_i, \sigma_i)(\alpha) \quad \text{by Remark 4.69}$$

$$= \bigcup_{i=1}^{n} BS(\tau'_i, \sigma_i)(\alpha) \quad \text{by the inductive hypothesis}$$

$$= \left(\bigcup_{i=1}^{n} BS(\tau_i, \sigma_i)\right)(\alpha)$$

$$= BS(F(\tau_1, \ldots, \tau_n), \sigma)(\alpha)$$

Finally suppose that $\sigma = G(\sigma_1, \ldots, \sigma_n)$. Then

$$\vdash^* BS(F(\tau_1, \ldots, \tau_n), G(\sigma_1, \ldots, \sigma_n))(\alpha) = T^*$$

$$= BS(F(\tau'_1, \ldots, \tau'_k), G(\sigma_1, \ldots, \sigma_n))(\alpha)$$

(Ax2) There is a proof of $\vdash \tau_1 \cap \tau_2 = \tau'_1 \cap \tau'_2$ with assumptions $\vdash \tau_1 = \tau'_1$ and $\vdash \tau_2 = \tau'_2$. By the inductive hypothesis

$$\vdash^* BS(\tau_1, \sigma)(\alpha) = BS(\tau'_1, \sigma)(\alpha)$$

$$\vdash^* BS(\tau_2, \sigma)(\alpha) = BS(\tau'_2, \sigma)(\alpha)$$

So, we have

$$\vdash^* BS(\tau_1 \cap \tau_2, \sigma)(\alpha) = (BS(\tau_1, \sigma) \cap BS(\tau_2, \sigma))(\alpha) \quad \text{by Remark 4.69}$$

$$= BS(\tau_1, \sigma)(\alpha) \cap BS(\tau_2, \sigma)(\alpha)$$

$$= BS(\tau'_1, \sigma)(\alpha) \cap BS(\tau'_2, \sigma)(\alpha)$$

$$= (BS(\tau'_1, \sigma) \cap BS(\tau'_2, \sigma))(\alpha)$$

$$= BS(\tau'_1 \cap \tau'_2, \sigma)(\alpha)$$

(Ax12) We have $\tau \cap \bot = \bot$. We have:

$$BS(\tau \cap \bot, \sigma)(\alpha) = (BS(\tau, \sigma) \cap BS(\bot, \sigma))(\alpha)$$

$$= BS(\tau, \sigma)(\alpha) \cap \bot = \emptyset(\alpha) = BS(\bot, \sigma)(\alpha)$$

\[\square\]
Lemma A.1. Assume that \( \tau_1 \cap \tau_2 \) is a subterm of a term in \( L \)-reduced form. Let \( \sigma \) be an assumption type without \( \top \). Then

\[
(A.48) \quad \vdash \tau_1 \cap \tau_2 \leq \sigma
\]

if and only if \( \vdash \tau_1 \leq \sigma \) or \( \vdash \tau_2 \leq \sigma \)

Proof. The if direction is straightforward (by Lemma 4.21). To prove the second direction assume

\[
\vdash \tau_1 \cap \tau_2 \leq \sigma
\]

Since \( \tau_1 \cap \tau_2 \) is a subterm of the type in \( L \)-reduced form we have that

\[
(A.49) \quad \vdash \tau_1 \cap \tau_2 = \alpha_1 \cap \cdots \cap \alpha_k \cap F(\tau') \tag{A.49}
\]

\[
(A.50) \quad \tau_1 \cap \tau_2 = \alpha_1 \cap \cdots \cap \alpha_k \tag{A.50}
\]

If (A.50) holds then \( \sigma \) must be a variable and the thesis follows by Lemma 4.19.

So, assume that (A.49) holds. So, at most one of \( \tau_1, \tau_2 \) is a product with element \( F(\rho) \). Without any loss of generality assume that it is \( \tau_1 \). Now, consider two cases:

- \( \sigma = \beta \), then \( \beta \in \{\alpha_1, \ldots, \alpha_n\} \) and the thesis holds.
- \( \sigma = F(\sigma_1, \ldots, \sigma_n) \). In this case \( \neg(\vdash \tau_2 \leq \sigma) \) which implies \( \vdash ^* \tau_1 \leq \sigma \)

\[\blacksquare\]

Lemma 4.73. Assume that \( \theta = BS(\tau, \sigma) \). Then for each \( \theta' \) such that \( \tau \leq \theta'(\sigma) \) we have

\[
\vdash ^* \theta \leq \theta'
\]

Proof. By Lemma 4.71 and Lemma 4.9 we have that

\[
\theta = BS(\tau, \sigma) = BS(N_L(\tau), \sigma)
\]

So, we can assume that \( \tau \) is in \( L \)-reduced form. The proof of Lemma 4.73 proceeds by induction on the structure of a computation of \( BS(\tau, \sigma) \).

- \( \tau = \bot \). Obvious, since for each \( \alpha \) we have \( \vdash ^* \theta(\alpha) = \bot = \theta'(\alpha) \)
- \( \sigma = \alpha \). We have \( \theta = \emptyset \cup \{\alpha := \tau\} \). Let \( \theta' \) be any substitution for which \( \tau \leq \theta'(\alpha) \). So, \( \vdash ^* \theta(\alpha) \leq \tau \leq \theta'(\alpha) \). For any variable \( \beta \neq \alpha \) we have \( \vdash ^* \theta(\beta) = \bot \leq \theta'(\beta) \)
- \( \tau = \tau_1 \cup \tau_2 \). Let \( \theta_1 = BS(\tau_1, \sigma), \theta_2 = BS(\tau_2, \sigma) \), and let \( \theta_3 = BS(\tau_1 \cup \tau_2, \sigma) = \theta_1 \cup \theta_2 \). The induction hypothesis says:

\[
(A.51) \quad (\forall \theta'_1) \tau_1 \leq \theta'_1(\sigma) \Rightarrow \vdash ^* \theta_1 \leq \theta'_1
\]

\[
(A.52) \quad (\forall \theta'_2) \tau_2 \leq \theta'_2(\sigma) \Rightarrow \vdash ^* \theta_2 \leq \theta'_2
\]

Let us take any \( \theta'_3 \), such that

\[
(A.53) \quad \vdash ^* \tau_1 \cup \tau_2 \leq \theta'_3(\sigma)
\]

Formula (A.53) is equivalent to

\[
\tau_1 \leq \theta'_3(\sigma) \text{ and } \tau_2 \leq \theta'_3(\sigma)
\]

So, we can apply the induction hypothesis, replacing \( \theta'_1 \) by \( \theta'_3 \) in (A.51), and \( \theta'_2 \) by \( \theta'_3 \) in (A.52), to obtain:

\[
(\forall \alpha) \vdash ^* \theta_1(\alpha) \leq \theta'_3(\alpha)
\]

\[
(\forall \alpha) \vdash ^* \theta_2(\alpha) \leq \theta'_3(\alpha)
\]
Appendix A. Proofs

So, for each $\alpha$ we have

$$\vdash^* \theta_1(\alpha) \cup \theta_2(\alpha) \leq \theta'_3(\alpha)$$

which implies

$$\vdash^* (\theta_1 \cup \theta_2)(\alpha) \leq \theta'_3(\alpha)$$

So, $\vdash^* \theta_1 \cup \theta_2 = BS(\tau_1 \cup \tau_2, \sigma) \leq \theta'_3$ which finishes the proof of this case.

• $\tau = \tau_1 \cap \tau_2$. Let $\theta_1 = BS(\tau_1, \sigma), \theta_2 = BS(\tau_2, \sigma), \theta_3 = BS(\tau_1 \cap \tau_2, \sigma) = \theta_1 \cap \theta_2$

The induction hypothesis says:

\[(A.54) \quad (\forall \theta'_1) \tau_1 \leq \theta'_1(\sigma) \Rightarrow \theta_1 \leq \theta'_1 \]
\[(A.55) \quad (\forall \theta'_2) \tau_2 \leq \theta'_2(\sigma) \Rightarrow \theta_2 \leq \theta'_2 \]

Let $\theta'_3$ be any substitution, such that

\[(A.56) \quad \vdash^* \tau_1 \cap \tau_2 \leq \theta'_3(\sigma) \]

which, since $\tau_1 \cap \tau_2$ is L-reduced as a subterm of L-reduced type, and by Lemma A.1, is equivalent to

$$\tau_1 \leq \theta'_3(\sigma) \text{ or } \tau_2 \leq \theta'_3(\sigma)$$

We can apply the induction hypothesis substituting $\theta'_3$ for $\theta'_1$ in (A.54) and $\theta'_2$ in (A.55) to obtain

$$(\forall \alpha)\theta_1(\alpha) \leq \theta'_3(\alpha) \text{ or } (\forall \alpha)\theta_2(\alpha) \leq \theta'_3(\alpha)$$

Without any lost of generality assume that the first case holds. By Remark 4.69 for each $\alpha$

$$\vdash^* (\theta_1 \cap \theta_2)(\alpha) = \theta_1(\alpha) \cap \theta_2(\alpha) \leq \theta_1(\alpha) \leq \theta'_3(\alpha)$$

So, $BS(\tau_1 \cap \tau_2, \sigma) = \theta_1 \cap \theta_2 \leq \theta'_3$ which we wanted to prove.

• $\tau = F(\tau_1, \ldots, \tau_n), \sigma = F(\sigma_1, \ldots, \sigma_n)$. Let

$$\theta_i = BS(\tau_i, \sigma_i)$$

The induction hypothesis says that for each $i \in \{1, \ldots, n\}$:

\[(A.57) \quad (\forall \theta'_i) \tau_i \leq \theta'_i(\sigma_i) \Rightarrow \theta_i \leq \theta'_i \]

Let us take any substitution $\theta'$ such that

\[(A.58) \quad \vdash^* F(\tau_1, \ldots, \tau_n) \leq \theta' F(\sigma_1, \ldots, \sigma_n) \]

(A.58) is equivalent to

$$\vdash^* F(\tau_1, \ldots, \tau_n) \leq F(\theta'_1 \sigma_1, \ldots, \theta'_n \sigma_n)$$

The last formula implies that for each $i \in \{1, \ldots, n\}$

\[(A.59) \quad \tau_i \leq \theta'_i(\sigma_i) \]

Applying (A.59) to inductive hypothesis, substituting the substitution $\theta'$ for every $\theta'_i$, we obtain that, for each $i \in \{1, \ldots, n\}$

$$\theta_i \leq \theta'$$

172
Repeating the reasoning used in the case $\tau = \tau_1 \cup \tau_2$, we obtain that
\[
\bigcup_{i=1}^{n} \theta_i \leq \theta'
\]
This, by definition of $BS$ gives that
\[
BS(F(\tau_1, \ldots, \tau_n), F(\sigma_1, \ldots, \sigma_m)) \leq \theta'
\]
which completes the proof of this case.

- None of the cases above. We have that $\theta = \emptyset \cup \{\alpha := T^* | \alpha \in \var(\sigma)\}$. There are the following possibilities:
  1. $\tau = F(\tau_1, \ldots, \tau_n), \sigma = G(\sigma_1, \ldots, \sigma_m), F \neq G$
  2. $\tau$ is atomic, $\sigma = G(\sigma_1, \ldots, \sigma_m)$
  3. $\tau = T, \sigma = G(\sigma_1, \ldots, \sigma_m)$

Let us take any $\theta'$ such that
\[
\vdash^* \tau \leq \theta'(\sigma)
\]
This is possible only if $\vdash^* \theta'(G(\sigma_1, \ldots, \sigma_n)) = T^*$, so, for each variable $\alpha \in \var(\sigma)$ we have that $\vdash^* \theta(\alpha) \leq \theta'(\alpha) = T^*$. For $\beta \notin \var(\sigma)$ $\theta(\beta) = \bot \leq \theta'(\beta)$ which gives us the thesis.

**Lemma 4.77.** If $\Gamma \vdash t : \tau$ then $\vdash BT(\Gamma, t) \leq \tau$

**Proof.** The proof proceeds by induction on the structure of a proof of $\Gamma \vdash t : \tau$.

Consider the cases:

- (T1) $\Gamma \vdash t : T$. Of course $\vdash BT(\Gamma, t) \leq T$.
- (T2) $\Gamma \vdash t : \bot$, false $\in \Gamma$. Since $BT(\Gamma, t) = \bot$, the thesis follows.
- (T3) $\Gamma \vdash X : \Gamma(X)$. Since $\vdash BT(\Gamma, X) \leq \Gamma(X)$ the thesis follows.
- (T4) $\Gamma \vdash t : \tau$ is a conclusion by (T4) with assumptions $\Gamma \vdash t : \tau'$ and $\vdash \tau' \leq \tau$. By the inductive hypothesis we have $\vdash BT(\Gamma, t) \leq \tau'$. So, by transitivity of $\leq$ we obtain $\vdash BT(\Gamma, t) \leq \tau$
- (T5) $\Gamma \vdash t : \tau_1 \cap \ldots \cap \tau_n$ is a conclusion with assumptions $\Gamma \vdash t : \tau_i$, for $i \in \{1, \ldots, n\}$.

By the induction hypothesis,
\[
BT(\Gamma, t) \leq \tau_i \text{ for } i \in \{1, \ldots, n\}
\]

So,
\[
\vdash BT(\Gamma, t) \leq \bigcap_{i=1}^{n} \tau_i
\]

which finishes the proof.

- (T6) Assume that $f : \sigma_1, \ldots, \sigma_n \rightarrow \sigma$ is a signature of $f$. Then

\[
\Gamma \vdash f(t_1, \ldots, t_n) : \theta(\sigma)
\]

with assumptions
\[
\Gamma \vdash t_i : \theta(\sigma_i)
\]

for $i \in \{1, \ldots, n\}$. Let $\tau_i = BT(\Gamma, t_i)$. By the inductive hypothesis, for each $i \in \{1, \ldots, n\}$, we have
\[
\vdash \tau_i \leq \theta(\sigma_i)
\]
Appendix A. Proofs

Let \( \theta'_i = BS(\tau_i, \sigma_i) \). Let

\[
\theta' = \bigcup_{i=1}^n \theta'_i
\]

By Lemma 4.73 we have \( \theta'_i \leq \theta \), and

\[
\vdash \theta'(\sigma) \leq \theta(\sigma)
\]

which, by \( \theta'(\sigma) = BT(\Gamma, f(t_1, \ldots, t_n)) \) gives the thesis.

\[\square\]

Lemma 4.92. Let \( \Gamma_1, \Gamma_2 \) be environments. We have

\[
s(\Gamma_1 \sqcup \Gamma_2) \leq s(\Gamma_1) + s(\Gamma_2)
\]

Proof. If \( \Gamma_1 = \{\text{false}\} \) then the thesis is obvious since \( \Gamma_1 \sqcup \Gamma_2 = \Gamma_2 \). Similarly when \( \Gamma_2 = \{\text{false}\} \). So, suppose that both \( \Gamma_1, \Gamma_2 \) are different than \( \{\text{false}\} \). Let

\[
V_1 = \{X : (X : \tau) \in \Gamma_1\}
\]

\[
V_2 = \{X : (X : \tau) \in \Gamma_2\}
\]

It is a simple corollary from the definition of \( \sqcup \) that

\[
(\text{A.60}) \quad \Gamma_1 \sqcup \Gamma_2 = \{(X : \Gamma_1(X) \cup \Gamma_2(X)) | X \in V_1 \cap V_2\}
\]

So, we have

\[
s(\Gamma_1 \sqcup \Gamma_2) = \sum_{X \in V_1 \cap V_2} s(\Gamma_1(X) \cup \Gamma_2(X))
\]

\[
= \sum_{X \in V_1 \cap V_2} s(\Gamma_1(X)) + \sum_{X \in V_1 \cap V_2} s(\Gamma_2(X))
\]

\[
\leq \sum_{X \in V_1} s(\Gamma_1(X)) + \sum_{X \in V_2} s(\Gamma_2(X))
\]

\[
= s(\Gamma_1) + s(\Gamma_2)
\]

\[\square\]

Lemma 4.93. Let \( \Gamma_1, \Gamma_2 \) be type environments. We have

\[
s(\Gamma_1 \sqcap \Gamma_2) \leq s(\Gamma_1) + s(\Gamma_2)
\]

Proof. If \( \Gamma_1 = \{\text{false}\} \) then the thesis is obvious since \( \Gamma_1 \sqcap \Gamma_2 = \{\text{false}\} \), and \( s(\{\text{false}\}) = 0 \). Similarly when \( \Gamma_2 = \{\text{false}\} \). So, assume that both \( \Gamma_1, \Gamma_2 \) are different than \( \{\text{false}\} \). Let

\[
V_1 = \{X : (X : \tau) \in \Gamma_1\}
\]

\[
V_2 = \{X : (X : \tau) \in \Gamma_2\}
\]

It is a simple corollary from the definition of \( \sqcap \) that

\[
\Gamma_1 \sqcap \Gamma_2 = \{(X : \Gamma_1(X) \cap \Gamma_2(X)) | X \in V_1 \cap V_2\} \cup
\]

\[
\{(X : \Gamma_1(X)) | X \in V_1 \setminus V_2\} \cup
\]

\[
\{(X : \Gamma_2(X)) | X \in V_2 \setminus V_1\}
\]

174
So, we have
\[
s(\Gamma_1 \sqcup \Gamma_2) = \sum_{X \in V_1 \cap V_2} s(\Gamma_1(X) \cap \Gamma_2(X)) \\
+ \sum_{X \in V_1 \setminus V_2} s(\Gamma_1(X)) + \sum_{X \in V_2 \setminus V_1} s(\Gamma_2(X)) \\
= \sum_{X \in V_1 \cap V_2} s(\Gamma_1(X)) + \sum_{X \in V_2 \cap V_2} s(\Gamma_2(X)) \\
+ \sum_{X \in V_1 \setminus V_2} s(\Gamma_1(X)) + \sum_{X \in V_2 \setminus V_1} s(\Gamma_2(X)) \\
= \sum_{X \in V_1} s(\Gamma_1(X)) + \sum_{X \in V_2} s(\Gamma_2(X)) \\
= s(\Gamma_1) + s(\Gamma_2)
\]

\[\square\]

**Lemma 4.97.** Let \( t \) be a term, \( \tau \) be a type and \( \Gamma \) be an environment. If \( \Gamma, t : \tau \Rightarrow \Gamma' \) then \( \emptyset, t : \tau \Rightarrow \Gamma^0 \), and \( \Gamma' = \Gamma \cap \Gamma^0 \).

**Proof.** We will show that by induction on the structure of the proof of \( \Gamma, t : \tau \Rightarrow \Gamma' \). Cases of \((K_1), (K_2)\) and \((K_7)\) are similar, and we shall present only one of them — \((K_2)\).

\((K_2)\) We have
\[
\Gamma, f(t_1, \ldots, t_n) : \theta(\mathcal{N}(\sigma)) \Rightarrow \Gamma'
\]
where \( f \) has a signature \( \sigma_1 \ast \cdots \ast \sigma_n \rightarrow \sigma \). So, for each \( i \in \{1, \ldots, n\} \)
\[
\Gamma; t_i : \theta(\sigma_i) \Rightarrow \Gamma_i
\]
and
\[\text{(A.61)}\]
\[
\Gamma' = \Gamma_1 \cap \cdots \cap \Gamma_n
\]
By the induction hypothesis we have that for each \( i \in \{1, \ldots, n\} \)
\[\text{(A.62)}\]
\[
\emptyset, t_i : \theta(\sigma_i) \Rightarrow \Gamma_i^0
\]
and
\[\text{(A.63)}\]
\[
\Gamma_i = \Gamma \cap \Gamma_i^0
\]
So, applying the rule \((K_2)\) to \((A.62)\) and \((A.63)\) we obtain
\[
\emptyset, f(t_1, \ldots, t_n) : \theta(\mathcal{N}(\sigma)) \Rightarrow \Gamma^0
\]
and \( \Gamma^0 = \Gamma_1^0 \cap \cdots \cap \Gamma_n^0 \) which together with \((A.61)\) implies
\[
\Gamma' = \Gamma_1 \cap \cdots \cap \Gamma_n
\]
\[
= (\Gamma \cap \Gamma_1^0) \cap \cdots \cap (\Gamma \cap \Gamma_n^0)
\]
\[
= \Gamma \cap (\Gamma_1^0 \cap \cdots \cap \Gamma_n^0)
\]
\[
= \Gamma \cap \Gamma^0.
\]

\((K_3)\) We have that \( \Gamma^0 = \Gamma' = \{\text{false}\} \). So \( \Gamma^0 \cap \Gamma = \{\text{false}\} = \Gamma' \).
(K₄) As above.

(K₅) We have \( \Gamma^0 = \emptyset \) and \( \Gamma' = \Gamma \). Hence, \( \Gamma' = \Gamma \cap \Gamma^0 \).

(K₆) As above.

(K₇) In the proof of \( \Gamma, t : \tau_1 \cup \cdots \cup \tau_n \Rightarrow \Gamma' \) we have

\[
\Gamma, \ (t : \tau_1) \Rightarrow \Gamma'_1, \ldots, \Gamma, \ (t : \tau_n) \Rightarrow \Gamma'_n
\]

and \( \Gamma' = \Gamma'_1 \cup \cdots \cup \Gamma'_n \). By the inductive hypothesis, for each \( i \in \{1, \ldots, n\} \)

\[
\emptyset, \ (t : \tau_i) \Rightarrow \Gamma'_i
\]

and \( \Gamma_i = \Gamma \cap \Gamma'_i \). By rule (K₇) we get also \( \Gamma^0 = \Gamma'_1 \cup \cdots \cup \Gamma'_n \). So, by Lemma 2.44 we obtain

\[
\Gamma' = \Gamma'_1 \cup \cdots \cup \Gamma'_n
\]

\[
= (\Gamma \cap \Gamma'_1) \cup \cdots \cup (\Gamma \cap \Gamma'_n)
\]

\[
= \Gamma \cap (\Gamma'_1 \cup \cdots \Gamma'_n)
\]

\[
= \Gamma \cap \Gamma^0
\]

(K₉) Let \( t = X \). We have that \( \Gamma' = \Gamma \cap \{X : \tau\} \). We have that \( \emptyset, X : \tau \Rightarrow \{X : \tau\} \). So, \( \Gamma' = \Gamma \cap \{\{X : \tau\}\} \).

\[\square\]

**Lemma 4.98.** If \( \emptyset, t : \tau \Rightarrow \Gamma' \) then \( s(\Gamma') \leq s(\tau) \cdot |t| \)

**Proof.** We will show that by induction on the structure of the proof \( \emptyset, t : \tau \Rightarrow \Gamma' \). The proofs in the case of \( (K_1), (K_2) \) and \( (K_7) \) are similar and we present only one of them.

(K₂) We have that

\[
\emptyset, f(t_1, \ldots, t_n) : \theta(\mathcal{R}(\sigma)) \Rightarrow \Gamma'
\]

where \( f \) has a signature \( \sigma_1 \ast \cdots \ast \sigma_n \rightarrow \sigma \). So, for each \( i \in \{1, \ldots, n\} \)

\[
\emptyset, t_i : \theta(\sigma_i) \Rightarrow \Gamma_i
\]

and \( \Gamma' = \Gamma_1 \cap \cdots \cap \Gamma_n \). By the inductive hypothesis we have that for each \( i \in \{1, \ldots, n\} \)

\[
s(\Gamma_i) \leq |t_i| \cdot s(\theta(\sigma_i))
\]

Hence

\[
s(\Gamma') = s(\Gamma_1 \cap \cdots \cap \Gamma_n)
\]

\[
\leq \sum_{i=1}^{n} s(\Gamma_i)
\]

by corollary 4.95

\[
\leq \sum_{i=1}^{n} |t_i| \cdot s(\theta(\sigma_i))
\]

by inductive hypothesis

\[
\leq \sum_{i=1}^{n} |t_i| \cdot s(\theta(\mathcal{R}(\sigma)))
\]

since \( (\forall i)s(\theta(\mathcal{R}(\sigma))) \geq s(\theta(\sigma_i)) \)

\[
= s(\theta(\mathcal{R}(\sigma))) \sum_{i=1}^{n} |t_i|
\]

\[
\leq s(\tau) \cdot |t|
\]
In this case \( s(Γ') = s(\{ \text{false} \}) = 0 \) and the thesis follows since both \(|t|\) and \(s(τ)\) are positive.

As above.

In this case \( Γ = \emptyset \), and \( s(Γ') = 0 \) as in the previous cases.

As above.

Since \( \emptyset, t : τ_1 \cup \cdots \cup τ_n \Rightarrow Γ \), we have \( \emptyset, t : τ_1 \Rightarrow Γ_i \) for each \( i \in \{1, \ldots, n\} \) and \( Γ = Γ_1 \cup \cdots \cup Γ_n \). By induction hypothesis we have that for each \( i \in \{1, \ldots, n\} \)

\[
s(Γ_i) \leq s(τ_i) \cdot |t|
\]

So,

\[
s(Γ) = s(Γ_1 \cup \cdots \cup Γ_n)
\]

\[
\leq \sum_{i=1}^{n} s(Γ_i)
\]

by corollary 4.94

\[
\leq \sum_{i=1}^{n} s(τ_i) \cdot |t|
\]

by inductive hypothesis

\[
= |t| \sum_{i=1}^{n} s(τ_i)
\]

\[
= |t| \cdot s(τ_1 \cup \cdots \cup τ_n)
\]

We have that \( Γ = \{ (X : τ) \} \). Then, \( s(Γ) = s(τ) = s(τ) \cdot |X| \).

\[\blacksquare\]

**Lemma 4.100.** Suppose that \( T^* \) do not appear in the range of \( BS(τ, σ) \). Then

\[
s(BS(τ, σ)) \leq s(τ)
\]

**Proof.** The proof is by induction on the structure of computation of \( BS \).

- \( τ = \bot \). Then \( s(BS(τ, σ)) = s(\emptyset) = 0 \leq s(τ) \).
- \( σ = α \). Then \( s(BS(τ, α)) = s(\{ α := τ \}) = s(τ) \)
- \( τ = τ_1 \cup τ_2 \). By Lemma 4.96 we have

\[
s(BS(τ_1 \cup τ_2, σ)) = s(BS(τ_1, σ) \cup BS(τ_2, σ))
\]

\[
\leq s(BS(τ_1, σ)) + s(BS(τ_2, σ))
\]

\[
\leq s(τ_1) + s(τ_2)
\]

\[
= s(τ_1 \cup τ_2)
\]

- \( τ = τ_1 \cap τ_2 \). As above.
- \( τ = F(τ_1, \ldots, τ_n), σ = F(σ_1, \ldots, σ_n) \) We have

\[
s(BS(F(τ_1, \ldots, τ_n), F(σ_1, \ldots, σ_n)))
\]

\[
= s(\bigcup_{i=1}^{n} BS(τ_i, σ_i)) \leq \sum_{i=1}^{n} s(BS(τ_i, σ_i))
\]

\[
\leq \sum_{i=1}^{n} s(τ_i) \leq s(F(τ_1, \ldots, τ_n))
\]

Other cases are excluded since \( T^* \) does not appear in the range of \( BS(τ, σ) \).

\[\blacksquare\]

**Lemma 4.101.** Assume that \( τ = BT(Γ, t) \). Then \( s(τ) \leq \max(1, s(Γ)) \cdot |t| \)

177
Appendix A. Proofs

Proof. The proof is by induction on the structure of computation of $BT$. Consider four cases.
- $false \in \Gamma$. Then $\tau = \bot$, and $s(\bot) \leq \max(1, s(\Gamma)) \leq \max(1, s(\Gamma)) \cdot |t|$
- $t = X$. Then $s(\tau) \leq s(\Gamma(X)) \leq \max(1, s(\Gamma)) = \max(1, s(\Gamma)) \cdot |X|
- $t = f(t_1, \ldots, t_n)$. Suppose that $\tau_i = BT(\Gamma, t_i)$ for each $i \in \{1, \ldots, n\}$. The induction hypothesis for the types $\tau_i$ is

\[(A.65) \quad s(\tau_i) \leq \max(1, s(\Gamma)) \cdot |t_i|\]

Let $\sigma_1 \cdots \sigma_n \rightarrow \sigma$ be the signature of $f$. Let

\[(A.66) \quad \tau = \theta(\sigma)\]

where $\theta = \bigcup_{i=1}^{n} BS(\tau_i, \sigma_i)$. By Lemma 4.100 and Lemma 4.96 we have

\[(A.67) \quad s(\theta) \leq \sum_{i=1}^{n} s(\tau_i)\]

So, we have:

\[
1 + s(\theta) \leq 1 + \sum_{i=1}^{n} s(\tau_i)
\]

\[
\leq 1 + \sum_{i=1}^{n} \max(1, s(\Gamma)) \cdot |t_i| \quad \text{by induction hypothesis}
\]

\[
\leq 1 + \max(1, s(\Gamma)) \sum_{i=1}^{n} |t_i|
\]

\[
\leq \max(1, s(\Gamma)) + \max(1, s(\Gamma)) \cdot \sum_{i=1}^{n} |t_i|
\]

\[
= \max(1, s(\Gamma)) \cdot |f(t_1, \ldots, t_n)|
\]

Since $\sigma$ is linear we have $s(\tau) = s(\theta(\sigma)) \leq 1 + s(\theta)$ and the thesis is follows.
- $\tau = \top$. The proof is the same as in the case ($\tau = \bot$).

$\square$

Lemma 4.106. If $\varnothing, t : \tau \Rightarrow \Gamma'$ then $h(\Gamma') \leq h(\tau)$

Proof. We will show that by induction on the structure of the proof of $\varnothing, t : \tau \Rightarrow \Gamma'$. The proofs in the case of $(K_1)$, $(K_2)$ and $(K_7)$ are similar. That is why we present only one of them — $(K_2)$.

$(K_2)$ We have that

\[
\varnothing, f(t_1, \ldots, t_n) : \theta(\theta(\sigma)) \Rightarrow \Gamma'
\]

where $f$ has a signature $\sigma_1 \cdots \sigma_n \rightarrow \sigma$. So, we have that for each $i \in \{1, \ldots, n\}$

\[
\varnothing, t_i : \theta(\sigma_i) \Rightarrow \Gamma_i
\]

and $\Gamma' = \Gamma_1 \cap \cdots \cap \Gamma_n$. By the inductive hypothesis we have that for each $i \in \{1, \ldots, n\}$

\[
h(\Gamma_i) \leq h(\theta(\sigma_i))
\]
Hence we have
\[
h(\Gamma') = h(\Gamma_1 \cap \cdots \cap \Gamma_n) = \max_{i=1}^{n} h(\Gamma_i) = \max_{i=1}^{n} h(\theta(\sigma_i)) \quad \text{by inductive hypothesis} \\
\leq 1 + h(\theta) \\
\leq h(\theta(\mathbb{N}(\sigma))) = h(\tau)
\]

\((K_4)\) In this case \(h(\Gamma') = h(\{\text{false}\}) = 0\) and the thesis is proven since both \(h(t)\) and \(h(\tau)\) are not negative.

\((K_5)\) As above.

\((K_6)\) In this case \(\Gamma' = \emptyset\) and \(h(\Gamma') = 0\) as in the previous cases.

\((K_7)\) As above.

\((K_8)\) Because \(\emptyset, t : \tau_1 \cup \cdots \cup \tau_n \Rightarrow \Gamma\) we have that for each \(i \in \{1, \ldots, n\}\)
\[
\emptyset, t : \tau_i \Rightarrow \Gamma_i
\]
and \(\Gamma = \Gamma_1 \cup \cdots \cup \Gamma_n\). By inductive hypothesis we have that for each \(i \in \{1, \ldots, n\}\)
\[
h(\Gamma_i) \leq h(\tau_i)
\]
So,
\[
h(\Gamma) = h(\Gamma_1 \cup \cdots \cup \Gamma_n) = \max_{i=1}^{n} h(\Gamma_i) \leq \max_{i=1}^{n} h(\tau_i) \quad \text{by inductive hypothesis} \\
= h(\tau_1 \cup \cdots \cup \tau_n)
\]

\((K_9)\) We have that \(\Gamma = \{(X : \tau)\}\). So, \(h(\Gamma) = h(\tau)\).

\clearpage

\textbf{Lemma 4.108.} Suppose that \(\top^*\) does not appear in the range of \(BS(\tau, \sigma)\). Then \(h(\mathbb{S}(\tau, \sigma)) \leq h(\tau)\)

\textbf{Proof.} The proof proceeds by induction on the structure of computation of \(BS\).

\begin{itemize}
  \item \(\tau = \bot\). We have \(h(\mathbb{S}(\tau, \sigma)) = h(\emptyset) = 0 \leq h(\tau)\)
  \item \(\sigma = \alpha\). We have \(h(\mathbb{S}(\tau, \alpha)) = h(\{\alpha := \tau\}) = h(\tau)\)
  \item \(\tau = \tau_1 \cup \tau_2\). By the Lemma 4.96 we have
    \[
    h(\mathbb{S}(\tau_1 \cup \tau_2, \sigma)) = h(\mathbb{S}(\tau_1, \sigma) \cup \mathbb{S}(\tau_2, \sigma)) \leq \max(\mathbb{S}(\tau_1, \sigma), \mathbb{S}(\tau_2, \sigma)) \\
    \leq \max(h(\tau_1), h(\tau_2)) = h(\tau_1 \cup \tau_2)
    \]
  \item \(\tau = \tau_1 \cap \tau_2\). As above.
\end{itemize}
Appendix A. Proofs

• $\tau = F(\tau_1, \ldots, \tau_n), \sigma = F(\sigma_1, \ldots, \sigma_n)$.

\[
h(\mathcal{BS}(F(\tau_1, \ldots, \tau_n), F(\sigma_1, \ldots, \sigma_n))) \\
= h\left(\bigcup_{i=1}^{n} \mathcal{BS}(\tau_i, \sigma_i)\right) \\
\leq \max_{i=1}^{n} h(\mathcal{BS}(\tau_i, \sigma_i)) \\
\leq \max_{i=1}^{n} h(\tau_i) \\
\leq 1 + \max_{i=1}^{n} h(\tau_i) = h(F(\tau_1, \ldots, \tau_n))
\]

Other cases are excluded since there are no $\top^*$ in the range of $\mathcal{BS}(\tau, \sigma)$.

\[\square\]

**Lemma 4.109.** Suppose that $\tau = \mathcal{BT}(\Gamma, t)$. Then $h(\tau) \leq h(\Gamma) + h(t)$.

**Proof.** The proof proceeds by induction on the structure of computation of $\mathcal{BT}$. Consider four cases.

• $\text{false} \in \Gamma$. Then $\tau = \bot$, and $h(\bot) = 0 \leq h(\Gamma) + h(t)$.
• $t = X$. There we have $h(\tau) \leq h(\Gamma(X)) \leq h(\Gamma) = h(\Gamma) + h(X)$
• $t = f(t_1, \ldots, t_n)$. Suppose that $\tau_i = \mathcal{BT}(\Gamma, t_i)$ for each $i \in \{1, \ldots, n\}$. The induction hypothesis for the types $\tau_i$ is

(A.68) \[h(\tau_i) \leq h(\Gamma) + h(t_i)\]

Suppose that $f$ has a signature $\sigma_1, \ldots, \sigma_n \rightarrow \sigma$. Let

(A.69) \[\tau = \theta(\sigma)\]

where $\theta = \bigsqcup_{i=1}^{n} \mathcal{BS}(\tau_i, \sigma_i)$. By Lemma 4.108 we have

(A.70) \[h(\theta) \leq \max_{i=1}^{n} h(\tau_i)\]

So, we have:

\[
h(\mathcal{BT}(\Gamma, t)) = h(\theta(\sigma)) \\
\leq h(\theta) + 1 \\
\leq 1 + \max_{i=1}^{n} h(\tau_i) \\
\leq 1 + \max_{i=1}^{n} (h(\Gamma) + h(t_i)) \\
= h(\Gamma) + 1 + \max_{i=1}^{n} h(t_i) \\
= h(\Gamma) + h(f(t_1, \ldots, t_n))
\]

which was to be shown.

• $\tau = \top$. The proof is the same as in the first case ($\tau = \bot$).

\[\square\]

A.4 Proofs for Chapter 6

A.4.1 Properties of System P

All the lemmas presented in Chapter 2 remain true in System P. All proofs, except mentioned below, remain unchanged.
Proof of Lemma 2.23  We omit the case of (Ax14).

Proof of Lemma 2.49  We substitute the case of $(K_3)$ by the following.
$(K_3^P)$  Obviously $\Gamma \leq \Gamma'$.

Proof of Lemma 2.50  We omit the case of (Ax14).

Proof of Lemma 2.51  We substitute the case of $(K_3)$ by the following.
$(K_3^P)$  Environments remain unchanged, thus $\Gamma_1' \leq \Gamma_2'$.

Proof of Lemma 2.65  We substitute the case of $(K_3)$ by the following.
$(K_3^P)$  Environments remain unchanged, and $\theta(\Gamma) \leq \theta(\Gamma)$ holds obviously.

A.4.2  Semantics

The proof of the soundness of the system $3$ remain unchanged, with the following exceptions.

- We omit the proof of the correctness of axiom (Ax14).
- The proof of the correctness of the rule $(K_3)$ is replaced by the following.

Correctness of the $(K_3^P)$  We want to show

$$M_v \models (\Gamma) \land M_v \models (f(t_1, \ldots, t_n) : \tau) \implies M_v \models \Gamma$$

which is obvious.

A.4.3  Type Checking and Guarantees

In this section we describe the differences between type checking and reconstructing guarantees in System $B$ and System $P$.

We must define new normal forms.

Definition A.2.  The type $\tau$ is in $L$-normal form if it is atomic or belongs to $\{\top, \bot\}$ or the following condition holds:
1.  $\tau$ has the form $\tau_1 \cup \cdots \cup \tau_n$
2.  For each $i, j \in \{1, \ldots, n\}$, such that for $i \neq j$ we have $\tau_i \not< \tau_j$
3.  Each $\tau_i$ has the form $\alpha_1^i \cap \cdots \cap \alpha_k^i$ or the form

$$\alpha_1^i \cap \cdots \cap \alpha_k^i \cap \bigcap_{j \in L_i} F_j(\sigma_{j,m_j}^i),$$

where $\{\alpha_1, \ldots, \alpha_n\}$ are different variables, $L_i$ are sets of indexes and each $\sigma_j^i$ is in $L$-normal form.

Definition A.3.  The type $\tau$ is in $R$-normal form if it is atomic or belongs to $\{\top, \bot\}$ or the following conditions hold:
1.  $\tau$ has the form $\tau_1 \cap \cdots \cap \tau_n$
2.  For each $i, j \in \{1, \ldots, n\}$, such that for $i \neq j$ we have $\tau_i \not< \tau_j$
3.  Each $\tau_i$ has the form $\alpha_1^i \cup \cdots \cup \alpha_k^i$ or the form

$$\alpha_1^i \cup \cdots \cup \alpha_k^i \cup \bigcup_{j=1}^{m_j} F_j^i(\sigma_{j,m_j}^i),$$

where $\{\alpha_1, \ldots, \alpha_n\}$ are different variables, and $\sigma_j^i$ is in $R$-normal form.
The reduction $N_R$ does not change. However, in $N_L$ some changes are necessary. We delete the rule (4.5). Moreover, the new rule (4.10) is

$$\begin{align*}
N'_L(l \bigcap_{k=1}^{m_k} F_k(\tau^i_1, \ldots, \tau^i_n)) = \\
\bigcap_{i=1}^{m_k} F_k(N_L(\bigcap_{i=1}^{m} \tau^i_1), \ldots, N_L(\bigcap_{i=1}^{m} \tau^i_n))
\end{align*}$$

Example A.1. In the weak system the type $F(\bot) \cap G(\top)$ is in L-normal form (also in R-normal).

The equivalent of the lemma 4.16 is

**Lemma A.4.** The following condition holds:

$$\bigcap_{i=1}^{k} \alpha_i \cap \bigcap_{k \in L} F_k(\tau^i_k) \leq \bigcup_{i=1}^{m} \beta_i \cup \bigcup_{k \in R \setminus l=1}^{s_k} F(\tau^i_k)$$

where $L, R$ are sets of indexes

holds if and only if

$$\bigcap_{i=1}^{k} \alpha_i \leq \bigcup_{i=1}^{m} \beta_i \text{ or } (\exists i)(\exists j) F(\tau^i_k) \leq F(\tau^j_i)$$

**Proof.** A proof is similar to the proof of lemma 4.16.

Because of the new Lemma A.4 the operator $SF$ also changes.

**Definition A.5.** Let $\tau_1 \leq \tau_2$ be in LR-reduced form. Then the value of the function $SF(\tau_1 \leq \tau_2)$ is a Boolean formula built from basic type formulas. It is given by the following inductive definition.

(A.71) $SF(\tau_1 \cup \cdots \cup \tau_n \leq \sigma_1 \cap \cdots \cap \sigma_m) = \bigwedge_{i,j} SF(\tau_i \leq \sigma_j)$

(A.72) $SF(\bigcap_{i=1}^{k} \alpha_i \cap \bigcap_{k \in L} F_k(\tau^i_k)) \leq \bigcup_{i=1}^{m} \beta_i \cup \bigcup_{k \in R \setminus l=1}^{s_k} F(\tau^i_k)) =

= \bigcap_{i=1}^{k} \alpha_i \leq \bigcup_{i=1}^{m} \beta_i \cup (\exists i)(\exists j) F(\tau^i_k) \leq F(\tau^j_i)$
function Less(τ₁, τ₂)
    τ₁, τ₂ — types which are compared
1. let τ₁ be \( \bigcup_{i=1}^{m} P_i \)
2. let τ₂ be \( \bigcap_{i=1}^{n} S_i \)
3. for \( i \in \{1, \ldots, m\}, j \in \{1, \ldots, n\} \) do
4.   let \( P_i \) be \( \bigcap_{k=1}^{p} \alpha_k \cap \bigcap_{k \in L} F_k(\tau'_k) \)
5.   let \( S_j \) be \( \bigcup_{k=1}^{q} \beta_k \cup \bigcup_{k \in R} \bigcup_{i=1}^{n} F_k(\tau'_k) \)
6.   if \( \{\alpha_1, \ldots, \alpha_p\} \cap \{\beta_1, \ldots, \beta_q\} = \emptyset \) then
7.     for each \( i \in L \cap R \) do
8.       for each \( j \) in \( \{1, \ldots, s_i\} \)
9.         if Less(\( \tau'_i, \tau'_j \)) then return 'Yes'
10.    return 'No'
11. return 'Yes'

Algorithm A.1: Algorithm which checks whether \( \tau_1 \leq \tau_2 \) holds

This changes slightly change the algorithm which checks the formulas \( \tau \leq \sigma \). Algorithm A.1 is a new version of algorithm 4.2.

Facts about correctness of this algorithm can be proved in the same way. In a similar way is proved the lemma 4.48.

There is a difference in managing constant types: the rule 4.3 can cause an exponential increase while performing.

However, we can apply to the checked formula the operator \( A'_s \) (defined below, similarly to \( A_s \)) and obtain that for discriminative types of the simple shape type checking works in \( \text{co-NP} \).

Here is the definition of \( A'_s \).

**Definition A.6.** The operator \( A'_s \) takes a shape \( \sigma \) as a first argument and transforms a type \( \tau \) which is its second argument according to \( \sigma \).

\[
\begin{align*}
A'_s(\sigma, \tau_1 \cup \tau_2) &= A'_s(\sigma, \tau_1) \cup A'_s(\sigma, \tau_2) \\
A'_s(\sigma, \tau_1 \cap \tau_2) &= A'_s(\sigma, \tau_1) \cap A'_s(\sigma, \tau_2) \\
A'_s(\top, \alpha) &= \alpha \\
A'_s(\bot, \alpha) &= \bot \\
A'_s(F(\sigma_1, \ldots, \sigma_n), \alpha) &= \top \\
A'_s(F(\sigma_1, \ldots, \sigma_n), \bot) &= \top \\
A'_s(F(\sigma_1, \ldots, \sigma_n), F(\tau_1, \ldots, \tau_n)) &= F(A_s(\sigma_1, \tau_1), \ldots, A_s(\sigma_n, \tau_n)) \\
A'_s(F(\sigma_1, \ldots, \sigma_n), G(\tau_1, \ldots, \tau_n)) &= \top \\
A'_s(F(\sigma_1, \ldots, \sigma_n), \top) &= \top \\
A'_s(\top, G(\tau_1, \ldots, \tau_n)) &= \top \\
\end{align*}
\]

where \( \alpha \) is an atomic type, \( G, F \) are any distinct type constructors.

The operator \( A'_s \) is used instead \( A_s \) in function which computes the best guarantee.

Other facts about type checking and guarantees stay unchanged.

183
Appendix A. Proofs

A.5 Proofs for Chapter 7

Lemma 7.20. Let τ be a type, let σ be an assumption type, and let θ be a substitution. Suppose that there are no \( T^* \) in the range of BS(τ, σ). Then we have

\[ \vdash BS(\theta(\tau), \sigma) = \theta \circ BS(\tau, \sigma) \]

Proof. The proof proceeds by induction on the structure of computation of BS(τ, σ). The proof is routine and we show only proofs of three cases.

- \( \sigma = \alpha \) (i.e. \( \sigma \) is a type variable). Here we have:

\[
BS(\theta(\tau), \alpha) = \emptyset \cup \{ \alpha := \theta(\tau) \} \\
= \emptyset \cup \emptyset \circ \{ \alpha := \tau \} \\
= \emptyset \circ (\emptyset \cup \{ \alpha := \tau \}) \\
= \emptyset \circ BS(\tau, \sigma)
\]

- \( \tau = \tau_1 \cup \tau_2 \). By inductive hypothesis we have

\[
BS(\theta(\tau_1), \sigma) = \theta \circ BS(\tau_1, \sigma) \\
BS(\theta(\tau_2), \sigma) = \theta \circ BS(\tau_2, \sigma)
\]

We have that

\[
BS(\theta(\tau_1 \cup \tau_2), \sigma) = BS(\theta(\tau_1) \cup \theta(\tau_2), \sigma) \\
= BS(\theta(\tau_1), \sigma) \cup BS(\theta(\tau_2), \sigma) \\
= (\theta \circ BS(\tau_1, \sigma)) \cup (\theta \circ BS(\tau_2, \sigma)) \quad \text{by ind. hypothesis} \\
= \theta \circ (BS(\tau_1, \sigma) \cup BS(\tau_2, \sigma)) \\
= \theta \circ BS(\tau_1 \cup \tau_2, \sigma)
\]

\[ \square \]

Lemma 7.21. Let \( t \) be a term, let \( \Gamma \) be an environment. Let \( \theta \) be a substitution. Moreover, suppose that for any subterm \( t' \) of \( t \) we have that

(A.73) \[ BT(\Gamma, t') = \top \quad \text{implies} \quad BT(\theta(\Gamma), t') = \top \]

Then we have that

\[
\theta(BT(\Gamma, t)) = BT(\theta(\Gamma), t)
\]

Proof. The proof proceeds by induction on the structure of a computation of BS(\( \Gamma, t \)).

- false \( \in \Gamma \). Then \( \theta(BT(\Gamma, t)) = \theta(\bot) = \bot = BT(\theta(\Gamma), t) \), and we have \( \Gamma \vdash t : \bot \), from rule (\( T_2 \)).
- \( t \) is a variable \( X \). We have \( \theta(BT(\Gamma, X)) = \theta(\Gamma(X)) = (\theta(\Gamma))(X) = BT(\theta(\Gamma), X) \).
- \( t = f(t_1, \ldots, t_n) \). Suppose that \( f \) has a signature \( \sigma : \sigma_1 \times \cdots \times \sigma_n \rightarrow \sigma \). Let \( \tau_i = BT(\Gamma, t_i) \), for each \( i \in \{1, \ldots, n\} \). By the inductive hypothesis we have

(A.74) \[ BT(\theta(\Gamma), t_i) = \theta(\tau_i) \]

Let \( \theta' = \bigcup_{i=1}^n BS(\tau_i, \sigma_i) \). By Lemma 7.20 we have

\[
\bigcup_{i=1}^n BS(\theta(\tau_i), \sigma_i) = \theta \circ \theta'
\]

184
We have that
\[ \theta(BT(\Gamma, t)) = \theta(\theta'(\sigma)) \]
\[ = \left( \bigcap_{i=1}^{n} BS(\theta(\tau_i), \sigma_i) \right)(\sigma) \]
\[ = \left( \bigcap_{i=1}^{n} BS(BT(\theta(\Gamma), t_i), \sigma_i) \right)(\sigma) \text{ by inductive hypothesis} \]
\[ = BT(\theta(\Gamma), t) \]

and the thesis follows.

- None of the cases above. Then $BT(\Gamma, t) = \top$. By (A.73) we have $BT(\theta(\Gamma), t) = \top = \theta(\top)$.

\[\Box\]

**Lemma 7.23.** Let $R = (\langle Q, T, P \rangle)$ be a RT-task. Suppose that for $R$ Algorithm 7.1 returns $T'$. Then $T'$ is a main type of $P$.

**Proof.** Sketch. Assume that $P$ consists of predicates $p_1, \ldots, p_n$. Suppose that mathcal{T}' is equal to
\[ \{(p_1 : \tau_1 \rightarrow \sigma_1), \ldots, (p_m : \tau_m \rightarrow \sigma_m)\} \]

Then for each predicate $i \in \{1, \ldots, m\}$ we have $\tau_i \rightarrow \sigma_i$ is a main type of $p_i$. Let $S$ be the set of equations obtained in algorithm. Let $\Theta$ be the most general unifier of $S$. If the predicate $p_i$ has a head $p(A_1^{\sigma_i}, \ldots, A_n^{\sigma_i})$ then we have $\text{shape}(\tau_i^j) = \text{shape}(\Theta(A_i))$, for $\tau_i = (\tau_i^1, \ldots, \tau_i^n)$. We will consider the run of the guarantee reconstruction algorithm. We will use indexes $j$ to number the runs of the function ResultType2. Running this function is a reason for changes of guarantees. We will denote the guarantee of the predicate $p_i$ after $j$-th run of ResultType2 by $\sigma_i^j$. In each execution of ResultType2 we have a sequence of environments. We name them by $\Gamma_i^j$. In this proof we call ugly such environments and types whose $L$-normal forms contains subexpressions of the form $\alpha \cap F(\tau_1, \ldots, \tau_n)$ or $F(\tau_1, \ldots, \tau_n) \cap G(\sigma_1, \ldots, \sigma_m)$.

We show by induction on the numbers $j, k$ that during computation

(i) For each program variable $X$ the $\Gamma_i^j(\text{X}) \cap \Theta(\text{X})$ is not ugly,
(ii) for all $i$ we have that $\vdash \text{shape}(\sigma_i^j) = \text{shape}(\tau_i)$,
(iii) for all variables $X \in \text{var}(P)$ we have $\vdash \text{shape}(\Theta(\text{X})) \leq \text{shape}(\Gamma_i^j(\text{X}))$,
(iv) if in a non-ugly environment $G$ we infer from atom $p_i(X_1, \ldots, X_n)$ taking a type $\tau_i \rightarrow \sigma_i^j$ then obtained type (i.e. $\theta(\sigma_i^j)$ for some $\theta$) and obtained environment are not ugly.

**Basic case.**

(i) We show that for each $V$ we have $\Gamma_i^0(V) \cap \Theta(V)$ is not ugly. Suppose that we analyze the clause $C$ with the head $p(A_1, \ldots, A_n)$. Suppose that $\tau_i = (\tau_i^1, \ldots, \tau_i^n)$. We have
\[ \varnothing, (X_1 : \tau_i^1, \ldots, X_n : \tau_i^n) \vdash \Gamma_i^0 = \{(A_1 : \tau_i^1), \ldots, (A_n : \tau_i^n)\} \]
and, since $\tau_i^1, \ldots, \tau_i^n$ are assumption types and for each $i \in \{1, \ldots, n\}$ shape($\tau_i^1$) = shape($\Theta(A_i)$) we have that the condition (i) is fulfilled for $A_i$. It is also fulfilled for a non-head variable $V$ since then $\Gamma_i^0(V) \cap \text{shape}(\Theta(V)) = \top \cap \text{shape}(\Theta(V))$ and is a simple shape.

(ii) $\text{shape}(\sigma_i^0) = \text{shape}(\tau_i)$ since $\sigma_i$ is the least type of the shape $\tau_i$.
(iii) Suppose that we analyze the clause $C$ with the head $p_i(A_1, \ldots, A_n)$. Suppose that $\tau_i = (\tau_i^1, \ldots, \tau_i^n)$. For $A_i \in \{A_1, \ldots, A_n\}$ we have that

$$\text{shape}(\Theta(A_i)) = \text{shape}(\Gamma^j_0(A_i)) = \text{shape}(\tau_i^1)$$

For $Y \notin \{A_1, \ldots, A_n\}$ we have that $\text{shape}(\Gamma^j_0(Y)) = \text{shape}(\top) = \top$ and

$$\text{shape}(\Theta(Y)) \leq \text{shape}(\Gamma^j_0(Y))$$

**Inductive case** From the other hand suppose that thesis holds for all $j' < j$ and $k' < k$. We show that it holds for $j, k$. Consider a clause

$$p_i(X_1^i, \ldots, X_n^i) : -a_1, \ldots, a_v$$

We consider three possible kinds of atoms in $C$ and the situation after running ResultType2. Let $\Gamma^i_0$ be the environment obtained after analysis of the the atom $a_k$. Consider possible forms of atom $a_k$.

1. $p_{m}(Y_1, \ldots, Y_n)$

   Because ResultType2 for clause $C$ has not returned *error*, we have that for each $s \in \{1, \ldots, n\}$

   $$\text{shape}(\text{BT}(\Gamma^j_{k-1}, Y_s)) \leq \text{shape}(\tau_i) = \text{shape}(\Theta(Y_s))$$

   By inductive hypothesis (ii) for $\Gamma^j_{k-1}$ we have that

   $$\text{shape}(\Theta(Y_s)) \leq \text{shape}(\Gamma^j_{k-1}(Y_s))$$

   So, these shapes are equal. From that we obtain that shapes of types in $\Gamma^j_{k-1}$ and shapes of types in $\Gamma^i_0$ are equal and the points (i), (iii), and (iv) hold.

2. $Y = t$

   First, we show that $\text{shape}(\Theta(X)) \leq \text{shape}(\text{BT}(\Gamma^i_0, X))$. Let

   $$\Gamma_s = \{X : \text{shape}(\Theta(X)) | X \in \text{var}(P)\}$$

   Let

   $$\Gamma_1 = \{X : \text{shape}(\Gamma^j_{k-1}(X)) | X \in \text{var}(C)\}$$

   $$\Gamma_2 = \{X : \text{shape}(\Gamma^i_0(X)) | X \in \text{var}(C)\}$$

   We have

   $$\Gamma_1, (t : \Gamma_1(Y), Y : \text{BT}(\Gamma_1, t)) \Rightarrow \Gamma_2$$

   $$\Gamma_2, (t : \Gamma_2(Y), Y : \text{BT}(\Gamma_2, t)) \Rightarrow \Gamma_3$$

   $$\Gamma_s, (t : \Gamma_s(Y), Y : \text{BT}(\Gamma_s, t)) \Rightarrow \Gamma_s$$

   By Lemma 2.58 we have that for each variable $V$

   $$\text{shape}(\Gamma^i_0(V)) = \text{shape}(\text{BT}(\Gamma_s, t))$$

   So, (iii) is proven. We have to show that (i) holds for $\Gamma^i_j$. One can show that

   $$(A.75) \quad \text{shape}(\Theta(Y)) = \text{shape}(\text{BT}(\Gamma_s, t))$$

   We have that

   $$\Gamma^j_{k-1}, (t : \Gamma_1(Y), Y : \text{BT}(\Gamma_1, t)) \Rightarrow \Gamma^j_k$$
Let \( \Gamma' \) be such that
\[
\Gamma'_{k-1} = \{ t: \Gamma_1(Y) \cap \text{shape}(\Theta(X)), Y: BT(\Gamma_1, t) \cap \text{shape}(\Theta(X)) \} \Rightarrow \Gamma'
\]
Since both \( \Gamma'_{k-1} \) \((\text{shape}(\Theta(Y))\) and \( BT(\Gamma_1, t) \cap \text{shape}(\Theta(X)) \) are not ugly and since \((A.75)\) we have that \( \Gamma' \) is not ugly. Since for \( V \in \text{var}(Y, t) \) \( \Gamma'(V) = \text{shape}(\Theta(V)) \cap \Gamma'_{k} \) we have that condition \((i)\) holds for \( \Gamma'_{k} \). Points \((ii), (iv)\) do not apply.

3. \( q(Y_1, \ldots, Y_{n_k}) \). The detailed proof for this case is omitted. Similar idea as in case \( Y = t \) can be used.

Because the value returned by ResultType2 is less than \( \tau_i \) and does not contain subexpressions of the form \( \alpha \cap F(p_1, \ldots, p_n) \) or of the form \( G(\zeta_1, \ldots, \zeta_n) \cap F(p_1, \ldots, p_n) \) it has the shape equal to shape of \( \tau_i \). So, after computing the union of it and current guarantee we also obtain the desired shape and the condition \((iii)\) is fulfilled.

Now, we show that assumption \( \tau_i \) is sufficient for a predicate \( p_i \). One can check that the rule \((K_6)\) was not used while computing all values of ResultType2. Moreover, if for a subterm from the program \( P \) the best type during run of ResultType2 was \( \top \) then it was connected with the shape types which cannot be changed by making assumption more accurate. So, by Lemma 7.21 and Lemma 7.22 we obtain that assumptions for predicates \( p_i \) are sufficient. So, by Lemma 7.19 \( \tau_i \rightarrow \sigma_i \) is the main type of \( p_i \).

A.6 Proofs for Chapter 8

A.6.1 Properties of System C

All the facts stated in Chapter 2 remain true in System C. Most proofs are valid without any changes. Some modifications are necessary in proofs of Lemmas 2.23, 2.26, 2.29, 2.34, 2.49, 2.50, 2.51, 2.64 and 2.65. The modification are simple, thus we present only few of them.

**Proof of Lemma 2.49** The case for \((K_{10})\) is very similar to the case of \((K_2)\). So, assume that
\[
\Gamma_1(X : \tau) \Rightarrow \Gamma[\alpha_X := \alpha_X \cap \tau]
\]
is obtain using rule \((K^2)\). Let \( \theta = [\alpha_X := \alpha_X \cap \tau] \) and \( \theta' = id \). Then obviously \( \theta \leq \theta' \). We want to show that \( \theta(\Gamma) = \Gamma \) which is equivalent to \( \theta(\Gamma) \leq \theta'(\Gamma) \). To do that we should prove that, for any variable \( X \), \( \theta(\Gamma(X)) \leq \theta'(\Gamma(X)) \) which holds by Lemma 2.26.

**Proof of Lemma 2.51** The case of \((K_{10})\) is very similar to the case of \((K_2)\). Consider rule \((K^2)\). In \((*)\) and \((***)\) we have
\[
\Gamma_1(X : \tau) \Rightarrow \Gamma_1[\alpha_X := \alpha_X \cap \tau],
\Gamma_2(X : \tau) \Rightarrow \Gamma_2[\alpha_X := \alpha_X \cap \tau],
\]
respectively. Let \( \theta = [\alpha_X := \alpha_X \cap \tau] \). For each variable \( X \), we have \( \Gamma_1(X) \leq \Gamma_2(X) \), thus, by Corollary 2.24, \( \theta(\Gamma_1(X)) \leq \theta(\Gamma_2(X)) \) which completes the proof.

A.6.2 Semantics of System C

We present here a supplement to the proofs of soundness of System C.

187
Appendix A. Proofs

Soundness of (Ax17): We want to show that

\[ [f^{-1}_k(\tau) \cup f^{-1}_k(\tau')]_v = [f^{-1}_k(\tau \cup \tau')]_v \]

Using (3.3) we can transform this equation into equivalent form:

\[ L = [f^{-1}_k(\tau)]_v \cup [f^{-1}_k(\tau')]_v = [f^{-1}_k(\tau \cup \tau')]_v = R \]

Assume that \( x \in L \). Then, by (8.12), either

\[ \exists x_1 \ldots x_n (f(x_1, \ldots, x_n) \in [\tau]_v \land x = x_k) \]

or

\[ \exists x_1 \ldots x_n (f(x_1, \ldots, x_n) \in [\tau']_v \land x = x_k) \]

In both cases it is easy to show, that \( x \in R \). Consider the first case. We have \( f(x_1, \ldots, x_n) \in [\tau]_v \subseteq [\tau]_v \cup [\tau']_v \), which, by (8.12), is equivalent to \( x \in R \). The second case is similar.

Now, assume that \( x \in R \). Then, by (8.12)

\[ \exists x_1 \ldots x_n (f(x_1, \ldots, x_n) \in [\tau \cup \tau']_v \land x = x_k) \]

This implies that either \( f(x_1, \ldots, x_n) \in [\tau]_v \) or \( f(x_1, \ldots, x_n) \in [\tau']_v \), which means that either \( x \in [f^{-1}_k(\tau)]_v \) or \( x \in [f^{-1}_k(\tau')]_v \). Hence \( x \in L \).

Soundness of (Ax18): Assume that \( f : \tau_1 \cdots \tau_m \rightarrow \tau \). Let \( \tau = F(y_1, \ldots, y_m) \).

We want to show that

\[ L = [f^{-1}_k(\psi)]_v \leq \theta(\tau_k) = R \]

Suppose that \( t \in L \). Then, by (8.12), there exists terms \( t_1, \ldots, t_n \) such that \( f(t_1, \ldots, t_n) \in [\theta(\tau)]_v \) and \( t = t_k \). Let us notice that

\[ \theta(\tau) = F(\theta(y_1), \ldots, \theta(y_n)). \]

So, we have \( f(t_1, \ldots, t_n) \in [F(\theta(y_1), \ldots, \theta(y_n))]_v \). This, by (3.2), implies that

\[ f : \sigma_1 \cdots \sigma_m \rightarrow F(y_1, \ldots, y_n), \]

\[ \exists \theta'(y_i = \perp \lor \theta'(y_i) = \theta(y_i)) \land t_i \in [\theta'(\sigma_i)]_v. \]

But, all signatures of \( f \) have the same left-hand side. So \( \sigma_i = \tau_i \), and we have

\[ f : \tau_1 \cdots \tau_m \rightarrow F(y_1, \ldots, y_n), \]

(A.76)

\[ \exists \theta'(y_i = \perp \lor \theta'(y_i) = \theta(y_i)) \land t_i \in [\theta'(\tau_i)]_v. \]

By definition of signature \( \text{var}(y_1, \ldots, y_n) = \text{var}(\tau_1, \ldots, \tau_n) \). So, since \( \theta'(y_i) = \theta(y_i) \), we have \( \theta'(\tau_i) = \theta(\tau_i) \). Moreover, \( t_i \in [\theta(\tau_i)]_v \) is given by (A.76), thus \( t_i \in [\theta(\tau_i)]_v \), which closes the proof.

Soundness of (Ax19): Assume that \( f : \tau_1 \cdots \tau_m \rightarrow \tau \), \( \tau \) is polymorphic type, and \( \text{head}(\sigma) \neq \text{head}(\tau) \). We shall prove that \( [f^{-1}_k(\sigma)]_v = [\tau]_v \).

Let \( \tau = F(y_1, \ldots, y_k), \sigma = G(a_1, \ldots, a_1) \). We have \( F \neq G \). For the sake of contradiction, assume that there exists \( t \) such that \( t \in [f^{-1}_k(\sigma)]_v \). This implies, by (8.12), that \( f(t_1, \ldots, t_n) \in [\sigma]_v = [G(a_1, \ldots, a_1)]_v \) for some \( t_1, \ldots, t_n \). Hence, by (3.2), we have \( f : \xi_1, \ldots, \xi_m \rightarrow G(z_1, \ldots, z_l) \), for some \( \xi_1, \ldots, \xi_m \) and some \( z_1, \ldots, z_l \), which is impossible since, by the definition of signature, \( f \) has only one signature, and we have assumed that this signature is \( \tau_1 \cdots \tau_m \rightarrow F(y_1, \ldots, y_k) \).

188
Soundness of (Ax20): Assume that $f : \tau_1 \cdots \tau_n \rightarrow \tau$, $\tau$ is an atomic type, and $[\tau \cap [\sigma]]_v = [\bot]_v$. We shall prove that $[f^{-1}_k(\sigma)]_v = [\bot]_v$. 

First, let us notice that

(A.77) $[\tau]_v \cap [\sigma]_v = \emptyset$

Now, for the sake of contradiction, suppose that there exists $t$ such that $t \in [f^{-1}_k(\sigma)]_v$. By (8.12) $f(t_1, \ldots, t_n) \in [\sigma]_v$, for some $t_1, \ldots, t_n$. By (3.2), it is possible only when $f : \xi_1, \ldots, \xi_m \rightarrow \sigma$. But, the left-hand side of each signature of $f$ must be the same, and $\sigma$ must be an atomic type. Thus we have $f : \tau_1 \cdots \tau_n \rightarrow \sigma$. (3.2), and $f(t_1, \ldots, t_n) \in [\sigma]_v$ imply that $t_i \in [\tau_i]_v$. We have assumed that $f : \tau_1 \cdots \tau_n \rightarrow \tau$, so we can use (3.2) to obtain that $f(t_1, \ldots, t_n) \in [\tau]_v$, which contradicts (A.77).

Soundness of (T_2): By (8.11), we have $[\alpha_X]_v = \{v(X)\}$. Therefore, $v(X) \in [\alpha_X]_v$, holds, which is equivalent to $M_v \models X : \alpha_X$.

Soundness of (K_10): We want to show that

(A.78) if $M_v \models \Gamma$ and $M_v \models f(t_1, \ldots, t_n) : \tau$ then $M_v \models \Gamma_1 \cdots \Gamma_n$.

We have the inductive hypothesis

(A.79) if $M_v \models \Gamma$ and $M_v \models t_i : f_i^{-1}(\tau)$ then $M_v \models \Gamma_i$

Assume that

$M_v \models \Gamma$ and $M_v \models f(t_1, \ldots, t_n) : \tau$.

This implies that $v(f(t_1, \ldots, t_n)) \in [\tau]_v$, and hence, by (8.12), we have $v(t_i) \in f_i^{-1}(\tau)$. Now, we can use (A.79) to obtain $M_v \models \Gamma_i$, for each $i \in \{1, \ldots, n\}$. This implies that

$M_v \models \Gamma_1 \cdots \Gamma_n$

which proves (A.78).

Soundness of (K_3): Assume that

(A.80) $M_v \models \Gamma$ and $M_v \models X : \tau$.

We want to show that $M_v \models \Gamma[\alpha_X := \alpha_X \cap \tau]$. From (A.80) it follows that $v(X) \in [\tau]_v$, thus $[\alpha_X]_v \cap [\tau]_v = [\alpha_X]_v$. Let $\theta = \{ \alpha_X/\alpha_X \cap \tau \}$ and $\theta' = \text{id}$. Hence $M_v \models \theta \leq \theta' \leq \theta$. Lemma (2.26) states that $\theta' \leq \theta$ implies $\theta'(\sigma) \leq \theta(\sigma)$, for any type $\sigma$. By Lemma 3.2, we have $M_v \models \theta'(\sigma) \leq \theta(\sigma)$, for any type $\sigma$. Thus, for each variable $Y$, we have

(A.81) $M_v \models \theta'(\Gamma(Y)) \leq \theta(\Gamma(Y))$.

Assume that $(Y : \sigma) \in \theta(\Gamma)$. Then $\sigma = \theta(\Gamma(Y))$. By (A.80), $v(Y) \in \Gamma(Y) = \theta'(\Gamma(Y))$. By (A.81), $v(Y) \in \theta(\Gamma(Y))$ which means that $M_v \models Y : \theta(\Gamma(Y))$. This holds for each $Y$, thus $M_v \models \theta(\Gamma)$ which is equal to $M_v \models \Gamma[\alpha_X := \alpha_X \cap \tau]$.

A.6.3 The Pruning Theorem for System C

The Pruning Theorem remains true in System C without rules (Ax19) and (Ax20), however the proofs have to be completed. We do not present here easy completions of proofs of Lemma 6.10 and Lemma 6.14.
Appendix A. Proofs

Proof of Lemma 6.6 The proofs in the cases for (Ax19) and (Ax20) are simple, and thus we omit them.

(Ax17) The axiom has the following form:
\[ \vdash f_k^{-1}(\tau) \cup f_k^{-1}(\tau') = f_k^{-1}(\tau \cup \tau'). \]
We want to show that
\[ (A.82) \quad \vdash \nabla_A(f_k^{-1}(\tau) \cup f_k^{-1}(\tau')) = \nabla_A(f_k^{-1}(\tau \cup \tau')). \]
Assume that either \( \nabla_A(f_k^{-1}(\tau)) = \top \) or \( \nabla_A(f_k^{-1}(\tau')) = \top \). Then, by the definition of pruning,
\[ \nabla_A(f_k^{-1}(\tau \cup \tau')) = \top. \]
On the other hand
\[ \nabla_A(f_k^{-1}(\tau) \cup f_k^{-1}(\tau')) = \nabla_A(f_k^{-1}(\tau)) \cup \nabla_A(f_k^{-1}(\tau')) = \top, \]
which proves (A.82).
Now, assume that \( \nabla_A(f_k^{-1}(\tau)) \neq \top \) and \( \nabla_A(f_k^{-1}(\tau')) \neq \top \). It implies that \( \nabla_A(\tau) \neq \top \) and \( \nabla_A(\tau') \neq \top \) and thus
\[ \nabla_A(f_k^{-1}(\tau \cup \tau')) = f_k^{-1}(\nabla_A(\tau \cup \tau')) \]
\[ = f_k^{-1}(\nabla_A(\tau) \cup \nabla_A(\tau')) \]
\[ = f_k^{-1}(\nabla_A(\tau)) \cup f_k^{-1}(\nabla_A(\tau')) \]
\[ = \nabla_A(f_k^{-1}(\tau)) \cup \nabla_A(f_k^{-1}(\tau')) \]
\[ = \nabla_A(f_k^{-1}(\tau) \cup f_k^{-1}(\tau')). \]
So (A.82) holds.

(Ax18) The axiom has the following form:
\[ \vdash f_k^{-1}(\theta(\tau)) = f_k^{-1}(\theta(\tau)) \cap \theta(\tau_k) \]
where \( f : \tau_1 \cdots \tau_n \rightarrow \tau \). We want to show that
\[ L = \nabla_A(f_k^{-1}(\theta(\tau))) = \nabla_A(f_k^{-1}(\theta(\tau)) \cap \theta(\tau_k)) = R. \]
Suppose that \( \nabla_A(\theta(\tau)) = \top \). By Definition 6.1, it implies \( \nabla_A(\theta(\tau_k)) = \top \).
So, by the definition of pruning, we have
\[ L = \nabla_A(f_k^{-1}(\theta(\tau))) = \top, \]
and \( \nabla_A(\theta(\tau_k)) = \top \). Hence
\[ R = \nabla_A(f_k^{-1}(\theta(\tau)) \cap \theta(\tau_k)) = L \cap \nabla_A(\theta(\tau_k)) = \top \cap \top = \top = L. \]
Now, suppose that \( \nabla_A(\theta(\tau)) \neq \top \). Let \( \hat{\theta} = \nabla_A(\theta) \). Thus, by Definition 6.1, we have
\[ L = f_k^{-1}(\nabla_A(\theta(\tau))) = f_k^{-1}(\hat{\theta}(\tau)). \]
Moreover,
\[ R = L \cap \nabla_A(\theta(\tau_k)) = f_k^{-1}(\hat{\theta}(\tau)) \cap \hat{\theta}(\tau_k) = f_k^{-1}(\hat{\theta}(\tau)) = L. \]
The last equality follows from (Ax18). \( \square \)
Proof of Lemma 6.8 Assume that \( \tau = f_k^{-1}(\sigma) \). If \( L = \nabla_A(f_k^{-1}(\sigma)) = \top \) then \( R = \nabla_A(\top) = \top \). So, assume that \( L = \nabla_A(f_k^{-1}(\sigma)) \neq \top \). Then, by the definition of pruning, \( \nabla_A(\sigma) \neq \top \) and thus, by the inductive hypothesis, \( \nabla_A(\nabla_A(\sigma)) \neq \top \). Hence

\[
R = \nabla_A(\nabla_A(f_k^{-1}(\sigma))) = \nabla_A(f_k^{-1}(\nabla_A(\sigma)))
= f_k^{-1}(\nabla_A(\nabla_A(\sigma)))
= f_k^{-1}(\nabla_A(\sigma)) \quad \text{(by the inductive hypothesis)}
= \nabla_A(f_k^{-1}(\tau)) = L.
\]

\[\square\]

Proof of Lemma 6.18

\((K_{10})\) In this case \( \Gamma, (f(t_1, \ldots, t_n) : \alpha) \Rightarrow \Gamma_1 \sqcup \cdots \sqcup \Gamma_n \) is proved using \( \Gamma, (t_i : f_i^{-1}(\alpha)) \Rightarrow \Gamma_i \) for \( i \in \{1, \ldots, n\} \).

Since \( \nabla_A(\alpha) = \top \), we have \( \nabla_A(f_i^{-1}(\alpha)) = \top \). By the inductive hypothesis,

\[
\Gamma_1 \nabla_A \cdots \nabla_A \Gamma_n \nabla_A \cdots \nabla_A \Gamma.
\]

Thus, by Corollary 6.16, \( \Gamma_n \nabla_A \cdots \nabla_A \Gamma \).

\((K_3)\) In this case we have \( \Gamma, (X : \tau) \Rightarrow \Gamma[\alpha_X := \alpha_X \cap \tau] \) is given by \((K_3)\). Let \( \theta = [\alpha_X/\alpha_X \cap \tau] \). We have \( \Gamma[\alpha_X := \alpha_X \cap \tau] = \theta(\Gamma) \). Suppose that

\[(\Lambda.83) \quad \vdash \sigma \leq \nabla_A(\theta(\sigma)), \text{ for any } (Y : \sigma) \in \Gamma.\]

This gives \( \theta(\sigma) \nabla_A \sigma \). Thus, by Lemma 6.14, \( \theta(\Gamma) \nabla_A \Gamma \). So, it remains to prove \((\Lambda.83)\). We proceed by induction on the structure of a type \( \sigma \). Let us notice that statement \( \vdash \sigma \leq \nabla_A(\theta(\sigma)) \) is a shorthand for \( \vdash \sigma \cap \nabla_A(\theta(\sigma)) = \sigma \).

- \( \sigma = \alpha_X \). Then

\[
\vdash \sigma \cap \nabla_A(\theta(\sigma)) = \alpha_X \cap \nabla_A(\theta(\alpha_X)) = \alpha_X \cap \nabla_A(\alpha_X \cap \tau)
= \alpha_X \cap \nabla_A(\alpha_X) \cap \nabla_A(\tau) = \alpha_X \cap \nabla_A(\alpha_X) \cap \top
= \alpha_X = \sigma.
\]

- \( \sigma = \beta \), where \( \beta \) is a variable. Let us notice that \( \beta \cap \nabla_A(\beta) = \beta \) independently on whether \( \beta \) belongs to \( A \) or not. So

\[
\sigma \cap \nabla_A(\theta(\sigma)) = \beta \cap \nabla_A(\theta(\beta)) = \beta \cap \nabla_A(\beta) = \beta = \sigma.
\]

- \( \sigma \) is an atomic type. Then \( \sigma \cap \nabla_A(\theta(\sigma)) = \sigma \cap \sigma = \sigma \).

- \( \sigma = F(\sigma_1, \ldots, \sigma_n) \). If \( \nabla_A(\theta(\sigma)) = \top \) then \( \sigma \cap \nabla_A(\theta(\sigma)) = \sigma \cap \top = \sigma \). Assume that \( \nabla_A(\theta(\sigma)) \neq \top \). Then

\[
\nabla_A(\theta(\sigma)) = \nabla_A(F(\theta(\sigma_1), \ldots, \theta(\sigma_n)))
= F(\nabla_A(\theta(\sigma_1)), \ldots, \nabla_A(\theta(\sigma_n))).
\]

Moreover

\[
\sigma \cap \nabla_A(\theta(\sigma)) = F(\sigma_1, \ldots, \sigma_n) \cap F(\nabla_A(\theta(\sigma_1)), \ldots, \nabla_A(\theta(\sigma_n)))
= F(\sigma_1 \cap \nabla_A(\theta(\sigma_1)), \ldots, \sigma_n \cap \nabla_A(\theta(\sigma_n)))
= F(\sigma_1, \ldots, \sigma_n) \quad \text{(by the inductive hypothesis)}
= \sigma.
\]

191
Appendix A. Proofs

• \( \sigma = \sigma_1 \cap \sigma_2 \). Then

\[
\sigma \cap \nabla_A(\theta(\sigma)) = \sigma_1 \cap \sigma_2 \cap \nabla_A(\theta(\sigma_1 \cap \sigma_2))
\]

\[
= \sigma_1 \cap \sigma_2 \cap \nabla_A(\theta(\sigma_1)) \cap \nabla_A(\theta(\sigma_2))
\]

\[
= (\sigma_1 \cap \nabla_A(\theta(\sigma_1))) \cap (\sigma_2 \cap \nabla_A(\theta(\sigma_2)))
\]

\[
= \sigma_1 \cap \sigma_2 \quad \text{(by the inductive hypothesis)}
\]

\[
= \sigma.
\]

In the following two cases we will the fact that \( \sigma \leq \tau \) is, by Lemma 2.20, equivalent to \( \tau \cup \sigma = \sigma \). Thus we shall show that \( \sigma \cup \nabla_A(\theta(\sigma)) = \nabla_A(\theta(\sigma)) \).

• \( \sigma = \sigma_1 \cup \sigma_2 \). Then

\[
\sigma \cup \nabla_A(\theta(\sigma)) = \sigma_1 \cup \sigma_2 \cup \nabla_A(\theta(\sigma_1 \cup \sigma_2))
\]

\[
= \sigma_1 \cup \sigma_2 \cup \nabla_A(\theta(\sigma_1)) \cup \nabla_A(\theta(\sigma_2))
\]

\[
= (\sigma_1 \cup \nabla_A(\theta(\sigma_1))) \cup (\sigma_2 \cup \nabla_A(\theta(\sigma_2)))
\]

\[
= \nabla_A(\theta(\sigma_1)) \cup \nabla_A(\theta(\sigma_2))
\]

\[
= \nabla_A(\theta(\sigma_1 \cup \sigma_2)) = \nabla_A(\theta(\sigma)).
\]

\((*)\) is given by the inductive hypothesis.

• \( \sigma = f_k^{-1}(\delta) \). If \( \nabla_A(\theta(\sigma)) = \top \) then \( \sigma \cup \nabla_A(\theta(\sigma)) = \sigma \cup \top = \top = \nabla_A(\theta(\sigma)) \).

Now, suppose that \( \nabla_A(\theta(\sigma)) \neq \top \). Then

\[
\nabla_A(\theta(\sigma)) = \nabla_A(\theta(f_k^{-1}(\delta))) = f_k^{-1}(\nabla_A(\theta(\delta))).
\]

Hence,

\[
\sigma \cup \nabla_A(\theta(\sigma)) = f_k^{-1}(\delta) \cup f_k^{-1}(\nabla_A(\theta(\delta)))
\]

\[
= f_k^{-1}(\delta \cup \nabla_A(\theta(\delta))) \quad \text{by \textit{Ax17}}
\]

\[
= f_k^{-1}(\nabla_A(\theta(\delta))) \quad \text{by the ind. hyp.}
\]

\[
= \nabla_A(\theta(f_k^{-1}(\delta))) = \nabla_A(\theta(\sigma)).
\]

Lemma A.7. Suppose that \( A \) is a set of type variables, \( \theta \) is any type substitution, \( \text{dom}(\theta) \cap A = \emptyset \), \( \hat{\theta} = \nabla_A(\theta) \), \( \tau \) and \( \hat{\tau} \) are types, and \( \tau \vdash \hat{\tau} \). Then \( \theta(\tau) \vdash \hat{\theta}(\hat{\tau}) \).

Proof. First, let us prove the following fact:

\[
(A.84) \quad \text{for any type } \tau, \quad \vdash \nabla_A(\theta(\tau)) = \nabla_A(\hat{\theta}(\hat{\tau}))
\]

The proof is by induction on the structure of type \( \tau \). Let \( L \) and \( R \) denote the left and the right-hand side of this equality respectively.

• If \( \tau \) is \( \top \), \( \bot \), \( \alpha_X \) (for any variable \( X \)) or an atomic type, then the proof is obvious.

• Suppose that \( \tau = \alpha \). If \( \alpha \not\in \text{dom}(\theta) \) then both sides of the equality are equal to \( \nabla_A(\alpha) \). If \( \alpha \in \text{dom}(\theta) \) then, by the definition of \( \hat{\theta} \), \( R = \nabla_A(\hat{\theta}(\tau)) = \nabla_A(\nabla_A(\theta(\tau))) \), which is, by Lemma 6.8, equal to \( \nabla_A(\theta) = L \).

• Suppose that \( \tau = F(\tau_1, \ldots, \tau_n) \). By induction we have \( \nabla_A(\theta(\tau_i)) = \nabla_A(\hat{\theta}(\tau_i)) \) for \( i \in \{1, \ldots, n\} \). Thus, the definition of pruning gives

\[
\vdash L = \nabla_A(F(\theta(\tau_1), \ldots, \theta(\tau_n))) = \nabla_A(F(\hat{\theta}(\tau_1), \ldots, \hat{\theta}(\tau_n))) = R.
\]

192
If \( \tau = f_k^{-1}(\tau) \) the proof proceeds in the same way as above (the definition of pruning for \( f_k^{-1}(\tau) \) is similar to that for \( F(\tau) \)).

- If \( \tau = \tau_1 \cup \tau_2 \) then by the definitions of substitution and pruning

  \[
  \vdash L = \nabla_A(\hat{\theta}(\tau_1 \cup \tau_2)) = \nabla_A(\theta(\tau_1)) \cup \nabla_A(\theta(\tau_2)) = \nabla_A(\hat{\theta}(\tau_1)) \cup \nabla_A(\hat{\theta}(\tau_2)) \text{ by the ind. hyp.} = \nabla_A(\hat{\theta}(\tau_1 \cup \tau_2)) = R.
  \]

- If \( \tau = \tau_1 \cap \tau_2 \) then the proof proceeds as above.

This completes the proof of (A.84). Now let us prove that

\[
(A.85) \quad \vdash \nabla_A(\hat{\theta}(\tau)) \geq \hat{\theta}(\nabla_A(\tau))
\]

As before, we shall prove it by induction on the structure of a type \( \tau \), and we shall use \( L \) and \( R \) to denote the left and the right-hand side of this inequality respectively.

- If \( \tau \) is \( \top, \bot, \alpha_X \) (for any variable \( X \)) or an atomic type, then the proof is obvious.

- Suppose that \( \tau = \alpha \). We consider two cases: \( \nabla_A(\tau) = \top \), and \( \nabla_A(\tau) \neq \top \).

  Suppose that \( \alpha \in A \). Then, as we have supposed, \( \alpha \not\in \text{dom}(\theta) \) and thus \( \theta(\alpha) = \alpha \). So, by the definition of \( \hat{\theta} \), we have

  \[
  \vdash L = \nabla_A(\hat{\theta}(\alpha)) = \nabla_A(\nabla_A(\theta(\alpha))) = \nabla_A(\nabla_A(\alpha)) = \nabla_A(\top) = \top,
  \]

  and \( \vdash R = \hat{\theta}(\nabla_A(\alpha)) = \hat{\theta}(\top) = \top \). So, \( L = R \).

  Now suppose that \( \nabla_A(\tau) \neq \top \) and thus \( \nabla_A(\alpha) = \alpha \). Then we have

  \[
  \vdash L = \nabla_A(\hat{\theta}(\alpha)) = \nabla_A(\nabla_A(\theta(\alpha))) = \nabla_A(\theta(\alpha)) = \hat{\theta}(\alpha) = \hat{\theta}(\nabla_A(\alpha)) = R.
  \]

- Suppose that \( \tau = F(\tau_1, \ldots, \tau_n) \). If \( \nabla_A(\tau) = \top \), then, by the definition of pruning, for each \( i \in \{1, \ldots, n\} \), \( \nabla_A(\tau_i) = \top \), and hence \( \hat{\theta}(\nabla_A(\tau_i)) = \top \). Thus, by induction, \( \nabla_A(\hat{\theta}(\tau_i)) \geq \top \), so by (Ax11), \( \nabla_A(\hat{\theta}(\tau_i)) = \top \), for each \( i \in \{1, \ldots, n\} \).

  By the definition of pruning, we have

  \[
  L = \nabla_A(F(\hat{\theta}(\tau_1), \ldots, \hat{\theta}(\tau_n))) = \top,
  \]

  and thus \( L = R \).

  Now, suppose that \( \nabla_A(\tau) \neq \top \). So

  \[
  \vdash R = \hat{\theta}(F(\nabla_A(\tau_1), \ldots, \nabla_A(\tau_n))) = F(\hat{\theta}(\nabla_A(\tau_1)), \ldots, \hat{\theta}(\nabla_A(\tau_n))) \leq F(\nabla_A(\hat{\theta}(\tau_1)), \ldots, \nabla_A(\hat{\theta}(\tau_1))) \text{ (by the inductive hypothesis)}.
  \]

  If \( \nabla_A(\hat{\theta}(\tau_i)) = \top \), for each \( i \in \{1, \ldots, n\} \) then, by the definition of pruning, \( L = \top \), and hence \( L \geq R \). If, on the contrary, \( \nabla_A(\hat{\theta}(\tau_k)) \neq \top \), for some \( k \), then

  \[
  \vdash L = F(\nabla_A(\hat{\theta}(\tau_1)), \ldots, \nabla_A(\hat{\theta}(\tau_1))) \geq R.
  \]
Appendix A. Proofs

- When \( \tau = f_k^{-1}(\tau) \) the proof proceeds as above.
- If \( \tau = \tau_1 \cup \tau_2 \) then by the definitions of substitution and pruning
  \[
  \vdash L = \nabla_A(\hat{\theta}(\tau_1 \cup \tau_2)) = \nabla_A(\hat{\theta}(\tau_1)) \cup \nabla_A(\hat{\theta}(\tau_2))
  \geq \hat{\theta}(\nabla_A(\tau_1)) \cup \hat{\theta}(\nabla_A(\tau_2)) \quad \text{(by the induct. hyp.)}
  = \hat{\theta}(\nabla_A(\tau_1 \cup \tau_2)) = R
  \]
- If \( \tau = \tau_1 \cup \tau_2 \) then the proof proceeds as above.
  
  Now, we can prove the lemma. To satisfy the definition of \( \theta(\tau) \overset{*}{\prec} \hat{\theta}(\tau) \) we have to prove that (a) \( \vdash \theta(\tau) \leq \hat{\theta}(\tau) \) and (b) \( \vdash \hat{\theta}(\tau) \leq \nabla_A(\theta(\tau)) \).

  We have assumed that \( \tau \overset{*}{\prec} \hat{\tau} \), so, by Definition 6.9, \( \vdash \tau \leq \hat{\tau} \). Thus, by Corollary 2.24 and Lemma 2.26, \( \vdash \theta(\tau) \leq \theta(\hat{\tau}) \leq \hat{\theta}(\tau) \). Thus (a) holds.

  From \( \tau \overset{*}{\prec} \hat{\tau} \), and from Definition 6.9 it follows also that \( \hat{\tau} \leq \nabla_A(\tau) \). Now, we have

  \[
  \vdash \hat{\theta}(\hat{\tau}) \leq \hat{\theta}(\nabla_A(\tau)) \quad \text{(by corollary 2.24)}
  \leq \nabla_A(\hat{\theta}(\tau)) \quad \text{(by (A.85))}
  = \nabla_A(\theta(\tau)) \quad \text{(by (A.84)).}
  \]

So (b) is proved, which conclude the proof. \( \square \)

Proof of the Lemma 6.18  For \( (K_{10})_1 \) the proof is very similar to that for \( (K_2) \).
So, consider rule \( (K_5^s) \). In the proof of \( (s) \) we have \( \Gamma, (X : \tau) \Rightarrow \Gamma[\alpha_X \sqcup \alpha_X \cap \tau] \).
By \( (K_5^s) \) we have \( \hat{\Gamma}, (X : \nabla_A(\tau)) \Rightarrow \hat{\Gamma}[\alpha_X \sqcup \alpha_X \cap \nabla_A(\tau)] \). Let \( \theta = \{\alpha_X / \alpha_X \cap \tau\} \) and \( \hat{\theta} = \{\alpha_X / \alpha_X \cap \nabla_A(\tau)\} \). It remains to show that \( \theta(\Gamma) \overset{*}{\prec} \hat{\theta}(\hat{\Gamma}) \), which, by Lemma 6.14, is equivalent to showing that, for any program variable \( X \), \( \theta(\Gamma(X)) \overset{*}{\prec} \hat{\theta}(\hat{\Gamma}(X)) \), which follows from Lemma A.7, since \( \Gamma(X) \overset{*}{\prec} \hat{\Gamma}(X) \) and \( \hat{\theta} = \nabla_A(\hat{\theta}) \). \( \square \)
Appendix B

Axioms and rules

B.1 Rules in System B

\[ \vdash \tau = \tau \]

\[
\vdash \tau_1 = \tau'_1, \ldots, \vdash \tau_n = \tau'_n \quad \vdash F(\tau_1, \ldots, \tau_n) = F(\tau'_1, \ldots, \tau'_n)
\]

for any type constructor \( F \) of arity \( n \)

\[
\vdash \tau_1 = \tau'_1, \quad \vdash \tau_2 = \tau'_2
\]

\[
\vdash \tau_1 \cap \tau_2 = \tau'_1 \cap \tau'_2
\]

\[
\vdash \tau_1 = \tau'_1, \quad \vdash \tau_2 = \tau'_2
\]

\[
\vdash \tau_1 \cup \tau_2 = \tau'_1 \cup \tau'_2
\]

\[
\vdash \tau_1 = \tau'_1, \quad \vdash \tau_2 = \tau'_2, \quad \vdash \tau_1 = \tau_2
\]

\[
\vdash \tau' = \tau
\]

For atomic types \( \tau \) and \( \sigma \) (like \( \text{int} \) or \( \text{real} \)) we also have

\[ \vdash \tau \cup \sigma = \text{or}(\tau, \sigma) \]

\[ \vdash \tau \cap \sigma = \text{and}(\tau, \sigma) \]
### Appendix B. Axioms and rules

#### Figure B.1: Term typing rules in System B

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>$\Gamma \vdash t : \top$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$\Gamma \cup { \text{false} } \vdash t : \bot$</td>
</tr>
<tr>
<td>$T_3$</td>
<td>$\Gamma \cup { X : \tau } \vdash X : \tau$</td>
</tr>
</tbody>
</table>
| $T_4$ | $\Gamma \vdash t : \tau', \tau' \leq \tau$  
| & $\Gamma \vdash t : \tau$ |
| $T_5$ | $\Gamma \vdash t : \tau_1, \ldots, \Gamma \vdash t : \tau_n$  
| & $\Gamma \vdash t : \tau_1 \cap \cdots \cap \tau_n$ |
| $T_6$ | $\Gamma \vdash t_i : \theta(\tau_i) \ (i \in \{1, \ldots, n\})$  
| & $\Gamma \vdash f(t_1, \ldots, t_n) : \theta(\tau) \ f : \tau_1 \ast \cdots \ast \tau_n \rightarrow \tau$ |

#### Figure B.2: Consequence rules in System B

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| $K_1$ | $\Gamma, (t_i : \tau_i) \Rightarrow \Gamma_i \ (1 \leq i \leq n)$  
| & $\Gamma_i(t_1 : \tau_1, \ldots, t_n : \tau_n) \Rightarrow \Gamma_1 \cap \cdots \cap \Gamma_n$ |
| $K_2$ | $\Gamma, (t_i : \theta(\tau_i)) \Rightarrow \Gamma_i \ (1 \leq i \leq n)$  
| & $\Gamma_i(f(t_1, \ldots, t_n) : \theta(\tau)) \Rightarrow \Gamma_1 \cap \cdots \cap \Gamma_n$  
| & if $f : \tau_1 \ast \cdots \ast \tau_n \rightarrow \tau$ |
| $K_3$ | $\Gamma, (f(t_1, \ldots, t_n) : \tau) \Rightarrow \{\text{false}\}$  
| & if there is no signature $\tau_1 \ast \cdots \ast \tau_n \rightarrow \tau$ assigned to $f$  
| & such that $\text{head}(\tau) = \text{head}(\tau_0)$ |
| $K_4$ | $\Gamma, (t : \bot) \Rightarrow \{\text{false}\}$ |
| $K_5$ | $\Gamma, (t : \top) \Rightarrow \Gamma$ |
| $K_6$ | $\Gamma, (f(t_1, \ldots, t_n) : \alpha) \Rightarrow \Gamma$ |
| $K_7$ | $\Gamma_i(t : \tau_i) \Rightarrow \Gamma_i \ (1 \leq i \leq n)$  
| & $\Gamma_i(t : \tau_1 \cap \cdots \cap \tau_n) \Rightarrow \Gamma_1 \cap \cdots \cap \Gamma_n$ |
| $K_8$ | $\Gamma_i(t : \tau_i) \Rightarrow \Gamma_i \ (1 \leq i \leq n)$  
| & $\Gamma_i(t : \tau_1 \cup \cdots \cup \tau_n) \Rightarrow \Gamma_1 \cup \cdots \cup \Gamma_n$ |
| $K_9$ | $\Gamma, (X : \tau) \Rightarrow \Gamma \cap \{ X : \tau \}$  
| & if $\tau \neq \bot$ and $\tau \neq \top$ |
\begin{figure}[h]
\centering
\fbox{
\begin{align*}
(P_1) \quad & \vdash \text{InferFromAtoms}(T, \Gamma, \langle \rangle, \Gamma) \\
\quad & \quad \Gamma \vdash t : \theta(\tau), \quad \Gamma, (t_1 : \theta(\sigma_1), \ldots, t_n : \theta(\sigma_n)) \Rightarrow \Gamma', \\
\quad & \quad \vdash \text{InferFromAtoms}(T, \Gamma', (a_2, \ldots, a_k), \Gamma'') \\
\quad & \quad \vdash \text{InferFromAtoms}(T, \Gamma, (p(t_1, \ldots, t_n), a_2, \ldots, a_k), \Gamma'') \\
(P_2) \quad & \quad \text{if } (p : (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)) \in T, \\
\quad & \quad \emptyset, (t_1 : \tau_1, \ldots, t_n : \tau_n) \Rightarrow \Gamma, \\
\quad & \quad \vdash \text{InferFromAtoms}(T, \Gamma, B, \Gamma_2), \quad \Gamma_2 \vdash t_1 : \sigma_1, \ldots, \Gamma_2 \vdash t_n : \sigma_n, \\
\quad & \quad \vdash \text{ClauseHasType}(T, \ p(t_1, \ldots, t_n) \leftarrow B, \ (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)) \\
\end{align*}
}
\caption{Clause typing rules in System B}
\end{figure}
B.2 Rules in the System P

\textbf{(Ax1)} \quad \vdash \tau = \tau

\textbf{(Ax2)}
\[
\vdash \tau_1 = \tau_1', \ldots, \vdash \tau_n = \tau_n' \\
\vdash F(\tau_1, \ldots, \tau_n) = F(\tau_1', \ldots, \tau_n')
\]
for any type constructor \( F \) of arity \( n \)

\textbf{(Ax3)}
\[
\vdash \tau_1 = \tau_1', \quad \vdash \tau_2 = \tau_2' \\
\vdash \tau_1 \cap \tau_2 = \tau_1' \cap \tau_2'
\]

\textbf{(Ax4)}
\[
\vdash \tau_1 = \tau_1', \quad \vdash \tau_2 = \tau_2' \\
\vdash \tau_1 \cup \tau_2 = \tau_1' \cup \tau_2'
\]

\textbf{(Ax5)}
\[
\vdash \tau_1 = \tau_1', \quad \vdash \tau_2 = \tau_2', \quad \vdash \tau_1 = \tau_2 \\
\vdash \tau_1' = \tau_2'
\]

\textbf{(Ax6)}
\[
\vdash \tau \cup \tau = \tau \\
\vdash \tau \cap \tau = \tau
\]

\textbf{(Ax7)}
\[
\vdash \tau \cup \sigma = \sigma \cup \tau \quad \vdash \tau \cap \sigma = \sigma \cap \tau
\]

\textbf{(Ax8)}
\[
\vdash \tau \cup (\sigma \cup \xi) = (\tau \cup \sigma) \cup \xi \quad \vdash \tau \cap (\sigma \cap \xi) = (\tau \cap \sigma) \cap \xi
\]

\textbf{(Ax9)}
\[
\vdash \tau \cup (\tau \cap \sigma) = \tau \quad \vdash \tau \cap (\tau \cup \sigma) = \tau
\]

\textbf{(Ax10)}
\[
\vdash \tau \cup (\sigma \cap \xi) = (\tau \cup \sigma) \cap (\tau \cup \xi) \quad \vdash \tau \cap (\sigma \cup \xi) = (\tau \cap \sigma) \cup (\tau \cap \xi)
\]

\textbf{(Ax11)}
\[
\vdash \tau \cup \top = \top \\
\vdash \tau \cap \top = \tau
\]

\textbf{(Ax12)}
\[
\vdash \tau \cup \bot = \tau \\
\vdash \tau \cap \bot = \bot
\]

\textbf{(Ax13)}
\[
\vdash F(\tau_1, \ldots, \tau_n) \cap F(\tau_1', \ldots, \tau_n') = F(\tau_1 \cap \tau_1', \ldots, \tau_n \cap \tau_n')
\]
for any type constructor \( F \) of arity \( n \)

\textbf{(Ax14)}
\text{omitted}

For atomic types \( \tau \) and \( \sigma \) (like \texttt{int} or \texttt{real}) we also have

\textbf{(Ax15)}
\[
\vdash \tau \cup \sigma = \text{or}(\tau, \sigma)
\]

\textbf{(Ax16)}
\[
\vdash \tau \cap \sigma = \text{and}(\tau, \sigma)
\]
### B.2. Rules in the System P

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(T1)</td>
<td>$\Gamma \vdash t : \top$</td>
</tr>
<tr>
<td>(T2)</td>
<td>$\Gamma \cup {\text{false}} \vdash t : \bot$</td>
</tr>
<tr>
<td>(T3)</td>
<td>$\Gamma \cup {X : \tau} \vdash X : \tau$</td>
</tr>
<tr>
<td>(T4)</td>
<td>$\Gamma \vdash t : \tau';\vdash \tau' \leq \tau$ $\Gamma \vdash t : \tau$</td>
</tr>
<tr>
<td>(T5)</td>
<td>$\Gamma \vdash t : \tau_1, \ldots \Gamma \vdash t : \tau_n$ $\Gamma \vdash t : \tau_1 \cap \cdots \cap \tau_n$</td>
</tr>
<tr>
<td>(T6)</td>
<td>$\Gamma \vdash t_i : \theta(\tau_i); (i \in {1, \ldots, n})$ $\Gamma \vdash f(t_1, \ldots, t_n) : \theta(\tau)$</td>
</tr>
</tbody>
</table>

**Figure B.4: Term typing rules in System P**

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(K1)</td>
<td>$\Gamma, (t_i : \tau_i) \Rightarrow \Gamma_i; (1 \leq i \leq n)$ $\Gamma, (t_1 : \tau_1, \ldots, t_n : \tau_n) \Rightarrow \Gamma_1 \sqcap \cdots \sqcap \Gamma_n$</td>
</tr>
<tr>
<td>(K2)</td>
<td>$\Gamma, (t_i : \theta(\tau_i)) \Rightarrow \Gamma_i; (1 \leq i \leq n)$ $\Gamma, (f(t_1, \ldots, t_n) : \theta(\theta(\tau))) \Rightarrow \Gamma_1 \sqcap \cdots \sqcap \Gamma_n$ if $f : \tau_1 \cdots \tau_n \rightarrow \tau$</td>
</tr>
<tr>
<td>(K3)</td>
<td>$\Gamma, (f(t_1, \ldots, t_n) : \tau) \Rightarrow \Gamma$ if there is no signature $\tau_1 \cdots \tau_n \rightarrow \tau_0$ assigned to $f$ such that $\text{head}(\tau) = \text{head}(\tau_0)$</td>
</tr>
<tr>
<td>(K4)</td>
<td>$\Gamma, (t : \bot) \Rightarrow {\text{false}}$</td>
</tr>
<tr>
<td>(K5)</td>
<td>$\Gamma, (t : \top) \Rightarrow \Gamma$</td>
</tr>
<tr>
<td>(K6)</td>
<td>$\Gamma, (f(t_1, \ldots, t_n) : \alpha) \Rightarrow \Gamma$</td>
</tr>
<tr>
<td>(K7)</td>
<td>$\Gamma, (t_1 : \tau_i) \Rightarrow \Gamma_i; (1 \leq i \leq n)$ $\Gamma, (t : \tau_1 \cap \cdots \cap \tau_n) \Rightarrow \Gamma_1 \sqcap \cdots \sqcap \Gamma_n$</td>
</tr>
<tr>
<td>(K8)</td>
<td>$\Gamma, (t_1 : \tau) \Rightarrow \Gamma_i; (1 \leq i \leq n)$ $\Gamma, (t : \tau_1 \cup \cdots \cup \tau_n) \Rightarrow \Gamma_1 \sqcup \cdots \sqcup \Gamma_n$</td>
</tr>
<tr>
<td>(K9)</td>
<td>$\Gamma, (X : \tau) \Rightarrow \Gamma \sqcap {X : \tau}$ if $\tau \neq \bot$ and $\tau \neq \top$</td>
</tr>
</tbody>
</table>

**Figure B.5: Consequence rules in System P**
### Appendix B. Axioms and rules

#### Figure B.6: Clause typing rules in System P

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P₁)</td>
<td>( \vdash \text{InferFromAtoms}(T, \Gamma; \emptyset, \Gamma) )</td>
</tr>
<tr>
<td>(P₂)</td>
<td>( \Gamma \vdash t_i : \theta(\tau_i), \quad \Gamma, (t_1 : \theta(\sigma_1), \ldots, t_n : \theta(\sigma_n)) \Rightarrow \Gamma' ), ( \vdash \text{InferFromAtoms}(T, \Gamma', \langle a_2, \ldots, a_k \rangle, \Gamma'') )</td>
</tr>
<tr>
<td>(P₃)</td>
<td>( \emptyset, (t_1 : \tau_1, \ldots, t_n : \tau_n) \Rightarrow \Gamma_1 ), ( \vdash \text{InferFromAtoms}(T, \Gamma_1, B, \Gamma_2) ), ( \Gamma_2 \vdash t_1 : \sigma_1, \ldots, \Gamma_2 \vdash t_n : \sigma_n )</td>
</tr>
</tbody>
</table>

\( \vdash \text{ClauseHasType}(T, p(t_1, \ldots, t_n) \Rightarrow B, (\tau_1, \ldots, \tau_n) \Rightarrow (\sigma_1, \ldots, \sigma_n)) \)
Bibliography


BIBLIOGRAPHY


## Index

$P_{\text{max}}$, 154  
$\mathcal{R}$, 31  
$\triangleright$, 161  
$\emptyset$, 85  
$\cap$-closed family of types, 94  
$\llbracket \cdot \rrbracket$, 53  
$\cap$, 31  
$\cap$, operation on substitutions, 85  
$\cup$, 31  
$\cup$, operation on substitutions, 85  
$\Rightarrow$, term rewriting system, 80  
$\top^*$, 85  
$\vdash^*$, 85  
$(\top, \bot)$-substitution, 74  
$A(\cdot)$, discriminative approximation, 73  
and, 19  
arity, 15  
$A_\lambda(\cdot, \cdot)$, 114  
assumption type, 85  
atom, 15  
atomic type, 17  

<table>
<thead>
<tr>
<th>Basic type formula</th>
<th>66</th>
<th>BS, best substitution</th>
<th>86</th>
<th>BT, best type</th>
<th>89</th>
</tr>
</thead>
</table>

clause, 15  
ClauseHasType, 33  
cluster, 161  
compatible, 115  
composition, 16, 17  
consequence rules, 31, 177  
Constr, 19  
constraint style, 166  

$\nabla$-derivation, 140  
directional type, 29  
of a program, 32  
directional type of a program, 119  
directly depends, 161  
discriminative approximation, 73  
discriminative type, 74  
domain, 16, 17  

environment, 17  
equality axioms, 20–21  
extension of a clause, 163  

$f_{ac}$, function, 63  
false, 17  
formula, 17  
full assumptions, 168  
generator, 154  
ground type, 17  

$h(\cdot)$, function on environments, 101  
h(·), function on substitutions, 101  
h: h(·), function, height of a type, 76  
head, 17  
head predicate symbol, 15  
height of a type, 76  
Herbrand model, 56  
Herbrand universe, 53  

InferFromAtoms, 33  
l_k(·), function, 75  
L-normal form, 62  
L-reduced, 65  
LR-reduced form, 66  

main type, 154  

$N_L$, function, 63  
non-conflicting, 38  
non-confusing names, 166  
normal form, 66  
$N_R$, function, 64  

or, 19  
ordinary type, 162  

$\mathcal{P}$, 126  

pattern of types, 174  
polymorphic type, 17  
program, 16  
proper, 154  
pruning, 139, 179
INDEX

of substitutions, 144

R-normal form, 62
R-reduced, 65
range, 16, 17
relation $\preceq$, 140
relation $\sim$, 45
relation $\bowtie$, 143
relation $\leq$
  for environments, 28
  for substitutions, 27
  for types, 17
relation $\sim$, 28
restriction, 16, 17
result environment, 91
result type, 91
RG-task, 119
RGP, 124
RMBF, 124
RMBF-term, 124

$s(\cdot)$, function on types, 98
$s(\cdot)$, function on environments, 99
$s(\cdot)$, function on substitutions, 99
safe, 162
saturated clause, 163
$SF$, function, 67
shape(·), 113
signature, 18, 138
simple ground type, 79
simple guarantee, 162
simple shape, 114
strongly typeable predicates, 163
structure, 55
substitution, 16, 17
sufficient assumptions, 167

term, 15, 177
TR-task, 161
type, 16, 176
type constructor, 16
type variable, 16

var, 16, 17

well-typedness, 34