From Monomorphic to Polymorphic Well-Typings and Beyond

Schrijvers, Tom; Bruynooghe, Maurice; Gallagher, John Patrick

From Monomorphic to Polymorphic
Well-Typings and Beyond

Tom Schrijvers[1]⋆, Maurice Bruynooghe[1], and John P. Gallagher[2]⋆⋆

[1] Dept. of Computer Science, K.U.Leuven, Belgium
[2] Dept. of Computer Science, Roskilde University, Denmark

Abstract. Type information has many applications; it can e.g. be used in optimized compilation, termination analysis and error detection. However, logic programs are typically untyped. A well-typed program has the property that it behaves identically on well-typed goals with or without type checking. Hence the automatic inference of a well-typing is worthwhile. Existing inferences are either cheap and inaccurate, or accurate and expensive. By giving up the requirement that all calls to a predicate have types that are instances of a unique polymorphic type but instead allowing multiple polymorphic typings for the same predicate, we obtain a novel strongly-connected-component-based analysis that provides a good compromise between accuracy and computational cost.

1 Introduction

While type information has many useful applications, e.g. in optimized compilation, termination analysis, documentation and debugging, most logic programming languages are untyped. In [4], Mycroft and O'Keefe propose a polymorphic type scheme for Prolog which makes static type checking possible and has the guarantee that well-typed programs behave identically with or without type checking, i.e., the types do not affect the execution (of well-typed goals). Lakshman and Reddy [3] provide a type reconstruction algorithm that, given the type definitions, infers missing predicate signatures.

Our aim is to construct a well-typing for a logic program automatically without prescribing any types or signatures. While there has been plenty of other work on such completely automatic type inference for logic programs (sometimes called "descriptive" typing), the goal was always to construct *success types*, that is, an approximation of the success set of a program represented by typed predicates. A well-typing by contrast represents in general neither an over- nor an under-approximation of the success set of the program. It was, to the best of our knowledge, not until [1] that a method was introduced to infer a well-typing for logic programs automatically, without given type definitions. The paper describes how to infer a so-called monomorphic well-typing which derives a type signature for every predicate. The well-typing has the property that the type

signature of each call is identical to that of the predicate signature. Below is a code fragment, followed by the results of the inference using the method of [1].

Example 1.

```
p(R) :- app([a],[b],M), app([M],[M],R).
app([],L,L).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).
% type definition:
:- type ablist ---> [] ; a ; b ; [ablist | ablist].
% predicate signatures:
:- app(ablist,ablist,ablist).
:- p(ablist).
```

Note that the list type `ablist` is not the standard one. The reason is that `app/3` is called once with lists of `a`s and `b`s and once with lists whose elements are those lists. The well-typing constraint, stating that both calls must have the same signature as the predicate `app/3` enforces the above unnatural solution. Hence, the monomorphic type inference is not so interesting for large programs as they typically use many different type instances of commonly used predicates.

In a language with polymorphic types such as Mercury, [6], one typically declares `app/3` as having type `app(list(T),list(T),list(T))`. The first call instantiates the type parameter `T` with the type `elem` defined as `elem ---> a ; b` while the second call instantiates `T` with `list(elem)`.

The sets of terms denoted by these polymorphic types `list(elem)` and `list(list(elem))` are proper subsets of the monomorphic type `ablist`. For instance, the term `[a|b]` is of type `ablist`, but not of type `list(elem)`. Hence, polymorphic types allow for a more accurate characterization of program terms.

The work in [1] also sketches the inference of a polymorphic well-typing. However, the rules presented there are incomplete. We refer to [5] for a comprehensive set. In this paper, we revisit the problem of inferring a polymorphic typing. However, we impose the restriction that calls to a predicate that occur inside the strongly connected component (SCC) that defines the predicate (in what follows we refer to them as recursive calls, although there can also be mutual recursion) have the same type signature as the predicate. Other calls, appearing higher in the call graph of the program have a type signature that is a polymorphic instance of the definition's type. The motivation of the restriction is that it can be computationally very demanding when a recursive call is allowed to have a type that is a true instance of the definition's type. Indeed, [2] showed that type checking in a similar setting is undecidable, and [5] gave strong indications that this undecidability also holds for type inference in the above setting. Applied on Example 1, polymorphic type inference [5] gives:

Example 2.

```
:- type elem  ---> a ; b.
:- type list1(T) ---> [] ; [T | list1(T)].
:- type list2(T) ---> [] ; [T | list2(T)].
:- app(list1(T),list2(T),list2(T)).
:- p(list2(list2(elem))).
:- call app1(list1(elem), list2(elem), list2(elem)).
```

2

```
            :- call app2(list1(list2(elem)),
                       list2(list2(elem)), list2(list2(elem))).
```

Both `list1` and `list2` are renamings of the standard `list` type, hence this well-typing is equivalent to what one would declare in a language such as Mercury. As for the type signatures of the calls, `call app`*i* refers to the *i*th call to `app/3` in `p/1`'s clause. In the first call, the polymorphic parameter is instantiated by `elem`, in the second by `list2(elem)`.

However, applying the analysis on a fragment where the second argument of the first call to `app/3` is not a list but a constant, one obtains a different result.

Example 3.
```
        q(R) :- app([a],b,M), app([M],[M],R).
        % type definition
        :- type elem ---> a.                          % <<<
        :- type list(T) ---> [] ; [T|list(T)] .
        :- type blist(T) ---> [] ; [T|blist(T)] ; b.   % <<<
        % signatures
        :- app(list(T),blist(T),blist(T)).
        :- q(blist(blist(elem))).
        :- call app1(list(elem), blist(elem), blist(elem)).
        :- call app2(list(blist(elem)),
                    blist(blist(elem)), blist(blist(elem))).
```

Note that the "erroneous" call spoils the type of `app/3` by constructing a type that makes that call well-typed. Indeed, the type `blist(T)` has an extra base case with the functor `b`. Moreover, it is not clear from the type information which call is at the origin of the spoiled type. This drawback, together with the high computational cost of the polymorphic analysis motivated us to search for another solution. This led to a third approach where different non-recursive calls to the same predicate can be instances of different polymorphic well-typings of that predicate. Moreover, the inference can be done SCC by SCC, which lowers its computational cost. For the lowest SCC, defining `app/3`, we obtain:

Example 4.
```
        :- type list(T)    ---> [] ; [T | list(T)].
        :- type stream(T) ---> [T | stream(T)].
        % signatures
        :- app(list(T),stream(T),stream(T)).
```

Note the `stream(T)` type for the second and third argument. This is a well-typing. Nothing in the definition enforces the list structure to be terminated by an empty list, hence this case is absent in the type for second and third argument. Note that neither of the two types is an instance of the other one.

For the SCC defining `p/1` one obtains the following.

Example 5.
```
        :- type elem ---> a ; b.
        :- type elist1 ---> [elem|elist1] ; [].
        :- type elist2 ---> [elem|elist2] ; [].
```

3

```
:- type elistlist1 ---> [elist2|elistlist1] ; [].
:- type elistlist2 ---> [elist2|elistlist2] ; [].
% signatures
:- p(elistlist2).
:- call app1(elist1, elist2, elist2).
:- call app2(elistlist1, elistlist2, elistlist2).
```

In this SCC, `app/3` is called with types that are instances of lists. These instances are represented as monomorphic types; with a small extra computational effort, one could separate them in the underlying polymorphic type and the parameter instances. Finally, for the SCC defining `q/1` one obtains the following type.

Example 6.
```
:- type elem ---> a.                          % <<<
:- type elist1 ---> [elem|elist1] ; [].
:- type eblist2 ---> [elem|eblist2] ; b.       % <<<
:- type elistlist1 ---> [eblist2|elistlist1] ; [].
:- type elistlist2 ---> [eblist2|elistlist2] ; [].
% signatures
:- q(elistlist2).
:- call app1(elist1, eblist2, eblist2).
:- call app2(elistlist1, elistlist2, elistlist2).
```

This reveals that `eblist2` is not a standard list and that it is the first call to `app/3` that employs this type.

This example shows that the SCC-based polymorphic analysis provides more useful information than the true polymorphic one, identifying which parts of a predicate's type originate in the predicate definition itself and which in the respective calls to that predicate. It is interesting also in giving up the usual concept underlying polymorphic typing that each predicate should have a unique principal typing and that all calls to that predicate should have a type that is an instance of it. The types of the first and second call to `app` are equivalent to instances of the type signatures `app(list1(T),blist2(T),blist2(T))` and `app(list1(T),list2(T),list2(T))` respectively, where the types `list1(T)` and `list2(T)` are the standard polymorphic list types but `blist2(T)` is defined as `blist2(T) ---> [T|eblist2(T)] ; b`.

Our contributions are summarized as follows.

- A new and efficient SCC-based polymorphic type analysis.
- A comparison with two other analyses, a cheap but inaccurate monomorphic analysis and an accurate but expensive polymorphic analysis.
- An evaluation showing the respective merits of the different analyses.

In Section 2, we introduce the necessary background on logic programs, types and well-typings and introduce a notion of minimal well-typing. In Section 3, we recall the previous work on monomorphic type inference. We do the same, very briefly, in Section 4 for the polymorphic type inference. In Section 5, we introduce our new SCC-based analysis. We experimentally compare the three approaches in Section 6 and conclude in Section 7.

2 Problem Statement and Background Knowledge

Logic Programs The syntax of logic programs considered here is as follows.

```
Program  := Clause*;
Clause   := Atom ':-' Goal;
Goal     := Atom | '(' Goal , Goal ')' | Term '=' Term | 'true' ;
Atom     := Pred '(' Term ',' ... ',' Term ')' ;
Term     := Var | Functor '(' Term ',' ... ',' Term ')' ;
```

`Pred`, `Functor` and `Var` refer to sets of predicate symbols, function symbols and variables respectively. Elements of `Pred` and `Functor` start with a lower case letter, whereas elements of `Var` start with an upper case letter.

Types For type definitions, we adopt the syntax of Mercury [6]. The set of *Type expressions (types)* $\mathcal{T}$ consists of terms constructed from an infinite set of type variables (parameters) $\mathcal{V}_{\mathcal{T}}$ and a finite alphabet of ranked type symbols $\Sigma_{\mathcal{T}}$; these are distinguishable —by context— from the set of variables V and alphabet of functors Σ used to construct terms. Variable-free types are called monomorphic; the others polymorphic. Type substitutions of the form $\{T_1/\tau_1, \ldots, T_n/\tau_n\}$, where T_i and τ_i $(1 \leq i \leq n)$ are parameters and types respectively, define mappings from types to types by the simultaneous replacement of each parameter T_i by the corresponding type τ_i.

Definition 1 (Type definition). *A* type rule *for a type symbol $h/n \in \Sigma_{\mathcal{T}}$ is of the form*

$$h(\overline{T}) \longrightarrow f_1(\bar{\tau}_1); \ldots; f_k(\bar{\tau}_k); \quad (k \geq 1)$$

where $\overline{T}$ is an n-tuple of distinct type variables, $f_1, \ldots, f_k$ are distinct function symbols from Σ, $\bar{\tau}_i$ $(1 \leq i \leq k)$ are tuples of corresponding arity from $\mathcal{T}$, and type variables in the right hand side, if any, are from $\overline{T}$. A type definition *is a finite set of type rules where no two left hand sides contain the same type symbol, and there is a rule for each type symbol occurring in the type rules.*

If $f_i(\bar{\tau}_i)$ is one of the alternatives in the type rule for $h(\bar{T})$, then the mapping $\bar{\tau}_i \rightarrow h(\bar{T})$ can be considered the type signature of the function symbol f_i. As in Mercury [6], a function symbol can occur in several type rules, hence can have several type signatures; we say its type is *overloaded*.

The instance relation – $t(\bar{\tau})$ being an instance of $t(\bar{T})$ – is a partial order over the set of types $\mathcal{T}$. However, we want to define a more expressive partial order, denoted $\leq$ (and $<$ when strict). While $list(T_1) \leq list(list(T_2))$ in the instance relation, we also want, in Example 4, that $stream(T) < list(T)$ as the former has less alternatives than the latter. More subtly, with $stream(T, A) \longrightarrow [T|A]$, we want $stream(T, A) < stream(T)$. With $stream1(T) \longrightarrow [T|stream2(T)]$ and $stream2(T) \longrightarrow [T|stream1(T)]$, we want $stream1(T) < stream(T)$ and $stream2(T) < stream(T)$ as strict relationships. The following definition defines a preorder achieving this.

Definition 2 (≤ Type inclusion). *A type τ is included in a type σ ($\tau \leq \sigma$) if there exist (i) a function ρ mapping non variable types used in τ to non variable types used in σ such that $\tau\rho = \sigma$ and (ii) a type substitution θ such that for each type $t(\bar{\tau})$ in the domain of ρ we can show that $t(\bar{\tau})$ is included in $t(\bar{\tau})\rho$.*

To verify (ii), assuming s is the type symbol in $t(\bar{\tau})\rho$ and the type rules for t and s are $t(\bar{T}) \longrightarrow f_1(\bar{\tau}_1); \ldots; f_m(\bar{\tau}_m)$ and $s(\bar{S}) \longrightarrow g_1(\bar{\sigma}_1); \ldots; g_n(\bar{\sigma}_n)$ respectively, we check that there exists a $\bar{\sigma}$ such that for each alternative $f_i(\bar{\tau_i})$ in the type rule for $t(\bar{T})$ there is an alternative $g_j(\bar{\sigma_j})$ such that $f_i = g_j$ and for each k, either $\tau_{i,k}\{\bar{T}/\bar{\tau}\}\rho\theta = \sigma_{j,k}\{\bar{S}/\bar{\sigma}\}$ or $\tau_{i,k}\{\bar{T}/\bar{\tau}\}\rho\theta \leq \sigma_{j,k}\{\bar{S}/\bar{\sigma}\}$.

Type inclusion is a preorder; using it to define equivalence classes, one obtains a partial order (which for convenience we also denote $\leq$). For the "subtle" examples above, $stream(T, A) < stream(T)$ with ρ and θ given by $\rho(stream(T, A)) = stream(T)$ and $\theta = \{A/stream(T)\}$. Similarly, $stream1(T) < stream(T)$, with $\rho(stream1(T)) = stream(T)$, $\rho(stream2(T)) = stream(T)$, and θ the empty substitution. As a final example, the type $elistlist1$ of Example 5 is equivalent to the instance $list(list(elem))$ of the polymorphic type $list(T)$. In one direction, $\rho(elistlist1) = list(list(elem))$, $\rho(elist2) = list(elem)$, and θ is empty. In the other direction, $\rho(list(list(elem))) = elistlist1$ and $\rho(list(elem)) = elist2$ while θ is also empty. Note that $list(list(T)) < elistlist1$ with $\rho(list(list(T))) = elistlist1$, $\rho(list(T)) = elist2$ and $\theta = \{T/elem\}$.

$$(\textsc{Var}) \quad \Gamma, X : \tau \vdash X : \tau$$

$$(\textsc{Term}) \quad \frac{(\texttt{:- type } \tau \longrightarrow \ldots \texttt{ ; } f(\tau_1, \ldots, \tau_n) \texttt{ ; } \ldots) \in \Gamma \qquad \Gamma \vdash t_i : \tau_i' \qquad \tau_i' = \tau_i\theta}{\Gamma \vdash f(t_1, \ldots, t_n) : \tau\theta}$$

$$(\textsc{True}) \quad \Gamma \vdash \texttt{true} : \diamond$$

$$(\textsc{Unif}) \quad \frac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1\texttt{=}t_2 : \diamond}$$

$$(\textsc{Conj}) \quad \frac{\Gamma \vdash g_1 : \diamond \qquad \Gamma \vdash g_2 : \diamond}{\Gamma \vdash (g_1\texttt{,}g_2) : \diamond}$$

$$(\textsc{Clause}) \quad \frac{p(\tau_1, \ldots, \tau_n) \in \Gamma \qquad \Gamma \vdash t_i : \tau_i \qquad \Gamma \vdash g : \diamond}{\Gamma \vdash p(t_1, \ldots, t_n) \texttt{ :- } g : \diamond}$$

$$(\textsc{Prog}) \quad \frac{\Gamma \vdash a_i \texttt{ :- } g_i : \diamond}{\Gamma \vdash \{\overline{a_i \texttt{ :- } g_i}\} : \diamond}$$

Fig. 1. The Common Type Judgement Rules

Typing Judgements A *predicate signature* is of the form $p(\bar{\tau})$ and declares a type τ_i for every argument of predicate p. A type environment E for a program $\mathcal{P}$ is a set of typings $X : \tau$, one for every variable X in $\mathcal{P}$, and predicate signatures $p(\bar{\tau})$, one for every predicate p in $\mathcal{P}$, and a type definition. A typing judgement $E \vdash e : \tau$ asserts that e has type τ for the *type environment E* and $E \vdash e : \diamond$

asserts that e is well-typed. A typing judgement is valid if it respects the typing rules of the type system. We will consider three different type systems, but they differ only in one place, namely in the typing of predicate calls in rule bodies. Figure 1 shows the typing rules for all the other language constructs, common to all type systems. The VAR rule states that a variable is typed as given in the type environment. The TERM rule constructs the type of a compound term (the quantifier $\forall i$ is omitted); the other rules state the well-typing of atoms and that a program is well-typed when all its parts are.

$$
\begin{array}{ll}
(\text{MonoCall}) & \dfrac{p(\tau_1,\ldots,\tau_n) \in \Gamma \qquad \Gamma \vdash t_i : \tau_i}{\Gamma \vdash p(t_1,\ldots,t_n) : \diamond} \\[2ex]
(\text{RecCall}) & \dfrac{p(\tau_1,\ldots,\tau_n) \in \Gamma \qquad \Gamma \vdash t_i : \tau_i}{\Gamma \vdash p(t_1,\ldots,t_n) : \diamond} \\[2ex]
(\text{PolyCall}) & \dfrac{p(\tau_1,\ldots,\tau_n) \in \Gamma \qquad \Gamma \vdash t_i : \tau_i' \qquad \tau_i' = \tau_i\theta}{\Gamma \vdash p(t_1,\ldots,t_n) : \diamond} \\[2ex]
(\text{SCCCall}) & \dfrac{\begin{array}{c} \Gamma \vdash t_i : \tau_i' \qquad (\texttt{:- type } \tau_i' \longrightarrow \ldots) \in \Gamma \\ \Gamma' \cup \{\texttt{:- type } \tau_i' \longrightarrow \ldots\} \cup \{p(\tau_1',\ldots,\tau_n')\} \vdash subprog(p/n) : \diamond \end{array}}{\Gamma \vdash p(t_1,\ldots,t_n) : \diamond}
\end{array}
$$

Fig. 2. The Call Rules

The different ways of well-typing a call are given in Figure 2. For the monomorphic analysis, the well-typing of a call is identical to that of the predicate in the environment (MonoCall rule). For the other two analyses, this only holds for the recursive calls (RecCall rule). The polymorphic analysis requires that the type of a non-recursive call is an instance (under type substitution θ) of the type of the predicate (PolyCall rule), while the SCC based analysis (SCCCall rule) requires that the well-typing of the call in Γ —which has predicate signature $p(\tau_1',\ldots,\tau_n')$— is such that there exists a typing environment (that can be different from Γ) with the following properties: the subprogram defining the predicate ($subprog(p/n)$) is well-typed in Γ' and the predicate signature of p/n is $p(\tau_1',\ldots,\tau_n')$ itself. Note that this implies that there exists a polymorphic type signature for p/n such that $p(\tau_1',\ldots,\tau_n')$ is equivalent to an instance of it; however, that polymorphic type can be different for different calls.

In all three analyses, we are interested in *least* solutions. To define formally the notion of least solution, we define a partial order on predicate signatures.

Definition 3 ($\preceq$, preorder on predicate signatures). *A predicate signature $p(\bar{\tau})$ is smaller than a predicate signature $p(\bar{\sigma})$ ($p(\bar{\tau}) \preceq p(\bar{\sigma})$) iff there is a mapping ρ from the types in $\bar{\tau}$ to larger types (for all i, $\tau_i \leq \tau_i\rho$) that preserves the type variables – it can possibly introduce extra type variables – such that $p(\bar{\tau})\rho\theta = p(\bar{\sigma})$ for some type substitution θ.*

This preorder can be extended to a partial order over equivalence classes. For convenience, we denote the latter also as $\preceq$, and write $\prec$ when the order is strict. With $p(\bar{\tau}_T)$ denoting the predicate signature of predicate p in a well-typing T, we can now also define a partial order over (equivalence classes) of well-typings:

Definition 4 ($\preceq$ Partial order on well-typings). *A well-typing T_1 of a program is smaller than well-typing T_2 of the same program ($T_1 \preceq T_2$) if, for each predicate p/n, $p(\bar{\tau}_{T_1}) \preceq p(\bar{\tau}_{T_2})$.*

We can now say that a well-typing of a program is a *least well-typing* if it is smaller than any other well-typing of the same program.

For example, consider the polymorphic types of `app/3` in Examples 2 and 4. We have $app(list(T), stream(T), stream(T)) \prec app(list1(T), list2(T), list2(T))$ using the mapping $\rho(list(T)) = list1(T)$ and $\rho(stream(T)) = list2(T)$ and θ the identity mapping to satisfy Definition 3. Also, $app(list1(T), list2(T), list2(T)) \prec app(list(T), list(T), list(T))$, with $\rho(list1(T)) = list(T)$, $\rho(list2(T)) = list(T)$.

Note that $app(list(T), list(T), list(T)) \preceq app(list1(T), list2(T), list2(T))$ is not the case. The latter well-typing is "better" than the former because it expresses that there can be no aliasing between the backbones of the lists in the first argument and those in the second argument. Note that `app/3` has many other well-typings such as $app(list(list(T)), stream(list(T)), stream(list(T)))$ and $app(list(list(T)), list(list(T)), list(list(T)))$. All of them are larger than the well-typing $app(list(T), stream(T), stream(T))$. For the former, ρ is the identity mapping and $\theta = \{T/list(T)\}$; for the latter, $\rho(list(T)) = list(T)$ and $\rho(stream(T)) = list(T)$ and $\theta = \{T/list(T)\}$.

Each type judgement gives rise to a least set of constraints from which a well-typing can be derived. Adding more constraints results in a larger well-typing.

Theorem 1. *There exists a* least *well-typing under each of the three type judgements introduced above.*

Observe that any well-typing satisfying the MonoCall rule also satisfies the PolyCall rule; furthermore any well-typing satisfying the latter also satisfies the SCCCall rule, hence also the following holds.

Theorem 2. *Let T_{Mono}, T_{Poly} and T_{SCC} be the least well-typings of a program P obtained with respectively the monomorphic, the polymorphic and the SCC-based analysis. Then $T_{SCC} \preceq T_{Poly} \preceq T_{Mono}$.*

Example 7. : The following program's three well-typings are all distinct:

```
r(f(X)).                p(Y) :- r(Y).            q(Z) :- r(Z).
                        p(a).                    q(f(b)).


% Monomorphic analysis          | % Polymorphic analysis
:- type t1 ---> a ; f(t2).      | :- type t3(T) ---> a ; f(T).
:- type t2 ---> b.              | :- type t4     ---> b.
:- pred r(t1).                  | :- pred r(t3(T)).
```

```
:- pred p(t1).                          | :- pred p(t3(T)).
:- pred q(t1).                          | :- pred q(t3(t4)).
--------------------------------------------------------------------

% SCC-based analysis
:- type t5(T) ---> f(T).                   :- pred r(t5(T)).
:- type t6(T) ---> a ; f(T).               :- pred p(t6(T)).
:- type t7    ---> f(t8).                   :- pred q(t7).
:- type t8    ---> b.
```

$$
\begin{array}{ccccc}
r(t5(T)) & \prec & r(t3(T)) & \prec & r(t1) \\
p(t6(T)) & = & p(t3(T)) & \prec & p(t1) \\
q(t7) & \prec & q(t3(t4)) & = & q(t1) \\
\hline
T_{SCC} & \prec & T_{Poly} & \prec & T_{Mono}
\end{array}
$$

3 The Monomorphic Type Analysis

The monomorphic type system is simple. It requires that all calls to a predicate have exactly the same typing as the signature (rule MONOCALL in Figure 2).

The monomorphic type inference (first described in [1]) consists of three phases: (1) Derive *type constraints* from the program text. (2) Normalize (or solve) the type constraints. (3) Extract *type definitions* and *type signatures* from the normalized constraints. A practical implementation may interleave these phases. In particular, (1) may be interleaved with (2) via incremental constraint solving. We discuss the three phases in more detail below.

Phase 1: Type Constraint Derivation For the purpose of constraint derivation we assume that a distinct type τ is associated with every occurrence of a term. In addition, every defined predicate p has an associated type signature $p(\overline{\tau})$ and every variable X an associated type τ; these are respectively denoted as $pred(p(\overline{\tau}))$ and $var(X) : \tau$. The associated type information serves as the initial assumption for the type environment E; initially all types are unconstrained.

Now we impose constraints on these types based on the program text. For the monomorphic system, we only need two different kinds of type constraint: $\tau_1 = \tau_2$: the two types are (syntactically) equal, and $\tau \supseteq f(\overline{\tau})$: the type definition of type τ contains a case $f(\overline{\tau})$.

Figure 3 shows what constraints are derived from various language constructs. The unlisted language constructs do not impose any constraints.

Phase 2: Type Constraint Normalization In the constraint normalization phase we rewrite the bag of derived constraints (the constraint store) to propagate all available information. The normalized form is obtained as the fixed point of just three rewrite steps. The first rewrite step drops trivial equalities.

$$(\textbf{Triv})\ C \cup \{\tau = \tau\} \Longrightarrow C$$

$$(\text{Var}) \quad \frac{X : \tau' \qquad var(X) : \tau}{\tau = \tau'}$$

$$(\text{Term}) \quad \frac{t_1 : \tau_1 \qquad \dots \qquad t_n : \tau_n \qquad f(t_1, \dots, t_n) : \tau}{\tau \supseteq f(\tau_1, \dots, \tau_n)}$$

$$(\text{Call}) \quad \frac{t_i : \tau_i' \qquad pred(p(\tau_1, \dots, \tau_n)) \\ (a \;\text{:-}\; g) \in P \qquad p(t_1, \dots, t_n) \in g}{\tau_i' = \tau_i}$$

$$(\text{Unif}) \quad \frac{t_1 : \tau_1 \qquad t_2 : \tau_2 \qquad t_1 \text{=} t_2 \in P}{\tau_1 = \tau_2}$$

$$(\text{Head}) \quad \frac{t_1 : \tau_1' \qquad \dots \qquad t_n : \tau_n' \qquad pred(p(\tau_1, \dots, \tau_n)) \qquad p(t_1, \dots, t_n)\text{:-}g \in P}{\tau_i' = \tau_i}$$

Fig. 3. Constraint derivation rules

The second rewrite step unifies equal types.

$$(\textbf{Unif}) \; C \cup \{\tau_1 = \tau_2\} \Longrightarrow C[\tau_2/\tau_1] \cup \{\tau_1 = \tau_2\}$$

where $\tau_1 \in C$ and $\tau_1 \not\equiv \tau_2$. The third step collapses equal function symbols.

$$(\textbf{Coll}) \; C \cup \{\tau \supseteq f(\overline{\tau_1}), \tau \supseteq f(\overline{\tau_2})\} \Longrightarrow C \cup \{\tau \supseteq f(\overline{\tau_1}), \overline{\tau_1} = \overline{\tau_2}\}$$

Phase 3: Type Information Extraction Type rules and type expressions are derived simultaneously from the normal form of the constraint store:

- A type α that does not appear as the first argument in a $\supseteq$ constraint gets as its type expression a unique type variable A.
- A type τ that appears on the lhs of one or more $\supseteq$ constraint is assigned a type expression $t(\dots)$ with t a unique type name and gives rise to a type rule of the form $t(\dots) \longrightarrow \dots$. The type expression $t(\dots)$ has as its arguments the type variables A_i occurring in the right hand side of the rule. For each constraint $\tau \supseteq f(\tau_1, \dots, \tau_k)$, the right hand side has a case $f(exp_1, \dots exp_k)$ where each exp_i is the type expression assigned to τ_i.
- A type τ_2 occurring in the right hand side of an equation $\tau_1 = \tau_2$ is assigned the type expression assigned to τ_1.

Theorem 3. *The monomorphic type inference terminates and infers a least well-typing.*

Indeed, the constraints created in Phase 1 are the minimal set of constraints to be satisfied by the well-typing; the first two rewriting steps in Phase 2 preserve the constraints, while the last one follows from the requirement that the right hand side of a type rule has distinct functor symbols and Phase 3 extracts the minimal types satisfying all constraints. Moreover, each phase terminates, hence the algorithm terminates and produces a minimal well-typing for the program.

Of particular interest is the time complexity of normalization:

Theorem 4 (Time Complexity). *The normalization algorithm has a near-linear time complexity $\mathcal{O}(n \cdot \alpha(n))$, where n is the program size and α is the inverse Ackermann function.*

The $\alpha(n)$ factor follows from the **Unif** step, if implemented with the optimal union-find algorithm.

4   The Polymorphic Type Analysis

The polymorphic type system relaxes the monomorphic one. The type signature of non-recursive predicate calls is an instance of the predicate signature, rather than being identical to it. In [5] a type inference is described that allows this also for recursive calls. It is straightforward to adapt that approach to the current setting where recursive calls have the same signature as the predicate (only the constraint derivation phase needs to be altered). It has a complex set of rules because there is propagation of type information between predicate signature and call signature in both directions. A new case in a type rule from a type in the signature of a call is propagated to the corresponding type in the signature of the definition and in turn propagated to the corresponding types in the signature of all other calls. There, it can potentially interact with other constraints, leading to yet another case and triggering a new round of propagation. When the signature of recursive calls is allowed to be an instance of the predicate signature, it is unclear whether termination is guaranteed. Indeed, Henglein [2] showed that type checking in a similar setting is undecidable while we have strong indications [5] that type inferencing in the setting of that paper is also undecidable. The restriction on the signatures of recursive types guarantees termination but complexity remains high. Experiments indicate a cubic time complexity.

5   The SCC Type Analysis

The SCC type analysis reconstructs type information according to the SCCCALL rule. We first present a simple, understandable, but rather naive approach for the analysis in Section 5.1, and subsequently refine it in Sections 5.2 and 5.3.

5.1   Naive Approach

A very simple way to implement an analysis of a program P for the SCCCALL rule is to apply the monomorphic analysis of Section 3 to a *transitive multi-variant* specialization (TMVS) of P.

Definition 5 (Transitive Multi-Variant Specialization). *We say that P' is a transitive multi-variant specialization of P if:*

- *There is at most one call to each predicate $p/n \in P'$ from outside the predicate's strongly connected component.*

11

– *There is an inductive equality relation between predicates of P and P' that implies structural equality.*

Effectively, we obtain a TMVS of P by creating named apart copies of each predicate definition, one for each call to the predicate.

Example 8. For Example 1, we obtain:

```
p(R) :- app1([a],[b],M), app2([M],[M],R).

app1([],L,L).    app1([X|Xs],Ys,[X|Zs]) :- app2(Xs,Ys,Zs).
app2([],L,L).    app2([X|Xs],Ys,[X|Zs]) :- app2(Xs,Ys,Zs).
app([],L,L).     app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).
```

Applying the monomorphic analysis, we obtain the type information as shown in Example 4 (for the app/3 definition) and in Example 5 (for predicate p/1).

While this approach is simple, it is also highly inefficient. Observe that the size of a TMVS is worst-case exponential in n, the size of the original program. Hence, this approach has worst-case $\mathcal{O}(e^n \cdot \alpha(e^n))$ time complexity.

Example 9. The following program's TMVS exhibits the exponential blow-up:

```
p_0(L) :- L = [a].
p_1(L) :- L = [A], p_0(A), p_0(L).
...
p_n(L) :- L = [A], p_{n-1}(A), p_{n-1}(L).
```

In the TMVS of this program, there are two copies of $p_{n-1}/1$, four copies of $p_{n-2}, \ldots$ and 2^n copies of $p_0/1$.

Below, we will make the analysis more efficient, but the worst case complexity remains exponential. However, in practice (see Section 6) the analysis turns out to be much better behaved than the polymorphic analysis.

5.2 Derive and Normalize Once

There is a clear inefficiency in the naive approach: the same work is being done repeatedly. Multiple copies of the same predicate definition are created, the same set of constraints is derived from each copy, and each copy of those constraints is normalized. This consideration leads us to propose a refined approach:

1. We analyze each predicate definition only once with Phases 1 and 2 of the monomorphic analysis.
2. We reuse the results for a whole *subprogram* multiple times.

In order to make good on the above two promises, we must traverse the program in a specific manner: SCC-by-SCC in a bottom-up fashion. In other words, we first visit the predicates that do not depend on any other predicates. Then we visit the predicates that only depend on the previous ones, and so on.

SCC Traversal The strongly connected components (SCCs) of a program are sets of predicates, where each component is either a singleton containing a non-recursive predicate or a maximal set of mutually recursive predicates. There is a partial order on the SCCs; for components s_1 and s_2, $s_1 \preceq s_2$ iff some predicate in s_1 depends (possibly indirectly) on a predicate in s_2. Hence, the refined analysis first computes the SCCs and topologically sorts them in ascending ordering wrt $\preceq$, yielding say $s_0, \ldots, s_m$. The SCCs are processed in that order. The SCCs of a program can be computed in $O(|P|)$ time [7].

Example 10. This means for Example 1 that we first visit `app/3` and then `p/1`: $s_0 = \{\texttt{app/3}\}$ and $s_1 = \{\texttt{p/1}\}$. For the program of Example 9 with exponential-size TMVS, the predicate definitions of $p_i/1$ are visited in the order of increasing values of i: $s_i = \{p_i/1\}$.

SCC Processing For each SCC s_i we generate a set of constraints and, after normalisation, we derive a signature for each predicate p/n in s_i. The set of type constraints for the clauses of s_i are generated using the same rules as Phase 1 of the monomorphic analysis (see Figure 3). There is one exception: the *Call* rule is modified for non-recursive calls to NONRECCALL.

Let $p(t_1, \ldots, t_n)$ be such a call to a predicate in SCC s_j, with $s_j \prec s_i$. Assume $\tau_1, \ldots, \tau_n$ are the argument types for the predicate signature in s_j and $\tau'_1, \ldots, \tau'_n$ the types of the arguments $t_1, \ldots, t_n$ in the call. Instead of generating the equalities $\tau'_i = \tau_i$ as in Figure 3, one uses a renaming ρ to give new names $\sigma_1, \ldots, \sigma_n$ to the types in $\tau_1, \ldots, \tau_n$ as well as new names to all types in the whole set C_j of constraints in s_j. The generated constraint then consists of the renamed set of constraints $C_j\rho$ together with the equalities $\sigma_i = \tau'_i$ as described in the rule of Fig. 4.

$$
(\textsc{NonRecCall}) \quad \frac{
\begin{array}{cc}
p/n \in s_j & pred(p(\tau_1, \ldots, \tau_n)) \\
\tau_i \rho = \sigma_i & t_i : \tau'_i \\
(a \texttt{ :- } g) \in P & p(t_1, \ldots, t_n) \in g
\end{array}
}{
C_j\rho \wedge \bigwedge_{i=1}^{n} \sigma_i = \tau'_i
}
$$

Fig. 4. Constraint derivation rule for non-recursive calls

Observe that, because of the copying of constraints, each call to a predicate in a lower SCC has its own type which does not interfere with calls to the same predicate elsewhere (interference is unavoidable in the polymorphic analysis).

Second, we normalize the derived constraints with the same rules as in Phase 2 of the monomorphic analysis. We call the resulting normalized constraint set C_i, and remember them for use in the subsequent SCCs.

Finally, we use the algorithm of Phase 3 of the monomorphic analysis to generate signatures for the predicates in s_i, based on the constraints C_i.

13

Example 11. Assume that for the program of Example 9 the constraint generation for $p_0/1$ yields for the signature $pred(p_0(\tau))$

$$C_0 \equiv \tau \supseteq [\tau_1 \,|\, \tau_2] \wedge \tau_1 \supseteq \mathtt{a} \wedge \tau_2 \supseteq \mathtt{[]}$$

Then for the analysis of p_1, we make two copies of C_0, one for the call $\mathtt{p_0(A)}$ and one for the call $\mathtt{p_0(L)}$. For instance, for the former call, we derive the constraints

$$\sigma \supseteq [\sigma_1 \,|\, \sigma_2] \wedge \sigma_1 \supseteq \mathtt{a} \wedge \sigma_2 \supseteq \mathtt{[]} \wedge \tau' = \sigma$$

where the type of $\mathtt{A}$ is τ' and the substitution $\rho = \{\sigma/\tau, \sigma_1/\tau_1, \sigma_2/\tau_2\}$.

Efficiency Improvements Although the actual worst-cased complexity is not improved, the actual efficiency is. Not only does this SCC-based approach avoid actually generating the TMVS of a program, the normalized set of constraints C_i is usually smaller than the originally generated set. Hence, the repeated copying of the normalized form is usually cheaper than repeated normalization.

We expect that an approach that shares rather than copies the constraints C_j of the lower SCC s_j would be even more efficient. It looks plausible to keep the set of constraints linear in the size of the program. However, the final extraction phase could in the worst case still give an exponential blow up (when each call has a signature that is an instance of a distinct polynomial predicate signature). We intend to explore this further in future work.

5.3 Projection

One can observe that there is no need to copy the whole constraint system C_j of the SCC s_j when processing a call to that SCC. For processing a non-recursive call to $\mathtt{p/n}$, it suffices to copy a projection $C_j^{p/n}$: the part of C_j that defines the types in the signature of p/n. This will usually be smaller than C_j.

Example 12. Consider the program with SCCs $s_1 = \{\mathtt{r/2}\}$, $s_2 = \{\mathtt{q/2}\}$ and $s_3 = \{\mathtt{p/2}\}$:

```
p(X) :- q(X).
q(Y) :- r(Y,Z).
r(a,b).
```

We have $C_2 \equiv (\tau_Y \supseteq \mathtt{a} \wedge \tau_Z \supseteq \mathtt{b})$ and signature $pred(\mathtt{q}(\tau_Y))$. However, if we project on the signature, then the second constraint of C_2 is dropped: $C_2^{\mathtt{q/1}} \equiv \tau_Y \supseteq \mathtt{a}$. In other words, for the call in s_3, we must copy half as many constraints with $C_2^{\mathtt{q/1}}$ than with C_2.

6   Evaluation

We evaluated the three algorithms on a suite of 45 small programs also used in [1]. Figure 5 displays the relative runtime (in % on a logarithmic scale) of both the polymorphic and SCC-based analyses with respect to the monomorphic analysis.
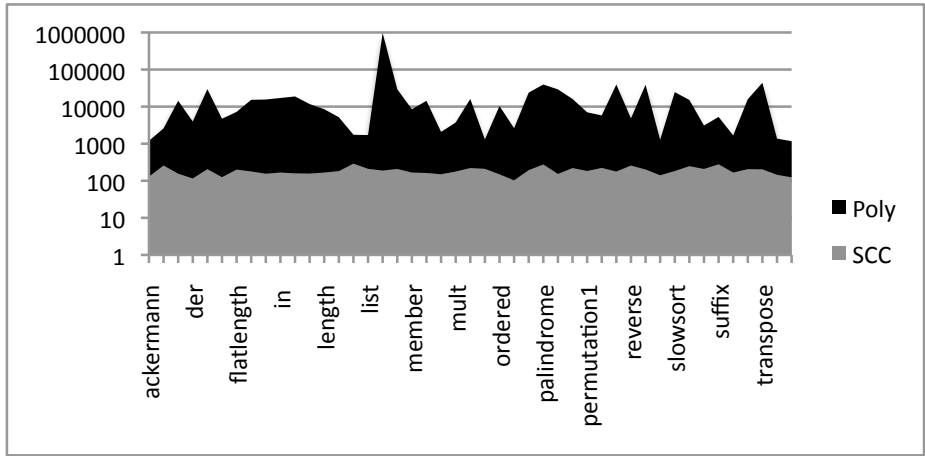
Fig. 5. Relative runtime (in %) wrt the monomorphic analysis

The monomorphic analysis finishes quickly, in less than 1 ms on a Pentium 4 2.00 GHz. The SCC analysis provides more accurate analysis in 1 to 3 times the time of the monomorphic analysis. The complex polymorphic analysis lags far behind; it is easily 10 to 100 times slower.

A contrived scalable benchmark based on Example 1 shows that the monomorphic and SCC analyses can scale linearly, while the polymorphic analysis exhibits a cubic behavior. The scalable program (parameter n) is constructed as follows:

```
app([],L,L).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).

r(R) :- app([a],[b],M1),
        app([M1],[M1],M2), ..., app([Mn],[Mn],R).
```

Below are the runtimes for the three inferences.[3]

Program	MONO	SCC	POLY
app-1	0.38 ms	0.65 ms	12 ms
app-10	1.20 ms	2.14 ms	206 ms
app-100	9.60 ms	18.10 ms	88,043 ms
app-1000	115.80 ms	238.05 ms	T/O
app-10000	1,402.40 ms	2,955.95 ms	T/O

7 Conclusion and Future Work

Within the framework of polymorphic well-typings of programs, it is customary to have a unique principal type signature for predicate definitions and type signatures of calls (from outside the SCC defining the predicate) that are instances

---

[3] T/O means time-out after 2 minutes.

of the principal type. We have presented a novel SCC-based type analysis that gives up the concept of a unique principal type and instead allows different calls to have type signatures that are instances of different well-typings of the predicate definition. This offers two advantages. Firstly, it is much more efficient than a true polymorphic analysis and is only slightly more expensive than a monomorphic one. In practice, it scales linearly with program size. Secondly, when an unexpected case appears in a type rule (which may hint at a program error), it is easy to figure out whether it is due to the predicate definition or to a particular call. This information cannot be reconstructed from the inferred types in the polymorphic and the monomorphic analyses.

In future work we plan to investigate the quality of the new analysis by performing type inference on Mercury programs where all type information has been removed and comparing the inferred types with the original ones.

References

1. M. Bruynooghe, J. P. Gallagher, and W. Van Humbeeck. Inference of well-typing for logic programs with application to termination analysis. In C. Hankin and I. Siveroni, editors, *Static Analysis, SAS 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2005.
2. F. Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993.
3. T. L. Lakshman and U. S. Reddy. Typed Prolog: A semantic reconstruction of the Mycroft-O'Keefe type system. In *Logic Programming, ISLP 1991*, pages 202–217, 1991.
4. A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.
5. T. Schrijvers and M. Bruynooghe. Polymorphic algebraic data type reconstruction. In A. Bossi and M. J. Maher, editors, *Proceedings of the 8th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 10-12, 2006, Venice, Italy*, pages 85–96, 2006.
6. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.
7. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.