Type Inference for Datalog with Complex Type Hierarchies

Max Schäfer Oege de Moor
Semmle Ltd., Oxford, United Kingdom
{max,oege}@semmle.com

Abstract

Type inference for Datalog can be understood as the problem of mapping programs to a sublanguage for which containment is decidable. To wit, given a program in Datalog, a schema describing the types of extensional relations, and a user-supplied set of facts about the basic types (stating conditions such as disjointness, implication or equivalence), we aim to infer an over-approximation of the semantics of the program, which should be expressible in a suitable sublanguage of Datalog.

We argue that Datalog with monadic extensionals is an appropriate choice for that sublanguage of types, and we present an inference algorithm. The inference algorithm is proved sound, and we also show that it infers the tightest possible over-approximation for a large class of Datalog programs. Furthermore, we present a practical containment check for a large subset of our type language. The crux of that containment check is a novel generalisation of Quine's procedure for computing prime implicants. The type system has been implemented in a state-of-the-art industrial database system, and we report on experiments with this implementation.

Categories and Subject Descriptors H.2.3 [*Database Management*]: Languages

General Terms Algorithms, Languages, Theory

Keywords Type Inference, Datalog, Type System

1. Introduction

We aim to infer precise types for queries where the entities in the database satisfy complex conditions regarding subtyping, disjointness and so on.

To illustrate, consider the Datalog program in Figure 1, which is a slight adaptation of an example in [1]. It assumes a database that contains monadic extensional relations for *employee*, *senior*, *junior*, *parttime*, *student* and *manager*. These monadic relations capture the entities that are manipulated by queries, and they are the building blocks of types. There is furthermore an extensional relation $salary(x,z)$.

In the program, additional intensional relations are defined: $bonus(x,y)$ computes the bonus y that an employee x will get. Non-students x get a bonus y that is computed by multiplying x's salary z by a factor f, which is returned by $factor(x,f)$. The factor depends on the seniority of the employee. For students, the bonus is always 50, independent of their salary. Finally, the program defines a query, which is to find the bonus of all senior managers.

$$\begin{array}{lcl} bonus(x,y) & \leftarrow & \exists z,f.(\ salary(x,z) \wedge factor(x,f) \wedge \\ & & \qquad y = f \times z \wedge \neg student(x)\ ) \\ & & \vee\,(y = 50 \wedge employee(x) \wedge student(x)) \\ \\ factor(x,y) & \leftarrow & (junior(x) \wedge y = 0.1) \vee \\ & & (senior(x) \wedge y = 0.15) \\ \\ query(x,y) & \leftarrow & bonus(x,y) \wedge senior(x) \wedge manager(x) \end{array}$$

Figure 1. Example Datalog program

$$\begin{array}{l} employee(x) \leftrightarrow junior(x) \vee senior(x) \\ \neg(junior(x) \wedge senior(x)) \\ parttime(x) \rightarrow employee(x) \\ student(x) \wedge employee(x) \rightarrow parttime(x) \\ manager(x) \rightarrow employee(x) \\ manager(x) \rightarrow \neg parttime(x) \end{array}$$

Figure 2. Example type hierarchy

The entity types are expected to satisfy many interesting facts that are useful in reasoning about types, for instance those shown in Figure 2. We call such a set of facts the *type hierarchy*. Note that the type hierarchy goes well beyond the usual notion of subtyping in traditional programming languages, in that one can state other relationships besides mere implication or equivalence. An example of such an interesting fact is the last statement, which says that managers cannot be part-time employees. The programmer will have to state these facts as part of the database definition, so that they can be checked when information is entered, and to enable their use in type inference and optimisation.

To do type inference, we of course also need to know some facts about extensional relations. For instance,

$$salary(x,z) \rightarrow employee(x) \wedge float(z)$$

states that the first column of the extensional relation *salary* will contain values of type *employee*, and the second one will contain floating point numbers. A *schema* provides this column-wise typing information for every extensional relation.

Given the schema and the hierarchy, type inference should infer the following implications from the definitions in the program:

$$\begin{array}{l} bonus(x,y) \rightarrow employee(x) \wedge float(y) \\ factor(x,y) \rightarrow employee(x) \wedge float(y) \\ query(x,y) \rightarrow senior(x) \wedge manager(x) \wedge float(y) \end{array}$$

POPL'10, January 17–23, 2010, Madrid, Spain.

Note how these depend on facts stated in the type hierarchy: for example, in inferring the type of *factor*, we need to know that *employee* is the union of *junior* and *senior*.

Type inference will catch errors: when a term is inferred to have the type $\bot$, we know that it denotes the empty relation, independent of the contents of the database. Also, the programmer may wish to declare types for intensional relations in her program, and then it needs to be checked that the inferred type implies the declared type.

The benefits of having such type inference for queries is not confined to merely checking for errors. Precise types are also useful in query optimisation. In particular, the above query can be optimised to:

$$\begin{aligned} \mathit{query}(x, y) \leftarrow \quad & \mathit{manager}(x) \wedge \mathit{senior}(x) \wedge \\ & \mathit{salary}(x, z) \wedge y = 0.15 \times z \end{aligned}$$

This rewriting relies on the fact that we are asking for a manager, and therefore the disjunct in the definition of *bonus* that talks about students does not apply: a *student* is a *parttime* employee, and managers cannot be *parttime*. Again notice how the complex type hierarchy is crucial in making that deduction. The elimination of disjuncts based on the types in the calling context is named *type specialisation*; type specialisation is very similar to virtual method resolution in optimising compilers for object-oriented languages. Similarly, the junior alternative in the definition of *factor* can be eliminated. Once type specialisation has been applied, we can eliminate a couple of superfluous type tests: for instance, $\neg \mathit{student}(x)$ is implied by $\mathit{manager}(x)$. That is called *type erasure*. The key to both type specialisation and type erasure is an efficient and complete test for type containment, which takes all the complex facts about types into account.

Overview The general problem we address is as follows. Given the class of all Datalog programs $\mathcal{P}$, we identify a sublanguage $\mathcal{T}$ of *type programs*. Type inference assigns an upper bound $\lceil p \rceil$ in the language of type programs to each program p. To say that $\lceil p \rceil$ is an upper bound is just to say that p (plus the schema and the type hierarchy) implies $\lceil p \rceil$. The idea of such 'upper envelopes' is due to Chaudhuri [8, 9].

In Section 2 we propose a definition of the class of type programs: any program where all extensionals called are monadic. Type programs can, on the other hand, contain (possibly recursive) definitions of intensional predicates of arbitrary arity. We then define a type inference procedure $\lceil p \rceil$, and prove that its definition is sound in the sense that $\lceil p \rceil$ is truly an upper bound of p. Furthermore, it is shown that for negation-free programs p, the inferred type $\lceil p \rceil$ is also optimal: $\lceil p \rceil$ is the smallest type program that is an upper bound of p.

For negation-free programs, the definition of $\lceil p \rceil$ is in terms of the semantics of p, so that the syntactic presentation of a query does not affect the inferred type. It follows that the application of logical equivalences by the programmer or by the query optimiser does not affect the result of type inference.

The restriction that programs be negation-free can be relaxed to allow negation to occur in front of sub-programs that already have the shape of a type program. Not much more can be hoped for, since sound and optimal type inference for programs with arbitrary negations is not decidable.

To also do type checking and apply type optimisations, we need an effective way of representing and manipulating type programs. To this end, we identify a normal form for a large and natural class of type programs in Section 3. The class consists of those type programs where the only negations are on monadic extensionals (*i.e.* entity types). There is a simple syntactic containment check on this representation, inspired by our earlier work reported in [13].

That simple containment check is sound but not complete. A geometric analysis of the incompleteness in Section 4 suggests a generalisation of the celebrated procedure of Quine [22] for computing prime implicants. We present that generalised procedure, and show that the combination of the simple containment check plus the computation of prime implicants yields a complete algorithm for testing containment of type programs.

The algorithm we present is very different from the well-known approach of propositionalisation, which could also be used to decide containment. The merit of our novel algorithm is that it allows an implementation that is efficient in common cases. We discuss that implementation in Section 5. We furthermore present experiments with an industrial database system that confirm our claims of efficiency on many useful queries.

Finally, we discuss related work in Section 6, and conclude in Section 7.

In summary, the original contributions of this paper include:

- The identification of a **language of type programs** for Datalog, suitable both for precise error checking and for type-based optimisations.
- The definition of a very **simple type inference procedure** that is proved **sound and optimal**.
- A **containment check** for type programs that relies on a novel generalisation of Quine's method of computing prime implicants.
- Experiments in an industrial database system that show that this containment algorithm is **efficient and useful**.

2. Type Programs

It is customary to write Datalog programs as a sequence of rules as seen in Fig. 1, but this syntax is too imprecise for our purposes. In the remainder of the paper, all Datalog programs will be written as formulae of Stratified Least Fixpoint Logic (SLFP) over signatures without function symbols.

This logic, presented for example in [10], extends first order logic (including equality) with fixpoint definitions of the form $[r(x_1, \ldots, x_n) \equiv \varphi]\psi$. Here, r is an n-ary relation symbol, the x_i are pairwise different element variables, and φ and ψ are formulae. Intuitively, the formula defines r by the formula φ, which may contain recursive occurrences of r, and the defined relation can then be used in ψ.

The x_i are considered bound in φ and free in ψ, whereas the relation r is free in φ and bound in ψ. To ensure stratification, free relation symbols are not allowed to occur under a negation, which avoids the problematic case of recursion through negation.

For example, the formula

$$[s(x, y) \equiv x \doteq y \vee \exists z. e(x, z) \wedge \mathbf{s}(z, y)] s(\mathbf{x}, \mathbf{y}) \qquad (1)$$

defines the relation s to be the reflexive transitive closure of e. By way of illustration (and only in this example), free occurrences of element variables and relation variables have been marked up in bold face. The formula is trivially stratified since no negations occur anywhere.

In this program, the relation symbol s is an *intensional* predicate since it is given a defining equation. On the other hand, relation e is an *extensional*, which we assume to be defined as part of a database on which the program is run. We will denote the set of intensional relation symbols as $\mathbb{I}$ and that of extensional relation symbols as $\mathbb{E}$, and require them to be disjoint.

Model-theoretically, the role of the database is played by an interpretation $\mathcal{I}$ which assigns, to every n-ary relation symbol $e \in \mathbb{E}$, a set of n-tuples of some non-empty domain. To handle free variables we further need an assignment σ that maps free element variables to domain elements and free relation variables to sets of tuples.

Just as in first order logic, we can then define a modelling judgement $\mathcal{I} \models_\sigma \varphi$ stating that formula φ is satisfied under interpretation $\mathcal{I}$ and assignment σ. In particular, stratification ensures that formulae can be interpreted as continuous maps of their free relation variables, so that fixpoint definitions can be semantically interpreted by the least fixpoints of their defining equations.

For a formula φ with a single free element variable x, we set $[\![\varphi]\!]_\mathcal{I} := \{c \mid \mathcal{I} \models_{x:=c} \varphi\}$, omitting the subscript $\mathcal{I}$ where unambiguous.

Just as the semantics of the intensional predicates is determined with respect to the semantics of the extensional predicates, the types of the intensional predicates are derived with respect to the types of the extensional predicates. These are provided by a *schema* $\mathscr{S}$ which assigns, to every n-ary extensional predicate, an n-tuple of types. As is customary, types are themselves monadic predicates from a set $\mathbb{T} \subseteq \mathbb{E}$ of *type symbols*. For consistency, we require that the schema assigns a type symbol to itself as its type.

For the program in (1), for example, we might have two type symbols a and b, with the schema specifying that $\mathscr{S}(e) = (a, b)$ and of course $\mathscr{S}(a) = (a)$, $\mathscr{S}(b) = (b)$.

Semantically, we understand this to mean that in every interpretation of the extensional symbols conforming to the schema, the extension of a given column of an extensional symbol should be exactly the same as the extension of the type symbol assigned to it by the schema.

We can define a first order formula, likewise denoted as $\mathscr{S}$, which expresses this constraint: For every n-ary relation symbol $e \in \mathbb{E} \setminus \mathbb{T}$ and every $1 \leq i \leq n$ where $u \in \mathbb{T}$ is the type assigned to the i-th column of e, the formula $\mathscr{S}$ contains a conjunct

$$\forall x_i.(\exists x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n.e(x_1, \ldots, x_n)) \leftrightarrow u(x_i) \tag{2}$$

Our example schema above, for instance, would be expressed by the formula

$$(\forall x.(\exists y.e(x, y)) \leftrightarrow a(x)) \land (\forall y.(\exists x.e(x, y)) \leftrightarrow b(y))$$

In the literature, the types assigned by the schema are often not required to fit exactly, but merely to provide an over-approximation of the semantics. We could achieve this by relaxing (2) to use an implication instead of a bi-implication.

Instead, we take a more general approach by allowing arbitrary additional information about the types to be expressed through a *hierarchy* formula $\mathscr{H}$. Our only requirements about this formula are that it be a closed first order formula containing only type symbols (and no other relation symbols).

In particular, the hierarchy may stipulate subtyping relations; in our example, the hierarchy could be $\forall x.a(x) \rightarrow b(x)$, expressing that a is a subtype of b. Perhaps more interestingly, it can also express disjointness of types, *e.g.* by stating that $\forall x.\neg a(x) \lor \neg b(x)$; we could then deduce that in the definition of the reflexive transitive closure e can, in fact, be iterated at most once. Many other kinds of constraints are just as easily expressed, and provide opportunities for advanced optimisations.

We now approximate programs by *type programs*, which are programs that only make use of type symbols (and no other extensional relation symbols), and do not contain negation[1]. To every SLFP formula φ we assign a type program $\lceil\varphi\rceil$ by replacing negated subformulae with $\top$ and approximating every use of an extensional relation symbol by the conjunction of its column types. The complete definition is given in Fig. 3.

For our example program, we would obtain the type

$$[s(x, y) \equiv x \doteq y \lor \exists z.a(x) \land b(z) \land s(z, y)]s(x, y)$$

$$\begin{array}{lcl}
\lceil\bot\rceil & = & \bot \\
\lceil t_1 \doteq t_2\rceil & = & t_1 \doteq t_2 \\
\lceil e(t_1, \ldots, t_n)\rceil & = & u_1(t_1) \land \ldots \land u_n(t_n) \\
 & & \text{for } e \in \mathbb{E} \text{ with } \mathscr{S}(e) = (u_1, \ldots, u_n) \\
\lceil r(t_1, \ldots, t_n)\rceil & = & r(t_1, \ldots, t_n) \\
 & & \text{for } r \in \mathbb{I} \\
\lceil\neg\varphi\rceil & = & \top \\
\lceil\varphi \land \psi\rceil & = & \lceil\varphi\rceil \land \lceil\psi\rceil \\
\lceil\varphi \lor \psi\rceil & = & \lceil\varphi\rceil \lor \lceil\psi\rceil \\
\lceil\exists x.\varphi\rceil & = & \exists x.\lceil\varphi\rceil \\
\lceil[r(\vec{x}) \equiv \varphi]\psi\rceil & = & [r(\vec{x}) \equiv \lceil\varphi\rceil]\lceil\psi\rceil
\end{array}$$

Figure 3. Rules for type inference

which is semantically equivalent to

$$x \doteq y \lor a(x) \land b(y)$$

We will see in the next section that, in fact, fixpoint definitions can always be eliminated from such type programs, yielding formulae of monadic first order logic.

As the example suggests, the types assigned to programs semantically over-approximate them in the following sense:

Theorem 1 (Soundness)**.** *For every program φ we have*

$$\mathscr{S}, \mathscr{H}, \varphi \models \lceil\varphi\rceil$$

That is, every interpretation and assignment satisfying the schema, the hierarchy and φ also satisfies $\lceil\varphi\rceil$.

Proof Sketch. We can show by induction on φ the following stronger version of the theorem: For any interpretation $\mathcal{I}$ with $\mathcal{I} \models \mathscr{S}$ and any two assignments σ and σ', which assign the same values to element variables, and where $\sigma(r) \subseteq \sigma'(r)$ for every $r \in \mathbb{I}$, we have $\mathcal{I} \models_{\sigma'} \lceil\varphi\rceil$ whenever $\mathcal{I} \models_\sigma \varphi$.

From this, of course, it follows that $\mathscr{S}, \varphi \models \lceil\varphi\rceil$, and again $\mathscr{S}, \mathscr{H}, \varphi \models \lceil\varphi\rceil$ by monotonicity of entailment. □

Perhaps more surprisingly, our type assignment also yields the tightest such over-approximation, in spite of our very lax restrictions on the hierarchy:

Theorem 2 (Optimality)**.** *For a negation-free program φ and a type program ϑ, if we have*

$$\mathscr{S}, \mathscr{H}, \varphi \models \vartheta$$

then also

$$\mathscr{S}, \mathscr{H}, \lceil\varphi\rceil \models \vartheta$$

To prove this theorem, we need a monotonicity result for the type assignment.

Lemma 3. *For two negation-free programs φ and ψ, if it is the case that $\mathscr{S}, \mathscr{H}, \varphi \models \psi$ then also $\mathscr{S}, \mathscr{H}, \lceil\varphi\rceil \models \lceil\psi\rceil$.*

An easy way to prove this result is to make use of *cartesianisation*, which is the semantic equivalent of the typing operator $\lceil\cdot\rceil$. For a relation $R \subseteq D^n$, we define its cartesianisation $\texttt{cart}(R) = \pi_1(R) \times \ldots \times \pi_n(R)$ to be the cartesian product of the columns of R. Likewise, the cartesianisation $\texttt{cart}(\mathcal{I})$ of an interpretation $\mathcal{I}$ is the interpretation which assigns $\texttt{cart}(\mathcal{I}(e))$ to every relation symbol e.

It is then not hard to see that

$$\texttt{cart}(\mathcal{I}) \models_\sigma \varphi \iff \mathcal{I} \models_\sigma \lceil\varphi\rceil \tag{3}$$

for any negation-free formula φ, interpretation $\mathcal{I}$ and assignment σ whenever $\mathcal{I} \models \mathscr{S}$. Also, $\texttt{cart}(\mathcal{I}) \models \mathscr{S}$ if $\mathcal{I} \models \mathscr{S}$, and $\texttt{cart}(\mathcal{I}) \models \mathscr{H}$ iff $\mathcal{I} \models \mathscr{H}$.

[1] Compare this with the hierarchy which *can* contain negations, but no fixpoint definitions.

From this observation, we easily get the

Proof of Lemma 3. Assume $\mathscr{S}, \mathscr{H}, \varphi \models \psi$, and let $\mathcal{I}$, σ be given with $\mathcal{I} \models \mathscr{S}$, $\mathcal{I} \models \mathscr{H}$ and $\mathcal{I} \models_\sigma \lceil\varphi\rceil$.

By (3), we have $\mathtt{cart}(\mathcal{I}) \models \mathscr{S}$, $\mathtt{cart}(\mathcal{I}) \models \mathscr{H}$ and also $\mathtt{cart}(\mathcal{I}) \models_\sigma \varphi$, so by assumption $\mathtt{cart}(\mathcal{I}) \models_\sigma \psi$, but then by applying (3) again we get $\mathcal{I} \models_\sigma \lceil\psi\rceil$. □

We briefly pause to remark that Lemma 3 also shows that type inference for negation-free programs respects semantic equivalence: If φ and ψ are semantically equivalent (under $\mathscr{S}$ and $\mathscr{H}$), then so are their types. This does, of course, not hold in the presence of negation, for given a type symbol u, we have $\lceil u(x)\rceil = u(x)$ yet $\lceil\neg\neg u(x)\rceil = \top$.

Continuing with our development, we can now give the

Proof of Theorem 2. Assume $\mathscr{S}, \mathscr{H}, \varphi \models \vartheta$; by Lemma 3 this means $\mathscr{S}, \mathscr{H}, \lceil\varphi\rceil \models \lceil\vartheta\rceil$. But since ϑ is a type program we have $\lceil\vartheta\rceil = \vartheta$, which gives the result. □

So far we have handled negation rather crudely, trivially approximating it by $\top$. In fact, we can allow negations in type programs and amend the definition of $\lceil\cdot\rceil$ for negations to read

$$\lceil\neg\varphi\rceil = \begin{cases} \neg\varphi & \text{if } \varphi \text{ is a type program} \\ \top & \text{otherwise} \end{cases}$$

All the above results, including soundness and optimality, go through with this new definition, and the optimality and monotonicity results can be generalised to hold on all programs which contain only *harmless* negations, *i.e.* negations where the negated formula is a type program.

There is little hope for an optimality result on an even larger class of programs: Using negation, equivalence checking of two programs can be reduced to emptiness, and a program is empty iff its optimal typing is $\bot$. As equivalence is undecidable [23], any type system that is both sound and optimal for all programs would likewise have to be undecidable.

Our proposed definition of type programs is significantly more liberal than previous proposals in the literature. Frühwirth *et al.* in a seminal paper on types for Datalog [16] propose the use of conjunctive queries, with no negation, no disjunction, no equalities and no existentials.

As an example where the additional power is useful, consider a database that stores the marriage register of a somewhat traditionally minded community. This database might have types *male* and *female*, specified to be disjoint by the type hierarchy, and an extensional relation *married* with schema $(\mathit{female}, \mathit{male})^2$.

Using our above terminology, we have $\mathbb{T} = \{\mathit{male}, \mathit{female}\}$ and $\mathbb{E} = \mathbb{T} \cup \{\mathit{married}\}$; $\mathscr{H}$ is $\forall x.\neg \mathit{male}(x) \vee \neg \mathit{female}(x)$.

Let us now define an intensional predicate *spouse*:

$$[\mathit{spouse}(x, y) \equiv \mathit{married}(x, y) \vee \mathit{married}(y, x)]\mathit{spouse}(x, y)$$

What is the type of $\mathit{spouse}(x, y)$? In the proposal of [16], we could only assign it $(\mathit{female}(x) \vee \mathit{male}(x)) \wedge (\mathit{female}(y) \vee \mathit{male}(y))$, so both arguments could be either male or female. By contrast, when employing our definition of type programs, the inferred type is

$$(\mathit{female}(x) \wedge \mathit{male}(y)) \vee (\mathit{male}(x) \wedge \mathit{female}(y))$$

This accurately reflects that x and y must be of opposite sex. By properly accounting for equality, we can also infer that the query $\mathit{spouse}(x, y) \wedge x \doteq y$ has type $\bot$ under the hierarchy: nobody can be married to themselves.

3. Representing Type Programs

For query optimisation and error checking, the single most important operation on types is containment checking, *i.e.* we want to decide, for two type programs ϑ and ϑ', whether $\mathscr{H} \wedge \vartheta \models \vartheta'$, which we will write $\vartheta \models_{\mathscr{H}} \vartheta'$ for short.

As mentioned in the introduction, we are often interested in checking whether a formula is unsatisfiable, that is whether it always gives an empty set of results regardless of the interpretation of the extensional relations. If such a formula occurs in a user program, it is regarded as a type error and the user is alerted to this fact; if it occurs during optimisation, it provides an opportunity for logical simplification. By the results of the previous section, a formula φ containing only harmless negations is unsatisfiable iff its type is equivalent to $\bot$, that is iff $\lceil\varphi\rceil \models_{\mathscr{H}} \bot$.

Our type programs only make use of monadic extensionals, and their containment can be decided by a straightforward extension of the well-known decision procedure for monadic first order logic [6]. This construction, however, incurs an exponential blowup in formula size and does not directly yield a practical algorithm.

Under some mild restrictions on the class of type programs and the hierarchy, however, we can represent type programs in a compact manner that allows efficient containment checking in practice.

The class of type programs we deal with falls in between the two classes discussed in the previous section by allowing negation, but only in front of type symbols (and never before equalities). The definition of $\lceil\cdot\rceil$ is easily changed to accommodate this by setting $\lceil\neg\varphi\rceil = \neg\varphi$ if φ is of the form $u(t)$ for $u \in \mathbb{T}$, and $\top$ otherwise. We call a possibly negated application of a type symbol to a variable a *type literal*.

Furthermore, we require the hierarchy $\mathscr{H}$ to be of the form $\forall x.h(x)$, where $h(x)$ does not contain quantifiers or equations, and only a single free variable x.

The basic unit of our representation are *type propositions*, which are Boolean combinations of type symbols, all of them applied to the same variable. In other words, a type proposition is a quantifier-free type program without fixpoint definitions or free relation variables, having precisely one free element variable. For example, the subformula $h(x)$ of $\mathscr{H}$ is a type proposition.

A type proposition φ behaves essentially the same as the propositional formula $|\varphi|$ that results from it by replacing every atom $u(x)$ with the propositional letter u; for example, it is easily seen that $\forall x.\varphi$ is (first order) valid iff $|\varphi|$ is (propositionally) valid [20]. Hence we will sometimes identify the two notationally.

For example, if a and b are type symbols, then $a(x) \vee \neg b(x)$ is a type proposition with $|a(x) \vee \neg b(x)| = a \vee \neg b$.

For the restricted class of type programs under consideration, fixpoint definitions can be eliminated. To see this, recall that fixpoint definitions can be expanded into infinite disjunctions [10]. It thus suffices to show that the type programs without free relation variables and without fixpoint definitions (which we shall henceforth call *flat*) form a complete lattice, so that even a *prima facie* infinite disjunction can in fact be represented by a single formula.

Lemma 4. *The flat type programs form a complete lattice under the order* $\models_{\mathscr{H}}$.

Proof. The lattice structure is obviously given by the logical operations of conjunction and disjunction. To show completeness, we show that the set of flat type programs is finite (modulo equivalence under $\models_{\mathscr{H}}$).

Notice that every flat type program with m free element variables from $X = \{x_1, \ldots, x_m\}$ can be written in prenex form as

$$\exists y_1, \ldots, y_n. \bigvee_i [\bigwedge_{1\leq j\leq m} \psi_{i,j}(x_j) \wedge \bigwedge_{1\leq k\leq n} \chi_{i,k}(y_k) \wedge \zeta]$$

[2] Note that this means that all *male* and *female* entities in the database are married, *i.e.* occur somewhere in the *married* relation.

for a set $Y = \{y_1, \dots, y_n\}$ of element variables, where all the $\psi_{i,j}$ and $\chi_{i,k}$ are type propositions, and ζ is a conjunction of equations between element variables from $X \cup Y$.

Existentials distribute over disjunctions, and also over conjunctions if the variable they bind is only free in one of the conjuncts. Also, note that $\exists x.(x \doteq y \wedge \varphi)$ is semantically equivalent to φ with y substituted for x. Hence we can further bring the type program into the form

$$\bigvee_i [\bigwedge_{1 \leq j \leq m} \psi'_{i,j}(x_j) \wedge \bigwedge_{1 \leq k \leq n} (\exists y_k . \chi'_{i,k}(y_k)) \wedge \zeta']$$

where ζ' now contains only variables from X. We will say that a formula of this shape is in *solved form*.

Clearly, there are only finitely many different formulae in solved form (modulo semantic equivalence), since there are only finitely many type propositions. □

Notice that the set of flat type programs is no longer finite if we allow negated equalities. Indeed, for every natural number n, let ε_n be a formula asserting that there are at least n different individuals; for example, ε_2 is $\exists x_1, x_2 . \neg x_1 \doteq x_2$. Then $\neg \varepsilon_n \models_{\mathscr{H}} \neg \varepsilon_{n+1}$ for every n, yet all the ε_n are semantically distinct; so the lattice is not even of finite height anymore.

Corollary 5. *Every type program without free relation variables is semantically equivalent to a flat type program, and hence to a formula of monadic first order logic.*

Here is an example of how a type program (with only a single disjunct) can be brought into solved form.

$$\begin{array}{ll}
 & \exists z . a(x) \wedge z \doteq y \wedge b(y) \wedge c(x) \wedge x \doteq y \wedge \neg c(z) \\
\equiv & \{\text{sorting by variable}\} \\
 & \exists z . a(x) \wedge c(x) \wedge b(y) \wedge x \doteq y \wedge z \doteq y \wedge \neg c(z) \\
\equiv & \{\text{pushing in } \exists\} \\
 & a(x) \wedge c(x) \wedge b(y) \wedge x \doteq y \wedge \exists z . z \doteq y \wedge \neg c(z) \\
\equiv & \{\text{eliminating } \doteq \text{ under } \exists\} \\
 & a(x) \wedge c(x) \wedge b(y) \wedge x \doteq y \wedge \neg c(y) \\
\equiv & \{\text{sorting again}\} \\
 & a(x) \wedge c(x) \wedge b(y) \wedge \neg c(y) \wedge x \doteq y
\end{array}$$

We will now introduce data structures that represent formulae in solved form, and develop a containment checking algorithm that works directly on this representation.

We define an order on type propositions by taking $\varphi <: \psi$ to mean $\models |h| \wedge |\varphi| \rightarrow |\psi|$, where $|h|$ is the propositional formula corresponding to $\mathscr{H}$ as defined above. It is not hard to see that this holds precisely when $\varphi \models_{\mathscr{H}} \psi$. In the remainder, we will identify type propositions that are equivalent under $<:$ so that it becomes a partial order. Where needed, we make equivalence under $<:$ explicit by writing it as $\equiv_{\mathscr{H}}$.

If t is a single type proposition and C a set of type propositions, we write $t <: C$ ($C <: t$) to mean that $t <: c$ ($c <: t$) for some $c \in C$.

The order $<:$ can be extended to sets of type propositions by defining $C <: D$ to mean that for any $d \in D$ there is some $c \in C$ with $c <: d$. This order is clearly reflexive and transitive, and is anti-symmetric (and hence a partial order) on *lean* sets, *i.e.* sets C where $c <: c'$ for $c, c' \in C$ implies $c = c'$. One way of making a set of type propositions lean is to discard its non-minimal elements, which we write as $\min(C)$.

We can represent every disjunct of a formula in solved form using the following data structure:

Definition 1. *An n-ary* type tuple constraint, *or TTC, τ is a structure $(t_1, \dots, t_n \mid p \mid C)$ where*

- *each component t_i is a type proposition,*
- *p is an equivalence relation over $\{1, \dots, n\}$,*
- *C is a set of type propositions called* inhabitation constraints*,*

such that

- *for all $1 \leq i, j \leq n$ with $i \sim_p j$ (*i.e.*, i and j belong to the same partition of p) we have $t_i = t_j$,*
- *C is lean,*
- *for all $1 \leq k \leq n$ we have $C <: t_k$.*

If p is a relation, we write $\overline{p}$ to mean the smallest equivalence relation containing it. The equivalence class of an element x under p is written $[x]_p$, where we omit the subscript p if it is clear from context. We call an equivalence relation q *finer* (or, more precisely, *no coarser*) than p if whenever $x \sim_q y$ then also $x \sim_p y$.

Whenever we have a pre-TTC $\tau = (t_1, \dots, t_n \mid p \mid C)$ that does not satisfy one of the requirements, we can build a TTC that does by setting

$$\|\tau\| := (\|t_1\|_{\overline{p}}, \dots, \|t_n\|_{\overline{p}} \mid \overline{p} \mid \min(C \cup \{t_1, \dots, t_n\}))$$

where $\|t_i\|_q = \bigwedge \{t_j \mid i \sim_q j\}$.

For a TTC $\tau = (t_1, \dots, t_n \mid p \mid C)$, we define $\tau_i := t_i$, and $\tau_{i:=r}$ is the TTC resulting from τ by replacing the component t_i (and every component t_j where $i \sim_p j$) by r and adding r to the inhabitation constraints.

The formula represented by a TTC is easily recovered:

Definition 2. *Given a list of n different element variables $x_1, \dots, x_n$, we define the type program $[\tau](x_1, \dots, x_n)$ corresponding to a TTC $\tau = (t_1, \dots, t_n \mid p \mid C)$ as*

$$\left(\bigwedge_{1 \leq i \leq n} t_i(x_i) \right) \wedge \left(\bigwedge_{i \sim_p j} x_i \doteq x_j \right) \wedge \left(\bigwedge_{c \in C} \exists y . c(y) \right)$$

A TTC is called *degenerate* if $\perp$ is among its inhabitation constraints; it represents an unsatisfiable formula. The equivalence relation is called trivial if it is in fact the identity relation `id`, and the set of inhabitation constraints is called trivial if it only constrains the component types to be inhabited. When writing a TTC, we will often leave out trivial partitionings or inhabitation constraints.

We extend the order $<:$ to TTCs component-wise:

Definition 3. *For two n-ary TTCs $\tau = (t_1, \dots, t_n \mid p \mid C)$ and $\tau' = (t'_1, \dots, t'_n \mid p' \mid C')$ we define $\tau <: \tau'$ to hold iff*

- *for every $1 \leq i \leq n$ we have $t_i <: t'_i$,*
- *p' is finer than p,*
- *$C <: C'$.*

It is routine to check that this defines a partial order with maximal element $\top^{\text{TTC}} = (\top, \dots, \top)$, with binary meets defined by

$$(t_1, \dots, t_n \mid p \mid C) \,\overline{\wedge}\, (t'_1, \dots, t'_n \mid p' \mid C') = \|(t_1 \wedge t'_1, \dots, t_n \wedge t'_n \mid p \cup p' \mid C \cup C')\| \quad (4)$$

Since all type programs can be brought into solved form, we can represent them as sets of TTCs.

Definition 4. *A lean set of non-degenerate TTCs is called a* typing*. The formula represented by a typing is*

$$[T] := \bigvee_{\tau \in T} [\tau]$$

The order $<:$ is extended to typings by defining $T <: T'$ to hold if, for every $\tau \in T$ there is a $\tau' \in T'$ with $\tau <: \tau'$.

To convert any set of TTCs into a typing we can eliminate all non-maximal and degenerate TTCs from it; this operation we will denote by $\max_{\perp}$.

The order on typings is again a partial order with least element $\emptyset$ and greatest element $\{\top^{\mathtt{TTC}}\}$. Binary meets can be defined pointwise as

$$T \mathbin{\overline{\wedge}} T' = \max{}_{\perp}(\{\tau \mathbin{\overline{\wedge}} \tau' \mid \tau \in T, \tau' \in T'\}) \tag{5}$$

and joins are given by

$$T \mathbin{\underline{\vee}} T' = \max{}_{\perp}(T \cup T') \tag{6}$$

We will now show that every type program can be translated to a typing. We already have enough structure to model conjunction and disjunction; existential quantification is also not hard to support.

Definition 5. *The i-th* existential *of a TTC $\tau = (t_1, \ldots, t_n \mid p \mid C)$, where $1 \le i \le n$, is defined as*

$$\exists_i(\tau) := (t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_n \mid p' \mid \min(C \cup \{t_i\}))$$

where p' is the restriction of p to $\{1, \ldots, i-1, i+1, \ldots, n\}$.

The existential of a typing is defined pointwise:

$$\exists_i(T) := \max{}_{\perp}\{\exists_i(\tau) \mid \tau \in T\}$$

To see that this models the existential quantifier on type programs, we can easily convince ourselves of the following:

Lemma 6. *Let τ be an n-ary TTC, $\mathcal{I}$ an interpretation and σ a statement. Then $\mathcal{I} \models_\sigma [\exists_i(\tau)]$ iff there is a domain element d such that $\mathcal{I} \models_{\sigma, x_i := d} [\tau]$. An analogous statement holds for typings.*

Although typings cannot directly represent type programs with fixpoint definitions or free relation variables, we can translate every closed type program to a typing by eliminating definitions. Let us start with a translation relative to an assignment to free relation symbols.

Definition 6. *Let φ be a type program without fixpoint definitions containing the intensional relation variables $r_1, \ldots, r_m$ and the element variables $x_1, \ldots, x_n$. We translate it to a mapping $\langle\varphi\rangle\colon \mathrm{Typing}^m \to \mathrm{Typing}$ by recursion on the structure of φ:*

$$\begin{array}{lcl}
\langle\perp\rangle(\vec{T}) & = & \emptyset \\
\langle x_i \doteq x_j\rangle(\vec{T}) & = & \{(\top, \ldots, \top \mid \overline{\{\{i,j\}\}})\} \\
\langle u(x_j)\rangle(\vec{T}) & = & \{\top^{\mathtt{TTC}}{}_{j:=u}\} \\
\langle \neg u(x_j)\rangle(\vec{T}) & = & \{\top^{\mathtt{TTC}}{}_{j:=\neg u}\} \\
\langle r_i(\vec{x})\rangle(\vec{T}) & = & T_i \\
\langle \psi \wedge \chi\rangle(\vec{T}) & = & \langle\psi\rangle(\vec{T}) \mathbin{\overline{\wedge}} \langle\chi\rangle(\vec{T}) \\
\langle \psi \vee \chi\rangle(\vec{T}) & = & \langle\psi\rangle(\vec{T}) \mathbin{\underline{\vee}} \langle\chi\rangle(\vec{T}) \\
\langle \exists x_i.\psi\rangle(\vec{T}) & = & \exists_i(\langle\psi\rangle(\vec{T}))
\end{array}$$

It is easy to check that $\langle\varphi\rangle$ is a monotonic mapping for every φ with respect to the order $<:$, and since the set of typings is finite its least fixpoint can be computed by iteration. Thus we can translate every type program φ which does not have free relation variables into a typing $\langle\varphi\rangle$ which represents it:

Lemma 7. *For every type program φ without free relation variables, we have $\varphi \mathrel{=\!\mid}\models_{\mathscr{H}} [\langle\varphi\rangle]$.*

Proof Sketch. In fact, we can show that, for any type program φ with m free relation variables and every m-tuple $\vec{T}$ of typings, we have the equivalence $\varphi([\vec{T}]) \mathrel{=\!\mid}\models [\langle\varphi\rangle(\vec{T})]$, where $[\vec{T}]$ is the m-tuple of type programs we get from applying $[\cdot]$ to every component of $\vec{T}$.

This can be proved by structural induction on φ by showing that $[\langle\cdot\rangle]$ commutes with the logical operators (using Lemma 6 for the case of existentials) and with fixpoint iteration. The lemma then follows for the case $m = 0$. □

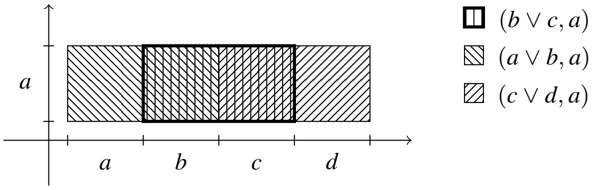

Figure 4. Example for Incompleteness of $<:$

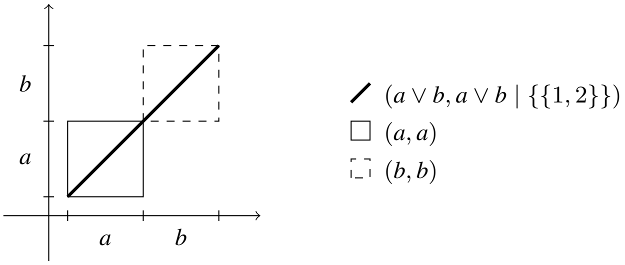

Figure 5. Further Example for Incompleteness of $<:$

4. Containment Checking

We are now going to investigate the relation between the orders $<:$ (*component-wise* or *syntactic* subtyping) and $\models_{\mathscr{H}}$ (*semantic* subtyping). The former is convenient in that it can be checked component-wise and only involves propositional formulae, so it would be nice if we could establish that $T_1 <: T_2$ iff $[T_1] \models_{\mathscr{H}} [T_2]$.

Unfortunately, this is not true. The ordering $<:$ is easily seen to be sound in the following sense:

Theorem 8. *For any two typings T and T', if $T <: T'$ then $[T] \models_{\mathscr{H}} [T']$.*

However, it is not complete. Consider, for example, the typings $T = \{(b \vee c, a)\}$ and $T' = \{(a \vee b, a), (c \vee d, a)\}$. Clearly we have $[T] \models_{\mathscr{H}} [T']$, yet $T \not<: T'$.

If we interpret type symbols as intervals of real numbers, this example has an intuitive geometric interpretation, shown in Fig. 4.

The individual TTCs then become rectangles in the two-dimensional plain, and it is easy to see that the single TTC in T (depicted by the single rectangle in heavy outline filled with vertical lines) is contained in the union of the TTCs in T' (the two rectangles filled with slanted lines), but not in either one of them in isolation.

A similar problem can arise from the presence of equality constraints: Consider the typings $T = \{(a \vee b, a \vee b \mid \{\{1, 2\}\})\}$ and $T' = \{(a, a), (b, b)\}$. The former appears in Fig. 5 as a heavy slanted line, which is not contained in either of the two squares representing the TTCs in T', but only in their union.

Intuitively, the problem is that $<:$ tests containment of TTCs in isolation, whereas the semantic inclusion check considers containments between sets of TTCs.

The following lemma paves the way for a solution:

Lemma 9. *The mapping $[\cdot]$ is the lower adjoint of a Galois connection,* i.e. *there is a mapping $(\!|\cdot|\!)$ from type programs to typings such that*

$$[T] \models_{\mathscr{H}} \vartheta \quad \textit{iff} \quad T <: (\!|\vartheta|\!)$$

Proof. As we have established, typings and type programs form complete lattices under their respective orders. Since $[\cdot]$ distributes over joins it must be the lower adjoint of a Galois connection between these lattices. □

An immediate consequence of this is that $[T_1] \models_{\mathscr{H}} [T_2]$ implies $T_1 <: \llparenthesis [T_2] \rrparenthesis$. To check, then, whether T_1 is semantically contained in T_2, we compute $\llparenthesis [T_2] \rrparenthesis$ and check for component-wise inclusion.

One possible implementation of $\llparenthesis \cdot \rrparenthesis$ comes directly from the definition of the Galois connection: For any type program ϑ, $\llparenthesis \vartheta \rrparenthesis$ is the greatest typing T such that $[T] \models_{\mathscr{H}} \vartheta$, and since there are only finitely many typings and containment is decidable we could perform an exhaustive search to find this T.

Clearly, this algorithm is not ideal, since one of our goals is to *avoid* having to explicitly decide the relation $\models_{\mathscr{H}}$. Inspiration for a better solution comes from a perhaps somewhat unexpected direction, namely the theory of Boolean function minimisation: Quine [22] introduces an algorithm whose first part determines the set of prime implicants of a Boolean formula, which are conjunctive formulae such that every conjunctive formula that implies the original formula implies one of the prime implicants.

Generalising from conjunctions to TTCs and from propositional formulae to typings, we will introduce a saturation mapping `sat` that generates the set of all prime implicant TTCs for a typing, which are TTCs such that every TTC that (semantically) entails the typing entails one of them component-wise. Hence whenever $[T_1] \models_{\mathscr{H}} [T_2]$ we have $T_1 <: \mathtt{sat}(T_2)$, so `sat` is an implementation of $\llparenthesis [\cdot] \rrparenthesis$. The saturation mapping also preserves semantics, so the other direction of the implication holds as well.

We can follow Quine's definitions and proofs almost paragraph-by-paragraph, starting with the generalisation of the consensus operation:

Definition 7. *The* consensus $\tau \oplus_J \tau'$ over J *of two n-ary TTCs* τ *and* τ'*, where* $J \subseteq \{1, \ldots, n\}$ *is an index set, is defined by*

$$(t_1, \ldots, t_n \mid p \mid C) \oplus_J (t'_1, \ldots, t'_n \mid p' \mid C') = \\ \|(u_1, \ldots, u_n \mid p \cup p' \cup J \times J \mid C \cup C')\|$$

where

$$u_i = \begin{cases} \bigwedge_{j \in J} t_j \vee \bigwedge_{j \in J} t'_j & \textit{if } i \in J \\ t_i \wedge t'_i & \textit{otherwise} \end{cases}$$

We will sometimes write $\bigwedge_J \tau$ for $\bigwedge_{j \in J} t_j$.

Roughly, the consensus operation equates all columns in J and preserves all other equalities and inhabitation constraints of its operands, takes disjunctions over the columns in J, and conjunctions over all the others.

For example, in the example shown in Fig. 5 we can take the consensus of (a, a) and (b, b) over $J = \{1, 2\}$ to obtain the TTC $(a \vee b, a \vee b \mid \{\{1, 2\}\})$, which is precisely the missing piece to show that T is a subtype of T'.

In the particular case where $J = \{j\}$ is a singleton set, and both partitions and inhabitation constraints are trivial, the consensus takes conjunctions on all columns except j, where it takes a disjunction. This is a generalisation of the behaviour of the consensus operator used in the minimisation of Boolean functions.

For example, in Fig. 4 we can take the consensus of $(a \vee b, a)$ and $(c \vee d, a)$ on $J = \{1\}$ to obtain the TTC $(a \vee b \vee c \vee d, a)$ which covers $(b \vee c, a)$.

We need another kind of consensus to deal with inhabitation constraints:

Definition 8. *The* existential consensus *of two n-ary TTCs* τ *and* τ' *is defined by*

$$(t_1, \ldots, t_n \mid p \mid C) \oplus_\exists (t'_1, \ldots, t'_n \mid p' \mid C') = \\ \|(t_1 \wedge t'_1, \ldots, t_n \wedge t'_n \mid p \cup p' \mid C \vee C')\|$$

where $C \vee C' = \{c \vee c' \mid c \in C, c' \in C'\}$.

Algorithm 1 SATURATE(T)

Require: A typing T of arity n.
Ensure: A saturated typing semantically equivalent to T.

```
repeat
    $T_{\mathtt{old}} \leftarrow T$
    for $\tau, \tau' \in T_{\mathtt{old}}$ do
        $T \leftarrow \max_\perp(T \cup \text{ALL-CONSENSUS}(n, \tau, \tau'))$
until $T_{\mathtt{old}} = T$
return $T$
```

Algorithm 2 ALL-CONSENSUS(n, τ, τ')

Require: Two TTCs τ and τ' of arity n.
Ensure: The lean set of all their consensus TTCs.

```
$S \leftarrow \emptyset$
for $J \subseteq \{1, \ldots, n\}$ do
    $S \leftarrow \max_\perp(S \cup \{\tau \oplus_J \tau'\})$
$S \leftarrow \max_\perp(S \cup \{\tau \oplus_\exists \tau'\})$
return $S$
```

To see why this is necessary, assume we have type symbols a, b, c with $\mathscr{H} \models \forall x.a(x) \vee b(x) \rightarrow c(x)$, and consider typings $T = \{(c \mid \mathtt{id} \mid a \vee b)\}$ and $T' = \{(c \mid \mathtt{id} \mid a), (c \mid \mathtt{id} \mid b)\}$. Clearly, $[T] = c(x_1) \wedge \exists z.(a(z) \vee b(z)) = [T']$, yet $T \not<: T'$ since $\{a \vee b\} \not<: \{a\}$ and $\{a \vee b\} \not<: \{b\}$. However, if we add the existential consensus $(c \mid \mathtt{id} \mid a \vee b)$ to T', we will be able to prove that $T <: T'$.

It is not hard to verify that adding the consensus of two TTCs to a typing does not change its semantics.

Lemma 10. *For two TTCs* τ, τ' *and an index set* J *we have* $[\tau \oplus_J \tau'] \models_{\mathscr{H}} [\{\tau, \tau'\}]$ *and* $[\tau \oplus_\exists \tau'] \models_{\mathscr{H}} [\{\tau, \tau'\}]$.

Proof Sketch. Clearly, if $J = \emptyset$ the consensus is just the meet, so nothing needs to be proved.

Otherwise, let $\mathcal{I}$ and σ be given such that $\mathcal{I} \models_\sigma [\tau \oplus_J \tau']$. Then, for some $i \in J$, either $\mathcal{I} \models_\sigma \bigwedge_{j \in J} t_j(x_i)$ or $\mathcal{I} \models_\sigma \bigwedge_{j \in J} t'_j(x_i)$; in the former case $\mathcal{I} \models_\sigma [\tau]$, and in the latter case $\mathcal{I} \models_\sigma [\tau']$.

For the existential consensus, assume $\mathcal{I} \models_\sigma [\tau \oplus_\exists \tau']$ and further assume there is some inhabitation constraint c of τ such that $\mathcal{I} \not\models \exists y.c(y)$ (for otherwise $\mathcal{I} \models_\sigma [\tau]$ is immediate). We must have $\mathcal{I} \models \exists y.c(y) \vee c'(y)$ for all inhabitation constraints c' of τ', so certainly $\mathcal{I} \models \exists y.c'(y)$. We then easily see that $\mathcal{I} \models_\sigma [\tau']$. □

Remember that typings do not contain non-maximal elements. The following result shows that we do not lose anything by eliminating them.

Lemma 11. *Consensus formation is monotonic in the sense that for TTCs* $\sigma, \sigma', \tau, \tau'$ *with* $\sigma <: \sigma'$ *and* $\tau <: \tau'$ *and for an index set* J *we have* $\sigma \oplus_J \tau <: \sigma' \oplus_J \tau'$ *and* $\sigma \oplus_\exists \tau <: \sigma' \oplus_\exists \tau'$

Proof Sketch. The components of the two TTCs are combined using conjunction and disjunction, which are monotonic with respect to subtyping, and the partitionings and inhabitation constraints are combined using set union which is also monotonic. □

Definition 9. *An n-ary typing* T *is said to be* saturated *if the consensus of any two TTCs contained in it is* covered *by the typing;* i.e. *for any* $\tau, \tau' \in T$ *and any index set* $J \subseteq \{1, \ldots, n\}$ *we have* $\tau \oplus_J \tau' <: T$ *and* $\tau \oplus_\exists \tau' <: T$.

Lemma 12. *Every typing T can be converted into a semantically equivalent, saturated typing $\mathtt{sat}(T)$ by exhaustive consensus formation.*

Proof. We use Algorithm 2 to collect all consensus TTCs of two given TTCs. As shown in Algorithm 1, this is performed for every pair of TTCs in the typing, and this procedure is repeated until a fixpoint is reached.

Notice that in each iteration (except the last) the set of TTCs covered by the typing becomes larger, and since there are only finitely many TTCs, the termination condition must become true eventually. □

We now generalise the concepts of implicants and prime implicants.

Definition 10. *A TTC τ is an* implicant *for a typing T' if we have $[\tau] \models_{\mathscr{H}} [T']$; it is a* prime implicant *if it is a $<:$-maximal implicant.*

More explicitly, a TTC π is a prime implicant for a typing T' if

1. π is an implicant of T'
2. For any τ with $\pi <: \tau$ and $[\tau] \models_{\mathscr{H}} [T']$, we have $\tau <: \pi$.

The second condition is equivalent to saying that for any τ with $\pi <: \tau$ and $\tau \not<: \pi$ we have $[\tau] \not\models_{\mathscr{H}} [T]$.

Lemma 13. *Every implicant implies a prime implicant.*

Proof. The set of all implicants of a typing is finite, so for any implicant τ there is a maximal implicant π with $\tau <: \pi$, which is then prime by definition. □

Remark 14. *If π is a prime implicant of T, then $\pi <: T$ iff $\pi \in T$.*

Proof. The direction from right to left is trivial. For the other direction, suppose $\pi <: T$, *i.e.* $\pi <: \pi'$ for some $\pi' \in T$. Certainly $[\pi'] \models_{\mathscr{H}} [T]$, so $\pi' <: \pi$ since π is prime. But this means that $\pi = \pi' \in T$. □

We want to show that $\mathtt{sat}(T)$ contains all prime implicants of T, so we show that saturation can continue as long as there is some prime implicant not included in the saturated set yet.

Lemma 15. *If there is a prime implicant π of T with $\pi \notin T$, then T is not saturated.*

Proof. Let π be a prime implicant of T with $\pi \notin T$. Consider the set

$$M := \{\tau \mid [\tau] \models_{\mathscr{H}} [\pi], \tau \not<: T\}$$

This set is not empty (it contains π by the preceding remark) and it is finite, so we can choose a $<:$-minimal element $\psi \in M$.

The proof proceeds by considering three cases. In the first, we shall show that a consensus step is possible, thus proving that T is not saturated. In the second, we show that an existential consensus step can be made, again proving non-saturation. Finally, we show that either the first or second case must apply, as their combined negation leads to a contradiction.

- Assume there is an index i for which $\psi_i \equiv_{\mathscr{H}} r' \vee r''$ for two r', r'' neither of which equals ψ_i.
  Then we form $\psi' := \psi_{i:=r'}$ and $\psi'' := \psi_{i:=r''}$. Observe that ψ' and ψ'' are strictly smaller than ψ, so they cannot be in M. They both, however, entail π, so there must be $\varphi', \varphi'' \in T$ with $\psi' <: \varphi'$ and $\psi'' <: \varphi''$.
  Note that $\psi = \psi' \oplus_{[i]} \psi''$, where $[i]$ is the equivalence class of i in ψ. By Lemma 11, this means that $\psi <: \varphi' \oplus_{[i]} \varphi''$, and since $\psi \not<: T$ we also have $\varphi' \oplus_{[i]} \varphi'' \not<: T$, which shows that T is not saturated.
- Assume for some $c \in C$, where C is the set of inhabitation constraints of ψ, we have $c \equiv_{\mathscr{H}} r' \vee r''$ for two r', r'' neither of which equal c.
  We form ψ' from ψ by setting the inhabitation constraints to $C \setminus \{c\} \cup \{r'\}$ and minimising, and similarly for ψ''. As above, those two are strictly smaller than ψ and we can find φ' and φ''. The proof goes through as in the previous case, since it is easy to see that $\psi = \psi' \oplus_{\exists} \psi''$.
- Otherwise, for any index i and r', r'' with $\psi_i \equiv_{\mathscr{H}} r' \vee r''$ we have either $\psi_i \equiv_{\mathscr{H}} r'$ or $\psi_i \equiv_{\mathscr{H}} r''$ (*i.e.* every ψ_i is join-irreducible) and likewise for all $c \in C$.
  We will show that in this case $\psi <: T$, contrary to our assumptions. We know that $[\psi] \models_{\mathscr{H}} [T]$, so any interpretation $\mathcal{I}$ and assignment σ with $\mathcal{I} \models \mathscr{H}$ and $\mathcal{I} \models_{\sigma} [\psi]$ will satisfy some TTC $\tau \in T$. Put differently, if $\mathcal{I} \models_{\sigma} (\mathscr{H} \wedge \psi_i)(x_i)$ for all i and $\mathcal{I} \models \exists y.(\mathscr{H} \wedge c)(y)$ for all inhabitation constraints c and moreover σ satisfies the partition of ψ, then we will find $\tau \in T$ with $\mathcal{I} \models_{\sigma} [\tau]$.
  Observe that for every i the (propositional) formula $|h| \wedge |\psi_i|$ can be written as a conjunction $l_{i,1} \wedge \ldots \wedge l_{i,m_i}$ of type literals. We can moreover assume that every type symbol occurs in precisely one of these literals, for if, say, type t does not occur, then $l_{i,1} \wedge \ldots \wedge l_{i,m_i} \wedge t \vee l_{i,1} \wedge \ldots \wedge l_{i,m_i} \wedge \neg t$ is a join reduction of $|h| \wedge |\psi_i|$.
  Let such a decomposition be fixed and let $L_i := \{l_{i,1}, \ldots, l_{i,m_i}\}$. Since $\psi_i = \psi_j$ whenever i and j are in the same partition, we can clearly choose the decomposition such that $L_i = L_j$ in this case. Likewise, for every inhabitation constraint c we can write $|h| \wedge |c|$ as $l_{c,1} \wedge \ldots \wedge l_{c,m_c}$, and define L_c by analogy.
  We define an interpretation $\mathcal{I}$ over the domain $\{x_{[1]}, \ldots, x_{[n]}\} \cup C$ of variables modulo the partition of ψ plus the inhabitation constraints by

  $$\mathcal{I}(b) = \{x_{[i]} \mid b \in L_i\} \cup \{c \mid b \in L_c\}$$

  for every type symbol b. The assignment is simply defined by $\sigma(x_i) = x_{[i]}$.
  It is easy to see that $x_{[i]} \in [\![l]\!]$ iff $l \in L_i$ for all $1 \leq i \leq n$, and $c \in [\![l]\!]$ iff $l \in L_c$ for all inhabitation constraints c, from which we deduce $\mathcal{I} \models_{\sigma} (\mathscr{H} \wedge \psi_i)(x_i)$ for all i, and $\mathcal{I} \models \exists y.(\mathscr{H} \wedge c)(y)$ for all inhabitation constraints c. By definition, σ satisfies the partition of ψ. So we have some $\tau \in T$ with $\mathcal{I} \models_{\sigma} [\tau]$.
  We claim that $\psi <: \tau$.
  Indeed, let an index $1 \leq i \leq n$ be given. Then we can write τ_i as a disjunction of conjunctions, such that one of its disjuncts, of the form $l'_{i,1} \wedge \ldots \wedge l'_{i,m'_i}$, is satisfied under $\mathcal{I}$ and σ, meaning that $x_{[i]} \in [\![l'_{i,j}]\!]_{\mathcal{I}}$ for all j, so all the $l'_{i,j}$ are in fact in L_i, so $\psi_i <: \tau_i$.
  Since σ satisfies the partition of τ, this partition must be finer than the partition of ψ.
  Finally, let an inhabitation constraint d of τ be given. Since $\mathcal{I} \models \exists y.(\mathscr{H} \wedge d)(y)$, the interpretation of $\mathscr{H} \wedge d$ cannot be empty. As above, we can write $\mathscr{H} \wedge d$ in a disjunctive form such that all the literals in one of its disjuncts are non-empty in $\mathcal{I}$. So there must be some domain element that occurs in the denotation of all these literals. If it is one of the $x_{[i]}$, then we have $\psi_i <: d$, which implies $c <: d$ for some inhabitation constraint c of ψ; if it is an inhabitation constraint c, then we have $c <: d$ directly.
  Taking these facts together, we get $\psi <: \tau$, whence $\psi <: T$. This contradicts our assumption, and we conclude that this subcase cannot occur.

□

Now we can declare victory:

Theorem 16. *If* $[T_1] \models_{\mathscr{H}} [T_2]$*, then* $T_1 <: \mathtt{sat}(T_2)$*.*

Proof. Assume $[T_1] \models_{\mathscr{H}} [T_2]$ and let $\tau \in T_1$ be given. Then $[\tau] \models_{\mathscr{H}} [T_2]$, so certainly $[\tau] \models_{\mathscr{H}} [\mathtt{sat}(T_2)]$, since saturation does not change the semantics. By Lemma 13 this means that there is a prime implicant π of $\mathtt{sat}(T_2)$ with $\tau <: \pi$. By Lemma 15 we must have $\pi \in \mathtt{sat}(T_2)$, so $\tau <: \mathtt{sat}(T_2)$. Since this holds for any $\tau \in T_1$ we get $T_1 <: \mathtt{sat}(T_2)$. □

5. Implementation

It is not immediately clear that the representation for type programs proposed in the preceding two sections can be implemented efficiently. While the mapping $\lceil\cdot\rceil$ from programs to type programs is, of course, easy to implement and linear in the size of the program, the translation of type programs into their corresponding typings involves fixpoint iterations to eliminate definitions. Saturation is also potentially very expensive, since we have to compute the consensus on every combination of columns for every pair of TTCs in a typing, and repeat this operation until a fixpoint is reached.

We report in this section on our experience with a prototype implementation based on Semmle's existing type checking technology as described in [13]. The type checker computes typings for programs and immediately proceeds to saturate them to prepare for containment checking. It uses a number of simple heuristics to determine whether a consensus operation needs to be performed. This drastically reduces the overhead of saturation in practice and makes our type inference algorithm practically usable.

TTCs can be compactly represented by encoding their component type propositions as binary decision diagrams. The same can be done for the hierarchy, so checking containment of type propositions can be done with simple BDD operations.

As with any use of BDDs, it is crucial to find a suitable variable order. We choose a very simple order that tries to assign neighbouring indices to type symbols that appear in the same conjunct of the hierarchy formula $\mathscr{H}$. For example, if the hierarchy contains a subtyping statement $u_1(x) \rightarrow u_2(x)$ or a disjointness constraint $\neg(u_1(x) \wedge u_2(x))$, then u_1 and u_2 will occupy adjacent BDD variables. This heuristic is simple to implement and yields reasonable BDD sizes as shown below.

To mitigate the combinatorial explosion that saturation might bring with it, we avoid *useless* consensus formation, *i.e.* we will not form a consensus $\tau \oplus \tau'$ if $\tau \oplus \tau' <: \{\tau, \tau'\}$ or $\tau \oplus \tau'$ is degenerate. The following lemma provides a number of sufficient conditions for a consensus to be useless:

Lemma 17. *Let* $\tau = (t_1, \ldots, t_n \mid p \mid C)$ *and* $\tau' = (t'_1, \ldots, t'_n \mid p' \mid C')$ *be two TTCs and* $J \subseteq \{1, \ldots, n\}$ *a set of indices. Then the following holds:*

1. *If* $N^\perp := \{i \mid t_i \wedge t'_i = \perp\} \not\subseteq J$ *then* $\tau \oplus_J \tau'$ *is degenerate.*
2. *If* $\bigwedge_{j\in J} t_j = \perp$ *or* $\bigwedge_{j\in J} t'_j = \perp$*, then* $\tau \oplus_{J'} \tau' <: \{\tau, \tau'\}$ *for any* $J' \supseteq J$*.*
3. *If we have* $l \notin J$ *with* $\bigwedge_{j\in J} t_j <: t_l$ *and* $\bigwedge_{j\in J} t'_j <: t'_l$*, then* $\tau \oplus_{J\cup\{l\}} \tau' <: \tau \oplus_J \tau'$*.*
4. *If* $\bigwedge_{j\in J} t_j <: \bigwedge_{j\in J} t'_j$ *or vice versa, then* $\tau \oplus_J \tau' <: \{\tau, \tau'\}$*.*
5. *For* $j \in J, j' \notin J$ *with* $j \sim_{\overline{p\cup p'}} j'$ *we have* $\tau \oplus_J \tau' <: \tau \oplus_{J\cup\{j'\}} \tau'$*.*
6. *If* $N^\perp \neq \emptyset$*, then* $\tau \oplus_\exists \tau'$ *is degenerate.*

Proof. Recall that $\tau \oplus_J \tau' = \|(u_1, \ldots, u_n \mid p \cup p' \cup J \times J \mid C \cup C')\|$ where

$$u_i = \begin{cases} \bigwedge_{j\in J} t_j \vee \bigwedge_{j\in J} t'_j & \text{if } i \in J \\ t_i \wedge t'_i & \text{otherwise} \end{cases}$$

We prove the individual claims:

1. Indeed, assume $k \in N^\perp \setminus J$, then $u_k = t_k \wedge t'_k = \perp$.
2. Assume, for example, $\bigwedge_{j\in J} t_j = \perp$ and let $J' \supseteq J$ be given, then $\bigwedge_{j\in J'} t_j = \perp$, so $u_i = \bigwedge_{j\in J'} t'_j <: t'_i$ if $i \in J$, and $u_i = t_i \wedge t'_i <: t'_i$ otherwise, so $\tau \oplus_{J'} \tau' <: \tau'$. If the other disjunct is $\perp$, we see that $\tau \oplus_{J'} \tau' <: \tau$ by a similar argument.
3. Note that $\bigwedge_{j\in J\cup\{l\}} t_j = \bigwedge_{j\in J} t_j$ and $\bigwedge_{j\in J\cup\{l\}} t'_j = \bigwedge_{j\in J} t'_j$, so $\tau \oplus_{J\cup\{l\}} \tau' <: \tau \oplus_J \tau'$.
4. The argument is similar to 2.
5. Writing J' for $J \cup \{j'\}$, we see that the partitioning will be the same for J and J'. Now consider the ith component of $\tau \otimes_J \tau'$. If $i \sim i'$ for some $i' \in J$, then $i \sim j'$, so we have

$$\begin{array}{rl} & (\tau \otimes_J \tau')_i \\ = & (\bigwedge_{j\in J} t_j \vee \bigwedge_{j\in J} t'_j) \wedge \bigwedge\{t_k \wedge t'_k \mid k \sim i, k \notin J\} \\ = & (\bigwedge_{j\in J} t_j \vee \bigwedge_{j\in J} t'_j) \wedge (t_{j'} \wedge t'_{j'}) \wedge \\ & \quad \bigwedge\{t_k \wedge t'_k \mid k \sim i, k \notin J'\} \\ = & (t'_{j'} \wedge \bigwedge_{j\in J'} t_j \vee t_{j'} \wedge \bigwedge_{j\in J'} t'_j) \wedge \\ & \quad \bigwedge\{t_k \wedge t'_k \mid k \sim i, k \notin J'\} \\ <: & (\bigwedge_{j\in J'} t_j \vee \bigwedge_{j\in J'} t'_j) \wedge \bigwedge\{t_k \wedge t'_k \mid k \sim i, k \notin J'\} \\ = & (\tau \oplus_{J'} \tau')_i \end{array}$$

   Otherwise we have $(\tau \otimes_J \tau')_i = (\tau \otimes_{J'} \tau')_i$, so overall $\tau \otimes_J \tau' <: \tau \otimes_{J'} \tau'$.
6. Obvious, since the components of $\tau \oplus_\exists \tau'$ are obtained by forming the meet.

□

Algorithm 3 ALL-CONSENSUS-OPT(n, τ, τ')

Require: Two TTCs τ and τ' of arity n.
Ensure: The lean set of all their consensus TTCs.

```
$\upsilon \leftarrow \tau \,\bar{\wedge}\, \tau'$
$N^\perp \leftarrow \emptyset$
$l, r \leftarrow \top$
for $i = 1$ to $n$ do
    if $\upsilon_i = \perp$ then
        $N^\perp \leftarrow N^\perp \cup \{i\}$
        $l, r \leftarrow l \wedge \tau_i, r \wedge \tau'_i$
$S \leftarrow$ ALL-CONSENSUS-REC$(n, \upsilon, \tau, l, \tau', r, N^\perp, 1)$
if $N^\perp = \emptyset$ then
    $S \leftarrow \max_\perp(S \cup \{\tau \oplus_\exists \tau'\})$
return $S$
```

This suggests an improved implementation of Algorithm 2, shown in two parts as Algorithm 3 and Algorithm 4, which straightforwardly exploit the above properties to avoid performing useless consensus operations whose result would be discarded by the $\max_\perp$ operator anyway. In the latter, we make use of an operation PART(υ, i) which returns the equivalence partition index i belongs to in TTC υ.

To show that these improvements make our type inference algorithm feasible in practice, we measure its performance on some typical programs from our intended application domain. The Datalog programs to be typed arise as an intermediate representation of programs written in a high-level object-oriented query language named *.QL* [14, 15], which are optimised and then compiled to one of several low-level representations for execution.

We measure the time it takes to infer types for the 89 queries in the so-called "Full Analysis" that ships with our source code analysis tool *SemmleCode*. These queries provide statistics and overview data on structural properties of the analysed code base, compute code metrics, and check for common mistakes or dubious coding practices.

Algorithm 4 ALL-CONSENSUS-REC$(n, \upsilon, \tau, l, \tau', r, J, i)$

Require: Two TTCs τ and τ' of arity n, their meet υ, a set $J \subseteq \{1, \ldots, n\}$, an index $1 \leq i \leq n$, and two partial results $l = \bigwedge\{j \in J \mid t_j\}$, $r = \bigwedge\{j \in J \mid t'_j\}$.

Ensure: The lean set of all consensus TTCs of τ and τ' over index sets $J' \supseteq J$ such that $J \cap \{1, \ldots, i-1\} = J' \cap \{1, \ldots, i-1\}$.

```
if $l = \bot \vee r = \bot$ then
    return $\emptyset$
if $i = n$ then
    if $l <: r \vee r <: l$ then
        return $\emptyset$
    return $\{\tau \oplus_J \tau'\}$
$S \leftarrow$ ALL-CONSENSUS-REC$(n, \upsilon, \tau, l, \tau', r, J, i+1)$
if $i \in J \vee l <: \tau_i \wedge r <: \tau'_i$ then
    return $S$
$l, r \leftarrow l \wedge \tau_i, r \wedge \tau'_i$
$J \leftarrow J \cup$ PART$(\upsilon, i)$
$i \leftarrow i + 1$
$S \leftarrow \max_\bot(S \cup$ ALL-CONSENSUS-REC$(n, \upsilon, \tau, l, \tau', r, J, i))$
return $S$
```

A type inference for the query being compiled is performed at three different points during the optimisation process, giving a total of 267 time measurements. All times were measured on a machine running a Java 1.6 virtual machine under Linux 2.6.28-13 on an Intel Core2 Duo at 2.1 GHz with four gigabytes of RAM, averaging over ten runs, and for each value discarding the lowest and highest reading.

Of the 267 calls to type inference, 65% finish in 0.5 seconds or less, 71% take no more than one second, 93% no more than two, and in 98% of the cases type inference is done in under three seconds. Only two type inferences take longer than four seconds, at around 4.5 seconds each.

The size of the programs for which types are inferred varies greatly, with most programs containing between 500 and 1500 subterms, but some programs are significantly larger at more than 3000 subterms. Interestingly, program size and type inference time are only very weakly correlated ($\rho < 0.4$), and the correlation between the number of stratification layers (which roughly corresponds to the number of fixpoint computations to be done) and type inference time is also not convincing ($\rho < 0.6$).

The low correlation is evident in Fig. 6 which shows the type inference time plotted on the *y*-axis versus the depth (*i.e.*, number of layers) of the program on the *x*-axis. In particular, the two cases in which type inference takes longer than four seconds are both of medium size and not among the deepest either.

This suggests that the asymptotic behaviour of the algorithm in terms of input size is masked by the strong influence of implementation details (at least for the kind of programs we would expect to deal with in our applications), and we can expect significant performance gains from fine tuning the implementation.

Our experiments also show that saturation, while extremely expensive in theory, is quite well-behaved in practice: in 78% of all cases, no new TTCs are added during the first iteration of Algorithm 1, so the typing is already saturated. In another 17% of all cases we need only one more iteration, and we almost never ($\approx 0.01\%$) need to do more than four iterations.

Although in every individual iteration we potentially have to take the consensus over every combination of columns, our heuristics manage to exclude all combinations of more than one column in 94% of all cases, and sometimes (14%) can even show that no consensus formation is needed at all.

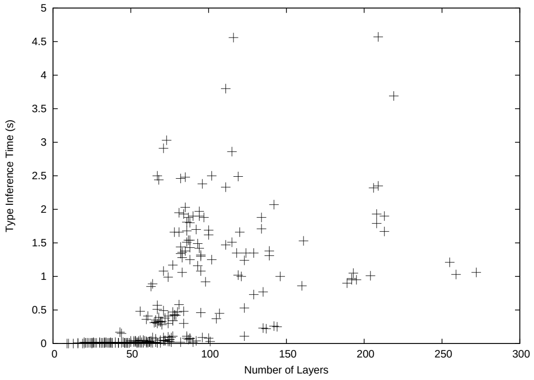

Figure 6. Depth of Programs and Type Inference Time

Since our type inference algorithm makes heavy use of BDDs, some statistics about their usage may also be of interest. We use about 300 BDD variables (one for every type symbol), with most of the BDDs constructed during type checking being of rather moderate size: the BDD to represent the type hierarchy (which encodes about 800 constraints automatically derived during translation from .*QL* to Datalog) occupies around 4000 nodes, while the BDDs for individual type propositions never take more than 100 nodes.

While we have anecdotal evidence to show that the optimisation techniques we have presented in earlier work [13] benefit greatly from combining them with the richer type hierarchies our type system supports, we leave the precise investigation of this matter to future work.

6. Related Work

We now discuss the relation of the present paper to previous work of others, as well as an earlier paper by ourselves.

Type inference for logic programs The present paper builds on a long and rich tradition of type inference for logic programs. An early proposal is that of Frühwirth *et al.* [16], which computes types on a field-by-field basis. Heintze and Jaffar [19] extract a system of set constraints from a logic program that likewise yield a field-by-field typing.

The advantage of these approaches is the efficiency of type checking, but as we have already indicated, in our experience it leads to an unacceptable loss of precision. Many others have since applied abstract interpretation to infer types for logic programs, *e.g.* [21, 18, 17, 5].

Interestingly, Bachmair *et al.* [3] have shown that Heintze and Jaffar's set constraints correspond to formulae of monadic first-order logic, so there is yet another connection between this logic and types for logic programs, albeit of a very different character.

In earlier work of our own, we put forward a type inference procedure for Datalog that bears a superficial resemblance to the present proposal [13]. The differences are however profound:

- In [13], we employed a similar notion of TTCs, but the notion of typing was syntactic rather than semantic: in particular, we used the syntactic check <: for subtype inclusion. The most important advance made here is a clean semantic framework (separating the syntactic procedure from the semantic goal of over-approximation), and the use of prime implicants to make the syntactic check complete.

- Furthermore, in [13] we placed restrictions on the type hierarchy, in particular we assumed just a subtype order and did not allow separate statements (of disjointness, equivalence and so on) as we do here. This is a very important improvement in practice, as realistic databases require such complex type hierarchies. Indeed we were forced to make this generalisation at the insistence of a client of our company.
- Also, we have found in practice it is very important to handle negated types, so that predicates like $\mathit{employee}(x) \wedge \neg \mathit{student}(x)$ are bona fide types that can be inferred. In our earlier work, it was impossible to handle negated types accurately.
- Finally, here we handle inhabitation constraints (existentials in type programs) precisely. There was no support for that in [13].

In summary, we have made substantial improvements over our own earlier proposal with a clean semantic approach and an elegant, natural choice of type language.

Recently, Zook *et al.* [27] of LogicBlox, Inc., have presented a type checking algorithm for Datalog that supports inclusion constraints. One of their main goals is to be able to implement the algorithm itself in Datalog, so it has to be polynomial time and hence cannot be complete.

We achieve more pleasing theoretical properties and can support a richer language of type constraints by sacrificing polynomial time guarantees, although our experiments show that this is a reasonable tradeoff for our application area.

Upper envelopes We have already alluded to the earlier work of Chaudhuri and Kolaitis in this area [8, 9]. They investigate the approximation of a Datalog program by a disjunction of conjunctive queries: an over-approximation is called an upper envelope, and an under-approximation a lower envelope. They show that in general it is not possible to compute a tightest upper envelope.

The conjunctive queries in an upper envelope are akin to our TTCs. However, while upper envelopes can contain extensional relation symbols of arity greater than one, TTCs are restricted to monadic extensionals. We have demonstrated that with this definition, it is possible to compute a tightest upper envelope.

Chaudhuri shows that when Cartesian products are forbidden (in so-called *connected envelopes*) tightest envelopes can be computed as well; this restriction seems unnatural in the context of type inference, but it would be interesting to see whether his techniques can be adapted to our setting.

Containment checking We have already mentioned the landmark result of Shmueli [23] that precludes decidability of general program containment. Much research has been done on restricted classes of programs for which containment is decidable, such as the classic result by Chandra and Merlin [7] showing NP-completeness of the containment problem for conjunctive queries.

Recently, Wei and Lausen [24] have investigated the containment problem for conjunctive queries in the presence of negation. While their algorithm also reduces the containment check to a series of checks on a family of extended queries, which is somewhat reminiscent of our reduction to component-wise checking, the technical details are quite different and not directly related to type checking.

Containment of monadic Datalog programs was proved decidable by Cosmadakis *et al.* [11]. In contrast to our work, however, they handle Datalog programs in which all *intensional* predicates are monadic (and extensionals can be of arbitrary arity), whereas our type programs have monadic *extensionals* (and put no restriction on the arities of intensional predicates).

Alternative Implementations While the performance of our prototype implementation is promising, other implementation approaches certainly exist and may be worth exploring. Bachmair *et al.* [4] develop a decision procedure for monadic first order logic based on the superposition calculus. Since type programs can readily be expressed as monadic first order formulae, their algorithm could be used to decide type containment. Another possibility would be to use Ackermann's translation from monadic first order logic to equality logic [2], and then employ a decision procedure for this latter logic.

7. Conclusion

We have presented a type inference procedure that assigns an upper envelope to each Datalog program. That envelope is itself a Datalog program that makes calls to monadic extensionals only. The algorithm is able to cope with complex type hierarchies, which may include statements of implication, equivalence and disjointness of entity types.

The type inference procedure is itself an extremely simple syntactic mapping. The hard work goes into an efficient method of checking containment between type programs. We achieve this via a novel adaption of Quine's algorithm for the computation of the prime implicants of a logical formula. Generalising from logical formulae to type programs, we bring types into a saturated form on which containment is easily checked.

As shown by our experiments, the algorithm for inferring a type and saturating it works well on practical examples. While it may still exhibit exponential behaviour in the worst case, such extreme cases do not seem to arise in our application area. Thus our algorithm is a marked improvement over well-known simpler constructions that always require an exponential overhead.

Many avenues for further work remain. Perhaps the most challenging of these is the production of good error messages when type errors are identified: this requires printing the Boolean formulae represented via TTCs in legible form. We have made some progress on this, employing Coudert *et al.*'s restrict operator on BDDs, which is another application of prime implicants [12].

There is also substantial further engineering work to be done in the implementation. Careful inspection of the statistics show that our use of BDDs is very much unlike their use in typical model checking applications [26], and we believe this could be exploited in the use of a specialised BDD package. For now we are using JavaBDD [25], which is a literal translation of a C-based BDD package into Java.

Finally, we will need to investigate how to best exploit the advanced features offered by our type system. In particular, much experience remains to be gained in how to make it easy and natural for the programmer to specify the constraints making up the type hierarchy, and which kinds of constraints benefit which kinds of programs.

Acknowledgments

We would like to thank Molham Aref for insisting that type inference must handle complex type hierarchies including statements of disjointness, implication and equivalence. His enthusiasm has been an inspiration throughout this project.

Damien Sereni has contributed crucial insights to this paper, in particular he pointed out the incompleteness of the syntactic subtyping check and came up with the geometric interpretation. Pavel Avgustinov showed incredible tenacity in tracking down insidious bugs. Julian Tibble suggested the BDD variable ordering and made sure we got it right in our implementation.

Michael Benedikt, Damien Sereni, and the anonymous reviewers provided valuable comments on earlier versions of this paper.

Notice *The technology described in this paper is proprietary; U.S. and other patents pending. For licensing information, write to* `licenses@semmle.com`.

References

[1] Serge Abiteboul, Georg Lausen, Heinz Uphoff, and Emmanuel Waller. Methods and rules. In *ACM SIGMOD International Conference on Management of Data*, pages 32–41. ACM Press, 1993.

[2] Wilhelm Ackermann. *Solvable Cases of the Decision Problem*. North-Holland Publishing Company, Amsterdam, 1954.

[3] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Set constraints are the monadic class. In *Logic in Computer Science*, pages 75–83, 1993.

[4] Leo Bachmair, Harald Ganzinger, and Uwe Waldmann. Superposition with Simplification as a Decision Procedure for the Monadic Class with Equality. In *KGC '93: Proceedings of the Third Kurt Gödel Colloquium on Computational Logic and Proof Theory*, pages 83–96, London, UK, 1993. Springer-Verlag.

[5] Sacha Berger, Emmanuel Coquery, Wlodzimierz Drabent, and Artur Wilk. Descriptive Typing Rules for Xcerpt and their soundness. In François Fages and Sylvain Soliman, editors, *Principles and Practice of Semantic Web Reasoning*, volume 3703 of *LNCS*, pages 85–100. Springer, 2005.

[6] George S. Boolos and Richard C. Jeffrey. *Computability and Logic*. Cambridge University Press, 3rd edition, 1989.

[7] Ashok K. Chandra and Philip M. Merlin. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 77–90, New York, NY, USA, 1977. ACM.

[8] Surajit Chaudhuri. Finding nonrecursive envelopes for Datalog predicates. In *Principles of Database Systems (PODS)*, pages 135–146, 1993.

[9] Surajit Chaudhuri and Phokion G. Kolaitis. Can Datalog be approximated? In *Principles of Database Systems (PODS)*, pages 86–96, 1994.

[10] Kevin J. Compton. Stratified least fixpoint logic. *Theoretical Computer Science*, 131(1):95–120, 1994.

[11] Stavros Cosmadakis, Haim Gaifman, Paris Kanellakis, and Moshe Vardi. Decidable Optimization Problems for Database Logic Programs. In *Proceedings of the 20th annual ACM Symposium on Computing*, pages 477–490, 1988.

[12] Olivier Coudert, Christian Berthet, and Jean Christophe Madre. Verification of synchronous sequential machines based on symbolic execution. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 365–373. Springer, 1989.

[13] Oege de Moor, Damien Sereni, Pavel Avgustinov, and Mathieu Verbaere. Type Inference for Datalog and Its Application to Query Optimisation. In *PODS '08: Proceedings of the twenty-seventh ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 291–300, New York, NY, USA, 2008. ACM.

[14] Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. .QL: Object-oriented queries made easy. In Ralf Lämmel, João Saraiva, and Joost Visser, editors, *Generative and Transformational Techniques in Software Engineering*, LNCS. Springer, 2007.

[15] Oege de Moor, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. .QL for source code analysis. In *Source Code Analysis and Manipulation (SCAM '07)*, pages 3–16. IEEE, 2007.

[16] Thom W. Frühwirth, Ehud Y. Shapiro, Moshe Y. Vardi, and Eyal Yardeni. Logic programs as types for logic programs. In *Logic in Computer Science (LICS)*, pages 300–309. IEEE Computer Society, 1991.

[17] John P. Gallagher, Kim S. Henriksen, and Gourinath Banda. Techniques for Scaling Up Analyses Based on Pre-Interpretations. In M. Gabbrielli and G. Gupta, editors, *International Conference on Logic Programming (ICLP '05)*, volume 3668 of *LNCS*, pages 280–296. Springer, 2005.

[18] John P. Gallagher and Germán Puebla. Abstract Interpretation over Non-deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Practical Aspects of Declarative Languages (PADL)*, volume 2257 of *LNCS*, pages 243–261. Springer, 2002.

[19] Nevin C. Heintze and Joxan Jaffar. A finite presentation theorem for approximating logic programs. In *Symposium on Principles of Programming Languages (POPL)*, pages 197–209, 1990.

[20] David Hilbert and Wilhelm Ackermann. *Grundzüge der Theoretischen Logik*. Julius Springer, Berlin, 1928.

[21] Kim Marriott, Harald Søndergaard, and Neil D. Jones. Denotational Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):607–648, 1994.

[22] Willard V. Quine. On Cores and Prime Implicants of Truth Functions. *The American Mathematical Monthly*, 66(9):755–760, Nov. 1959.

[23] Oded Shmueli. Equivalence of Datalog Queries is Undecidable. *Journal of Logic Programming*, 15(3):231–241, 1993.

[24] Fang Wei and Georg Lausen. Containment of Conjunctive Queries with Safe Negation. In *ICDT '03: Proceedings of the 9th International Conference on Database Theory*, pages 346–360, London, UK, 2002. Springer-Verlag.

[25] John Whaley. JavaBDD. http://javabdd.sourceforge.net/, 2007.

[26] Bwolen Yang, Randal E. Bryant, David R. O'Halloran, Armin Biere, Olivier Coudert, Geert Janssen, Rajeev K. Ranjan, and Fabio Somenzi. A Performance Study of BDD-Based Model Checking. In *2nd International Conference on Formal Methods in Computer-Aided Design*, volume 1522 of *Lecture Notes in Computer Science*, pages 255–289. Springer, 1998.

[27] David Zook, Emir Pasalic, and Beata Sarna-Starosta. Typed Datalog. In *Practical Aspects of Declarative Languages*, pages 168–182. Springer Berlin/Heidelberg, 2009.