A Practical Type Analysis for Verification of Modular Prolog Programs

Paweł Pietrzak
School of Computer Science, Technical University of Madrid, Spain
pawel@fi.upm.es

Jesús Correas
School of Computer Science, Complutense University of Madrid, Spain
jcorreas@fdi.ucm.es

Germán Puebla
School of Computer Science, Technical University of Madrid, Spain
german@fi.upm.es

Manuel V. Hermenegildo
School of C.S., T.U. Madrid, IMDEA-Software Institute, Spain
Depts. of CS and ECE, Univ. of New Mexico
herme@fi.upm.es

Abstract

Regular types are a powerful tool for computing very precise descriptive types for logic programs. However, in the context of real-life, modular Prolog programs, the accurate results obtained by regular types often come at the price of efficiency. In this paper we propose a combination of techniques aimed at improving analysis efficiency in this context. As a first technique we allow optionally reducing the accuracy of inferred types by using only the types defined by the user or present in the libraries. We claim that, for the purpose of verifying type signatures given in the form of assertions the precision obtained using this approach is sufficient, and show that analysis times can be reduced significantly. Our second technique is aimed at dealing with situations where we would like to limit the amount of reanalysis performed, especially for library modules. Borrowing some ideas from polymorphic type systems, we show how to solve the problem by admitting parameters in type specifications. This allows us to compose new call patterns with some precomputed analysis info without losing any information. We argue that together these two techniques contribute to the practical and scalable analysis and verification of types in Prolog programs.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Assertion checkers; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Assertions; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis; D.2.5 [*Software Engineering*]: Testing and Debugging—Debugging aids, Diagnostics, Symbolic execution; D.3.2 [*Software Engineering*]: Language Classifications—Constraint and logic languages

General Terms Languages, Verification

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profi t or commercial advantage and that copies bear this notice and the full citation on the fi rst page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specifi c permission and/or a fee.
PEPM'08, January 7–8, 2008, San Francisco, California, USA.
Copyright © 2008 ACM 978-1-59593-977-7/08/0001. . . $5.00

Keywords Program Analysis, Abstract Interpretation, Types, Verification, Modular Logic Programs, Logic Programming, Scalability

1. Introduction

Types are widely recognized as being useful for several purposes, which include early detection, i.e., at compile-time, of certain programming errors, enforcement of disciplined programming, and documentation of code. In the terminology of (Pierce 2002), Pure Logic Programming is a *safe* programming language in that the semantics of programs is well-defined and the execution of a program does not depend on the particular compiler used. This is achieved, without the need for types, thanks to the declarative nature of Pure Logic Programming, which is *untyped*. However, as soon as we introduce predefined operations in the programming language, for example arithmetic, certain type checks are required in order to preserve the safety of the programming language. As a result, Prolog, which is the most widely used logic programming language, is no longer an untyped programming language, but rather it is a *dynamically checked* typed language: In order to preserve language safety, calls to predefined operations which do not satisfy their calling conventions result in run-time errors or exceptions. However, some of the desirable features of types mentioned above in fact only apply to *statically checked* typed languages.

A clear possibility in order to obtain a statically checked typed logic programming language is to design a new programming language from scratch with static checking in mind. Two proposals along these lines are Gödel (Hill and Lloyd 1994) and Mercury (Somogyi et al. 1996). In these languages, types (called *prescriptive types*) are a part of the language itself (both the syntax and semantics). In spite of the undoubted contribution of these proposals, Gödel is no longer maintained and Mercury, while certainly interesting in many ways, deviates in a number of respects from logic programming. In practice, Prolog (including its different extensions) remains by far the most widely used logic programming language.

Another possibility, which is the one we will focus on in this work, is to provide a mechanism for performing static checking of types directly for Prolog. We believe that this will have more practical impact than designing yet another strongly typed logic programming language. Several proposals have been made and implemented in the direction of augmenting Prolog with static checking, such as, e.g., Ciao (Hermenegildo et al. 2005; Bueno et al. 2006). In this context, even though the language itself is not statically typed, it is possible to infer static information about the program in terms of *regular types*. The types inferred are called *descriptive types* in that they describe the program behavior, but they do not provide *directly* any assurance about the nonexistence of run-time errors since we can obtain descriptive types for any program. In Ciao, users can *optionally* provide type definitions and assign types to predicate arguments and thus describe the expected behavior of the program (Hermenegildo et al. 2005). Success of static checking occurs if the descriptive types inferred imply the type declarations provided by the user and those present in system libraries. Alternatively, type-related errors may be detected statically.

Abstract interpretation-based type analysis using regular types is a powerful technique for computing very precise descriptive types for logic programs in general and for Prolog in particular. In Ciao, a multi-variant, context sensitive analysis engine (Hermenegildo et al. 2000) is used which is parametric w.r.t. the abstract domain of interest and which can analyze (Correas et al. 2006; Puebla et al. 2004) and check (Pietrzak et al. 2006) modular programs. Unfortunately, in this setting, the analysis of real-life, modular Prolog programs, using regular types turns out to be too expensive in practice. The abstract domain of regular types is infinite and in order to guarantee termination of the analysis process a widening operator is required. Such operators may, in some cases, be quite sophisticated procedures (cf. (Mildner 1999)). It is this ability of the widening-based analyses to create new types that brings the precise results, but at the same time the presence of a large number of very detailed types inevitably affects analysis performance.

In this paper we propose a combination of techniques aimed at improving analysis efficiency in this context while preserving a reasonable accuracy. The techniques proposed are implemented as extensions of the generic analyzer in the Ciao Preprocessor, CiaoPP (Hermenegildo et al. 2000, 2005), with the type domains of (Janssens and Bruynooghe 1992; Vaucheret and Bueno 2002). As a first technique we allow *optionally* reducing the accuracy of inferred types by using only the types defined by the user or present in the libraries. In every iteration our analysis replaces the inferred types with such types. We will show that in this way we ensure faster convergence to the fixed point and that analysis times can indeed be reduced significantly. Also, we claim that, for the purpose of verifying type signatures given in the form of assertions, the precision obtained using this approach is adequate.

Our second technique is aimed at dealing with situations where we would like to limit the amount of reanalysis performed for library modules while increasing precision. To this end we allow using parametric type assertions in the specification. Such assertions are specially useful in libraries implementing generic data manipulation predicates (like, e.g., lists or AVL-trees) which we do not want to have to reanalyze every time we analyze a program that uses the library. In this case we can instantiate parameters in the trusted assertion (now playing the role of module interface) according to the actual call pattern, and simply reuse the resulting success pattern without analyzing the library module. In this way we incorporate some specific characteristics of polymorphic type systems for logic programming (Mycroft and O'Keefe 1984; Hill and Lloyd 1994), without changing the source language and while remaining in descriptive types, i.e., types which describe approximations of the program semantics.

The main application of our analysis is in verification of programs with respect to a partial specification written in the form of a number of type assertions and type definitions (see (Puebla et al. 2000; Pietrzak et al. 2006)). Note that any assertion present in the program must refer to types which *are* defined in user or library modules anyway, and therefore in this case we may not lose many opportunities for verifying assertions.

2. Related work

The issue of introducing type systems for static checking of types in logic programming dates back to the early papers of Mishra (Mishra 1984) and Mycroft–O'Keefe (Mycroft and O'Keefe 1984). There has been a number of proposals since then for providing adequate notions of types and typing (see for example (Pfenning 1992)).

As mentioned before, our work follows the *descriptive typing* approach in which types approximate the program semantics. This idea was first presented in (Mishra 1984), where types are described by *regular term grammars* that reappeared in the literature in one formalism or another.

Deriving descriptive types from a program (this process is also called "type inference" or "type analysis"), essentially means finding, at compile-time, an approximate description (in our context, a safe approximation) of the values that program variables can take at run-time. Descriptive types can be inferred that approximate various semantics of a logic program. In (Heintze and Jaffar 1990) descriptive types are computed using *set constraints* analysis. Their types approximate the declarative semantics of programs. Another approach to approximate the declarative semantics is to construct an abstract counterpart of the *immediate consequences operator*, T_P, (see e.g. (Lloyd 1987)) in order to obtain a superset of the success set of the program. An example of this approach is (Gallagher and de Waal 1994), in which regular descriptive types are called "regular approximations"). Also, the T_P operator is approximated in (Yardeni and Shapiro 1990), but with the goal of verifying a program w.r.t. given success types, rather than inferring the types. The relative power of different regular approximations of T_P is discussed in (Heintze and Jaffar 1992).

In other approaches, descriptive types approximate operational semantics, following a top-down execution strategy with the Prolog selection rule. This allows distinguishing call and success types, which makes it feasible to verify call patterns for certain predicates. Examples of this line are (Janssens and Bruynooghe 1992; Van Hentenryck et al. 1995; Vaucheret and Bueno 2002). In our approach we also deal with operational semantics. As already mentioned, we use a generic, context-sensitive, multi-variant analysis framework (Hermenegildo et al. 2000; Bruynooghe 1991; Muthukumar and Hermenegildo 1992; de la Banda et al. 1996) based on *abstract interpretation* (Cousot and Cousot 1977) and specialized to our type domain.

In the above mentioned approaches, types are constructed on the fly during the iterative analysis process over an abstract domain of types which is infinite. Therefore a *widening* operator is introduced, to ensure that no infinite ascending chain is generated during the fixed point computation, and thus that the computation terminates. For a comprehensive study of different widening operators see (Mildner 1999).

In our approach, also new type definitions are generated on-the-fly. However, as soon as they are generated, the analyzer tries to replace them by picking a type from a predefined collection of definitions. These definitions correspond to the types which have been defined by the user or which are present in library modules used by the program. This type replacement has to be correct –we always replace types with super-types– and accurate –we never lose more precision than strictly required. Therefore, the type definitions present in the abstract descriptions at each iteration step originate from a finite set. This guarantees termination without the need for a widening operator.

Another alternative to the widening option is to generate a type domain which is specific to a given program. An example of such an approach is (Gallagher and Puebla 2002), which proposes an abstract interpretation over non-deterministic tree automata. The authors exploit the observation, due to (Cousot and Cousot 1995), that for a particular program one may automatically build a finite domain of tree automata (or regular tree grammars), and thus make sure that the analysis (a fixed point iteration) terminates. Another approach that allows constructing program-specific type domains is proposed in (Gallagher and Henriksen 2004). The constructed domain incorporates also instantiation information and it is *condensing*, i.e., combining the result of bottom-up analysis with an initial goal pattern is as precise as the output of analyzing the program top-down in the goal-dependent fashion, for the same initial goal pattern. The fact that the domain is condensing is attractive for intermodular analysis as it enables fully compositional approach. Nevertheless, in our view, a disadvantage of these approaches w.r.t. considering the defined types is that since types are automatically generated the resulting types are not intuitive and are hard to understand.

Our approach is strongly related to other work in which types for each function symbol are defined prior to the analysis, and analysis itself infers types for predicates. In these analyses, the output shown to the user contains familiar types and, thus, it is easy to interpret. Among other papers, (Lu 1995) follows this line, and shows an analysis method that combines types with sharing and aliasing. A rather complex polymorphic analysis is presented in (Lu 1998). In contrast, as our main concern is simplicity and efficiency, we do not infer polymorphic types, even if we do make use of parametric type rules for describing module interfaces (see Section 5). In some of this work, like (Barbuti and Giacobazzi 1992), only a well-typed part of the program semantics (a success set in this case) is described by the analysis output. In this sense (Barbuti and Giacobazzi 1992) is a prescriptive typing approach. Types for function symbols are also required by (Codish and Lagoon 2000), where an elegant theory of ACI-unification (associative, commutative and idempotent) is used to infer (polymorphic) type information from the program (abstractly compiled before the analysis). The resulting domain is condensing. A technique which for given type definitions infers a combination of prescriptive and descriptive types is given in (Saglam and Gallagher 1995). In all the above work, typing rules for function symbols are given prior to the analysis. In some cases, (like for example (Codish and Lagoon 2000)) the rules are quite restrictive and require that each function symbol is of exactly one type. In our work, predefined types are used differently. There is no notion of a type signature for function symbols. Instead, during the analysis the inferred (descriptive) types are inspected and replaced by predefined types that match them as precisely as possible.

There are also some similarities between our work and strongly typed logic languages such as Gödel (Hill and Lloyd 1994) or Mercury (Somogyi et al. 1996) (whose type systems are based on (Mycroft and O'Keefe 1984)), especially as regards the usage of parametric rules. However, in both Gödel and Mercury the programmer is required to write, together with the code, the types, both for function symbols and for predicates. Moreover, subtyping is often not permitted. In contrast, in our setting writing type definitions is optional and subtyping is allowed.

The way we handle parametric type rules resembles that of (Drabent et al. 2002), where complete specification given in the form of parametric descriptive types (called therein *parametric set constraints*) was required from the programmer and used to verify a program and diagnose errors. In our approach parametric type rules have different application - they facilitate separate analysis and verification of modules in multi-modular programs.

Our work is thus unique in combining both the flexibility of descriptive typing approaches, where type definitions are optional and have clear semantics, with some features of prescriptive types, where the output of analysis is presented in terms of types known to the user, and parametric type rules are allowed.

3. Preliminaries

3.1 Static (modular) program analysis

We assume that the reader is familiar with the basic concepts of logic programming (see e.g.(Lloyd 1987)). As a technique for program analysis we use *abstract interpretation* (Cousot and Cousot 1977) in which the semantics of the program is conservatively approximated using an *abstract domain* D_α (equipped with a partial order $\sqsubseteq$) which is simpler than the actual, *concrete domain* D. Abstract values and sets of concrete values are related via a pair of monotonic mappings $\langle\alpha,\gamma\rangle$: *abstraction* $\alpha : D \rightarrow D_\alpha$, and *concretization* $\gamma : D_\alpha \rightarrow D$. Goal dependent abstract interpretation takes as input a program R and an initial call pattern[1] $P{:}\lambda$, where P is an atom, and λ is a restriction of the run-time bindings of P expressed as an abstract substitution in the abstract domain D_α. Such an abstract interpretation computes a set of triples $analysis(R, P{:}\lambda) = \{\langle P_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle P_n, \lambda_n^c, \lambda_n^s\rangle\}$. In each triple $\langle P_i, \lambda_i^c, \lambda_i^s\rangle$, P_i is an atom and λ_i^c and λ_i^s are, respectively, the abstract call and success substitutions. Let $P{:}\lambda$ be an abstract initial call pattern, and let Q be the set of concrete queries described by $P{:}\lambda$, i.e., $Q = \{P\theta \mid \theta \in \gamma(\lambda)\}$. An analysis is said to be *multivariant on calls* if more than one triple $\langle P, \lambda_1^c, \lambda_1^s\rangle$, $\ldots$, $\langle P, \lambda_n^c, \lambda_n^s\rangle$ $n \geq 0$ with $\lambda_i^c \neq \lambda_j^c$ for some i, j may be computed for the same predicate. An analysis is said to be *multivariant on successes* if more than one triple $\langle P, \lambda^c, \lambda_1^s\rangle, \ldots, \langle P, \lambda^c, \lambda_n^s\rangle$ $n \geq 0$ with $\lambda_i^s \neq \lambda_j^s$ for some i, j may be computed for the same predicate p and call substitution λ^c. Different analyses may be defined with different levels of multivariance (Muthukumar and Hermenegildo 1992; Van Hentenryck et al. 1993). In general, goal dependent, multivariant analysis algorithms are more precise than those obtained with goal independent or monovariant analyses. Many implementations of abstract interpreters are multivariant on calls. However, most of them are not multivariant on successes, mainly for efficiency reasons. The analysis tool used for this paper is the analyzer in CiaoPP, which allows both types of multivariance, but multivariance on success is switched off by default.

Throughout this paper we deal with modular programs, for which we assume a *strict* module system, i.e., a system in which modules can only communicate via their *interface*. The interface of a module contains the names of the *exported* predicates and the names of the *imported* modules. We extensively use the framework for modular program analysis presented in (Puebla et al. 2004). In summary, this framework works as follows: given the top-level module m, analysis computes an intermodular fixed point by iterating through the modules in the entire program unit (i.e., the set of modules reached from m), analyzing them one by one. When the intermodular fixed point has been reached, the analysis results for exported predicates can be found in a *Global Answer Table*, in the form of atom/call pattern/success pattern triples similar to the ones computed by non-modular analyses.

Since an intermodular fixed point computation is performed, and the analysis is context sensitive, it may often be the case that the analysis of one module requires the analysis results for call patterns of imported predicates which have not been analyzed yet, or that have been analyzed for different call patterns. In that case, the success information for that call pattern must be approximated by applying a *success policy* to decide how to approximate the

[1] We will use sets of initial calls patterns in the rest of the paper. Extending the framework is trivial.

information not yet available for the imported predicate, either over-approximating the existing results of the analysis (SP^+), or under-approximating them (SP^-) (Puebla et al. 2004).

3.2 Regular types domain

Assume a finite set $\mathcal{F}$ of ranked function symbols. Let $Term(\mathcal{F}, \mathcal{V})$ denote a set of terms built from function symbols $\mathcal{F}$ and variables $\mathcal{V}$. A *regular term grammar* is a tuple $G = \langle \mathcal{T}, \mathcal{F}, R\rangle$, where:

- $\mathcal{T}$ is a set of non-terminal symbols (constants), called here *type symbols*,
- R is a set of rules of the form $l \rightarrow r$, where $l \in \mathcal{T}$, $r = f(T_1, \ldots, T_n)$ ($f/n \in \mathcal{F}, T_i \in \mathcal{T}$).

We use the notation $t_1 \Rightarrow_G t_2$ (or $t_1 \Rightarrow t_2$ if G is clear from the context) to denote the usual derivability relation, i.e. if t_2 is obtained from t_1 by replacing t (where t is a subterm of t_1) by a term r where $t \rightarrow r \in R$. Let $\overset{*}{\Rightarrow}$ denote the transitive and reflexive closure of $\Rightarrow$.

A type symbol T defined in grammar G denotes a (regular) set of ground terms $Type_G(T) = \{t \in Term(\mathcal{F}, \emptyset) \mid T \overset{*}{\Rightarrow} t\}$. As before, we drop the subscript G if it is clear from the context. In order to describe sets of numbers or Prolog atoms, we introduce *b*ase types and corresponding base type symbols, like *int*, *num*, *atm*, etc., denoting respectively sets of integers, all numbers, Prolog atoms, etc. The base types can be seen as defined by a set of rules, fixed for a fixed signature, with constants in the right hand sides. Moreover, we introduce the “top” type symbol $\top$, s.t. $Type(\top) = Term(\mathcal{F}, \emptyset)$ (i.e. $\top$ denotes the set of all ground terms) and “bottom” type symbol $\bot$, s.t. $Type(\bot) = \emptyset$.

The regular type domain (e.g., (Dart and Zobel 1992)) is equipped with standard operations that satisfy the corresponding properties:

(inclusion $\sqsubseteq$) $T_1 \sqsubseteq T_2$ iff $Type(T_1) \subseteq Type(T_2)$[2]

(intersection $\sqcap$) $Type(T_1 \sqcap T_2) = Type(T_1) \cap Type(T_2)$

(union $\sqcup$) $Type(T_1 \sqcup T_2) \supseteq Type(T_1) \cup Type(T_2)$

Also, we use equality between type symbols $T_1 = T_2$ as a shortcut for $Type(T_1) = Type(T_2)$. Note that type union is approximate. This is due to the fact that we, as many other researchers, use *deterministic* (or *tuple distributive*) types, in which if $f(a, b) \in Type(T)$ and $f(c, d) \in Type(T)$ then also $f(a, d) \in Type(T)$ and $f(c, b) \in Type(T)$.

We use regular types as abstract domain for the analysis. An abstract substitution is then a mapping from variables to types. Let $dom(\lambda)$ denote a domain of an abstract substitution λ, and let $\lambda_{|\overline{X}}$ be a projection of λ over variables $\overline{X}$. For an abstract substitution $\lambda = \{X_1/T_1, \ldots, X_n/T_n\}$, a value of the concretization function γ is given by: $\gamma(\lambda) = \{\{X_1/t_1, \ldots, X_n/t_n\} \mid t_i \in Type(T_i), 1 \leq i \leq n\}$.

4. Type Analysis with Predefined Types

As mentioned before, our type analysis is a part of the Ciao Preprocessor, CiaoPP (Hermenegildo et al. 2005), and uses one of its analysis engines (Hermenegildo et al. 2000). Moreover, as an underlying type inference system, we use the type analysis of (Janssens and Bruynooghe 1992) and (Vaucheret and Bueno 2002) with various widenings (see (Mildner 1999) for a comprehensive study on widenings in type domains). These analyses synthesize new types out of function symbols and constants present in the program. However, as a first technique in order to speed up analysis, especially in the context of large programs, we introduce a key new feature: in our analysis, types synthesized during analysis can be *optionally* replaced by predefined types which are in the scope of the module being analyzed. These may have been written by the user in the module being processed, or imported from other user modules or from a library. The predefined types chosen as replacements are less precise or equivalent to the inferred ones, so that a safe (albeit potentially less precise) approximation of the semantics is still obtained.

Let $\mathcal{T}_0$ denote a set of predefined type symbols, including $\top$, $\bot$, and some more symbols. Every iteration of the analysis contains two steps: (1) synthesizing new types $T_1, \ldots, T_n$, as explained elsewhere (e.g., (Janssens and Bruynooghe 1992; Vaucheret and Bueno 2002)), and (2) replacing them with $\lceil T_1\rceil, \ldots, \lceil T_n\rceil$ where $\lceil . \rceil$ is a replacement operator which for $T' = \lceil T\rceil$ satisfies the following:

- it returns a predefined type, i.e., $T' \in \mathcal{T}_0$,
- it safely approximates T, i.e., $T \sqsubseteq T'$,
- and it is as precise as possible: $\nexists T'' \in \mathcal{T}_0$ s.t. $(T'' \sqsubset T' \wedge T \sqsubseteq T'')$.

Note that it is not always possible to find a unique best matching predefined type for a given synthesized type. There may be two or more types that are incompatible (or equivalent) and at the same time match a given synthesized type. As a heuristics, in the case of conflicts, we give priority to types which are defined in user modules (over those in library modules) since they are likely to look more familiar to the user. Also, types that are closer in the module hierarchy (i.e., defined in the current module or a closer module) are preferred.

In order to speed up the analysis, the $\sqsubseteq$ relation over $\mathcal{T}_0$ (let us denote it $\sqsubseteq_0$) is initially precomputed (with library and builtin types) before the analysis starts and, during the analysis of each module, incrementally complemented with types specific to that module. Thanks to this, checking the subtyping is efficient. Note however, that since $\mathcal{T}_0$ contains arbitrary types (we make no assumptions about $\mathcal{T}_0$ except that it always contains $\top$ and $\bot$) $(\mathcal{T}_0, \sqsubseteq_0, \sqcap_0, \sqcup_0)$ cannot directly serve as an abstract domain, as the following does not hold $\forall_{T_1,T_2 \in \mathcal{T}_0} Type(T_1 \sqcap T_2) = Type(T_1) \cap Type(T_2)$. For example consider $\mathcal{T}_0 = \{list, atm, \top, \bot\}$. The g.l.b. of atm and $list$ induced by $\sqsubseteq_0$ would give $\bot$, whereas properly computed $atm \sqcap list$ should contain the empty list. A remedy for this is to use the standard $\sqcap$ (like for example type intersection of (Dart and Zobel 1992)) and apply $\lceil . \rceil$ to the result, and thus to redefine $\sqcap_0$ so that $\forall_{T_1,T_2 \in \mathcal{T}_0} T_1 \sqcap_0 T_2 = \lceil T_1 \sqcap T_2\rceil$. Clearly, $\sqcup_0$ can be directly computed by traversing the graph corresponding to the $\sqsubseteq_0$ relation.

5. Using Parametric Rules and Type Assertions

In order to make any analysis which works with modular programs realistic in practice, there must exist some degree of *separate* handling of code fragments. I.e., for scalability reasons, it is not realistic to expect that all modules related to an application should be available to analysis. A clear example for this are library modules. For them, we would like to have analysis information readily available, without the need of analyzing them over and over again for each application which uses them. To this end, our second technique consists in allowing developers of libraries (and modules which can be reused) to write, besides the usual regular type definitions, *parametric type rules*. It is important to note that regular types do not include in principle parametric type rules. Therefore, analysis does not infer this kind of rules and checking them will

[2] As pointed out in (Lu 2001) this operation is incorrectly defined in (Dart and Zobel 1992). Our system uses the correct version of $\sqsubseteq$, implemented by Pedro López García, independently from (Lu 2001).

require some additional mechanism, as we propose in Section 5.3 below.

Let us introduce some notation. Let $\mathcal{TV}$ be a set of *type variables* or *parameters*. Now we admit also non-nullary symbols in $\mathcal{T}$. The notion of type symbol changes a bit, now it is a ground (parameter-free) term built of symbols from $\mathcal{T}$ (i.e. an element of $Term(\mathcal{T}, \emptyset)$). Parametric type rules have the form $l \rightarrow r$ where $l \in Term(\mathcal{T}, \mathcal{TV})$, $r = f(T_1, \ldots, T_n)$ ($f/n \in \mathcal{F}, T_i \in Term(\mathcal{T}, \mathcal{TV})$), and $vars(r) \subseteq vars(l)$.

EXAMPLE 5.1. *Consider the standard definition of a list.*

$$\begin{array}{lcl} list(\alpha) & \rightarrow & [\,] \\ list(\alpha) & \rightarrow & [\alpha \mid list(\alpha)] \end{array}$$

In our framework parametric rules have no denotation unless the parameters are instantiated to regular types by a *parameter substitution*. Let $T = t(\alpha_1, \ldots, \alpha_n)$ ($t/n \in \mathcal{T}$) where $\alpha_1, \ldots, \alpha_n$ are parameters. Then, the parameter substitution Ψ is a mapping $\{\alpha_1 \mapsto T_1, \ldots, \alpha_n \mapsto T_n\}$ where $T_1, \ldots, T_n$ are regular type symbols. Applying Ψ to T, written $\Psi(T)$, means replacing any occurrence of α_i in rules defining T, by T_i. After that, the l.h.s. of the rules become a type symbol (likewise all symbols in the r.h.s.), and thus parametric rules becomes parameter-free grammar rules, as presented in Section 3.2, and can be added to the type grammar.

Naturally, a parametric rule can be instantiated multiple times with different types, and resulting in different types, e.g., $list(int)$ and $list(atm)$. Note also that type $list(\bot)$ denotes an empty list.

The process of replacing synthesized types by predefined ones also takes parametric rules into account. During analysis, types constructed by instantiating parametric rules are added to the set $\mathcal{T}_0$ of predefined types. The new instances are created on the fly, by generating type substitutions $\Psi : \mathcal{T} \mapsto \mathcal{T}_0$, such that for a synthesized type T and a parametric type symbol T_p, the instance of T_p can serve as a good approximation of T, i.e., $\Psi(T_p) = \lceil T \rceil$.

EXAMPLE 5.2. *Assume that at some intermediate step of the analysis the following abstract substitution is generated $\lambda = \{X/T\}$, where* Type$(T) = \{[a]\}$*, i.e. T denotes a one-element list of a's. Assume also that the definition of list (see Example 5.1) is present in the system. Since the constant a is described by the built-in type atm the analyzer would generate the parameter substitution $\Psi = \{\alpha \mapsto atm\}$ and finally would replace T by $\lceil T \rceil = \Psi(list(\alpha))$.*

Obviously, an abstract domain constructed as described above contains an infinite number of types, e.g., $list(num)$, $list(list(num))$, $list(list(list(num)))$, …etc. Similarly to (Barbuti and Giacobazzi 1992), we restrict the maximum depth of terms in parametric type symbols to an arbitrary number. Type symbols that occur below the maximum depth are simply replaced by $\top$. Our experiments show that depth value 3 is seldom exceeded in many programs and thus, in practice, no precision is usually lost in this step.

5.1 Type assertions

Information about intended or inferred call and success patterns is given in the form of *assertions* (Puebla et al. 2000). In this paper we limit ourselves (without loss of generality) to just one form of assertion, "pred" assertions, written (in simplified form) as `pred` $P : Pre \Rightarrow Post$. *P* is a *predicate descriptor*, i.e., it has a predicate symbol as main functor and all arguments are distinct free variables, and *Pre* and *Post* are pre- and post-conditions respectively. For our purposes it is sufficient to consider that *Pre* and *Post* correspond to abstract substitutions (λ_{Pre} and λ_{Post} resp.) over variables of P.

The meaning of `pred` assertions is twofold. First, the precondition *Pre* expresses properties which should hold in calls to *P*. Second, the postcondition *Post* expresses properties which should hold on termination of a successful computation of *P*, provided that *Pre* holds on call. Types are by default understood in "instantiation" mode (Puebla et al. 2000), i.e., `=> list(L)` implies that at procedure output `L` is *instantiated* to a list[3], and `=> list(L,T)` implies that at procedure output `L` is instantiated to a list whose elements are of type `T`. Note that type expressions in assertions differ from type expressions as defined in previous sections. Their first argument is a variable whose type is described by the expression (this first argument should not be confused with a type parameter). There can be more than one `pred` assertion per predicate, each one describing a different usage of the predicate, for example:

```
:- pred length(L,N) : (var(L), int(N))  => list(L).
:- pred length(L,N) : (var(N), list(L)) => int(N).
```

In this case the *union* (disjunction) of the *Pre* parts expresses the properties which should hold in any call and the *Post* parts apply for calls matching their respective *Pre* part.

Herein we are interested in call and success patterns conveying only type information. It is possible to write a `pred` assertion with parametric types like:

```
:- pred reverse(X,Y) : list(X,A) => list(Y,A).
```

This assertion tells us that the predicate `reverse/2` is meant to be invoked with the first argument bound to a list whose element can be of any type, denoted by the type variable `A`. Upon success, the procedure returns in the second argument a list whose elements must be of type `A`.

5.2 Using parametric type assertions in modular analysis

Assume a scenario where assertions are written in a (library) module and that for efficiency we do not want to analyze this module if possible. If no assertions are present in the module for exported predicates or if the preconditions in such assertions do not match the calling patterns the module will simply be entered and analyzed during modular analysis, as described in Section 3 (see also (Puebla et al. 2004; Pietrzak et al. 2006)). However, if suitable parametric assertions are present, assume that both precondition Pre and postcondition $Post$ contain parameters $\overline{A}$ and $\overline{B}$ respectively. The assertion takes the following form: $: -$`pred` $P : Pre(\overline{A}) \Rightarrow Post(\overline{B})$. We require $\overline{B} \subseteq \overline{A}$. Our goal is to find, for a given call pattern λ_c, a substitution Ψ of parameters $\overline{A}$ (and therefore $\overline{B}$) such that $\lambda_c \sqsubseteq \lambda_{\Psi(Pre(\overline{A}))}$. Moreover, we are interested in finding a Ψ that gives $\Psi(Pre(\overline{A}))$ that is as precise as possible. In order to achieve this we use the *matching* operation of (Drabent et al. 2002). Matching resembles checking of type inclusion (see (Dart and Zobel 1992; Lu 2001)). We match a parameterless type T_1 against a (possibly parametric) type T_2, and denote this operation $T_1 \dot{\sqsubseteq} T_2$. Matching finds a (possibly small) parameter substitution Ψ so that $T_1 \sqsubseteq \Psi(T_2)$, or fails if such a substitution does not exist. The whole procedure starts with empty Ψ. Then matching, similarly to inclusion checking, traverses the type rules and involved terms recursively, and compares the corresponding structures. If at some point matching is about to compare a type parameter α and a type symbol, say T, then $\alpha \mapsto T$ is added to Ψ. It might however happen that another binding $\alpha \mapsto T'$ is already present in Ψ. In this case, $\alpha \mapsto T'$ is replaced by $\alpha \mapsto T' \sqcup T$.

EXAMPLE 5.3. *Assume the following assertion describing a use of* `append/3`*:*

```
:- pred append(X,Y,Z): (list(X,A), list(Y,A))
                     => list(Z,A).
```

[3] Alternatively we can consider "compatibility" mode, meaning that whatever `L` is instantiated to is *compatible* with being a list (this includes for example `[]`, `[X|Y]`, or simply a variable).

If the analyzer finds a call to append$(X1, X2, X3)$ *with an abstract substitution* $\{X1/list(int), X2/list(int), X3/term\}$ *matching will generate the parameter substitution* $\{$A $\mapsto int\}$. *If however the call pattern has an abstract substitution* $\{X1/list(int), X2/list(atm), X3/term\}$ *then the parameter substitution computed by matching would be* $\{$A $\mapsto int \sqcup atm\}$.

Note that parameter handling is substantially different in typed logic programming (e.g., (Mycroft and O'Keefe 1984; Hill and Lloyd 1994)), where type inference involves type unification that tries to bind a type parameter to a single type and fails if any subsequent binding is incompatible with the previous one. Obviously, if T_2 has no parameters in $T_1 \dot{\sqsubseteq} T_2$, matching reduces to inclusion checking.

Note that without admitting parameters, in order to avoid reanalysis of the library using our proposed rules, the library developer would have to write a specific assertion for each possible type of list elements, which obviously is not feasible. Similarly, standard modular analysis would also save triples for each type of list elements that occurs every time that analysis enters the library. Another remedy is to write the most general assertion:

```
:- pred reverse(X,Y): list(X,term) => list(Y,term).
```

but in this case we unnecessarily lose precision.

5.3 Parametric type assertions in verification and debugging

As mentioned above, proving an assertion with parameters, like the one of Example 5.3, cannot be directly done by using the results of analysis. E.g., consider the assertion $:-$pred $P : Pre(\overline{A}) \Rightarrow Post(\overline{B})$ (1) (where, as before, we assume that $\overline{B} \subseteq \overline{A}$). Essentially, proving (1) means that we want to show: $\forall\Psi(\lambda_c \sqsubseteq \lambda_{\Psi(Pre(\overline{A}))} \Rightarrow \lambda_s \sqsubseteq \lambda_{\Psi(Post(\overline{B}))})$ (2) where λ_c and λ_s are, respectively, the call and success patterns computed by abstract interpretation. We propose a proving method which resembles the well-known skolemization technique. For every parameter α_i we introduce a dummy type c_i, such that $\forall T \in \mathcal{T} : T \neq \top \Rightarrow c_i \sqcap T = \bot$ and $\forall T \in \mathcal{T} : T \neq \bot \Rightarrow c_i \sqcup T = \top$. The intuition is that *Type*(c_i) is disjoint from any terms in the program and initial goal. Let Ψ_c be a parameter substitution $\{\alpha_1 \mapsto c_1, \ldots, \alpha_k \mapsto c_k\}$.

PROPOSITION 5.4. *Consider assertion (1). If abstract interpretation for a top goal P with the initial call pattern $\lambda_{\Psi_c(Pre(\overline{A}))}$ computes a success pattern $\lambda_c = \lambda_{\Psi_c(Post(\overline{B}))}$ then (2) holds.*
Proof (outline): Inductively, for every abstract operation we show that if the operation preserves a dummy type c_i, it will preserve an arbitrary type. Consider a conjunction of two abstract substitutions $\lambda_1 \wedge \lambda_2 = \lambda$. Assume that a dummy type c occurs in λ_1, i.e., for some variable $X \in dom(\lambda_1)$ we have $\lambda_{1|X} = \{X/c\}$. If c propagates to the result of the conjunction, meaning that $\lambda_{|X} = \{X/c\}$, then either $X \notin dom(\lambda_2)$, or $\lambda_{2|X} = \{X/c\}$, or $\lambda_{2|X} = \{X/\top\}$ (as otherwise we would have $\lambda_{|X} = \{X/\bot\}$). It is clear that in either case any other type would propagate the same way as c. A similar reasoning can be performed for disjunction and projection. □

The intuition behind Proposition 5.4 is that if a dummy type can be passed through the entire analysis process, meaning that it has not been "touched" by any abstract operation, we can conclude that any other type would be passed the same way.

6. Example

In this section we illustrate with a simple example our type analysis and its application to program verification. Our example consists of three modules. We start by describing the top-level module of the application, called main, whose code is shown below:

```
:- module(main,[p/2],[assertions,regtypes,functional]).
:- use_module(qs,[qsort/2]).

:- pred p(X,Y): list(X,num) => dlist(Y).    % #1
p(X,Y) :-  dlist(X), qsort(X,Y).

:- regtype dlist/1. dlist := [] | [~digit|dlist].

:- regtype digit/1. digit := 0|1|2|3|4|5|6|7|8|9.
```

In this code, the first line of the program contains the module declaration (in Ciao (Bueno et al. 2006)), which defines the module name and the list of exported predicates, as well as declaring that several *packages* should be used (e.g., for assertion processing). Next, the use_module declaration informs the compiler and analyzer that this module imports procedure qsort/2 from module qs. The main module contains two definitions of regular types: dlist and digit. These definitions are in fact also ordinary Prolog procedures written using Ciao's functional syntax (Casas et al. 2006), and therefore besides their use in the post-condition of assertion #1 they can also be used as regular (test) procedures, as is actually done in the clause defining p/2.

We now present the module qs, which implements *quicksort*, and whose code is shown below:

```
:- module(qs, [qsort/2], [assertions]).
:- use_module(library(lists),[append/3]).

:- pred qsort(X,Y) : list(X,A)  => list(Y,A).% #2
qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L1,R1),
        qsort(L2,R2),
        append(R1,[X|R2],R).
qsort([],[]).

partition([],_B,[],[]).
partition([E|R],C,[E|Left1],Right):-
        E @< C, !,
        partition(R,C,Left1,Right).
partition([E|R],C,Left,[E|Right1]):-
        E @>= C,
        partition(R,C,Left,Right1).
```

An interesting feature of this implementation of quicksort is that the qsort/2 procedure is not restricted to lists of numbers and can in fact accept lists of any kind of elements due to the use of @< and @>= in the comparisons. In this case, the developer of the module has opted to include a pred parametric assertion, marked in the code as #2. This assertion states that on call, the first argument of the qsort/2 procedure is meant to be a list of elements of any type A, whereas upon success, the second argument should be bound to a list of type A. As can be observed, qs imports the standard append/3 predicate, for list concatenation, from the lists library. Therefore, the third and final module in our example is the *lists* library. For our purposes, there is no need to show its code here.

Assume now that we want to prove assertion #1 by analyzing statically module main. If we apply the inter-modular static analysis in (Puebla et al. 2004) using standard regular types, we can prove such assertion since analysis obtains the type dlist (or equivalently list(digit)) for the second argument of p/2. However, and as already argued, in practice this approach is often too costly (both time- and memory-wise) (Correas et al. 2006) since analysis iterates over all modules, including libraries, analyzing for different call patterns until a global fixed point is reached.

We now show on this example how our proposal preserves the required accuracy in order to perform the verification task at hand, i.e., proving assertion #1, while simplifying the analysis process. A first step in our approach is, as described in Section 5, to avoid analyzing libraries, which in general is desirable except in initial

Bench	Mod	Cls	Orig	Optim	SU
ann	3	227	9711	7738	1.25
bid	8	69	7399	3392	2.18
boyer	4	145	1789	1905	0.94
manag_proj	8	907	289564	30962	9.35
check_links	6	576	41862	32392	1.29
grades	4	168	19392	9255	2.09
grade_listing	10	1553	86410	17427	4.96
Wgt. Arith. mean			117194	21297	5.50
Wgt. Geom. mean			69387	18178	3.82

Table 1. Intermodular analysis from scratch, using an underapproximating success policy (SP^-) and a top-down scheduling policy.

phases of library development, verification, and testing. For this, the following assertion is present in the `lists` library:

```
:- checked pred append(X,Y,Z):(list(X,A),list(Y,A))
                            =>  list(Z,A).     % #3
```

Which states that, as expected, the result of concatenating (appending) two lists of a given type `A` results in a list of exactly such type `A`. The `checked` flag (Puebla et al. 2000) in front of the assertion indicates that the assertion has been automatically proved to hold using the method described in Proposition 5.4.

Next, assume that we restrict ourselves to defined types (using the $\lceil . \rceil$ operation) as described in Section 4. With this assumption, when we analyze the procedure `qsort/2` for the call pattern induced by the `main` module, we get the type `list(digit)` in its first argument. Since we know that the `main` module belongs to the set of modules being analyzed, we consider the types defined in `main` as defined types, and therefore no accuracy is lost due to the use of defined types. Next, analysis will reach the call to `append/3`. By using assertion `#3`, the analyzer can deduce, without reanalyzing append, that upon success of `append(R1,[X|R2],R)` in the first clause of `qsort/2`, `R` will be bound to `list(digit)`. This type is propagated through the success of `qsort/2` to the calling module `main`, consequently allowing the system to prove assertion `#1`. Note that if we would like module `qs` to be reusable in any context without reanalyzing `qs` over and over again for different calling patterns, our approach allows introducing parametric assertions, such as assertion `#2`. Then, Proposition 5.4 can again be used to prove this parametric assertion once and for all.

7. Experimental evaluation

In order to evaluate the practical impact of our proposal, we have performed some preliminary benchmarking of modular analysis, in the context of inferring regular types, both with and without our proposed optimizations.

The analysis framework implemented in CiaoPP and used for this paper can be configured selecting specific values for several parameters of the framework. The main parameters that can be selected when performing intermodular analysis are the scheduling policy and the success policy (see (Puebla et al. 2004) for more information on those and other parameters.) The scheduling policy allows the user to select how the framework decides at each iteration which module must be the next one selected for analysis during the intermodular fixed point computation. Two main approaches have been implemented: a *top-down* policy, traversing the intermodular dependency graph and selecting first the module requiring analysis which is higher in the graph (the top-level module is the top of the graph). The *bottom-up* policy takes first the deepest module in the intermodular dependency graph which requires analysis. When a program is analyzed from scratch, the first module analyzed is always the top-level one. The success policy, as already mentioned, selects how temporary results for calls to imported predicates are approximated when the exact success pattern is not available in the Global Answer Table. During the experiments, the parameters for both policies have been set to the most advantageous setting for the original type analysis, namely top-down and SP^-, respectively, in order to highlight the speedup obtained with just defined types.

The benchmark programs used are modular programs of medium size, ranging from three to ten modules, and from 69 to 1553 clauses. The number of modules and clauses for each program is detailed in Table 1. A brief description of the selected benchmarks follows. **ann** is the &-Prolog implementation of the MEL automatic parallelizer (by K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo) (Muthukumar and Hermenegildo 1990). **bid** computes an opening bid for a bridge hand (by J. Conery). The **boyer** benchmark is a reduced version of the Boyer/Moore theorem prover (by E. Tick). The program has been separated in four modules with a cycle between two modules. **managing_project** is a program developed by the authors for EU project management. **check_links** is an sample program for the *Pillow* HTML/XML/HTTP connectivity package (by D. Cabeza and M. Hermenegildo) that checks that links contained in a given URL address are reachable. Note that the whole Pillow package is analyzed together with the sample program. And finally, **grades** and **grade_listing** are programs used by the authors for grading students, and are composed of 4 and 10 modules, respectively. The experiments have been run on a Dell PowerEdge 4600 with two Pentium processors at 2 Ghz and 4 Gb of memory, and normal workload. Analysis time in the experiments corresponds to the time spent (in milliseconds) analyzing code. Tasks related to program loading and unloading, and saving analysis results to disk are not part of the optimizations described in this paper and have been excluded from the tables.

The first experiment involves analyzing from scratch the benchmark programs using the intermodular analysis algorithm. The results are shown in Table 1. Column **Mod** contains the number of modules that compose each benchmark, while column **Cls** includes the number of clauses. Column **Orig** is the time spent by the original regular type analysis, and column **Optim** is the time spent by the analysis with the optimizations described in this paper. Finally, column **SU** (speedup) shows the improvement brought by the optimized version over the traditional type analysis. In this case the improvement is considerable, supporting our thesis that the optimizations proposed are specially appropriate for modular analysis. The last two lines of tables 1 and 2 show the arithmetic and geometric means of the results obtained for each column, weighted by the number of clauses in each program. On average, our proposed optimizations speed up intermodular analysis by a factor of 3.82-5.5. One case in which our approach actually brings a slow-down is boyer. In this case around one third of the time is spent in a single module (`equal.pl`, 640 ms out of 1905 ms), which contains a few tables with a relatively large set of facts each and which have complex data structures as arguments. Types are inferred for all those facts and then lubbed in order to use defined types only, which in this case is more costly than simply creating new types.

Table 2 shows how the optimizations proposed improve the analysis results when benchmark programs are reanalyzed in an incremental way, after several specific modifications are made to the source code. For evaluating this, it is important to make experiments which are representative of the kind of changes which occur in real-life. We have followed here the approach used in (Correas et al. 2006), were three different kinds of source code modifications have been studied. For the three classes of changes, one module is modified each time and then the program is reanalyzed, in order to incrementally recompute the analysis results visiting only the modules that require reanalysis: the changed module, plus possibly

	module touch			more general clause			recursion removal		
Bench	**Orig**	**Optim**	**SU**	**Orig**	**Optim**	**SU**	**Orig**	**Optim**	**SU**
ann	1798	1134	1.59	6026	3204	1.88	3335	1585	2.10
bid	620	206	3.01	2035	574	3.54	1068	297	3.60
boyer	222	559	0.40	331	625	0.53	401	595	0.67
manag_proj	15134	15077	1.00	41664	14668	2.84	53383	1419	37.62
check_links	7439	6449	1.15	18232	9958	1.83	10353	6964	1.49
grades	3385	734	4.61	5432	1196	4.54	4186	933	4.49
grade_listing	4119	4557	0.90	36458	7173	5.08	16707	7353	2.27
Wgt. Arith. mean	6985	6843	1.02	29459	8570	3.44	22475	4757	4.72
Wgt. Geom. mean	5054	4741	1.07	21724	6838	3.18	14364	3406	4.22

Table 2. Reanalysis after several kinds of changes, using an underapproximating success policy (SP^-) and a bottom-up scheduling policy (in the case of recursion removal, SP^+ and top-down scheduling have been used.)

other modules transitively affected by this change. The numbers shown in Table 2 are the average of the times taken by the reanalysis of the program when each of the modules in the benchmark programs is modified.

In the first kind of change, a simple modification is made in a single module in such a way that this modification does not change the results of analysis for that module (named **module touch** in that table). This has been implemented by "touching" the module, i.e., changing the modification time without actually modifying its contents, in order to force CiaoPP to reanalyze it. It can be observed that the optimizations introduced do not provide much speedup. The main reason for this behavior is that only the modified module is visited by the modular analysis. This means that the reduction in the number of intermodular iterations and number of types used is not relevant in this case. Also, a single module in a modular program is typically a rather small piece of code, which may not be large enough to take advantage of the use of just defined types.

The second kind of modification shown in Table 2 is a modification in the source code such that after the change exported predicates produce more general analysis results (named **more general clause** in Table 2). It is implemented by adding a most general fact to all exported predicates of a given module. This kind of change is an extreme situation in which all exported predicates are affected. This modification in general requires that not only the modified module be reanalyzed, but also some other related modules, since the analysis results for the modified module are different, and very likely affect the modules which import the modified one. It is encouraging to observe that the optimizations introduced by our approach appear to be specially relevant in this case, since due to them the process of reanalyzing the program is sped up by a factor of more than three.

The third case corresponds to a source code modification in which exported predicates produce a more precise answer pattern (named **recursion removal** in Table 2). In this case all the clauses of the exported predicates of a given module have been replaced by the first non-recursive clause of the predicate. As in the previous case, this is an extreme case in which all exported predicates are affected, except that now a more particular answer is obtained for them instead of a more general one. Again, the reanalysis of the program after the change generally requires analyzing other modules besides the modified one. As shown in the table the modular analysis is very competitive when using our proposed approach, bringing significant speedup also in this case, by a factor of more than four.

Finally, the third experiment, shown in Table 3, presents the time spent when analyzing the programs in a non-modular way, i.e., as if all the program code were located in a single module (we refer to this as the "monolithic analysis"). The results suggest that although the optimizations presented in this paper aim at improving modular analysis, they are also useful in the case of large non-modular programs, since the analysis speed improves for most of the benchmarks. However, as can be seen for **ann**, there may be cases where the original analysis is somewhat faster. This can be explained by the fact that though using the $\lceil . \rceil$ operation results in analyzing fewer call patterns, the replacement operation also introduces some overhead. The results also show that modular analysis times remain, as expected, somewhat higher than when analyzing in a monolithic way, but are reasonable in comparison, and the comparison with the analysis times for reanalysis after partial changes are quite encouraging, since they often improve on the monolithic analysis times, notably for some of the programs with larger execution times. Note also that of course modular analysis is vital when programs cannot be analyzed monolithically due to, e.g., memory limitations.

Bench	**Orig**	**Optim**	**SU**
ann	1291	1605	0.80
bid	1972	731	2.70
boyer	973	935	1.04
manag_proj	37177	28849	1.29
check_links	10266	7158	1.43
grades	5653	2348	2.41
grade_listing	60552	16052	3.77

Table 3. Time spent by the monolithic analysis of different benchmark programs.

8. Conclusions

We have proposed a combination of techniques aimed at improving analysis efficiency in type inference and verification for modular Prolog programs. In particular, we have presented a type analysis which optionally reduces the accuracy of inferred types during the analysis process by using only the types defined by the user or present in the libraries. Also, borrowing some ideas from polymorphic type systems, we have proposed a method that allows using polymorphic type rules for specifying library module boundaries, and we have proposed a novel method in order to use such type rules in the context of a regular type-based analysis system. Finally, we have also implemented our approach and reported experimental results from the analysis of a number of modular programs.

Our experimental results suggest that the optimizations presented do contribute significantly to increasing analysis efficiency both for the monolithic case and, even more, for the case of analyzing programs module by module. This holds both when analyzing programs from scratch as well when doing it incrementally after changes to a single module. Modular analysis times remain, as expected, somewhat higher than analyzing them in a monolithic way, but are reasonable, and the results from reanalysis after partial changes are quite encouraging, improving sometimes on the monolithic analysis times. Another advantage of the proposed approach is that the output of analysis is more readable, since it is presented to the user in terms of known types, rather than in terms of new, inferred ones, which are typically more detailed and have automatically generated names, and which can sometimes be difficult to interpret. Furthermore, the precision obtained appears to be sufficient for the purpose of verifying type signatures given in the form of assertions, as the inferred types are exactly those which occur in the assertions.

In summary, we argue that our proposal is of practical relevance, since it allows reducing analysis cost significantly while preserving a useful level of accuracy.

Acknowledgments

This work was funded in part by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under project FP6 IST-15905 *MOBIUS*, the Spanish Ministry of Science and Education under projects TIN2005-09207-C03-01 and 03 *MERIT-COMVERS* and *MERIT-FORMS*. M. Hermenegildo is also supported in part by the Prince of Asturias Chair in Information Science and Technology at the University of New Mexico. P. Pietrzak is supported by a “Juan de la Cierva” grant provided by the Spanish MEC.

References

R. Barbuti and R. Giacobazzi. A bottom-up polymorphic type inference in logic programming. *Science of Computer Programming*, 19(3):281–313, 1992.

M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.

A. Casas, D. Cabeza, and M. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *The 8th International Symposium on Functional and Logic Programming (FLOPS’06)*, Fuji Susono (Japan), April 2006.

M. Codish and V. Lagoon. Type Dependencies for Logic Programs using ACI-unifi cation. *Journal of Theoretical Computer Science*, 238:131–159, 2000.

J. Correas, G. Puebla, M. Hermenegildo, and F. Bueno. Experiments in Context-Sensitive Analysis of Modular Programs. In *15th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR’05)*, number 3901 in LNCS, pages 163–178. Springer-Verlag, April 2006.

P. Cousot and R. Cousot. Abstract Interpretation: a Unifi ed Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of POPL’77*, pages 238–252, 1977.

P. Cousot and R. Cousot. Formal language, grammar and set-constraint-based program analysis by abstract interpretation. In *Proceedings of the Seventh ACM Conference on Functional Programming Languages and Computer Architecture*, pages 170–181, La Jolla, California, 1995. ACM Press, New York, NY.

P.W. Dart and J. Zobel. A Regular Type Language for Logic Programs. In *Types in Logic Programming*, pages 157–187. MIT Press, 1992.

M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, September 1996.

W. Drabent, J. Małuszyński, and P. Pietrzak. Using parametric set constraints for locating errors in CLP programs. *Theory and Practice of Logic Programming*, 2(4–5):549–611, 2002.

J. Gallagher and K. Henriksen. Abstract domains based on regular types. In B. Demoen and V. Lifschitz, editors, *ICLP 2004, Proceedings*, volume 3132 of *LNCS*, pages 27–42. Springer-Verlag, 2004. ISBN 3-540-22671-0.

J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In *Fourth International Symposium on Practical Aspects of Declarative Languages*, number 2257 in LNCS, pages 243–261. Springer-Verlag, January 2002.

J.P. Gallagher and D.A. de Waal. Fast and precise regular approximations of logic programs. In Pascal Van Hentenryck, editor, *Proc. of the 11th International Conference on Logic Programming*, pages 599–613. MIT Press, 1994.

N. Heintze and J. Jaffar. A fi nite presentation theorem for approximating logic programs. In *Proc. 17th POPL*, pages 197–209, 1990.

N. Heintze and J. Jaffar. Semantic types for logic programs. In Pfenning (1992), pages 141–155.

M. Hermenegildo, G. Puebla, F. Bueno, and P. López García. Integrated Program Debugging, Verifi cation, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.

P.M. Hill and J.W. Lloyd. *The Gödel Programming Language*. Logic Programming Series. MIT Pres, 1994.

G. Janssens and M. Bruynooghe. Deriving Descriptions of Possible Values of Program Variables by means of Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):205–258, July 1992.

J.W. Lloyd. *Foundations of Logic Programming*. Springer, 2nd Ext. Ed., 1987.

L. Lu. Type Analysis of Logic Programs in the Presence of Type Defi nitions. In *Proceedings of the 1995 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based program manipulation*, pages 241–252. The ACM Press, 1995. URL `http://www.oakland.edu/ l2lu/pubs/DetType.ps`.

L. Lu. Polymorphic Type Analysis of Logic Programs by Abstract Interpretation. *Journal of Logic Programming*, 36(1):1–54, 1998. URL `http://www.oakland.edu/ l2lu/pubs/PolyType.ps`.

L. Lu. On Dart-Zobel Algorithm for Testing Regular Type Inclusion. *SIGPLAN NOTICES*, 36(9):81–85, 2001. URL `http://www.oakland.edu/ L2LU/pubs/DartZobel.ps`.

P. Mildner. *Type Domains for Abstract Interpretation: A Critical Study*. PhD thesis, Computing Science Department - Uppsala University, Uppsala, 1999.

P. Mishra. Towards a theory of types in prolog. In *International Symposium on Logic Programming*, pages 289–298, Silver Spring, MD, February 1984. Atlantic City, IEEE Computer Society.

K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Int’l. Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

A. Mycroft and R.A. O’Keefe. A Polymorphic Type System for Prolog. *Artifi cial Intelligence*, 23:295–307, 1984.

F. Pfenning, editor. *Types in Logic Programming*. MIT Press, 1992.

Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, February 2002.

P. Pietrzak, J. Correas, G. Puebla, and M. Hermenegildo. Context-Sensitive Multivariant Assertion Checking in Modular Programs. In *13th International Conference on Logic for Programming Artifi cial Intelligence and Reasoning (LPAR’06)*, number 4246 in LNCS, pages 392–406. Springer-Verlag, November 2006.

G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, pages 23–61. Springer LNCS 1870, 2000.

G. Puebla, J. Correas, M. Hermenegildo, F. Bueno, M. García de la Banda, K. Marriott, and P. J. Stuckey. A Generic Framework for Context-Sensitive Analysis of Modular Programs. In M. Bruynooghe and K. Lau, editors, *Program Development in Computational Logic, A Decade of Research Advances in Logic-Based Program Development*, number 3049 in LNCS, pages 234–261. Springer-Verlag, Heidelberg, Germany, August 2004. ISBN 3-540-22152-2.

H. Saglam and J. Gallagher. Approximating constraint logic programs using polymorphic types and regular descriptions. Technical Report CSTR-95-17, Department of Computer Science, University of Bristol, Bristol BS8 1TR, 1995.

Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury: an Effi cient Purely Declarative Logic Programming Language. *JLP*, 29(1–3), October 1996.

P. Van Hentenryck, A. Cortesi, and B. Le Charlier. Type analysis of prolog using type graphs. *Journal of Logic Programming*, 22(3):179–209, 1995.

P. Van Hentenryck, O. Degimbe, B. Le Charlier, and L. Michael. The Impact of Granularity in Abstract Interpretation of Prolog. In *Workshop on Static Analysis*, number 724 in LNCS, pages 1–14. Springer-Verlag, September 1993.

C. Vaucheret and F. Bueno. More Precise yet Effi cient Type Inference for Logic Programs. In *International Static Analysis Symposium*, number 2477 in LNCS, pages 102–116. Springer-Verlag, September 2002.

E. Yardeni and E.Y. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10(2):125–154, 1990.