Type Inference for CLP($\mathcal{FD}$)

Paweł Pietrzak*
Department of Computer and Information Science
Linköping University
S-581 83 Linköping, Sweden
pawpi@ida.liu.se

September 15, 1997

Abstract

This paper describes a type system for CLP($\mathcal{FD}$) where the CHIP language is considered. The proposed type system is an extension of regular types known in LP. We propose an approach where a type for a finite domain variable is a its domain, i.e. a set of all values it can take. These finite domain types can occur as leaves of terms denoting regular types.

Then type inference is discussed. We propose to adopt and modify some techniques developed originally in the context of LP. The described system performs compile-time constraint propagation what allows to deduce about possible domains of some variables.

1 Introduction

The paper describes a type system for CLP($\mathcal{FD}$) where the CHIP language is considered. The proposed type system is an extension of regular types known in LP. We are also focused on type inference system, which is an adaptation of the system originally developed for LP. The applied algorithm is based on a bottom-up fixpoint abstract interpretation.

Most of logic programming languages have no types. This gives a flexibility to programmer but also causes number of problems like difficult debugging or inefficient computations. A number of approaches dealing with types in logic programming have been proposed. They belong to two main perspectives.

One of them is the *prescriptive typing* approach which is based on typed logic languages. A programmer is required to provide type declarations for every function symbol and/or every predicate, wrt a given type system. The type system consists of type definitions which are provided to a type checker and used for verification of well-typedness of a program. Then the type checker eliminates programs having ill-typed statements, what is useful in discovering some errors (e.g. misspelling). Within this framework types are syntactic objects. This approach is a basis of a few programming languages (e.g. TypedProlog [LR91], Gödel [HL94]).

The *descriptive typing* perspective brings us to types as approximations of program semantics (e.g. the least Herbrand model or s-semantics). This makes it very close to abstract

*This work was supported by ESPRIT project DiSCiPl (22532).

interpretation. In fact, abstract interpretation techniques are used in (descriptive) type inferencing (see for instance [BG92, SG95]). Note that a prescriptive type, however being a syntactic object, has the meaning which is a set of values. A recent work of Cousot shows how prescriptive types can be related to abstract interpretation.

Type inference is a process of determining types for arguments of program predicates. This issue was first studied within the functional programming paradigm [Mil78]. The first application of the ML-like type scheme to Prolog has been done by [MO84]. In the context of untyped logic programming (descriptive types) a number of works concerning type inference has been reported (e.g. [Mis84, BG92, SG95]). These are focused on deriving over-approximations of success sets of predicates. Some other proposals are devoted to type inference within prescriptive type systems [LR91, HT92, Bei95]. This is the case where only types for constants and functions symbols are known ("partially typed" programs), but types for predicates have to be reconstructed.

We are focused on inferencing types for the programming language CHIP [Hen89, Cos96] which has no type system. Consequently the types we consider are descriptive types. For a while our interest is restricted to finite domains (and Herbrand domain of terms). We propose to adopt and extend the type inference system described in [SG95, GdW94, GdW92]. In this approach the programmer is not required to declare types for predicates and terms. We introduce some predefined types for most common terms (like lists) and for finite domains. We also impose types for built-in predicates operating on finite domain variables.

Our original motivation is using inferred types during diagnosis process. The programmer can compare obtained type description (which is an over-approximation of a program) with expected (but not declared) types. This can be used to identify an erroneous part of a program. See [BDD$^+$97] for details.

A prototype version of our type inference system has been implemented and it is still under development.

2 Type system

Types are usually treated as a set of all possible values. In logic programming, types are sometimes considered to be a *decidable set of terms closed under substitution* [AM94]. Throughout this paper we assume type as a set of ground terms.

2.1 Basic notions

Let $\mathcal{L}$ be an underlying language with the set of constants C, function symbols F and variables V. The universe of terms constructable from C, F and V is written as $Term(C, F, V)$.

We assume existence of a set $\mathcal{BT}$ of constants including $\perp$ and any, called *base type symbols*. The role of $\perp$ and any will be explained later. Let us introduce a set $\mathcal{TC}$ of *type constructors*, including binary constructor $\cup$. $\mathcal{TV}$ denotes a set of *type variables* or *parameters*. *Type expressions* are elements of $Term(\mathcal{BT}, \mathcal{TC}, \mathcal{TV})$. Given a type expression τ the function $tvars(\tau)$ denotes a set of type variables contained in τ. Type substitutions are defined in the same way as standard substitutions operating on ordinary terms. To distinguish between type symbols and types we introduce a function

$$Type : Term(\mathcal{BT}, \mathcal{TC}, \emptyset) \rightarrow \mathcal{P}(Term(C, F, \emptyset))$$

which for a given ground type expression returns a type denoted by it. A definition of this function depends on a particular type system (see Sect. 2.2). Note that a non-ground type expression describes a kind of generic type or set of types rather than type itself. Thus the function $Type$ is undefined for non-ground type expressions.

Given two type expressions τ_1 and τ_2 the term $\tau_1 \cup \tau_2$ is called their *union* and denotes the smallest type containing both $Type(\tau_1)$ and $Type(\tau_2)$. The union of two types is not always a union of respective sets (due to some properties of types, e.g. tuple-distributivity, Def. 2.1). The type $Type(\perp)$ is called *undefined* type. The symbol any describes a superset of all the types, i.e. $Type(\mathsf{any}) = Term(C, F, \emptyset)$.

In order to have decidable set of terms we need them to be *tuple-distributive*. It has been proven (e.g. [Boy96]) that tuple-distributivity is one of four criteria which constitute a sufficient condition for type decidability[1]. We follow the definition presented in [Mis84, SG95].

Definition 2.1 Let T be a set of ground terms. Let $t \in T$ such that $f(s_1, \ldots, s_n)$ is a subterm of t. Let $u \in T$ be a term identical to t except that the subterm $f(s_1, \ldots, s_n)$ is replaced by $f(r_1, \ldots, r_n)$. We write $u = t[f(r_1, \ldots, r_n)/f(s_1, \ldots, s_n)]$. Then T is **tuple-distributive** if for all such $t, u \in T$,

$$\{t[f(v_1, \ldots, v_n)/f(s_1, \ldots, s_n)] \mid v_i = s_i \textit{ or } v_i = r_i\} \subseteq T$$

If T is a set of term then the smallest tuple-distributive set containing T is called the **tuple-distributive closure** of T. □

Example 1 If $T = \{f(a, b), f(c, d)\}$ then its tuple-distributive closure is

$$\{f(a, b), f(a, d), f(c, b), f(c, d)\}$$

□

2.2 Regular types with polymorphism

There are several papers concerning regular types in the context of logic programming e.g.[Mis84, YS90, DZ92]. *Regular type with polymorphism* is a set of terms specified with a *regular term grammar*.

Definition 2.2 A **regular term grammar** is a tuple $G = (C, F, \mathcal{BT}, \mathcal{TC}, \mathcal{TV}, \mathcal{R})$, where C is a set of constants, F is a set of function symbols, $\mathcal{BT}$, $\mathcal{TC}$ and $\mathcal{TV}$ are sets of basic types, type constructors and type variables respectively, and $\mathcal{R}$ is a set of rules $l \to r$ of the following form

- $l \in (\mathcal{BT} - \{\mathsf{any}\})$ or has the form $t(\alpha_1, \ldots, \alpha_n)$, where $t \in \mathcal{TC}$ and $\alpha_i \in Term(\emptyset, \{\cup\}, \mathcal{TV})$, $1 \leq i \leq n$
- $r \in Term(C \cup \mathcal{BT}, F \cup \mathcal{TC}, \mathcal{TV})$
- $tvars(r) \subseteq tvars(l)$

[1] Others criteria are: decidable subset relation, computable intersection and decidable realizability. See [Boy96] for details.

Given a ground term t over $\mathcal{BT}$ and $\mathcal{TC}$ we define $Def(t)$ a *definition* of t, as the set

$$Def(t) = \{r\theta \mid t \equiv l\theta, l \to r \in \mathcal{R}\}$$

A ground type expression t could be rewritten using rules from R as follows: Select a rule $l \to r$ an a subterm u of t such that $u = l\theta$ and replace u by $r\theta$. Let's call the resulting term t'. The single rewriting step is denoted by $t \to t'$ and transitive closure of $\to$ we write as $\stackrel{*}{\to}$. Now we can define the *Type* function

$$Type(\tau) = \{t \in Term(C, F, \emptyset) \mid \tau \stackrel{*}{\to} t\}$$

□

We allow the unions of type variables (parameters) to occur on the lefthandsides of rules. Regular types defined above are not necessary tuple-distributive (Def. 2.1). To ensure this property we require regular types to be *discriminative.*

Definition 2.3 A regular grammar term is **discriminative** if it is in a *normal form* (i.e. all its rules have the form $l \to f(t_1, \ldots, t_n)$, where $n \geq 0$ and $t_1, \ldots, t_n \in Term(\mathcal{BT}, \mathcal{TC}, \mathcal{TV})$) and for every ground term t over $\mathcal{BT}$ and $\mathcal{TC}$ the top level functors in $Def(t)$ are pairwise distinct. The types generated by discriminative grammars are called **discriminative regular types (with polymorphism)**. □

Example 2 Let us define a homogeneous list structure

$$\mathsf{list}(\perp) \to []$$
$$\mathsf{list}(\alpha) \to [\alpha \mid \mathsf{list}(\alpha)]$$

Type denoting heterogeneous lists could be defined as follows

$$\mathsf{list}(\perp) \to []$$
$$\mathsf{list}(\alpha \cup \beta) \to [\alpha \mid \mathsf{list}(\beta)]$$

□

2.3 Towards types for finite domains

A concept of *finite domains* or simply *domains* was introduced to logic programming by [Hen89]. We will follow this framework.

Some of variables, called *domain variables*, range over domains which are non-empty finite set of constants. Assume a finite set $\mathcal{FD}$ of domains such that $d \in \mathcal{FD}$ implies $e \in \mathcal{FD}$ for each subset e of d. Let DV denote a set of domain variables and let $D = \bigcup d_i$, for all $d_i \in \mathcal{FD}$. The remaining variables range over the set of terms and are called *simple variables.* Note that a simple variable could be bound to a domain variable.

In the CHIP programming language [Hen89, Cos96] domains consists of natural numbers. We assume a set of arithmetic function symbols $A = \{+, -, *\}$ to be interpreted. Let us impose the functors from A to have only arithmetic terms as arguments. Thus terms occurring in a CHIP program (without rationals and Booleans) come from $Term(C \cup D, F \cup A, V \cup DV)$.

Now we extend a notion of type with respect to finite domains. If a type of a domain variable is a set of its all possible values then the domain could be treated as a type. Assume existence of finite set $\mathcal{FT}$ *domain type symbols.* Each symbol from $\mathcal{FT}$ corresponds to a domain, namely

$$\mathcal{FT} = \{\mathsf{fd}^d \mid d \in \mathcal{FD}\}$$

We have to reintroduce the *Type* function:

$$Type : Term(\mathcal{BT} \cup \mathcal{FT}, \mathcal{TC}, \emptyset) \to \mathcal{P}(Term(C \cup D, F, \emptyset))$$

Types, as previously, are sets of ground terms. Observe that we don't have arithmetic function symbols in types and their respective symbols in the set of type constructors. Because they are interpreted we assign to them types from D and type symbols from $\mathcal{FT}$. Basic operations on domain types like union and intersection on domain are straightforward.

Consider regular term grammars (Def.2.2). Given a rule $l \to r$ we require an additional condition $l \notin \mathcal{FT}$. This kind of types are called *semi-regular types.* Types associated with $\mathcal{FT}$ are pre-defined types and are not defined by a regular term grammar.

However, domain types could be defined as 'pure' regular types. Namely with each domain $d = \{c_1, \ldots, c_n\}$ we could assign a set of n rules $\mathsf{fd}^d \to c_i$, $1 \leq i \leq n$. We don't want to consider domain types syntactically. For the purpose of type synthesis we prefer to look at domain types as at black boxes and use a constraint solver to deduce their properties. Therefore the semi-regular definitions seems to be more relevant.

Example 3 Consider variables with the finite domain {1, 2, 3, 5}, list consisting of them and type denoting this structure

$\mathsf{list}(\perp) \to []$
$\mathsf{list}(\mathsf{fd}^{\{1,2,3,5\}}) \to [\mathsf{fd}^{\{1,2,3,5\}} \mid \mathsf{list}(\mathsf{fd}^{\{1,2,3,5\}})]$

□

3 Regular approximation

The following definitions are quoted from [GdW92, GdW94].

Definition 3.1 A **regular unary clause** is of the form $t_0(f(x_1, \ldots, x_n)) \leftarrow t_1(x_1), \ldots, t_n(x_n)$ $n \geq 0$, where $x_1, \ldots, x_n$ are distinct variables. □

Definition 3.2 A **regular unary logic program (RUL)** is a finite set of regular unary clauses, in which no two different clause heads have a common instance. □

The class of RUL programs was originally introduced in [YS90] to describe regular types.

Example 4 A type of lists of terms `a` and `b` can be expressed in a RUL program as follows:

```
list([]).
list([X|Xs]):-
      elem(X),
      list(Xs).
elem(a).
elem(b).
```

□

Note that the condition of Def.3.2 telling that no pair of clause heads are unifiable corresponds to the concept of discriminative regular types (Def.2.3), and ensures types to be tuple-distributive.

Definition 3.3 Let P be a definite program and P' a RUL program containing a special unary predicate approx/1. Let M(P) denote the least Herbrand model of P. Then P' is a **regular approximation** of P if the least Herbrand model o P is a subset of the set $\{A \mid \mathsf{approx}(A) \in M(P')\}$ □

Example 5 Let P be a program containing `reverse(X,Y)` where `Y` is the reverse of list `X`. The regular approximation of P includes the following clauses

```
approx(reverse(X,Y)):-t1(X), t1(Y).

t1([]).
t1([X|Xs]):-any(X),t1(Xs).
```

□

Predicate `any/1` plays a special role denoting any terms. It strictly corresponds to type with the symbol any. Note that using approx/1 in the clause $\mathsf{approx}(p(x_1,\ldots,x_n)) \leftarrow t_1(x_1),\ldots,t_n(x_n)$ we abuse notation by confusing predicate and function symbols. Each predicate p_i should be replaced by a new function symbol with the same arity. Later we will omit this by writing the above clause as $p(x_1,\ldots,x_n) \leftarrow t_1(x_1),\ldots,t_n(x_n)$.

Computing a regular approximation is a kind of bottom-up abstract interpretation which for a given program gives its over-approximation. The effective algorithm computing a regular approximation has been implemented. Readers interested in details as well as in operations on RUL programs and a comprehensive discussion are referred to [GdW92, GdW94].

4 Type inference

Our method of type inferencing is a modification of the algorithm presented by [SG95]. We extend the method by adding types for finite domains. We assume a set of some pre-defined, built-in types, assigned to certain terms. For instance let lists be denoted by the type $\mathsf{list}(\alpha)$, where α is a parameter describing elements of the list. We also impose types for arithmetic functions to be finite domain types. Built-in predicates have determined types too, e.g. CHIP's predicates like `::/2`, `#=/2`, `#</2`, ... operate on domain variables (or integers).

To illustrate our method let us consider the following simple example and continue it through next subsections.

Example 6 This is a small program which computes a sum of elements of a given list.

```
sum([],0).
sum([X|Xs],Sum):-
      Sum::0..100,
      sum(Xs,Sum1),
      Sum #= Sum1 + X.
```

The declaration of a domain for the `Sum` variable is required by the CHIP system. The elements of the list may be integer values as well as domain variables. □

4.1 Propagation

In this stage of the analysis the system performs propagation of finite domains in a scope of a single clause. This allows to possibly narrow domains of variables declared within the considered clause. During the analysis, variables not declared in the clause and occurring in constraints have no domains attached. Before the compile-time propagation, the maximum domain available in CHIP (i.e. `0..100000`) is assumed for the undeclared variables. Then these domains are narrowed by the propagation which involves others, possibly declared, variables.

This step is done simply by executing constraint predicates. It is an application of the *looking ahead inference rule* (LAIR) restricted to one clause. Soundness of LAIR has been proven by [Hen89] but its completeness doesn't hold. Therefore, it is clear that propagation used in the analysis gives a safe over-approximation of the values appearing in a solution.

Example 7 Consider the following clause:
`p(X,Y,Z):-[X,Y,Z]::1..10, X #> Y + 3, Y #> Z + 2.` [2]
After executing them, the initial domains of `X`, `Y` and `Z` are reduced and types for `p/3` are

$$\mathsf{p}(\mathsf{fd}^{\{8,9,10\}},\mathsf{fd}^{\{4,5,6\}},\mathsf{fd}^{\{1,2,3\}}).$$

□

During this phase of analysis constraints which have been executed are removed from the program and are not considered further. We still have domain variables in the analyzed program and we need some method of describing their types. We deal with this problem describing finite domain type by a list containing all its values. If the domain size is too big (the notion 'too big' is given by the user) then we denote the type by an interval. We can loose information about gaps in the domain.

Example 8 After this phase of analysis our program (Ex. 6) has the following form.

```
sum([],0).
sum([fd(int(0,100))|Xs],fd(int(0,100))):-
      sum(Xs,fd(int(0,100))).
```

We use the term `fd(Dom)` to denote a finite domain type for the domain `Dom`. If the domain is described by an interval we denote it by the term `int(Lower,Upper)`, where `Lower` and `Upper` are obviously lower and upper bound of a given domain. Note that by the propagation we could infer a maximum domain for members of the list, even so their domain is not explicitly declared in the definition of `sum/2`. □

4.2 Type reduction

Having types (specified by a discriminative regular term grammar) assigned to (some of) function symbols and (some of) constants in program we want to determine types of terms (or at least some of their subterms). Consider the definition of a regular grammar (Def. 2.2). We extend it by additional rules:

[2] In the CHIP language the predicate `Vars::FDom` says that variables in `Vars` (`Vars` is a single variable or a list of variables) range over the domain `FDom`.

$\alpha \to \alpha \cup \alpha$

$\alpha \to \alpha \cup \bot$

$\alpha \to \bot \cup \alpha$

These rules obviously abuse the discriminativity condition, but let's treat them separately, as special rules.

Given the term s, $s \in Term(C, F, V)$, and the grammar G we rewrite s as follows:

1. Replace every domain variable v in s by a symbol fd^d, where d is a domain of v.
2. Select a rule $l \to r$ and a subterm u in s such that u (u is not a variable) matches r, where $mgu(u, r) = \theta$.
3. Reduce s by replacing the occurrence of u by l and applying θ to the result.
4. Go to 1 unless no more reduction is possible.

As stated in [SG95] the final result of reduction doesn't depend on the order of reduction. A given program is called *type-reduced* if all the terms in its all clauses have been rewritten (with respect to applicable rules).

Example 9 Assume the following grammar rules

(1) $\mathsf{list}(\bot) \to$ `[]`

(2) $\mathsf{list}(\alpha \cup \beta) \to$ `[`α `|` $\mathsf{list}(\beta)$`]`

(3) $\mathsf{char} \to$ `a`

and a term to reduce `[a|L]`. The first applied rule is (3), the occurrence of `a` is replaced by char and we get `[char| L]`. Then we apply the rule (2) and obtain the term $\mathsf{list}(\mathsf{char} \cup \beta)$. No further reduction is possible. □

Example 10 We consider the grammar which defines a type for heterogeneous lists (see Ex.2). Let us continue the example 8. After type reduction the program becomes

```
sum(list(bottom),0).
sum(list(Z1),fd(int(0,100))):-
      union(fd(int(0,100)),X1,Z1),
      sum(list(X1),fd(int(0,100))).
```

The predicate `union/3` describes a union ($\cup$) of types. □

4.3 Regular approximation of type-reduced programs

Consider the final stage of analysis. Its purpose is to compute a regular approximation of a program which has been type-reduced during steps shown in 4.1 and 4.2. The regular approximation procedure, described in [GdW94] is a bottom-up fixpoint algorithm. It computes a regular description of each clause body with respect to the approximation inferred in the previous iteration; then the regular approximation is the tuple-distributive upper bound of the approximation obtained for the clause heads.

We modified slightly some steps of the regular approximation procedure. Operations of computing upper bound and intersection have been changed. Namely, whenever considered RUL predicates describe a finite domain type then their upper bound (resp. intersection) is a sum (resp. intersection) of appropriate finite domains.

If the type union symbol ($\cup$) occurs in the type-reduced program, we use the *Tuple-Distributive Union Type assumption.* It is related to tuple-distributivity and says that we can't distinguish between union of types and a set of particular types at the same argument position. For example, types $\{\mathsf{list}(\mathsf{char}), \mathsf{list}(\mathsf{fd}^{\{1,2\}})\}$ and $\{\mathsf{list}(\mathsf{char} \cup \mathsf{fd}^{\{1,2\}})\}$ are equivalent.

Example 11 Consider the program shown in the Ex. 10. Finally, after applying the regular approximation procedure we obtain the following RUL program:

```
sum(X1,X2):-t18(X1),t21(X2).
t18(list(X1)):-t16(X1).
t21(fd(int(0,100))).
t16(fd(int(0,100))).
t16(bottom):-true.
```

The above program clearly describes types for predicate `sum/2`. □

5 Conclusions

We have described the type inference method for CLP($\mathcal{FD}$) programs. The algorithm is an extension of the method given by [SG95].

In typical CHIP programs (with finite domains) propagation occurs after some steps of labeling. Therefore performing the compile-time propagation, like our technique does, may give sometimes no information. However we believe that this kind of analysis could be profitable during debugging. Our experiments have shown that the method is particularly useful in the case of incompleteness errors (missing answer).

For some applications it could be sufficient to have type saying that a variable is a domain variable, without information about domain. In this case the analysis could be more efficient. Another problem appears in some recursively defined predicates like `sorted/1`

```
sorted([]).
sorted([_]).
sorted([A,B|T]):-
      A #<= B,
      sorted([B|T]).
```

The algorithm returns approximations/types for predicates rather than for clause heads. Consequently it computes an upper bound of types coming from different clauses. In the above example the system is not able to determine that the argument of `sorted/1` is a list of finite domain variables and/or values. The result will be `list(any)`. In fact the call `?-sorted([a])` succeeds.

Our ongoing work concerns developing and evaluating application of the system to non-toy programs. Further research can include utilizing our analysis in debugging and program optimizing. Another direction may concern types for another CHIP solver i.e. for rationals numbers.

References

[AM94] K. Apt and E. Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 3, 1994.

[BDD+97] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Małuszyński, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In M. Kamkar, editor, *Proc. of the AADEBUG'97 (The Third International Workshop on Automated Debugging)*, pages 155–169. Linköping University, 1997.

[Bei95] C. Beierle. Type inferencing for polymorphic order-sorted logic programs. In L. Sterling, editor, *Logic Programming – Proc. of the 12th International Conference*, pages 765–779. MIT Press, 1995.

[BG92] R. Barbuti and R. Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Science of Computer Programming*, 19(3):281–313, 1992.

[Boy96] J. Boye. *Directional Types in Logic Programming.* Linköping studies in science and technology, dissertation no. 437, Linköping University, 1996.

[Cos96] Cosytec SA. *CHIP System Documentation*, 1996.

[DZ92] P. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

[GdW92] J. Gallagher and D.A. de Waal. Regular Approximations of Logic Programs and Their Uses. Technical Report CSTR-92-06, Department of Computer Science, University of Bristol, 1992.

[GdW94] J. Gallagher and D.A. de Waal. Fast and Precise Regular Approximations of Logic Programs. In P. Van Hentenryck, editor, *Proc. of the Eleventh International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

[Hen89] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[HL94] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.

[HT92] P.M. Hill and R.W. Topor. A semantics for typed logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 1–62. MIT Press, 1992.

[LR91] T.K. Lakshman and U.S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In V. Saraswat and K. Ueda, editors, *Proc. of the 8th International Logic Programming Symposium*, pages 202–217. MIT Press, 1991.

[Mil78] R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Mis84] P. Mishra. Towards a theory of types in prolog. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 289–298, 1984.

[MO84] A. Mycroft and R. O'Keefe. A polymorphic type sytem for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[SG95] H. Saglam and J. Gallagher. Approximating Constraint Logic Programs Using Polymorphic Types and Regular Descriptions. Technical Report CSTR-95-017, Department of Computer Science, University of Bristol, 1995.

[YS90] E. Yardeni and E.Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–154, 1990.