Appl. Math. Lett. Vol. 4, No. 3, pp. 25-28, 1991
Printed in Great Britain.

0893-9659/91 $3.00 + 0.00

Incomplete Types for Logic Databases

VERONICA DAHL*

Simon Fraser University

(Received July 1990)

Abstract. We propose to extend Horn-clause terms by invisibly associating them with *incomplete types*, that is, convenient representations of taxonomic information. We formally define typed logic databases and show their uses to reduce the length of the proofs needed to deduce some kinds of facts, to provide intensional replies, to perform quick semantic agreement verifications on natural language queries, and to achieve partial execution of some queries.

1. MOTIVATION

1.1. Property Inheritance

Consider the query: *Is Crocky happy?*, given a database where *Crocky* is a reptile, reptiles are animals, and animals are happy. In Prolog, a positive answer to the query:

```
?- happy(crocky).
```

can be obtained in three resolution steps from the database:

```
happy(A):- animal(A).
animal(A):- reptile(A).
reptile(crocky).
```

Instead, we propose to use typed Horn-clause logic and then compile the taxonomic part of this information into convenient terms, so that we can obtain the same property inheritance in just one resolution step. We write:

```
happy(A ∈ animal).
reptile ⊂ animal.
crocky ∈ reptile.
?- happy(crocky).
```

and a compiler transforms these clauses into:

```
happy(A ∈ [animal ⊃ Y]).                                    (1)
?- happy(crocky ∈ [animal ⊃ reptile ⊃ crocky]).             (2)
```

Types have been replaced by a representation of their relevant set inclusion relationships. It is *incomplete* in that it contains a tail variable which allows for further instantiation. Thus, `[animal ⊃ Y]` stands for "at least of animal type." Since the representation for constants should not allow for further instantiation, the constant itself closes the representation, as in the above query (we abusively keep the inclusion sign for uniformity of notation). Resolving (1) and (2) unifies `Y` with `reptile ⊃ crocky`, thus making the type further known as *both* reptile and animal.

For strictly hierarchical taxonomies, we have thus reduced type checking involving a chain of n set inclusion relationships to *one* rather than n resolution steps.

*We are grateful to Warren Burton, Carl Vogel and Tao Zhang, for helpful comments on this article's first draft, and to Andrew Kurn for his help with typesetting. This work was done under NSERC grant 06-4244.

1.2. Search Space Pruning and Intensional Replies

Let us now add the fact that all reptiles crawl: `crawl(A ∈ reptile)`. This can compile into:

```
crawl(A ∈ [animal ⊃ reptile ⊃ Z]).
```

Now consider the query: *Which animals crawl?*, that is:

```
?- animal(A), crawl(A).                                    (3)
```

A typical Prolog database would test, for each animal, whether it crawls, and provide alternative answers upon backtracking. Our compilation of the above query, however, can easily provide an intensional reply and only proceed to find specific answer instances if further prompted by the user. We can compile query (3) into:

```
?- crawl(A ∈ [animal ⊃ Y]), answer(A ∈ [animal ⊃ Y]).
```

whose execution unifies `Y` with `reptile ⊃ Z`. Now, `answer` can be defined as a database primitive, so that it outputs the intensional answer, "All reptiles. Would you like a list of them?" Specific answers, if requested, will examine only the *reptile* domain.

In the interest of generality, our formal definitions of the next section extend to lattice taxonomies, although these require a little more than the simple unification we can get away with for trees[1].

2. TYPED LOGIC DATABASES

We define as follows:

— Let K be a finite set of symbols called *proper names.*

— Let T be a finite set of symbols called *types.*

— Let R be a finite set of symbols called *relational symbols.*

— Let X be a set of variables.

— To each symbol $k \in K$ corresponds a symbol $t = \text{type}(k)$, with $t \in T$.

— To each variable $x \in X$ corresponds a symbol $t = \text{type}(x)$, with $t \in T$.

— To each symbol $r \in R$ we associate:

  - a positive integer $n = \text{degree}(r)$.
  - a list $[t_1, \ldots, t_n] = \text{domain}(r)$, where $t_i \in T$.

— Let $E(t)$ represent the set of proper names whose type is t.

— Let $L = \{E(t) \mid t \in T\}$.

— Let $U = \bigcup_{t \in T} E(t)$.

Then a lattice is defined by L, the partial ordering relation of set inclusion ($\subset$), and the binary operators of set union ($\cup$) and intersection ($\cap$). It is bounded by the top U and the bottom $\{\ \}$.

Definition *A typed database g is an application which associates, to each relational symbol $r \in R$ of degree n and domain $[t_1, \ldots, t_n]$, an n-ary relation $\rho = g(r)$, which maps $E(t_1) \times \cdots \times E(t_n) \rightarrow \{\text{true}, \text{false}\}$.*

[1] Where the lattice of types specializes to a tree or a set of trees, incomplete types are unique for every type named by the user and can be carried explicitly but invisibly. Where the lattice is not a strict hierarchy, type checking involves not only unification but also two calculations of incomplete types. This is still more efficient than the n resolution steps required for a chain of n set inclusion relationships in untyped logic databases.

In logic programming terms, a typed database g is a logic program in which variables and constants are typed (e.g., `A` $\in$ `animal`), and in which the set inclusion relationships between these types have been declared through clauses of the form[2]

```
ti ⊂ tj.
k ∈ ti.
```

where `ti, tj` $\in T$ and `k` $\in K$.

Definition *An incomplete type for t, noted $h(t)$, is a term of the form*[3]*: $[t_1, \ldots, t_{n-1}, t \mid V]$, where the $t_i \in T$, V is a variable ranging over incomplete types, $t \subset t_{n-1} \subset \cdots \subset t_1$ and there exists no $t_0 \in T$, such that $t_1 \subset t_0$. $\mid$ is a binary operator in infix notation that separates the head from the rest of a list.*

Property *Let $s, t \in T$. Then $s \subset t \Leftrightarrow \exists\, h(t), h(s)$ and a substitution Θ, such that $h(s) = h(t)\,\Theta$.*

Proof Let $s, t \in T$. Let $s \subset t$. We need to prove that $\exists\, h(t), h(s)$ and a substitution Θ, such that $h(s) = h(t)\,\Theta$. Since $s \subset t$,

$$\exists\, h(s) = [t_1, \ldots, t_n, t, t_{n+1}, \ldots, s \mid V_s].$$

Therefore, $\exists\, h(t) = [t_1, \ldots, t_n, t \mid V_t]$. Then if we take

$$\Theta = \{V_t \leftarrow [t_{n+1}, \ldots, s \mid V_s]\},$$

then $h(s) = h(t)\,\Theta$.

Now, let $\exists\, h(t), h(s)$ and a substitution Θ, such that $h(s) = h(t)\,\Theta$. The forms of the incomplete types for s and t are, by definition:

$$h(s) = [s_1, \ldots, s_n, s \mid V_s],$$

$$h(t) = [t_1, \ldots, t_m, t \mid V_t].$$

Because there exists a substitution Θ such that $h(s) = h(t)\,\Theta$, it must be the case that the first $m + 1$ elements in both incomplete types, being ground, are the same, and that V_t unifies to the remaining elements in $h(s)$. Therefore,

$$h(s) = [t_1, \ldots, t_n, t, t_{n+1}, \ldots, s \mid V_s],$$

which by definition means that $s \subset t$.

Remark *As a practical consequence of this property, a type s can be proven to be a subtype of t simply by unifying $h(s)$ and $h(t)$, and checking that t's tail variable has become instantiated.*

3. NATURAL LANGUAGE AGREEMENT AND PARTIAL EXECUTION

Suppose lexical definitions in a database's natural language front end contain incomplete types associated to the arguments of the relationships they induce, as in the following rewrite rules[4]

```
verb(barks(A: [animal, mammal, dog | V])) → [barks].
name(fido:  [animal, mammal, dog | fido]) → [fido].
name(crocky:  [animal, reptile | crocky]) → [crocky].
```

[2] In practice, the symbols $\subset$, $\in$ are replaced by available keyboard symbols, such as < and :. These are declared as binary operators in infix notation.

[3] We now use Prolog list notation rather than the terms with explicit inclusion symbols used in the introduction.

[4] Rewriting rules such as these are a syntactic variant of Prolog, convenient to reduce parsing to deduction while thinking in the traditional terms of formal grammars; terminals are noted as lists (e.g., `[barks]`). Non-traditional features are also present; for instance, both terminals and non-terminals may include arguments (e.g., the non-terminal `verb` has a unary function as its argument). The : stands for $\in$.

The first rule recognizes the word *barks* in an input sentence as a verb, and associates it with a representation `barks(A)`, whose argument is explicitly typed by h(`dog`). The two others take a proper name as its own internal representation, and associate it with an appropriate incomplete type. Checking for semantic type agreement (e.g., rejecting a sentence such as `Crocky barks.`) now reduces to unifying the incomplete types of the arguments, e.g., through a rule:

```
sentence:- name(Sa), verb(barks(Sa)).
```

A query such as: *Which animals bark?* would unify the incomplete type induced by the noun (namely, `[animal | V1]`) with that induced by the verb (namely, `[animal, mammal, dog | V]`). This not only quickly verifies semantic compatibility, but also makes the type of the answer more precise, by attaching it to the `dog` category. Thus, the query's translation by the language analyser contains an intensional reply that can be extracted with no further database consultation ("All dogs."). In this sense, incomplete types can be regarded as a means for partial execution of some queries.

4. RELATED WORK

Most other approaches to dealing with inheritance and taxonomic information in automated deduction are not concerned with improving the efficiency of the deduction process. Others improve it at the cost of rewriting the Prolog interpreter or compiler to extend the unification process [1]. Our approach, while being more modest, is directly implementable through a simple compiler written in Prolog itself.

The idea of reducing semantic agreement to unification with respect to type hierarchies is outlined in [2], and has inspired interesting work in computational linguistics [4]. Its application to inheritance, as well, is embryonically contained in [2], and also relates to [5].

Intensional replies in a logical setting have been studied, for instance, in [3], where rule transformations are used to include rules in the answers of queries.

REFERENCES

1. H. Ait-Kaci and R. Nasr, Login: A logic programming language with built-in inheritance, *Logic Programming Journal* **3**(3), 1986.
2. V. Dahl, On database systems development through logic, *ACM Transactions on Database Systems* **7**(1), 1982.
3. T. Imielinski, Intelligent query answering in rule based systems, *Logic Programming Journal* **4**(3), 1987.
4. C. Mellish, Implementing systemic classification by unification, *Computational Linguistics* **14**(1), 1988.
5. A. Porto, A framework for deducing useful answers to queries, *Universidade Nova de Lisboa* DI/UNL-16/88, 1988.

Simon Fraser University, Burnaby, B.C., Canada V5A 1S6