A First-Order Theory of Types and Polymorphism in Logic Programming *

Michael Kifer
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794
kifer@cs.sunysb.edu

James Wu
Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794
wujwo@cs.sunysb.edu

Technical Report #90/23
July 1990
Department of Computer Science
SUNY at Stony Brook

Abstract

We describe a new logic called *typed predicate calculus* ($\mathcal{TPC}$) that gives declarative meaning to logic programs with type declarations and type inference. $\mathcal{TPC}$ supports all popular types of polymorphism, such as parametric, inclusion, and ad hoc polymorphism. The proper interaction between parametric and inclusion varieties of polymorphism is achieved through a new construct, called *type dependency*, which is reminiscent of implication types of [PR89] but yields more natural and succinct specifications. Unlike other proposals where typing has extra-logical status, in $\mathcal{TPC}$ the notion of type-correctness has precise model-theoretic meaning that is independent of any specific type-checking or type-inference procedure. Moreover, many different approaches to typing that were proposed in the past can be studied and compared within the framework of our logic. As an illustration, we apply $\mathcal{TPC}$ to interpret and compare the results reported in [MO84, Smo88, HT90, Mis84, XW88]. Another novel feature of $\mathcal{TPC}$ is its reflexivity with respect to type declarations; namely, in $\mathcal{TPC}$ these declarations can be queried the same way as any other data. This should be contrasted with other proposals where typing is part of meta-specifications about the program, inaccessible from within the program. Type reflexivity is useful for browsing knowledge bases and, potentially, for debugging logic programs.

*Work supported in part by the NSF grant IRI-8903507

1 Introduction

Classical logic programming lacks the notion of typing and thus cannot protect predicates from taking semantically meaningless arguments. As a result, subgoals that are formed in this way—often referred to as *type errors*—are indistinguishable from the regular failing subgoals, which makes logic programs hard to debug. Typing information can also suggest various ways to optimize programs, and this opportunity is missed out in classical logic programming.

Several researchers have addressed this problem by attempting to enrich logic programming with various typing disciplines. These efforts can be classified into *declaration-based* and *inference-based* (also known as reconstruction-based or descriptive) approaches. Bruynooghe [Bru82] proposed a declaration-based type system, that requires the user to augment logic programs with type information, thereby enabling the system to detect type errors at compile time. Mycroft and O'Keefe [MO84] extended [Bru82] to support parametric polymorphism. Recently, Dietrich and Hagl [DH88] and Jacobs [Jac90] further extended [MO84] to handle inclusion polymorphism. Another approach to polymorphic type checking, based on HiLog [CKW89a, CKW89b], is discussed in [Fru89, YFS90].

Other researchers argued that declaration-based typing may severely impact the simplicity and flexibility of logic programming, since it requires every predicate symbol to be annotated with a type. Mishra [Mis84] was the first one to show that the idea of type inference that is widely used in functional languages is applicable to logic programs as well. This work was then followed by [MR85, KH85, Zob87, Azz88, Fru88, PR89], where various inference-based approaches were studied. The idea of type inference is to let the compiler *infer* the types of predicates automatically, with little or no declarations supplied by the user. With type inference, the compiler normally cannot detect type errors, since types are inferred from the syntactic structure of untyped programs. Nevertheless, the types inferred by the system can be presented to the user who—upon inspection—will decide whether the typing is consistent with the user's intentions.

A drawback of many of the aforementioned approaches is that the notion of type errors and type-correctness lacks model-theoretic semantics. This creates a disparity between the semantics of classical (untyped) programs and their typed counterparts. Furthermore, *correctness* of type checking or inference algorithms cannot be independently verified and thus many of these studies bear a significant ad hoc component.

In an attempt to close the semantic gap between programs and types, Xu and Warren [XW88, Xu89, XW90] developed a model theory for type-inference based on the notion of partial interpretations. They convincingly argued that in practice both paradigms, declaration-based and inference-based, are useful. They proposed an algorithm that takes into account user-supplied type declarations and then infers types for the rest of the predicates. Unfortunately, this approach does not work well in the presence of inclusion polymorphism, as discussed in Section 9.3.

In this paper, we propose a framework that unifies many of the above works. Section 9 presents a case-study of several representative approaches [MO84, Smo88, HT90, Mis84, XW88, XW90]. Our results shed new light on these works, and allow to compare them in a single framework.

We start with the classical first-order predicate calculus (abbr., $\mathcal{PC}$) and extend it into a new logic, called *typed predicate calculus* (abbr., $\mathcal{TPC}$), that supports all major types of polymorphism: inclusion, parametric, and ad hoc.

The notoriously hard problem in regard with polymorphism in logic systems is the accommodation of parametric types and type hierarchy. In $\mathcal{TPC}$, this is achieved, in part, through the novel notion of *type dependency*. Polymorphic types are treated as first-class citizens and are represented as ordinary

terms.[1] The relationship among types and their instances is defined by the user, via logic formulae that correspond to type declarations of [MO84, YS87, XW88, Jac90], but are more general.

It should be noted that, despite its name, $\mathcal{TPC}$ is *not* a typed language. Instead, the type of a predicate is declared via logical formulas, called signatures, and types are enforced by semantic, model-theoretic means (as opposed to the syntactic means, such as many-sorted algebras [Smo88, HT90]). Thus, unlike other approaches, type declarations in $\mathcal{TPC}$ are formulae in the *object language* of the logic, rather than syntactic expressions in a *meta-language*. This bridges the gap between the calculi of types and data and enables the user to *query* type declarations in quite sophisticated ways (see (2) of Section 5.3), which is useful for browsing the structure of knowledge bases and, potentially, also for debugging logic programs.

In $\mathcal{TPC}$, type-correctness and type errors have precise *model-theoretic* meaning that is independent of any specific type checking algorithm. Likewise, the semantics of inferred types is independent of the inference algorithm. Therefore, various such algorithms can be formally studied and compared in our framework. Last but not least, as in [Xu89, XW90], type inference under partial type declarations has a natural semantics in $\mathcal{TPC}$.

2 Typed Predicate Calculus

In this section, we introduce our new logic, Typed Predicate Calculus, that extends classical first-order predicate calculus both syntactically and semantically.

2.1 Syntax

The alphabet of a language $\mathcal{L}$ of $\mathcal{TPC}$ consists of (1) a set $\mathcal{F}$ of *function* symbols, (2) a set $\mathcal{P}$ of *predicate* symbols, and (3) a set $\mathcal{V}$ of *variables*, as in classical $\mathcal{PC}$. Each function and predicate symbol has an *arity*—a nonnegative integer denoting the number of arguments that the symbol can take. A 0-ary function symbol is called a *constant*.

As in classical $\mathcal{PC}$, a *term* is a constant, a variable, or a statement of the form $f(t_1, ..., t_n)$, where f is a function symbol of arity $n > 0$ and the t_i's are terms. In $\mathcal{TPC}$, *ground* (i.e., variable-free) terms denote individual objects (as in $\mathcal{PC}$) as well as *classes* (or types) of objects.[2]

So as not to overwhelm the reader with too many new concepts, we first describe a 1-sorted version of our logic, which does not distinguish between individuals and classes. In particular, we will use just one sort of variables that range over the names of types as well as over the names of individuals populating these types. Section 7 presents a simple 2-sorted extension of $\mathcal{TPC}$ that allows us to distinguish between types and individuals and yet preserve the reflexivity of $\mathcal{TPC}$, i.e., the ability to manipulate types the same way as data. In our formalism, typing is defined via the notion of a class of entities (or objects), and to foster this intuition we will be using the terms *class* and *type* interchangeably.

Atomic formulae of classical predicate calculus are called *data atoms* in $\mathcal{TPC}$, in order to distinguish them from is-a and signature atoms introduced below.

[1] Jacobs [Jac90] uses a similar (but less general) term-based representation of types. His overall approach is not semantic, though, and the notion of type-correctness is algorithm-dependent. In a different context, term-based representation of types also appears in [KL89].

[2] The view of types as classes comes from object-oriented logics [KL89, KLW90].

To enable modeling the inclusion polymorphism, $\mathcal{TPC}$ lets one organize terms in an is-a hierarchy via a new kind of atomic formulae, called is-a atoms. An *is-a atom* $S : T$, where S and T are terms, states that S is of the type T or, more precisely, S belongs to the class of things denoted by the term T. The notion of is-a atoms is the primary tool for representing polymorphic types. As will be seen later, the class denoted by T includes T itself (as a "typical" element) and thus is-a atoms denote non-strict subclass relationship. In particular, if $S = T$ holds in all semantic structures then so do $S : T$ and $T : S$.

Type constraints are imposed on predicates by means of another kind of atomic formulae, called signature atoms.[3] A *signature* atom (abbr., *signature*), $p\langle T_1, ..., T_n / \alpha_1 \rightarrow \beta_1; ...; \alpha_k \rightarrow \beta_k \rangle$, for an n-ary predicate symbol p consists of a tuple of types represented as terms $T_1, ..., T_n$ and a (possibly empty) set $\{ \alpha_j \rightarrow \beta_j \}_{j=1}^{k}$ of type-dependencies for p. A *type-dependency* $\alpha \rightarrow \beta$ (abbr., TD) for a predicate symbol p is a statement of the form $\alpha \rightarrow \beta$, where α and β are sets of argument positions of p.

We shall see shortly that TDs behave similarly to functional dependencies of the database theory (although, by themselves, they do not enforce any functional dependency on predicates). Type dependencies generalize *modes* in Prolog literature [War77, DW86, DH88]. Intuitively, TDs represent the flow of control between predicate arguments. Section 5.2 (see after Example 5.5) shows that, in the presence of inclusion polymorphism, mode declarations alone are incapable of capturing certain subtleties of interaction between the types assigned to predicate arguments. TDs are also related to *implicational types* of [Red88], which serve similar purpose.

Informally, a signature such as $p\langle u, v, w / \{1,2\} \rightarrow \{3\}; \{3\} \rightarrow \{1\} \rangle$ states that for every data atom $p(a, b, c)$, if a and b are of the types u and v, respectively, then c must be of the type w. Likewise, if c is of the type w then a is of the type u. For notational simplicity, we will often omit braces in signature atoms; e.g., we will write $p\langle u, v, w / 1, 2 \rightarrow 3;\ 3 \rightarrow 1 \rangle$ instead of the above.

Signature atoms allow us to express parametric polymorphic type constraints. For example, the signature $same\_type\langle T, T / 1 \rightarrow 2; 2 \rightarrow 1 \rangle$ means that the second argument of *same_type* must conform to the type of the first and, at the same time, the first argument must conform to the type of the second one. Similarly, a type constraint for *append* can be defined by a signature $append\langle list(T), list(T), list(T) / 1, 2 \rightarrow 3; 3 \rightarrow 1, 2 \rangle$, which requires that for all t, if the first two arguments of *append* have the type $list(t)$ then so does the third argument, and vice-versa.

Given a TD $\alpha \rightarrow \beta$, the argument positions in the sets α and $(\beta - \alpha)$ are called *domain* and *range* positions, respectively ($\beta - \alpha$ denotes the ordinary set-difference). We note that either α, or β, or both may be empty, e.g., $husband\langle man, woman / 1, 2 \rightarrow \emptyset \rangle$ or $parent\langle person, person / \emptyset \rightarrow 1, 2 \rangle$. Likewise, $\alpha \cap \beta$ may be non-empty. The meaning of such TDs will become apparent shortly. To shorten the notation, the set that contains all argument positions of a predicate is denoted by "ω" (the predicate in question will be always clear from the context). For example, we can write $husband\langle man, woman / \emptyset \rightarrow \omega \rangle$ instead of $husband\langle man, woman / \emptyset \rightarrow 1, 2 \rangle$.

In summary, the language of $\mathcal{TPC}$ consists of the following three kinds of atomic formulae:

- *data* atoms $p(T_1, ..., T_n)$, where p is an n-ary predicate symbol and $T_1, ..., T_n$ are terms;
- *is-a* atoms $S : T$, where S and T are terms;
- *signature* atoms $p\langle T_1, ..., T_n / \alpha_1 \rightarrow \beta_1; ...; \alpha_k \rightarrow \beta_k \rangle$, where p is an n-ary predicate symbol, $T_1, ..., T_n$ are terms, and $\alpha_j \rightarrow \beta_j,\ 1 \leq j \leq k,$ are TDs of p.

[3] By analogy with functional programming languages, these can be viewed as signatures of the respective predicates.

The *well-formed formulae* of $\mathcal{TPC}$ are constructed from the atomic formulae by means of the connectives $\neg$, $\wedge$, $\vee$ and the quantifiers $\exists$ and $\forall$, in the usual way.

Let α be a sequence of argument positions of an n-ary predicate. Given an n-tuple $\vec{t}$, we will write $\vec{t}[\alpha]$ to denote the projection of that tuple on the positions of α (e.g., if $\vec{t} = \langle a, b, c\rangle$ then $\vec{t}[2,1,1] = \langle b, a, a\rangle$). In particular, if α is the empty set then $\vec{t}[\alpha]$ is the empty tuple. Likewise, if R is a relation of an appropriate arity, then $R[\alpha]$ denotes the relation obtained by projecting every tuple of R on α.

2.2 Semantics

Given a language $\mathcal{L}$ of $\mathcal{TPC}$, a semantic structure I is a tuple $\langle U, \prec_U, I_\mathcal{F}, I_\mathcal{P}, I_\mathcal{T}\rangle$. Here U is the domain of I, partially ordered by $\prec_U$. As usual, we write $a \preceq_U b$ whenever $a \prec_U b$ or $a = b$. The ordering $\prec_U$ defines the subclass relationship among the elements of U. Intuitively, $a \prec_U b$ says that a is a *subclass* (or a subtype) of b. We extend $\preceq_U$ to tuples over U in the usual way: for $\vec{u}, \vec{v} \in U^n$, $\vec{u} \preceq_U \vec{v}$ if this relation holds component-wise.

Function symbols are interpreted using the mapping $I_\mathcal{F}$ that assigns to each constant an element in U and to each n-ary ($n > 0$) function symbol $f \in \mathcal{F}$ a function $I_\mathcal{F}(f) : U^n \rightarrow U$.

Predicates too are interpreted exactly as in classical $\mathcal{PC}$. For each n-ary predicate symbol p, $I_\mathcal{P}(p)$ is a subset of U^n, i.e., each n-ary predicate is interpreted via an n-ary relation. The equality predicate is interpreted in the standard way: $I_\mathcal{P}(=) \stackrel{\text{def}}{=} \{\langle u, u\rangle | u \in U\}$. Occasionally, we will refer to the tuples of $I_\mathcal{P}(p)$ as *data-tuples* and to $I_\mathcal{P}(p)$ itself as the *data-relation* assigned to p by I, to distinguish them from tuples and relations of the typing constraints associated with p (see next).

The typing constraints on predicates are defined via the mapping $I_\mathcal{T}$ that assigns to each n-ary predicate symbol p a relation $I_\mathcal{T}(p) \subseteq U^n \times \mathcal{TD}^{(n)}$, where $\mathcal{TD}^{(n)}$ denotes the set of all type-dependencies on n-ary predicates. Each tuple $\langle \vec{t}, \alpha \rightarrow \beta\rangle$ in $I_\mathcal{T}(p)$ represents a *typing constraint*; such a constraint is *satisfied* by a data-tuple $\vec{u}$ in $I_\mathcal{P}(p)$, if whenever $\vec{u}$ conforms to the type $\vec{t}$ over the positions α, then $\vec{u}$ also conforms to the type $\vec{t}$ over the positions β. Formally, $\vec{u}$ *satisfies* $\Gamma = \langle \vec{t}, \alpha \rightarrow \beta\rangle$ if and only if

- $\vec{u}\,[\alpha] \preceq_U \vec{t}\,[\alpha]$ implies $\vec{u}\,[\beta] \preceq_U \vec{t}\,[\beta]$.

Observe that $\vec{u}$ satisfies $\langle \vec{t}, \alpha \rightarrow \beta\rangle$ if and only if it satisfies $\langle \vec{t}, \alpha \rightarrow (\beta - \alpha)\rangle$, i.e., repeating some or all of the argument-positions of α in β does not change the meaning of the TD $\alpha \rightarrow \beta$. Also, if $\alpha = \emptyset$ then $\vec{u}\,[\beta] \preceq_U \vec{t}\,[\beta]$ holds unconditionally and if $\beta \subseteq \alpha$ (in particular, if $\beta = \emptyset$) then $\langle \vec{t}, \alpha \rightarrow \beta\rangle$ constrains nothing.

We require each $I_\mathcal{T}(p)$ to be closed with respect to the *implication* of TDs as well as the *supertyping* relationship:

TD Implication:

> Let $\Gamma, \Gamma_1, ..., \Gamma_k$ be type dependencies in $\mathcal{TD}^{(n)}$. We say that $\{\Gamma_1, ..., \Gamma_k\}$ *logically implies* Γ, denoted $\{\Gamma_1, ..., \Gamma_k\} \models_{\mathcal{TD}} \Gamma$, if
>
> - for all domains $\langle V, \prec_V\rangle$ and for all tuples $\vec{u}, \vec{t} \in V^n$, if $\vec{u}$ satisfies all of $\langle \vec{t}, \Gamma_1\rangle$,..., $\langle \vec{t}, \Gamma_k\rangle$ then it also satisfies $\langle \vec{t}, \Gamma\rangle$.
>
> (E.g., $1, 2 \rightarrow 3$ and $3 \rightarrow 4, 5$ imply $1, 2 \rightarrow 4, 5$; inference rules for TDs appear in Section 4.)

Supertyping:

Let $\vec{t}$ and $\vec{s}$ be tuples of the same arity and $\tau_1 = \langle \vec{t}, \alpha \rightarrow \beta \rangle$, $\tau_2 = \langle \vec{s}, \alpha \rightarrow \beta \rangle$ be a pair of typing constraints tagged with the same TD. We say that τ_1 is a *supertyping* of τ_2 (or that τ_1 is a typing constraint *weaker* than τ_2), denoted $\tau_1 \preceq_{\mathcal{T}} \tau_2$, if

1. $\vec{t}[\omega - (\alpha \cup \beta)] = \vec{s}[\omega - (\alpha \cup \beta)]$, where ω denotes the set of all component-places of $\vec{t}$ (and $\vec{s}$);
2. $\vec{t}[\alpha] \preceq_U \vec{s}[\alpha]$; and
3. $\vec{s}[\beta - \alpha] \preceq_U \vec{t}[\beta - \alpha]$.

Condition (2) above requires $\vec{t}$ to be a subtype of $\vec{s}$ in the domain-positions of the TD, while condition (3) says that $\vec{t}$ must be a supertype of $\vec{s}$ in the range-positions. Clearly, a data-tuple that satisfies τ_2 also satisfies its supertyping, τ_1. If we view predicates as set-valued functions from the domain positions to the range positions, these conditions become identical to the usual statements about functional types.

A relation $I_{\mathcal{T}}(p)$ is *closed* with respect to the implication of TDs ($\models_{\mathcal{TD}}$) and the supertyping relationship ($\preceq_{\mathcal{T}}$) if and only if the following two conditions hold:

1. If $\langle \vec{t}, \Gamma_1 \rangle, ..., \langle \vec{t}, \Gamma_k \rangle \in I_{\mathcal{T}}(p)$, and $\{\Gamma_1, ..., \Gamma_k\} \models_{\mathcal{TD}} \Gamma$, then $\langle \vec{t}, \Gamma \rangle \in I_{\mathcal{T}}(p)$; and
2. If $\tau \in I_{\mathcal{T}}(p)$ and $\tau' \preceq_{\mathcal{T}} \tau$ then also $\tau' \in I_{\mathcal{T}}(p)$.

In a semantic structure I, the data-relation associated to a predicate symbol, p, must satisfy the typing constraints associated with p. Formally, this means that the relationship between $I_{\mathcal{P}}$ and $I_{\mathcal{T}}$ is governed by the following rules that glue together the semantics of predicates with that of their signatures.

The Well-Typing Conditions: For every predicate p in $\mathcal{P}$, and every data-tuple $\vec{s} \in I_{\mathcal{P}}(p)$:

1. $\vec{s}$ satisfies every typing constraint in $I_{\mathcal{T}}(p)$;
2. there is a typing constraint $\langle \vec{t}, \alpha \rightarrow \beta \rangle$ in $I_{\mathcal{T}}(p)$, such that $\vec{s} \preceq_U \vec{t}$.

Condition (2) here requires that every data-tuple s must conform to at least one tuple of types, $\vec{t}$, in $I_{\mathcal{T}}(p)$.

A *variable-assignment*, ν, is a mapping from variables, $\mathcal{V}$, to the domain U. It is extended to terms in the usual way: $\nu(f(t_1, ..., t_n)) = I_{\mathcal{F}}(f)(\nu(t_1), ..., \nu(t_n))$. Let I be a semantic structure and ν a variable-assignment. Satisfaction of an atom A under a semantic structure I and a variable-assignment ν, denoted $I \models_\nu A$, is defined as follows:

1. For an is-a atom $S : T$, $I \models_\nu S : T$ if and only if
   - $\nu(S) \preceq_U \nu(T)$.
2. For a signature atom $A = p\langle T_1, ..., T_n / \alpha_1 \rightarrow \beta_1; ...; \alpha_k \rightarrow \beta_k \rangle$, $I \models_\nu A$ if and only if
   - When $k \geq 1$: $\langle \nu(T_1), ..., \nu(T_n), \alpha_i \rightarrow \beta_i \rangle \in I_{\mathcal{T}}(p)$, for $i = 1, ..., k$;
   - When $k = 0$: $\langle \nu(T_1), ..., \nu(T_n), \emptyset \rightarrow \emptyset \rangle \in I_{\mathcal{T}}(p)$.
3. For a data atom $A = p(T_1, ..., T_n)$, $I \models_\nu A$ if and only if
   - $\langle \nu(T_1), ..., \nu(T_n) \rangle \in I_{\mathcal{P}}(p)$.

The meaning of the formulae $\phi \vee \psi$, $\phi \wedge \psi$, and $\neg\phi$ is defined in the standard way: $I \models_\nu \phi \vee \psi$ (or $I \models_\nu \phi \wedge \psi$, or $I \models_\nu \neg\phi$) if and only if $I \models_\nu \phi$ or $I \models_\nu \psi$ (resp., $I \models_\nu \phi$ and $I \models_\nu \psi$, resp., $I \not\models_\nu \phi$). The meaning of quantifiers is also standard: $I \models_\nu \psi$, where $\psi = (\forall X)\phi$ (or $\psi = (\exists X)\phi$) if for every (resp., some) μ that agrees with ν everywhere, except possibly on X, $I \models_\mu \phi$.

Clearly, for a closed formula, ψ, its meaning is independent of a variable-assignment, and we can thus write $I \models \psi$, when ψ is closed. A semantic structure I is a *model* of ψ if $I \models \psi$. If **S** is a set of formulae and ψ is a formula, we write $\mathbf{S} \models \psi$ (ψ is *logically implied or entailed* by **S**) if and only if ψ is true in every model of **S**.

2.3 Untyped Semantic Structures

In order to distinguish the well-typed $\mathcal{TPC}$ programs from the ill-typed ones, we introduce the notion of *untyped* semantic structures and models.

Untyped semantic structures for a $\mathcal{TPC}$ language $\mathcal{L}$ are defined similarly to ordinary $\mathcal{TPC}$ semantic structures, except that the former are not required to satisfy the well-typing conditions of the previous subsection. From now on, the semantic structures introduced in the previous subsection will be called *typed* semantic structures.

The notion of formula satisfaction by untyped structures goes unchanged. We use the same notation $I \models_\nu \psi$ ($I \models \psi$, for closed formulae) to denote the fact that I satisfies ψ. An untyped semantic structure I is an *untyped model* of a closed formula ψ if $I \models \psi$.

It is easy to see that every typed semantic structure (or model) is also an untyped structure (resp., model) and that any untyped model that is also a typed semantic structure is a typed model.

Although the notion of untyped logical entailment between formulae is immediate, we are not going to use it in the future. Therefore, if **S** is a set of formulae then $\mathbf{S} \models \psi$ will always stand for the typed entailment defined in Section 2.2.

In the sequel, unqualified references to semantic structures (or models) will refer to the typed versions of these concepts. However, we shall often explicitly add the qualifier "typed," for the contrast. On the other hand, when untyped semantic structures are meant, the qualifier "untyped" will always be present.

3 Skolem and Herbrand's Theorems

Since quantification in $\mathcal{PC}$ and $\mathcal{TPC}$ is similar, Skolemization is similar too. For instance, a Skolem normal form for $\forall X \exists Y(p\langle X, Y\rangle \vee q(Y))$ would be $\forall X(p\langle X, f(X)\rangle \vee q(f(X)))$, where f is a new function symbol.

Theorem 3.1 (cf. **Skolem Theorem**) *Let ϕ be a formula (that may contain data, is-a, and signature atoms) and ϕ' be its Skolemization. Then ϕ is unsatisfiable if and only if so is ϕ'.*

Given a language $\mathcal{L}$, its Herbrand universe, denoted $U(\mathcal{L})$, is the set of all *ground* (i.e., variable-free) terms in the language. The Herbrand base of $\mathcal{L}$, denoted $H(\mathcal{L})$, is the set of all ground atomic formulae in the language, including is-a, signature, and data atoms. A *Herbrand interpretation*, H, is a semantic structure with the domain $U(\mathcal{L})$ and the standard interpretation I_F^H of function symbols: if f is a k-ary function symbol then $I_F^H(f)(t_1, ..., t_k) = f(t_1, ..., t_k)$. Equivalently, Herbrand interpretations can be defined as subsets of $H(\mathcal{L})$ that are *closed* under the logical implication "$\models$". The closure

requirement is convenient here since ground atoms may imply other atoms in a non-trivial way, e.g. $\{p\langle a,b,c/1 \rightarrow 2\rangle, p\langle a,b,c/1 \rightarrow 3\rangle\} \models p\langle a,b,c/1 \rightarrow 2,3\rangle$. This is similar to predicate calculus with equality, where sets of ground atoms may imply other atoms.

Proposition 3.1 *Let* **S** *be a set of clauses. Then* **S** *is unsatisfiable if and only if* **S** *has no Herbrand model.*

Proof: The proof is similar to that in $\mathcal{PC}$. □

Proposition 3.2 *Let $\mathcal{L}$ be an arbitrary language of $\mathcal{TPC}$. There is a corresponding language $\mathcal{L}'$ of $\mathcal{PC}$ and a pair of transformations*

Φ : *$\mathcal{TPC}$-formulae of $\mathcal{L}$* $\longmapsto$ *$\mathcal{PC}$-formulae of $\mathcal{L}'$*
Ψ : *$\mathcal{TPC}$-semantic structures for $\mathcal{L}$* $\longmapsto$ *$\mathcal{PC}$-semantic structures for $\mathcal{L}'$*

such that for every $\mathcal{TPC}$-formula ϕ in $\mathcal{L}$ and a $\mathcal{TPC}$-semantic structure M, $M \models_{\mathcal{TPC}} \phi$ if and only if $\Psi(M) \models_{\mathcal{PC}} \Phi(\phi)$ ($\models_{\mathcal{TPC}}$ and $\models_{\mathcal{PC}}$ denote logical implication in $\mathcal{TPC}$ and $\mathcal{PC}$, respectively).

Proof: In constructing these mappings, one needs to introduce the "signature-version" for each predicate p (i.e., a new predicate name p^{sig}) and then use formulae of predicate calculus to encode the unique features of $\mathcal{TPC}$-semantic structures, including the is-a relationship, the well-typing conditions, the closure requirements of TDs, and the supertyping relationship. □

Theorem 3.2 (cf. **Herbrand's Theorem**) *A set of clauses* **S** *is unsatisfiable if and only if so is some finite subset of ground instances of clauses in* **S**.

Proof: The result follows from Proposition 3.2 and Herbrand's theorem for $\mathcal{PC}$. □

As in $\mathcal{PC}$, Herbrand's theorem for $\mathcal{TPC}$ is in the basis of the completeness proof for the resolution-based proof procedure that is presented in the next section.

4 Proof Theory

The existence of *some* proof theory for $\mathcal{TPC}$ is not at all surprising in view of Proposition 3.2. What is remarkable though is the existence of a complete proof theory that combines ordinary resolution with type-inference rules commonly found in the works on typing of functional programs.

In view of Skolem theorem, we can restrict our attention to formulas in the *clausal* form, i.e., universally quantified disjunctions of literals, where *literal* is either an atomic formula or negation of an atomic formula.

A *substitution*, σ, is a mapping {Variables of $\mathcal{L}$} $\longrightarrow$ {Terms of $\mathcal{L}$} that is an identity almost everywhere but on a finite number of variables. The notion of unifiers in $\mathcal{TPC}$ is the same as in $\mathcal{PC}$: Given a set L_1, ..., L_n of literals of the same kind (is-a, data, or signature), a substitution σ is a *unifier* if $\sigma(L_1) = ... = \sigma(L_n)$. A unifier σ is *most general* if for every other unifier, γ, there is a substitution θ such that $\gamma = \theta \circ \sigma$.

Consider a set $\mathbf{S} = \{p\langle \vec{T}/\Gamma_1\rangle, ..., p\langle \vec{T}/\Gamma_k\rangle\}$ of signature-atoms that differ only in the TD-part. It is easy to check that **S** logically entails every atom $p\langle \vec{T}/\Gamma\rangle$ such that each member of Γ is $\models_{\mathcal{TD}}$-entailed by the union of $\Gamma_1, ..., \Gamma_k$. For instance, since $\{1 \rightarrow 2,3\} \models_{\mathcal{TD}} 1 \rightarrow 2$ and $\{1 \rightarrow 2,3\} \models_{\mathcal{TD}} 1 \rightarrow 3$, we

have $\{p\langle a, b, c/1 \rightarrow 2, 3\rangle\} \models p\langle a, b, c/1 \rightarrow 2; 1 \rightarrow 3\rangle$. To account for the derivations of this kind, we need to characterize the logical implication among sets of TDs. The following inference rules are sound and complete for TDs. Incidentally, these rules coincide with the well-known Armstrong's axioms for functional dependencies [Ull88]. Let Γ be a set of TDs for n-ary predicates, and let $\omega_n = \{1, ..., n\}$.

Reflexivity : If $\beta \subseteq \alpha \subseteq \omega_n$, then $\Gamma \vdash_{\mathcal{TD}} \alpha \rightarrow \beta$.

Augmentation : If $\Gamma \vdash_{\mathcal{TD}} \alpha \rightarrow \beta$ then $\Gamma \vdash_{\mathcal{TD}} (\alpha \cup \delta) \rightarrow (\beta \cup \delta)$, for some $\delta \subseteq \omega_n$.

Transitivity : If $\Gamma \vdash_{\mathcal{TD}} \alpha \rightarrow \beta$ and $\Gamma \vdash_{\mathcal{TD}} \beta \rightarrow \gamma$ then $\Gamma \vdash_{\mathcal{TD}} \alpha \rightarrow \gamma$.

A *derivation* of a TD γ from Γ is a sequence γ_1, ..., γ_m such that $\gamma_m = \gamma$ and each γ_i is either in Γ or is derived from γ_1, ..., γ_{i-1} by one of the above rules. We write $\Gamma \vdash_{\mathcal{TD}} \Gamma'$, where Γ' is a set of TDs, if every member of Γ' can be derived from Γ using a derivation sequence from Γ. The question of whether or not $\Gamma \vdash_{\mathcal{TD}} \gamma$, can be decided in linear time in the sizes of Γ and γ [BB79].

Proposition 4.1 *The inference rules for TDs are sound and complete, that is,*

$$\Gamma \models_{\mathcal{TD}} \gamma \quad \textit{if and only if} \quad \Gamma \vdash_{\mathcal{TD}} \gamma.$$

Proof: Similar to the corresponding proof for functional dependencies [Ull88]. □

It follows from the above rules for TDs that $\emptyset \rightarrow \emptyset$, $\omega \rightarrow \emptyset$ and, in fact, any TD $\alpha \rightarrow \beta$, where $\beta \subseteq \alpha$, is a tautology. Thus, for instance, $p\langle \vec{t}/\omega \rightarrow \emptyset\rangle$ is logically equivalent to $p\langle \vec{t}/\emptyset \rightarrow \emptyset\ \rangle$ and to $p\langle \vec{t}/\ \rangle$. Tautological TDs will be called *trivial* in the sequel.

Figure 1 lists the inference rules for $\mathcal{TPC}$. In this figure, if $\vec{S}$ and $\vec{T}$ are tuples of terms of the same arity n then $mgu(\vec{T}, \vec{S})$ denotes the most general *simultaneous* unifier of $\vec{T}[i]$ and $\vec{S}[i]$ $(i = 1, ..., n)$. In the paramodulation rule, $L[T]$ denotes a literal that contains T as a term; $L[S']$ denotes the same literal with one occurrence of T replaced by S'. A *derivation* of a clause C from a set of clauses P is a sequence C_0, ..., C_n, where $C_n = C$ and each C_i is either in P or is derived from C_0, ..., C_{i-1} using one of the derivation rules of Figure 1. Existence of such a derivation, is denoted by $P \vdash C$. A *refutation* of P is a derivation of an empty clause from P.

Theorem 4.1 (Soundness) *If $P \vdash C$ then $P \models C$.*

Theorem 4.2 (Completeness) *If a set of clauses P is unsatisfiable then there is a refutation of P.*

5 Well-Typed Logic Programs

$\mathcal{TPC}$ is much too general than what we actually might need in order to write programs. Apart from the usual restrictions on the form of rules in logic programs, we will dissect $\mathcal{TPC}$ programs into the following three disjoint components:

1. *class-hierarchy* (or is-a hierarchy) declaration;
2. *type* declaration; and
3. *data* definition.

Core Inference Rules

1. Resolution: $\neg A \vee C_1,\ A' \vee C_2,\ \theta = mgu(A, A') \ \vdash\ \theta(C_1 \vee C_2)$
2. Factorization: $L \vee L' \vee C,\ \theta = mgu(L, L') \ \vdash\ \theta(L \vee C)$
3. Paramodulation: $L[T] \vee C_1,\ T' = S' \vee C_2,\ \theta = mgu(T, T') \ \vdash\ \theta(L[S'] \vee C_1 \vee C_2)$

Inference Rules for ISA

4. ISA-reflexivity: $\vdash\ T : T$
5. ISA-transitivity: $S : R \vee C_1,\ R' : T \vee C_2,\ \theta = mgu(R, R') \ \vdash\ \theta(S : T \vee C_1 \vee C_2)$
6. ISA-acyclicity: $S : T \vee C_1,\ T' : S' \vee C_2,\ \theta = mgu(\langle S, T\rangle, \langle S', T'\rangle) \ \vdash\ \theta(S = T \vee C_1 \vee C_2)$

Inference Rules for Typing

7. TD-derivation: $p\langle \vec{T}/\Gamma\rangle \vee C_1,\ p\langle \vec{T}'/\Gamma'\rangle \vee C_2,\ \Gamma \cup \Gamma' \vdash_{\mathcal{TD}} \Gamma'', \theta = mgu(\vec{T}, \vec{T}')$
$\vdash\ \theta(p\langle \vec{T}/\Gamma''\rangle \vee C_1 \vee C_2)$
8. Supertyping #1: $p\langle T_1, ..., T_n/\Gamma\rangle \vee C_1,\ S : T' \vee C_2,\ \alpha \to \beta \in \Gamma,\ i \in \alpha,$
$\theta = mgu(T_i, T') \ \vdash\ \theta(p\langle T_1, ..., T_{i-1}, S, T_{i+1}, ..., T_n/\alpha \to \beta\rangle \vee C_1 \vee C_2)$
9. Supertyping #2: $p\langle T_1, ..., T_n/\Gamma\rangle \vee C_1,\ T' : S \vee C_2,\ \alpha \to \beta \in \Gamma,\ j \in \beta - \alpha,$
$\theta = mgu(T_j, T') \ \vdash\ \theta(p\langle T_1, ..., T_{j-1}, S, T_{j+1}, ..., T_n/\alpha \to \beta\rangle \vee C_1 \vee C_2)$

Inference Rules for Type Checking

10. Well-typing #1: $p(\vec{S}) \vee C_1,\ p\langle \vec{T}/\Gamma\rangle \vee C_2,\ \ \alpha \to \beta \in \Gamma,\ \ \theta = mgu(\vec{S}\,[\alpha], \vec{T}\,[\alpha]),$
$\vdash\ \theta(\vec{S}\,[j] : \vec{T}\,[j] \vee C_1 \vee C_2),$ for all $j \in \beta - \alpha$
11. Well-typing #2: $p(\vec{S}) \vee C \ \vdash\ p\langle \vec{S}/\emptyset \to \emptyset\,\rangle \vee C$

Figure 1: $\mathcal{TPC}$ Inference Rules

Strictly speaking, this separation is not necessary for our theory to work. However, this seems to be a common practice so far in the works on types in logic programming and we do not intend to break the tradition in this paper. We note that object-oriented logic languages [KLW90] may require additional flexibility of being able to define classes that contain all objects with certain typing or internal state. In such a case, the above three components might not be independent.

The following subsections discuss the aforesaid program components separately. As we shall see, the monotonic semantics of Section 2.2 is not strong enough to capture all aspects of typing. We then define a nonmonotonic semantics of logic programs on top of the logical entailment relation "$\models$" of Section 2.2 and argue that this gives an adequate semantics to the notion of type-correctness.

5.1 Class-Hierarchy Declaration

Class or *is-a hierarchies* are specified via sets of definite Horn clauses that involve is-a and equality atoms only. Such a declaration, ISA, says what is a subclass (or a subtype) of what, i.e., if ISA $\models p : q$, then p is a subclass of q. It also specifies which types can be referred to via different names. For

instance, if ISA $\models$ $(human = person)$ then *human* and *person* denote the same type. In the following, we will use the terms class/subclass and type/subtype interchangeably.

Unfortunately, as given, class declarations are too weak since definite Horn clauses cannot tell what is *not* a subclass of another class. The solution is to use the Closed World Assumption (abbr., CWA)—a standard technique for nonmonotonic reasoning that allows to infer negative information from Horn clause programs; it states that whenever ISA $\not\models p : q$ then one can nonmonotonically derive $\neg p : q$. As is well known, for definite Horn clauses, CWA is tantamount to restricting logical entailment to minimal models only.

The minimality order on untyped semantic structures is defined as follows: Let $\dot{I}$ and $\ddot{I}$ be a pair of semantic structures. Then $\dot{I} \ll^{min} \ddot{I}$ if and only if for every ground atom ϕ, $\dot{I} \models \phi$ implies $\ddot{I} \models \phi$.

The class hierarchy defined by P is the set of all ground is-a atoms satisfied by some minimal model of P. Note that since ISA is a set of definite Horn clauses and is-a atoms do not depend on the rest of P, it does not matter which minimal model is chosen here, even if other components of P are non-Horn.

Example 5.1 Consider the following class-hierarchy declaration:

$man : person$	$john : man$	$mary : woman$
$woman : person$	$bob : man$	$ann : woman$
	$peter : man$	$sue : woman.$

This declaration states not only that *john* is a *man*, but also that *john* is *not* a *woman*. The former is defined explicitly, while the latter is derived nonmonotonically by CWA, since $john : woman$ is not logically entailed by the declaration.

The first pair of clauses above define the subtype relationship among *man*, *woman*, and *person*; subtype relationship is a key ingredient in the notion of *inclusion polymorphism*. □

Example 5.2 The type of all objects, *all*, and the type of natural numbers, *nat*, can be defined thus:

$$X : all \tag{1}$$

$$zero : nat$$
$$succ(X) : nat \quad \longleftarrow \quad X : nat.$$

The first clause says that every thing is of the type *all* (including the constant *all* itself). From now on, the symbol *all* will be reserved to denote the type of all things, whenever it is defined (note: (1) may not be part of ISA and thus *all* may not be defined). The last pair of clauses recursively defines the instances of *nat*: *zero*, $succ(zero)$, $succ(succ(zero))$, and so on. □

In $\mathcal{TPC}$, types are denoted by terms, so they may contain variables. This gives us the flexibility needed to support *parametric polymorphism*.

Example 5.3 The following pair of clauses defines a parametric type $list(T)$:

$$nil : list(T)$$
$$cons(X, Y) : list(T) \quad \longleftarrow \quad X : T \ \& \ Y : list(T).$$

Let *nat* denote the type of natural numbers. Then $list(nat)$ is the type of all lists of natural numbers, containing elements such as nil, $cons(zero, nil)$, $cons(zero, cons(succ(zero), nil))$, and so on. □

Observe that we do not require types to be tuple-distributive in the sense of [Mis84]. However, usually they will come out that way, if the semantic of a specific type calls for this feature (cf. $list(T)$ above). Likewise, in $\mathcal{TPC}$, type constructors do not have to be monotonic. For instance, the definition of $list(T)$ in Example 5.3 does not guarantee that $list(nat) : list(int)$ holds, even if $nat : int$ is known to hold. It is true, however, that if $a : list(nat)$ holds for some term a, then so does $a : list(int)$, as expected. If monotonicity is desired, it can be declared explicitly, e.g.,

$$list(T) : list(S) \quad \longleftarrow \quad T : S.$$

5.2 Type Declaration

A *type declaration* is a set of definite Horn clauses with signature atoms in the head and signature or is-a atoms in the body. The purpose of type declarations is to impose type constraints on the arguments of predicates.

Unfortunately, as in the case of class hierarchies, the monotonic semantics of $\mathcal{TPC}$ is not (and cannot be) strong enough to capture the idea of typing. For instance, $plus\langle int, int, int/1, 2 \rightarrow 3\rangle$ and $plus$("a","b","c") are consistent with each other: an appropriate model could be the one where $\langle$"a", "b", "c", $\{1, 2\} \rightarrow \{3\}\rangle$ belongs to $I_{\mathcal{T}}(plus)$.

Luckily, CWA rescues the situation once again. It is clear that in order to eliminate the "wrong" models where $plus$("a","b","c") is true, we need to ensure that $I_{\mathcal{T}}$ contains no "accidental" tuples, like $\langle$"a", "b", "c"$\rangle$, i.e., tuples that are not logically mandated by the given type declarations. This can be achieved by minimizing semantic structures with respect to signature-atoms that they satisfy, and the minimal model semantics discussed in the previous section can be used once again. Intuitively, we want the typing of a program P to be the set of all ground signature-atoms satisfied by some minimal model of P. Again, the specifics of the minimal model do not matter, even when P is non-Horn, because TYPE and ISA *are* Horn and is-a and signature-atoms do not depend on the rest of P. (As noted earlier, this independence of ISA and TYPE from the rest of the program does not really matter, but it is traditional and simplifies the discussion somewhat.)

Example 5.4 A possible type declaration for *husband*, *parent*, and *play* relations is:

$$husband\langle man, woman/\emptyset \rightarrow \emptyset\rangle \qquad play\langle team, team/1 \rightarrow 2; 2 \rightarrow 1\rangle$$
$$parent\langle person, person/\emptyset \rightarrow \emptyset\rangle \qquad play\langle jr\_team, jr\_team/1 \rightarrow 2; 2 \rightarrow 1\rangle.$$

We assume that $man : person$, $woman : person$, and $jr\_team : team$ hold and these classes are suitably defined. The typing constraint imposed by the first signature requires *husband* to be a relation between instances of the classes *man* and *woman*. The second signature requires *parent* to be a relation between persons. The typing for *play* says that, in general, teams play against teams, but junior teams (jr_team) play against junior teams.

For instance, if $P \stackrel{\text{def}}{=}$ ISA $\cup$ TYPE $\cup$ DATA is a program and ISA $\models giants : team \wedge yankees : team$ then $play(giants, yankees)$ is well-typed. However, if ISA $\models pee\_wee : jr\_team$ but ISA $\not\models giants : jr\_team$ then $play(giants, pee\_wee)$ is ill-typed in the sense that P has no typed minimal model, i.e., none of its untyped minimal models satisfies the well-typing conditions of Section 2.2. □

Each TD represents a particular type-enforcement pattern. One can therefore expect that signatures tagged with different TDs will result in different typing constraints.

Example 5.5 Consider the following pair of type declarations:

T_1 : $p\langle student, student/\emptyset \rightarrow \emptyset\rangle$
$p\langle employee, employee/\emptyset \rightarrow \emptyset\rangle$

T_2 : $p\langle student, student/\emptyset \rightarrow \omega\rangle$
$p\langle employee, employee/\emptyset \rightarrow \omega\rangle$.

Here, T_1 says that every tuple of p must be *either* a student-student pair *or* an employee-employee pair, and nothing else. In contrast, the signatures in T_2 require each tuple in p to be *both* a student-student pair *and* an employee-employee pair. It is instructive to consult the definitions in Section 2.2 to see why this is indeed the case.

Another interesting observation is that T_1 and T_2 are examples of so called *ad hoc* polymorphism: normally *student* and *employee* are incomparable in the class hierarchy and therefore the set of tuples in p may be heterogeneous. □

The above examples illustrate the importance of TDs for specifying the typing of predicate symbols. It follows from the definitions that whenever a predicate symbol, p, has at most one *ground* signature-atom associated with it, then the specifics of TDs that tag this atom are immaterial. For instance, in Example 5.4, the TDs associated with the signature of *husband* or *parent* do not matter much: $husband\langle man, woman/\emptyset \rightarrow \omega\rangle$ would yield the same semantics. Indeed, according to the definitions in Section 2.2 and the notion of well-typedness given below (in Section 5.4), both signatures mean the same: an atom $husband(a, b)$ is consistent with any of the above signatures for *husband* if and only if $\text{ISA} \models a : man \wedge b : woman$. (It is important to understand, however, that even though $husband\langle man, woman/\emptyset \rightarrow \omega\rangle$ and $husband\langle man, woman/\emptyset \rightarrow \emptyset\rangle$ result in the same typing for *husband*, these two signatures are logically inequivalent.)

The difference in tagging shows up in the presence of multiple signatures for the same predicate in the presence of the subtype relationship. One example to this effect is given in Example 5.5. Similarly, the typing of *play* in Example 5.4 would have had a different meaning if we changed the TDs, e.g., $play\langle jr\_team, jr\_team/\emptyset \rightarrow \emptyset\rangle$. Such tagging would have made $play(giants, pee\_wee)$ type-compatible with the signatures for *play*. Even more subtle cases arise when parametric types interact with subtyping. For instance, suppose that $int : real$ and the signature for *append* is $append\langle list(T), list(T), list(T)/\emptyset \rightarrow \emptyset\ \rangle$. Then $append([1], [2, 3], [2.3, 1.4])$ will be consistent with this signature, namely with its instance $append\langle list(real), list(real), list(real)/\emptyset \rightarrow \emptyset\ \rangle$. Even the more intuitive typing $append\langle list(T), list(T), list(T)/1, 2 \rightarrow 3\rangle$ is not completely adequate, since it permits atoms such as $append([0.5], [1, 3.2], [3, 4])$, i.e., appending a pair of lists of non-integers that yields a list of integers. This does not contradict the signature since *int* is a *real*, according to our assumption. A better signature would be $append\langle list(T), list(T), list(T)/1, 2 \rightarrow 3; 3 \rightarrow 1, 2\rangle$. For example, the atom $append([0.5], [1, 3.2], [3, 4])$ is not consistent with this signature, since the TD $\{3\} \rightarrow \{1, 2\}$ requires the first two arguments of append to be of the type $list(int)$, whenever so is the third argument.

Observe that, in contrast to our approach, predicate modes for *append* [DH88] can express only what amounts to a single TD $\{1, 2\} \rightarrow \{3\}$, and therefore the anomaly exhibited above is inevitable. This shows that several subtleties of the interplay between argument types elude the mode declaration idea.

5.3 Data Definition

A *data definition*, DATA, is a set of rules where data-atoms are in the head, while the rule-bodies may consist of arbitrary, possibly negated, literals (is-a, data, or signature). The purpose of data definitions is to define the meaning of predicates. We call DATA the "data definition" because it defines the object-level data, such as $parent(tom, john)$ or $play(giants, yankees)$; in contrast, class-hierarchy and type declarations specify what can be viewed as "meta" or "higher-order" data.

It is easy to see that every data definition that consists of definite Horn rules has a unique least Herbrand model, which can be taken as its semantics. With negative literals in rule-bodies, there is a number of different competing semantics [GRS88, GL88, Van89, AV88, KP88], each of which can easily be adapted to $\mathcal{TPC}$. All of these semantics have in common that the logical entailment is defined with respect to some subset of models, called *canonic* models (which may be different for different proposals).

We can thus assume that there is a subclass of *canonic* models among all the $\mathcal{TPC}$-models and that *untyped canonic models* of $P = \text{DATA} \cup \text{ISA} \cup \text{TYPE}$ are precisely the canonic models within the class of all untyped models of P. (It is essential here that we talk about models of P and not just of DATA.) A *typed* canonic model is any typed semantic structure that is also an untyped canonic model.

Since all our examples employ definite Horn clauses only, the reader can handily think that canonic models coincide with $\ll^{min}$-minimal models defined in Section 5.1. Precise definitions of canonic models in $\mathcal{TPC}$ according to the prefect [Prz88], well-founded [GRS88, Van89], or stable model semantics [GL88] can be given along the lines of the corresponding developments in $\mathcal{PC}$ and are beyond the scope of this paper. We just note that in the case of well-founded models the glue conditions of Section 2.2 need to be modified to account for the fact that well-founded models are 3-valued.

It follows from the definition that signature-atoms are allowed to appear in the body of the rules in data definitions. This makes $\mathcal{TPC}$ a *reflexive* theory where programs can examine and reason about the typing of predicates via the same mechanism as the one used for querying data. For instance, the query

$$?- \ play\langle X, Y/\emptyset \rightarrow \emptyset\rangle \ \& \ play(V, W) \ \& \ V : X \ \& \ W : Y \tag{2}$$

would return all the typings for *play* such that this predicate *actually* has a data-tuple that conforms to this typing. The latter condition is not a trivial one, since it is quite possible for a polymorphic predicate not to have any data-tuple conforming to some of its legal typings. For instance, if in Example 5.4 *play* had no data about junior teams, this query would have returned the answer $X = Y = team$ (since $play\langle team, team/\emptyset \rightarrow \emptyset\rangle$ is a logical consequence of the clause $play\langle team, team/1 \rightarrow 2;\ 2 \rightarrow 1\rangle$ in Example 5.4), leaving the other candidate answer, $X = Y = jr\_team$, out.

Notice that in (2) the result depends on the available object-level data *as well as* type declarations. We envision that queries against type declarations of $\mathcal{TPC}$ logic programs will be quite useful for debugging. In [KLW90] such queries are shown to be useful for browsing knowledge bases.

5.4 The Semantics of Well-Typedness

We now pull together the various pieces discussed so far and define a semantics for $\mathcal{TPC}$ programs.

Example 5.6 The following is a complete definition for *append*, which includes a class-hierarchy declaration, a type declaration, and a data definition:

$nil : list(T)$ % Class-hierarchy declaration
$cons(X, L) : list(T) \ \longleftarrow \ X : T \ \& \ L : list(T)$

$append\langle list(T), list(T), list(T)/1, 2 \rightarrow 3;\ 3 \rightarrow 1, 2\rangle$ % Type declaration

$append(nil, L, L) \ \longleftarrow \ L : list(T)$ % Data definition
$append(cons(X, L), M, cons(X, N))$
$\quad \longleftarrow \ append(L, M, N) \ \& \ X : T \ \& \ N : list(T).$

□

Semantics of logic programs is now defined with respect to the class of typed canonic models. Since this is only a subset of all models of P, the corresponding logical entailment is nonmonotonic and is denoted by "$|\!\approx$": Given a logic program P, we write $P \mid\!\approx \psi$ if and only if for all typed canonic models M of P, $M \models \psi$.

Now we can formally define type errors and well-typed programs. A logic program has a *type-error* if it has no typed canonic models but does have untyped ones. A program is *well-typed* (or *type-correct*) if and only if it has no type error. This *model-theoretic* notion of type error is unique to $\mathcal{TPC}$; it separates programs that have no models due to some intrinsic logical inconsistency in the data from programs that lack models because of type-incompatibility.

Note that in Example 5.6 the program may not be well-typed without the last two conjuncts in the body of the last clause. that—perhaps unexpectedly—this program may be ill-typed. Indeed, suppose our type system contains only *int* and *string*, which do not have a common supertype. Then, because of the unrestricted variable X in the head of the last clause for *append*, the data definition may produce a fact of the form *append*([1], ["*abc*"], [1, "*abc*"]). Since *int* and *string* have no common supertype, every untyped canonic model for *append* must be such that $I_{\mathcal{T}}(append)$ has no type-constraint that covers ⟨[1], ["*abc*"], [1, "*abc*"]⟩ and hence the second well-typing condition in Section 2.2 cannot be satisfied. On the other hand, if Isa had the "type of all things" (see (1) of Section 5.1), then the program in Example 5.6 would have been well-typed.

In general, in the presence of polymorphic types the programmer must be careful with variables that are not restricted by positive literals in the rule body, and it might be a good programming practice to explicitly indicate which types are allowed for such variables.

5.5 Discussion

Various approaches to types in logic programming can be classified as syntactic or semantic. Syntactic approaches, usually found in the works on type checking (e.g., [MO84, DH88, Smo88, Fru89, YFS90, Jac90, HT90]), see the notion of type-correctness as a syntactic concept. Their notion of well-typing is defined by a set of "well-formedness rules" that are imposed directly on the syntactic structure of logic programs. In contrast, semantic approaches, where $\mathcal{TPC}$ and most of the works on type inference (e.g., [Mis84, YS87, Xu89]) belong, define the concept of type-correctness in model-theoretic or set-theoretic terms. For instance, in $\mathcal{TPC}$, a program is regarded as well-typed if and only if the intended model of the program satisfies the well-typing conditions of Section 2.2.

As a rule, the criteria for well-typedness in syntactic approaches are stricter than in semantic approaches. For instance, a program may be ill-typed according to Mycroft and O'Keefe [MO84] but not in $\mathcal{TPC}$. As an example, let P have an is-a hierarchy declaration Isa such that Isa $\not\models a : int$ and let the rest of P include

$$\begin{array}{l} p\langle int/\,\rangle \\ q\langle all/\,\rangle \\ p(a) \longleftarrow false \end{array} \qquad (3)$$

where $false$ is a propositional constant such that $P \not\models false$. This program is well-typed in $\mathcal{TPC}$, for $p(a)$ is never derived. In Mycroft-O'Keefe's type system [MO84] this program is clearly ill-typed because $p(a)$ is ill-formed regardless of whether it can be derived or not.

One advantage of syntactic approaches is that the type-checking problem is (usually) decidable, since only finitely many program components need be examined. In semantic approaches, however,

type checking is undecidable, in general. The following lemma shows that this is also the case in $\mathcal{TPC}$.

Lemma 5.1 *The class of the well-typed Horn programs in $\mathcal{TPC}$ is not recursively enumerable.*

Proof: The proof is by a reduction from the *query emptiness* problem. Query emptiness is a problem of whether the set of solutions to a logical query $?- q(...)$ defined by a program P is empty. The class of all empty queries is known to be nonrecursively enumerable. (For instance, in the proof of Theorem 2-6, Manna [Man74, pages 105-106] shows that for every Post system S there is a Horn program P and a query $?- p(Z, Z)$ such that this query has *no* answers if and only if S has *no* solution.)

The reduction from query emptiness to type-correctness is carried out as follows. Let P be a program and q be a query predicate (it suffices to consider only atomic queries). We can also assume that q appears only in the heads of the rules in P, which can be ensured by a simple program transformation. We now construct a signature Σ for P so that for every predicate p in P, except q, Σ contains the signature atom $p\langle X_1, ..., X_k/\emptyset \rightarrow \emptyset\rangle$. However, Σ does not contain any signature for q. Clearly, the $\mathcal{TPC}$ program $P \cup \Sigma$ is well-typed if and only if $?- q(X_1, ..., X_n)$ has no answers. □

Another advantage of syntactic approaches is that they go farther (than semantic approaches) towards capturing the notion of ill-typed atoms in rule bodies, i.e., atoms that are instantiated with type-incompatible arguments. The importance of capturing type inconsistency in rule bodies lies in the fact that this leads to the notion of *useless clauses* [YS87], i.e., clauses that do not affect the meaning of a program. Let the program P be:

$$\begin{array}{ll} r\langle string/\,\rangle & r(X) \longleftarrow p(X)\,\&\,q(X). \\ p\langle string/\,\rangle & \\ q\langle int/\,\rangle & \end{array} \qquad (4)$$

Suppose *int* and *string* have no common subtypes and elements. Then any type assignment for X makes either $p(X)$ or $q(X)$ ill-typed because *string* and *int* do not share common subtype or instances. As a consequence, P would be regarded as ill-typed by syntactic approaches, such as Mycroft-O'Keefe's [MO84]. In contrast, the notion of well-typedness in $\mathcal{TPC}$ given earlier does not capture the above intuition and is too weak to classify P as ill-typed.

There are deficiencies in syntactic approaches, too. Many such approaches insist that well-typed programs should not "go wrong" [Mil78, MO84]. Although this requirement causes problems even without subtyping, the latter makes it unavoidable to think of better criteria for syntactic well-typedness. For instance, the not-going-wrong requirement makes it illegal to ask for the intersection of a pair of predicates that define heterogeneous sets. Suppose p is true of a term whenever it belongs to the class *employee* and q is true of and only *student* objects. Then the query $?- p(X)\,\&\,q(X)$ is likely to "go wrong" whenever $p(X)$ is instantiated to an employee-nonstudent fact or when $q(X)$ is instantiated to a student-nonemployee fact.

Another problem is in applying syntactic well-typedness to programs with negation. Consider the following example:

$$\begin{array}{ll} r\langle all/\,\rangle & r(X) \longleftarrow p(X)\,\&\,\neg q(X). \\ p\langle all/\,\rangle & \\ q\langle int/\,\rangle & \end{array}$$

The intention of the last clause is to compute the set difference of p and q. Here, the variable X should be given the type *all* to cover the type of p. However, under this type assignment X becomes type-incompatible with q and, say, Mycroft-O'Keefe's well-typing rules would label this program as ill-typed.

None of these problems arise in $\mathcal{TPC}$, which considers the above two programs as type-correct, since they do not entail ill-typed atoms. Actually, these can also be rendered type-correct due to the syntactic approaches if we replace p and q by p' and q', declared as $p\langle all/\,\rangle$ and $q\langle all/\,\rangle$, and define $p'(X) \longleftarrow p(X)$ and $q'(X) \longleftarrow q(X)$. However, this way of circumventing type-checking is hardly an elegant solution to the problem of heterogeneous set intersection and set difference.

With all the merits and demerits of the two major points of view discussed above, it is intriguing to see if there exists a reasonable notion of type-correctness that combines the benefits of both approaches. Some of the problems with the notion of type correctness in $\mathcal{TPC}$, such as the one with (3), can be easily rectified; other deficiencies are harder to eliminate, e.g., the problem in (4).

6 Semantics of Type Inference

Type inference is a procedure that determines the "most appropriate" type declaration for a program. Compared to type checking, which works only when complete type information is given, type inference is more widely applicable since it can be used to extract typing from programs with partial type information or none at all. In logic programming, type inference can be used as a debugging tool as follows: Once the typing is inferred, the programmer can examine it and see if this is what was intended. If unwanted types are detected, the programmer may suspect a program error. For instance, consider the following defective program that is supposed to sum up elements of a list:

$$\begin{array}{l} sum(cons(X, nil), X) \longleftarrow X : int \\ sum(cons(X, L), Z') \longleftarrow sum(Z, L)\ \&\ plus(X, Z, Z'). \end{array}$$

The mistake here appears to be in the second rule, where the arguments in $sum(Z, L)$ are misplaced. Assuming that only the trivial TDs are associated with *sum*, the programmer would expect the following signature: $\{\, sum\langle list(int), int/\emptyset \rightarrow \emptyset\rangle \,\}$. However, the inferred typing can only be $\{\, sum\langle all, int/\emptyset \rightarrow \emptyset\rangle \,\}$, because in the rule-body L is bound to arguments of type *int* instead of $list(int)$ and therefore $cons(X, L)$ is not of the type $list(int)$ (*all* is the type of everything, defined in (1)). This discrepancy may alert the programmer to the mistake in the second rule.

Type inference can also be used for query optimization, e.g., to detect that certain goals are bound to fail. For instance, suppose the set of signatures inferred from a program is $\{\, husband\langle man, woman/\emptyset \rightarrow \emptyset\rangle \,\}$. The system could then immediately reject queries such as $?-\ husband(mary, X)$ and inform the user that the first argument of *husband* is expected to be a *man* (assuming $mary : man$ does not logically follow from the program).

Type inference algorithms usually need the following input: a class-hierarchy declaration, ISA, a data definition, DATA, and a type-dependency declaration, TD. A *type-dependencies declaration* (abbr., a TD-declaration) is a mapping that assigns to each predicate that occurs in DATA a set of TDs. The objective of type inference is to infer a set of signatures TYPE for predicates appearing in DATA, such that $P \stackrel{\text{def}}{=} \text{ISA} \cup \text{DATA} \cup \text{TYPE}$ is a well-typed program, where every signature in TYPE is of the form $p\langle \vec{t}/\text{TD}(p)\rangle$ (i.e., the TDs in each such signature must be precisely the ones that are associated to p via TD). Furthermore, TYPE must be the *strongest* type declaration for which P is well-typed. The latter requirement is important, for otherwise signatures like $sum\langle all, all/\emptyset \rightarrow \emptyset\rangle$ will always do the job.

More precisely, given a pair of type declarations T and T', we say that T is *stronger* than T' modulo a class-hierarchy ISA, denoted $T \trianglerighteq T'$ **mod** ISA, if and only if:

- For every data definition DATA, whenever $\text{ISA} \cup \text{DATA} \cup T$ is well-typed, $\text{ISA} \cup \text{DATA} \cup T'$ is also well-typed.

A type declaration T is *nonredundant* (with respect to ISA)if there is no T' such that $T' \subset T$ and T and T' are equally strong (i.e., $T \trianglerighteq T'$ **mod** ISA and $T' \trianglerighteq T$ **mod** ISA).

Example 6.1 TDs play a crucial role in determining the condition $T \trianglerighteq T'$, as shown next. Let

$$\begin{array}{lll} T: & p\langle student, student/\emptyset \rightarrow \emptyset\rangle, & T': \quad p\langle student, student/\emptyset \rightarrow \emptyset\rangle \\ & p\langle employee, employee/\emptyset \rightarrow \emptyset\rangle & \\ T'': & p\langle student, student/\emptyset \rightarrow \omega\rangle, & T''': \quad p\langle student, student/\emptyset \rightarrow \omega\rangle. \\ & p\langle employee, employee/\emptyset \rightarrow \omega\rangle & \end{array}$$

Suppose that the ISA-part is empty here. Then every data tuple that is type-compatible with T' is also compatible with T and thus $T' \trianglerighteq T$. However, with different TDs the situation may reverse itself. For instance, the tuples that are compatible with T'' are now also compatible with T''', hence $T'' \trianglerighteq T'''$. □

The following lemma gives a sufficient condition for one signature to be stronger than another:

Lemma 6.1 *Let T_1 and T_2 be collections of signatures that associate each predicate (occurring in these signatures) with exactly one TD (the TD may be different for different predicates, though). Then $T_1 \trianglerighteq T_2$* **mod** ISA *if the following conditions hold:*

1. *For every ground instance $p\langle t_1, ..., t_n/\alpha \rightarrow \beta\rangle$ of a signature in T_1, there is a ground instance $p\langle t'_1, ..., t'_n/\alpha \rightarrow \beta\rangle$ of some signature in T_2, such that* ISA $\models t_i : t'_i$, *for $i = 1, ..., n$; and*
2. *For every ground instance $p\langle t'_1, ..., t'_n/\alpha \rightarrow \beta\rangle$ of a signature in T_2, if $\beta - \alpha$ is non-empty then there is a ground instance $p\langle t_1, ..., t_n/\alpha \rightarrow \beta\rangle$ of a signature in T_1, such that* ISA $\models t'_i : t_i$, *for all $i \in \alpha$, and* ISA $\models t_j : t'_j$, *for all $j \in \beta - \alpha$.*

Proof: It easily follows from the definitions that for every ground data atom A, if ISA $\cup \{A\} \cup T_1$ is well-typed, then ISA $\cup \{A\} \cup T_2$ is also well-typed. Hence $T_1 \trianglerighteq T_2$ **mod** ISA. □

It should be noted that the user does not always have to supply the type-dependency declaration TD. In extreme cases, the system can assume all TDs to be trivial (e.g., $\emptyset \rightarrow \emptyset$, $\omega \rightarrow \emptyset$) or, alternatively, the strongest ones (i.e., $\emptyset \rightarrow \omega$). For instance, Mishra [Mis84] and Xu and Warren [XW88, Xu89, XW90] implicitly assume all type dependencies to be trivial. User-supplied TDs may improve the accuracy of type inference algorithms, for in case of trivial TDs too many signatures may be inferred, leading to information glut. In contrast, if only the strongest TDs $\emptyset \rightarrow \omega$ are considered, too general signatures are likely to be inferred, which may not be informative enough.

The user can further control type inference by supplying a set of ground terms as a *type base*. The type inference procedure will then look only for those signatures whose type-names are drawn from the type base. There are two reasons to supply user-defined type base. One is to guide type inference towards returning more informative answers. The other is to improve the speed of type inference by considering a relatively small type base.

The semantics of type inference can be understood as a form of abductive reasoning. Given a quadruple $\langle$ISA, DATA, TD, BASE$\rangle$, where ISA is a class declaration, DATA is a data definition, TD is a TD-declaration, and BASE is a type base, an *ideal inferred type declaration* is a type declaration TYPE such that

- ISA $\cup$ DATA $\cup$ TYPE is a well-typed program;

- TYPE is a set of signatures that mention only the type-names appearing in BASE;
- For each signature $p\langle T_1, ..., T_n/\Gamma\rangle$ in TYPE, Γ is the set of signatures associated with p via TD;
- TYPE is a strongest (modulo ISA) nonredundant type declaration satisfying the above conditions.

Despite its name, an ideal inferred type declaration may not be unique. For instance, suppose ISA consists of the following rules:

$$X:c \longleftarrow X:a \,\&\, X:b \,\&\, X \neq a \,\&\, X \neq b \qquad X:a \longleftarrow X:c \,\&\, X:d \,\&\, X \neq c \,\&\, X \neq d$$
$$X:d \longleftarrow X:a \,\&\, X:b \,\&\, X \neq a \,\&\, X \neq b \qquad X:b \longleftarrow X:c \,\&\, X:d \,\&\, X \neq c \,\&\, X \neq d.$$

This means that intersection of the extensions of classes a and b coincides with that of the extensions of c and d. It is easy to see that the sets of signatures $S_1 = \{p\langle a/\emptyset \to \omega\rangle, p\langle b/\emptyset \to \omega\rangle\}$ and $S_2 = \{p\langle c/\emptyset \to \omega\rangle, p\langle d/\emptyset \to \omega\rangle\}$ are equally strong modulo ISA and none of them is redundant. Therefore, if S_1 is an ideal inferred type for some program then so is S_2, provided that BASE contains a, b, c, and d.

Let DATA be a data definition. A *naive type inference* algorithm for $\langle$ISA, DATA, TD, BASE$\rangle$, where TD assigns a trivial TD to every predicate, works as follows:

1. [Evaluate]: Use the bottom-up evaluation method to compute the canonic model $\mathcal{W}$ of DATA.
2. [Transform]: For each data atom $p(s_1, ..., s_n)$ in $\mathcal{W}$, find a minimal supertype t_i of s_i in BASE, $i = 1, ..., n$; set $\mathcal{V} := \mathcal{V} \cup \{p\langle t_1, ..., t_n/\emptyset \to \emptyset\rangle\}$.
3. [Postprocess]: Remove redundancies in $\mathcal{V}$. Return $\mathcal{V}$ as the inferred type declaration.

Unfortunately, this algorithm may not stop if $\mathcal{W}$ is infinite. However, we can approximate $\mathcal{W}$ using a particular group of type bases with finite domains. For instance, for the type bases Π_n in [XW88] that contain terms up to a certain depth, the termination of the algorithm is guaranteed. We will discuss the type bases Π_n and the type inference algorithm of [XW88] in Section 9.3.

The above type inference can be combined with partial type declarations, which works like this: The user declares signatures for *some* of the predicates (possibly none, if "pure" type inference is desired). Then the system checks types of the predicates whose signatures are declared in the partial type declaration, and infers signatures for predicates whose signatures are not mentioned.

More specificly, let TYPE_{init} be a partial type declaration for a program P. For every predicate p that appears in P but not in TYPE_{init}, a signature $p\langle all, ..., all/\emptyset \to \emptyset\rangle$ is added to TYPE_{init}, where *all* is the largest type, defined by the clause $X : all$. Such signatures impose no restriction on p. Then the system uses a type-checking procedure to decide whether the program P is well-typed under the augmented type declaration. If P is well-typed, then the predicates in TYPE_{init} satisfy the partial type declaration. At the last stage, a type-inference procedure is applied to the program, as in pure type-inference. Finally, signatures for the predicates that are not mentioned in TYPE_{init} are reported to the user.

7 Separating Individuals from Classes

So far, $\mathcal{TPC}$ did not distinguish between classes and individuals, and any individual (e.g., *john*) was also regarded as a class that contains at least itself as an instance (since, e.g., $john : john$ is a tautology). However, it is often useful to distinguish classes from individuals.

First, users do not expect class names to appear among answers to the "ordinary" queries. Second, separating class names from individuals is even more important for the typing; otherwise certain parametric signatures may become too strong. Consider the following signature:

$$same\_type\langle X, X/1 \to 2; 2 \to 1\rangle, \tag{5}$$

that declares both arguments of *same_type* to have precisely the same type or, equivalently, to be instances of the same set of classes. However, if individuals are also classes then this would be clearly too strong a condition. For instance, $same\_type(1,2)$ is not well-typed relatively to the above signature, since the variable X will range over 1 and 2 as well, and hence 1 would have to belong to the class 2 and vice versa. This implies that $1 = 2$ in all untyped models. Since in the standard interpretation of integers $1 \neq 2$, $same\_type(1,2)$ would be incompatible with (5). In fact, it is easy to see that only the atoms of the form $same\_type(t,t)$, where t is a term, are compatible with (5).

There are several different ways to deal with this problem. In our opinion, the most elegant solution consists in splitting the set of $\mathcal{TPC}$ terms into two different sorts: classes and individuals. Formally, this is done as follows. All variables and function symbols (and constants as a special case) can be split into three categories: those whose range of values is the sort of individuals, the sort of classes, and those whose range is the universal sort (the union of the first two sorts). There is no need to impose sorts on the arguments of function and predicate symbols. Syntacticly, we can distinguish the class-valued symbols by attaching to them the prefix "$\sharp$", e.g., $\sharp X$, $\sharp union$; the individuals-valued symbols can be prefixed with "!", e.g., $!Y$, $!cons$; and the symbols whose range is universal will begin with a letter. Then, we postulate that a term belongs to a class-sort, the sort of individuals, or the universal sort if and only if its outermost symbol (a function symbol, a constant, or a variable) is prefixed with "$\sharp$", "!", or starts with a letter, respectively.

The next step would be to restrict the language in such a way that

- The terms in the signature atoms must be class-terms (i.e., their outermost symbol must be prefixed with a "$\sharp$"-sign);
- For an is-a term, $Cl : SupCl$, we may require $SupCl$ to be a class-term (i.e., it should really be $\sharp SupCl$).

These restrictions ensure that individuals will not be used as classes and that signatures do not meddle with individuals. For instance, (5) above will not be a syntacticly correct formula; in the sorted version of $\mathcal{TPC}$, a correct signature corresponding to (5) would be

$$same\_type\langle \sharp X, \sharp X/1 \to 2; 2 \to 1\rangle,$$

which imposes the desired typing constraint.

Semantics of the sorted version of $\mathcal{TPC}$ is a simple modification of the model theory given in Section 2.2. In a nutshell, domains of semantic structures now become disjoint unions of the subdomains for classes and individuals, respectively: $U = \sharp U \uplus !U$, where $\uplus$ denotes the operator of disjoint union of sets. Function symbols of the form $\sharp f$ (or $!f$) are now interpreted by the maps $U^k \longmapsto \sharp U$ (resp., $U^k \longmapsto !U$). Symbols starting with a letter are interpreted by functions $U^k \longmapsto U$. Notice that since arguments of function symbols are not sorted, the domain of these functions is U^k, not $\sharp U^k$ or $!U^k$. Another change involves the definition of variable-assignments. An assignment ν now maps variables of the form $\sharp X$ into $\sharp U$ and those of the form $!X$ to $!U$. The details are straightforward and are left to the reader.

The proof theory does not require many changes either, except that the definition of substitutions must be modified to preserve sorts, i.e., substitutions should map the "♯"-variables (or "!"-variables) to "♯"-terms (resp., "!"-terms) only; unrestricted variables can be mapped into any term.

8 Evaluation

Query evaluation in $\mathcal{TPC}$ is based on the proof theory of Section 4. First, note that in terms of the proof theory, the purpose of type checking is precisely that of the elimination (or, at least, reducing) of the need in applying the well-typing inference rules of Figure 1. Indeed, once the user has type-checked his program using one of the schemes discussed earlier (pure type-checking, pure type-inference, or type inference under partial type declarations), the use of the well-typing rules is redundant, since all atomic formulae entailed by this program will be guaranteed to conform to the declared (or inferred) typing.

For type-correct programs it can be shown that a $\mathcal{TPC}$ analogue of the SLD-resolution [Llo87] is sound and complete for Horn programs. Similarly, SLDNF-resolution can be defined and the corresponding classical results carry over to $\mathcal{TPC}$ programs.

9 Case Study

9.1 Syntactic Approaches to Type Checking

Majority of the approaches to type checking in logic programming (e.g., [MO84, DH88, Smo88, Fru89, YFS90, Jac90, HT90]) can be classified as syntactic. The notion of type-correctness is established via a set of well-typing rules, defined inductively on the syntactic structure of formulae. Consider first Mycroft-O'Keefe's type checking system [MO84]. Their well-typing rules are derived from the slogan that "well-typed programs do not go wrong", coined in by Milner [Mil78]. By "not going wrong" in [MO84] is meant that subgoals are not called with bindings that disagree with the declared typing of the corresponding predicates. Although this principle may be appropriate in functional programming, it cannot be fully applied to logic programs since, in fact, logic programs *never* go wrong in the logical sense. Even if a subgoal with "wrong" arguments is generated, it is discarded later because of the subsequent mismatch with the database. This is different from functional programming where ill-typed input may cause run-time errors because functions are usually undefined outside their declared argument-types. Consider a variation on the program (3) in Section 5.5 that does not involve subtyping and therefore can be represented in Mycroft-O'Keefe's system:

$$\begin{array}{ll} p\langle string/\emptyset \rightarrow \emptyset\rangle & p(\text{“}abc\text{”}) \\ p\langle int/\emptyset \rightarrow \emptyset\rangle & p(1) \\ q\langle int/\emptyset \rightarrow \emptyset\rangle & q(1). \end{array} \qquad (6)$$

The typing for p says that its arguments are either of the type *string* or *int*; q's arguments are integers only. According to the "not-going-wrong" condition, the query $?- p(Z)\ \&\ q(Z)$ is ill-typed since it leads to an ill-typed subgoal $?- q(\text{“}abc\text{”})$. On the other hand, during the course of the top-down evaluation, the subgoal $?- q(\text{“}abc\text{”})$ is discarded immediately, and only the well-typed answer $Z = 1$ is returned. What this effectively means is that an innocent query that asks for the intersection of two heterogeneous sets (like p and q above) is illegitimate according to [MO84]. Dietrich and Hagl [DH88] and Jacobs [Jac90] have a similar problem if p is typed as $p\langle all/\emptyset \rightarrow \emptyset\rangle$. Hence, the well-typing

conditions in [MO84, DH88, Jac90] are too restrictive. It is easy to see that in $\mathcal{TPC}$ the above program is well-typed.

Furthermore, the original approach in [MO84] has no model-theoretic semantics. As a result, some confusion may arise if the typed logic programs are interpreted naively. For instance, the program

$$\begin{array}{l} p\langle int/\emptyset \rightarrow \emptyset \rangle \\ p(X) \end{array} \tag{7}$$

is well-typed according to [MO84] because for any well-typed query, its top-down evaluation generates only well-typed subgoals. However, this program logically entails ill-typed atoms, such as $p(\text{“}abc\text{”})$ (assuming “abc” occurs somewhere else in the program), which suggests that (7) is going to misbehave under bottom-up evaluation.

The recent work by Reddy [Red91] rectifies the problem by reconstructing Mycroft-O'Keefe's approach on the basis of the many-sorted algebra [GTW78]. In particular, predicate p in (7) would have the sort *int* and so will the variable X. As a result, $p(X)$ would no longer generate ill-typed atoms and (7) becomes equivalent to the following well-typed $\mathcal{TPC}$ program:

$$\begin{array}{l} p\langle int/\emptyset \rightarrow \emptyset \rangle \\ p(X) \longleftarrow X : int \end{array} \tag{8}$$

The program (8) is called the *one-sorted* form of (7). There is a standard procedure to convert sorted programs into the one-sorted form (see [End72, page 279]). The following theorem establishes a correspondence between the well-typed programs of [MO84] (in Reddy's reconstruction) and those of $\mathcal{TPC}$.

Theorem 9.1 *Let P be a well-typed program with respect to a type assignment Σ in the sense of Mycroft-O'Keefe-Reddy. Let P^* be the one-sorted version of P. Then there is an is-a hierarchy declaration* Isa *and a type declaration* Type *corresponding to Σ, such that* Isa $\cup$ Type $\cup$ P^* *is well-typed in $\mathcal{TPC}$.*

Mycroft-O'Keefe's type system can be extended to support inclusion polymorphism using the idea of parametric order-sorted logic. Such extensions are described in [Smo88] and [HT90]. Results similar to Theorem 9.1 also hold about the relationship of these extensions to $\mathcal{TPC}$.

9.2 Mishra's Type Inference System

Mishra [Mis84] proposed an inference-based type system for Prolog, based on the idea of automatic type inference in ML. This type system is able to infer typings for all predicates in any logic program without explicit type declarations. The inferred type declarations are represented as *regular trees* and satisfy the well-typing principle according to which the inferred typing must cover all atoms that can be successfully derived from the program. This principle is widely accepted in the logic programming community, and several other researchers [Zob87, Azz88, PR89] further extended Mishra's work by proposing different type inference algorithms. In this subsection we explain how $\mathcal{TPC}$ captures the notion of Mishra's regular trees and show that his well-typing principle is consistent with the notion of type-correctness in $\mathcal{TPC}$. In particular, this means that the type-inference algorithms developed in [Mis84, Zob87, Azz88, PR89] can be used in $\mathcal{TPC}$, provided that the type-base is restricted to regular trees.

Example 9.1 Consider the standard Prolog program for *append*:

$$append(nil, K, K)$$
$$append(cons(X, L), M, cons(X, N)) \longleftarrow append(L, M, N).$$

The typing computed by Mishra's algorithm would be

$$T(append) = T_1 \sharp T_2 \sharp T_3, \text{ where } \begin{array}{l} T_1 = Z\,!\; nil + cons(X, Z) \\ T_2 = Y \\ T_3 = Z\,!\; Y + cons(X, Z). \end{array}$$

Here, T_1, T_2, and T_3 are regular trees that describe the types of each of the three arguments of *append*; "$\sharp$" is the argument separator. Regular trees are extensions of the first-order terms with the addition of the union operator "+" and the fixpoint operator "!". The former is used to construct unions of types, while the latter is a means of creation of recursive types. Each regular tree T is associated with a set of ground terms, called *instances* of T. For example, $Z\,!\; nil + cons(int, Z)$ is associated with the terms nil, $cons(1, nil)$, $cons(2, cons(1, nil))$, and so on.

The typing for *append*, $T_1 \sharp T_2 \sharp T_3$, is the Cartesian product of the sets denoted by T_1, T_2 and T_3. It covers all *append*-tuples that can be successfully derived from the program. That is, if $append(s_1, s_2, s_3)$ is derived then the tuple $\langle s_1, s_2, s_3 \rangle$ must be an instance of $T_1 \sharp T_2 \sharp T_3$. □

Regular trees can be represented in $\mathcal{TPC}$ as follows. First, we transform each regular tree t into a first-order term, $tr(t)$. New function symbols—*union*, *fixpt*, and *seed*—are used to express the union and the fixpoint operators.

- For a constant c, $tr(c) \stackrel{\text{def}}{=} c$.
- For a regular tree $f(t_1, ..., t_n)$, $tr(f(t_1, ..., t_n)) \stackrel{\text{def}}{=} f(tr(t_1), ..., tr(t_n))$, where f is a function symbol.
- For a union-expression $r + s$, $tr(r + s) \stackrel{\text{def}}{=} union(tr(r), tr(s))$.
- For a fixpoint-expression τ of the form $Z\,!\, p + q(Z)$, where $q(Z)$ is a regular tree involving a variable Z, $tr(\tau) \stackrel{\text{def}}{=} fixpt(tr(p), tr(q(seed))$.

The instances of regular trees can be defined using class-hierarchy declarations. Given a set of regular trees **T**, the class-hierarchy declaration corresponding to **T** consists of the following rules:

(1) For every function symbol f:

$$f(X_1, ..., X_n) : f(Y_1, ..., Y_n) \longleftarrow X_1 : Y_1 \;\&\; \ldots \;\&\; X_n : Y_n.$$

(2) For the union operator:

$$X : union(X, Y)$$
$$Y : union(X, Y).$$

(3) For every fixpoint expression $Z\,!\, p + q(Z)$ in **T**:

$$tr(p) : fixpt(tr(p), tr(q(seed)))$$
$$tr(q(Z)) : fixpt(tr(p), tr(q(seed))) \longleftarrow Z : fixpt(tr(p) \;\&\; tr(q(seed))).$$

Rule (1) here says that every function symbol is monotone with respect to types. The second rule says that for every a and b, $union(a, b)$ contains (in the least model) precisely the instances of the classes a and b. The last rule defines instances of the class $tr(\tau)$, the translation of the fixpoint expression, by mimicing Mishra's semantics for such expressions.

With the above translation, we can transplant Mishra's type inference on $\mathcal{TPC}$. Let P be a data definition in question. For every predicate p in P, if the typing inferred by Mishra's algorithm is $\tau_1 \sharp \cdots \sharp \tau_n$, then let IsA_P contain the class-hierarchy declaration that corresponds to $\tau_1, ..., \tau_n$ and let Type_P contain the signature $p\langle tr(\tau_1), ..., tr(\tau_n)/\emptyset \rightarrow \emptyset\rangle$. Then IsA_P and Type_P define the typing for p in $\mathcal{TPC}$ that corresponds to the typing generated by Mishra's algorithm.

Example 9.2 Consider the logic program of Example 9.1. The signature for *append* generated by Mishra's algorithm (in the $\mathcal{TPC}$ notation) would be

$$append\langle fixpt(nil, cons(X, seed)), Y, fixpt(Y, cons(X, seed))/\emptyset \rightarrow \emptyset\rangle. \tag{9}$$

The corresponding class-hierarchy declaration IsA_P in $\mathcal{TPC}$ is

$$\begin{array}{l} cons(X_1, X_2) : cons(Y_1, Y_2) \longleftarrow X_1 : Y_1 \ \& \ X_2 : Y_2 \\ Y : fixpt(Y, cons(X, seed)) \\ cons(Y, Z) : fixpt(Y, cons(X, seed)) \longleftarrow Z : fixpt(Y, cons(X, seed)). \end{array}$$

It is worth comparing (9) with a much more natural typing $append\langle list(T), list(T), list(T)/1, 2 \rightarrow 3;\ 3 \rightarrow 1, 2\rangle$ of Example 5.6 or even with $append\langle list(all), list(all), list(all)/\emptyset \rightarrow \emptyset\rangle$ that would have been derived by the naive algorithm described in Section 6. □

Theorem 9.2 *Consider a logic program P such that IsA_P and Type_P are constructed as explained above. Then $\text{IsA}_P \cup \text{Type}_P \cup P$ is a well-typed program in $\mathcal{TPC}$.*

It should be noted, however, that although the typing Type_P produced by Mishra's algorithm is a legal inferred typing for P in $\mathcal{TPC}$, it is not the strongest one. For instance, for *append*, the typing $append\langle list(all), list(all), list(all)/\emptyset \rightarrow \emptyset\rangle$ is stronger than (9).

9.3 Xu-Warren's Type Inference [XW88, Xu89]

In [XW88], the typing of a program is defined via typed interpretations. Given a type base Π, a *typed interpretation* A under Π is a semantic structure where terms are interpreted by the types of Π, and predicates are interpreted as relations over Π. Given a Horn clause program P, the typing of P is defined as the *minimum typed model* of P under Π.

The type-inference algorithm of [XW88] correctly (relatively to their semantics) finds the minimum typed model of any Horn clause program. Unfortunately, no such algorithm can terminate in all cases. As a practical solution, Xu and Warren consider a particular group of type bases, Π_n, for which their type inference algorithm always terminates. Given a natural number n, Π_n consists of a finite set of truncated terms, where for every function symbol f truncation occurs below each n-th repeated nested occurrence of f. Subterms that are removed by truncation are replaced by the special symbol "$*$". For instance, if the language $\mathcal{L}$ has one constant a, a unary functor f, and a 2-ary functor g, then the type base Π_1 would be

$$\begin{array}{l} a,\ f(a),\ f(f(*)), \\ f(g(a,a)),\ f(g(a,f(*))),\ f(g(a,g(*,*))), \\ f(g(f(*),a)),\ f(g(g(*,*),a)), \\ \ldots \quad \ldots \quad \ldots \end{array}$$

A truncated term, say, $f(g(a, f(*)))$ represents the set $\{f(g(a, f(t)))|$ for all ground terms t of $\mathcal{L}\}$. It is shown in [XW88] that their type inference algorithm terminates for every type base Π_n. As an example, consider the following program P that subtracts 1 from its first argument:

$$sub1(succ(zero), zero)$$
$$sub1(succ(X), succ(Y)) \longleftarrow sub1(X, Y).$$

Let the type base be Π_1. The type declaration inferred by the algorithm in [XW88] (represented in the $\mathcal{TPC}$-notation) is:

$$sub1\langle succ(zero), zero/\emptyset \rightarrow \emptyset\rangle$$
$$sub1\langle succ(succ(all)), succ(succ(all))/\emptyset \rightarrow \emptyset\rangle,$$

where "*all*" is the class of all things defined in (1).

For each truncated term t of [XW88], its $\mathcal{TPC}$ counterpart is obtained from t by replacing each occurrence of "$*$" by *all*. The class-hierarchy declaration corresponding to Π_n consists of the declaration for *all* as the class of all things $(X : all)$ and of the monotonicity condition

$$f(X_1, \ldots, X_k) : f(Y_1, \ldots, Y_k) \longleftarrow X_1 : Y_1 \ \& \ \ldots \ \& \ X_k : Y_k$$

per each k-ary function symbol f, $k > 0$. The type-dependency declaration corresponding to the formalism of [XW88] is the simplest possible one: every signature is tagged with a trivial TD.

Theorem 9.3 *Let P be a program and* TYPE *be the type declaration for P generated by Xu-Warren's type inference algorithm. Let* ISA *be the class-hierarchy declaration obtained from Π_n as explained above. Then* ISA $\cup$ TYPE $\cup$ *P is a well-typed $\mathcal{TPC}$-program.*

In [XW90], Xu and Warren extended the framework of [XW88] and introduced type declarations, making it possible to do type checking together with type inference, as in Section 6. Unfortunately, in the presence of inclusion polymorphism the extended theory is not completely adequate, for it is not sufficiently strong and lets certain ill-typed programs to slip through:

$$p(X) \rightsquigarrow man(X) \qquad p(X) \longleftarrow q(X)$$
$$q(X) \rightsquigarrow person(X) \qquad q(mary)$$
$$q(john).$$

Here, the first pair of clauses declares p to be a set of men while q is a set of persons. Intuitively, this program is ill-typed because the query $?- p(Y)$ returns *mary* as one of the answers. This contradicts the declaration of p as a set of men (assuming *mary* is not a *man*). It is not clear whether an adequate solution to the problem of type declarations can be found within the framework of partial interpretations developed in [XW88, XW90].

10 Conclusion

We presented a new logic, typed predicate calculus ($\mathcal{TPC}$), that provides a natural framework for representing logic programs with type declaration and for reasoning about type-correctness of these programs. The logic naturally supports inclusion, parametric, and ad hoc polymorphism. Type declarations in $\mathcal{TPC}$ are more expressive than in [MO84, Xu89, Jac90] and other works, since multiple signatures and functional-style type-enforcement patterns, can be used to capture polymorphic types.

We combined the benefits of declaration-based and inference-based approaches, and showed how type inference under partial type declarations fits in our framework. Most importantly, we gave precise model-theoretic meaning to the notion of type-correctness, independent of any specific type-checking or type-inference procedure. This enables us to accommodate many different type-checking and type-inference algorithms that suit in different situations. As an illustration, we showed how the formalisms of [MO84, Smo88, HT90, Mis84, XW88, Xu89] can be accounted for in our framework. Finally, we note that all results in this paper can be easily extended to typing HiLog [CKW89a, CKW89b]—a recently proposed higher-order logic programming language.

Acknowledgements: We thank Catriel Beeri, Thom Fruehwirth, Dean Jacobs, Prateek Mishra, Uday Reddy, David Warren and Jiyang Xu for their valuable suggestions and illuminating discussions.

References

[AV88] S. Abiteboul and V. Vianu. Procedural and declarative database update languages. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 240–250, 1988.

[Azz88] H. Azzoune. Type inference in Prolog. In *Proceedings of International Conference on Automated Deduction*, pages 258–277, 1988.

[BB79] C. Beeri and P.A. Bernstein. Computational problems related to the design of normal form relational schemes. *ACM Transactions on Database Systems*, 4(1):30–59, March 1979.

[Bru82] M. Bruynooghe. Adding redundancy to obtain more reliable and more readable Prolog programs. In *Proceedings of the First International Logic Programming Conference*, pages 129–133, Marseille, France, 1982.

[CKW89a] W. Chen, M. Kifer, and D.S. Warren. HiLog: A first-order semantics for higher-order logic programming constructs. In *Proceedings of the North American Conference on Logic Programming*, October 1989.

[CKW89b] W. Chen, M. Kifer, and D.S. Warren. HiLog as a platform for database programming languages (or why predicate calculus is not enough). In R. Hull, R. Morrison, and D. Stemple, editors, *Database Programming Languages*, pages 315–329. Morgan-Kaufmann, June 1989.

[DH88] R. Dietrich and F. Hagl. A polymorphic type system with subtypes for Prolog. In *Proceedings of the 2-nd European Symp. on Programming, Lecture Notes in Computer Science 300*, pages 79–93. Springer Verlag, 1988.

[DW86] S.K. Debray and D.S. Warren. Automatic mode inference for Prolog programs. In *IEEE Symposium on Logic Programming*, 1986.

[End72] H.B. Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

[Fru88] T. Fruehwirth. Type inference by program transformation and partial evaluation. In *IEEE Intl. Conf. on Computer Languages*, pages 347–355, Miami Beach, FL, 1988.

[Fru89] T. Fruehwirth. Polymorphic type checking for Prolog in HiLog. In *6th Israel Conference on Artificial Intelligence and Computer Vision*, Tel Aviv, Israel, 1989.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the Fifth Conference and Symposium*, pages 1070–1080, 1988.

[GRS88] A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 221–230, 1988.

[GTW78] J.A. Goguen, J.W. Thatcher, and E.W. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In R.T. Yeh, editor, *Current Trends in Programming Methodology*, pages 80–149. Prentice-Hall, 1978.

[HT90] P. Hill and R. Topor. A semantics for typed logic programs. manuscript, 1990.

[Jac90] D. Jacobs. Type declarations as subtype constraints in logic programming. In *Proceedings of the ACM SIGPLAN-90 Conference on Programming Language Design and Implementation*, pages 165–173, June 1990.

[KH85] T. Kanamori and K. Horiuchi. Type inference in Prolog and its application. In *Intl. Joint Conference on Artificial Intelligence*, pages 704–707, 1985.

[KL89] M. Kifer and G. Lausen. F-logic: A higher-order language for reasoning about objects, inheritance and schema. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 134–146, 1989.

[KLW90] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical Report 90/14, Department of Computer Science, SUNY at Stony Brook, July 1990.

[KP88] P.G. Kolaitis and C.H. Papadimitriou. Why not negation by fixpoint. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 231–239, 1988.

[Llo87] J.W. Lloyd. *Foundations of Logic Programming (Second Edition)*. Springer Verlag, 1987.

[Man74] Z. Manna. *Mathematical Theory of Computation*. McGraw-Hill Book Co., 1974.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

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

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

[MR85] P. Mishra and U.S. Reddy. Declaration-free type checking. In *ACM Symposium on Principles of Programming Languages*, pages 7–21, 1985.

[PR89] C. Pyo and U.S. Reddy. Inference of polymorphic types for logic programs. In *Proceedings of the North American Conference on Logic Programming*, pages 1115–1132, 1989.

[Prz88] T.C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 193–216. Morgan Kaufmann, Los Altos, CA, 1988.

[Red88] U.S. Reddy. Notions of polymorphism for predicate logic programs. In *Intl. Conference on Logic Programming*, pages (addendum, distributed at conference) 17–34. MIT Press, August 1988. also to appear in J. Logic Programming.

[Red91] U. Reddy. A perspective on types for logic programs. In F. Pfenning, editor, *Types in Logic Programming*. MIT Press, 1991. to appear.

[Smo88] G. Smolka. Logic programming with polymorphically order-sorted types. Technical Report ILOG Report 55, IBM Deutchland GmbH, 1988.

[Ull88] J.F. Ullman. *Principles of Database and Knowledge-Base Systems, Volume 1*. Computer Science Press, 1988.

[Van89] A. Van Gelder. The alternating fixpoint of logic programs with negation. In *Proceedings of the ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–10, 1989.

[War77] D.H.D. Warren. Implementing Prolog – Compiling predicate logic programs. Technical Report 39 and 40, Dept. of Artificial Intelligence, University of Edinburgh, 1977.

[Xu89] J. Xu. *A Theory of Types and Type Inference in Logic Programming Languages*. PhD thesis, SUNY at Stony Brook, 1989.

[XW88] J. Xu and D.S. Warren. A type inference system for Prolog. In *Intl. Conference on Logic Programming*, pages 604–619, 1988.

[XW90] J. Xu and D.S. Warren. Semantics of types in logic programming. manuscript, 1990.

[YFS90] E. Yardeni, T. Fruehwirth, and E. Shapiro. Polymorphically typed logic programs. manuscript, 1990.

[YS87] E. Yardeni and E. Shapiro. A type system for logic programs. In E. Shapiro, editor, *Concurrent Prolog*, volume 2. MIT Press, 1987.

[Zob87] J. Zobel. Derivation of polymorphic types for Prolog programs. In *Proceedings of the Fourth International Logic Programming Conference*, Australia, 1987.