Polymorphic Directional Types for Logic Programming

Pawel Rychlikowski
Institute of Computer Science
Wroclaw University
Poland
prych@ii.uni.wroc.pl

Tomasz Truderung
Institute of Computer Science
Wroclaw University
Poland
tt@ii.uni.wroc.pl

ABSTRACT

In this paper we present a new type system for logic programs. Our system combines ideas of the classical polymorphic, but not very precise, system due to Mycroft and O’Keefe [16], and the complementary system of directional types that has been proposed by Aiken and Lalshman [1].

Our main technical inventions are a new method of deriving more specific types from a given type, which we call pruning, and the notion of the main type from which, using a combination of substitution, subtyping and pruning all types of a predicate can be obtained.

We describe a type checking algorithm, and a type reconstruction algorithm which for a given program and a predicate finds its main type. A complexity analysis of these algorithms is provided. In spite of large theoretical complexity bounds these algorithms work quite fast in practice.

1. INTRODUCTION

There are two traditional approaches to types for logic programs1. The classical system of polymorphic types of Mycroft and O’Keefe was based on ideas of Milner’s type system for the functional language ML, and has been implemented in the Goeodel programming language [13]. This system, however, lacked flexibility and precision (some improvements have been proposed for example in [11]). As a practical consequence many program errors were not detected.

Another approach due to Aiken and Lalshman is based on the notion of a directional type [5, 1, 4, 3, 18] which distinguished input and output types of a predicate. This system is much more precise, it captures both procedural and declarative properties of logic programs, provides a good facility to describing data-flow in a program but its usefulness is limited by the lack of polymorphism. Moreover, in this

This work has been partially supported by the Polish KBN grant No. 8T11C04319

A review of notions related to types for logic programs can be found in [17].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP 2001, Florence, Italy
Copyright 2001 ACM ...$5.00.

system there was not a good notion of a principal type of a predicate. This was related to the fact that predicates could be used in many different modes, (e.g. the predicate append can be used either to concatenate or to split a list).

In this paper we describe a type system for logic programs with the LD-derivation rule [2] which enjoys all essential features of the existing directional type systems; it is moreover polymorphic, incremental, and has subtypes. Furthermore, a predicate has the principal type, from which all types of this predicate can be obtained by subtyping, substitution and a new operation called pruning.

To a large extent we follow Aiken and Lalshman [1]. Directional type of a predicate is a pair

$$\langle (\tau_1, \ldots, \tau_n), (\sigma_1, \ldots, \sigma_n) \rangle$$

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 a predicate. A predicate $$p$$ has the directional type $$(\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)$$ if the following is true: whenever, for $$i \in \{1, \ldots, n\}$$, the term $$t_i$$ has the type $$\tau_i$$ then after a successful derivation of the goal $$p(t_1, \ldots, t_n)$$, the term $$t_i$$ has the type $$\sigma_i$$. In order to describe input and output types Aiken and Lalshman used regular sets of terms (see [9, 15, 12]). They provided a procedure which checks whether a program is well typed with respect to a given set of directional types. The complexity of the type verification and inference in the Aiken-Lalshman system was studied by Charatonik and Podelski [7] who gave an algorithm for the inferring directional types. Charatonik [6] showed that directional type checking is EXPTIME-complete. Types in the Aiken-Lalshman system are not polymorphic. J. Boye and J. Mańuszyski in [3, 4] designed a system where directional types are merged with polymorphic types of terms.

In our type system input and output types are not regular sets. To describe them we follow the notion of a polymorphic type taken from [16] and [14]. Types of terms are constructed from variables and type constructors, we also allow union and intersection of types. Furthermore, we have a subtyping relation. Subtyping and well-typedness are described by a system of derivation rules.

A predicate can have many directional types. Each of them is related to a particular way of using a predicate. In our system the set of all types of a predicate can be described in a compact way using the notion of the main type of a predicate that we develop using a new operation called pruning which like substitution and subtyping, is a method of deriving types.

We provide algorithms for type-checking, and for recon-
struction of main types. We give also some upper and lower bounds for problems of type checking and type reconstruction. The implementation, which is briefly described in the paper, indicates the potential that the system can be used in practice.

This paper is an extended abstract of [19] where technical details and proofs are given.

The paper is organized as follows. In Section 2 we describe types of terms. In Section 3 we introduce types of predicates. We give inference rules that describe the notion of well-typefulness of a program, and we shortly describe a type-checking procedure based on these rules. In Section 4 we introduce the notion of the main type and provide an algorithm computing it. In Section 5 we describe how our type system can be used to eliminate run time errors, and finally in Section 6 the implementation of our system is shortly discussed.

We assume that the reader is familiar with basic concepts concerning logic programming and Prolog (presented for instance in [2] and [20]).

2. TYPES OF TERMS

Let V be the set of program variables (denoted by letters X, Y, . . .), Σ - the set of term constructors (denoted by f, g, . . .), V T - the set of type variables (denoted by α, β, . . .), Σ T - the set of type constructors (like int, real, list, tree, denoted by F, G, . . .). Every element of Σ and Σ T has a fixed arity (number of arguments).

Definition 1. A type is defined by the following grammar:

\[ \tau ::= \alpha | \tau_1 \mid \tau_2 | F(\tau_1, \ldots, \tau_n) | \tau_1 \cap \tau_2 \mid \tau_1 \cup \tau_2 \]

where α ∈ V T, F ∈ Σ T, and F has the arity n. Types constructed using constructor of the arity 0, such that int, real, are called atomic. Types containing type variables are called polymorphic, while types without type variables are called monomorphic.

Definition 2. A type formula has the form τ = σ or t : τ, where τ and σ are types, and t is a term over Σ and V. We denote the set of type formulas by Φ, and elements of this set by φ, ψ, . . .

A formula τ ≤ σ is a shorthand for τ σ = τ. The symbol ≤ expresses a subtyping relation.

Definition 3. An environment Γ is a finite set of formulas of the form (X : τ), where X is a program variable and τ is a type, and for each variable X there is in Γ at most one element of the form (X : τ).

The set \{X | there exists a type τ such that (X : τ) ∈ Γ\} is denoted by dom(Γ). If X ∈ dom(Γ), then Γ(X) denotes the type τ such that (X : τ) ∈ Γ. If X ∉ dom(Γ) we agree to set Γ(X) = τ.

In the remainder of the paper we introduce axioms and rules which allow to prove facts of the form Γ ⊢ φ. We write Γ ⊢ φ instead of \{\}. We write Γ ⊢ φ instead of Γ ⊢ φ.

In our type system (as in the system of Mycroft and O’Keefe) each term constructor has an assigned signature. We write f : s if f is the signature of f. Signatures assigned to term constructors determine types of terms.

We assume that signatures satisfy conditions which are necessary to prove some important properties of our type system related to the notion of the main type presented in Section 4. Since these conditions have technical character, we give them in Appendix A.

Example 1. Assume following signatures for the standard term constructors:

\[ \begin{align*}
\text{h} & : \text{a} \to \text{list(a)} \\
\text{id} & : \text{list(a)} \\
\text{pair} & : \alpha \to \beta \to \text{prod(\alpha, \beta)} \\
0, 1, 2 & : \text{int} \\
5.2 & : \text{real} \\
+ & : \text{expr(\alpha)} \to \text{expr(\alpha)} \\
\end{align*} \]

We use the Prolog-like convention of writing lists. Thus [X|Y|L] represents a list with the head X and the tail L. Moreover, [X,Y|L] is a shorthand for [X,Y|L].

We assume that there is a predefined ordering ≤ S of monomorphic types satisfying conditions given in Appendix A.

Example 2. We can assume the following order on standard types:

\[ \text{int} \leq \text{real}, \quad \text{int} \leq \text{expr(int)}, \quad \text{real} \leq \text{expr(real)} \]

We will use signatures from Example 1 and inequalities from Example 2 in the rest of the paper.

Our system is equipped with equality axioms and term typing rules. They have a fairly standard form, and are given in Appendix B.

Recall that τ ≤ σ is defined as a shorthand for τ σ = τ, thus the equality axioms allow us to prove facts of the form Γ ⊢ τ ≤ σ. Equality axioms guarantee that Γ ⊢ τ ≤ σ whenever τ ≤ σ.

Example 3. Using equality axioms one can prove that

\[ \begin{align*}
\Gamma & \vdash \alpha \cap (\beta \cup \gamma) = (\alpha \cap \beta) \cup (\alpha \cap \gamma), \\
\Gamma & \vdash \text{list(\alpha)} \cap \text{list(\beta)} = \text{list(\alpha \cap \beta)}, \\
\Gamma & \vdash \alpha \leq \text{T}, \\
\Gamma & \vdash \bot \leq \alpha, \\
\Gamma & \vdash \text{list(int)} \cap \alpha \leq \text{list(real)} \cup \beta.
\end{align*} \]

Example 4. We can use term typing rules to prove that

\[ \begin{align*}
\Gamma & \vdash \text{1 + 2} : \text{expr(int)}, \\
\{X : \text{real}\} & \vdash \text{1 + X} : \text{expr(real)}, \\
\{X : \text{real}, L : \text{list(int)}\} & \vdash [X, 2|L] : \text{list(real)}, \\
\{X : \alpha, L : \text{list(\alpha)}\} & \vdash [X, 2|L] : \text{list(\alpha \cup int)}. \end{align*} \]

3. TYPES OF PREDICATES

Now we define types for predicates, namely directional types. We also give some basic intuitions and examples.

Definition 4. If τ1 and σi (for 1 ≤ i ≤ n) are types, then (τ1, . . . , τn) → (σ1, . . . , σn) 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. Sometimes, when it does not cause any confusions, we write τ instead of (τ1, . . . , τn).
The intuitive meaning of a directional types is as follows. A predicate \( p \) has a directional 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 the type \( \tau_i \), then after a successful derivation of the goal \( p(t_1, \ldots, t_n) \), the term \( t_i \) has the type \( \sigma_i \).

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 5.** Consider the following predicates and their types.

\[
\begin{align*}
q(X) & \iff X = 2, \\
q(X) & \iff X = 3.14, \\
r(X, Y, \text{pair}(X, Y)) & : T \rightarrow \text{real}, \\
q & : (T, \tau, \text{prod}(\alpha, \beta)) \rightarrow (\alpha, \beta, \text{prod}(\alpha, \beta))
\end{align*}
\]

The type of the predicate \( q \) means that \( X \), after a 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 a successful call of \( r \) the first argument is of the type \( \tau \), while the second is of the type \( \sigma \).

**Example 6.** A predicate may have many types. Let us consider the program:

\[
p(1). \\
\text{idem}(X, X).
\]

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

**Example 7.** Now consider the predicate **append**.

\[
\text{append}([], X, X). \\
\text{append}([X|Xs], Ys, [X|Xs]) \iff \text{append}(Xs, Ys, Zs).
\]

It has the following types:

\[
\begin{align*}
(1) & : (T, \tau, \text{list}(\gamma)) \rightarrow (\text{list}(\gamma), \text{list}(\gamma), \text{list}(\gamma)), \\
(2) & : (\text{list}(\alpha), \text{list}(\beta), \tau) \rightarrow (\text{list}(\alpha), \text{list}(\beta), \text{list}(\alpha \cup \beta)), \\
(3) & : (\text{list}(\alpha), \text{list}(\beta), \tau) \rightarrow (\text{list}(\alpha \cap \beta), \text{list}(\beta \cap \gamma), \text{list}(\gamma \cap (\alpha \cup \beta))), \\
(4) & : (\tau, \text{list}(\beta), \text{list}(\gamma)) \rightarrow (\text{list}(\beta \cap \gamma), \text{list}(\gamma \cap (\alpha \cup \beta)), \text{list}(\gamma)), \\
(5) & : (\text{list}(\alpha), \tau, \text{list}(\gamma)) \rightarrow (\text{list}(\alpha \cap \gamma), \text{list}(\gamma), \text{list}(\gamma)).
\end{align*}
\]

Let us observe that different types correspond to different ways in which the predicate is used: (1) corresponds to splitting a list into two parts, (2) to appending lists, (3) to checking whether two lists joined together are equal to the third one, (4) to deleting the suffix of the list, (5) to deleting the prefix of the list.

Example 7 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.

### 3.1 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**, relations \( \text{InferFromAtoms} \), and \( \text{ClauseHasType} \).

**Definition 5.** For environments \( \Gamma_1 \) and \( \Gamma_2 \)

\[
\begin{align*}
\Gamma_1 \cap \Gamma_2 & = \{ (X : \tau) \mid X \in \text{dom}(\Gamma_1) \cup \text{dom}(\Gamma_2), \\
& \quad \tau = \Gamma_1(X) \cap \Gamma_2(X) \} \\
\Gamma_1 \cup \Gamma_2 & = \{ (X : \tau) \mid X \in \text{dom}(\Gamma_1) \cap \text{dom}(\Gamma_2), \\
& \quad \tau = \Gamma_1(X) \cup \Gamma_2(X) \}
\end{align*}
\]

In Figure 1 we present rules which allow us to deduce some new information from facts of the form \( \langle t : \tau \rangle \) and accumulate them in an environment. In order to do that, we introduce a new 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 \( \Gamma' \). If \( \varphi \) is of the form \( \langle X : \tau \rangle \), it can be just added to \( \Gamma \), what in fact is done in \( (K_0) \).

The consequence rules are constructed in such a way that the choice of a rule to apply is determined by \( t \) and \( \tau \). Therefore, we have \( (K_2) \) and \( (K_4) \) instead of a more general rule \( \Gamma, \langle t : \tau \rangle \Rightarrow \Gamma \). In rule \( (K_4) \) we deduce an arbitrary environment \( \Gamma'' \) from the obviously false assumption \( t : \bot \) (it is false because no term has the type \( \bot \)).

**Example 8.** One can prove that

\[
\{ X : \alpha, Y : \beta, \{ [X | Y] : \text{list}(\text{int}) \} \Rightarrow \{ X : \alpha \cap \text{int}, Y : \beta \cap \text{list}(\text{int}) \}
\]

**Definition 6.** A directional type of a program is a set

\[
T \subseteq \{ p : \tau \rightarrow \sigma \mid p \text{ is a predicate name}, \\
\quad \tau \rightarrow \sigma \text{ is a directional type} \}
\]

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

In Figure 2 we give rules for relations \( \text{InferFromAtoms} \) and \( \text{ClauseHasType} \). These relations will be used in the definition of well-typedness. The relation

\[
(6) \quad \text{InferFromAtoms}(T, \Gamma, \{ a_1, \ldots, a_n \}, \Gamma')
\]

is defined for a directional type \( T \) of a program, environments \( \Gamma, \Gamma' \) and a sequence \( \{ a_1, \ldots, a_n \} \) of atoms. \( \{ \} \) in the rule \( (P_1) \) represents the empty sequence. The intended meaning of (6) is if predicates have types described by \( T \), variables have types described by \( \Gamma \), then, after the execution of the sequence \( \{ a_1, \ldots, a_n \} \) of atoms, variables get types described by \( \Gamma' \). The relation

\[
(7) \quad \text{ClauseHasType}(T, C, \tau \rightarrow \sigma)
\]

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 (7) is if the clause \( C \) is executed with arguments of the type \( \tau \), then after the execution the arguments 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 the type \( \tau \). Second, \( \text{InferFromAtoms} \) is performed to obtain types of variables after executing the body of the clause. And finally, guarantees
Figure 1: Consequence rules

\[
\begin{align*}
(K_1) & \quad \frac{\Gamma_i, t_i : \tau_i \Rightarrow \Gamma_i \quad (1 \leq i \leq n)}{\Gamma_i, t_1 : \tau_1, \ldots, t_n : \tau_n \Rightarrow \Gamma_i \cap \cdots \cap \Gamma_n} \\
(K_2) & \quad \frac{\Gamma_i, (t_i : \tau_i \theta) \Rightarrow \Gamma_i \quad (1 \leq i \leq n)}{\Gamma_i, (f(t_1, \ldots, t_n) : \tau \theta) \Rightarrow \Gamma_i \cap \cdots \cap \Gamma_n} \\
& \quad \text{if } f : \tau_1 \cdots \cdots \tau_n \rightarrow \tau \\
(K_3) & \quad \frac{\Gamma_i, (f(t_1, \ldots, t_n) : \tau) \Rightarrow \Gamma}{\text{if there is no signature } \tau_1 \cdots \cdots \tau_n \rightarrow \tau_0 \text{ assigned to } f} \\
& \quad \text{such that } \text{head}(\tau) = \text{head}(\tau_0) \\
(K_4) & \quad \frac{\Gamma_i, (t : \bot) \Rightarrow \Gamma}{\Gamma'} \\
(K_5) & \quad \frac{\Gamma_i, (t : \top) \Rightarrow \Gamma}{\Gamma} \\
(K_6) & \quad \frac{\Gamma_i, (f(t_1, \ldots, t_n) : \alpha) \Rightarrow \Gamma}{\Gamma} \\
(K_7) & \quad \frac{\Gamma_i, (t : \tau_i) \Rightarrow \Gamma_i \quad (1 \leq i \leq n)}{\Gamma_i, (t : \tau_1 \cap \cdots \cap \tau_n) \Rightarrow \Gamma_i \cap \cdots \cap \Gamma_n} \\
(K_8) & \quad \frac{\Gamma_i, (t : \tau_i) \Rightarrow \Gamma_i \quad (1 \leq i \leq n)}{\Gamma_i, (t : \tau_1 \cup \cdots \cup \tau_n) \Rightarrow \Gamma_i \cup \cdots \cup \Gamma_n} \\
(K_9) & \quad \frac{\Gamma_i, (X : \tau) \Rightarrow \Gamma \cap \{X : \tau\} \quad \text{if } \tau \neq \bot \text{ and } \tau \neq \top}{\Gamma} 
\end{align*}
\]

Figure 2: Program typing rules

\[
\begin{align*}
(P_1) & \quad \vdash \text{InferFromAtoms}(T, \Gamma, (\), \Gamma) \\
& \quad \Gamma \vdash t_i : \tau_i \theta, \quad \Gamma \vdash t_i : \tau_i \theta, \quad \Gamma, (t_1 : \sigma_1 \theta, \ldots, t_n : \sigma_n \theta) \Rightarrow \Gamma' \quad \vdash \text{InferFromAtoms}(T, \Gamma', (a_2, \ldots, a_k, \Gamma'')) \\
& \quad \vdash \text{InferFromAtoms}(T, \Gamma, (p(t_1, \ldots, t_n), a_2, \ldots, a_k, \Gamma'')) \\
& \quad \text{if } (p : (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)) \in T, \\
\end{align*}
\]

\[
\begin{align*}
(P_2) & \quad \vdash \text{InferFromAtoms}(T, \Gamma, (p(t_1, \ldots, t_n), a_2, \ldots, a_k, \Gamma'')) \\
& \quad \emptyset, (t_1 : \tau_1, \ldots, t_n : \tau_n) \Rightarrow \Gamma_1, \\
& \quad \vdash \text{InferFromAtoms}(T, \Gamma_1, B, \Gamma_2), \quad \Gamma_2 \vdash t_1 : \sigma_1, \ldots, \Gamma_2 \vdash t_n : \sigma_n \\
& \quad \vdash \text{ClauseHasType}(T, p(t_1, \ldots, t_n) \vdash B, (\tau_1, \ldots, \tau_n) \rightarrow (\sigma_1, \ldots, \sigma_n)) \\
\end{align*}
\]

\[
\begin{align*}
(P_3) & \quad \vdash \text{ClauseHasType}(T, C, \tau \rightarrow \sigma) \\
& \quad \text{holds for each } C \in P, \text{ and for each directional type } (\tau \rightarrow \sigma) \text{ such that } (h(C) : \tau \rightarrow \sigma) \in T. \\
\end{align*}
\]

are checked, i.e. it is proved that argument of the predicate has the type \( \sigma \).

The next definition introduces the notion of well-typedness of a program. In this definition, \( h(C) \) denotes the predicate symbol of the head of the clause \( C \), i.e. if \( C \) is of the form \( p_0(t_1^0, \ldots, t_m^0) \vdash p_1(t_1^1, \ldots, t_n^1), \ldots, p_k(t_1^k, \ldots, t_n^k) \) then \( h(C) = p_0 \).

**Definition 7.** A program \( P \) is well-typed with respect to a directional type \( T \) if:

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

holds for each \( C \in P \), and for each directional type \( (\tau \rightarrow \sigma) \) such that \( (h(C) : \tau \rightarrow \sigma) \in T \).

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

In Appendix C we present the Soundness Theorem which states that the proof system defined by axioms and rules of our system is sound.

### 3.2 Type checking

In this section we shortly describe the type checking algorithm. Details and proofs can be found in [19]. We need the following definition.

**Definition 8.** The result type for the clause \( C \), the directional type \( T \) of a program and the assumption \( \tau \) is the least type \( \sigma \) such that \( \vdash \text{ClauseHasType}(T, C, \tau \rightarrow \sigma) \).

The program typing rules \((P_1)-(P_3)\) together with the consequence rules \((K_1)-(K_3)\) can be used to compute the result type.

Now suppose that we have a program \( P \) and a directional type \( T \). In order to check whether or not \( P \) is well-typed with respect to \( T \), we have to check, according to Definition 7, for every clause \( C \) and for every \( (h(C) : \tau \rightarrow \sigma) \in P \)
whether ⊢ ClauseHasType($T, C, \tau \rightarrow \sigma$) holds. In order to do it we compute the result type for $C$, $\tau$ and $T$ and then check whether it is less than $\sigma$. So the type checking algorithm consist of two stages: computing the result types and comparing these types with the guarantees from the types which we want to prove.

There is a form of type inequalities which can be checked in polynomial time. Our algorithm in order to compare types transforms them and obtains the desired form of the checked formula. It can cause the exponential growth of the formula. The disjunctive and conjunctive normal forms of boolean formulas can serve here as a not very far analogy.

**Lemma 1.** The type-checking is coNP-hard.

**Proof (sketch).** We will show how to reduce checking whether a boolean formula is a tautology to checking whether a program has a given type.

For a given propositional formula $\Phi$ we construct a program $P$ and a directional type $T$ of a program in such a way that $\Phi$ is a tautology if and only the program $P$ written below has the type $T$. Assume that $\Phi$ is built using the variables $\alpha_1, \ldots, \alpha_n$. We can assume, without lost of generality, that the negation appear only in front of the variables.

Here is the program $P$:

$$
\begin{align*}
\text{or} & (X,Y,X). \\
\text{or} & (X,T,T). \\
\text{and} & (X_1, X_2, \ldots, X_n, \forall \alpha_1 \rightarrow \forall \alpha_n). \text{ where } \forall \alpha \text{ has arity } n=1 \\
p & (X_1, X_1', \ldots, X_n, X_n', Z):
\end{align*}
$$

and the directional type of a program $T$:

$$
\begin{align*}
p & : (\alpha_1, \alpha_1', \ldots, \alpha_n, \alpha_n', T) \rightarrow \\
(\alpha_1, \alpha_1', \ldots, \alpha_n, \alpha_n', \forall (\alpha_1 \cap \alpha_1') \cup \ldots \cup (\alpha_n \cap \alpha_n')) \\
or & : (\alpha, \beta, T) \rightarrow (\alpha, \beta, \alpha \cup \beta) \\
\text{and} & : (\alpha_1, \ldots, \alpha_n, T) \rightarrow (\alpha_1, \ldots, \alpha_n, \alpha_1 \cap \ldots \cap \alpha_n)
\end{align*}
$$

where $\forall \alpha$ is obtained from $\Phi$ by replacing each literal $\neg \alpha_i$ by a new variable $\alpha_i'$, and replacing $\land$ by $\lor$ and $\lor$ by $\land$. □

**Lemma 2.** Our type checking algorithm works in EXP-TIME.

4. **MAIN TYPE**

4.1 **Deriving types**

A predicate can have many directional types. Polymorphism and subtyping describes natural relations between them. In our system these relations are formulated as follows:

**Definition 9.** Let $(\tau_1 \rightarrow \sigma_1)$, $(\tau_2 \rightarrow \sigma_2)$ be directional types. $(\tau_2 \rightarrow \sigma_2)$ is derived from $(\tau_1 \rightarrow \sigma_1)$ if there exists a substitution $\theta$ such that $\vdash \tau_2 \leq \tau_1 \theta$ and $\vdash \sigma_2 \leq \theta \sigma_1$.

When $(\tau_2 \rightarrow \sigma_2)$ is derived from $(\tau_1 \rightarrow \sigma_1)$, we write $(\tau_1 \rightarrow \sigma_1) \vdash (\tau_2 \rightarrow \sigma_2)$.

As we can see, $(\tau_1 \rightarrow \sigma_1) \vdash (\tau_2 \rightarrow \sigma_2)$ holds if $(\tau_2 \rightarrow \sigma_2)$ can be obtained from $(\tau_1 \rightarrow \sigma_1)$ by weakening (wrt. subtyping relation) and substitution. It is easy to prove that relation is transitive and reflexive.

**Theorem 1.** (Derivation Theorem). Assume that a program $P$ is well typed with respect to $T$, and

$$
T' = \{(p : \tau \rightarrow \sigma) \mid (p : \tau \rightarrow \sigma) \in T, \quad (\tau \rightarrow \sigma) \vdash (\tau' \rightarrow \sigma')\}
$$

Then the program $P$ is also well typed with respect to $T'$.

Informally, Theorem 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 $(\tau \rightarrow \sigma) \vdash (\tau' \rightarrow \sigma')$.

Owing to the Derivation Theorem we can obtain a family of valid types of some predicate from one type of this predicate. In some type systems such a mechanism guarantees the existence of the principal type, i.e. the type of a predicate from which other types of this predicate can be derived.

In the case of directional types existence of principal types is rather problematic. In Example 7 we have given five different types of append. All of them are useful, but there is no type from which all these types can be derived. This creates a serious problem, because some predicates may have exponentially many types (with respect to the length of a predicate). In this section we present a new method of deriving types, called pruning, which enables us to develop the notion of the main type of a predicate.

Before we give the exact definition, we introduce the pruning operation informally. We define pruning for a set $A$ of type variables, and denote it by $\nabla_A$. Pruning preserves $\bot$, $\top$, $\land$ and $\lor$. Furthermore, $\nabla_A(\alpha) = \alpha$, when $\alpha \notin A$. The crucial point of the definition is as follows: If a type variable $\alpha \in A$ then $\nabla_A(\alpha) = \top$. Moreover, $\nabla_A(F(\tau_1, \ldots, \tau_n))$ collapses to $\top$ whenever $\nabla_A$ recursively applied to $\tau_1, \ldots, \tau_n$ gives $\top$.

**Definition 10.** For a set $A$ of type variables $\nabla_A$ is the function $\nabla_A$ whose domain and range are sets of types. $\nabla_A$ is defined by the following equations:

$$
\begin{align*}
\nabla_A(\alpha) & = \begin{cases} 
\top & \text{if } \alpha \in A \\
\alpha & \text{if } \alpha \notin A
\end{cases} \\
\nabla_A(F(\tau_1, \ldots, \tau_n)) & = \begin{cases} 
\top & \text{if } \forall i \in \{1, \ldots, n\} \\
\vdash \nabla_A(\tau_i) = \top & \\
F(\nabla_A(\tau_1), \ldots, \nabla_A(\tau_n)) & \text{otherwise}
\end{cases}
\end{align*}
$$

As we can see, pruning is a simple syntactic operation. Consider an example. Let $\tau = \text{prod } \text{list}(\alpha), \text{list}(\alpha \cap \beta)$. Then we have

$$
\begin{align*}
\nabla_{\{\alpha\}}(\tau) & = \text{prod } \top, \text{list}(\top \cap \beta) \\
\nabla_{\{\alpha, \beta\}}(\tau) & = \top
\end{align*}
$$

Since $\vdash \top \cap \beta = \beta$, we can write

$$
\vdash \nabla_{\{\alpha\}}(\tau) = \text{prod } \top, \text{list}(\beta)
$$

**Theorem 2.** (Pruning Theorem). Assume that a program $P$ is well typed with respect to $T$ and

$$
T' = T \cup \{(p : \nabla_A(\tau) \rightarrow \nabla_A(\sigma)) \mid A \subseteq V_T, \quad (p : \tau \rightarrow \sigma) \in T\},
$$
then the program \( P \) is also well typed with respect to \( T' \).

The proof is given in [19]. Informally, the theorem states that, for any set \( A \) of type variables, if a predicate \( p \) has the type \( \tau \rightarrow \sigma \) then it also has the type \( \forall A(\tau) \rightarrow \forall A(\sigma) \).

Now, as we have a new tool that, together with substitution and weakening, can derive a correct type of a predicate from another correct type, we can extend the definition of a derivation.

**Definition 11.** Let \( \tau_1 \rightarrow \sigma_1, \tau_2 \rightarrow \sigma_2 \) be directional types.

\[
(\tau_1 \rightarrow \sigma_1) \vdash (\tau_2 \rightarrow \sigma_2) \quad \text{iff} \quad \exists A \, \forall A(\tau_1) \rightarrow \forall A(\sigma_1), \quad (\tau_2 \rightarrow \sigma_2).
\]

**Definition 12.** Let \( T \) and \( T' \) be directional types of a program. \( T \) is \( \forall \)-derivable from \( T' \) if for each \( (p : \tau \rightarrow \sigma) \in T \) there exists \( (p : \tau' \rightarrow \sigma') \in T' \) such that \( \forall A(\tau) \rightarrow \forall A(\sigma) \).

The following corollary is a consequence of Theorem 1 and Theorem 2.

**Corollary 1.** If \( T' \) is \( \forall \)-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 9.** Let us analyze how pruning works for the predicate \( \text{append} \). In our system one can prove that \( \text{append} \) has the type:

\[
(\text{list}(\alpha), \text{list}(\beta), \text{list}(\gamma)) \rightarrow (\text{list}(\alpha \cap \gamma), \text{list}(\beta \cap \gamma), \text{list}(\gamma \cap (\alpha \cup \beta)))
\]

Let us denote this type by \( \tau \) and prune it as follows:

\[
\vdash \forall(\tau) = (T, \text{list}(\beta), \text{list}(\gamma)) \rightarrow (\text{list}(\gamma), \text{list}(\beta \cap \gamma), \text{list}(\gamma))
\]

\[
\vdash \forall(\beta) = (\text{list}(\alpha), T, \text{list}(\gamma)) \rightarrow (\text{list}(\alpha \cap \gamma), \text{list}(\gamma), \text{list}(\gamma))
\]

\[
\vdash \forall(\alpha) = (T, \text{list}(\gamma), \text{list}(\gamma)) \rightarrow (\text{list}(\gamma), \text{list}(\gamma), \text{list}(\gamma))
\]

\[
\vdash \forall(\gamma) = (T, T, \text{list}(\gamma)) \rightarrow (\text{list}(\gamma), \text{list}(\gamma), \text{list}(\gamma))
\]

\[
\vdash \forall(\alpha, \beta, \gamma) = (T, T, T) \rightarrow (T, T, T)
\]

All the types above are types of \( \text{append} \), and all, except of the last trivial one, were discussed in this paper. Therefore, the type (8) is a good candidate for the main type of \( \text{append} \).

**4.2 Definition of the Main Type**

**Definition 13.** A directional type \( \tau \rightarrow \sigma \) is proper if \( \forall_0(\sigma) = \sigma \). A directional type \( T \) of a program is proper if all types occurring in \( T \) are proper.

One can see that the directional type \( (\tau \rightarrow \sigma) \) is not proper when \( \sigma \) contains a part of the form \( \forall F(T, \ldots, T) \). For instance, if \( \sigma = \text{prod}(T, \alpha) \) then \( (\tau \rightarrow \sigma) \) is proper. But when \( \sigma = \text{prod}(T, T) \), then \( (\tau \rightarrow \sigma) \) is not proper. That is because \( \forall_0(\text{prod}(T, T)) = T \).

**Example 10.** As an another example can serve directional type

\[
(9) \quad (T, T, T) \rightarrow (\text{list}(T), T, T).
\]

One can prove that it is a type of the predicate \( \text{append} \). This type is not proper, since \( \forall_0(\text{list}(T), T, T) = (T, T, T) \).

Now we define the set of all proper types of a given program, namely \( T_{prop} \).

**Definition 14.** For a program \( P \) let \( X = \{ T \mid T \) is a proper directional type such that \( P \) is well typed with respect to \( T \} \). We define \( T_{prop} = \bigcup_{T \in X} T \).

Roughly speaking, \( T_{prop} \) contains all proper types of predicates of \( P \). One can easily check that \( P \) is well typed w.r.t \( T_{prop} \). Moreover, \( T_{prop} \) is the biggest set of proper types with this property: if we add any proper type to \( T_{prop} \) obtaining \( T' \), then \( P \) is not well typed w.r.t \( T' \).

**Definition 15.** A directional type \( T \) of a program \( P \) is the main type of \( P \) if

(i) for each predicate \( p \) of the program \( P \), \( T \) contains exactly one element of the form \( (p : T \rightarrow \sigma) \),

(ii) \( T_{prop} \) is \( \forall \)-derivable from \( T \).

For a fixed program \( P \), \( \tau \rightarrow \sigma \) is the main type of a predicate \( p \) if \( (p : T \rightarrow \sigma) \) belongs to the main type of \( P \).

Not all programs have main types, which is not an disadvantage. Prolog is a type-free programming language, so it is easy to imagine syntactically correct Prolog programs that are not type-correct, no matter how we exactly understand the latter. Type systems can reject such programs, moreover, it should do so if it is to be an error detector. On the other hand, it is quite understandable to want other predicates meeting the requirements of type correctness to have the main type.

Now we have to specify the term ‘type-correctness’ so far used informally. Our perspective will be the Mycroft-O’Keefe type paradigm on which many existing type systems for logic programming languages are based. As a result of that, type correct programs are understood as those for which principal types in the Mycroft-O’Keefe type system can be reconstructed. In the next section we show that there are main types for this class of programs (in fact, due to subtyping, the main type exists for the slightly wider class of programs)\(^2\). The algorithm finding such a type is also given in the next section.

It should be also mentioned that, directional types derived from the main type of a predicate describe semantics of the predicate more precisely than those in the Mycroft-O’Keefe type system.

As it is stated in Definition 15, all proper types of predicates can be \( \forall \)-derived from the main type. But it may be not the case when we consider some improper types. For instance, it turns out that (9) cannot be \( \forall \)-derived from the main type of \( \text{append} \). It can be shown that all other types of \( \text{append} \) can be \( \forall \)-derived from the main type.

\(^2\)It can be also shown that the main type of such programs is unique modulo variables renaming and type equivalence.
4.3 Reconstruction of types

In this section we present an algorithm which for a given program $P$ finds its main type. For the sake of simplicity of presentation we restrict our attention to programs which do not contain mutually recursive predicates. The algorithm, however, could be easily adapted to handle the case of mutually recursive predicates (see [14], [19]).

Definition 16. Let $P$ be a program. Let $p$, $q$ be predicate names. We say that $q \preceq p$ if and only if there is a clause in $P$ with $p$ in the head and $q$ in the body. We denote by $\preceq^+$ the transitive and reflexive closure of $\preceq$.

While reconstructing types we analyze predicates in order given by $\preceq^+$. The input of a single step is a predicate $p$, and types of predicates which are called in $p$. Such a step consists of two stages. In the first stage, which is similar to the type reconstruction in functional languages [8] or in some logic programming languages [14], we find the left-hand side of the main type. In the second stage we reconstruct the right-hand side: we start from the very strong guarantees (probably incorrect) and try to make them correct by weakening.

We emphasize the connection between the algorithm which finds the main type of a program with the type reconstruction algorithm for the Mycroft-O’Keefe type system (we call the latter MOTR). Our algorithm uses MOTR as a procedure. The input to this procedure consists of an untyped predicate $p$, non-directional types for predicates used in $p$ and signatures for term constructors. The equivalent version of the algorithm without such ‘external calls’ can be found in [19].

The following definitions describe how to obtain non-directional types used in the Mycroft-O’Keefe type system (we will call them MO-types\(^3\)) when directional types are given. The function $\mathcal{E}$ finds for a given type the set of equations. The solution of these equations is then used while converting directional types to MO-types.

Definition 17. The function $\mathcal{E}$ takes a type and a variable and returns the set of equations. It is given by the following equations.

\[
\begin{align*}
\mathcal{E}(\alpha, \beta) &= \{ \alpha = \beta \} \\
\mathcal{E}(\tau_1 \cap \tau_2, \beta) &= \mathcal{E}(\tau_1, \beta) \cup \mathcal{E}(\tau_2, \beta) \\
\mathcal{E}(\tau_1 \cup \tau_2, \beta) &= \mathcal{E}(\tau_1, \beta) \cup \mathcal{E}(\tau_2, \beta) \\
\mathcal{E}(F(\tau_1, \ldots, \tau_n), \beta) &= \mathcal{E}(\tau_1, \beta_1) \cup \cdots \cup \mathcal{E}(\tau_n, \beta_n) \\
&\cup \{ F(\beta_1, \ldots, \beta_n) = \beta \} \\
\mathcal{E}(\alpha, \beta) &= \emptyset \\
\mathcal{E}(\tau_1, \ldots, \tau_n, \beta) &= \mathcal{E}(\tau_1, \beta_1) \cup \cdots \cup \mathcal{E}(\tau_n, \beta_n) \\
&\cup \{ \beta_1, \ldots, \beta_n = \beta \}
\end{align*}
\]

where $\alpha, \beta$ are variables, $\beta_1, \ldots, \beta_n$ are new distinct variables, $\tau_1, \tau_2$ are types, $\alpha$ is an atomic type or $a \in \{ \bot, T \}$, and $F$ is a type constructor.

The function $U$, which is defined below, converts a directional type to the MO-type.

Definition 18. Let $\tau \rightarrow \sigma$ be a directional type. Let $S$ be $\mathcal{E}(\sigma, a)$ where $a$ is a new variable. Suppose that the most general unifier for $S$ exists and is equal to $\Theta$. Then we define

\[
U(\tau \rightarrow \sigma) = \Theta(a)
\]

For example, for every type $\tau \rightarrow \sigma$ of the predicate `append` defined in Example 7 we obtain

\[
U(\tau \rightarrow \sigma) = (\text{list}(a), \text{list}(a), \text{list}(a)).
\]

Figure 3 shows the type reconstruction algorithm. The function Guarantee reconstructs the right-hand side of the main type when the left-hand side is given, function MainTypeForPredicate reconstructs the main type of one predicate, while the reconstruction of the main type of a program is done by the function MainType.

We compute new signatures $S'$ before calling MOTR, because there is no subtyping in Mycroft-O’Keefe type system, and an attempt to unify `int` and `real` would cause a type clash. So we postpone the problems with atomic types up to the reconstruction of guarantee.

In the next example we show how the function MainTypeForPredicate works during the reconstruction of the main type of `append`.

Example 11. There are no predicates used in `append`. Suppose, however, to make this example more informative, that the second clause of `append` has the form:

\[
\text{append}(\text{IX}[[x]], \text{Y}, \text{IZ}[[z]])::= x=2, \text{append}(\text{X}, \text{Y}, \text{Z}).
\]

The predicate $=$ (the only predicate used in `append`) has type $T = (\alpha, \beta) \rightarrow (\alpha \land \beta, \alpha \land \beta)$ and $U(T) = (\alpha, \alpha)$. There are no terms of atomic types in $U$, so standard signatures can be used. Now we call MOTR using standard signatures and type $(\alpha, \alpha)$ for $=$. This procedure returns the type $(\text{list}(\alpha), \text{list}(\beta), \text{list}(\gamma))$ which is then transformed to

\[
\tau = (\alpha_0 \land \text{list}(\alpha), \beta_0 \land \text{list}(\beta), \gamma_0 \land \text{list}(\gamma)).
\]

which is the left-hand side of the main type.

While computing the function Guarantee we start with $\sigma_0 = (\bot, \bot, \bot)$, then, after analyzing of the first clause, we obtain the type

\[
\sigma_1 = (\alpha_0 \land \text{list}(\bot), \beta_0 \land \text{list}(\beta \land \gamma), \gamma_0 \land \text{list}(\beta \land \gamma)).
\]

The value of ResultType for the second clause is equal to

\[
\sigma_2 = \sigma_3 =
\]

\[
(\alpha_0 \land \text{list}(\alpha \land \gamma), \beta_0 \land \text{list}(\beta \land \gamma), \gamma_0 \land \text{list}(\alpha \land \beta \land \gamma)).
\]

Taking the first or the second clause again cannot cause any increase of the guarantee, so we have reached the fixed point. The pair \((10)\) and \((11)\) is the main type of `append`.

In this case the variables $\alpha_0, \beta_0$ and $\gamma_0$ do not propagate any information from the assumption to the guarantee and therefore can be deleted in order to make the type more readable:

\[
(\text{list}(\alpha), \text{list}(\beta), \text{list}(\gamma))
\]

Now we state some important facts. The proofs can be found in [19].
function Guarantee(p : predicate, τ : assumptions, T : typing)
    let σ₀ be (⊥, ..., ⊥)
    add (p : τ → σ₀) to T
    let i be 0
    repeat
        let i be i + 1
        take a clause C from P with b(C) = p
        let σᵢ be σᵢ₋₁ ⊔ ResultType(T, τ, C)
        replace (p : τ → σᵢ₋₁) by (p : τ → σᵢ) in T
        until ⊢ σᵢ₋₁ = σᵢ
    return σᵢ

function MainTypeForPredicate(p : predicate, T : typing, S : signatures)
    let Q be the set of predicates called in p
    for each predicate q ∈ Q with a main type τ → σ
        compute mᵦ = U(τ → σ)
        let S' be {e : a | (e : a) ∈ S, a is atomic, a is a new variable} ∪ {f : s | (f : s) ∈ S, s = (τ₁ ⋯ τₙ → τ), n > 0}
        let M be {mᵦ | q ∈ Q}
        let mᵦ be MOTR(S', M, p)
    if mᵦ = "type error" then return "type error"
    let τ₀ be mᵦ in which each variable is replaced by a new distinct one
    let τ be the type obtained from τ₀ by replacing
        each subexpression of the form F(...(α F(...(β F(...)))) by α F(...)
        where α is a new distinct variable
    let σ be Guarantee(p, τ, T)
    return τ → σ

function MainType(P : program, S : signatures)
    set T to ⊥
    repeat
        take an untyped predicate p in P minimal according to ≤ₚ
        let T be MainTypeForPredicate(p, T, S)
        if T = "type error" return "type error"
        else add (p : T) to T
    until all predicates have main types
    return T

Theorem 3. The function MainType eventually stops and returns the main types of all predicates in a program or reports ‘type error’.

Theorem 4. If all predicates in P have types which could be found by the algorithm MOTR then the function MainType returns the main type of P.

Many type systems for logic programming are based on the Mycroft-O’Keefe paradigm. The last theorem says that ‘reasonable’ predicates, i.e. predicates which have reconstructable types in the Mycroft-O’Keefe type system, have main types in our type system, and the algorithm MainType is able to find them.

Lemma 3. The reconstruction of guarantees is EXPTIME-hard.

Proof of the last lemma is by reduction of RMBF problem

We have no proof of the complexity of the function MainType in the general case. However, there is an important class of discriminative types for which we are able to give some upper bounds. We need the following definitions:

Definition 19. Let τ₁, τ₂ are types. We say that τ₂ is more discriminative than τ₁ (written τ₁ ≤D τ₂) if there are types τ₁', τ₂' such that ⊢ τ₁ = τ₁', τ₂' is obtained from τ₁' by replacing some expression of the form F(σ₁, ..., σₙ) ∪ F(ρ₁, ..., ρₙ) by F(σ₁ ∪ ρ₁, ..., σₙ ∪ ρₙ) and ⊢ τ₂ = τ₂'.

Definition 20. A type τ is discriminative if for every type τ' such that τ ≤D τ' we have that ⊢ τ = τ'.

Example 12. Types list(γ ∩ (α ∪ β)) and expr(real) are discriminative, while list(α) ∪ list(β) is not since

list(α) ∪ list(β) ≤D list(α ∪ β)

and ⊢ list(α) ∪ list(β) = list(α ∪ β) does not hold. Similarly, we can show that prod(T, expr(int) ∪ expr(β ∩ real)) is not discriminative.
In our type system for almost all predicates their main types are discriminative. Our notion of discriminativeness corresponds to discriminative types from [1] and to tuple distributive sets from [23] (see [29] for details).

The following lemma holds:

**Lemma 4.** If predicates have discriminative main types then the algorithm MainTypeOfPredicate works in 2-EXPSPACE.

As our practical experiments show this high theoretical complexity has no importance in practice.

5. **CONSTRANDED DIRECTIONAL TYPES**

Constrained directional types provide a mechanism which allows us to express restrictions of the way in which predicates can be applied.

The main idea is a convenient and compact method of representing the set of all proper types of a predicate. When we attempt to type a new predicate, all we need to know about previously typed predicates are their main types, since other proper types can be \( \land \)-derived from them. Since each predicate has a proper type of the form \((\top, \ldots, \top) \rightarrow (\top, \ldots, \top)\), our type system so far does not give a method to restrict possibility of applying a predicate to a given term.

**Definition 21.** A pair

\[
Q = ((\tau \rightarrow \sigma), (a_1 \leq b_1, \ldots, a_n \leq b_2))
\]

is a constrained directional type if \( \tau \rightarrow \sigma \) is a directional type, and, for each \( i \in \{1, \ldots, n\} \), \( b_i \) is a monomorphic type, and \( a_i \) is a type constructed using only operators \( \land, \lor \) and variables occurring in \( \tau \) and \( \sigma \).

For a constrained directional type \( Q = ((\tau \rightarrow \sigma), (a_1 \leq b_1, \ldots, a_n \leq b_2)) \) we define the set \( G(Q) \) of types generated from \( Q \):

\[
G(Q) = \{ \theta(\nabla_A(\tau)) \rightarrow \theta(\nabla_A(\sigma)) \mid \theta \text{ is a type substitution, } A \subseteq V, \text{ and, for } i \in \{1, \ldots, n\}, \theta(\nabla_A(a_i)) \leq \theta(\nabla_A(b_i)) \}.
\]

**Definition 22.** A set \( S \) of the form

\[
\{ (p : Q) \mid p \text{ is a predicate name, } Q\text{ is a constrained directional type} \}
\]

is called a constrained type 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 \( S \) if it is well-typed with respect to \( G(S) \).

**Definition 23.** \( Q \) is the most general constrained type of a program \( P \) iff \( P \) is well-typed with respect to \( Q \), and, for any constrained type \( Q' \) such that \( P \) is well-typed with respect to \( Q' \), \( G(Q) \supseteq G(Q') \).

One can show that a program has the most general constrained type, whenever it has the main type. In fact, the most general constrained type consists of the main type, and the less restrictive inequations.

The idea of deal with constrained types is as follows. The user can provide constrained types for some predicates. Moreover, for some standard predicates (like arithmetic predicates) constrained types are provided.

Now, for the untyped part of a program, the system should reconstruct the most general constrained type. We have an algorithm of reconstructing the most general constrained type. It is in fact a slightly modified version of the type reconstruction algorithm where conditions are computed together with guarantees (see Figure 3).

**Example 13.** Assume that the arithmetic predicate \( \equiv \) has constrained type

\[
(\alpha, \beta) \rightarrow (\alpha, \beta), \quad \alpha \leq \text{expr}(\text{real}), \beta \leq \text{expr}(\text{real}).
\]

(The condition in this type can be expressed in the equivalent form \( \alpha \land \beta \leq \text{expr}(\text{real}) \) as well.) Consider the following predicate.

\[
\tau(X, Y) := X \equiv Y, \quad Y < Y.
\]

The following constrained type for \( \tau \) is reconstructed:

\[
(\alpha, \beta) \rightarrow (\alpha \land \beta, 0 \land \beta), \quad \alpha \land \beta \leq \text{expr}(\text{real}).
\]

Note that an ordering of atoms in a clause is meaningful. If we change the ordering of atoms, we can obtain different conditions. Consider a predicate \( \tau' \):

\[
\tau'(X, Y) := X < Y, \quad X = Y.
\]

The most general constrained type for this predicate is

\[
(\alpha, \beta) \rightarrow (\alpha \land \beta, \alpha \land \beta), \quad \alpha \land \beta \leq \text{expr}(\text{real}).
\]

Hence, goal \( :- \tau'(2, X) \) is typeable, while goal \( :- \tau'(2, X) \) is not (the system reports a type-error).

More involved example is given in the next section where implementation of our system is discussed.

6. **IMPLEMENTATION**

Our type system (including the type checking and constrained type reconstruction algorithm) has been implemented in C++. We tested our implementation on predicates from the Sterling & Shapiro book [20]. In spite of the high theoretical complexity type checking and type reconstruction algorithms perform quite well. Reconstruction of the main type for a program with 74 predicates, 151 clauses and 497 lines took on computer PC with Pentium II and gcc compiler less then 2 seconds.

Besides Prolog constructs discussed in this paper we considered the following: 

\[
\cup, \cap, \lor, \land, \neg, \not, \text{findall, bagof, assert, retract}, \text{etc.} \]

The second order predicates such as findall or assert can be used only when the second order argument is given explicitly, for instance:

\[
\text{findall}(X, \text{perms}(\{1,2,3\}, X), Y).
\]

Below we use + and * instead of \( \cup \) and \( \land \) every signature definition begins with the keyword \( \text{sig} \), and after the function name we write \( \text{::} \). Type declarations are proceeded by the keyword \( \text{typeof} \). We separate the parts of a constrained type using the keyword \( \text{where} \).

Let us consider the program:

\[
\text{sig} \quad '.' \quad A \ast \text{list}(A) \rightarrow \text{list}(A).
\]

\[
\text{sig} \quad [\] \quad \text{::} \text{list}(	ext{bot}).
\]

\[
\text{typeof} \quad '\checkmark' \quad (A,B) \rightarrow \text{list}(A, B) \text{ where } A + B = \text{expr}.
\]

\[
\text{typeof} \quad '=' \quad (A,B) \rightarrow \text{list}(A, B) \text{ where } A + B = \text{expr}.
\]
quicksort((X|Xs), Ye) :-
    partition(Xe, X, Ls, Bbig),
    quicksort(Ls, Ls),
    quicksort(Bbig, Bs),
    append(Ls, (X|Xs), Ye).

quicksort([], []).

partition((X|Xs), Y, (X|Xs), Bs) :-
    X =< Y, partition(Xs, Y, Ls, Bs).

partition((X|Xs), Y, Ls, Bs) :-
    X > Y, partition(Xs, Y, Ls, Bs).

partition([], [], []).

For this program the type reconstruction algorithm computed the following main types:

qusort : (list(a), list(b) ) -> (list(a), list(a ∩ b)),
α ≤ exp(real)

partition : (list(a), β, list(gamma), list(delta))
            -> (list(a ∩ (γ ∪ δ)), β, list(a ∩ γ ), list(a ∩ δ)),
            α ∩ δ ≤ exp(real),
            α ∩ γ ≤ exp(real), β ≤ exp(real)

By pruning we can obtain types which correspond to the most common ways of using these predicates:

qusort : (list(a), T ) -> (list(a), list(a)), α ≤ exp(real)
partition : (list(a), β, T, T ) -> (list(a), β, list(a), list(a)),
α ≤ exp(real), β ≤ exp(real)

The restrictions described by constrained type is caused by Prolog's way of treating the arithmetic predicate =<. For instance using quicksort with first argument which is not a list of arithmetic expressions would cause the run-time error: in this situation =< would be called with arguments which cannot be evaluated to numbers.

7. CONCLUSIONS

In this paper we presented a type system which combines the Mycroft-O’Keefe polymorphic types with the notion of directional types introduced by Aiken and Lakshman in [3]. We introduced a new method of deriving types, called pruning, and developed the notion of the main type. All proper types of the predicate can be derived from its main type. Moreover we give an algorithm which computes the main type for a given program. We think that in spite of the high theoretical complexity, our type checking algorithm performs sufficiently fast to be used in practice.

8. ACKNOWLEDGEMENTS

We are grateful to professor Leszek Pacholski for his careful and constructive criticism. We also would like to thank the anonymous referees for their helpful comments and suggestions.

9. REFERENCES


APPENDIX

A. SIGNATURES

Definition 24. $\tau_1 \ast \cdots \ast \tau_n \rightarrow \tau$ is a signature of the arity $n$ if

(i) $\tau$ has the form $F(\alpha_1, \ldots, \alpha_k)$, where $\alpha_1, \ldots, \alpha_k$ are distinct type variables,

(ii) for each $i \in \{1, \ldots, n\}$, $\tau_i$, either is a type variable belonging to $\{\alpha_1, \ldots, \alpha_k\}$ or has the form $G(\beta_1, \ldots, \beta_k)$, where $\{\beta_1, \ldots, \beta_k\} = \{\alpha_1, \ldots, \alpha_k\}$,

(iii) if $\tau$ is atomic, then $n = 0$ (i.e. the left hand side is empty).

If a signature has the arity 0 then we write $\tau$ instead of $(\rightarrow \tau)$.

We assign to each term constructor $f \in \Sigma$ of the arity $n$ a signature of the arity $n$. We write $f : s$ if the signature $s$ is assigned to $f$.

Conditions (ii) and (iii) of Definition 24 are necessary to proving the Pruning Theorem, and Theorem 3. These conditions may seem somewhat restrictively (especially (iii) which prevents signatures like $+ : \text{int} \ast \text{int} \rightarrow \text{int}$), nevertheless these limitations turn out to be not serious when considered together with subtyping scheme of our type system (see Examples 1 and 2).

We assume that there is a predefined ordering $\leq_S$ of monomorphic types such that

(i) if $\tau \leq_S \sigma$, and $\tau \neq \sigma$ then $\tau$ is an atomic type, and $\sigma$ is a monomorphic type (i.e. type without variables),

(ii) if $\tau \leq_S F(\sigma_1, \ldots, \sigma_n)$ and $\tau \leq_S F(\sigma'_1, \ldots, \sigma'_n)$, then, for all $i \in \{1, \ldots, n\}$, $\sigma_i = \sigma'_i$.

B. AXIOMS AND RULES

In Figure 4 we present equality axioms of our system. Axioms (Ax1) - (Ax5) allow us to prove all standard properties of equality, like reflexivity, symmetry and transitivity. Axioms (Ax6) - (Ax9) are well known universal properties of a distributive lattice. Axioms (Ax20) - (Ax22) are specific to our system.

In Figure 5 we present term typing rules. Most of them are well-known from other type systems. These rules allow us to prove facts of the term $t : \tau$ (the term $t$ has the type $\tau$). In these rules $\theta$ denotes a type substitution.

C. CORRECTNESS OF THE SYSTEM

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

from the set of monomorphic types into the powerset of $H$.

Let $\Delta$ be the set of functions from types into the powerset of $H$, i.e. $\Delta = \{\delta \mid \delta : T \rightarrow 2^T\}$. We define a partial order $\leq$ on $\Delta$. Let $\delta, \delta' \in \Delta$. Then $\delta \leq \delta'$ if for all $\tau \in T$. $\delta(\tau) \subseteq \delta'(\tau)$.

Definition 25. Let $[\cdot] \in \Delta$ be the least function such that

(i) $[\text{true}] = \emptyset$, and $[\text{false}] = H$,

(ii) $[\tau \cup \sigma] = [\tau] \cup [\sigma]$,

(iii) $[\tau \cap \sigma] = [\tau] \cap [\sigma]$,

(iv) if $\tau = F(\tau_1, \ldots, \tau_n)$, then $[\tau] = \{f(t_1, \ldots, t_k) \mid f : \sigma_1 \ast \cdots \ast \sigma_k \rightarrow \sigma_0, $ there exists $\theta$ such that $\sigma_0 \theta \leq_S \tau$, and for each $i \in \{1, \ldots, k\}$ $t_i \in [\sigma_i \theta] \}$

Correctness of this definition is proved in [19].
Figure 5: Term Typing Rules

\[
\begin{align*}
(T_1) & \quad \Gamma \vdash t : \top \\
(T_2) & \quad \Gamma \cup \{X : \tau\} \vdash X : \tau \\
(T_3) & \quad \frac{\Gamma \vdash t : \tau', \ \vdash \tau' \leq \tau}{\Gamma \vdash t : \tau} \\
(T_4) & \quad \frac{\Gamma \vdash t : \tau_1, \ldots, \Gamma \vdash t : \tau_n}{\Gamma \vdash t : \tau_1 \cap \cdots \cap \tau_n} \\
(T_5) & \quad \frac{\Gamma \vdash t_i : \tau_i \theta \ (1 \leq i \leq n) \quad \text{if } f : \tau_1 \ast \cdots \ast \tau_n \rightarrow \tau}{\Gamma \vdash f(t_1, \ldots, t_n) : \tau \theta}
\end{align*}
\]

Definition 26. \(\models t : \tau\) if and only if, for any ground term substitution \(\zeta\), \(t \zeta \in [\tau]\).

Theorem 5 (Soundness Theorem). 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 \(\models t : \tau \theta\), and \(p(t)\) evaluates successfully with computed answer substitution\(^5\) \(\eta\), then \(\models t \eta : \sigma \theta\).

\(^5\)See [2].