From monomorphic to polymorphic well-typings and beyond Extended report

Tom Schrijvers
John Gallagher
Maurice Bruynooghe

Report CW 518, June 2008



Katholieke Universiteit Leuven
Department of Computer Science
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

From monomorphic to polymorphic well-typings and beyond Extended report

Tom Schrijvers
John Gallagher
Maurice Bruynooghe

Report CW 518, June 2008

Department of Computer Science, K.U.Leuven

Abstract

Type information has many applications, it can be used for optimized compilation, termination analysis, error detection, ... However logic programs are typically untyped. A well-typed program has the property that it behaves identically 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 concept 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.

Keywords : logic programming, type reconstruction, polymorphic types.
CR Subject Classification : D.3.2, D.3.3.

From Monomorphic to Polymorphic Well-Typings and Beyond

Extended Abstract

Tom Schrijvers[1]*, John Gallagher[2], and Maurice Bruynooghe[1]

[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 be used for optimized compilation, termination analysis, error detection, ... However logic programs are typically untyped. A well-typed program has the property that it behaves identically 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 concept 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, debugging, ..., as a matter of fact, most logic programming languages are untyped. In [3], Mycroft and O'Keefe propose a polymorphic type schema 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. While there is plenty of work on automatic type inference for logic programs, it was, to the best of our knowledge, not until [1] that a method was introduced to automatically infer a well-typing for logic programs. 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.

```
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:
```

* Post-doctoral researcher of the Fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen)

```
:- type list ---> [] ; a ; b ; [list | list].
% predicate signatures:
:- app(list,list,list).
:- p(list).
```

Note that the list type 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 the former 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 likely use many different type instances of base predicates.

In a language with polymorphic types such as Mercury, [5], 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 a 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 `list`. For instance, the term `[a|b]` is of type `list`, 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 [4] 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 (for simplicity we refer to them as recursive calls) 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. Henglein [2] showed that type checking in such a setting is undecidable, and Schrijvers and Bruynooghe [4] have strong indications of similar undecidability for type inference. Applied on the above program fragment, one obtains the following well-typing:

```
:- 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)).
:- 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 appi` refers to that of the ith call. 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:

```
q(R) :- app([a],b,M), app([M],[M],R).
% type definition
:- type elem ---> a.                              % <<<
:- type list1(T) ---> [] ; [T|list1(T)] .
:- type list2(T) ---> [] ; [T|list2(T1)] ; b.    % <<<
% signatures
:- app(list1(T),list2(T),list2(T)).
:- q(list2(list2(elem))).
:- call app1(list1(elem), list2(elem), list2(elem)).
:- call app2(list1(list2(elem)),
             list2(list2(elem)), list2(list2(elem))).
```

Note that the erroneous call spoils the type of `app/3`. Indeed, the type `list2(T)` has an extra 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, together with the complexity of the polymorphic analysis motivated us to consider yet another setting where we derive types SCC by SCC. For the lowest SCC, defining `app/3`, we obtain:

```
:- 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 that the list structure is terminated by an empty list, hence this case is absent in the type for second and third argument. Note that none of the two types is an instance of the other one.

For the SCC defining `p/1` one obtains:

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

This reveals that in the SCC of `p/1`, `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:

```
:- 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. It is interesting in another aspect. It gives up the usual concept underlying polymorphic typing that each predicate should have a unique principal typing and that all calls should have a type that is an instance of it. Indeed, the types of the first and second call 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 the following:

- We propose a new and efficient polymorphic type analysis based on an SCC by SCC traversal of the program.
- We compare our approach with two other analyses, a cheap but inaccurate monomorphic analysis and an accurate but expensive polymorphic analysis.
- Our small evaluation shows the respective merits of the different analyses.

In the rest of this abstract, we describe the different well-typings and their inference in more detail and we end with a small evaluation of their merits.

2 Problem Statement and Background Knowledge

Logic Programs Our syntax of logic programs is defined 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 the first two sets are denoted with strings starting with a lower case, whereas elements of `Var` start with an upper case.

Types We adopt the terminology of Mercury [5] for our types. They are built from a number of type constructors $t_0, t_1, \ldots$ and type variables $\phi_0, \phi_1, \ldots$:

$$\tau := \quad \phi \;\mid\; t(\bar{\tau})$$

where $\bar{\tau}$ stands for $\tau_1, \ldots, \tau_n$ and the type constructors t are defined by a *type definition*, which is a finite set of *type rules* of the form:

$$t(\bar{\phi}) \longrightarrow f_1(\bar{\tau_1}); \ldots; f_n(\bar{\tau_n})$$

where f_i are distinct function symbols and all type variables in $\bar{\tau_i}$ also appear in $\bar{\phi}$. No two type rules have the same type constructor in the left-hand side.

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

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

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

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

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

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

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

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 of 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 other rules state the well-typing of atoms and that a program is well-typed when all its parts are.

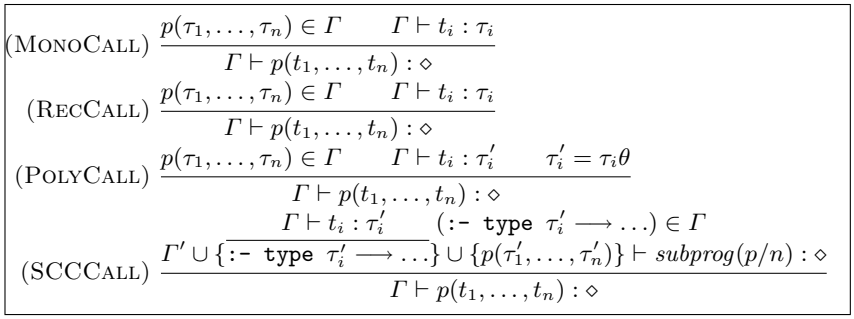

Fig. 2. The Call Rules

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 is $p(\tau'_1,\ldots,\tau'_n)$— is such that there exists a typing environment (that can be different from Γ) with the following properties: the subprogram defining the predicate ($\mathit{subprog}(p/n)$) 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 an instance of it; however, that polymorphic type can be different for different calls.

In all three analyses, we are interested in *minimal* solutions. Informally: fewer cases in a type rule is better and one type is better than another, when the latter is equivalent to an instance of the former.

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.

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

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

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

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

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

Fig. 3. Constraint derivation rules

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 but three rewrite steps. The first rewrite step drops trivial equalities.

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

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 rewrite 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 definitions and type expressions are derived simultaneously from the normal form of the constraint store.

We infer the type expressions from:

– 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 a $\supseteq$ constraint is assigned a unique type name t. This type name has as its arguments the type variables A_i of corresponding types α_i such that $\tau \rightsquigarrow \alpha_i$[1] are the type expressions of the τ_i.

The type definitions follow from the type expressions in a straightforward manner. For each type τ with type expression $t(\overline{A})$ we get a type definition $t(\overline{A}) \longrightarrow \ldots$. This definition contains one case $f(\ldots)$ for each constraint $\tau \supseteq f(\overline{\tau})$, where the argument expressions are the type expressions of the types $\overline{\tau}$.

Properties It is fairly easy to see that by the above approach we get a *sound*, *complete* and *terminating* algorithm.

Of particular interest is the time complexity of normalization:

Theorem 1 (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 [4], a type inference is described that allows this also for recursive calls. constraint derivation to the current setting). 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. Experiments indicate a cubic time complexity.

5 The SCC Type Analysis

We summarize briefly the procedure for the SCC-based type inference. 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.

[1] I.e., α_i is reachable from τ using the $\supseteq$ constraints.

Basic Analysis In the SCC type analysis we first compute the SCCs and topologically sort them in ascending ordering wrt $\preceq$, yielding say $s_0, \ldots, s_m$. Type constraints for the clauses for s_i are then generated using the same rules as in Figure 3, except that in the *Call* rule applied to non-recursive calls we use a renamed copy of the signature of the called predicate, and extend the solved type constraints for $s_0, \ldots, s_{i-1}$ by a renamed version of the type rules for the types in the renamed signature. Thus 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 which is unavoidable in the polymorphic analysis).

The complexity of the SCC type generation and solution is now related to the number of generated constraints including the renamed copies. This appears to be roughly quadratic in the size of the program, since the number of copies is proportional to $m * k * (k+1)/2$ where k is the number of SCCs and m is the number of calls.

The above description requires that all (solved) type constraints of the lower SCCs $s_0, \ldots, s_{i-1}$ are copied for a non-recursive call in SCC s_i. In practice, it is not necessary to copy all these constraints, only those constraints relevant for the predicate definition. In other words, we may project all constraints on the variables in the predicate signature, before copying. Usually projection yields a much smaller set of constraints.

6 Evaluation

We evaluated the three algorithms on a suite of small programs (see Appendix A), also used in [1]. 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's easily 10 to 100 times slower.

A contrived scalable benchmark based on the first example in this paper shows that the monomorphic and SCC analyses can scale linearly, while the polymorphic analysis exhibits a cubic behavior. The scalable program, which varies with 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[2].

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

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

Another scalable benchmark (Appendix B) provokes the worst-case quadratic behavior of the SCC analysis, which is still much better than the cubic behavior and constant factors of the polymorphic analysis.

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 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. A. Mycroft and R. A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23(3):295–307, 1984.
4. 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.
5. 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.

A Benchmark Suite

The following table shows the runtimes for the three type inference algorithms for 45 small logic programs, ranging from 4 tot 44 lines of code. The runtimes for the monomorphic analysis (MONO) are given in ms. Those of the SCC-based approach (SCC) and the polymorphic analysis (POLY) are relative to the monomorphic analysis.

Program	MONO	SCC	POLY
ackermann	0.32	128.93 %	1066.04 %
append	0.12	258.33 %	2325.00 %
delete	0.78	155.13 %	14230.77 %
der	1.56	115.38 %	3846.15 %
factor	0.44	205.88 %	29411.76 %
flat	0.24	125.00 %	4583.33 %
flatlength	0.30	200.00 %	7000.00 %
frontier	0.40	178.39 %	15075.38 %
g	0.78	155.13 %	15384.62 %
in	0.42	166.67 %	16904.76 %
inorder	0.32	158.39 %	18633.54 %
insert	0.52	156.37 %	11583.01 %
length	0.12	166.67 %	8333.33 %
length1	0.22	181.82 %	5000.00 %
less	0.14	289.86 %	1449.28 %
list	0.10	210.00 %	1500.00 %
mapcolor	1.70	188.60 %	969447.71 %
maximum	0.24	208.33 %	29166.67 %
member	0.12	166.67 %	8333.33 %
mergesort	0.98	162.24 %	14183.67 %
minimum	0.14	150.00 %	1928.57 %
mult	0.28	177.30 %	3546.10 %
naivereverse	0.32	220.50 %	15838.51 %
numeral	0.10	210.00 %	1090.00 %
ordered	0.40	149.25 %	10199.00 %
p	0.40	103.02 %	2512.56 %
pnonlin	0.72	193.59 %	23676.88 %
palindrome	0.26	275.19 %	39147.29 %
parse	1.24	152.98 %	28985.51 %
permutation	0.32	220.50 %	15838.51 %
permutation1	0.28	183.45 %	6834.53 %
prefix	0.18	222.22 %	5555.56 %
quicksort	0.84	177.38 %	39285.71 %
reverse	0.20	257.58 %	4545.45 %
searchtree	0.70	201.43 %	38571.43 %
select	0.14	140.85 %	1133.80 %
slowsort	0.82	182.93 %	24390.24 %
sublist	0.20	247.52 %	14851.49 %
subset	0.24	208.33 %	2875.00 %
suffix	0.18	277.78 %	5000.00 %
sum	0.12	166.67 %	1500.00 %
t	0.44	205.88 %	15610.86 %
transpose	0.78	203.85 %	43717.95 %
tree	0.14	144.93 %	1224.64 %
treemember	0.16	123.46 %	1043.21 %

B Scaling List Benchmark

The following contrived benchmark exhibits the worst-case quadratic behavior of the SCC analysis. Projection is ineffective, and each SCC copies all constraints from the previous SCC.

```
list([]).
list([X|Xs]) :- list(Xs).
```

```
p0(L) :-        L = [a], list(L).
p1(L) :- p0(A), L = [A], list(L).
...
pn(L) :- pn−1(A), L = [A], list(L).
```

The figure below sets out the runtime for values of n in the range 1-100.

