In *Types in Logic Programming* (F. Pfenning, ed.), pp. 91–140,
MIT Press, 1992

Chapter 3

Logic Programming with Type Specifications

by Michael Hanus[1]

In this chapter, we propose a framework for logic programming with different type systems. In this framework a typed logic program consists of a type specification and a Horn clause program which is well-typed with respect to the type specification. The type specification defines all types which can be used in the logic program. Relations between types are expressed by equations on the level of types. This permits the specification of many-sorted, order-sorted, polymorphic and polymorphically order-sorted type systems.

We present the declarative semantics of our framework and two proof procedures (deduction and resolution) for typed logic programs. An interesting application is a type system that combines parametric polymorphism with order-sorted typing and permits higher-order logic programming. Moreover, our framework sheds some new light on the rôle of types in logic programming.

3.1 Overview and Examples

The absence of types in logic programming languages is a disadvantage for the development of large software systems. It have been also argued that logic programs often make implicit assumptions about types and a logic program only satisfies the

[1] An extended abstract of a previous version of this chapter has appeared in the Proceedings of the Second International Conference on Algebraic and Logic Programming, Nancy, France, October 1990, Springer Lecture Notes in Computer Science 463, 1990.

intended meaning if type information is added to the program [Nai87]. Therefore much research has been carried out in order to integrate types into logic programming languages. The proposed integrations can be classified into two groups: inference-based and declaration-based approaches.

The *inference-based approaches* try to compute a superset of the success set of the program. If this superset is empty (for some goal), then the goal cannot succeed which is usually a hint for a type error in the program. Examples for inference-based approaches can be found in [Mis84] [Zob87] [XW88b] [BG89] (among others). The computation of the superset of the success set is guided by term patterns representing sets of terms. The term patterns are considered as *types*, i.e., types are interpreted as sets of (ground) terms. The advantage of the inference-based approach is simplicity for the programmer since he need not declare any types: The type inference system deduces type information from an untyped logic program. This information can be used by the compiler to perform optimizations in the target program [GZ86].

But there are several problems with the inference-based approach: First, the semantics of types is only based on Herbrand interpretations, i.e., types are viewed as sets of ground terms. But Herbrand models are not sufficient for characterizing the declarative semantics of a logic program (e.g., if `a` and `b` are the only ground terms in a program and the program consists of the facts `p(a)` and `p(b)`, then $\forall$`X p(X)` is true in all Herbrand models but not a logical consequence of the program [Llo87]). In order to give types a *declarative semantics*, types must have a meaning in all interpretations and not only in Herbrand interpretations, similarly to function and predicate symbols. Therefore Barbuti and Giacobazzi [BG89] use term interpretations with variables as the semantic foundation of their type inference system.

The main problem of inference-based approaches is that the inference of types from a completely untyped program yields only in a few cases the types expected by the programmer. For instance, assume *list* denotes the set of all terms of the form `[]` or `[E|L]` where `L` is a term from *list*. Then the inferred type for the predicate `append` defined by

```
append([],L,L) ←
append([E|R],L,[E|RL]) ← append(R,L,RL)
```

may be "$list \times \alpha \times \alpha \cup list \times \beta \times list$" [XW88a], where α and β denote arbitrary types. But the type expected by the programmer is "$list \times list \times list$" since `append` should be only used to concatenate lists. The problem in this example is the first clause which defines `append` to be true not only for lists but also for other terms. E.g., `append([],2,2)` is true but usually considered as an ill-typed goal. In order to obtain the expected type "$list \times list \times list$", `append` must be defined by

```
append([],[],[]) ←
```

```
append([],[E|R],[E|R]) ← append([],R,R)
append([E|R],L,[E|RL]) ← append(R,L,RL)
```

(the exact definition depends on the type inference system).

Another problem of inference-based approaches is the strong dependence from the syntactic form of the clauses: A type inference system may deduce different types for two declaratively equivalent programs if the clauses are syntactically different. For instance, assume the type system allows polymorphic data structures [XW88b] and $list(\alpha)$ denotes the set of all terms of the form `[]` or `[E|L]` where `E` and `L` are terms from α and $list(\alpha)$, respectively, and α is an arbitrary type. Then the type inferred for the predicate `member` defined by

```
member(E,[E|L]) ←
member(E,[F|L]) ← member(E,L)
```

is "$\alpha \times list(\alpha)$", i.e., `member` can be used on lists of arbitrary types. The literal `member(2,[1,2,3])` is a logical consequence of the clauses for `member` (we assume that the natural numbers are always contained in our programs). Hence we can add this literal as a new fact and obtain the declaratively equivalent program

```
member(2,[1,2,3]) ←
member(E,[E|L]) ←
member(E,[F|L]) ← member(E,L)
```

The type inferred by an ML-based inference system [DM82] is "$nat \times list(nat)$" since almost all polymorphic type systems for logic programming require that the left-hand sides of all clauses for a predicate must have equivalent types [MO84] [DH88] [Smo89].

These examples show that in many cases the inference of types from a completely untyped program does not yield sufficient results since an untyped logic program does not contain the type information which has the programmer in mind (see also [Nai87]). A type system should *allow user declarations for types*. These declarations are not a burden on the programmer but documents the expected meaning of predicates and improves the readability of large programs. Another advantage of extending logic programs by type declarations is the possibility to give types a true declarative meaning, i.e., types can be interpreted as subsets of the carrier sets in all interpretations. This will be done in our approach.

The important question which has to be answered by a *declaration-based type system* is: Which kind of type structures can be specified? Several answers have been given in the literature: The type system of Turbo-Prolog is comparable to *many-sorted* Horn logic [Pad88] and many-sorted logic programs can be executed with the same efficiency as untyped logic programs, but this type system is too restricted for a lot of applications [Han87]. A more flexible type system motivated from ML was proposed by Mycroft and O'Keefe [MO84]. It offers *parametric poly-*

morphism [DM82], needs no type-checking at run time and enables the writing of compact and reusable programs. The restrictions of Mycroft/O'Keefe's type system have been dropped in [Han89a]: The result is a type system which allows the application of higher-order programming techniques. In general it is necessary to consider the types at run time, but it has been shown that for Prolog-like applications of higher-order programming all type information can be omitted at run time [Han89b]. This type system can also be applied to a language that combines functional and logic programming [Han90]. Another direction for typing logic programs are *order-sorted* type systems where different types may be related by an inclusion relation [SNGM89]. Such inclusion relations occur in Prolog (for instance, the set of all constants is the union of the set of numbers and the set of atoms) and therefore a lot of inference-based type systems offer inclusion polymorphism ([Mis84] [Zob87] [XW88b] among others). In order-sorted logic programming [HV87] types are present at run time, but the type information can be used to avoid unnecessary computations and reduce the search space [SS85] [HV87]. Smolka [Smo89] has proposed the combination of parametric polymorphism and order-sorted typing for a logic programming language. There are several restrictions in his type system so that higher-order programming techniques cannot be used. One particular instance of the framework proposed in this chapter is a type system that combines parametric polymorphism with order-sorted typing and allows the application of higher-order logic programming techniques.

Type systems with parametric polymorphism have been extensively studied in the context of functional programming languages [DM82] [CW85]. Therefore several proposals for polymorphic type systems for logic programming are based on these ideas [MO84] [DH88] [Smo89]. But we think that logic programming languages need other type systems than functional programming languages because:

1. The data flow is not fixed in logic programs since there are no "input" and "output" parameters in contrast to functional programs.

2. In functional languages a unary function `f` is defined by an equation of the form

   ```
   f(A) = E
   ```

   (multiple equations for different argument patterns can be seen as syntactic sugar). There is no doubt about the type of `f`: The argument type is the most general type of `A` and the result type is the most general type of `E`. But in logic languages the semantics of a predicate is defined by several independent clauses that should be satisfied by any model for the predicate, i.e., a logic program is a specification of the predicate's properties. If a unary predicate `p` is defined by n clauses which characterizes different properties of `p`, i.e., `p(`A_1`)` $\leftarrow \cdots$

...

`p(`A_n`)` $\leftarrow \cdots$

then the type of `p` is unclear if the arguments $A_1, \ldots, A_n$ have different types. Since the type system in [MO84] is influenced from the ML system, Mycroft and O'Keefe require the argument types in different clause heads to be equivalent (equal up to type variable renaming). But this restriction prevents a useful logic programming technique: Optimization of the resolution process by *lemma generation*. In untyped logic programming it is possible to add a new fact L to a program without changing the program semantics if L is a logical consequence of the program. The new fact L can be used to obtain shorter proofs for subsequent goals that include L. For instance, the literal `append([1,2],[3,4],[1,2,3,4])` is a logical consequence of the program

```
append([],L,L) ←
append([E|R],L,[E|RL]) ← append(R,L,RL)
```

and therefore it may be added at the beginning of the program. If `append` has type "$list(\alpha), list(\alpha), list(\alpha)$", then the new fact is ill-typed w.r.t. Mycroft/O'Keefe's type system. From a declarative point of view there is no reason to forbid such specialized clauses. Therefore our language allows such clauses since any instance of the declared predicate type is allowed in the left-hand side of the clause.

Summarizing our discussion of various type systems for logic programming we think that declaration-based type systems are adequate for logic programming because in these type systems the types of functions and predicates are independent of the syntactic form of the clauses and it is possible to give types a pure declarative meaning. Since typing all variables, functions and predicates in a logic program can be tedious, it should be allowed to omit some of the type declarations in the program, but such a program is viewed as a short-hand for a fully typed program. This point of view simplifies the semantics of the language since only well-typed expressions must have a meaning (see [MH88] for a more detailed discussion in the context of ML). In some cases a type inference procedure can be used to insert the omitted type declarations (the existence of such inference procedures depends on the restrictions of the type system). For instance, in ML [HMM86] the programmer has to declare the argument and result types of data type constructors. The types of all variables and functions in an ML program are inferred by a type inference procedure [DM82].

A further requirement to a type system for logic programming is flexibility: In logic programming it is possible to define one predicate which can be applied to arguments of different types (e.g., `append` can be applied to lists where the elements have an arbitrary type). Therefore a type system should support some sort of *polymorphism*, i.e., a predicate may have several types. Furthermore, the type system should also support logic programming techniques like lemma generation and the use of higher-order predicates.

This chapter proposes a framework for such flexible type systems. We present a general mechanism for the specification of type systems where particular instances of this framework are order-sorted, polymorphic or polymorphically order-sorted type systems. Our proposal generalizes previous approaches since it allows the application of typical logic programming techniques, i.e., it is influenced but more general than type systems for functional languages. Since all predicates, functions, variables and clauses are explicitly typed in our approach, the well-typedness of a program is decidable. For practical applications it should be allowed to omit some of the type declarations in the program which should be automatically inserted by a type inference procedure. But such procedures are only known for particular instances of our general framework. The development of more powerful type inference procedures and necessary restrictions to the programs is a topic for further research.

The general idea of our framework is to divide typed logic programs into two parts: a specification of the type structure and a well-typed logic program. Since the second part depends on the first part, we may view it as a two-level approach. In the context of algebraic specifications, Poigné [Poi86] has proposed a two-level approach for algebraic specifications with higher-types. Each level consists of an equational specification where the first-level describes a type structure and the second level is an equational specification with sort expressions from the first level. While he has used the approach for the specification of the typed λ-calculus, we will use a similar approach for our framework for typed logic programming. In our two-level approach the first level consists of a specification of a type structure for the logic program and contains all types which will be used inside the logic program and some relations between types specified by equational axioms. Hence the first level is a many-sorted equational specification [EM85] and we can use results from this area for our purposes. The second level is based on the specified type structure and consists of a specification of the types of all variables, constants, functions, and predicates occurring in the logic program and a set of Horn clauses which must be well-typed with respect to the type specification. The operational semantics, which is resolution with a unification procedure on well-typed terms, ensures that type errors do not occur while executing well-typed programs. We give some examples to show the basic ideas.

Example 3.1 *Parametric polymorphism* is used for defining universal data structures which can be applied to different concrete types. A classical example are polymorphic lists which can be applied to integers giving lists of integers, to Booleans giving lists of Booleans, etc. The following signature specifies a type structure for a program which uses the basic types of integers and Booleans and the polymorphic types of lists and pairs of elements:

`TYPEOPS`	*int*:		$\rightarrow$	*type*
	bool:		$\rightarrow$	*type*
	list:	*type*	$\rightarrow$	*type*
	pair:	*type*, *type*	$\rightarrow$	*type*

This type structure has only a single sort *type*. Hence all types can be used as arguments for the polymorphic type constructors *list* and *pair*. The set of all types specified by this signature is the set of all well-formed terms which may contain some *type variables*. For instance, types w.r.t. the above specification are

$$int \qquad bool \qquad list(int) \qquad list(\alpha) \qquad pair(bool,\beta) \qquad pair(\alpha, list(\alpha)) \qquad \cdots$$

where α and β are type variables. A typed logic program consists of type declarations for variables, functions and predicates (constants are functions without arguments) and a set of well-typed Horn clauses. The following program defines two polymorphic predicates on lists (throughout this chapter we use the Prolog notation for lists [CM87]):

func `[]`: $\rightarrow$ $list(\alpha)$
func `[..|..]`: α, $list(\alpha)$ $\rightarrow$ $list(\alpha)$

pred `append`: $list(\alpha)$, $list(\alpha)$, $list(\alpha)$
pred `member`: α, $list(\alpha)$

vars `L, R, RL`:$list(\alpha)$, `E, E1`:α

```
append([],L,L) ←
append([E|R],L,[E|RL]) ← append(R,L,RL)

member(E,[E|R]) ←
member(E,[E1|R]) ← member(E,R)
```

The clauses for `append` and `member` are well-typed in our sense (cf. Section 3.2) w.r.t. the type definitions.

We view *subtyping* as the possibility of applying a function or predicate to all types which are subtypes of the declared type of the function or predicate. Hence we specify a type that has some subtypes as a function which is the identity on the subtypes. This will be illustrated by the next example.

Example 3.2 We want to specify a type structure with types *nat*, *zero* and *posint* where *zero* and *posint* are subtypes of *nat*. Hence we specify *nat* as a function on types which is the identity on *zero* and *posint*:

$$
\begin{array}{lllcl}
\texttt{TYPEOPS} & \mathit{zero}: & & \rightarrow & \mathit{type} \\
 & \mathit{posint}: & & \rightarrow & \mathit{type} \\
 & \mathit{nat}: & \mathit{type} & \rightarrow & \mathit{type} \\
\texttt{TYPEAXIOMS} & \mathit{nat}(\mathit{zero}) & = & \mathit{zero} & \\
 & \mathit{nat}(\mathit{posint}) & = & \mathit{posint} &
\end{array}
$$

The type axioms state that *nat* is not a free type constructor like *list* but is the identity on the subtypes of *nat*. It is possible to apply *nat* to other types than *zero* and *posint*, but our logic programs which are based on this specification do not contain any ground terms of type $nat(\tau)$ where $\tau \notin \{zero, posint\}$. Therefore the type $nat(\alpha)$ describes the union of *zero* and *posint* in the initial model of the following program:

func `0:` $\rightarrow$ *zero*
func `s:` $nat(\alpha) \rightarrow posint$

pred `plus:` $nat(\alpha)$, $nat(\beta)$, $nat(\gamma)$

vars `N, N1:`$nat(\alpha)$`, N2:`$nat(\beta)$`, N3:`$nat(\gamma)$

`plus(0,N,N)` $\leftarrow$
`plus(s(N1),N2,s(N3))` $\leftarrow$ `plus(N1,N2,N3)`

The clauses for `plus` are well-typed in our sense (cf. Section 3.2) w.r.t. the type definitions (note that the type of the first argument of the clause head is "*zero*" in the first and "*posint*" in the second clause). Since the argument types of `plus` are defined to be arbitrary naturals, we can apply `plus` with an arbitrary subtype of the naturals. It is possible to build nonsensical types like *nat*(*bool*) (if the basic type *bool* is added to the type structure), but our program contains no ground term of this type and therefore such a type denotes an empty set in the initial model of this program. Moreover, our proof procedure (resolution with typed unifiers, cf. Section 3.6) ensures that such types do not occur in the computation if they are not present in the initial goal.

Since order-sorted type structures are polymorphic type specifications with equational axioms which describe the subsort relationship, it is clear that there is no problem in the combination of polymorphic and order-sorted type structures in our framework. It is also possible to express subsort relationships between polymorphic types:

Example 3.3 We want to specifiy a type structure for polymorphic lists so that the polymorphic type *list* is the union of *elist* (empty lists) and *nelist* (non-empty lists). Therefore we have to express the subtype relationships $elist < list(\alpha)$ and $nelist(\alpha) < list(\alpha)$. As in the previous example, we add an additional argument to a type constructor having some subtypes and express the subtype relationship by type equations:

TYPEOPS	$elist$:		$\rightarrow$	$type$
	$nelist$:	$type$	$\rightarrow$	$type$
	$list$:	$type,\ type$	$\rightarrow$	$type$
TYPEAXIOMS	$list(\alpha, elist)$		$=$	$elist$
	$list(\alpha, nelist(\alpha))$		$=$	$nelist(\alpha)$

The `append`-program is specified w.r.t. this type structure as follows:

func `[]`: $\rightarrow\ elist$
func `[..|..]`: $\alpha,\ list(\alpha,\beta)\ \rightarrow\ nelist(\alpha)$

pred `append`: $list(\alpha,\beta_1),\ list(\alpha,\beta_2),\ list(\alpha,\beta_3)$

vars `R`:$list(\alpha,\beta_1)$, `L`:$list(\alpha,\beta_2)$, `RL`:$list(\alpha,\beta_3)$, `E`:α

```
append([],L,L) ←
append([E|R],L,[E|RL]) ← append(R,L,RL)
```

The type variable α in all argument types of `append` expresses that `append` concatenates lists of the same element type, whereas the different type variables $\beta_1, \beta_2, \beta_3$ show that an arbitrary subtype of an α-list (empty or non-empty list) can be used in each argument.

The example shows that logic programs with a polymorphically order-sorted type structure are allowed in our framework. Moreover, in Section 3.7 we will give an example of a logic program with higher-order predicates which is well-typed in our framework.

Example 3.4 The type specifications in the previous examples are single-sorted specifications with only one sort "*type*". Since we also allow many-sorted specifications, this feature can be used to restrict the quantification of type variables. For instance, the following type specification may be part of a program for symbolic computations:

TYPEOPS	int:		$\rightarrow$	$ring$
	$polynom$:	$ring$	$\rightarrow$	alg_type

Thus *polynom* is a type constructor where the argument is restricted to be a ring. The type *polynom*(*int*) describes the polynomials with integer coefficients. The type declaration for a generic predicate denoting the addition of two polynomials is

$$\textbf{pred}\ \texttt{poly\_add:}\quad polynom(\rho),\ polynom(\rho),\ polynom(\rho)$$

Because of the particular type structure, the type variable ρ is not quantified over all possible types but only over types of sort "*ring*", e.g., *int* is a valid instance of ρ.

The example shows that the sort of a type can be used to express a *property of a type*. It may be also desirable to use order-sorted equational specifications for the type structure which allows us to express dependencies between type properties, e.g., "$alg\_type < type$" (an algebraic type is also a general type). Though this is a useful feature for computer algebra systems (as pointed out in [SLC88]), we omit it for the sake of simplicity. But we emphasize that the restriction to many-sorted type specifications will not be used in the proofs of our results.

In the following we present our framework for typed logic programming in detail. The main topics of this chapter are:

- We present a two-level approach to typed logic programming: The first level is a specification of the basic type structure, and the second level contains a well-typed logic program which is based on the specified type structure. The type structure is specified by a many-sorted signature with equational axioms. In contrast to other approaches to polymorphic type systems for logic programming, we do not restrict the use of types inside program clauses.

- Our approach to typed logic programming is declarative: In contrast to many other type systems for logic programming where types are viewed as sets of ground terms (i.e., they are only valid in the initial model), we define define the semantics of types in a model-theoretic way, i.e., types are subsets of the carrier sets in all interpretations.

- We present sound and complete deduction and resolution methods for typed logic programs. For the soundness of the resolution method it is necessary to define the unification procedure on well-typed terms which is based on a unification procedure for the equational type theory. This sheds some new light on the rôle of types in logic programming since the complexity of the type structure directly influences the complexity of the unification procedure. A powerful type structure (e.g., polymorphic types combined with subtypes) implies a complex unification procedure.

- We show that higher-order programming techniques can be applied in our general framework. We give an example of a typed logic program with higher-order predicates which is ill-typed in the sense of other polymorphic type systems for logic programming.

- The presented approach is a framework for the definition of different type structures for logic programs. The type structure influences only the unification procedure for the execution of the program. Therefore different type structures can be used for different applications where the specification of the type structure can be compiled into a specific unification procedure. It is not necessary to use a powerful order-sorted unification procedure for simple applications like those possible in Turbo-Prolog.

This chapter is organized as follows. In the next section the basic notions and the syntax of typed logic programs are defined. Section 3.3 defines the semantics of typed logic programs which is based on interpretations in algebraic structures. Section 3.4 presents a deduction method for typed logic programs. Section 3.5 presents a solution to the unification problem of typed terms which is based on a given unification procedure for the type theory. The unification procedure on typed terms will be used for the resolution method presented in Section 3.6. Section 3.7 concludes with an interesting application of our framework.

3.2 Logic Programs with Type Specifications

We use many-sorted equational logic for the specification of type structures. Therefore we recall some basic notions from algebraic specifications [GTW78] [EM85]. A many-sorted **signature** Σ is a pair (S, O), where S is a set of **sorts** and O is a family of **operator** sets of the form $O = (O_{w,s} | w \in S^*, s \in S)$. We write $o: s_1, \ldots, s_n \to s \in O$ instead of $o \in O_{(s_1,\ldots,s_n),s}$. An operator of the form $o: \to s$ is also called a **constant** of sort s. A signature $\Sigma = (S, O)$ is interpreted by a **Σ-algebra** $A = (S_A, O_A)$ which consists of an S-sorted domain $S_A = (S_{A,s} | s \in S)$ and an operation $o_A: S_{A,s_1}, \ldots, S_{A,s_n} \to S_{A,s} \in O_A$ for any $o: s_1, \ldots, s_n \to s \in O$. A set of **$\Sigma$-variables** is an S-sorted set $X = (X_s | s \in S)$. The set of **Σ-terms** of sort s with variables from X, denoted $T_{\Sigma,s}(X)$, is inductively defined by $x \in T_{\Sigma,s}(X)$ for all $x \in X_s$, $c \in T_{\Sigma,s}(X)$ for all $c: \to s \in O$, and $o(t_1, \ldots, t_n) \in T_{\Sigma,s}(X)$ for all $o: s_1, \ldots, s_n \to s \in O$ $(n > 0)$ and all $t_i \in T_{\Sigma,s_i}(X)$. Given a term t, $var(t)$ denotes the set of all variables occurring in t. We write $T_\Sigma(X)$ for all Σ-terms with variables from X and T_Σ for the set of **ground terms** $T_\Sigma(\emptyset)$. By $T_\Sigma(X)$ we also denote the term algebra.

A **Σ-equation** is a pair of Σ-terms (t_1, t_2) of the same sort, usually written $t_1 = t_2$. An **equational specification** is a triple $Sp = (S, O, E)$ where $\Sigma = (S, O)$ is a signature and E is a set of Σ-equations. In the following we denote by Sp also the signature (S, O) contained in Sp, e.g., $T_{Sp}(X)$ is the set of (S, O)-terms with variables from X.

A **variable assignment** is a mapping $a: X \to S_A$ with $a(x) \in S_{A,s}$ for all variables $x \in X_s$ (more precisely, it is a family of mappings $(a_s: X_s \to S_{A,s} | s \in S)$). A **$\Sigma$-homomorphism** from a Σ-algebra $A = (S_A, O_A)$ into a Σ-algebra $B = (S_B, O_B)$ is a mapping (family of mappings) $h: S_A \to S_B$ with the property $h_s(o_A(a_1, \ldots, a_n)) = o_B(h_{s_1}(a_1), \ldots, h_{s_n}(a_n))$ for all $o: s_1, \ldots, s_n \to s \in O$ $(n \geq 0)$ and all $a_i \in S_{A,s_i}$. A **Σ-congruence** on a Σ-algebra $A = (S_A, O_A)$ is a family of binary equivalence relations $\sim_s \subseteq S_{A,s} \times S_{A,s}$ $(s \in S)$ so that $o_A(a_1, \ldots, a_n) \sim_s o_A(b_1, \ldots, b_n)$ for all $o: s_1, \ldots, s_n \to s \in O$ $(n > 0)$ and all $a_i, b_i \in S_{A,s_i}$ with $a_i \sim_{s_i} b_i$. The following lemma shows an important property of term algebras:

Lemma 3.5 (Free term algebra) *Let Σ be a signature, $A = (S_A, O_A)$ be a Σ-algebra and $a: X \to S_A$ be an assignment for variables from X. There exists a unique Σ-homomorphism $a^*: T_\Sigma(X) \to S_A$ with $a^*(x) = a(x)$ for all $x \in X$.*

Let $Sp = (S, O, E)$ be an equational specification and $A = (S_A, O_A)$ be an (S, O)-algebra. An Sp-equation $t_1 = t_2$ is **valid** in A, denoted $A \models t_1 = t_2$, if $a^*(t_1) = a^*(t_2)$ holds for all variable assignments $a: var(t_1) \cup var(t_2) \to S_A$. A is a **model** for Sp if every equation from E is valid in A. We write

$$Sp \models t_1 = t_2$$

if $t_1 = t_2$ is valid in all models for Sp. We remark that an initial model for a specification Sp is $T_{Sp}/\equiv_E$, the quotient of the ground term algebra T_{Sp} by the congruence $\equiv_E$ generated by the equations E.

Lemma 3.6 *Let Σ be a signature, $T_{Sp}(X)/\equiv_E$ be the quotient of the term algebra $T_{Sp}(X)$ by the congruence $\equiv_E$ generated by the equations E, $A = (S_A, O_A)$ be a Σ-algebra and $a: X \to S_A$ be an assignment for variables from X. There exists a unique Σ-homomorphism $a^*: T_{Sp}(X)/\equiv_E \to S_A$ with $a^*([x]) = a(x)$ for all $x \in X$ where $[x]$ denotes the equivalence class of x w.r.t. $\equiv_E$.*

The definition of *types* is based on equational specifications: $\mathcal{T} = (Ts, Top, Tax)$ is a **specification of types** if $\mathcal{T}$ is an equational specification. Constants from $\mathcal{T}$ are called **basic types**. By X we denote an infinite set of **type variables** (precisely, $X = (X_s | s \in Ts)$ is a family of infinite sets of type variables, but we identify the family of sets with one set since we assume that the sets X_s are disjoint). A **type expression** or **type** is a term from $T_\mathcal{T}(X)$.

A **type substitution** σ is a $\mathcal{T}$-homomorphism $\sigma: T_\mathcal{T}(X) \to T_\mathcal{T}(X)$. $\boldsymbol{TS(\mathcal{T}, X)}$ denotes the class of all type substitutions. Two types $\tau_1, \tau_2 \in T_\mathcal{T}(X)$ are called **$\mathcal{T}$-equal**, denoted $\tau_1 =_\mathcal{T} \tau_2$, if $\mathcal{T} \models \tau_1 = \tau_2$.

A **polymorphic signature $\boldsymbol{\Sigma}$** for logic programs is a triple $(\boldsymbol{\mathcal{T}, Func, Pred})$ with:

- $\mathcal{T}$ is a specification of types with $T_{\mathcal{T},s}(\emptyset) \neq \emptyset$ for all $s \in Ts$.

- $Func$ is a set of **function declarations** of the form $f{:}\tau_1, \ldots, \tau_n \to \tau$ with $\tau_i, \tau \in T_{\mathcal{T}}(X)$, $n \geq 0$.

- $Pred$ is a set of **predicate declarations** of the form $p{:}\tau_1, \ldots, \tau_n$ with $\tau_i \in T_{\mathcal{T}}(X)$ $(n \geq 0)$.

Since we do not deal with the problem of type checking or type inference in our framework, we do not forbid *overloading* in contrast to [Han89a] or [Smo89]. The type specifications together with the definitions of function and predicate types in the examples of Section 3.1 are polymorphic signatures. In the rest of this chapter we assume that $\Sigma = (\mathcal{T}, Func, Pred)$ is a polymorphic signature for logic programs. Similarly to other typed logics, the variables in a typed logic program are not quantified over all objects, but vary only over objects of a particular type. Thus each variable is annotated with a type expression: Let $\boldsymbol{Var}$ be an infinite set of variable names that are distinguishable from symbols in polymorphic signatures and type variables. Then the set V is called a **set of typed variables** if

- each element of V has the form $x{:}\tau$ where $x \in Var$ is a variable name and $\tau \in T_{\mathcal{T}}(X)$ is a type, and

- $x{:}\tau, x{:}\tau' \in V$ implies $\tau = \tau'$.

We only consider sets of typed variables with unique types so that type errors can be detected at compile time. For instance, if a variable in a clause occurs in two different contexts so that it has type "int" in one context and type "$list(int)$" in the other context, this indicates a type error if all variables in a clause are required to have unique types. In the rest of this chapter we assume that $V, V', V_0, V_1, \ldots$ denote sets of typed variables.

In Church's formulation of the theory of types [Chu40] types are embedded in terms, i.e., each symbol in a term is annotated with an appropriate type expression. These annotations are useful for the unification of typed terms (see Section 3.5). We call $L \leftarrow G$ a *typed program clause* if there is a set of typed variables V and $V \Vdash L \leftarrow G$ is derivable by the inference rules in figure 3.1. The typing rules show that both parametric polymorphism and subtype polymorphism are covered by our framework: If the declared type of a function or predicate contains type variables, then this function or predicate can be applied to any type which is the result of replacing the type variables by other types (parametric polymorphism). If the type specification contains subtype relations as in example 3.2, then a function or predicate with declared argument type $nat(\alpha)$ can also be applied to the subtypes $nat(zero)$ $(=_{\mathcal{T}}$ $zero)$ and $nat(posint)$ $(=_{\mathcal{T}}$ $posint)$.

Note that we have no restrictions on the use of types and type variables in the left-hand side of program clauses in contrast to [MO84] [DH88] [Smo89] and similar

Variable:	$\dfrac{}{V \Vdash x{:}\tau'}$	($x{:}\tau \in V$ and $\tau =_{\mathcal{T}} \tau'$)
Constant:	$\dfrac{}{V \Vdash c{:}\tau'}$	($c{:} \to \tau_c \in Func$ so that there is a $\sigma \in TS(\mathcal{T}, X)$ with $\sigma(\tau_c) =_{\mathcal{T}} \tau'$)
Composite term:	$\dfrac{V \Vdash t_1{:}\tau_1, \ldots, V \Vdash t_n{:}\tau_n}{V \Vdash f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau'}$	($f{:}\tau_f \in Func$ so that there exists $\sigma \in TS(\mathcal{T}, X)$ with $\sigma(\tau_f) = \tau_1, \ldots, \tau_n \to \tau$ and $\tau =_{\mathcal{T}} \tau'$, $n > 0$)
Atom:	$\dfrac{V \Vdash t_1{:}\tau_1, \ldots, V \Vdash t_n{:}\tau_n}{V \Vdash p(t_1{:}\tau_1, \ldots, t_n{:}\tau_n)}$	($p{:}\tau_p \in Pred$ so that there exists $\sigma \in TS(\mathcal{T}, X)$ with $\sigma(\tau_p) = \tau_1, \ldots, \tau_n$, $n \geq 0$)
Goal:	$\dfrac{V \Vdash L_1, \ldots, V \Vdash L_n}{V \Vdash L_1, \ldots, L_n}$	(each L_i is an atom, i.e., has the form $p(\cdots)$, $i = 1, \ldots, n$)
Clause:	$\dfrac{V \Vdash L,\ V \Vdash G}{V \Vdash L \leftarrow G}$	(L is an atom and G is a goal)

Figure 3.1: Typing rules for program clauses

polymorphic type systems.[2] For instance, it is allowed to add the clause

```
member(2,[1,2,3]) ←
```

to the program in example 3.1. By dropping this restriction it is also possible to apply higher-order programming techniques in our framework (cf. Section 3.7).

We call variables, constants and composite terms derivable by these inference rules **($\boldsymbol{\Sigma, X, V}$)-terms** or **well-typed terms**. $\boldsymbol{Term_{\Sigma}(X, V)}$ denotes the set of all (Σ, X, V)-terms. A **ground term** is a term from the set $Term_{\Sigma}(X, \emptyset)$. Well-typed or (Σ, X, V)-atoms, -goals and -clauses are similarly defined (a goal is a set of atoms, but for convenience we denote it without curly brackets). A **Σ-term** (atom, goal, clause) is a (Σ, X, V)-term (atom, goal, clause) for some set of typed variables V.

Lemma 3.7 *If $t{:}\tau$ is a well-typed term and $\tau =_{\mathcal{T}} \tau'$, then $t{:}\tau'$ is also a well-typed term.*

In the following, if s is a syntactic construction (type, term, atom, ...), $tvar(s)$

[2] In these type systems the left-hand side of a clause for a polymorphic predicate must have a type which is equivalent to the declared type of the predicate.

and $var(s)$ will denote the set of type variables and typed variables that occur in s, respectively (i.e., $var(s)$ is a set of typed variables so that s is a $(\Sigma, X, var(s))$-term, atom, ...). For instance, if

$$Tax = \{s_1(s_3) = s_3, s_2(s_3) = s_3\}$$

and $s = f(X{:}s_1(s_3), X{:}s_2(s_3)){:}s_3$, then both $\{X{:}s_3\}$ and $\{X{:}s_1(s_3)\}$ satisfy the definition of $var(s)$, but it is always the case that these different sets are $\mathcal{T}$-equal sets of typed variables. Therefore we can choose one of these sets as $var(s)$. Furthermore, we define $uvar(s) := \{x \mid \exists \tau \in T_{\mathcal{T}}(X)\colon x{:}\tau \in var(s)\}$ as the set of variable names that occur in s.

A **typed logic program** or **typed Horn clause program** $P = (\Sigma, C)$ consists of a polymorphic signature Σ and a set C of Σ-clauses. If it is clear from the context, we will omit the type annotations in the clauses of example programs. Therefore we have written the clauses of the examples in the first chapter without type annotations but we have defined the types of the variables. For instance, the clause

member(E,[E|R]) $\leftarrow$

in example 3.1 denotes the fully typed clause

member(E:α,[E:α|R:$list(\alpha)$]:$list(\alpha)$) $\leftarrow$

and the clause

plus(0,N,N) $\leftarrow$

in example 3.2 denotes the fully typed clause

plus(0:$nat(zero)$,N:$nat(\alpha)$,N:$nat(\alpha)$) $\leftarrow$

This clause is well-typed because "$nat(zero), nat(\alpha), nat(\alpha)$" is an instance of the declared type "$nat(\alpha)$, $nat(\beta)$, $nat(\gamma)$" of the predicate plus and 0:$nat(zero)$ is a well-typed term since $nat(zero) =_{\mathcal{T}} zero$ (where $\mathcal{T}$ is the type specification of example 3.2). The term

[1:nat | []:$list(nat, elist)$]:$nelist(nat)$

is a well-typed term w.r.t. example 3.3 (we assume that 1 is a constant of type nat) since [] is a constant of type $elist$ and $list(nat, elist) =_{\mathcal{T}} elist$ holds in the specified type structure.

3.3 Semantics of Typed Logic Programs

Typed logic programs are interpreted by algebraic structures similar to the ones introduced in [Poi86]. An interpretation of a typed logic program consists of an algebra that satisfies the type specification and a structure for the derived polymorphic signature. A structure is an interpretation of types (elements of sort *type*) as sets, function symbols as operations on these sets and predicate symbols as relations between these sets. Type variables vary over all types of the interpretation and typed variables vary over appropriate carrier sets. The necessary notions are defined in this section.

If $\mathcal{T} = (Ts, Top, Tax)$ is a specification of types, a $\mathcal{T}$-algebra $A = (Ts_A, Top_A)$ which satisfies all equations from Tax is also called **$\mathcal{T}$-type algebra**. The **signature** $\Sigma(A) = (Ts_A, Func_A, Pred_A)$ **derived from** Σ **and** A is defined by

$$\begin{array}{lll} Func_A & := & \{f{:}\sigma(\tau_f) \mid f{:}\tau_f \in Func,\ \sigma{:}\, X \to Ts_A \text{ is a type variable assignment}\} \\ Pred_A & := & \{p{:}\sigma(\tau_p) \mid p{:}\tau_p \in Pred,\ \sigma{:}\, X \to Ts_A \text{ is a type variable assignment}\} \end{array}$$

An **interpretation** of a polymorphic signature Σ (or **Σ-interpretation**) is a $\mathcal{T}$-type algebra $A = (Ts_A, Top_A)$ together with a $\Sigma(A)$-structure (S, δ) which consists of a Ts_A-sorted set $S = (S_\tau | \tau \in Ts_A)$ (the **carrier** of the interpretation) and a denotation δ with:

1. If $f{:}\tau_1, \ldots, \tau_n \to \tau \in Func_A$, then $\delta_{f:\tau_1,\ldots,\tau_n\to\tau}{:}\ S_{\tau_1} \times \cdots \times S_{\tau_n} \ \to\ S_\tau$ is a function.
2. If $p{:}\tau_1, \ldots, \tau_n \in Pred_A$, then $\delta_{p:\tau_1,\ldots,\tau_n} \subseteq S_{\tau_1} \times \cdots \times S_{\tau_n}$ is a relation.

Hence (polymorphic) functions and predicates are interpreted as families of functions and predicates on the given types. In order to compare different interpretations, we define homomorphisms between them. At first, we define $\Sigma(A)$-homomorphisms to compare different $\Sigma(A)$-structures: Let $A = (Ts_A, Top_A)$ be a $\mathcal{T}$-type algebra and (S, δ), (S', δ') be $\Sigma(A)$-structures. A **$\Sigma(A)$-homomorphism** h from (S, δ) into (S', δ') is a family of functions $(h_\tau | \tau \in Ts_A)$ with:

1. $h_\tau{:}\, S_\tau \to S'_\tau$
2. If $f{:}\tau_f \in Func_A$ with $\tau_f = \tau_1, \ldots, \tau_n \to \tau$ $(n \geq 0)$ and $a_i \in S_{\tau_i}$ $(i = 1, \ldots, n)$, then:
   $h_\tau(\delta_{f:\tau_f}(a_1, \ldots, a_n)) = \delta'_{f:\tau_f}(h_{\tau_1}(a_1), \ldots, h_{\tau_n}(a_n))$
3. If $p{:}\tau_p \in Pred_A$ with $\tau_p = \tau_1, \ldots, \tau_n$ $(n \geq 0)$ and $(a_1, \ldots, a_n) \in \delta_{p:\tau_p}$, then:
   $(h_{\tau_1}(a_1), \ldots, h_{\tau_n}(a_n)) \in \delta'_{p:\tau_p}$

If it is clear from the context we omit the indices τ in the functions h_τ. Note that the composition of two $\Sigma(A)$-homomorphisms is again a $\Sigma(A)$-homomorphism. The class of all $\Sigma(A)$-structures together with the $\Sigma(A)$-homomorphisms is a category [EM85]. We denote this category by $Cat_{\Sigma(A)}$.

If A and A' are $\mathcal{T}$-type algebras, then every $\mathcal{T}$-homomorphism $\sigma{:}\, A \to A'$ induces a **signature morphism** $\sigma{:}\, \Sigma(A) \to \Sigma(A')$ and a **forgetful functor**

$U_\sigma: Cat_{\Sigma(A')} \rightarrow Cat_{\Sigma(A)}$ from the category of $\Sigma(A')$-structures into the category of $\Sigma(A)$-structures (see [EM85] for details). Therefore we define a **Σ-homomorphism** from a Σ-interpretation (A, S, δ) into another Σ-interpretation (A', S', δ') as a pair (σ, h), where $\sigma: A \rightarrow A'$ is a $\mathcal{T}$-homomorphism and $h: (S, \delta) \rightarrow U_\sigma((S', \delta'))$ is a $\Sigma(A)$-homomorphism. The class of all Σ-interpretations with the composition

$$(\sigma', h') \circ (\sigma, h) := (\sigma' \circ \sigma, U_\sigma(h') \circ h)$$

of two Σ-homomorphisms is a category. Thus we call a Σ-interpretation (A, S, δ) **initial** in a class of Σ-interpretations $\mathcal{C}$ iff for all Σ-interpretations $(A', S', \delta') \in \mathcal{C}$ there exists a unique Σ-homomorphism from (A, S, δ) into (A', S', δ').

A homomorphism in our typed framework consists of a mapping between type algebras and a mapping between appropriate structures. Consequently, a variable assignment in the typed framework maps type variables into types and typed variables into objects of appropriate types: If $I = ((Ts_A, Top_A), S, \delta)$ is a Σ-interpretation, then a **variable assignment** for (X, V) in I is a pair of mappings $v = (v_X, v_V)$ where $v_X: X \rightarrow Ts_A$ is a type variable assignment and $v_V: V \rightarrow S'$ with $(S', \delta') := U_{v_X}((S, \delta))$ and $v_V(x{:}\tau) \in S'_\tau$ $(= S_{v_X(\tau)})$ for all $x{:}\tau \in V$.

In many-sorted logic, a canonical interpretation for a signature is the *term interpretation* where the carrier sets consist of well-typed terms. In a term interpretation every variable assignment can be uniquely extended to a homomorphism. In our typed framework the situation is more complicated because a variable may correspond to syntactically different terms. For instance, if $s_1 = s_2 \in Tax$, then the variable $x{:}s_1 \in V$ corresponds to the (Σ, X, V)-terms $x{:}s_1$ and $x{:}s_2$. In order to identify such syntactically different terms, we define *canonical terms* as terms where the type annotations are replaced by equivalence classes of types. For this purpose we define a mapping $\mathcal{C}$ which replaces all type annotations in a typed term by equivalence classes of types ($[\tau]$ denotes the equivalence class of the type τ defined by $[\tau] = \{\tau' \mid \tau =_{\mathcal{T}} \tau'\}$):

- $\mathcal{C}(x{:}\tau') := x{:}[\tau]$ for all $x{:}\tau \in V$ and $\tau' =_{\mathcal{T}} \tau$
- $\mathcal{C}(f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau) := f(\mathcal{C}(t_1{:}\tau_1), \ldots, \mathcal{C}(t_n{:}\tau_n)){:}[\tau]$ for all $f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau \in Term_\Sigma(X, V)$ $(n \geq 0)$

$\boldsymbol{CTerm_\Sigma(X, V)} := \{\mathcal{C}(t{:}\tau) \mid t{:}\tau \in Term_\Sigma(X, V)\}$ is the set of **canonical terms**. Now we are able to define the **canonical term interpretation $\boldsymbol{T_\Sigma(X, V)}$** over X and V:
$T_\Sigma(X, V) := (T_{Tax}(X), S, \delta)$, where

1. $T_{Tax}(X) := T_{\mathcal{T}}(X)/\equiv_{Tax}$ is the quotient of the algebra of type expressions by the congruence relation $\equiv_{Tax}$ generated by the axioms in the type specification $\mathcal{T} = (Ts, Top, Tax)$ (the elements of the domain of $T_{Tax}(X)$ are equivalence classes of types).

2. For all $[\tau] \in T_{Tax}(X)$,

$$S_{[\tau]} := \{t{:}[\tau] \mid t{:}[\tau] \in CTerm_{\Sigma}(X,V)\}$$

3. If $f{:}[\tau_1],\ldots,[\tau_n] \to [\tau] \in Func_{T_{Tax}(X)}$ and $t_i{:}[\tau_i] \in S_{[\tau_i]}$ for $i = 1,\ldots,n$, then

$$\delta_{f:[\tau_1],\ldots,[\tau_n]\to[\tau]}(t_1{:}[\tau_1],\ldots,t_n{:}[\tau_n]) := f(t_1{:}[\tau_1],\ldots,t_n{:}[\tau_n]){:}[\tau]$$

4. $\delta_{p:[\tau_1],\ldots,[\tau_n]} := \emptyset$ for all $p{:}[\tau_1],\ldots,[\tau_n] \in Pred_{T_{Tax}(X)}$.

The mappings $\delta_{f:[\tau_1],\ldots,[\tau_n]\to[\tau]}$ in the definition are well-defined by lemma 3.7. Similarly to the notion of "term algebra" in the field of algebraic specification [EM85], a term interpretation $T_{\Sigma}(X,V)$ does not interpret the predicates but supplies a standard structure with objects built from functions and typed variables. Therefore the denotation of predicates are empty sets.

Now we are able to show that any variable assignment can be uniquely extended to a homomorphism:

Lemma 3.8 (Free term structure) *Let (A,S,δ) be a Σ-interpretation and $v = (v_X, v_V)$ be an assignment for (X,V) in (A,S,δ). There exists a unique Σ-homomorphism (σ,h) from $T_{\Sigma}(X,V)$ into (A,S,δ) with $\sigma([\alpha]) = v_X(\alpha)$ for all $\alpha \in X$ and $h(x{:}[\tau]) = v_V(x{:}\tau)$ for all $x{:}\tau \in V$.*

Proof: By lemma 3.6, v_X can be uniquely extended to a $\mathcal{T}$-homomorphism $\sigma{:}T_{Tax}(X) \to A$ with the property $\sigma([\alpha]) = v_X(\alpha)$ for all $\alpha \in X$. We define a $\Sigma(T_{Tax}(X))$-homomorphism h from $T_{\Sigma}(X,V)$ into $U_{\sigma}((S,\delta))$:

1. $h(x{:}[\tau]) := v_V(x{:}\tau)$ for all $x{:}\tau \in V$.

2. $h(c{:}[\tau]) := \delta_{c:\to\sigma([\tau])} \in S_{\sigma([\tau])}$ for all $c{:} \to \tau \in Func_{T_{\mathcal{T}}(X)}$.

3. $$h(f(t_1{:}[\tau_1],\ldots,t_n{:}[\tau_n]){:}[\tau]) := \delta_{f:\sigma([\tau_1]),\ldots,\sigma([\tau_n])\to\sigma([\tau])}(h(t_1{:}[\tau_1]),\ldots,h(t_n{:}[\tau_n]))$$
for all $f{:}\tau_1,\ldots,\tau_n \to \tau \in Func_{T_{\mathcal{T}}(X)}$ and all $t_i{:}[\tau_i] \in CTerm_{\Sigma}(X,V)$.

Clearly h is a $\Sigma(T_{Tax}(X))$-homomorphism. Hence (σ,h) is a Σ-homomorphism. To proof uniqueness of this homomorphism, we assume another Σ-homomorphism (σ',h') from $T_{\Sigma}(X,V)$ into (A,S,δ) with $\sigma'([\alpha]) = v_X(\alpha)$ for all $\alpha \in X$ and $h'(x{:}[\tau]) = v_V(x{:}\tau)$ for all $x{:}\tau \in V$. $\sigma = \sigma'$ by lemma 3.6. We show $h = h'$ by induction on the term structure:

1. $x{:}\tau \in V$: $h'(x{:}[\tau]) = v_V(x{:}\tau) = h(x{:}[\tau])$.

2. $c{:} \to \tau \in Func_{T_{\mathcal{T}}(X)}$: $h'(c{:}[\tau]) = \delta_{c:\to\sigma'([\tau])} = \delta_{c:\to\sigma([\tau])} = h(c{:}[\tau])$.

3. $f(t_1{:}[\tau_1],\ldots,t_n{:}[\tau_n]){:}[\tau]) \in CTerm_\Sigma(X,V)$, $n > 0$:

$$\begin{aligned}
& h'(f(t_1{:}[\tau_1],\ldots,t_n{:}[\tau_n]){:}[\tau]) \\
= \;& \delta_{f:\sigma'([\tau_1]),\ldots,\sigma'([\tau_n])\to\sigma'([\tau])}(h'(t_1{:}[\tau_1]),\ldots,h'(t_n{:}[\tau_n])) \\
= \;& \delta_{f:\sigma([\tau_1]),\ldots,\sigma([\tau_n])\to\sigma([\tau])}(h(t_1{:}[\tau_1]),\ldots,h(t_n{:}[\tau_n])) \\
= \;& h(f(t_1{:}[\tau_1],\ldots,t_n{:}[\tau_n]){:}[\tau])
\end{aligned}$$

■

This lemma is only valid if $T_\Sigma(X,V)$ and the $\mathcal{T}$-algebra A satisfies all equations from Tax. If this is not the case, there exist several different Σ-homomorphisms which extend the variable assignment. For instance, if $s_1 = s_2 \in Tax$ and A has different interpretations of the sorts s_1 and s_2, then the terms $x{:}s_1$ and $x{:}s_2$ may be mapped into different values by different homomorphisms, provided that $x{:}s_1 \in V$.

As a special case ($X = V = \emptyset$) the lemma shows that every ground term without type variables corresponds to a unique value in a given Σ-interpretation. Generally, any variable assignment v can be extended to a Σ-homomorphism in a unique way. In the following we denote that Σ-homomorphism again by v. Since v_X and v_V are only applied to equivalence classes of type expressions and canonical terms, respectively, we omit the indices X and V and write v for both v_X and v_V.

We are not interested in all interpretations of a polymorphic signature but only in those interpretations that satisfy the clauses of a given typed logic program. In order to formalize that we define the validity of atoms, goals and clauses relative to a given Σ-interpretation $I = (A, S, \delta)$:

- Let v be an assignment for (X,V) in I.

  $\boldsymbol{I, v} \models \boldsymbol{L}$ if $L = p(t_1{:}\tau_1,\ldots,t_n{:}\tau_n)$ is a (Σ,X,V)-atom with

  $$(v(\mathcal{C}(t_1{:}\tau_1)),\ldots,v(\mathcal{C}(t_n{:}\tau_n))) \in \delta'_{p:[\tau_1],\ldots,[\tau_n]}$$

  where $U_v((S,\delta)) = (S',\delta')$, i.e., $\delta'_{p:[\tau_1],\ldots,[\tau_n]} = \delta_{p:v([\tau_1]),\ldots,v([\tau_n])}$.

  $\boldsymbol{I, v} \models \boldsymbol{G}$ if G is a (Σ,X,V)-goal with $I, v \models L$ for all $L \in G$

  $\boldsymbol{I, v} \models \boldsymbol{L} \leftarrow \boldsymbol{G}$ if $L \leftarrow G$ is a (Σ,X,V)-clause where $I, v \models G$ implies $I, v \models L$

- $\boldsymbol{I, V} \models \boldsymbol{\mathcal{F}}$ if $\mathcal{F}$ is a (Σ,X,V)-atom, -goal or -clause with $I, v \models \mathcal{F}$ for all variable assignments v for (X,V) in I

We say "L is **valid in** I" if I is a Σ-interpretation with $I, var(L) \models L$ (analogously for goals and clauses). A Σ-interpretation $I = (A,S,\delta)$ is called **model** for a typed logic program (Σ, C) if $I, var(L \leftarrow G) \models L \leftarrow G$ for all clauses $L \leftarrow G \in C$. A (Σ, X, V)-goal G is called **valid in** (Σ, C) relative to V if $I, V \models G$ for every model I of (Σ, C). We shall write: $(\boldsymbol{\Sigma}, \boldsymbol{C}, \boldsymbol{V}) \models \boldsymbol{G}$.

This notion of validity is the extension of validity in untyped Horn clause logic to the typed case: In untyped Horn clause logic an atom, goal or clause is said to be true iff it is true for all variable assignments. In the typed case an atom, goal or clause is said to be true iff it is true for all assignments of type variables and typed variables. The reason for the definition of validity relative to a set of variables is that carrier sets in our interpretations may be empty in contrast to untyped Horn logic. This is also the case in many-sorted logic [GM84]. Validity relative to variables is different from validity in the sense of untyped logic. An example for such a difference can be found in [Han89a], p. 231. Validity in our sense is equivalent to validity in the sense of untyped logic if the types of the variables denote non-empty sets in all interpretations. But a requirement for non-empty carrier sets is not reasonable in the context of polymorphic types.

Example 3.9 The following interpretation is a model for the program of example 3.2. The type specification is interpreted by the $\mathcal{T}$-type algebra $A = (Ts_A, Top_A)$ where $Ts_A = \{nat, zero, posint\}$ and Top_A contains the functions $zero_A$ with $zero_A() = zero$, $posint_A$ with $posint_A() = posint$, and nat_A with $nat_A(\tau) = \tau$ for all $\tau \in Ts_A$. The carrier sets of the interpretation are:

$$S_{zero} = \{0\}$$
$$S_{posint} = \{n \in Nat \mid n > 0\}$$
$$S_{nat} = S_{zero} \cup S_{posint}$$

The constant `0` and the function `s` are interpreted as follows:

$$\delta_{0:\rightarrow zero} = 0$$
$$\delta_{s:zero \rightarrow posint}(0) = 1$$
$$\delta_{s:posint \rightarrow posint}(n) = n + 1 \quad \text{for all } n \in S_{posint}$$
$$\delta_{s:nat \rightarrow posint}(n) = n + 1 \quad \text{for all } n \in S_{nat}$$
$$\delta_{plus:nat,nat,nat} = \{(n_1, n_2, n_3) \in Nat^3 \mid n_1 + n_2 = n_3\}$$
$$\ldots$$

The remaining interpretations of `plus` are the restriction of $\delta_{plus:nat,nat,nat}$ to appropriate subsets. It is easy to show that this interpretation is a model.

3.4 Deduction and Initial Models

In order to define the semantics of typed logic programs we have used canonical terms which are annotated with equivalence classes of types. Since these equivalence classes are sets which may contain an infinite number of elements, this representation is unsuitable for proof procedures like deduction or resolution. Such proof procedures should work on well-typed terms which can be easily handled. Therefore we have to define substitutions on well-typed terms and introduce a relation on well-typed terms that establishes the link to canonical terms.

3.4.1 Typed substitutions

Let $\mu: X \to T_{\mathcal{T}}(X)$ be a mapping from type variables into type expressions and $val: V \to Term_\Sigma(X, V')$ be a mapping from typed variables into well-typed terms over X and V' with the following properties:

- μ is a type variable assignment.
- $val(x{:}\tau) = t{:}\mu(\tau)$ for all $x{:}\tau \in V$, i.e., typed variables of sort τ are mapped into well-typed terms of type $\mu(\tau)$.

We extend the mappings μ and val to mappings on types and well-typed terms, respectively, in the following way:

- $\mu(b) = b$ for all basic types b in $\mathcal{T}$.
- $\mu(h(\tau_1, \ldots, \tau_n)) = h(\mu(\tau_1), \ldots, \mu(\tau_n))$ for all n-ary operation symbols h in $\mathcal{T}$ $(n > 0)$ and all appropriate types $\tau_1, \ldots, \tau_n \in T_{\mathcal{T}}(X)$.
- $val(x{:}\tau') = t{:}\mu(\tau')$ for all $x{:}\tau \in V$ with $val(x{:}\tau) = t{:}\mu(\tau)$ and $\tau' =_{\mathcal{T}} \tau$.
- $val(c{:}\tau) = c{:}\mu(\tau)$ for all well-typed constants $c{:}\tau \in Term_\Sigma(X, V)$.
- $val(f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau) = f(val(t_1{:}\tau_1), \ldots, val(t_n{:}\tau_n)){:}\mu(\tau)$ for all well-typed terms
  $f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau \in Term_\Sigma(X, V)$, $n > 0$.

The mappings are similarly extended on atoms, goals and clauses. We call (μ, val) a **typed substitution**. $\boldsymbol{Sub_\Sigma(X, V, V')}$ denotes the class of all typed substitutions from $(T_{\mathcal{T}}(X), Term_\Sigma(X, V))$ into $(T_{\mathcal{T}}(X), Term_\Sigma(X, V'))$. $\boldsymbol{id_{X,V}} \in Sub_\Sigma(X, V, V)$ denotes the identity in $Sub_\Sigma(X, V, V)$. $\boldsymbol{tdom(\sigma)} := \{\alpha \in X \mid \sigma(\alpha) \neq \alpha\}$ is the **type domain** of a typed substitution σ. A typed substitution keeps the set of type variables X but may change the set of typed variables because the types of the variables influence validity (see Section 3.3). Sometimes we represent typed substitutions by sets. For instance, the set

$$\sigma = \{\alpha/nat,\ x{:}\alpha/0{:}nat\}$$

represents a typed substitution that replaces the type variable α by the type nat and the typed variable $x{:}\alpha$ by the term $0{:}nat$. Hence the result of applying σ to the atom $p(x{:}\alpha, y{:}\alpha)$ is the atom $p(0{:}nat, y{:}nat)$.

The next lemma shows that val is well-defined:

Lemma 3.10 *Let (μ, val) defined as above. Then $val(t{:}\tau) = t'{:}\mu(\tau) \in Term_\Sigma(X, V')$ for all well-typed terms $t{:}\tau \in Term_\Sigma(X, V)$.*

Proof: By induction on the structure of all well-typed terms from $Term_\Sigma(X, V)$:

- $x{:}\tau'$ where $x{:}\tau \in V$, $\tau' =_{\mathcal{T}} \tau$ and $val(x{:}\tau) = t{:}\mu(\tau)$: By definition, $val(x{:}\tau') = t{:}\mu(\tau')$. $t{:}\mu(\tau) \in Term_\Sigma(X, V')$ is a well-typed term and $\mu(\tau) =_{\mathcal{T}} \mu(\tau')$. By lemma 3.7, $t{:}\mu(\tau')$ is also a well-typed term.
- $c{:}\tau$ where $c{:} \rightarrow \tau_c \in Func$ and there exists a type substitution σ with $\sigma(\tau_c) =_{\mathcal{T}} \tau$: $\mu \circ \sigma$ is a type substitution with $\mu(\sigma(\tau_c)) =_{\mathcal{T}} \mu(\tau)$. Hence $val(c{:}\tau) = c{:}\mu(\tau)$ is a well-typed term.
- $f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau$ where $f{:}\tau_f \in Func$ and there exists a type substitution σ with $\sigma(\tau_f) = \tau_1, \ldots, \tau_n \rightarrow \tau'$ and $\tau' =_{\mathcal{T}} \tau$. $\mu \circ \sigma$ is a type substitution with $\mu(\sigma(\tau_f)) = \mu(\tau_1), \ldots, \mu(\tau_n) \rightarrow \mu(\tau')$ and $\mu(\tau') =_{\mathcal{T}} \mu(\tau)$. Hence $val(f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau) = f(val(t_1{:}\tau_1), \ldots, val(t_n{:}\tau_n)){:}\mu(\tau)$ is a well-typed term since $val(t_i{:}\tau_i) = t'_i{:}\mu(\tau_i)$ is well-typed for $i = 1, \ldots, n$ by induction hypothesis.

■

The following lemma states the relationship between typed substitutions and Σ-homomorphisms on canonical term interpretations:

Lemma 3.11 *Let $(\mu, val) \in Sub_\Sigma(X, V, V')$ be a typed substitution. Then there exists a unique Σ-homomorphism σ from $T_\Sigma(X, V)$ into $T_\Sigma(X, V')$ with*

- $\sigma([\alpha]) = [\mu(\alpha)]$ *for all* $\alpha \in X$
- $\sigma(x{:}[\tau]) = \mathcal{C}(val(x{:}\tau))$ *for all* $x{:}\tau \in V$

Furthermore,

$$\sigma([\tau]) = [\mu(\tau)] \text{ for all } \tau \in T_{\mathcal{T}}(X) \tag{1}$$

and

$$\sigma(\mathcal{C}(t{:}\tau)) = \mathcal{C}(val(t{:}\tau)) \text{ for all } t{:}\tau \in Term_\Sigma(X, V) \tag{2}$$

Proof: Let $\sigma_X{:} X \rightarrow T_{Tax}(X)$ be defined by $\sigma_X(\alpha) := [\mu(\alpha)]$ for all $\alpha \in X$. By lemma 3.6, there exists a unique $\mathcal{T}$-homomorphism $\sigma{:} T_{Tax}(X) \rightarrow T_{Tax}(X)$ with $\sigma([\alpha]) = \sigma_X(\alpha)$ for all $\alpha \in X$. If σ_X also denotes the unique extension $\sigma_X{:} T_{\mathcal{T}}(X) \rightarrow T_{Tax}(X)$ (which exists by lemma 3.5), then σ has the property $\sigma_X = \sigma \circ nat$ where nat is the canonical $\mathcal{T}$-homomorphism $nat(\tau) = [\tau]$ for all $\tau \in T_{\mathcal{T}}(X)$ (cf. [EM85], p. 82). We show (1) by induction on the size of τ:

- $\sigma([\alpha]) = \sigma_X(\alpha) = [\mu(\alpha)]$ for all $\alpha \in X$.

- $\sigma([b]) = \sigma_X(b) = [b] = [\mu(b)]$ for all basic types b in $\mathcal{T}$.
- For all n-ary operation symbols h in $\mathcal{T}$ and all appropriate types $\tau_1, \ldots, \tau_n \in T_{\mathcal{T}}(X)$:

$$\begin{aligned}
& \sigma([h(\tau_1, \ldots, \tau_n)]) \\
= \; & \sigma_X(h(\tau_1, \ldots, \tau_n)) \\
= \; & h'(\sigma_X(\tau_1), \ldots, \sigma_X(\tau_n)) \quad (h' \text{ is the interpretation of } h \text{ in } T_{Tax}(X)) \\
= \; & h'(\sigma([\tau_1]), \ldots, \sigma([\tau_n])) \\
= \; & h'([\mu(\tau_1)], \ldots, [\mu(\tau_n)]) \quad \text{(by induction hypothesis)} \\
= \; & [h(\mu(\tau_1), \ldots, \mu(\tau_n))] \quad (\text{by definition of } h') \\
= \; & [\mu(h(\tau_1, \ldots, \tau_n))]
\end{aligned}$$

Let $\sigma_V : V \to CTerm_\Sigma(X, V')$ be defined by $\sigma_V(x{:}\tau) := \mathcal{C}(val(x{:}\tau))$ for $x{:}\tau \in V$. $val(x{:}\tau)$ is a well-typed term of type $\mu(\tau)$, hence $\mathcal{C}(val(x{:}\tau))$ has the form $t{:}[\mu(\tau)] = t{:}\sigma([\tau]) = t{:}\sigma_X(\tau)$. Therefore (σ_X, σ_V) is a variable assignment for (X, V) in $T_\Sigma(X, V')$ which can be uniquely extended to a Σ-homomorphism σ from $T_\Sigma(X, V)$ into $T_\Sigma(X, V')$ by lemma 3.8. We prove (2) by induction on the size of terms:

- For all $x{:}\tau \in V$ with $val(x{:}\tau) = t{:}\mu(\tau)$ and $\tau' =_{\mathcal{T}} \tau$: $\sigma(\mathcal{C}(x{:}\tau')) = \sigma(x{:}[\tau']) = \sigma(x{:}[\tau]) = \sigma_V(x{:}\tau) = \mathcal{C}(val(x{:}\tau)) = \mathcal{C}(t{:}\mu(\tau)) = \mathcal{C}(t{:}\mu(\tau')) = \mathcal{C}(val(x{:}\tau'))$.
- $\sigma(\mathcal{C}(c{:}\tau)) = \sigma(c{:}[\tau]) = c{:}\sigma([\tau]) = c{:}[\mu(\tau)] = \mathcal{C}(c{:}\mu(\tau)) = \mathcal{C}(val(c{:}\tau))$ for all constants $c{:}\tau \in Term_\Sigma(X, V)$.
- For all terms $f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau \in Term_\Sigma(X, V)$, $n > 0$:

$$\begin{aligned}
\sigma(\mathcal{C}(f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau)) &= \sigma(f(\mathcal{C}(t_1{:}\tau_1), \ldots, \mathcal{C}(t_n{:}\tau_n)){:}[\tau]) \\
&= f(\sigma(\mathcal{C}(t_1{:}\tau_1)), \ldots, \sigma(\mathcal{C}(t_n{:}\tau_n))){:}\sigma([\tau]) \\
&= f(\mathcal{C}(val(t_1{:}\tau_1)), \ldots, \mathcal{C}(val(t_n{:}\tau_n))){:}[\mu(\tau)] \\
&= \mathcal{C}(f(val(t_1{:}\tau_1), \ldots, val(t_n{:}\tau_n)){:}\mu(\tau)) \\
&= \mathcal{C}(val(f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau))
\end{aligned}$$

Uniqueness can be simply shown by induction on the size of terms. ∎

The above lemma shows that typed substitutions which are directly applied to well-typed terms correspond to Σ-homomorphisms between canonical term interpretations in a unique way. Hence $\hat{\sigma}$ denotes the Σ-homomorphism from $T_\Sigma(X, V)$ into $T_\Sigma(X, V')$ corresponding to the typed substitution $\sigma \in Sub_\Sigma(X, V, V')$. The following lemma shows a relationship between variable assignments and typed substitutions w.r.t. validity:

Lemma 3.12 *Let I be a Σ-interpretation, G be a (Σ, X, V)-goal, $\sigma \in Sub_\Sigma(X, V, V')$ and v be a variable assignment for (X, V') in I. Then $I, v \models \sigma(G)$ iff $I, v \circ \hat{\sigma} \models G$.*

Proof: Let G, σ, $v = (v_X, v_V)$ and $I = (A, S, \delta)$ be given. The composition $v' := v \circ \hat{\sigma}$ between Σ-homomorphisms is defined by $v' = (v'_X, v'_V)$ with $v'_X([\alpha]) = v_X(\hat{\sigma}([\alpha]))$ for all $\alpha \in X$ and

$$v'_{V,[\tau]}(x{:}[\tau]) = (U_{\hat{\sigma}}(v_V) \circ \hat{\sigma})_{[\tau]}(x{:}[\tau]) = v_{V,\hat{\sigma}([\tau])}(\hat{\sigma}(x{:}[\tau]))$$

for all $x{:}\tau \in V$. Thus v' is a variable assignment for (X, V) in I. Let $p(\ldots t_i{:}\tau_i \ldots) \in G$. Then

$$\begin{array}{ll}
 & I, v \models \sigma\,(p(\ldots t_i{:}\tau_i \ldots)) \\
\Longleftrightarrow & I, v \models p(\ldots \sigma(t_i{:}\tau_i) \ldots) \\
\Longleftrightarrow & (\ldots v_V(\mathcal{C}(\sigma(t_i{:}\tau_i))) \ldots) \in \delta_{p:\ldots v_X(\hat{\sigma}([\tau_i]))\ldots} \\
\Longleftrightarrow & (\ldots v_V(\hat{\sigma}(\mathcal{C}(t_i{:}\tau_i))) \ldots) \in \delta_{p:\ldots v_X(\hat{\sigma}([\tau_i]))\ldots} \quad \text{(by lemma 3.11)} \\
\Longleftrightarrow & (\ldots v'_V(\mathcal{C}(t_i{:}\tau_i)) \ldots) \in \delta_{p:\ldots v'_X([\tau_i])\ldots} \\
\Longleftrightarrow & I, v' \models p(\ldots t_i{:}\tau_i \ldots)
\end{array}$$

This proves the lemma. ■

A term $t' \in Term_\Sigma(X, V')$ is called an **instance** of a term $t \in Term_\Sigma(X, V)$ if a typed substitution $\sigma \in Sub_\Sigma(X, V, V')$ exists with $t' = \sigma(t)$. The definition of instances can be extended to atoms, goals and clauses. We omit the simple definitions here. The next lemma shows the relationship between the validity of a clause and the validity of all its instances:

Lemma 3.13 *Let $I = (A, S, \delta)$ be a Σ-interpretation and $L \leftarrow G$ be a (Σ, X, V)-clause. Then:*

$$I, V \models L \leftarrow G \qquad \Longleftrightarrow \qquad I, V' \models \sigma(L) \leftarrow \sigma(G) \text{ for all } \sigma \in Sub_\Sigma(X, V, V')$$

Proof: The direction "$\Longleftarrow$" is trivial if we use the identity $id_{X,V}$ for the typed substitution σ. Let $I, V \models L \leftarrow G$ and $\sigma \in Sub_\Sigma(X, V, V')$ be a typed substitution. We have to show $I, V' \models \sigma(L) \leftarrow \sigma(G)$. Let v be a variable assignment for (X, V') in I with $I, v \models \sigma(G)$ (if there exists no such variable assignment, $I, V' \models \sigma(L) \leftarrow \sigma(G)$ is trivially true). Lemma 3.12 yields $I, v \circ \hat{\sigma} \models G$. This implies $I, v \circ \hat{\sigma} \models L$ since $I, V \models L \leftarrow G$. Again by lemma 3.12, it follows $I, v \models \sigma(L)$. ■

Along with a set of Σ-clauses C we define the **set of instantiated clauses** $\widehat{C}$ as follows:

$$\widehat{C} \;:=\; \{L \leftarrow G \mid L \leftarrow G \text{ is an instance of a clause from } C\}$$

The set $\widehat{C}$ contains all clauses which are obtained from clauses in C by substituting type expressions for type variables and well-typed terms for typed variables.

Corollary 3.14 *A Σ-interpretation is a model for (Σ, C) iff it is a model for $(\Sigma, \widehat{C})$.*

Proof: The theorem follows by definition of $\widehat{C}$ and lemma 3.13. ∎

3.4.2 Equality w.r.t. the type structure

Our proof procedures (deduction, resolution) manipulate only well-typed terms and use typed substitutions. For that purpose we define an important relation on well-typed terms: Two Σ-terms t and t' are called **$\mathcal{T}$-equal**, denoted $t =_{\mathcal{T}} t'$, if $\mathcal{C}(t) = \mathcal{C}(t')$. $\mathcal{T}$-equality on atoms is analogously defined. Two finite sets of typed variables V_1 and V_2 are called $\mathcal{T}$-equal if $V_1 = \{x_1{:}\xi_1, \ldots, x_m{:}\xi_m\}$, $V_2 = \{x_1{:}\xi_1', \ldots, x_m{:}\xi_m'\}$ and $\xi_i =_{\mathcal{T}} \xi_i'$ for $i = 1, \ldots, m$.

Example 3.15 If the type specification of example 3.2 is given, then the following pairs of well-typed terms are $\mathcal{T}$-equal:

$$\begin{aligned} \mathtt{0}{:}nat(zero) &=_{\mathcal{T}} \mathtt{0}{:}zero \\ \mathtt{N}{:}posint &=_{\mathcal{T}} \mathtt{N}{:}nat(posint) \end{aligned}$$

The proof of the following two lemmas is straightforward:

Lemma 3.16 *If two Σ-terms t and t' are $\mathcal{T}$-equal, then $var(t)$ and $var(t')$ are $\mathcal{T}$-equal sets of typed variables.*

Lemma 3.17 *If two Σ-terms t and t' are $\mathcal{T}$-equal, then all instances $\sigma(t)$ and $\sigma(t')$ are $\mathcal{T}$-equal.*

The next lemma shows that $\mathcal{T}$-equal atoms have the same meaning in all interpretations:

Lemma 3.18 *Let Σ be a polymorphic signature, V be a set of typed variables, and L_1 and L_2 be two $\mathcal{T}$-equal (Σ, X, V)-atoms. If I is a Σ-interpretation and v is a variable assignment for V in I, then:*

$$I, v \models L_1 \qquad \Longleftrightarrow \qquad I, v \models L_2$$

Proof: Let $I = (A, S, \delta)$ be a Σ-interpretation and v be a variable assignment for V in I. Let L_1 and L_2 be two $\mathcal{T}$-equal (Σ, X, V)-atoms. Hence $L_1 = p(t_1{:}\tau_1, \ldots, t_k{:}\tau_k)$, $L_2 = p(t'_1{:}\tau'_1, \ldots, t'_k{:}\tau'_k)$, and $t_i{:}\tau_i =_\mathcal{T} t'_i{:}\tau'_i$ for $i = 1, \ldots, k$. By definition of $\mathcal{T}$-equality, $v(\mathcal{C}(t_i{:}\tau_i)) = v(\mathcal{C}(t'_i{:}\tau'_i))$ for $i = 1, \ldots, k$. $I, v \models L_1$ is equivalent to

$$(v(\mathcal{C}(t_1{:}\tau_1)), \ldots, v(\mathcal{C}(t_k{:}\tau_k))) \in \delta_{p:v([\tau_1]),\ldots,v([\tau_k])}$$

Since $[\tau_i] = [\tau'_i]$ for $i = 1, \ldots, k$, we obtain

$$(v(\mathcal{C}(t'_1{:}\tau'_1)), \ldots, v(\mathcal{C}(t'_k{:}\tau'_k))) \in \delta_{p:v([\tau'_1]),\ldots,v([\tau'_k])}$$

which is equivalent to $I, v \models L_2$. The other direction is symmetric. ∎

3.4.3 The typed Horn clause calculus

This section presents an inference system for proving validity in typed logic programs. In contrast to the untyped Horn clause calculus it is necessary to collect all variables used in a derivation of the inference system since validity depends on the types of variables. Let (Σ, C) be a typed logic program. We assume that equality between types (relation $=_\mathcal{T}$) is decidable. The **typed Horn clause calculus** contains the following inference rules (remember that goals are finite sets of atoms and therefore we use set notations for the modification of goals):

1. **Axioms:** If V is a set of typed variables and $L \leftarrow G \in C$ is a (Σ, X, V)-clause, then $(\Sigma, C, V) \vdash L \leftarrow G$.

2. **Substitution rule:** If $(\Sigma, C, V) \vdash L \leftarrow G$ and $\sigma \in Sub_\Sigma(X, V, V')$, then $(\Sigma, C, V') \vdash \sigma(L) \leftarrow \sigma(G)$.

3. **Cut rule:** If $(\Sigma, C, V) \vdash L \leftarrow G_0 \cup \{L_0\}$, $(\Sigma, C, V) \vdash L_1 \leftarrow G_1$, and $L_0 =_\mathcal{T} L_1$,
   then $(\Sigma, C, V) \vdash L \leftarrow G_0 \cup G_1$.

We write $\boldsymbol{(\Sigma, C, V) \vdash L}$ if $(\Sigma, C, V) \vdash L \leftarrow \emptyset$ can be deduced by these inference rules.

The soundness of the typed Horn clause calculus can be shown by proving the soundness of each inference rule:

Theorem 3.19 (Soundness of deduction) *Let (Σ, C) be a typed logic program, V be a set of typed variables and L be a (Σ, X, V)-atom. If $(\Sigma, C, V) \vdash L$, then $(\Sigma, C, V) \models L$.*

Proof: Let M be a model for (Σ, C). By induction on the length of a deduction we show that $M, V_i \models L_i \leftarrow G_i$ for each element $(\Sigma, C, V_i) \vdash L_i \leftarrow G_i$ in a deduction for $L \leftarrow \emptyset$.

1. *Axioms:* If $L_i \leftarrow G_i \in C$, then $M, var(L_i \leftarrow G_i) \models L_i \leftarrow G_i$. Let v be a variable assignment for (X, V_i) in M (if there exists no such variable assignment, then $M, V_i \models L_i \leftarrow G_i$ is trivially true). Let v' be the restriction of v to $(X, var(L_i \leftarrow G_i))$. Then $M, v' \models L_i \leftarrow G_i$ is true and therefore $M, v \models L_i \leftarrow G_i$ is also true.

2. *Substitution rule:* Let $\sigma \in Sub_\Sigma(X, V_i, V_i')$ be a typed substitution, $\hat{\sigma}$ be the corresponding Σ-homomorphism (cf. lemma 3.11) and v' be a variable assignment for (X, V_i') in M (if there exists no such variable assignment, then $M, V_i' \models \sigma(L_i) \leftarrow \sigma(G_i)$ is trivially true). $v := v' \circ \hat{\sigma}$ is a variable assignment for (X, V_i) in M. By induction hypothesis, $M, v \models L_i \leftarrow G_i$. Suppose now that $M, v' \models \sigma(G_i)$. Lemma 3.12 yields $M, v \models G_i$. This implies $M, v \models L_i$ and, again by lemma 3.12, $M, v' \models \sigma(L_i)$. Therefore, $M, v' \models \sigma(L_i) \leftarrow \sigma(G_i)$.

3. *Cut rule:* Let $(\Sigma, C, V_i) \vdash L_i \leftarrow G_i \cup \{L_i'\}$ and $(\Sigma, C, V_j) \vdash L_j \leftarrow G_j$ be elements of the deduction with $V_i = V_j$ and $L_i' =_{\mathcal{T}} L_j$. Let v be a variable assignment for (X, V_i) in M with $M, v \models G_i \cup G_j$ (if there exists no such variable assignment, then $M, V_i \models L_i \leftarrow G_i \cup G_j$ is trivially true). By induction hypothesis, $M, v \models L_i \leftarrow G_i \cup \{L_i'\}$ and $M, v \models L_j \leftarrow G_j$. Since $M, v \models G_j$, we obtain $M, v \models L_j$ which is equivalent to $M, v \models L_i'$ by lemma 3.18. On the other hand, $M, v \models G_i$. Hence $M, v \models G_i \cup \{L_i'\}$ and $M, v \models L_i$. Therefore, $M, v \models L_i \leftarrow G_i \cup G_j$, as required.

■

The completeness of deduction is proved by the construction of a particular model that is the extension of a free term interpretation to an interpretation with particular predicate denotations.

Let V be a set of typed variables. The **deductive term interpretation** $\boldsymbol{T_{\Sigma,C}(X, V)}$ of the typed logic program (Σ, C) is the triple $(T_{Tax}(X), S, \delta)$ with:

1. $T_{Tax}(X) := T_{\mathcal{T}}(X)/\equiv_{Tax}$, the quotient of the algebra of type expressions by the congruence relation $\equiv_{Tax}$ generated by the axioms in the type specification $\mathcal{T} = (Ts, Top, Tax)$.

2. For all $[\tau] \in T_{Tax}(X)$,

$$S_{[\tau]} \;:=\; \{t{:}[\tau] \mid t{:}[\tau] \in CTerm_\Sigma(X, V)\}$$

3. If $f{:}[\tau_1], \ldots, [\tau_n] \to [\tau] \in Func_{T_{Tax}(X)}$ and $t_i{:}[\tau_i] \in S_{[\tau_i]}$ for $i = 1, \ldots, n$, then

$$\delta_{f:[\tau_1],\ldots,[\tau_n]\to[\tau]}(t_1{:}[\tau_1], \ldots, t_n{:}[\tau_n]) := f(t_1{:}[\tau_1], \ldots, t_n{:}[\tau_n]){:}[\tau]$$

4. If $p{:}[\tau_1],\ldots,[\tau_n] \in Pred_{T_{Tax}(X)}$, then

$$\delta_{p:[\tau_1],\ldots,[\tau_n]} := \{(t_1{:}[\tau_1],\ldots,t_n{:}[\tau_n]) \mid \begin{array}{l}(\Sigma,C,V) \vdash p(t'_1{:}\tau'_1,\ldots,t'_n{:}\tau'_n) \text{ and} \\ t_i{:}[\tau_i] = \mathcal{C}(t'_i{:}\tau'_i)\}\end{array}$$

The difference between $T_\Sigma(X,V)$ and $T_{\Sigma,C}(X,V)$ is the denotation of predicate symbols.

The completeness proof of the typed Horn clause calculus is based on the fact that $T_{\Sigma,C}(X,V)$ is a model for (Σ,C). Therefore we need the following lemma:

Lemma 3.20 *Let (Σ,C) be a typed logic program and V be a set of typed variables. $T_{\Sigma,C}(X,V)$ is a model for (Σ,C).*

Proof: It is clear by the above definition that $T_{\Sigma,C}(X,V) = (T_{Tax}(X),S,\delta)$ is a Σ-interpretation. We have to prove that all clauses from C are valid in $T_{\Sigma,C}(X,V)$. Let $L \leftarrow G$ be a clause from C and v be a variable assignment for (X,V_c) in $T_{\Sigma,C}(X,V)$ with $T_{\Sigma,C}(X,V),v \models G$, where $V_c := var(L \leftarrow G)$. $v(\alpha) \in T_{Tax}(X)$ for all $\alpha \in X$ and $v(x{:}\tau) \in S_{v(\tau)}$ for all $x{:}\tau \in V_c$, i.e., each variable from X and V_c is mapped into an equivalence class of types and a canonical term, respectively. We choose from each equivalence class $v(\alpha)$ a representative τ_α with $v(\alpha) = [\tau_\alpha]$ and for each canonical term $v(x{:}\tau)$ a well-typed term $t_x{:}\tau_x$ with $v(x{:}\tau) = \mathcal{C}(t_x{:}\tau_x)$. We define a typed substitution by $v'(\alpha) = \tau_\alpha$ for all $\alpha \in X$ and $v'(x{:}\tau) = t_x{:}v'(\tau)$ for all $x{:}\tau \in V_c$ ($t_x{:}v'(\tau)$ is a well-typed term by lemma 3.7 since $[\tau_x] = v(\tau) = [v'(\tau)]$). Lemma 3.11 yields $v([\tau]) = [v'(\tau)]$ for all $\tau \in T_{\mathcal{T}}(X)$ and $v(\mathcal{C}(t{:}\tau)) = \mathcal{C}(v'(t{:}\tau))$ for all $t{:}\tau \in Term_\Sigma(X,V_c)$. If $G = L_1,\ldots,L_k$, then $T_{\Sigma,C}(X,V),v \models L_i$, for $i = 1,\ldots,k$. If $L_i = p_i(t_{i1}{:}\tau_{i1},\ldots,t_{in_i}{:}\tau_{in_i})$, we obtain

$$(v(\mathcal{C}(t_{i1}{:}\tau_{i1})),\ldots,v(\mathcal{C}(t_{in_i}{:}\tau_{in_i}))) \in \delta_{p_i:v([\tau_{i1}]),\ldots,v([\tau_{in_i}])}$$

and, by lemma 3.11,

$$(\mathcal{C}(v'(t_{i1}{:}\tau_{i1})),\ldots,\mathcal{C}(v'(t_{in_i}{:}\tau_{in_i}))) \in \delta_{p_i:[v'(\tau_{i1})],\ldots,[v'(\tau_{in_i})]}$$

By definition of $T_{\Sigma,C}(X,V)$, there exists a (Σ,X,V)-atom L'_i with

$$(\Sigma,C,V) \ \vdash \ L'_i \quad \text{and} \quad L'_i =_{\mathcal{T}} p_i(v'(t_{i1}{:}\tau_{i1}),\ldots,v'(t_{in_i}{:}\tau_{in_i}))$$

On the other hand, $(\Sigma,C,V_c) \vdash L \leftarrow G$ is true, and therefore $(\Sigma,C,V) \vdash v'(L) \leftarrow v'(G)$ by the substitution rule. By the cut rule, we can infer $(\Sigma,C,V) \vdash v'(L) \leftarrow$. If $L = p(t_1{:}\tau_1,\ldots,t_n{:}\tau_n)$, then

$$(\Sigma,C,V) \vdash p(v'(t_1{:}\tau_1),\ldots,v'(t_n{:}\tau_n))$$

By definition of $T_{\Sigma,C}(X,V)$,

$$(\mathcal{C}(v'(t_1{:}\tau_1)),\ldots,\mathcal{C}(v'(t_n{:}\tau_n))) \in \delta_{p:[v'(\tau_1)],\ldots,[v'(\tau_n)]}$$

Lemma 3.11 yields

$$(v(\mathcal{C}(t_1{:}\tau_1)), \ldots, v(\mathcal{C}(t_n{:}\tau_n))) \in \delta_{p:v([\tau_1]),\ldots,v([\tau_n])}$$

which implies $T_{\Sigma,C}(X,V), v \models L$. ■

Now we are prepared to state the completeness of the typed Horn clause calculus:

Theorem 3.21 (Completeness of deduction) *Let (Σ, C) be a typed logic program, V be a set of typed variables and L be a (Σ, X, V)-atom with $(\Sigma, C, V) \models L$. Then there exists a (Σ, X, V)-atom L' with $L =_{\mathcal{T}} L'$ and $(\Sigma, C, V) \vdash L'$.*

Proof: Let $(\Sigma, C, V) \models L$ and $L = p(t_1{:}\tau_1, \ldots, t_n{:}\tau_n)$. By the last lemma, $T_{\Sigma,C}(X,V) = (T_{Tax}(X), S, \delta)$ is a model for (Σ, C). This implies $T_{\Sigma,C}(X,V), V \models L$. In particular we have $T_{\Sigma,C}(X,V), id \models L$ (where $id(\alpha) = [\alpha]$ for all $\alpha \in X$ and $id(x{:}\tau) = x{:}[\tau]$ for all $x{:}\tau \in V$) which implies

$$(\mathcal{C}(t_1{:}\tau_1), \ldots, \mathcal{C}(t_n{:}\tau_n)) \in \delta_{p:[\tau_1],\ldots,[\tau_n]}$$

By definition of $T_{\Sigma,C}(X,V)$, there exist $t'_i{:}\tau'_i$ $(i = 1, \ldots, n)$ with $(\Sigma, C, V) \vdash L'$ where $L' = p(t'_1{:}\tau'_1, \ldots, t'_n{:}\tau'_n)$ and $\mathcal{C}(t_i{:}\tau_i) = \mathcal{C}(t'_i{:}\tau'_i)$ for $i = 1, \ldots, n$. Thus $L =_{\mathcal{T}} L'$, as required. ■

The typed Horn clause calculus is only complete up to $\mathcal{T}$-equality since $\mathcal{T}$-equal atoms are only compared in the cut rule. For instance, if

`p(0:`$zero$`)` $\leftarrow$

is the only clause for predicate `p`$:\alpha$ and $zero =_{\mathcal{T}} nat(zero)$, then $(\Sigma, C, \emptyset) \models$ `p(0:`$nat(zero)$`)` (by lemma 3.18), but $(\Sigma, C, \emptyset) \vdash$ `p(0:`$nat(zero)$`)` is not provable in the typed Horn clause calculus.

3.4.4 Initial model

This section shows the existence of an initial model for any typed logic program. The carrier set of this initial model contains all canonical terms without type variables and typed variables. This result is a consequence of the previous section on the typed Horn clause calculus.

Theorem 3.22 (Initial model) *Let (Σ, C) be a typed logic program. Then $T_{\Sigma,C} := T_{\Sigma,C}(\emptyset, \emptyset)$ is initial in the class of all models for (Σ, C).*

Proof: Let $T_{\Sigma,C} = (T_{Tax}, S_I, \delta_I)$. By lemma 3.20, this is a model for (Σ, C). Let $I = (A, S, \delta)$ be another model for (Σ, C) and $T_\Sigma(\emptyset, \emptyset)$ be the term interpretation with ground terms. By lemma 3.8 (free term structure), there exists a unique Σ-homomorphism (σ, h) from $T_\Sigma(\emptyset, \emptyset)$ into I. In order to show that (σ, h) is a Σ-homomorphism from $T_{\Sigma,C}$ into I, we have to prove the following implication

$$(t_1{:}[\tau_1], \ldots, t_n{:}[\tau_n]) \in \delta_{I,p:[\tau_1],\ldots,[\tau_n]} \Longrightarrow (h(t_1{:}[\tau_1]), \ldots, h(t_n{:}[\tau_n])) \in \delta_{p:\sigma([\tau_1]),\ldots,\sigma([\tau_n])}$$

because the only difference between $T_\Sigma(\emptyset, \emptyset)$ and $T_{\Sigma,C}$ is the denotation of predicate symbols.

Let $p{:}[\tau_1], \ldots, [\tau_n] \in Pred_{T_{Tax}}$ and $(t_1{:}[\tau_1], \ldots, t_n{:}[\tau_n]) \in \delta_{I,p:[\tau_1],\ldots,[\tau_n]}$. By definition of $T_{\Sigma,C}$, there exist $t'_i{:}\tau'_i$ with $\mathcal{C}(t'_i{:}\tau'_i) = t_i{:}[\tau_i]$ $(i = 1, \ldots, n)$ and

$$(\Sigma, C, \emptyset) \vdash p(t'_1{:}\tau'_1, \ldots, t'_n{:}\tau'_n)$$

Theorem 3.19 implies $(\Sigma, C, \emptyset) \models p(t'_1{:}\tau'_1, \ldots, t'_n{:}\tau'_n)$.

$\Longrightarrow \quad I, \emptyset \models p(t'_1{:}\tau'_1, \ldots, t'_n{:}\tau'_n)$

$\Longrightarrow \quad I, (\sigma, h) \models p(t'_1{:}\tau'_1, \ldots, t'_n{:}\tau'_n)$

$\Longrightarrow \quad (h(\mathcal{C}(t'_1{:}\tau'_1)), \ldots, h(\mathcal{C}(t'_n{:}\tau'_n))) \in \delta_{p:\sigma([\tau'_1]),\ldots,\sigma([\tau'_n])}$

$\Longrightarrow \quad (h(t_1{:}[\tau_1]), \ldots, h(t_n{:}[\tau_n])) \in \delta_{p:\sigma([\tau_1]),\ldots,\sigma([\tau_n])}$

Therefore (σ, h) is a Σ-homomorphism from $T_{\Sigma,C}$ into I which implies the initiality of $T_{\Sigma,C}$. ∎

3.5 Unification

In logic programming we are interested in a systematic method for proving validity of goals. The typed Horn clause calculus is very inefficient for this purpose. In untyped Horn clause logic the resolution principle [Rob65] is the basic proof method where a most general unifier of two atoms must be computed in each resolution step. We need a similar operation for the resolution method in our typed framework. As in order-sorted logic, the unification problem is not unitary in our general framework and therefore complete sets of unifiers must be considered. This section defines the unification w.r.t. a type specification $\mathcal{T}$ and presents a non-deterministic algorithm for computing complete sets of unifiers.

Example 3.23 Consider example 3.2. The first clause for `plus`

`plus(0:`$nat(zero)$`,N:`$nat(\alpha)$`,N:`$nat(\alpha)$`)`

cannot be applied to prove the goal

`plus(N1:`$nat(posint)$`,N2:`$nat(\beta)$`,N3:`$nat(\gamma)$`)`

since this would cause the binding of variable `N1` to `0` which yields the ill-typed term `0`:$nat(posint)$. In order to avoid such bindings, the unification procedure has to take into account that `N1` and `0` have the non-unifiable types $nat(posint)$ and $nat(zero)$. On the other hand, if the clause

`p(N:`$nat(zero)$`)` $\leftarrow \cdots$

is applied to prove the goal

`p(N1:`$nat(\alpha)$`)`

then the variable `N1` is constrained to type $nat(zero)$ which may avoid some unnecessary search and backtracking steps in the subsequent proof. Therefore the unification procedure has to consider the types of the terms. An untyped unification cannot be applied in our framework.

We have mentioned in Section 3.4 that our proof procedures should manipulate well-typed terms rather than canonical terms. Therefore we have introduced *typed substitutions* which are mappings on type expressions and well-typed terms and directly related to Σ-homomorphisms between canonical term interpretations. Hence we want to define a unifier w.r.t. a type specification $\mathcal{T}$ as a distinct typed substitution. Since the composition of two typed substitutions is again a typed substitution, we can define the following notions (we assume that V, V_1, V_2 are sets of typed variables):

- Let $\sigma, \sigma' \in Sub_\Sigma(X, V, V_1)$ be typed substitutions. We write $\sigma =_\mathcal{T} \sigma'$ iff $\sigma(\alpha) =_\mathcal{T} \sigma'(\alpha)$ for all $\alpha \in X$ and $\sigma(x{:}\tau) =_\mathcal{T} \sigma'(x{:}\tau)$ for all $x{:}\tau \in V$.

- Let $\sigma \in Sub_\Sigma(X, V, V_1)$ and $\sigma' \in Sub_\Sigma(X, V, V_2)$ be typed substitutions. σ is **more general** than σ' w.r.t. $\mathcal{T}$ or σ' is a ***$\mathcal{T}$*-instance** of σ, denoted $\sigma \leq_\mathcal{T} \sigma'$, iff there exists $\phi \in Sub_\Sigma(X, V_1, V_2)$ with $\phi \circ \sigma =_\mathcal{T} \sigma'$.

- Let t and t' be (Σ, X, V)-terms. t and t' are ***$\mathcal{T}$*-unifiable** if there exists a typed substitution $\sigma \in Sub_\Sigma(X, V, V')$ with $\sigma(t) =_\mathcal{T} \sigma(t')$ for a set of typed variables V'. In this case σ is called a ***$\mathcal{T}$*-unifier** for t and t'. By $\boldsymbol{SU_\mathcal{T}(t, t')}$ we denote the set of all $\mathcal{T}$-unifiers for t and t'.

- Let t and t' be (Σ, X, V)-terms. We call a set of typed substitutions $\boldsymbol{CSU_\mathcal{T}(t, t')}$ a **complete set of *$\mathcal{T}$*-unifiers** for t and t' if the following conditions hold:

  - $CSU_\mathcal{T}(t, t') \subseteq SU_\mathcal{T}(t, t')$
  - For all $\sigma' \in SU_\mathcal{T}(t, t')$ there exists a typed substitution $\sigma \in CSU_\mathcal{T}(t, t')$ with $\sigma \leq_\mathcal{T} \sigma'$.

(T) $(\sigma,\ \langle t_1{:}\tau_1 = t_2{:}\tau_2, E_r\rangle) \quad \stackrel{unif}{\Longrightarrow} \quad (\phi \circ \sigma,\ \langle \phi(t_1{:}\tau_1) = \phi(t_2{:}\tau_2), \phi(E_r)\rangle)$
if $\phi \in CSU_{\mathcal{T}}(\tau_1, \tau_2)$ and not $\tau_1 =_{\mathcal{T}} \tau_2$

(E1) $(\sigma,\ \langle x{:}\tau = t{:}\tau', E_r\rangle) \quad \stackrel{unif}{\Longrightarrow} \quad (\sigma' \circ \sigma,\ \sigma'(E_r))$
if $\tau =_{\mathcal{T}} \tau'$, $x \in Var$, x does not occur in $t{:}\tau'$ and $\sigma' = \{x{:}\tau / t{:}\tau\}$

(E2) $(\sigma,\ \langle t{:}\tau' = x{:}\tau, E_r\rangle) \quad \stackrel{unif}{\Longrightarrow} \quad (\sigma' \circ \sigma,\ \sigma'(E_r))$
if $\tau =_{\mathcal{T}} \tau'$, $x \in Var$, x does not occur in $t{:}\tau'$ and $\sigma' = \{x{:}\tau / t{:}\tau\}$

(D) $(\sigma,\ \langle f(t_1, \ldots, t_n){:}\tau = f(t'_1, \ldots, t'_n){:}\tau', E_r\rangle) \quad \stackrel{unif}{\Longrightarrow} \quad (\sigma,\ \langle t_1 = t'_1, \ldots, t_n = t'_n, E_r\rangle)$
if $\tau =_{\mathcal{T}} \tau'$ $(n \geq 0)$

Figure 3.2: Rules for $\mathcal{T}$-unification of well-typed terms. In the first rule (T) the type substitution ϕ is extended to a typed substitution by $\phi(x{:}\tau) := x{:}\phi(\tau)$ for all $x{:}\tau \in V'$ if $\sigma \in Sub_\Sigma(X, V, V')$.

$\mathcal{T}$-unifiers and complete sets of $\mathcal{T}$-unifiers for type expressions are analogously defined as particular (sets of) type substitutions.

Obviously, the set of all $\mathcal{T}$-unifiers is also a complete set of $\mathcal{T}$-unifiers, but usually we are interested in algorithms which enumerate a complete set of $\mathcal{T}$-unifiers with some minimality condition. We do not discuss this in detail here. We assume a given algorithm that enumerates a complete set of $\mathcal{T}$-unifiers for two arbitrary type expressions and construct an algorithm which enumerates a complete set of $\mathcal{T}$-unifiers for two arbitrary well-typed terms. We formulate the algorithm as a non-deterministic procedure for computing a $\mathcal{T}$-unifier for a given list of pairs of well-typed terms.

For that purpose we define a binary relation $\stackrel{unif}{\Longrightarrow}$ on pairs of the form (σ, E) where σ is a typed substitution and E is a list of appropriate equations, i.e., if $\sigma \in Sub_\Sigma(X, V, V')$ then E is a list of pairs of (Σ, X, V')-terms. We write $\langle t = t', E_r\rangle$ for an equation list where the pair (t, t') is the first equation and E_r is the list of the remaining equations. The relation $\stackrel{unif}{\Longrightarrow}$ is defined by the rules in figure 3.2. In the first rule (T) the result types of the left-hand side and the right-hand side of the first equation are unified by a $\mathcal{T}$-unifier, i.e., the result types are $\mathcal{T}$-equal after an application of this rule. $\mathcal{T}$-equality of these result types is a precondition for the applicability of the other rules. The rules (E1) and (E2) eliminate an equation containing a variable in one side. The typed substitution σ' in these elimination rules is well-defined since $t{:}\tau$ is well-typed by $\tau =_{\mathcal{T}} \tau'$ and lemma 3.7. The rule (D) decomposes an equation if the left-hand side and the right-hand side are compound terms with the same main functor and arity.

Let $\stackrel{unif}{\Longrightarrow}^{+}$ be the transitive closure of $\stackrel{unif}{\Longrightarrow}$. The result of unifying the (Σ, X, V)-terms t and t' is the set

$$Unif(t,t') := \{\ \sigma \mid (id_{X,V}, \langle t = t' \rangle) \stackrel{unif}{\Longrightarrow}^{+} (\sigma, \langle\rangle)\ \}$$

where $\langle\rangle$ denotes the empty list of equations.

Note that $\stackrel{unif}{\Longrightarrow}^{+}$ is an extension of Robinson's unification algorithm [Rob65] [BC83]: If one term is a variable which does not occur in the other term, then this variable is bound to the other term. If two composite terms have to be unified, then all corresponding components of the terms are unified. The only (but essential) difference is that the types of two terms are $\mathcal{T}$-unified before the terms will be unified.

We will show that $Unif(t,t')$ is a complete set of $\mathcal{T}$-unifiers for t and t'. First we show that there are no infinite chains in the computation of $Unif(t,t')$:

Lemma 3.24 *Let t and t' be (Σ, X, V)-terms. Then any sequence*

$$(id_{X,V}, \langle t = t' \rangle) \stackrel{unif}{\Longrightarrow} (\sigma_1, E_1) \stackrel{unif}{\Longrightarrow} (\sigma_2, E_2) \stackrel{unif}{\Longrightarrow} \cdots$$

terminates.

Proof: We define the complexity $\|t{:}\tau\|$ of a term $t{:}\tau$ by

- $\|x{:}\tau'\| := 1$ for all variables $x{:}\tau \in V$
- $\|f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau\| := \|t_1{:}\tau_1\| + \cdots + \|t_n{:}\tau_n\| + 1$ for all terms $f(t_1{:}\tau_1, \ldots, t_n{:}\tau_n){:}\tau \in Term_\Sigma(X, V)$ $(n \geq 0)$

and the type difference $tdiff(t{:}\tau, t'{:}\tau')$ of two terms by

$$tdiff(t{:}\tau, t'{:}\tau') := \begin{cases} 0 & \text{if } \tau =_{\mathcal{T}} \tau' \\ 1 & \text{otherwise} \end{cases}$$

The complexity of a list of equations $E = \langle t_1 = t_1', \ldots, t_k = t_k' \rangle$ is defined to be the triple

$$\|E\| := \left(\left| \bigcup_{i=1}^{k} uvar(t_i) \cup uvar(t_i') \right|, \sum_{i=1}^{k} \|t_i\| + \|t_i'\|, \sum_{i=1}^{k} tdiff(t_i, t_i') \right)$$

where $|\cdots|$ denotes the cardinality of a set. The lexicographic ordering on tuples of natural numbers is a noetherian ordering, i.e., there is no infinite sequence

$\langle e_1, e_2, e_3, \ldots\rangle$ with $e_i > e_{i+1}$. By definition of the relation $\stackrel{unif}{\Longrightarrow}$ it is clear that $\|E\| > \|E'\|$ if $(\sigma, E) \stackrel{unif}{\Longrightarrow} (\sigma', E')$ since the first rule decrements only the third component of the complexity, the rules (E1) and (E2) eliminate a variable which reduces the first component of the complexity, and the last rule decreases the second component of the complexity. Therefore any $\stackrel{unif}{\Longrightarrow}$-sequence terminates. ∎

In the proofs of the following lemmas we use the notion of $\mathcal{T}$-unifiers on lists of equations: If $\langle t_1 = t_1', \ldots, t_k = t_k'\rangle$ is a list of equations, then a $\mathcal{T}$-unifier for this list is a typed substitution that $\mathcal{T}$-unifies each pair $t_i = t_i'$ $(i = 1, \ldots, k)$. The next lemma shows the soundness of the $\mathcal{T}$-unification procedure:

Lemma 3.25 *Let t and t' be (Σ, X, V)-terms and $\sigma \in Unif(t, t')$. Then $\sigma \in SU_{\mathcal{T}}(t, t')$, i.e., σ is a $\mathcal{T}$-unifier for t and t'.*

Proof: Since $\sigma \in Unif(t, t')$, there is a sequence

$$(\sigma_0, E_0) \stackrel{unif}{\Longrightarrow} (\sigma_1, E_1) \stackrel{unif}{\Longrightarrow} \cdots \stackrel{unif}{\Longrightarrow} (\sigma_k, \langle\rangle)$$

where $\sigma_0 = id_{X,V}$ and $E_0 = \langle t = t'\rangle$. We show by induction on the elements of the sequence: For each $0 \le i \le k$ there exists a typed substitution ϕ_i with $\sigma_k = \phi_i \circ \sigma_i$ and $\phi_i \in SU_{\mathcal{T}}(E_i)$.

For $i = k$ we choose $\phi_k = id_{X,V'}$ (where $\sigma_k \in Sub_\Sigma(X, V, V')$). For the induction step we assume the existence of a typed substitution ϕ_i with $\sigma_k = \phi_i \circ \sigma_i$ and $\phi_i \in SU_{\mathcal{T}}(E_i)$. Since $(\sigma_{i-1}, E_{i-1}) \stackrel{unif}{\Longrightarrow} (\sigma_i, E_i)$, there are four possible cases:

1. Rule (T) has been applied in this step. Then $E_{i-1} = \langle t_1{:}\tau_1 = t_2{:}\tau_2, E_r\rangle$ and $\phi \in CSU_{\mathcal{T}}(\tau_1, \tau_2)$ with $\sigma_i = \phi \circ \sigma_{i-1}$ and $E_i = \langle \phi(t_1{:}\tau_1) = \phi(t_2{:}\tau_2), \phi(E_r)\rangle$. Hence $\sigma_k = \phi_i \circ \sigma_i = \phi_i \circ \phi \circ \sigma_{i-1} = \phi_{i-1} \circ \sigma_{i-1}$ with $\phi_{i-1} := \phi_i \circ \phi$. Moreover, if $l = r$ occurs in E_{i-1}, then $\phi(l) = \phi(r)$ occurs in E_i and ϕ_i is a $\mathcal{T}$-unifier for $\phi(l)$ and $\phi(r)$ which implies $\phi_{i-1} \in SU_{\mathcal{T}}(E_{i-1})$.

2. Rule (E1) has been applied in this step. Then $E_{i-1} = \langle x{:}\tau = t{:}\tau', E_r\rangle$ with $\tau =_{\mathcal{T}} \tau'$, $x \in Var$, $x \notin uvar(t{:}\tau')$, $\sigma_i = \sigma' \circ \sigma_{i-1}$ and $E_i = \sigma'(E_r)$ where $\sigma' = \{x{:}\tau / t{:}\tau\}$. Hence $\sigma_k = \phi_i \circ \sigma_i = \phi_i \circ \sigma' \circ \sigma_{i-1} = \phi_{i-1} \circ \sigma_{i-1}$ with $\phi_{i-1} := \phi_i \circ \sigma'$. Moreover,

$$\begin{aligned}
\phi_{i-1}(x{:}\tau) &= \phi_i \circ \sigma'(x{:}\tau) \\
&= \phi_i(t{:}\tau) \\
&=_{\mathcal{T}} \phi_i(t{:}\tau') \quad \text{(since } \tau =_{\mathcal{T}} \tau'\text{)} \\
&= \phi_i \circ \sigma'(t{:}\tau') \quad \text{(since } x \notin uvar(t{:}\tau')\text{)} \\
&= \phi_{i-1}(t{:}\tau')
\end{aligned}$$

If E_r contains an equation $l = r$, then $\sigma'(l) = \sigma'(r)$ occurs in E_i and ϕ_i is a $\mathcal{T}$-unifier for $\sigma'(l)$ and $\sigma'(r)$, i.e., ϕ_{i-1} is a $\mathcal{T}$-unifier for l and r. Thus $\phi_{i-1} \in SU_{\mathcal{T}}(E_{i-1})$.

3. The application of rule (E2) is symmetric to the previous case.

4. Rule (D) has been applied in this step. Then $E_{i-1} = \langle f(t_1, \ldots, t_n){:}\tau = f(t'_1, \ldots, t'_n){:}\tau', E_r\rangle$ with $\tau =_{\mathcal{T}} \tau'$, $\sigma_i = \sigma_{i-1}$ and $E_i = \langle t_1 = t'_1, \ldots, t_n = t'_n, E_r\rangle$. Hence $\sigma_k = \phi_i \circ \sigma_i = \phi_i \circ \sigma_{i-1}$. Moreover,

$$\begin{array}{lll} & \phi_i(f(t_1, \ldots, t_n){:}\tau) & \\ = & f(\phi_i(t_1), \ldots, \phi_i(t_n)){:}\phi_i(\tau) & \\ =_{\mathcal{T}} & f(\phi_i(t_1), \ldots, \phi_i(t_n)){:}\phi_i(\tau') & (\text{since } \tau =_{\mathcal{T}} \tau') \\ =_{\mathcal{T}} & f(\phi_i(t'_1), \ldots, \phi_i(t'_n)){:}\phi_i(\tau') & (\phi_i \in SU_{\mathcal{T}}(E_i) \text{ by ind. hypothesis}) \\ = & \phi_i(f(t'_1, \ldots, t'_n){:}\tau') & \end{array}$$

If E_r contains an equation $l = r$, then $l = r$ occurs also in E_i, i.e., ϕ_i is a $\mathcal{T}$-unifier for l and r. Thus $\phi_i \in SU_{\mathcal{T}}(E_{i-1})$.

We obtain for $i = 0$: $\sigma_k = \phi_0 \circ \sigma_0 = \phi_0$ and $\sigma_k \in SU_{\mathcal{T}}(t, t')$. ■

The next lemma shows the completeness of the $\mathcal{T}$-unification procedure:

Lemma 3.26 *Let t and t' be (Σ, X, V)-terms and $\theta \in SU_{\mathcal{T}}(t, t')$. Then there exists a typed substitution $\sigma \in Unif(t, t')$ such that $\theta =_{\mathcal{T}} \phi \circ \sigma$ for a typed substitution ϕ.*

Proof: First we prove the following proposition:

> Let $\sigma \in Sub_{\Sigma}(X, V, V')$ be a typed substitution, E be a non-empty list of (Σ, X, V')-equations, $\theta \in SU_{\mathcal{T}}(E)$. Then there exists a pair (σ', E') with $(\sigma, E) \overset{unif}{\Longrightarrow} (\sigma', E')$ and $\theta \circ \sigma =_{\mathcal{T}} \theta' \circ \sigma'$ for some typed substitution $\theta' \in SU_{\mathcal{T}}(E')$.

To prove this proposition we assume a $\mathcal{T}$-unifier θ for the non-empty list of equations E. We distinguish the following cases:

1. $E = \langle t_1{:}\tau_1 = t_2{:}\tau_2, E_r\rangle$ and not $\tau_1 =_{\mathcal{T}} \tau_2$. Since $\theta \in SU_{\mathcal{T}}(t_1{:}\tau_1, t_2{:}\tau_2)$, the restriction of θ on $T_{\mathcal{T}}(X)$ is also a $\mathcal{T}$-unifier for τ_1 and τ_2. Hence there exists a type substitution $\phi \in CSU_{\mathcal{T}}(\tau_1, \tau_2)$ with $\theta|_{T_{\mathcal{T}}(X)} =_{\mathcal{T}} \psi \circ \phi$ for some type substitution ψ. It is straightforward to extend ϕ and ψ to typed substitutions such that $\theta =_{\mathcal{T}} \psi \circ \phi$. Thus there is the following unification step by rule (T):

$$(\sigma, E) \overset{unif}{\Longrightarrow} (\phi \circ \sigma, \phi(E))$$

and $\theta \circ \sigma =_{\mathcal{T}} \psi \circ \phi \circ \sigma$. To see that $\psi \in SU_{\mathcal{T}}(\phi(E))$ assume an equation $l = r$ from E. Then

$$\psi(\phi(l)) \ =_{\mathcal{T}} \ \theta(l) \ =_{\mathcal{T}} \ \theta(r) \ =_{\mathcal{T}} \ \psi(\phi(l))$$

since θ is a $\mathcal{T}$-unifier for E.

2. $E = \langle t_1{:}\tau_1 = t_2{:}\tau_2, E_r\rangle$ and $\tau_1 =_{\mathcal{T}} \tau_2$. Then there are the following cases:

   (a) $t_1 \in Var$: Since θ is a $\mathcal{T}$-unifier for $t_1{:}\tau_1$ and $t_2{:}\tau_2$, $t_1 \notin uvar(t_2{:}\tau_2)$ and thus there is the following unification step:

   $$(\sigma, E) \stackrel{unif}{\Longrightarrow} (\sigma' \circ \sigma, \sigma'(E_r))$$

   where σ' is the typed substitution $\{t_1{:}\tau_1/t_2{:}\tau_1\}$. It is easy to show that $\theta \circ \sigma' =_{\mathcal{T}} \theta$ and therefore $\theta \circ \sigma \ =_{\mathcal{T}} \ \theta \circ \sigma' \circ \sigma$. To see that $\theta \in SU_{\mathcal{T}}(\sigma'(E_r))$ assume an equation $l = r$ from E_r. Then

   $$\theta(\sigma'(l)) \ =_{\mathcal{T}} \ \theta(l) \ =_{\mathcal{T}} \ \theta(r) \ =_{\mathcal{T}} \ \theta(\sigma'(l))$$

   since θ is a $\mathcal{T}$-unifier for E.

   (b) $t_2 \in Var$: This is symmetric to the previous case.

   (c) $t_1{:}\tau_1 = f(r_1, \ldots, r_n){:}\tau_1$ and $t_2{:}\tau_2$ is not a typed variable: Since θ is a $\mathcal{T}$-unifier for $t_1{:}\tau_1$ and $t_2{:}\tau_2$, it must be $t_2{:}\tau_2 = f(r_1', \ldots, r_n'){:}\tau_2$. Then there is the following unification step:

   $$(\sigma, E) \stackrel{unif}{\Longrightarrow} (\sigma, \langle r_1 = r_1', \ldots, r_n = r_n', E_r\rangle)$$

   θ is a $\mathcal{T}$-unifier for all equations $r_i = r_i'$ and for E_r since θ is a $\mathcal{T}$-unifier for E.

Hence the proposition is true. Let θ be a $\mathcal{T}$-unifier for the (Σ, X, V)-terms t and t'. By the above proposition, there is a sequence

$$(id_{X,V}, \langle t = t'\rangle) \stackrel{unif}{\Longrightarrow} (\sigma_1, E_1) \stackrel{unif}{\Longrightarrow} (\sigma_2, E_2) \stackrel{unif}{\Longrightarrow} \cdots$$

with $\theta =_{\mathcal{T}} \theta \circ id_{X,V} =_{\mathcal{T}} \theta_1 \circ \sigma_1 =_{\mathcal{T}} \theta_2 \circ \sigma_2 =_{\mathcal{T}} \cdots$ for some typed substitutions $\theta_1, \theta_2, \ldots$ Since all $\stackrel{unif}{\Longrightarrow}$-sequences are finite (lemma 3.24), there must be a last element $(\sigma_k, \langle\rangle)$ in the sequence. Thus $\sigma_k \in Unif(t, t')$ and $\theta =_{\mathcal{T}} \theta_k \circ \sigma_k$. ■

Theorem 3.27 ($\mathcal{T}$-unification) *Let t and t' be (Σ, X, V)-terms. Then $Unif(t, t')$ is a complete set of $\mathcal{T}$-unifiers.*

Proof: $Unif(t,t') \subseteq SU_{\mathcal{T}}(t,t')$ follows from lemma 3.25 and completeness follows from lemma 3.26. ■

Example 3.28 Consider the polymorphic signature of example 3.2. The terms $\mathtt{0}{:}zero$ and $\mathtt{N}{:}nat(\alpha)$ should be unified by our unification procedure. First, the types of terms $zero$ and $nat(\alpha)$ are $\mathcal{T}$-unified and the result is the $\mathcal{T}$-unifier $\{\alpha/zero\}$. Then `N` is bound to `0` and the result is the $\mathcal{T}$-unifier $\{\alpha/zero,\ \mathtt{N}{:}nat(\alpha)/\mathtt{0}{:}nat(zero)\}$. For the unification of the terms $\mathtt{s(N1}{:}nat(posint)\mathtt{)}{:}posint$ and $\mathtt{s(N2}{:}nat(\alpha)\mathtt{)}{:}nat(posint)$ the following steps are performed:

- The types $posint$ and $nat(posint)$ are $\mathcal{T}$-equal and need not be unified.
- By the decomposition rule, the terms $\mathtt{N1}{:}nat(posint)$ and $\mathtt{N2}{:}nat(\alpha)$ are unified in the next unification step.
- The types $nat(posint)$ and $nat(\alpha)$ are $\mathcal{T}$-unified. The result is the type substitution $\{\alpha/posint\}$.
- `N2` is bound to `N1` (or vice versa). Thus the complete result of the unification is the typed substitution

$$\{\alpha/posint,\ \mathtt{N2}{:}nat(\alpha)/\mathtt{N1}{:}nat(posint)\}$$

Example 3.29 Consider the following type specification $\mathcal{T}$:

$$\begin{array}{llcl}
\mathtt{TYPEOPS} & s_0: & & \rightarrow\ type \\
& s_1: & type & \rightarrow\ type \\
& s_2: & type & \rightarrow\ type \\
\mathtt{TYPEAXIOMS} & s_1(s_0) & = & s_0 \\
& s_2(s_0) & = & s_0
\end{array}$$

Thus s_0 is a common subtype of s_1 and s_2. The unification of the typed terms $\mathtt{X}{:}s_1(\alpha)$ and $\mathtt{Y}{:}s_2(\beta)$ requires a $\mathcal{T}$-unifier for the type expressions $s_1(\alpha)$ and $s_2(\beta)$ which can be computed by the narrowing procedure (see remarks at the end of Section 3.6). Hence the type substitution $\{\alpha/s_0, \beta/s_0\}$ is a $\mathcal{T}$-unifier for the type expressions $s_1(\alpha)$ and $s_2(\beta)$ and the typed substitution

$$\{\alpha/s_0,\ \beta/s_0,\ \mathtt{X}{:}s_1(\alpha)/\mathtt{Y}{:}s_1(s_0)\}$$

is a $\mathcal{T}$-unifier for the terms $\mathtt{X}{:}s_1(\alpha)$ and $\mathtt{Y}{:}s_2(\beta)$. Therefore the variables `X` and `Y` are constrained to the common subsort s_0 by the unification procedure (note the analogy to order-sorted unification [SNGM89]).

In the next section we will see that resolution is a sound and complete proof procedure for typed logic programs if the unification procedure used in the resolution steps computes a complete set of $\mathcal{T}$-unifiers. Therefore the unification procedure presented in this section gives us some information about the rôle of different type systems for logic programming. We have seen that the classical unification algorithm of Robinson can be adapted to the typed framework if the types of terms are unified before unifying the terms. Hence our unification procedure shows that the decidability of the typed unification problem is dependent on the decidability of the unification problem in the type theory: If it is decidable whether two types are unifiable w.r.t. the type specification $\mathcal{T}$, then the unification problem for typed terms w.r.t. $\mathcal{T}$ is also decidable because all $\stackrel{unif}{\Longrightarrow}$-sequences terminate (lemma 3.24) and unifiable terms can always be derived to an empty equation list (lemma 3.26). Moreover, different type structures influence the complexity of the unification procedure. For the general case a complex procedure for the unification of type terms w.r.t. the equational type specification is necessary. But for simpler type structures a less complex unification procedure may be sufficient:

- If the type structure is *many-sorted* without overloading, i.e., there are only basic types and no equations in the type structure and there is exactly one type declaration for each function and predicate symbol, then all types can be omitted while unifying two terms or atoms since two composite terms or atoms with the same functor or predicate, respectively, have always the same type.

- If the type structure is *polymorphic* without any equations between types, then the $\mathcal{T}$-unifier for two types is the unifier of the type expressions in the free type term algebra. Hence there exists a most general unifier for two unifiable type terms which can be computed by Robinson's unification algorithm. This implies the existence of a most general unifier for two $\mathcal{T}$-unifiable typed terms and Robinson's unification algorithm can be used as a $\mathcal{T}$-unification procedure on typed terms if type expressions are represented as first-order terms (cf. [Han89a]). Moreover, if the polymorphic signature and the typed program satisfy some additional restrictions, it has been shown that such programs are executable without any type information at run time [Han89b]. The type system of Mycroft and O'Keefe [MO84] is a special case of a polymorphic type structure.

- If the type structure is *order-sorted*, i.e., the type specification contains equations between types, then there does not exist a most general $\mathcal{T}$-unifier for any two type expressions. Hence the $\mathcal{T}$-unification procedure on typed terms must compute complete sets of $\mathcal{T}$-unifiers. Nevertheless, for practical applications it is desirable that the complete sets of $\mathcal{T}$-unifiers are finite which depends on the type specification. Criteria for finitary or unitary order-sorted unification can be found in [Wal89]. An overview of unification in equational theories can be found in [SS82].

- For *polymorphically order-sorted* type structures a full unification procedure for the equational type theory is necessary. Nevertheless, Smolka [Smo89] has shown that there are also restricted classes of polymorphically order-sorted typed logic programs where more efficient unification procedures exist.

From a conceptual point of view our unification procedure shows up the influence of types in logic programming. But for an efficient operational semantics it is necessary to omit type information at run time whenever it is possible. In [Han89a] and [Han89b] it is shown how this could be done in the polymorphic case. Similar results for the general case are a topic for further research.

3.6 Resolution

The resolution principle in untyped Horn logic (see [Rob65]) can be used as a proof procedure for typed Horn clause programs if the untyped unification is replaced by the $\mathcal{T}$-unification as defined in the last section. We call a Σ-clause a **variant** of another Σ-clause if it is obtained by replacing type variables and typed variables by other type variables and typed variables, respectively, such that different variables are replaced by new different variables. Let (Σ, C) be a typed logic program.

a) Let G be a (Σ, X, V)-goal and the (Σ, X, V)-clause $L' \leftarrow G'$ be a variant of a clause from C with $tvar(G) \cap tvar(L' \leftarrow G') = \emptyset$ and $uvar(G) \cap uvar(L' \leftarrow G') = \emptyset$. If there exists a $\mathcal{T}$-unifier $\sigma \in Sub_\Sigma(X, V, V')$ for an atom $L \in G$ and L', then $\sigma(G - \{L\}) \cup \sigma(G')$ is said to be **derived by $\mathcal{T}$-resolution** from G relative to σ and $L' \leftarrow G'$. Notation:

$$(\Sigma, C, V) \;\; G \;\; \models_{\mathrm{R}} \sigma \;\; \sigma(G - \{L\}) \cup \sigma(G')$$

b) Let G_0 be a (Σ, X, V_0)-goal. A **(Σ, C, V_0)-resolution** or **$\mathcal{T}$-resolution** of G_0 is a sequence of the form

$$(\Sigma, C, V_0) \;\; G_0 \;\; \models_{\mathrm{R}} \sigma_1 \;\; G_1 \;\; \models_{\mathrm{R}} \sigma_2 \;\; G_2 \;\; \models_{\mathrm{R}} \;\; \cdots \;\; \models_{\mathrm{R}} \sigma_n \;\; G_n$$

where $(\Sigma, C, V_i) \;\; G_i \;\; \models_{\mathrm{R}} \sigma_{i+1} \;\; G_{i+1}$ with $\sigma_{i+1} \in Sub_\Sigma(X, V_i, V_{i+1})$ for $i = 0, 1, 2, \ldots, n-1$. The (Σ, C, V_0)-resolution is called **successful** if $G_n = \emptyset$. In this case n is called the length of the (Σ, C, V_0)-resolution and $\sigma := \sigma_n \circ \cdots \circ \sigma_1$ is called a **computed answer**. Notation:

$$(\Sigma, C, V_0) \;\; \models_{\mathrm{R}} \sigma \;\; G_0$$

If we replace the requirement for a $\mathcal{T}$-unifier for L and L' by the condition "$\sigma \in CSU_{\mathcal{T}}(L, L')$", then the resolution is called a ***CSU*$_{\mathcal{T}}$-resolution** (*resolution with complete sets of $\mathcal{T}$-unifiers*) and the symbol $\models_{\mathrm{R}}$ is replaced by $\models_{\mathrm{RC}}$. If we drop the requirement for disjoint sets of type variables and typed variables in the goal and the applied clause, we call the resolution **unrestricted** and replace the symbol $\models_{\mathrm{R}}$ by $\models_{\mathrm{UR}}$.

The soundness of $\mathcal{T}$-resolution can be directly proved:

Theorem 3.30 (Soundness of $\mathcal{T}$-resolution) *Let (Σ, C) be a typed logic program and G be a (Σ, X, V)-goal. If there is a successful $\mathcal{T}$-resolution $(\Sigma, C, V) \models_{\mathrm{R}} \sigma\ G$ with computed answer $\sigma \in Sub_\Sigma(X, V, V')$, then $(\Sigma, C, V') \models \sigma(G)$.*

Proof: By induction on the length n of a successful (Σ, C, V)-resolution:
$n = 1$: Then there is a (Σ, C, V)-resolution

$$(\Sigma, C, V) \quad G \quad \models_{\mathrm{R}}\sigma \quad \emptyset$$

where $\sigma \in Sub_\Sigma(X, V, V')$, i.e., G is a (Σ, C, V)-atom. By definition of $\mathcal{T}$-resolution, there exists a variant $L' \leftarrow$ of a clause from C with $\sigma(L') =_{\mathcal{T}} \sigma(G)$. Hence there exist V'', a (Σ, X, V'')-clause $L'' \leftarrow$ from C and $\sigma'' \in Sub_\Sigma(X, V'', V)$ with $\sigma''(L'') = L'$. By lemma 3.13, $(\Sigma, C, V') \models \sigma(L')$ since $\sigma(L') = \sigma \circ \sigma''(L'')$ and $(\Sigma, C, V'') \models L'$. Hence $(\Sigma, C, V') \models \sigma(G)$ by lemma 3.18.
$n > 1$: Then there is a (Σ, C, V)-resolution

$$(\Sigma, C, V) \quad G \quad \models_{\mathrm{R}}\sigma_1 \quad G_1 \quad \models_{\mathrm{R}}\sigma_2 \quad G_2 \quad \models_{\mathrm{R}} \quad \cdots \quad \models_{\mathrm{R}}\sigma_n \quad \emptyset$$

with $\sigma = \sigma_n \circ \cdots \circ \sigma_1 \in Sub_\Sigma(X, V, V')$. By definition of $\mathcal{T}$-resolution, there exists a variant $L' \leftarrow G'$ of a clause from C with $\sigma_1(L') =_{\mathcal{T}} \sigma_1(L_0)$ where $G = G_0 \cup \{L_0\}$. Let $\sigma_1 \in Sub(X, V, V_1)$. Then

$$(\Sigma, C, V_1) \quad \sigma_1(G') \cup \sigma_1(G_0) \quad \models_{\mathrm{R}}\sigma_2 \quad G_2 \quad \models_{\mathrm{R}} \quad \cdots \quad \models_{\mathrm{R}}\sigma_n \quad \emptyset$$

is a (Σ, C, V_1)-resolution of length $n - 1$. By induction hypothesis, $(\Sigma, C, V') \models \sigma(G') \cup \sigma(G_0)$. Since $L' \leftarrow G'$ is a variant of a clause from C, there exist V'', a (Σ, X, V'')-clause $L'' \leftarrow G'' \in C$ and $\sigma'' \in Sub_\Sigma(X, V'', V)$ with $\sigma''(L'' \leftarrow G'') = L' \leftarrow G'$. By lemma 3.13, $(\Sigma, C, V') \models \sigma(L' \leftarrow G')$. From the fact $(\Sigma, C, V') \models \sigma(G')$ we infer $(\Sigma, C, V') \models \sigma(L')$. Lemma 3.17 and lemma 3.18 yield $(\Sigma, C, V') \models \sigma(L_0)$. Hence we have $(\Sigma, C, V') \models \sigma(G)$. ■

The completeness of resolution in untyped Horn logic can be proved by a fixpoint theorem using a transformation on Herbrand interpretations [vEK76] [Llo87]. In [Han91] this proof method is adapted to polymorphic logic programs. In this chapter we will show the completeness of $\mathcal{T}$-resolution for typed logic programs by simulating each deduction in the typed Horn clause calculus by $\mathcal{T}$-resolution. [Pad88] has presented such a proof for many-sorted Horn clause logic with equality, but he has required that all types are interpreted as non-empty sets. This simplifies the proof but is not reasonable in our context.

In the rest of this section we assume that (Σ, C) is a typed logic program. A few technical lemmas will help to structure the completeness proof. The first lemma shows that the substitution rule is not necessary if $\widehat{C}$ (the set of instantiated clauses) is used in a deduction.

Lemma 3.31 *Let $(\Sigma, C, V) \vdash L \leftarrow G$. Then for any typed substitution $\sigma \in Sub_\Sigma(X, V, V')$ there exists a deduction for $(\Sigma, \widehat{C}, V') \vdash \sigma(L \leftarrow G)$ where only axioms and cut rules are applied.*

Proof: We prove the lemma by induction on the number n of cut rule applications in a shortest deduction of $(\Sigma, C, V) \vdash L \leftarrow G$. The case $n = 0$ is trivial since $\sigma_0(L_0) \leftarrow \sigma_0(G_0) \in \widehat{C}$ for all $L_0 \leftarrow G_0 \in C$ and all appropriate typed substitutions σ_0. Otherwise there is a last application of the cut rule in the deduction, say

$$(\Sigma, C, V_i) \vdash L_i \leftarrow G_i \cup \{L_i'\} \quad \text{and} \quad (\Sigma, C, V_i) \vdash L_j \leftarrow G_j \quad \text{with} \quad L_i' =_{\mathcal{T}} L_j$$

occur in the deduction before the last application of the cut rule. Let $\sigma_1 \in Sub_\Sigma(X, V_i, V_i')$. We have to show that $(\Sigma, \widehat{C}, V_i') \vdash \sigma_1(L_i) \leftarrow \sigma_1(G_i \cup G_j)$ can be deduced without an application of the substitution rule. The number of cut rule applications in shortest derivations of

$$(\Sigma, C, V_i) \vdash L_i \leftarrow G_i \cup \{L_i'\} \qquad \text{and} \qquad (\Sigma, C, V_i) \vdash L_j \leftarrow G_j$$

is less than n. By induction hypothesis,

$$(\Sigma, \widehat{C}, V_i') \vdash \sigma_1(L_i) \leftarrow \sigma_1(G_i \cup \{L_i'\}) \qquad \text{and} \qquad (\Sigma, \widehat{C}, V_i') \vdash \sigma_1(L_j) \leftarrow \sigma_1(G_j)$$

can be deduced without an application of the substitution rule. Lemma 3.17 yields $\sigma_1(L_i') =_{\mathcal{T}} \sigma_1(L_j)$. By an application of the cut rule, we obtain

$$(\Sigma, \widehat{C}, V_i') \vdash \sigma_1(L_i) \leftarrow \sigma_1(G_i \cup G_j)$$

This proves the lemma. ■

Lemma 3.32 *If $(\Sigma, C, V) \vdash L \leftarrow G$ where only axioms and cut rules are applied, then $(\Sigma, C', V) \models_{\text{UR}} id_{X,V}\ L'$ for all (Σ, X, V)-atoms $L' =_{\mathcal{T}} L$ where $C' = C \cup \{P \leftarrow \mid P \in G\}$, and each substitution in the $\mathcal{T}$-resolution is equal to $id_{X,V}$.*

Proof: The lemma is proved by induction on the length of the deduction. Let $d_1, \ldots, d_n$ be a deduction for $(\Sigma, C, V) \vdash L \leftarrow G$ where only axioms and cut rules are applied and L' be a (Σ, X, V)-atom with $L' =_{\mathcal{T}} L$.

If $L \leftarrow G \in C$, then $(\Sigma, C', V)\ L' \mid\!\!\frac{}{\text{UR}}\, id_{X,V}\ G$ is a $\mathcal{T}$-resolution step since $id_{X,V}(L') =_{\mathcal{T}} id_{X,V}(L)$. If G consists of k Σ-atoms, then we achieve the empty goal with k further unrestricted $\mathcal{T}$-resolution steps with substitutions $id_{X,V}$.

If $L \leftarrow G \notin C$, then the clause must be derived by an application of the cut rule, i.e., there are

$$\begin{aligned} d_i &= (\Sigma, C, V) \vdash L \leftarrow G_0 \cup \{L_0\} \\ d_j &= (\Sigma, C, V) \vdash L_1 \leftarrow G_1 \end{aligned}$$

with $L_0 =_{\mathcal{T}} L_1$, $G = G_0 \cup G_1$ and $i, j < n$. By induction hypothesis,

$$(\Sigma, C' \cup \{L_0 \leftarrow\}, V) \ \mid\!\!\frac{}{\text{UR}}\, id_{X,V} \ \ L' \quad \text{for all } L' =_{\mathcal{T}} L \tag{1}$$

and

$$(\Sigma, C', V) \ \mid\!\!\frac{}{\text{UR}}\, id_{X,V} \ \ L'_1 \quad \text{for all } L'_1 =_{\mathcal{T}} L_1 \tag{2}$$

since $G = G_0 \cup G_1$. If the clause $L_0 \leftarrow$ is used in resolution (1), then, by (2), it is possible to replace the resolution step by a sequence of resolution steps that derives L_0 to the empty goal using clauses from C'. Thus $(\Sigma, C', V) \mid\!\!\frac{}{\text{UR}}\, id_{X,V}\ L'$ for all $L' =_{\mathcal{T}} L$ and each substitution in this $\mathcal{T}$-resolution is equal to $id_{X,V}$. ■

Now we can prove the completeness of $\mathcal{T}$-resolution:

Theorem 3.33 (Completeness of $\mathcal{T}$-resolution for atoms) *Let V, V' be finite sets of typed variables and A be a (Σ, X, V)-atom. If $\sigma \in Sub_\Sigma(X, V, V')$ is a typed substitution with $(\Sigma, C, V') \models \sigma(A)$, then there exists a set of typed variables V_0 and a typed substitution $\sigma_0 \in Sub_\Sigma(X, V_0, V')$ with $(\Sigma, C, V_0) \mid\!\!\frac{}{\text{R}}\, \sigma_0\ A$ and $\sigma_0(A) = \sigma(A)$.*

Proof: W.l.o.g. we assume that σ affects only a finite number of type variables since V is finite, i.e., the type domain $tdom(\sigma)$ is finite. Let $(\Sigma, C, V') \models \sigma(A)$. By theorem 3.21, there exists a (Σ, X, V)-atom A' with $A' =_{\mathcal{T}} \sigma(A)$ and $(\Sigma, C, V') \vdash A'$. By lemma 3.31 and lemma 3.32, there exists a successful unrestricted $\mathcal{T}$-resolution of the form

$$(\Sigma, \widehat{C}, V') \ \ \sigma(A) \ \mid\!\!\frac{}{\text{UR}}\, id_{X,V'} \ \ G_1 \ \ \mid\!\!\frac{}{\text{UR}}\, id_{X,V'} \ \ \cdots \ \ \mid\!\!\frac{}{\text{UR}}\, id_{X,V'} \ \ \emptyset$$

In the first resolution step there exist $L_0 \leftarrow R_0 \in C$, V'_0 and $\sigma_0 \in Sub_\Sigma(X, V'_0, V')$ with $\sigma_0(L_0) =_{\mathcal{T}} \sigma(A)$ and $\sigma_0(R_0) = G_1$.

W.l.o.g. we assume $(tdom(\sigma) \cup tvar(A)) \cap tvar(L_0 \leftarrow R_0) = \emptyset$ and $uvar(V) \cap uvar(V'_0) = \emptyset$ (otherwise we choose an appropriate variant of $L_0 \leftarrow R_0$ and an appropriate typed substitution σ_0). We define $V_0 := V \cup var(L_0 \leftarrow R_0)$ and combine σ and σ_0 into a typed substitution $\sigma_1 \in Sub_\Sigma(X, V_0, V')$ with

$$\sigma_1(\alpha) \;=\; \begin{cases} \sigma(\alpha) & \text{if } \alpha \in tdom(\sigma) \cup tvar(A) \\ \sigma_0(\alpha) & \text{otherwise} \end{cases}$$

and

$$\sigma_1(x{:}\tau) \;=\; \begin{cases} \sigma(x{:}\tau) & \text{if } x{:}\tau \in V \\ \sigma_0(x{:}\tau) & \text{if } x{:}\tau \in var(L_0 \leftarrow R_0) \end{cases}$$

Then $\sigma_1(A) = \sigma(A) =_{\mathcal{T}} \sigma_0(L_0) = \sigma_1(L_0)$ and $\sigma_1(R_0) = \sigma_0(R_0) = G_1$. Therefore

$$(\Sigma, C, V_0) \;\; A \;\; \models_{\mathrm{R}} \sigma_1 \;\; G_1$$

is a $\mathcal{T}$-resolution step. If $G_1 = \emptyset$, then the proof is finished, otherwise there is a second resolution step

$$(\Sigma, \widehat{C}, V') \;\; G_1 \;\; \models_{\mathrm{UR}} id_{X,V'} \;\; G_2$$

Let $L_1' \leftarrow R_1' \in \widehat{C}$ be the clause used in this resolution step, i.e., there exist $L_1 \leftarrow R_1 \in C$, V_1' and $\sigma_1' \in Sub_\Sigma(X, V_1', V')$ with $\sigma_1'(L_1 \leftarrow R_1) = L_1' \leftarrow R_1'$. Similarly to the first resolution step, we combine σ_1' and $id_{X,V'}$ into a typed substitution $\sigma_2 \in Sub_\Sigma(X, V_1, V')$, where $V_1 := V' \cup var(L_1 \leftarrow R_1)$, such that

$$(\Sigma, C, V_1) \;\; G_1 \;\; \models_{\mathrm{R}} \sigma_2 \;\; G_2$$

is a $\mathcal{T}$-resolution step. Since $V' \subseteq V_1$, we can extend σ_1 to a typed substitution $\sigma_1 \in Sub_\Sigma(X, V_0, V_1)$. Hence we obtain the $\mathcal{T}$-resolution

$$(\Sigma, C, V_0) \;\; A \;\; \models_{\mathrm{R}} \sigma_1 \;\; G_1 \;\; \models_{\mathrm{R}} \sigma_2 \;\; G_2$$

with $\sigma_2(\sigma_1(A)) = \sigma_2(\sigma(A)) = \sigma(A)$ and $\sigma_2 \circ \sigma_1 \in Sub_\Sigma(X, V_0, V')$. If we apply the transformation of the second resolution step in the same way to the remaining resolution steps, we obtain a $\mathcal{T}$-resolution

$$(\Sigma, C, V_0) \;\; A \;\; \models_{\mathrm{R}} \sigma_1 \;\; G_1 \;\; \models_{\mathrm{R}} \sigma_2 \;\; \cdots \;\; \models_{\mathrm{R}} \sigma_n \;\; \emptyset$$

with $\sigma_n \circ \cdots \circ \sigma_1(A) = \sigma(A)$ and $\sigma_n \circ \cdots \circ \sigma_1 \in Sub_\Sigma(X, V_0, V')$. ■

We need the next lemma to prove the completeness of $\mathcal{T}$-resolution for general goals:

Lemma 3.34 *Let G be a (Σ, X, V)-goal with $var(G) = \{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}$. Let p be a new symbol that does not occur in Σ, $\Sigma' := (H, Func, Pred \cup \{p{:}\tau_1, \ldots, \tau_n\})$, $L := p(x_1{:}\tau_1, \ldots, x_n{:}\tau_n)$ and $C' := C \cup \{L \leftarrow G\}$. Then*

$$(\Sigma, C, V') \models \sigma(G) \qquad \Longrightarrow \qquad (\Sigma', C', V') \models \sigma(L)$$

for all $\sigma \in Sub_{\Sigma'}(X, V, V')$.

Proof: Let $(\Sigma, C, V') \models \sigma(G)$ and M' be a model for (Σ', C'). Then M' is also a model for (Σ', C) and $M', var(L \leftarrow G) \models L \leftarrow G$. By lemma 3.13, $M', V' \models \sigma(L) \leftarrow \sigma(G)$. Suppose v is a variable assignment for (X, V') in M'. M' is also a model for (Σ, C) if we omit the interpretation of the predicate symbol p in M'. Therefore $M', v \models \sigma(G)$. $M', v \models \sigma(L) \leftarrow \sigma(G)$ implies $M', v \models \sigma(L)$. Hence we obtain $M', V' \models \sigma(L)$. ∎

Theorem 3.35 (Completeness of $\mathcal{T}$-resolution) *Let V be a finite set of typed variables and G be a (Σ, X, V)-goal. If $\sigma \in Sub_\Sigma(X, V, V')$ is a typed substitution with $(\Sigma, C, V') \models \sigma(G)$, then there exist a set of typed variables V_0 and a typed substitution $\sigma_0 \in Sub_\Sigma(X, V_0, V')$ with $(\Sigma, C, V_0) \models_{\mathrm{R}} \sigma_0\ G$ and $\sigma_0(G) = \sigma(G)$.*

Proof: Let $var(G) = \{x_1{:}\tau_1, \ldots, x_n{:}\tau_n\}$ and p, L, Σ' and C' be defined as in the last lemma. $(\Sigma, C, V') \models \sigma(G)$ implies $(\Sigma', C', V') \models \sigma(L)$. By theorem 3.33, there exist V_0 and a typed substitution $\sigma_0 \in Sub_\Sigma(X, V_0, V')$ with $(\Sigma', C', V_0) \models_{\mathrm{R}} \sigma_0\ L$ and $\sigma_0(L) = \sigma(L)$. Since the only clause for the elimination of an atom with predicate symbol p is $L \leftarrow G$, there is a resolution

$$(\Sigma, C', V_0) \quad L \quad \models_{\mathrm{R}} \sigma_1 \quad \sigma_1(G) \quad \models_{\mathrm{R}} \sigma_2 \quad G_2 \quad \cdots \quad \models_{\mathrm{R}} \sigma_n \quad \emptyset$$

with $\sigma_0 = \sigma_n \circ \cdots \circ \sigma_1$. We can combine the typed substitution σ_1 with the typed substitution σ_2 in the second resolution step and obtain a (Σ, C, V_0)-resolution for G with the same computed answer. ∎

We need the following lemma to prove completeness of $CSU_\mathcal{T}$-resolution:

Lemma 3.36 (CSU-lemma) *If there is a $\mathcal{T}$-resolution*

$$(\Sigma, C, V) \quad G \quad \models_{\mathrm{R}} \sigma_1 \quad G_1 \quad \models_{\mathrm{R}} \sigma_2 \quad G_2 \quad \models_{\mathrm{R}} \quad \cdots \quad \models_{\mathrm{R}} \sigma_n \quad \emptyset$$

for the (Σ, X, V)-goal G, then there exists a $CSU_\mathcal{T}$-resolution

$$(\Sigma, C, V) \quad G \quad \models_{\mathrm{RC}} \sigma'_1 \quad G'_1 \quad \models_{\mathrm{RC}} \sigma'_2 \quad G'_2 \quad \models_{\mathrm{RC}} \quad \cdots \quad \models_{\mathrm{RC}} \sigma'_n \quad \emptyset$$

where $\sigma'_n \circ \cdots \circ \sigma'_1 \in Sub_\Sigma(X, V, V')$. Furthermore, there exists a typed substitution $\phi \in Sub_\Sigma(X, V', V'')$ with $\phi \circ \sigma'_n \circ \cdots \circ \sigma'_1 =_\mathcal{T} \sigma_n \circ \cdots \circ \sigma_1$.

Proof: By induction on the length n of the $\mathcal{T}$-resolution:
If $n = 1$, then $(\Sigma, C, V)\ G \models_{\mathrm{R}} \sigma_1\ \emptyset$. Hence there exists a variant $L \leftarrow \emptyset$ of a clause from C with $\sigma_1(G) =_\mathcal{T} \sigma_1(L)$. By definition of complete sets of $\mathcal{T}$-unifiers, there exist a unifier $\sigma'_1 \in CSU_\mathcal{T}(G, L)$ with $\sigma'_1 \in Sub_\Sigma(X, V, V')$ and a typed substitution $\phi \in Sub_\Sigma(X, V', V'')$ with $\phi \circ \sigma'_1 =_\mathcal{T} \sigma_1$. Thus $(\Sigma, C, V)\ G \models_{\mathrm{RC}} \sigma'_1\ \emptyset$ is a $CSU_\mathcal{T}$-resolution for G.

If $n > 1$, then there is a $\mathcal{T}$-resolution

$$(\Sigma, C, V) \ \ G \ \ \models_{\mathrm{R}} \sigma_1 \ \ G_1 \ \ \models_{\mathrm{R}} \sigma_2 \ \ G_2 \ \ \models_{\mathrm{R}} \ \ \cdots \ \ \models_{\mathrm{R}} \sigma_n \ \ \emptyset$$

Hence there exists a variant $L' \leftarrow G'$ of a clause from C with $\sigma_1(L') =_{\mathcal{T}} \sigma_1(L)$ where $G = G_0 \cup \{L\}$. By definition of $CSU_{\mathcal{T}}$, there exist a unifier $\sigma_1' \in CSU_{\mathcal{T}}(L', L)$ with $\sigma_1' \in Sub_{\Sigma}(X, V, V')$ and a typed substitution $\phi \in Sub_{\Sigma}(X, V', V'')$ with $\phi \circ \sigma_1' =_{\mathcal{T}} \sigma_1$. If $G_1' := \sigma_1'(G_0 \cup G')$, then

$$(\Sigma, C, V) \ \ G \ \ \models_{\mathrm{RC}} \sigma_1' \ \ G_1' \ \ \models_{\mathrm{R}} \sigma_2 \circ \phi \ \ G_2''$$

is a $\mathcal{T}$-resolution with $G_2'' =_{\mathcal{T}} G_2$ (w.l.o.g. we assume that ϕ does not alter any type variables or typed variables from the clause used in the second resolution step). Since

$$(\Sigma, C, V'') \ \ G_2 \ \ \models_{\mathrm{R}} \sigma_3 \ \ \cdots \ \ \models_{\mathrm{R}} \sigma_n \ \ \emptyset$$

is a $\mathcal{T}$-resolution for G_2 and $G_2 =_{\mathcal{T}} G_2''$, it is clear from the definition of $\mathcal{T}$-resolution that there exists a $\mathcal{T}$-resolution

$$(\Sigma, C, V'') \ \ G_2'' \ \ \models_{\mathrm{R}} \sigma_3 \ \ \cdots \ \ \models_{\mathrm{R}} \sigma_n \ \ \emptyset$$

for G_2'' of the same length and with the same $\mathcal{T}$-unifiers. Hence

$$(\Sigma, C, V') \ \ G_1' \ \ \models_{\mathrm{R}} \sigma_2 \circ \phi \ \ G_2'' \ \ \models_{\mathrm{R}} \sigma_3 \ \ \cdots \ \ \models_{\mathrm{R}} \sigma_n \ \ \emptyset$$

is a $\mathcal{T}$-resolution for G_1' of length $n - 1$. By induction hypothesis, there exists a $CSU_{\mathcal{T}}$-resolution

$$(\Sigma, C, V') \ \ G_1' \ \ \models_{\mathrm{RC}} \sigma_2' \ \ G_2' \ \ \models_{\mathrm{RC}} \ \ \cdots \ \ \models_{\mathrm{RC}} \sigma_n' \ \ \emptyset$$

where $\sigma_n' \circ \cdots \circ \sigma_2' \in Sub_{\Sigma}(X, V', V_1)$, and there exists a typed substitution $\rho \in Sub_{\Sigma}(X, V_1, V_2)$ with $\rho \circ \sigma_n' \circ \cdots \circ \sigma_2' =_{\mathcal{T}} \sigma_n \circ \cdots \circ \sigma_2 \circ \phi$. Hence we obtain a $CSU_{\mathcal{T}}$-resolution

$$(\Sigma, C, V) \ \ G \ \ \models_{\mathrm{RC}} \sigma_1' \ \ G_1' \ \ \models_{\mathrm{RC}} \sigma_2' \ \ G_2' \ \ \models_{\mathrm{RC}} \ \ \cdots \ \ \models_{\mathrm{RC}} \sigma_n' \ \ \emptyset$$

where $\sigma_n' \circ \cdots \circ \sigma_1' \in Sub_{\Sigma}(X, V, V_1)$, and $\rho \in Sub_{\Sigma}(X, V_1, V_2)$ is a typed substitution with

$$\rho \circ \sigma_n' \circ \cdots \circ \sigma_1' \ =_{\mathcal{T}} \ \sigma_n \circ \cdots \circ \sigma_2 \circ \phi \circ \sigma_1' \ =_{\mathcal{T}} \ \sigma_n \circ \cdots \circ \sigma_2 \circ \sigma_1.$$

■

The completeness of $CSU_{\mathcal{T}}$-resolution follows from completeness of $\mathcal{T}$-resolution and CSU-lemma 3.36:

Theorem 3.37 (Completeness of $CSU_{\mathcal{T}}$-resolution) *Let (Σ, C) be a typed logic program, V be a finite set of typed variables and G be a (Σ, X, V)-goal. If $\sigma \in Sub_\Sigma(X, V, V')$ is a typed substitution with $(\Sigma, C, V') \models \sigma(G)$, then there exist a set of typed variables V_0 and a typed substitution $\sigma_0 \in Sub_\Sigma(X, V_0, V_1)$ with $(\Sigma, C, V_0) \models_{\mathrm{RC}} \sigma_0\ G$, and there is a typed substitution $\phi \in Sub_\Sigma(X, V_1, V')$ with $\phi(\sigma_0(G)) =_{\mathcal{T}} \sigma(G)$.*

Proof: By completeness theorem 3.35, there exist a set of typed variables V_0 and a $\mathcal{T}$-resolution of the form

$$(\Sigma, C, V_0) \quad G \quad \models_{\mathrm{R}} \sigma_1 \quad G_1 \quad \models_{\mathrm{R}} \sigma_2 \quad G_2 \quad \models_{\mathrm{R}} \quad \cdots \quad \models_{\mathrm{R}} \sigma_n \quad \emptyset$$

with $\sigma_n \circ \cdots \circ \sigma_1 \in Sub_\Sigma(X, V_0, V')$ and $\sigma_n \circ \cdots \circ \sigma_1(G) = \sigma(G)$. CSU-lemma 3.36 yields a $CSU_{\mathcal{T}}$-resolution

$$(\Sigma, C, V_0) \quad G \quad \models_{\mathrm{RC}} \sigma_1' \quad G_1' \quad \models_{\mathrm{RC}} \sigma_2' \quad G_2' \quad \cdots \quad \models_{\mathrm{RC}} \sigma_n' \quad \emptyset$$

and a typed substitution $\phi \in Sub_\Sigma(X, V_1, V')$ (where $\sigma_0 := \sigma_n' \circ \cdots \circ \sigma_1' \in Sub_\Sigma(X, V_0, V_1)$) with $\phi \circ \sigma_n' \circ \cdots \circ \sigma_1'(G) =_{\mathcal{T}} \sigma_n \circ \cdots \circ \sigma_1(G) = \sigma(G)$. ■

Soundness theorem 3.30 and completeness theorem 3.37 justify the implementation of $CSU_{\mathcal{T}}$-resolution as a proof method for typed logic programs. A complete resolution method must enumerate all possible derivations. If we use a backtracking method like Prolog, the resolution method becomes incomplete because of infinite derivations (in our typed framework the search tree may have an infinite depth as well as an infinite breadth because $CSU_{\mathcal{T}}(L, L')$ may be an infinite set). If we accept this drawback, we can implement the resolution like Prolog with the difference that the unification is extended to typed terms. In Section 3.5 we have shown that the classical unification algorithm can be used if the types of the terms are unified before unifying the terms. For the unification of type expressions w.r.t. the type specification a unification procedure for equational theories is needed. It is known that the *narrowing procedure* [Sla74] [Fay79] [Hul80] (a combination of unification and term rewriting) can be used for this purpose. Narrowing an expression is applying to it the most general substitution such that the expression is reducible and then reduce it. But the narrowing procedure computes a complete set of unifiers w.r.t. an equational theory only if the set of equations is a canonical (i.e., confluent and terminating) term rewriting system. A set of equations can be transformed into a canonical term rewriting system by the Knuth-Bendix procedure [KB70] which is successful for our applications. For instance, let $\mathcal{T}$ be a type structure for integer numbers with appropriate subtype relationships, i.e., *zero* and *posint* are subtypes of the natural numbers, and the negative integers and the natural numbers are subtypes of the integer numbers. Therefore $\mathcal{T}$ is the following equational specification:

$$\begin{array}{lllll}
\texttt{TYPEOPS} & zero: & & \rightarrow & type \\
 & posint: & & \rightarrow & type \\
 & nat: & type & \rightarrow & type \\
 & negint: & & \rightarrow & type \\
 & int: & type & \rightarrow & type
\end{array}$$

$$\begin{array}{llll}
\texttt{TYPEAXIOMS} & nat(zero) & = & zero \\
 & nat(posint) & = & posint \\
 & int(negint) & = & negint \\
 & int(nat(\alpha)) & = & nat(\alpha)
\end{array}$$

The Knuth-Bendix procedure transforms this specification into the following set of rewrite rules:

$$\begin{array}{lll}
nat(zero) & \Rightarrow & zero \\
nat(posint) & \Rightarrow & posint \\
int(negint) & \Rightarrow & negint \\
int(nat(\alpha)) & \Rightarrow & nat(\alpha) \\
int(zero) & \Rightarrow & zero \\
int(posint) & \Rightarrow & posint
\end{array}$$

All equations are oriented from left to right and two additional rewrite rules are generated ("*zero* and *posint* are subtypes of the integer numbers") which corresponds to the computation of the transitive closure of the subtype relation specified in $\mathcal{T}$. This set of rewrite rules is a canoncial term rewriting system and therefore the narrowing procedure w.r.t. these rules can be used to compute $\mathcal{T}$-unifiers for two type expressions. Thus the resolution procedure can be implemented by the following two steps:

1. Transform the given type specification into a canonical term rewriting system. For this purpose the Knuth-Bendix completion procedure can be applied. It computes the transitive closure of the subtype relation.

2. The $\mathcal{T}$-unification procedure for typed terms can be implemented like the classical unification procedure with the difference that types are $\mathcal{T}$-unified by the narrowing procedure w.r.t. the rewrite rules computed in step 1 before unifiying corresponding terms.

Note that the $\mathcal{T}$-unification procedure can be simplified if the type specification does not contain subtype relations (see remarks at the end of Section 3.5). If the type specification contains subtype relations, then these subtype relations have influence on the success or failure of unification. Therefore type information at run time is not superfluous in the context of logic programming but may avoid unnecessary computations since variables can be constraint to values *and* to types by the $\mathcal{T}$-unification prodedure. Therefore typed logic programs can be executed more efficiently than their untyped equivalents [SS85] [HV87]. One reason for this efficiency is the existence of a procedure which decides whether a system of type constraints has a solution. As shown above, we solve type constraints by a narrowing procedure which is based on the type equations. This is sufficient to solve type constraints in order-sorted type structures, but in a more general setting narrowing cannot decide the solvability of constraints but enumerates only a complete set of solutions. Narrowing can only be used as a decision procedure if each narrowing derivation is finite. Hullot [Hul80] has shown that a terminating $\mathcal{T}$-unification algorithm can be constructed by narrowing if any basic narrowing derivation for the right-hand sides of the rules is finite. This is the case in our simple examples and therefore narrowing on type expressions yields a decidability unification procedure for our examples. For another polymorphically order-sorted typed framework, Smolka [Smo89] has shown that type constraints can be efficiently solved. Therefore the development of efficient type constraint solvers for (restricted classes of) our framework is a topic for further research.

3.7 Applications

We have mentioned in the introduction that a new application of our proposed framework for typed logic programming is the possibility of higher-order logic programming with polymorphic and order-sorted type structures. It is clear that our framework combines polymorphic and order-sorted type structures (take the union of the type specifications of examples 3.1 and 3.2, or example 3.3). A semantically clean amalgamation of higher-order objects with logic programming needs a higher-order logic. Miller and Nadathur [MN86] have proposed a higher-order logic programming language based on the typed lambda calculus. The operational semantics is based on resolution with a unification procedure for typed lambda expressions which is a complex and semi-decidable problem. Moreover, the proof procedure is only complete for goals which contain no type variables.

Warren [War82] has argued that no extension to Horn clause logic is necessary because the usual higher-order programming techniques can be simulated in first-order Horn clause logic. The general idea is an explicit definition of a predicate `apply` which is used for the application of an (at compile time) unknown predicate to some arguments. It is shown in [Han89b] that Warren's approach is incompatible with polymorphic type systems for logic programming like [MO84] and [Smo89]. Since we have dropped some restrictions of these type systems, we can use Warren's approach to integrate higher-order programming techniques in our framework.

Example 3.38 We give an example for the definition of a predicate `map` which applies a binary predicate to corresponding elements of two lists. To define the type of `map` we must express the type of binary predicates which are arguments to other predicates. Therefore we introduce a type constructor *pred*2 that denotes the type of binary predicates, i.e., the type specification for our example program is:

TYPEOPS	*int*:		$\rightarrow$	*type*
	bool:		$\rightarrow$	*type*
	list:	*type*	$\rightarrow$	*type*
	*pred*2:	*type*, *type*	$\rightarrow$	*type*

For each binary predicate p of type "τ_1, τ_2" we introduce a corresponding constant λp of type "$pred2(\tau_1, \tau_2)$". The relation between each predicate p and the constant λp is defined by clauses for the predicate `apply2`. Hence we get the following example program for the predicate `map` (we omit the definitions of the predicates `inc` and `bool` and the type annotations in program clauses):

```
func []:          → list(α)
func [..|..]: α, list(α), → list(α)
func λnot:   → pred2(bool,bool)
func λinc:   → pred2(int,int)
...
pred not:  bool, bool
pred inc:  int, int
pred map:  pred2(α,β), list(α), list(β)
pred apply2:  pred2(α,β), α, β

vars P:pred2(α,β), E1:α, E2:β, L1:list(α), L2:list(β),
     B1,B2:bool, I1,I2:int

map(P,[],[]) ←
map(P,[E1|L1],[E2|L2]) ← apply2(P,E1,E2), map(P,L1,L2)
apply2(λnot,B1,B2) ← not(B1,B2)
apply2(λinc,I1,I2) ← inc(I1,I2)
...
```

The first two clauses constitute the standard definition of the predicate `map` (cf. [SS86], p. 281), and the clauses for `apply2` relate the predicate names to the corresponding binary predicates. Since the semantics of typed logic programs is based on a typed first-order logic, the predicate symbol `map` is semantically not interpreted as a higher-order predicate. The constants λ`not` and λ`inc` are also interpreted as values and not as relations. But the clauses for `apply2` ensures that in every model of the program the constants λ`not` and λ`inc` are related to the binary predicates `not` and `inc`, respectively.

This example shows the possibility to deal with higher-order objects in our typed framework. Higher-order objects are related to predicates by particular clauses for an `apply` predicate. It is also possible to permit lambda expressions which can be translated into new identifiers and `apply` clauses for these identifiers (see [War82] and [CvER90] for more discussion). The translation was explicitly done in our examples, but this is a simple task and can be automatically done. If the underlying system implements indexing on the clauses, e.g., indexing on the first arguments of predicates (as done in most compilers for Prolog, cf. [War83] [Han88]), then there is no essential loss of efficiency in our translation scheme for higher-order objects in comparison to a specific implementation of higher-order objects [War82].

More details about this method of higher-order logic programming in a polymorphically typed framework can be found in [Han89b].

3.8 Conclusions

We have presented a general framework for typed logic programming. It consists of a specification of a type structure and a set of well-typed Horn clauses together with type declarations for the syntactic objects occurring in the set of Horn clauses. For the definition of the type structure we have used equational specifications. This allows the specification of both polymorphic and order-sorted type structures and has the advantage that there exist well-known unification procedures for a lot of equational theories. We have defined a procedure to enumerate complete sets of unifiers for typed terms with respect to a type specification which is based on a unification procedure for the equational type specification. Furthermore, we have shown that resolution is sound and complete if this unification procedure is used to unify an atom with a clause head. This framework permits polymorphic and order-sorted type structures and the possibility of the application of useful logic programming techniques like lemma generation and higher-order programming.

The presented framework yields a new view on the rôle of types in logic programming. A type specification can be compiled into a suitable unification algorithm which is used in the resolution procedure. Therefore different type structures imply different unification algorithms. A many-sorted type structure does not require any type information at run time, in a polymorphic type structure a most general unifier exists for two unifiable terms and can be computed by Robinson's unification algorithm, and in order-sorted type structures there may exist several unifiers which are not comparable, but a complete set of unifiers can be computed by a procedure which is based on a unification procedure for the type theory.

Further work remains to be done. We have mentioned that the presence of types at run time is not superfluous but may reduce the search space of the resolution method. Nevertheless, there are a lot of cases where type annotations can be omitted at run time and the unification remains to be correct. For polymorphic type structures these cases are analyzed in [Han89a] and [Han89b]. New criteria for omitting type annotations must be developed in our general typed framework. Another important point is the automatic inference of types. For practical applications it is tedious to write typed program clauses since each syntactic element must be given an appropriate type. Therefore it is necessary to deduce the right types for a clause without type annotations by a type inference algorithm. This is a difficult problem in our general framework but their are successful approaches to the type inference problem for restricted classes of type structures. For instance, in the case of polymorphic type structures the type inference algorithm of ML [DM82] can be used to infer the types of the variables in a clause if the types of all functions and predicates are explicitly declared [Han89a]. For a restricted class of polymorphically order-sorted type structures Smolka has found an algorithm which infers the types of variables in most cases [Smo89]. Similar solutions must be developed for particular instances of our approach.

Bibliography

[BC83] M. Bidoit and J. Corbin. A Rehabilitation of Robinson's Unification Algorithm. In *Proc. IFIP '83*, pp. 909–914. North-Holland, 1983.

[BG89] R. Barbuti and R. Giacobazzi. A Bottom-Up Polymorphic Type Inference in Logic Programming. Technical Report 27/89, Dip. di Informatica, Università di Pisa, 1989.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, Vol. 5, pp. 56–68, 1940.

[CM87] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer, third rev. and ext. edition, 1987.

[CvER90] M.H.M. Cheng, M.H. van Emden, and B.E. Richards. On Warren's Method for Functional Programming in Logic. In *Proc. Seventh International Conference on Logic Programming*, pp. 546–560. MIT Press, 1990.

[CW85] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *acm computing surveys*, Vol. 17, No. 4, pp. 471–523, 1985.

[DH88] R. Dietrich and F. Hagl. A polymorphic type system with subtypes for Prolog. In *Proc. ESOP 88, Nancy*, pp. 79–93. Springer LNCS 300, 1988.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proc. 9th Annual Symposium on Principles of Programming Languages*, pp. 207–212, 1982.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1: Equations and Initial Semantics*, volume 6 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

[Fay79] M.J. Fay. First-Order Unification in an Equational Theory. In *Proc. 4th Workshop on Automated Deduction*, pp. 161–167, Austin (Texas), 1979. Academic Press.

[GM84] J.A. Goguen and J. Meseguer. Completeness of Many-Sorted Equational Logic. Report No. CSLI-84-15, Stanford University, 1984.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.G. Wagner. An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types. In R. Yeh, editor, *Current Trends in Programming Methodology*, volume 4, pp. 80–149. Prentice Hall, Englewood Cliffs NJ, 1978.

[GZ86] Y. Gang and X. Zhiliang. An Efficient Type System for Prolog. In *Proc. IFIP '86*, pp. 355–359. North-Holland, 1986.

[Han87] W. Hankley. Feature Analysis of Turbo Prolog. *SIGPLAN Notices*, Vol. 22, No. 3, pp. 111–118, 1987.

[Han88] M. Hanus. Formal Specification of a Prolog Compiler. In *Proc. of the Workshop on Programming Language Implementation and Logic Programming*, pp. 273–282, Orléans, 1988. Springer LNCS 348.

[Han89a] M. Hanus. Horn Clause Programs with Polymorphic Types: Semantics and Resolution. In *Proc. of the TAPSOFT '89*, pp. 225–240. Springer LNCS 352, 1989. Extended version in [Han91].

[Han89b] M. Hanus. Polymorphic Higher-Order Programming in Prolog. In *Proc. Sixth International Conference on Logic Programming (Lisboa)*, pp. 382–397. MIT Press, 1989.

[Han90] M. Hanus. A Functional and Logic Language with Polymorphic Types. In *Proc. Int. Symposium on Design and Implementation of Symbolic Computation Systems*, pp. 215–224. Springer LNCS 429, 1990.

[Han91] M. Hanus. Horn Clause Programs with Polymorphic Types: Semantics and Resolution. *Theoretical Computer Science*, Vol. 89, pp. 63–106, 1991.

[HMM86] R. Harper, D.B. MacQueen, and R. Milner. Standard ML. LFCS Report Series ECS-LFCS-86-2, University of Edinburgh, 1986.

[Hul80] J.-M. Hullot. Canonical Forms and Unification. In *Proc. 5th Conference on Automated Deduction*, pp. 318–334. Springer LNCS 87, 1980.

[HV87] M. Huber and I. Varsek. Extended Prolog with Order-Sorted Resolution. In *Proc. 4th IEEE Internat. Symposium on Logic Programming*, pp. 34–43, San Francisco, 1987.

[KB70] D.E. Knuth and P.B. Bendix. Simple Word Problems in Universal Algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pp. 263–297. Pergamon Press, 1970.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[MH88] J.C. Mitchell and R. Harper. The Essence of ML. In *Proc. of the 15th ACM Symposium on Principles of Programming Languages*, pp. 28–46, San Diego, 1988.

[Mis84] P. Mishra. Towards a theory of types in Prolog. In *Proc. IEEE Internat. Symposium on Logic Programming*, pp. 289–298, Atlantic City, 1984.

[MN86] D.A. Miller and G. Nadathur. Higher-Order Logic Programming. In *Proc. Third International Conference on Logic Programming (London)*, pp. 448–462. Springer LNCS 225, 1986.

[MO84] A. Mycroft and R.A. O'Keefe. A Polymorphic Type System for Prolog. *Artificial Intelligence*, Vol. 23, pp. 295–307, 1984.

[Nai87] L. Naish. Specification = Program + Types. In *Proc. Foundations of Software Technology and Theoretical Computer Science*, pp. 326–339. Springer LNCS 287, 1987.

[Pad88] P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1988.

[Poi86] A. Poigné. On Specifications, Theories, and Models with Higher Types. *Information and Control*, Vol. 68, No. 1-3, 1986.

[Rob65] J.A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, Vol. 12, No. 1, pp. 23–41, 1965.

[Sla74] J.R. Slagle. Automated Theorem-Proving for Theories with Simplifiers, Commutativity, and Associativity. *Journal of the ACM*, Vol. 21, No. 4, pp. 622–642, 1974.

[SLC88] Ph. Schnoebelen, D. Lugiez, and H. Comon. A Semantics for Polymorphic Subtypes in Computer Algebra. Technical Report RR 711, Laboratoire d'Informatique Fondamentale et d'Intelligence Artificelle, Grenoble, France, 1988.

[Smo89] G. Smolka. *Logic Programming over Polymorphically Order-Sorted Types.* Dissertation, FB Informatik, Univ. Kaiserslautern, 1989.

[SNGM89] G. Smolka, W. Nutt, J.A. Goguen, and J. Meseguer. Order-Sorted Equational Computation. In Hassan Aït-Kaci and Maurice Nivat, editors, *Resolution of Equations in Algebraic Structures, Volume 2, Rewriting Techniques*, chapter 10, pp. 297–367. Academic Press, New York, 1989.

[SS82] J. Siekmann and P. Szabó. Universal Unification and a Classification of Equational Theories. In *Proc. 6th Conference on Automated Deduction*, pp. 369–389. Springer LNCS 138, 1982.

[SS85] M. Schmidt-Schauss. A Many Sorted Calculus with Polymorphic Functions Based on Resolution and Paramodulation. In *Proc. 9th IJCAI.* W. Kaufmann, 1985.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[vEK76] M.H. van Emden and J.A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, Vol. 23, No. 4, pp. 733–742, 1976.

[Wal89] U. Waldmann. Unification in Order-Sorted Signatures. Technical Report 298, FB Informatik, Univ. Dortmund, 1989.

[War82] D.H.D. Warren. Higher-order extensions to PROLOG: are they needed? In *Machine Intelligence 10*, pp. 441–454, 1982.

[War83] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, Stanford, 1983.

[XW88a] J. Xu and D.S. Warren. A Theory of Types and Type Inference in Logic Programming. Technical Report 88/15, SUNY at Stony Brook, 1988.

[XW88b] J. Xu and D.S. Warren. A Type Inference System For Prolog. In *Proc. 5th Conference on Logic Programming & 5th Symposium on Logic Programming (Seattle)*, pp. 604–619, 1988.

[Zob87] J. Zobel. Derivation of Polymorphic Types for Prolog Programs. In *Proc. Fourth International Conference on Logic Programming (Melbourne)*, pp. 817–838. MIT Press, 1987.