ASAP

IST-2001-38059

Advanced Analysis and Specialization for Pervasive Systems

Combined Static and Dynamic Checking

Deliverable number:	D17
Workpackage:	Correctness in Pervasive Computing (WP6)
Preparation date:	1 November 2004
Due date:	1 November 2004
Classification:	Public
Lead participant:	Univ. of Southampton
Partners contributed:	Tech. Univ. of Madrid (UPM), Univ. of Southampton, Roskilde Univ

Project funded by the European Community under the "Information Society Technologies" (IST) Programme (1998–2002).

Short description:

Correctness of programs can only be determined with respect to some expected properties, specified for example by means of assertions to be checked. Such an assertion checking process can be performed either at run-time (dynamic checking) or at compile-time (static checking).

Goals of Task 6.2. This deliverable is an outcome of task 6.2 *Combined Static and Dynamic Checking*, whose goal is to study the use of abstract interpretation to validate assertions at compile time, and identify properties that have to be checked at run-time. In this deliverable we focus on static assertion checking (dynamic checking will be the focus of Deliverable D20, WP6).

Advances in compile-time checking of assertions leads on the one hand to *performance improvements*, since a run-time computation can be omitted if static analysis shows that it is redundant. In addition, compile-time checking of assertions leads to *increased reliability* of software, by compile-time checking of assertions related to avoidance of run-time errors. Both performance and reliability are particularly important in the context of pervasive systems.

Broadly speaking we divide the work into two streams: the first is concerned with annotating user-written programs with assertions, specifying desired properties that are supposed to hold at certain program points. These assertions are checked statically. The second approach seeks to validate properties by construction, by automatically deriving code that satisfies some specification.

Introduction to the contents of Deliverable D17 .

Part I A self-contained description of the `CiaoPP` assertion language is given first in Part I. Assertions capture a variety of properties of the computation state, including properties that hold at success of a call and properties that are intended to hold at call time, as well as properties of the computation itself such as termination, determinacy, and side-effect freeness.

Recent improvements (as well as ongoing work) in assertion checking in `CiaoPP` are reported in this section. The main emphasis has been put on increasing the accuracy of assertion checking, thereby enabling more assertions to be checked at compile-time and reducing the number of run-time tests. We have developed an assertion checking scheme that exploits multivariant analysis information and is thus more precise. Moreover, an improved method for checking “success with pre-condition” assertions is discussed.

Part II In Part II we consider the question of increasing the flexibility and applicability of assertions. We consider the general problem of *transforming* assertions into different abstract domains. The `CiaoPP` assertions are specified in a rich language, but in order to take advantage of them it is necessary to perform one of a fixed number of analyses. It is not practical to foresee each and every domain in which they should be incorporated. To overcome this problem we define a procedure for transforming assertions from the `CiaoPP` assertion database into abstract models of predicates over arbitrary domains based on pre-interpretations.

The problem of obtaining abstractions of builtins is often a stumbling block to analysing real application programs accurately. It requires considerable effort to specify the properties of builtins over each different abstract domain. Without such an effort though, once is forced to make coarse over-approximations of the builtins.

The method described in Part II allows the existing, permanent assertions for the builtins over a rich set of system types to be transported automatically to any given domain defined as a pre-interpretation. Such domains can be constructed from arbitrary regular types.

The method is based on building a complete signature of the program and all its types. Some of the types are contextual, that is, defined in terms of a particular signature. Once the compete types are obtained, they are determinized and then the predicate models are projected from the determinized type domain onto the types of interest.

Apart from abstractions of builtins, the method allows the analysis results for modules that have been analysed over one domain to be imported into modules which are being analysed over a different domain. This adds to the flexibility and practicability of module-based analyses.

The method has been fully implemented and initial experiments are reported.

Part III In Part III we consider the application of *backwards analysis* for inferring conditions which guarantee that given runtime assertions will be satisfied. A novel method for backwards analysis in a standard abstract interpretation framework was developed in the ASAP project during the first period. The deliverable shows how `CiaoPP` assertions on built-ins and other imported predicates can be propagated backwards, deriving conditions on the entry calls that guarantee satisfaction of the assertions.

The first experimental result reported in Part III show the flexibility of our approach compared to other results reports in the literature, which discuss similar experiments but are limited to simple Boolean domains like Pos. In our approach, we can use arbitrary regular types, mixing system types with user definitions.

Part IV The final contribution describes work in the second stream mentioned above - namely, to satisfy program properties by construction. Here we make use of ASAP tools not to perform the checking, but rather to generate the code. Hence the tools employed are concerned with program specialization rather than analysis.

Provably correct compilation is an important aspect in development of high assurance software systems. We have thus explored approaches to provably correct code generation based on programming language semantics, particularly *Horn logical semantics*, and partial evaluation. We show that the *definite clause grammar (DCG)* notation can be used for specifying both the syntax and semantics of imperative languages. We next show that continuation semantics can also be expressed in the Horn logical framework, and we have used the LOGEN partial evaluator, as developed in the ASAP project during the first period, to automatically derive "provably correct" compilers from those semantic specifications.

A case study discussed in Part IV is the compilation of a domain specific language SCR (Software Cost Reduction) requirements language, which is targetted at embedded systems.

Part IV was published in the Proceedings of Seventh International Symposium on Practical Aspects of Declarative Languages (PADL'05), Editors: Manuel Hermenegildo, Daniel Cabeza, Springer LNCS, Volume 3350, 2005.

Conclusions: Results of Task 6.2. Static checking is a large area of research, potentially including all of program analysis and verification. In this task we have focussed on two areas: firstly, to enhance and exploit the foundation of assertions in the `CiaoPP` system, increasing the accuracy, flexibility and range of applications of the assertions. This aspect relies on analysis tools developed in the ASAP project. Secondly, we use ASAP specialization tools to generate code from higher level specifications.

In both these areas this deliverable reports incremental progress. The existing assertion checking tools are strengthened, and new ways of exploiting assertions are developed, by incorporating them in other abstract domains, and by performing backwards analysis. Preliminary experimental results in these areas are promising and have contributed also to integration of the ASAP tools. In the area of code generation, the results of Part IV show that the difficult problem of taking semantic specifications of a language as the starting point for a provably correct compiler has yielded some promising results.

In an area as large as this it is important to identify problems and limitations. Consideration could be given to the assertion language of `CiaoPP`, although rich, could be rationalised and made more expressive in certain areas. The problem of mapping assertions to other domains introduces complexity problems due to the potential explosion of disjoint types generated in the procedure. Current research is investigating more compact representations of determinized types. In backwards analysis, a key problem is to analyse with respect to sufficiently precise domains to allow back-propagation of the relevant information. Otherwise the results are likely to be too coarse to be useful. In code generation, the problems are of course to extend the method to more complex languages, and to generate efficient code.

Attachments:

Part I (D17.1) —Some Improvement in Compile-Time Assertion Checking in `CiaoPP`.

Part II (D17.2) — Integrating `CiaoPP` Assertions into Abstract Interpretations.

Part III (D17.2) — Experiments with Backwards Analysis.

Part IV (D17.4) — Towards Provably Correct Code Generation via Horn Logic Continuation Semantics. Appears in Proceedings of Seventh International Symposium on Practical Aspects of Declarative Languages (PADL'05), Editors: Manuel Hermenegildo, Daniel Cabeza, Springer LNCS, Volume 3350, 2005.

Contents

Part I

Improved Compile-Time Assertion Checking in CiaoPP

1 Introduction

This work presents some aspects of static assertion checking in the CiaoPP (Ciao Prolog preprocessor) system. In previous papers on this topic [47, 48], a flexible framework for assertion checking has been introduced which allows assertion-based debugging of (constraint) logic programs.

Assertions are linguistic constructions for expressing properties of programs. In this work we recall several assertion schemas for writing (partial) specifications for constraint logic programs using quite general properties. The entire framework is aimed at detecting deviations of the program behavior (symptoms) with respect to the given assertions, either at compile-time (i.e., statically) or run-time (i.e., dynamically). In general, it is desirable that the amount of run-time checks is limited, as it produces an overhead during program execution. Reducing the number of run-time checks means to perform them at compile-time, whenever feasible. This work concerns static assertion checking and extends the existing framework in two directions:

- exploiting multivariant analysis information, including the special case which is multi-success analysis;
- improving the accuracy of checking 'success-with-precondition' assertions.

Thus, we are focused on increasing power of compile-time assertion checking.

2 The Ciao assertion language

Below we recall some basics of the Ciao assertion language (see [47] for more in-depth discussion on the assertion language).

2.1 Assertions and program correctness

Assertions are statements about program's properties. Every assertion A is conceptually composed of two logic formulae which we refer to as app_A and sat_A. Evaluation of these logic formulae should return either the value *true* or the value *false* when evaluated on the corresponding context (i.e., execution state, correct answer, computation, or whatever is the "semantic context" which the assertion refers to) by using an appropriate inference system. The formula app_A determines the *applicability set* of the assertion: a context s is in the applicability set of A iff app_A takes the value *true* in s. Also, we say that an assertion A is *applicable* in context s iff app_A holds in s. The formula sat_A determines the *satisfiability set* of the assertion: a context s is in the satisfiability set of A iff sat_A takes the value *true* in s. If we can prove that there is a context which is in the applicability set of an assertion A but is not in its satisfiability set then the program is definitely incorrect w.r.t. A. Conversely, if we can prove that every context in which A is applicable is in the satisfiability set of A then the program is validated w.r.t. A. We consider a program P correct only if it is correct w.r.t. all assertions for P.

2.2 Properties related to computation states

When considering the operational behaviour of a program, it is natural to associate (sets of) execution states with certain syntactic elements of the program. As usually in CLP, a program is composed of a set of *predicates* (also known as *procedures*). Also, a program can be seen, at a finer-grained level, as composed of a set of *program points*. Thus, we first introduce several assertion schemes whose applicability context is related to a given predicate. Then we introduce an assertion schema whose applicability context is related to a particular program point. We refer to the former kind of assertions as *predicate* assertions, and to the second one as *program-point* assertions. Though a simple program transformation technique could be used to express program-point assertions in terms of predicate assertions, we maintain both kinds of assertions in our language for pragmatic reasons.

As a general rule, we restrict the properties expressible by means of assertions about execution states to those which refer to the values of certain variables in the store of the corresponding execution state. This has the advantage that, in order to check whether the app_A and sat_A logic formulae hold or not, it suffices to inspect the store at the corresponding execution state. Also, the variables (arguments) on whose value we may state properties are also restricted in some way. In the case of predicate assertions, the arguments whose value we can inspect are those in the head of the predicate. In the case of program-point assertions, they are the variables in the clause to which the program point belongs.

Note that more than one predicate assertion may be given for the same predicate. In such a case, all of them should hold for the program to be correct and composition of predicate assertions should be interpreted as their conjunction.

2.2.1 An assertion schema for success states

This assertion schema is used in order to express properties which should hold on termination of any successful computation of a given predicate. They are similar in nature to the *postconditions* used in program verification. They can be expressed in our assertion language using the assertion schema:

`:- success` *Pred* `=>` *Postcond*`.`

This assertion schema has to be instantiated with suitable values for *Pred* and *Postcond*. *Pred* is a *predicate descriptor*, i.e., it has a predicate symbol as main functor and all arguments are distinct free variables, and *Postcond* is a logic formula about execution states, and which plays the role of the sat_A formula. The resulting assertion should be interpreted as "in any activation of *Pred* which succeeds, *Postcond* should hold in the success state."

Example 2.1 *We can use the following assertion in order to require that the output (second argument) of procedure* `qsort` *for sorting lists be indeed sorted:*

```
:- success qsort(L,R) => sorted(R).
```

Clearly, we are assuming that `sorted(R)` *is interpreted in a suitable inference system, in which it takes the value true iff* `R` *is bound to a sorted list. The assertion establishes that this (atomic) formula is applicable to all execution states which correspond to a success of* `qsort`.

An important thing to note is that in contrast to other programming paradigms, in (C)LP a call to a predicate may generate zero (if the call fails), one, or several success states, in addition to looping (or returning error). The postcondition stated in a `success` assertion refers to *all* the success states (possibly none).

2.2.2 Adding preconditions to the success schema

The `success` schema can be used to consider only those successful states which correspond to activations of the predicate which at the time of calling the predicate satisfy certain *precondition*. The preconditions we consider are, in the same way as *Postcond*, logic formulae about states. The success schema with precondition takes the form:

`:- success` *Pred* `:` *Precond* `=>` *Postcond*`.`

and it should be interpreted as "in any invocation of *Pred* if *Precond* holds in the calling state and the computation succeeds, then *Postcond* should also hold in the success state." The *Precond* formula should not be confused with the applicability condition app_A. Note that '`:- success` *Pred* `=>` *Postcond*' is equivalent to '`:- success` *Pred* `:` *true* `=>` *Postcond*'.

It is important to also note that even though both *Precond* and *Postcond* are logic formulae about execution states, they refer to different execution states. *Precond* must be evaluated w.r.t. the store at the calling state to the predicate, whereas *Postcond* must be evaluated w.r.t. the store at the success state of the predicate.

Example 2.2 *The following assertion requires that if* `qsort` *is called with a list of integers in the first argument position and the call succeeds, then on success the second argument position should also be a list of integers:*

```
:- success qsort(L,R) : list(L,int) => list(R,int).
```

where `list(A,int)` *is an atomic formula which takes the value true iff* `A` *is bound to a list of integers in the corresponding state.*

2.2.3 An assertion schema for call states

We now introduce an assertion schema whose aim is to express properties which should hold in any call to a given predicate. These properties are similar in nature to the classical *preconditions* used in program verification. A typical situation in which this kind of assertions are of interest is when the implementation of a predicate assumes certain restrictions on the values of the input arguments to the predicate. Such implementation is often not guaranteed to produce correct results unless such restrictions hold. Assertions built using this schema can be used to check whether any of the calls for the predicate is not in the expected set of calls (i.e., the call is "inadmissible" [46]). This schema has the form:

`:- calls` *Pred* `:` *Precond*`.`

This assertion schema has to be instantiated with a predicate descriptor *Pred* and a logic formula about execution states *Precond*. The resulting assertion should be interpreted as "in all activations of *Pred* the formula *Precond* should hold in the calling state."

Example 2.3 *The following assertion built using the* `calls` *schema expresses that in all calls to predicate* `qsort` *the first argument should be bound to a list:*

```
:- calls qsort(L,R) : list(L).
```

2.2.4 An assertion schema for query states

It is often the case that one wants to describe the exported uses of a given predicate, i.e., its valid queries. Thus, in addition to describing calling and success states, we also consider using assertions to describe *query states*, i.e., valid input data. In terms of the operational semantics, in which program executions are sequences of states, query states are the initial states in such sequences. These can be described in our assertion language using the `entry` schema, which has the form:

`:- entry` *Pred* `:` *Precond*.

where, as usual, *Pred* is a predicate descriptor and *Precond* is a logic formula about execution states. It should be interpreted as "*Precond* should hold in all initial queries to *Pred*."

Example 2.4 *The following assertion indicates that the predicate* `qsort/2` *can be subject to top-level queries provided that such queries have a list of numbers in the first argument position:*

```
:- entry qsort(L,R) : list(L,num).
```

The set of all `entry` assertions is considered *closed* in the sense that they must cover all valid initial queries. This is equivalent to considering that an assertion of the form '`:- entry` *Pred* `: false.`' exists for all predicates *Pred* for which no `entry` assertion has been provided.

It can be noted that `entry` and `calls` schemes are syntactically (and semantically) similar. However, their applicability set is different. The assertion in the example above only applies to the initial calls to `qsort`, whereas, for example, the assertion '`:- calls qsort(L,R) : list(L,num).`' applies to any call to `qsort`, including all recursive (internal) calls. Thus, `entry` assertions allow providing more precise descriptions of initial calls, as the properties expressed do not need to hold for the internal calls.

Example 2.5 *Consider the following program with an entry assertion:*

```
:- entry p(A) : ground(A).
p(a).
p(X):- p(Y).
```

If instead of the `entry` *above we had written* '`:- calls p(A) : ground(A).`' *then such assertion would not hold in the given program. For example, the execution of* `p(b)` *produces calls to* `p` *with the argument being a free variable. However, the execution of* `p(b)` *satisfies the* `entry` *assertion since the internal calls to* `p` *are not in the applicability context of the assertion.*

2.2.5 Program-point assertions

As already mentioned, usually, when considering operational semantics of a program, in addition to predicates we also have the notion of *program points*. The program points that we will consider are the places in a program in which a new literal may be added, i.e., before the first literal (if any) of a clause, between two literals, and after the last literal (if any) of a clause. For simplicity, we add program-point assertions to a program by adding a new literal at the corresponding program point. This literal is of the form:

`check`(*Cond*).

an it should be interpreted as "whenever execution reaches a state originated at the program point in which the assertion is, *Cond* should hold." Intuitively, each execution state can be seen as originated at a given program point.

Example 2.6 *Consider the following clause* '`p(X):- q(X,Y), r(Y).`' *Imagine for example that whenever the clause is reached by execution, after the successful execution of the literal* `q(X,Y)`, `X` *should be greater than* `Y` *and* `Y` *should be positive. This can be expressed by replacing the previous clause by the following one in which a program-point assertion has been added:*

```
p(X):-  q(X,Y), check((X>Y,Y>=0)), r(Y).
```

An important difference between program-point assertions and predicate assertions is that while the latter are not part of the program, program-point assertions are, as they have been introduced as new literals in some program clauses. In order to avoid program-point assertions from altering the behaviour of the program (at least if dynamic checking has not been enabled), we assume that the predicate `check/1` is defined as

```
check(_Prop).
```

i.e., any call to `check` trivially succeeds. If dynamic checking is being performed, this definition is overridden by another one which actually performs the (run-time) checking. This topic is beyond scope of this work.

2.3 Disjunctions in assertions

In this section we advocate using a limited form of disjunctions in assertions related to computation states. The need for disjunctions in assertions becomes clear when we employ multivariant analysis and assertion checking as discussed in Section 3.3, below. Consider the following program:

```
:- entry p(_,_).
p(a,a).
p(b,b).
```

Assume that we want to precisely describe the success set of `p/2` by means of assertions. An assertion[1]

```
:- check success p(A,B) => (t_ab(A), t_ab(B)).
:- regtype t_ab/1.
t_ab(a). t_ab(b).
```

is not precise enough, since the description also includes successes `p(a,b)` and `p(b,a)`. Having two separate assertions:

```
:- check success p(A,B) => (t_a(A), t_a(B)).
:- check success p(A,B) => (t_b(A), t_b(B)).
:- regtype t_a/1.
t_a(a).
:- regtype t_b/1.
t_b(b).
```

does not improve the situation; the `success` scheme reads "for every success the property holds", so the two assertions cannot hold at the same time. It seems that the only option is to allow disjunctions:

```
:- check success p(A,B) => ((t_a(A), t_a(B)) ; (t_b(A), t_b(B))).
```

Clearly, disjunctions in postconditions do increase expressive power of the assertion language. However, they are not needed in precondition of success assertions, as a disjunction in a precondition can be replaced by a pair of assertions.

Disjunctions in `calls` assertions should be admitted as well, in order to express multiple way of using a predicate (for example a classical `append/3` can be used either for list concatenation or list decomposition and thus it is called in two different ways).

We believe also that in order to keep the right balance between expressive power and efficiency, nested disjunctions should not be allowed.

[1]A tag `regtype` indicates a property predicate that defi nes a regular type.

2.4 Properties of computations

We have seen properties and corresponding assertions which refer to computation states. Another kind of properties consider whole computations. They can be expressed in the assertion language by the `comp` schema.

The `comp` schema is, in the same way as `success` and `calls` schemes, associated to predicates and is inherently operational. The `success` and `calls` schemes allow expressing properties about the execution states both when the predicate is called and when it terminates its execution with success. However, as we mentioned above, many other properties which refer to the computation of the predicate (rather than the input-output behaviour) are not expressible with such schemes. In particular, no property which refers to (a sequence of) intermediate states in the computation of the predicate can be (easily) expressed using `calls` and `success` predicate assertions only. Examples of properties of the computation which we may be interested in are: determinacy [44], non-failure [27], computational cost [32, 33], termination, etc.

In our language, this sort of properties are expressed using the schema:

`:- comp` *Pred* [`:` *Precond*] `+` *Comp-prop* `.`

where *Pred* is a predicate descriptor, *Precond* is a logic formula on execution states, and *Comp-prop* is a logic formula on computations. As in the case of `success` assertions, the field '`:` *Precond*' is optional. An assertion built using the `comp` schema should be interpreted as "in any activation of *Pred* if *Precond* holds in the calling state then *Comp-prop* should also hold for the computation of *Pred*." Alternatively, it can be interpreted as "the applicability set of the assertion is the set of computations of $Pred$ in which the logic formula on states *Precond* holds at the calling state, and its satisfiability set has all computations in which the logic formula on computations *Comp-prop* holds."

Example 2.7 *The following assertion could be used to express that all computations of predicate* `qsort` *with the first argument being a list of numbers and the second an unconstrained variable at the calling state should produce at least one solution in finite time.*

```
:- comp qsort(L,R) : ( list(L,num), var(R) ) + succeeds.
```

where the atom `succeeds` *is implicitly interpreted as* `succeeds(qsort(L,R))`*, with an extra argument, i.e., it is the execution of* `qsort(L,R)` *that has to succeed.*

2.5 Status of assertions

Assertions can be used in different tools for different purposes. In some of them we may be interested in expressing expected properties of the program if it were correct, i.e., intended properties, whereas in other contexts we may also be interested in expressing properties of the actual program in hand, i.e., actual properties, which may or may not correspond to the user's intention. For example, we can use program analysis techniques to infer properties of the program in hand and then use assertions in order to express the results of analysis. Thus, the assertion language should be able to express both intended and actual properties of programs. However, all the assertions presented in the examples in previous sections relate to intended properties. We have delayed the other uses of assertions until now for clarity of the presentation.

In our assertion language we allow adding in front of an assertion a flag which clearly identifies the *status* of the assertion. The status indicates whether the assertion refers to intended or actual properties, and possibly some additional information. Five different status are considered. We list them below, grouped according to who is usually the generator of such assertions:

- For assertions written by the user:

  `check` The assertion expresses an intended property. Note that the assertion may hold or not in the current version of the program.

  `trust` The assertion expresses a property which analysis can trust, i.e., it is taken as an actual property at compile-time. In contrast to status `true` introduced below, this information is given by the user and it may not be possible to infer it automatically. As a result, it makes sense to check them at run-time.

- For assertions which are results of static analyses:

  `true` The assertion expresses an actual property of the current version of the program. Such property has been automatically inferred.

- For assertions which are the result of static checking:

  `checked` A `check` assertion which expresses an intended property is rewritten with the status `checked` during compile-time checking (see Section 3.2) when such property is proved to actually hold in the current version of the program for any valid initial query.

  `false` Similarly, a `check` assertion is rewritten with the status `false` during compile-time checking when such property is proved not to hold in the current version of the program for some valid initial query. In addition, an error message will be issued by the preprocessor.

As already mentioned, all the assertions presented in the previous sections express intended properties and are assumed to be written by the user. Thus, they should have the status `check`. However, for pragmatic reasons, the status `check` is considered optional and if no status is given, `check` is assumed by default. For example, the assertion:

```
:- check success p(X) : ground(X).
```

can also be written "`:- success p(X) : ground(X).`"

Note also that the program-point assertions seen in Section 2.2.5 were introduced in the program as literals of the `check/1` predicate. This is because their status is `check`. If, however, we would like to add a program-point assertion with a different status we simply replace `check` by the corresponding status (`true`, `trust`, `checked` or `false`).

3 Compile-time assertion checking

In this section we recall basic compile-time assertion checking techniques, as well as discuss the recent improvements.

3.1 Evaluating assertions

As we have seen, schemes for predicate assertions have to be instantiated with a predicate descriptor *Pred* and one or two logic formulae on execution states, and the schema for program-point assertions also has to be instantiated with a logic formula about execution states.

We allow conjunctions and disjunctions in the formulae, and choose to write them down, for simplicity, in the usual CLP syntax. Such formulae have to be evaluated as part of the checking of an assertion. Evaluation of an assertion can be seen as composed of three steps. First, an appropriate inference system IS must be used to evaluate each of the atomic formulae AF of the assertion on the appropriate store θ. This presents a technical difficulty in the case of predicate assertions, since the formula is referred to the variables of the predicate descriptor $Pred$ in the assertion, whereas it has to be evaluated on a store θ which refers to variables different from those in *Pred*. We assume that a consistent renaming has been applied on the assertion, and thus on AF, so that it refers to the corresponding variables of θ. We denote by $\mathsf{eval}(AF, \theta, P, IS)$ the result of the evaluation of AF in θ by IS w.r.t. the definitions of property predicates P (i.e. the predicates that define abstract properties to be verified, see [47]). The inference system must be correct in the sense that if $\mathsf{eval}(AF, \theta, P, IS) = true$ then AF must actually hold in θ and if $\mathsf{eval}(AF, \theta, P, IS) = false$ then AF must actually do not hold in θ. However, we also allow incompleteness of IS, i.e., $\mathsf{eval}(AF, \theta, P, IS)$ does not necessarily return either *true* or *false*. If IS is not able to guarantee that AF holds nor that it does not hold in θ then it can return AF itself. Thus, if $\mathsf{eval}(AF, \theta, P, IS) = AF$ it can be interpreted as a "don't know" result.

The second step involves obtaining the truth value of the logic formulae app_A and sat_A as a whole from the results of the evaluation of each atomic formula. For this, standard simplification techniques for boolean expressions can be used. We denote by $simp(F)$ the result of simplifying a logic formula F. Since $\mathsf{eval}(AF, \theta, P, IS)$ may take the value AF for some atomic formulae in F, $simp(F)$ may take values different from *true* and *false*, which are not simplified further.

The third step corresponds to obtaining the truth value of the assertion as a whole from the values obtained for $simp(app_A)$ and $simp(sat_A)$. The assertion is proved to hold either if $simp(app_A) = false$ or $simp(sat_A) = true$. The assertion is proved not to hold if $simp(app_A) = true$ and $simp(sat_A) = false$. Once again, we may not be able to prove not to disprove the assertion if $simp(app_A)$ and/or $simp(sat_A)$ are not either *true* nor *false*. A program is correct for given valid queries if all its assertions have been proved for all the states that may appear in the computation of the program with the given queries (see [48] for a formal presentation of correctness and completeness w.r.t. these kinds of assertions).

In order to compute the value of $\mathsf{eval}(p(t_1, \ldots, t_n), \theta, P, IS)$ three cases are considered. The first one is that IS is *complete* w.r.t. p/n, i.e., it can always return either $true$ or $false$ for any store θ and any terms $t_1, \ldots, t_n$. The second case is when IS can return the value $true$ or the value $false$ for some store θ and terms $t_1, \ldots, t_n$ but not for all. In this case we say that IS *partially captures* the predicate p/n. This is usually based on sufficient conditions. The third case is when IS cannot return the value $true$ nor the value $false$ for any θ, i.e., IS *does not capture* (or it does not "understand") p/n.

Usually, given an inference system IS, there is a set of property predicates for which IS is *complete*. In addition, the user can often define other predicates for which IS is also complete by using some fixed and restricted syntax (consider, for example, defining a new type). The assertion language has to provide means to do this. Similarly, we call a predicate *provable* (resp. *disprovable*) in IS if IS can sometimes evaluate it to $true$ (resp. $false$).

The previous discussion assumes that the store on which the logic formulae are evaluated is given. This is feasible when assertions, and thus logic formulae, are evaluated at run-time, since the store θ is available. However, if static checking is being performed, only descriptions of stores and execution states rather than exact knowledge on such stores is available. There are two reasons for this. One is that at compile-time the actual values of the (valid) input data to the program are usually not available. The second one is that in order to ensure termination of static checking, some approximation of the actual computation must be performed which loses part of the information on the actual execution states.

In return for the loss of information introduced by static checking, static analysis systems often compute safe approximations of the stores reached during computation. This makes it possible under certain conditions to validate the program w.r.t. the assertions [26], since the results of analysis include all valid executions of the program. Thus, if a property can be proved in a safe approximation of a store θ then it is also proved to hold in θ; if it can be proved that it does not hold in the approximation of θ then it does not hold either in θ. This is done for example in the compile-time checking technique presented in Section 3.2.

3.2 Basic compile-time checking

We now show informally how the actual checking of the assertions at compile-time is performed by means of an example. Then, we briefly discuss on the technique used in the preprocessor for “reducing” (i.e., validating and detecting violations of) assertions. Details on how to reduce assertions at compile-time can be found in [48].

Example 3.1 *Assume that we have the following user-provided assertions:*

```
:- check calls p(X,Y): ground(X).
:- check success p(X,Y) => (list(X,int), list(Y,int)).
:- check comp p(X,Y): (list(X,int), var(Y)) + (does_not_fail,terminates).
```

The preprocessor is provided with links between specific properties and inference systems that can handle them. For example, it is known that `ground/1` *and* `var/1` *are captured by* `shfr` *(see [45] for the* `shfr` *sharing-freeness abstract domain), and* `list/2` *by* `eterms` *(see [52]).*

Assertion checking can be seen as computing the truth value of assertions by composing the value eval(AF, θ, P, IS) of the atomic properties AF at the corresponding stores θ reachable during execution. In the case of compile-time checking we must consider all possible stores reachable from any valid query. The abstract interpretation-based inference systems `shfr` and `eterms` compute a description (abstract substitution) for the calls and success states of each predicate (in fact they also do so for every program point). In compile-time checking we consider an *abstract evaluation* function *eval*$_\alpha(AF, \lambda, P, IS)$ in which the concrete store θ has been replaced by an abstract description λ. We denote by $\gamma(\lambda)$ the set of stores which a description λ represents. Correctness of abstract interpretation guarantees that $\gamma(\lambda)$ is a safe approximation of the set of all possible stores reached from valid initial queries, i.e., all such states are in $\gamma(\lambda)$.

Example 3.2 *After performing static analysis of the program (whose text we do not show as the discussion is independent of it) using* `shfr` *and* `eterms` *we obtain a description of the calls and success states for predicate* `p/2`. *For readability, we now show the results of such analyses in terms of assertions:*

```
:- true pred p(X,Y):(ground(X),var(Y)) => (ground(X),ground(Y)).
:- true pred p(X,Y):(list(X,int),term(Y)) => (list(X,int),int(Y)).
```

We denote by $\lambda^c(\mathrm{p}/2)$ and $\lambda^s(\mathrm{p}/2)$ the description of the calling and success states, respectively, of `p/2`. In our example, the static inference system `shfr` allows us to conclude that the evaluation of the three atomic properties $eval_\alpha($`ground(X)`$, \lambda^c(\mathrm{p}/2), P,$ `shfr`$)$, $eval_\alpha($`ground(Y)`$, \lambda^s(\mathrm{p}/2), P$ and $eval_\alpha($`var(Y)`$, \lambda^c(\mathrm{p}/2), P,$ `shfr`$)$ take the value $true$. Additionally, the `eterms` analysis determines that on success of `p/2`, i.e., in $\lambda^s(\mathrm{p}/2)$, the type of argument `Y` is `int`, which is a predefined type in `eterms`. This type is incompatible with `Y` being a list of integers, which is what was expected. Thus, $eval_\alpha($`list(Y,int)`$, \lambda^c(\mathrm{p}/2), P,$ `eterms`$)$ takes the value $false$. The implementation of $eval_\alpha$ in the preprocessor is based on the notion of *abstract executability* [50, 49].

Regarding the `non_failure` inference system, we assume it is implemented (as in [31]) as a program analysis which uses the results of the `shfr` and `eterms` analyses for approximating the calling patterns to each predicate, and then infers whether the program predicates with the given calling patterns might fail or not based on whether the type is recursively "covered".

The next step consists of composing and simplifying the truth value of each logic formula from the truth value computed by $eval_\alpha$.

Example 3.3 *After composing the results of evaluating each atomic property in the assertion formulae we obtain:*

```
:- check calls p(X,Y) : true.
:- check success p(X,Y) => (true, false).
:- check comp p(X,Y) : (true, true) +   (true, terminates).
```

We can now apply typical simplification of logical expressions and obtain:

```
:- check calls p(X,Y): true.
:- check success p(X,Y) => false.
:- check comp p(X,Y) : true + terminates.
```

The third and last step is to obtain, if possible, the truth value of the assertion as a whole. As assertion takes the value *true*, i.e., it is validated if either its precondition (more formally, the app_A formula of [47]) takes the value *false* (i.e., the assertion is never applicable) or if its postcondition (more formally, the sat_A formula of [47]) takes the value *true*. The postcondition of the first assertion of our example takes the value *true*. Thus, there is no need to consider such an assertion in run-time checking, and we can rewrite it with the tag `checked`. An assertion is violated if its precondition takes the value *true* and its postcondition takes the value *false*.[2] This happens to the second assertion in our example. Thus, we can rewrite it with the tag `false`. Whenever an assertion is detected to be false at compile-time, in addition to being rewritten with the `false` tag, the preprocessor also issues an error message. This allows the user to be aware of an incorrectness problem without looking at the assertions obtained by compile-time checking.

If it is not possible to modify the tag of an assertion, then such assertion is left as a `check` assertion, for which run-time checks might be generated. However, as the assertion may have been simplified, this allows reducing the number of properties which have to be checked at run-time.

Example 3.4 *The final result of compile-time checking of assertions is:*

```
:- checked calls p(X,Y) : ground(X).
:- false success p(X,Y) => (list(X,int), list(Y,int)).
:- check comp p(X,Y) + terminates.
```

where the third assertion still has the tag `check` *since it is not guaranteed to hold nor to be violated. Note also that the two assertions whose tag has changed appear as in the original version rather than the simplified one. The preprocessor does so as we believe it is more informative.*

3.3 Assertion checking using multivariant analysis

CiaoPP facilitates static multivariant analysis, which means that (as opposed to monovariant analysis) different call patterns for a given predicate can be analyzed separately. A goal dependent analysis (based on abstract interpretation) takes as input a program P, a predicate symbol p

[2]There is a caveat in this case due to the use of over-approximations in program analysis. It may be the case that a compile-time error is issued which does not occur at run-time for any valid input data. This is because though any activation of the predicate would be erroneous, it may also be the case that the predicate is never reached in any valid execution but analysis is not able to notice this. However, we believe that such situations do not happen so often and also the error flagged is actually an error of the program code. Though it can never show up in the current program it could do so if the erroneous part of the program is used in another context (in which it is actually used) in another program.

(denoting the entry point), and, optionally, a restriction of the run-time bindings of p expressed as an abstract substitution λ in the abstract domain D_α. Such an abstract interpretation computes a set of triples $Analysis(P, p, \lambda, D_\alpha) = \{\langle p_1, \lambda_1^c, \lambda_1^s\rangle, \ldots, \langle p_n, \lambda_n^c, \lambda_n^s\rangle\}$. 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^c are, respectively, the abstract call and success substitutions.

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. Note that if $n = 0$ then the corresponding predicate is not needed for solving any goal in the considered class (p, λ) and is thus dead code and may be eliminated. 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.

Let us illustrate the idea of multivariant analysis on the following trivial example.

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

q(X,X).
```

The monovariant type analysis returns:

```
:- true pred q(A,B): (rt6(A), term(B)) => (rt6(A), rt6(B)).

:- regtype rt6/1.
rt6(a). rt6(b).
```

Observe that both call patterns of `q/2` have been collapsed into a single one. The same has happened to the corresponding success patterns. In contrast to that, multivariant analysis makes distinction between the calls and successes and gives the following output:

```
:- true pred q(b,A): term(A) => tb(A).
:- true pred q(a,A): term(A) => ta(A).

:- regtype ta/1.
ta(a).
:- regtype tb/1.
tb(b).
```

Assertion checking for the multivariant case is performed in a quite similar way to the monovariant case. Finding the truth value of atomic properties must be now replaced by computing this value w.r.t. each and every triple containing a given atom p, in the analysis results. Let us use *eval*$'_\alpha$ to denote a new function, i.e. a function that evaluates AF w.r.t. all the triples including p.

Let A^p denote a set of all such triples. Thus evaluation of a atomic property AF is replaced by evaluating

$$\mathit{eval}'_\alpha(AF, P, IS) = \bigwedge_{\langle p,\lambda^c,\lambda^s\rangle \in A^p} \mathit{eval}_\alpha(AF, \lambda^c, P, IS) \tag{1}$$

for a call assertion or

$$\mathit{eval}'_\alpha(AF, P, IS) = \bigwedge_{\langle p,\lambda^c,\lambda^s\rangle \in A^p} \mathit{eval}_\alpha(AF, \lambda^s, P, IS) \tag{2}$$

for a success assertion.

where λ is either a call or success substitution, (depending whether we are interested in proving a call or success assertion) and *eval*$_\alpha$abstractly executes AF w.r.t. a single abstract substitution (likewise in the monovariant case). I.e. if *eval*$_\alpha$evaluates to *true* in every abstract substitution then the atomic property AF holds. If (1) (or (2)) becomes *false* then the property does not hold as we know for sure that at least one variant does not satisfy AF. Otherwise, the status of the assertion cannot be determined to be `true` nor `false` at compile time, and the value AF is returned by *eval*$'_\alpha$ to the assertion simplifier.

Assertion checking that uses multivariant analysis results is more accurate. This follows from the basic fact that the *lub* (least upper bound) operation in the abstract domain typically looses some precision, i.e. for any two abstract substitutions λ_1 and λ_2 we have $\gamma(\lambda_1 \sqcup \lambda_2) \supseteq \gamma(\lambda_1) \cup \gamma(\lambda_2)$. In order to demonstrate the advantage of using multivariant assertion checking we allow disjunctions in assertions. Consider the program from the previous example. and the assertion

```
:- check success q(A,B) => (ta(A),ta(B); tb(A),tb(B)).
:- regtype ta/1.
ta(a).
:- regtype tb/1.
tb(b).
```

With the monovariant analysis, as shown above, the assertion cannot be proven true (nor disproven). On the other hand applying multivariant analysis makes it possible to prove the assertion, as every disjunct becomes true wrt. different variant.

Multi-success case

Yet another form of multivariant analysis takes place when there are several success patterns assigned to a single call pattern. (There might be several call patterns to the same predicate, each of them generating multiple successes.) Consider the following program:

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

:-check success p(A,B) => (ta(A), ta(B)).
q(X,X).

:- regtype ta/1.
ta(a).
```

The type analysis without a multiple success feature would collapse the two possible success patterns and produce the following output:

```
:- true pred p(A,B): (term(A), term(B)) => (rt6(A), rt6(B)).
:- regtype rt6/1.
rt6(a). rt6(b).
```

whereas taking into account multiple successes results in:

```
:- true pred p(A,B): (term(A), term(B)) => (ta(A), tb(B)).
:- true pred p(A,B): (term(A), term(B)) => (tb(A), ta(B)).
:- regtype ta/1.
ta(a).
:- regtype tb/1.
tb(b).
```

Consequently, the multiple success analysis and corresponding assertion checking are able to detect that the assertion

```
:-check success p(A,B) => (ta(A), ta(B)).
```

is false. Formally, Whenever the expression

$$\bigwedge_{\langle p,\lambda_i^c,\lambda_i^s\rangle \in A^p} \quad \bigwedge_{\langle p,\lambda_j^c,\lambda_j^s\rangle \in A^p \wedge \lambda_j^c = \lambda_i^c} \mathit{eval}_\alpha(AF, \lambda_j^s, P, IS) \tag{3}$$

evaluates to *true*, the atomic property AF is found to hold. Observe that in order to decide if the property is true, it has to be proven wrt. each and every abstract success within every complete.

Disproving the atomic property wrt. multiple success information is a bit different. It has to be demonstrated that the property does not hold for any of the abstract successes within triples with the same call pattern. Thus, the evaluation of AF returns *false* if the expression

$$\bigwedge_{\langle p,\lambda_i^c,\lambda_i^s\rangle\in A^p} \quad \bigvee_{\langle p,\lambda_j^c,\lambda_j^s\rangle\in A^p\wedge\lambda_i^c=\lambda_j^c} \mathit{eval}_\alpha(AF,\lambda_j^s,P,IS) \tag{4}$$

evaluates to *false*. In summary:

$$\mathit{eval}'_\alpha(AF,P,IS)=\begin{cases} \mathit{true} & \text{iff} \quad \bigwedge_{\langle p,\lambda_i^c,\lambda_i^s\rangle\in A^p} \bigwedge_{\langle p,\lambda_j^c,\lambda_j^s\rangle\in A^p\wedge\lambda_j^c=\lambda_i^c} \mathit{eval}_\alpha(AF,\lambda_j^s,P,IS)=true \\ \mathit{false} & \text{iff} \quad \bigwedge_{\langle p,\lambda_i^c,\lambda_i^s\rangle\in A^p} \bigvee_{\langle p,\lambda_j^c,\lambda_j^s\rangle\in A^p\wedge\lambda_j^c=\lambda_i^c} \mathit{eval}_\alpha(AF,\lambda_j^s,P,IS)=false \\ AF & \text{otherwise} \end{cases}$$

3.4 Improved checking for success-with-precondition scheme

Assertions with the scheme: `:- success` *A: Pre* => *Post*, have the meaning: “whenever a call to the predicate A satisfies the precondition *Pre*, and A successfully terminates the postcondition *Post* holds”. So it is desirable to limit checking the postcondition to successes that follow calls satisfying the precondition, i.e. not any successes. A motivating example to make this case of assertion checking more expressive is the following program (assume `shfr` [45] as an analysis domain):

```
:- entry q(A,B).
:- success q(A,B): ground(A) => ground(B).
q(X,X).
```

with the intention to get the assertion `checked`, even though we do not know anything specific about arguments in the entry point. The key idea is to exploit the information contained in the precondition.

For simplicity of the presentation we assume that post- and precondtions are atomic. The description can be easily extended to the non-atomic case. In order to evaluate the assertion we abstractly unify Pre and λ^s

$$\Theta = \mathsf{unify}_\alpha(Pre,\lambda^s)$$

and then evaluate

$$\mathit{eval}_\alpha(Post,\lambda^s\Theta,P,IS)$$

The idea is to evaluate $Post$ not wrt. abstract success λ^s alone, but also with some information incorporated from the precondition Pre. In this way the checking can be more precise. There is however one problems with this approach. Namely, it works correctly only for properties which are *downwards closed*[3]. Otherwise wrong results might be obtained, as illustrated by the assertion

[3]A property is downwards closed iff whenever it holds for an atom it also holds for all its instances.

```
:- success q(A,B): var(A) => var(B).
```

Applying the above (naive) method would result in changing the assertion status to `checked`, which is incorrect as in the execution there might appear a call like `q(X,foo)` which satisfies the precondition and makes the postcondition not to hold for the corresponding success. To overcome this problem we have to compose the call substitution (the Θ substitution in our case) with the *topmost substitution* substitution (see [25]). We call an abstract substitution topmost wrt. a set of variables V iff $vars(\alpha) = V$, and $\alpha' \sqsubseteq \alpha$ for any other substitution α', s.t. $vars(\alpha') = V$.

So, the correct way to handle assertion `:- success` *A: Pre* `=>` *Post*, is find an abstract substitution Θ as following:

$$\Theta = \mathsf{unify}_\alpha(Pre, \lambda^s) \circ \alpha^\top$$

where $\alpha^\top$ is a topmost substitution for $vars(A)$ and $\circ$ is a substitution composition. Then we can proceed as before, that is, whenever we have to evaluate an atomic property in postcondition we apply *eval*$_\alpha$as the following:

$$\textit{eval}_\alpha(Post, \lambda^s\Theta, P, IS)$$

Let us return to the example. Now the applying the topmost substitution enforces variable `B` to take an abstract value “non-free”, rather than “free” (“free” corresponds to `var` property in the assertion), and thus the assertion remains not proven, i.e. it holds status `check`.

Let us illustrate the new assertion checking scheme with the two examples.

```
:- entry switch(A,B).
:- success switch(A,B): t_off(A) => t_on(B).
:- success switch(A,B): t_on(A) => t_off(B).
switch(on,off).
switch(off,on).

:- regtype t_on/1.
t_on(on).
:- regtype t_off/1.
t_off(off).
```

When running the type analysis (for instance `eterms`, see: [52]) with the multi-success option turned on (otherwise the two possible successes would collapse into one), as discussed in Section 3.3, we are able to verify both assertions to be `checked`. In fact, if the first argument of `switch/2` holds `off` on call, the second argument will take the value `on` upon success. This conditional cannot be easily verified otherwise.

The next program is to be analyzed with multivariant option, not necessarily the multi success one.

```
:- entry p(A,B).
:- success p(A,B): ta(A) => ta(B).
:- success p(A,B) => ta(B).

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

:- regtype ta/1.
ta(a).
:- regtype tb/1.
tb(b).
```

First assertion becomes `checked`, as indeed second argument of `p/2` takes value `a` on success, if the first one does so on call. The second assertion is found to be false, as the analysis finds one variant (a call - success pair) which does not satisfy the assertion.

An alternative way to deal with `:- success` *A: Pre* => *Post* assertions is to start a new analysis for a atom *A* and *Pre* as an entry point pattern. This option could be even more precise than our approach. Consider the standard `append/3` predicate:

```
append([],Ys,Ys).
append([H|Xs],Ys,[H|Zs]) :- append(Xs,Ys,Zs).
```

With our approach we are able to prove the following assertions:

```
:- success append(A,B,C): (ground(A),ground(B)) => ground(C).
:- success append(A,B,C): ground(C) => (ground(A),ground(B)).
```

but not the two ones below:

```
:- success app(A,B,C): (list(A),list(B)) => list(C).
:- success app(A,B,C): list(C) => (list(A),list(B)).
```

On the other hand, these assertions are shown true, if we run a type analysis with two entries that correspond to the preconditions in the assertions:

```
:- entry app(A,B,C) : (list(A), list(B)).
:- entry app(A,B,C) : list(C).
```

4 Sample session

Consider the following erroneous program that implements deletion elements from the list of numbers, using the predicate app/3 in two different modes.

```
:- entry del(A,B,C) : numlist(A).

:- pred del(A,B,C) : list(A) => list(C).                           % A1
del(L,E,R) :- app(L1,[L2|E],L), app(L1,L2,R).     % <-- a bug

:- calls app(A,B,C) : ((numlist(A), numlist(B)) ; numlist(C)).    % A2
:- success app(A,B,C) => (numlist(A), numlist(B), numlist(C)).    % A3
app([],Ys,Ys).
app([H|Xs],Ys,[H|Zs]) :- app(Xs,Ys,Zs).

:- regtype numlist/1.
numlist([]).
numlist([N|Ns]) :-
        num(N),
        numlist(Ns).
```

The definition of regular type numlist "list of numbers" is also included. There is a bug in the marked line; the first call to app/3 should be app(L1,[E|L2],L). For sake of simplicity we shall consider only assertion A3, and static checking at the predicate level. Let us first take a look on the output of assertion checking w.r.t. monovariant type analysis:

```
{WARNING (ctchecks_pred): Cannot verify assertion:
:- check success app(A,B,C)
        => ( del:numlist(A), del:numlist(B), del:numlist(C) ).

because on success del:app(A,B,C) :

[eterms] : del:numlist(A),rt32(B),rt33(C)
with: rt33 ::= num;[rt24|rt34]
      rt34 ::= [];num;[rt24|rt34]
      rt24 ::= num;[rt24|rt24]
      rt32 ::= num;[num|del:numlist]

Left to prove: del:numlist(B),del:numlist(C)
}
```

The assertion cannot be proven (verified). The user is also informed about the abstract success pattern found by the analyzer, and is given necessary type definitions. As it is shown, the `numlist(A)` part of the assertion has been proven. The rest is left to be proven. However, there is no incompatibility between types `rt32` and `numlist` (and `rt33` and `numlist`), i.e. corresponding type intersections are not empty. This is the reason why the assertion is not disproven.

Applying multivariant analysis gives more accurate answer:

```
{ERROR (ctchecks_pred): False assertion:
:- check success app(A,B,C)
        => ( del:numlist(A), del:numlist(B), del:numlist(C) ).

because on success del:app(A,B,C) :

[eterms] : del:numlist(A),basic_props:num(B),rt24(C)  OR
           del:numlist(A),rt37(B),rt37(C)  OR
           del:numlist(A),rt37(B),rt37(C)
with: rt37 ::= [num|del:numlist]
      rt24 ::= num;[rt24|rt24]
}
```

CiaoPP shows multivariant analysis output, where completes are separated by `OR`. The assertion is clearly disproven, as the variable `B` takes type `num` in the first complete, which is incompatible with the expected type `numlist`.

5 Conclusions

In this work we have described recent advances in Compile-time assertion checking in CiaoPP. We focused on improving (in terms of precision) predicate-level assertion checking. The proposed enhancements include exploiting multivariant analysis information and propagating information from precondition to postcondition in the "success-with-precodition" assertions. Both techniques are implemented in the recent version of CiaoPP. Our improvements require further evaluation and comparisons with other techniques. For example, exploiting multivariant analysis at the predicate-level assertion checking has to be carefully compared against program-point assertion checking and the monovariant analysis, in order to determine which one can capture more errors. Also, "success-with-precondition" assertions can be verified in the more accurate manner, as it is briefly discussed in the end of Section 3.4.

Part II

Integrating Ciao-Prolog Assertions into Abstract Interpretations

6 Motivation

When analysing a logic program, the source code for some parts of it may be inaccessible for some reason (external modules, builtin system predicates, foreign-language code, and so on). Thus there is a need to represent properties of external code, and incorporate the information in a static analysis. In this paper, an established notation for representing properties of logic program predicates is taken, namely the Ciao-Prolog assertion language.

The problem to be solved is to interpret assertions in that language in the context of a given specific abstract interpretation, rather than have to rewrite the assertions for each analysis domain. It usually requires considerable effort to specify the properties of builtins over each different abstract domain. However, without an abstract description of builtins one is forced to make coarse over-approximations.

The method described in this paper allows us to take any given assertions about a module's imported predicates and translate them safely into the domain under consideration. We use domains based on regular types, realised as pre-interpretations [40].

For instance, assertions about arithmetic predicates are provided in Ciao-Prolog. These assertions are given in terms of arithmetic objects such as integers and arithmetic expressions. If the analysis of interest concerns modes such as ground, non-ground and variable then a relation has to be established between arithmetic assertions and mode information.

An alternative approach is to define relationships between analysis domains in advance (a type lattice). This approach is currently followed in the Ciao assertion language. For example, the fact that arithmetic expressions are ground can be pre-defined. Once that is done, an assertion that the predicate $<$ succeeds with both arguments bound to arithmetic expressions can safely be translated into the modes domain as an assertion that both arguments are ground.

In contrast, the approach defined here allows arbitrary relationships to be derived automatically, for user-defined domains as well as pre-defined ones. We focus here on transforming assertions about success, but the same approach can be followed for assertions on calls.

7 Analysis Domains Based on Regular Types

The method described below covers the integration of assertions with abstract domains based on regular types. As summary of the main concepts is given here; a full account of such domains is available elsewhere [40].

7.1 Regular Types and Pre-interpretations

A *regular type* is defined by rules specifying a set of terms over some signature Σ. The rules are of the form $f(d_1, \ldots, d_n) \rightarrow d$, where $f/n \in \Sigma$ and $d, d_1, \ldots, d_n$ are type symbols. For example the type of lists is defined by the rules $[] \rightarrow list$, $[dynamic|list] \rightarrow list$ (where $dynamic$ is the type of all terms). A (set of) regular type definitions can be regarded as a finite tree automaton (FTA). For our purposes we regard the two notions as interchangeable, and speak of the states of an FTA as "types". It is known [29] that an arbitrary FTA can be transformed to an equivalent *bottom-up deterministic* FTA (or DFTA). An arbitrary FTA can also be *completed*, meaning that it is extended with rules that handle any term over Σ.

A *pre-interpretation* J of a signature Σ is defined by a domain D_J and a mapping I_J mapping each n-ary function symbol $f/n \in \Sigma$ to a function $D_J^n \rightarrow D_J$, denoted f_J. I_J is extended to interpret terms in Term_Σ, such that for a ground term $f(t_1, \ldots, t_n) \in \mathsf{Term}_\Sigma$, $I_J(f(t_1, \ldots, t_n)) = f_J(I_J(t_1), \ldots, I_J(t_n))$.

An element $d \in D_J$ denotes a set of terms $\gamma(d) \subseteq \mathsf{Term}_\Sigma$, namely $\gamma(d) = \{t \mid I_J(t) = d\}$. The mapping γ extends to atoms whose arguments are elements of D_j, and to sets of atoms, where $\gamma(p(d_1, \ldots, d_n)) = \{p(t_1, \ldots, t_n) \mid t_i \in \gamma(d_i), 1 \leq i \leq n\}$, and $\gamma(S) = \bigcup\{\gamma(s) \mid s \in S\}$.

It was shown in [40] that a completed DFTA is equivalent to a *pre-interpretation* of Σ. Thus when we speak of a pre-interpretation we could equally well refer to a completed DFTA and vice versa. Each rule $f(d_1, \ldots, d_n) \rightarrow d$ in the DFTA corresponds to an equation $f_J(d_1, \ldots, d_n) = d$ in the pre-interpretation J. The set of rules with the same function f/n on the left defines the function f_J onto which f/n is mapped by I_J. The DFTA is complete, hence the function f_J is completely defined.

Each state d in an FTA is split into a set of disjoint states $\{d'_1, \ldots, d'_k\}$ in the corresponding DFTA. Hence we can define $\gamma(d)$ where d is a state in an arbitrary FTA, as $\bigcup\{\gamma(d'_1), \ldots, \gamma(d'_k)\}$, and thus also extend γ to to atoms and to sets of atoms as above.

The least model of a program P over a pre-interpretation J can be obtained by a fixpoint computation. The obtained model $M_J[P]$ is an abstraction of the least Herbrand model $M[P]$, in the sense that $\gamma(M_J[P]) \supseteq M[P]$.

7.2 Approximations of success sets

The model of a predicate p/n over an FTA with states (types) D is a set M_p of "domain facts" of the form $p(d_1, \ldots, d_n)$, where $d_1, \ldots, d_n \in D$. Given a program P containing some externally defined predicates, we can specify a model for an external predicate for a pre-interpretation J.

A model for a predicate p/n should be an abstraction of the intended interpretation of p/n. That is, $\gamma(M_p)$ should include the intended interpretation of p/n.

Example 7.1 *Let the domain of a pre-interpretation be* $\{\mathsf{g}, \mathsf{ng}\}$ *standing for* ground *terms and* non-ground *terms respectively. Suitable models for the predicates* < *and* `=..` *are* $\{$`g < g`$\}$ *and* $\{$`g =.. g, ng =.. ng`$\}$ *respectively.*

Let the domain of a pre-interpretation be $\{\mathsf{l}, \mathsf{nl}\}$ *standing for* list *terms and* non-list *terms respectively. Suitable models for the same predicates are* $\{$`nl < nl`$\}$ *and* $\{$`l =.. l, nl =.. l`$\}$ *respectively.*

8 Overview of the Procedure

The procedure described below takes as input the following, of which the first two are program-specific and the rest are a permanent part of the Ciao assertion database. This information is extracted automatically in our implementation.

1. a program module P, having a set E_P of zero or more external predicates;
2. a set of *user types* T_u, specified as an NFTA F_u;
3. a set of *defined system types* T_s, specified as an NFTA F_s;
4. a set of *primitive types* T_{prim};
5. a set of *contextual types* T_{cntxt};
6. a model of the success set for each predicate in E_P, over the *system types* $T_{sys} = T_s \cup T_{prim} \cup T_{cntxt}$.

The output of the procedure is a model of the predicates in E_P, over the set of types defined in T_u.

We define the set of types $T = T_P \cup T_u \cup T_s \cup T_{prim} \cup T_{cntxt} \cup \{dynamic\}$. The extra type $dynamic$ is assumed not to occur in the other sets of types. An example of a defined system type in T_s is $arithexpr$ defining the set of arithmetic expressions. Examples of primitive types in T_{prim} are int, $nnegint$, $string$ and such like. We assume that primitive types define constants (that is, no compound term has a primitive type). Such types cannot conveniently be written down as NFTAs because they contain an infinite (or very large) number of constants. Instead, there are procedures for checking whether a given constant is of a given primitive type.

The contextual types T_{cntxt} depend on the signature of the program under consideration; such types are gnd (ground terms), $term$ (all terms) and $struct$ (all non-atomic terms). The details of T_s, F_s and Σ_s are given in Section 11.3. T_{prim} is given in Section 11.5, and T_{cntxt} in Section 11.6.

The same type is allowed to occur in different components of T. So for example a system type such as $list$ might also be one of the user types.

8.1 The Global Signature

Let Σ_P be the set of function symbols occurring in P, Σ_u be the set of function symbols occurring in F_u, and Σ_s be the set of function symbols occurring in F_s. In addition there is a *primitive signature* Σ_{prim}, disjoint from Σ_P, Σ_u and Σ_s, which contains sufficient constants to distinguish each of the primitive types T_{prim}. More precisely, for each non-empty subset $D = \{d_1, \ldots, d_k\}$ of T_{prim}, let Σ_D be the set of constants that are of type d_i for all $d_i \in D$, and are not of any other type. Then Σ_{prim} contains at least one constant from each non-empty set Σ_D.

For instance, if there are primitive types nat and $number$ such that all constants of type nat are of type $number$ but not vice versa, then Σ_{prim} contains at least one constant that is both nat and $number$, and one that is of type $number$ but not nat.

We also assume that Σ_{prim} contains a constant, say v, that is not of any primitive type and does not appear in any of the other signatures. The full definition of T_{prim} and Σ_{prim} is given in Section 11.5.

Let the *global signature* $\Sigma = \Sigma_P \cup \Sigma_u \cup \Sigma_s \cup \Sigma_{prim}$.

8.2 Constructing the primitive and contextual type definitions

We assume that the Prolog system provides some procedure, such as a builtin predicate, for testing whether a given constant is of a given primitive type. Hence given a signature Σ we can enumerate the set of rules $F_{prim}$of the form $c \rightarrow d$ where $c \in \Sigma$ is a constant, $d \in T_{prim}$ and c is of type d.

The types in T_{cntxt} are those whose definitions depend on the signature, such as gnd (ground terms). Given the global signature Σ the NFTA F_{cntxt} is a set of rules defining each type in T_{cntxt} in terms of $\Sigma \setminus \{v\}$. For instance, the type gnd is defined by the set of rules $f(gnd, \ldots, gnd) \rightarrow$

gnd, for each n-ary function $f \in \Sigma \setminus \{v\}$. In Section 11.6 the detailed specifications of the contextual types are shown.

We define F_{dyn} to consist of the rules $f(dynamic, \ldots, dynamic) \to dynamic$, for each n-ary function $f \in \Sigma$. Note that the extra constant v is of type $dynamic$, but not of any other type except the system type $term$.

Having constructed the primitive and contextual types, the global NFTA $F = F_u \cup F_s \cup F_{prim} \cup F_{cntxt} \cup F_{dyn}$.

8.3 The global type system

The global signature Σ, the complete set of types T and their definitions F, together define the global type system. We can then apply the *determinization* algorithm on finite tree automata [29]. The output is a DFTA, in which the set of states T' is a subset of 2^T. In the worst case, the set of states in the determinized automaton explodes, but in practice the size of T' is usually of the same order of magnitude as the size of T (and can even be smaller).

In the following procedure we do not need to consider the set of determinized rules; we need only use the determinized types T'.

8.4 Converting the models of external predicates

As mentioned, we are supplied with a model of the success set of each of the external predicates in E_P. (If no model is supplied for some predicate p/n, we assume that its success set contains all possible atoms $p(d_1, \ldots, d_n)$, where $d_1, \ldots, d_n \in T$.)

We first define $dettypes(d)$, the set of determinized types in T' corresponding to a type $d \in T$. Recall that elements of T' are sets of elements of T. Define $dettypes(d) = \{d' \mid d \in T', d \in d'\}$, which is the set of all the determinized types that comprise d. In other words, $dettypes(d)$ is a partition of d into non-empty disjoint types in T'.

Let $p/n \in E_P$ and let its model over T_s be M_p. Then the corresponding model, defined over the set of determinized types T', is defined as $M'_p = \{p(d'_1, \ldots, d'_n) \mid p(d_1, \ldots, d_n) \in M_p \wedge \forall i : 1 \leq i \leq n[d_i \in dettypes(d_i)]\}$.

That is, the model of p over the global types contains an atom $p(d'_1, \ldots, d'_n)$ iff there exists an atom $p(d_1, \ldots, d_n) \in M_p$ such that each of the arguments d'_i is a deteminized component of d_i.

Example 8.1 *Given the model of $<$ as $\{gng < gnd\}$, and the set of determinized types T',* *namely*

$$\{\{gnd, list, nonvar, dynamic\}, \{gnd, nonvar, dynamic\},$$
$$\{nonvar, dynamic\}, \{list, nonvar, dynamic\}\}$$

then the model of $<$ *over* T' *is*

$$\{\{gnd, list, nonvar, dynamic\} < \{gnd, list, nonvar, dynamic\},$$
$$\{gnd, list, nonvar, dynamic\} < \{gnd, nonvar, dynamic\},$$
$$\{gnd, nonvar, dynamic\} < \{gnd, list, nonvar, dynamic\},$$
$$\{gnd, nonvar, dynamic\} < \{gnd, nonvar, dynamic\}\}$$

8.5 Projecting a model onto user types

The next stage is to project the model over the determinized types onto the user types. Let $p \in E_P$ be an external predicate and let M'_p be the model of p over the determinized types T'. Then the projection of M'_p onto the user types T_u is defined as $M^u_p = \{p((d'_1 \cap T^d_u), \ldots, (d'_n \cap T^d_u)) \mid p(d'_1, \ldots, d'_n) \in M'_p\}$, where $T^d_u = T_u \cup \{dynamic\}$. Note that each argument $d'_1 \cap T^d_u$ is non-empty since $dynamic$ is in every element of d'_i and of T^d_u.

Note that the arguments of atoms in M^u_p are not in general elements of T'; rather, they are subsets of T_u.

Example 8.2 *Let* $M'_<$ *be the model of* $<$ *over* T' *as in the previous example. Let* $T_u = \{list\}$. *Then the projection of* $M'_<$ *onto* T_u *is*

$$\{\{list, dynamic\} < \{list, dynamic\},$$
$$\{list, dynamic\} < \{dynamic\},$$
$$\{dynamic\} < \{list, dynamic\},$$
$$\{dynamic\} < \{dynamic\}\}$$

The projected model is not expressed directly in the set of user types T_u but rather in the disjoint types resulting from determinizing $T_u \cup \{dynamic\}$. The model expressed in this form is exactly what is required for computing a model of the program P over the user types.

9 Soundness

The projected models are safe approximations of the models over the system types.

Proposition 9.1 *Let* P *be a program and let* p *be an externally defined predicate occurring in* P. *Let* M_p *be the model of* p *over the system types* T_s *and* M^u_p *be the model projected onto the user types* $T_u \cup \{dynamic\}$. *Then* $\gamma(M_p) \subseteq \gamma(M^u_p)$, *where* γ *is evaluated over the global signature*

Proof 1 *(Sketch.)* $\gamma(M_p) = \gamma(M'_p)$ *by definition of* γ*. Let* $p(t_1, \ldots, t_n) \in \gamma(M'_p)$*. Then there exists some* $p(d'_1, \ldots, d'_n) \in M'_p$ *such that* $t_i \in \gamma(d'_i)$ *for all* $1 \leq i \leq n$*. For all* $1 \leq i \leq n$*,* $d'_i \cap T^d_u \neq \emptyset$ *since* $dynamic$ *is in every element of* T' *and of* T^d_u*. Hence* $p((d'_1 \cap T^d_u), \ldots, (d'_n \cap T^d_u)) \in M^u_p$ *and hence* $p(t_1, \ldots, t_n) \in \gamma(M^u_p)$*. Hence* $\gamma(M_p) \subseteq \gamma(M^u_p)$*.*

10 Implementation and Experiments

The procedure was implemented in the context of the Ciao-Prolog assertion language. The input to the procedure is a Ciao-Prolog module and a set of user type definitions. The Ciao-Prolog Preprocessor `ciaopp` provides an interface to the assertions stored for each imported predicate. The implementation picks out the success assertions (from among the many different kinds of assertion in `ciaopp`).

The initial uses of the procedure are to generate tables of builtin assertions for mode and binding type analyses, and backwards analyses. The implemented procedure is used off-line, generating a table which is then incorporated into analysis tools based on pre-interpretations [40].

We have successfully generated assertions on builtin success over the domain of binding types used in the LOGEN automatic Binding Time Analysis [30]. The standard BTA domain consists of the types $static$, $dynamic$, var, $nonvar$, $list$. Note that the builtin models obtained from the Ciao-Prolog assertions are not necessarily the most precise possible for the given domain. This can be either because the stored assertions are not precise enough, or because information was lost when projecting onto the user types. In some cases the derivation of the success model suggests improvements in the assertions stored for builtins.

11 Detailed Example

In this section we illustrate the process by showing the various components of the analysis for an example. For the purposes of illustration we show a small module containing various external predicates ==, \==, `=..`, `var`, `atomic` and `nonvar`.

```
:- module(vars, [vars/2]).

vars(T,Vs) :-
        vars3(T,[],Vs).

vars3(X,Vs,Vs1) :-
```

```
         var(X),
         insertvar(X,Vs,Vs1).
vars3(X,Vs,Vs) :-
         atomic(X).
vars3(X,Vs,Vs1) :-
         nonvar(X),
         X =.. [_|Args],
         argvars(Args,Vs,Vs1).

argvars([],Q,Q).
argvars([X|Xs],Vs,Vs2) :-
         vars3(X,Vs,Vs1),
         argvars(Xs,Vs1,Vs2).

insertvar(X,[],[X]).
insertvar(X,[Y|Vs],[Y|Vs]) :-
         X == Y.
insertvar(X,[Y|Vs],[Y|Vs1]) :-
         X \== Y,
         insertvar(X,Vs,Vs1).
```

11.1 Program Signature

The program manipulates only list structures and its signature is

```
./2, []/0
```

11.2 User Type Signature

We first analyse with respect to the binding types $static$ and $dynamic$. The user type signature for these types is empty, since these types are defined contextually over the global signature. Secondly we analyse over the user type $list$, for which the signature is

```
./2, []/0
```

11.3 System Types

11.3.1 System Type Signature

The following functors appear in the system type definitions from the Ciao-Prolog assertion database. In practice only those system types that depend on some type in the model of the predicates appearing in the module need to be used, but the complete list is shown here.

```
no/0,yes/0,datime/6,$stream/2,user/0,user_error/0,
user_output/0, user_input/0, lock_nb/1,lock/1,
lock_nb/0,lock/0,exclusive/0, shared/0,append/0,
write/0,read/0, dictionary/1,lines/2,singletons/1,
variable_names/1,variables/1,walltime/0,
systemtime/0,usertime/0, runtime/0,choice/0,
trail/0,local_stack/0,global_stack/0, program/0,
symbols/0,memory/0,stack_shifts/0,
garbage_collection/0, wallclockfreq/0,
systemclockfreq/0,userclockfreq/0,wallclick/0,
systemclick/0,userclick/0,runclick/0,$ref/2,xf/0,
yf/0,xfx/0,xfy/0,yfx/0, fx/0,fy/0,gcd/2,atan/1,cos/1,
sin/1,sqrt/1,log/1,exp/1,# /2,(\)/1,\/ /2,/\ /2, << /2,
>> /2,** /2,ceiling/1,round/1,floor/1,float/1,truncate/1,
integer/1, float_fractional_part/1,float_integer_part/1,
sign/1,abs/1,mod/2,rem/2, / /2,// /2,* /2,(-)/2,(+)/2,
(++)/1,(--)/1,(+)/1,(-)/1
```

11.4 Definitions of System Types

The module names of the types have been omitted here to make the definitions more readable. For example, `arithexpr` should actually be written in full as `arithmetic:arithexpr`.

```
(basictype(num)->arithexpression),
(-arithexpression->arithexpression),
(+arithexpression->arithexpression),
(--arithexpression->arithexpression),
(++arithexpression->arithexpression),
(arithexpression+arithexpression->arithexpression),
(arithexpression-arithexpression->arithexpression),
(arithexpression*arithexpression->arithexpression),
(arithexpression//arithexpression->arithexpression),
(arithexpression/arithexpression->arithexpression),
(arithexpression rem arithexpression->arithexpression),
(arithexpression mod arithexpression->arithexpression),
(abs(arithexpression)->arithexpression),
(sign(arithexpression)->arithexpression),
(float_integer_part(arithexpression)->arithexpression),
(float_fractional_part(arithexpression)->arithexpression),
(integer(arithexpression)->arithexpression),
(truncate(arithexpression)->arithexpression),
(float(arithexpression)->arithexpression),
(floor(arithexpression)->arithexpression),
(round(arithexpression)->arithexpression),
(ceiling(arithexpression)->arithexpression),
(arithexpression**arithexpression->arithexpression),
(arithexpression>>arithexpression->arithexpression),
(arithexpression<<arithexpression->arithexpression),
(arithexpression/\arithexpression->arithexpression),
(arithexpression\/arithexpression->arithexpression),
(arithexpression->arithexpression),
(arithexpression\#arithexpression->arithexpression),
(exp(arithexpression)->arithexpression),
(log(arithexpression)->arithexpression),
(sqrt(arithexpression)->arithexpression),
(sin(arithexpression)->arithexpression),
(cos(arithexpression)->arithexpression),
(atan(arithexpression)->arithexpression),
(gcd(arithexpression,arithexpression)->arithexpression),
([num|niltype]->arithexpression),
([]->niltype),

(basictype(atm)->atm),

(basictype(atm)->atm_or_atm_list),
([atm|pt2]->atm_or_atm_list),
([]->pt2),
([atm|pt2]->pt2),

(basictype(atm)->callable),
(basictype(struct)->callable),

(basictype(int)->character_code),

(basictype(atm)->constant),
(basictype(num)->constant),

(basictype(flt)->flt),

(basictype(gnd)->gnd),

(basictype(int)->int),

([]->list),
([term|list]->list),

(basictype(nnegint)->nnegint),

(basictype(num)->num),

(fy->operator_specifier),
(fx->operator_specifier),
(yfx->operator_specifier)
(xfy->operator_specifier),
(xfx->operator_specifier),
(yf->operator_specifier),
(xf->operator_specifier),

(atm/nnegint->predname)

([]->string),
([character_code|pt1]->string),

([]->pt1),
([character_code|pt1]->pt1),

(basictype(struct)->struct),

(basictype(term)->term),

($ref(term,term)->reference),

([character_code|pt1]->format_control),
(basictype(atm)->format_control),

(runclick->click_option),
(userclick->click_option),
(systemclick->click_option),
(wallclick->click_option),

([num|newtype0]->click_result),

([num|niltype]->newtype0),
([]->niltype),

(userclockfreq->clockfreq_option),
(systemclockfreq->clockfreq_option),
(wallclockfreq->clockfreq_option),
(basictype(num)->clockfreq_result),

(garbage_collection->garbage_collection_option),
(stack_shifts->garbage_collection_option),

([int|newtype1]->gc_result),

([int|newtype2]->newtype1),
([int|niltype]->newtype2),
([]->niltype),

(memory->memory_option),
(symbols->memory_option),
(program->memory_option),
(global_stack->memory_option),
(local_stack->memory_option),
(trail->memory_option),
(choice->memory_option),

([int|newtype3]->memory_result),

([int|niltype]->newtype3),
([]->niltype),

(symbols->symbol_option),

([int|newtype4]->symbol_result),

([int|niltype]->newtype4),
([]->niltype),

(runtime->time_option),
(usertime->time_option),
(systemtime->time_option),
(walltime->time_option),

([num|newtype5]->time_result),

([num|niltype]->newtype5),
([]->niltype),

(variables(term)->read_option),
(variable_names(term)->read_option),
(singletons(term)->read_option),
(lines(term,term)->read_option),
(dictionary(term)->read_option),

([]->keylist),
([newtype6|keylist]->keylist),

(term-term->newtype6),
(term-term->keypair),

(read->io_mode),
(write->io_mode),
(append->io_mode),

(read->lock_mode),
(shared->lock_mode),
(write->lock_mode),
(exclusive->lock_mode),

(lock->open_option),
(lock_nb->open_option),
(lock(lock_mode)->open_option),
(lock_nb(lock_mode)->open_option),

([]->open_option_list),
([open_option|pt0]->open_option_list),

([]->pt0),
([open_option|pt0]->pt0),

(basictype(atm)->sourcename),
(basictype(struct)->sourcename),

(user_input->stream),
(user_output->stream),
(user_error->stream),
(user->stream),
($stream(int,int)->stream),
(user_input->stream_alias),
(user_output->stream_alias),
(user_error->stream_alias),

(datime(int,int,int,int,int,int)->datime_struct),

(read->popen_mode),
(write->popen_mode),

(yes->yesno),
(no->yesno),
```

11.5 Definitions of Primitive Types

The following primitive types are defined in the Ciao assertion language.

```
nnegint,  flt,  int, atm, num.
```

Each 0-ary functor in the global signature is tested against the primitive types, and a rule `(c -> primitive_type)` is produced whenever a constant `c` is of type `primitive_type`.

```
(no->atm),(yes->atm),(user->atm),(user_error->atm),
(user_output->atm),(user_input->atm),(lock_nb->atm),
(lock->atm), (exclusive->atm),(shared->atm),
(append->atm),(write->atm), (read->atm),(walltime->atm),
(systemtime->atm),(usertime->atm), (runtime->atm),
(choice->atm),(trail->atm),(local_stack->atm),
(global_stack->atm),(program->atm),
(symbols->atm), (memory->atm),(stack_shifts->atm),
(garbage_collection->atm), (wallclockfreq->atm),
(systemclockfreq->atm),(userclockfreq->atm), (wallclick->atm),
(systemclick->atm),(userclick->atm),(runclick->atm), (xf->atm),
(yf->atm),(xfx->atm),(xfy->atm),(yfx->atm),(fx->atm),
(fy->atm),(0->nnegint),(0->int),(0->num),(1->nnegint),
(1->int),(1->num), (1.0->flt),(1.0->num),(-1->int),
(-1->num),($CONST->atm),([]->atm)
```

11.6 Contextual Type Definitions

The following contextual types are defined in the system: `struct`, `term`, `gnd`. In addition the type $dynamic$ is added to the set of types.

`term` is defined by a set of rules of form `f(term,...,term) -> term` for each function symbol `f` (including the special constant v) in the signature.

`struct` is defined by a set of rules of form `f(term,...,term) -> struct` for each function symbol `f` of arity greater than zero in the signature.

`gnd` is defined by a set of rules of form `f(gnd,...,gnd) -> gnd` for each function symbol `f` (except the special constant v) in the signature.

`dynamic` is equivalent to `term`. `dynamic` is included only to ensure completeness.

User-defined contextual types, like `static` or `nonvar`, can be defined. Note that identical types are merged during determinzation, so redundant results do not arise.

11.7 Success Model over System Types

Now consider the projection of the external predicates in the `vars` module shown at the start of Section 11.

The builtin predicates `==`, `\==`, `=..`, `var`, `atomic` and `nonvar` have the following success model over the system types.

```
term\==term, term==term, term=..list,
dynamic, atomic(atom_or_number), var(term)
```

11.8 Success Model over User Types

First suppose that the user types are `static` and `dynamic`. The derived success model is then as follows.

```
var(_), atomic([dynamic,static]), nonvar(_), _=..[dynamic],
_=..[dynamic,static]), _==_, _\==_
```

Note that here `[dynamic,static]` means *static* and `[dynamic]` means *non-static*. The anonymous argument _ stands for either of the types `[dynamic]` or `[dynamic,static]`.

When analysed over the domain `list` and `dynamic`, the derived success model is the following.

```
var(_), atomic([dynamic,list]), atomic([dynamic]), nonvar(_),
_=..[dynamic,list], _==_, _\==_
```

Note that here `[dynamic,list]` means *list* and `[dynamic]` means *non-list*. As before, the anonymous argument _ stands for either type.

Finally consider projecting onto a user-defined type given as follows:

```
[] -> niltype.
[dynamic|niltype] -> nonemptylist.
[dynamic|nonemptylist] -> nonemptylist.
```

The following projected model is then derived.

```
var(_), atomic([dynamic,niltype]), atomic([dynamic]), nonvar(_),
_=..[dynamic,niltype], _=..[dynamic,nonemptylist], _==_, _\==_
```

Note that, for example, that `atomic` cannot succeed with a `nonemptylist`.

12 Conclusion

The problem of obtaining abstractions of builtins is often a stumbling block to analysing real application programs accurately. It requires considerable effort to specify the properties of builtins over each different abstract domain. Without such an effort though, one is forced to make coarse over-approximations of the builtins.

The method described above allows the existing, permanent assertions for the builtins over a rich set of system types to be transported automatically to any given domain defined as a pre-interpretation. Such domains can be constructed from arbitrary regular types.

The method was based on building a complete signature of the program and all its types. Some of the types are contextual, that is, defined in terms of a particular signature. Once the compete types are obtained, they are determinized and then the predicate models are projected from the determinized type domain onto the types of interest.

Apart from abstractions of builtins, the method allows the analysis results for modules that have been analysed over one domain to be imported into modules which are being analysed over a different domain. This adds to the flexibility and practicability of module-based analyses.

The method has been fully implemented and run on modules including those used to implement the procedure itself.

Part III
Experiments with Backwards Analysis

13 Motivation

In this work we report an implementation of a novel backwards analysis technique [38] and experiments using the analysis in the context of the Ciao-Prolog module system.

The specific problem tackled in our experiments is to analyse a module, which might contain one or more calls to predicates that can potentially produce runtime errors. Such predicates are usually system predicates, or “builtins”, which are required to be called with their arguments instantiated in certain ways.

Other uses of backwards analysis, discussed by King and Lu [42] and Genaim and Codish [41], include deriving conditions that guarantee termination of calls to given predicates.

The range of applications of backwards analysis has a common core: the input to backwards analysis in general is a program together with properties that are required to hold at given program points. The purpose of the analysis is to derive initial goals that guarantee that, when those goals are executed, the given properties hold.

Thus the experiments reported here can be seen as a component in a more general setting where arbitrary assertions can be stated at given program points, and we wish to derive conditions that guarantee that they hold at execution time. Stated in this general way, the problem is clearly an old one in computing; the weakest precondition calculus can be seen as the original backwards analysis framework, since the aim there is to derive preconditions that guarantee that given assertions hold at given program points. The key feature in our approach is the fact that we use approximation semantics to derive the preconditions. The aim of our experiments is to confirm the practicality of the method for deriving conditions that guarantee avoidance of runtime errors, with the longer term aim of using the method to tackle a range of properties related to security and reliability of software.

The works [42, 41] cited above require special abstract interpretation frameworks. The approach described in our method [38], by contrast, uses a standard abstract interpretation framework that computes over-approximations of the semantics.

This might be surprising in view of the fact that the final result of a backwards analysis, namely (a description of) the set of initial goals that guarantee the establishment of the given properties, should be an *under* approximation of the actual set of goals that satisfy the requirements. This is what has led investigators to develop special abstract interpretations that give an under approximation.

14 Outline of the backwards analysis method

A number of properties that are required to hold at body atoms at specific program points are identified. A *meta-program* is then automatically constructed, which captures the dependencies between initial goals and the specified program points. This meta-program is based on the *resultants* semantics of logic programs [36, 35], in which the meaning of a program is the set of all pairs (A, R) where $A = A'\theta$ and there is an LD derivation from $\leftarrow A'$ to $\leftarrow R$ with computed answer θ. An abstraction of the resultants semantics is then defined, containing all pairs (A, B) such that $A = A'\theta$ and there is an LD derivation from $\leftarrow A'$ to $\leftarrow B, B_1, \ldots, B_m$ with computed answer θ, where B corresponds to one of the specified program points. (This semantics is closely related to the binary clause semantics defined by Codish and Taboch [28]). The semantics is captured by a meta-program defining a meta-predicate `d/2`, such that `d(A,B)` is a consequence of the meta-program whenever a pair (A, B) as defined above exists. Standard abstract interpretation techniques are applied to the meta-program; from the results of the analysis, conditions on initial goals can be derived which guarantee that all the given properties hold whenever the specified program points are reached.

15 Implementation

Our backwards analysis tool has the following components.

1. A module for extracting assertions from the Ciao-Prolog assertion database, and identifying the program points at which they should hold.

2. A transformation to the meta-program form outlined above and specified precisely in [38].

3. Conversion of the stated assertions into a given pre-interpretation, as described in Part II. This conversion also transforms any success assertions on external predicates into the given pre-interpretation.

4. Analysis of the meta-program over the domain specified by the pre-interpretation. This is a general-purpose abstract interpretation using the approach defined in our previous work [40].

5. Processing of the abstract model of the meta-program, in order to obtain the preconditions that guarantee the program-point assertions. This step is fully described in [38].

Steps 3 and 4 are performed by standard tools that apply to both forwards and backwards analysis. The analysis becomes “backwards” due to the use of the meta-program that expresses the dependencies between initial goals and program points, and the final inspection of the model of the meta-program.

16 Experiments

We focus in our initial experiments on arithmetic predicates. Success models for predicates `is`, >, <, >=, =< are obtained from the Ciao-Prolog database, in terms of the system type `arithmetic:arithexpr`. The definition of this type is also obtained from the assertion database, and is as follows.

```
0->'arithmetic:arithexpression'.
1->'arithmetic:arithexpression'.
1.0->'arithmetic:arithexpression'.
-1->'arithmetic:arithexpression'.
-'arithmetic:arithexpression'->'arithmetic:arithexpression'.
+'arithmetic:arithexpression'->'arithmetic:arithexpression'.
--'arithmetic:arithexpression'->'arithmetic:arithexpression'.
++'arithmetic:arithexpression'->'arithmetic:arithexpression'.
'arithmetic:arithexpression'+'arithmetic:arithexpression'->
   'arithmetic:arithexpression'.
'arithmetic:arithexpression'-'arithmetic:arithexpression'->
   'arithmetic:arithexpression'.
'arithmetic:arithexpression'*'arithmetic:arithexpression'->
   'arithmetic:arithexpression'.
'arithmetic:arithexpression'//'arithmetic:arithexpression'->
   'arithmetic:arithexpression'.
'arithmetic:arithexpression'/'arithmetic:arithexpression'->
   'arithmetic:arithexpression'.
'arithmetic:arithexpression' rem 'arithmetic:arithexpression'->
   'arithmetic:arithexpression'.
'arithmetic:arithexpression' mod 'arithmetic:arithexpression'->
   'arithmetic:arithexpression'.
abs('arithmetic:arithexpression')->'arithmetic:arithexpression'.
```

```
sign('arithmetic:arithexpression')->'arithmetic:arithexpression'.
float_integer_part('arithmetic:arithexpression')->
   'arithmetic:arithexpression'.
float_fractional_part('arithmetic:arithexpression')->
   'arithmetic:arithexpression'.
integer('arithmetic:arithexpression')->
   'arithmetic:arithexpression'.
truncate('arithmetic:arithexpression')->
   'arithmetic:arithexpression'.
float('arithmetic:arithexpression')->
   'arithmetic:arithexpression'.
floor('arithmetic:arithexpression')->
   'arithmetic:arithexpression'.
round('arithmetic:arithexpression')->
   'arithmetic:arithexpression'.
ceiling('arithmetic:arithexpression')->
   'arithmetic:arithexpression'.
'arithmetic:arithexpression'**'arithmetic:arithexpression'->
   'arithmetic:arithexpression'.
'arithmetic:arithexpression'>>'arithmetic:arithexpression' ->
   'arithmetic:arithexpression'.
'arithmetic:arithexpression'<<'arithmetic:arithexpression'->
   'arithmetic:arithexpression'.
'arithmetic:arithexpression'/\'arithmetic:arithexpression'->
   'arithmetic:arithexpression'.
'arithmetic:arithexpression'\/'arithmetic:arithexpression'->
   'arithmetic:arithexpression'.
\'arithmetic:arithexpression'->'arithmetic:arithexpression'.
'arithmetic:arithexpression'#'arithmetic:arithexpression'->
   'arithmetic:arithexpression'.
exp('arithmetic:arithexpression')->'arithmetic:arithexpression'.
log('arithmetic:arithexpression')->'arithmetic:arithexpression'.
sqrt('arithmetic:arithexpression')->'arithmetic:arithexpression'.
sin('arithmetic:arithexpression')->'arithmetic:arithexpression'.
cos('arithmetic:arithexpression')->'arithmetic:arithexpression'.
atan('arithmetic:arithexpression')->'arithmetic:arithexpression'.
```

```
gcd('arithmetic:arithexpression','arithmetic:arithexpression')->
   'arithmetic:arithexpression'.
['basic_props:num'|niltype]->'arithmetic:arithexpression'.
[]->niltype.
```

We then write a user type, defining lists of arithmetic expressions, in which we can refer to the system type.

```
[]->listnum.
['arithmetic:arithexpression'|listnum] ->listnum.
```

To illustrate the process we analyse the `quicksort` module given below.

```
:- module(qsort1,_).

qs(X,Y) :- qs(X,Y,[]).

qs([],Y,Y).
qs([X|Xs],Ys,Zs) :-
        partition(X,Xs,Xs1,Xs2),
        qs(Xs1,Ys,[X|U]),
        qs(Xs2,U,Zs).

partition(X,[],[],[]) :- true.
partition(X,[Y|Ys],Ys1,[Y|Ys2]) :-
        X =< Y,
        partition(X,Ys,Ys1,Ys2).
partition(X,[Y|Ys],[Y|Ys1],Ys2) :-
        X > Y,
        partition(X,Ys,Ys1,Ys2).
```

The meta program expressing the dependencies is shown below. The predicates `$dqs2`, `$dqs3` and `$dpartition4` are the predicates capturing the dependencies. In each of these predicates, the first argument is a call (to the respective program predicate) and the second is a subequent call to a builtin predicate.

```
qs(A,B) :-
    qs(A,B,[]).
qs([],A,A) :-
    true.
qs([A|B],C,D) :-
    partition(A,B,E,F),
    qs(E,C,[A|G]),
    qs(F,G,D).
partition(A,[],[],[]) :-
    true.
partition(A,[B|C],D,[B|E]) :-
    'arithmetic:=<'(A,B),
    partition(A,C,D,E).
partition(A,[B|C],[B|D],E) :-
    'arithmetic:>'(A,B),
    partition(A,C,D,E).
'$dqs2'(atom(qs(A,B)),Q) :-
    '$dqs3'(atom(qs(A,B,[])),Q).
'$dqs3'(atom(qs([A|B],C,D)),Q) :-
    partition(A,B,E,F),qs(E,C,[A|G]),
    '$dqs3'(atom(qs(F,G,D)),Q).
'$dqs3'(atom(qs([A|B],C,D)),Q) :-
    partition(A,B,E,F),
    '$dqs3'(atom(qs(E,C,[A|G])),Q).
'$dqs3'(atom(qs([A|B],C,D)),Q) :-
    '$dpartition4'(atom(partition(A,B,E,F)),Q).
'$dpartition4'(atom(partition(A,[B|C],D,[B|E])),Q) :-
    'arithmetic:=<'(A,B),
    '$dpartition4'(atom(partition(A,C,D,E)),Q).
'$dpartition4'(atom(partition(A,[B|C],D,[B|E])),
                    atom('arithmetic:=<'(A,B))) :-
    true.
'$dpartition4'(atom(partition(A,[B|C],[B|D],E)),Q) :-
    'arithmetic:>'(A,B),
    '$dpartition4'(atom(partition(A,C,D,E)),Q).
'$dpartition4'(atom(partition(A,[B|C],[B|D],E)),
                    atom('arithmetic:>'(A,B))) :-
    true.
```

The analysis results, analysing the above program over the types given above, supplemented with the type `dynamic` to ensure completeness, yields the following analysis for the meta-predicate expressing the dependence of builtins on the entry predicate `qs/2`. In the notation below the value `q4` in the determinized domain is the element denoting arithmetic expressions, `q1` denotes lists of arithmetic expressions, and `q3` denotes other terms.

```
[$dqs2(A,B) :
     [$dqs2(qs(q1,A),arithmetic:>(q4,q4)),
     $dqs2(qs(q1,A),arithmetic:=<(q4,q4)),
     $dqs2(qs(q3,A),arithmetic:>(B,C)),
     $dqs2(qs(q3,A),arithmetic:=<(B,C))].
```

Inspecting these results, it can be seen that unsafe calls to arithmetic predicates can occur only when the first argument of `qs` is `q3`. When the first argument is a list of arithmetic expressions, the calls to the arithmetic predicates contain arithmetic expressions in both arguments.

17 Discussion

The results above show the flexibility of our approach w.r.t. other results reports in the literature. King and Lu [42] discuss similar experiments but are limited to simple Boolean domains like Pos. In our approach, as shown, we can use arbitrary regular types, mixing system types with user definitions. For arithmetic predicates, groundness alone of the first argument is insufficient to guarantee absence pf runtime errors, since calling say, `X is a`, results in a runtime error although the second argument is ground. We are able to analyse with respect to the more precise type of arithmetic expressions.

A second point is that, in order to obtain useful results for the `qsort` program, we needed to introduce a user type, namely lists of arithmetic expressions. This raises a general issue: what types are needed in order to propagate the conditions on builtins back to the entry goals accurately enough? This is a topic for future research, but we believe that automatic type inference using approaches such as regular approximations [39] and [37] can suggest appropriate types.

Part IV

Towards Provably Correct Code Generation via Horn Logical Continuation Semantics

18 Introduction

Provably correct compilation is an important aspect in development of high assurance software systems. In this work we explore approaches to provably correct code generation based on programming language semantics, particularly *Horn logical semantics*, and partial evaluation. We show that the *definite clause grammar (DCG)* notation can be used for specifying both the syntax and semantics of imperative languages. We next show that continuation semantics can also be expressed in the Horn logical framework.

Ensuring the correctness of the compilation process is an important consideration in construction of reliable software. If the compiler generates code that is not faithful to the original program code of a system, then all our efforts spent in proving the correctness of the system could be futile. Proving that target code is correct w.r.t. the program source is especially important for high assurance systems, as unfaithful target code can lead to loss of life and/or property. Considerable research has been done in this area, starting from the work of McCarthy [16]. Most efforts directed at proving compiler correctness fall into three categories:

- Those that treat the compiler as just another program and use standard verification techniques to manually or semi-automatically establish its correctness (e.g., [3]). However, even with semi-automation this is a very labour intensive and expensive undertaking, which has to be repeated for every new language, or if the compiler is changed.

- Those that *generate* the compiler automatically from the mathematical semantics of the language. Typically the semantics used is denotational (see for example Chapter 10 of [20]). The automatically generated compilers, however, have not been used in practice due to their slowness and/or inefficiency/poor quality of the code generated.

- Those that use program transformation systems to transform source code into target code [15, 18]. The disadvantage in this approach is that specifying the compiler operationally can be quite a lengthy process. Also, the compilation time can be quite large.

In [5] we developed an approach for generating code for imperative languages in a provably correct manner based on partial evaluation and a type of semantics called *Horn logical semantics*. This approach is similar in spirit to semantics-based approaches, however, its basis is Horn-logical semantics [5] which possesses both an operational as well as a denotational (declarative) flavor. In the Horn logical semantics approach, both the syntax and semantics of a language is specified using Horn logic statements (or pure Prolog).

Taking an operational view, one immediately obtains an interpreter of the language $\mathcal{L}$ from the Horn-logical semantic description of the language $\mathcal{L}$. The semantics can be viewed dually as operational or denotational. Given a program $\mathcal{P}$ written in language $\mathcal{L}$, the interpreter obtained for $\mathcal{L}$ can be used to execute the program. Moreover, given a partial evaluator for pure Prolog, the interpreter can be *partially evaluated* w.r.t. the program $\mathcal{P}$ to obtain compiled code for $\mathcal{P}$. Since the compiled code is obtained automatically via partial evaluation of the interpreter, it is faithful to the source of $\mathcal{P}$, provided the partial evaluator is correct. The correctness of the partial evaluator, however, has to be proven only once. The correctness of the code generation process for *any* language can be certified, provided the compiled code is obtained via partial evaluation. Given that efficient execution engines have been developed for Horn Logic (pure Prolog), partial evaluation is relatively fast. Also, the declarative nature of the Horn logical semantics allows for language semantics to be rapidly obtained.

In this contribution, we further develop our approach and show that in Horn logical semantics not only the syntax but also the semantics can be expressed using the definite clause grammar notation. The semantics expressed in the DCG notation allows for the store argument to be naturally (syntactically) hidden. We show that continuation semantics can also be expressed in Horn logic. Continuation semantics model the semantics of imperative constructs such as *goto statements*, *exception handling mechanisms, abort*, and *catch/throw constructs* more naturally. We also show that continuation semantics expressed as DCGs can be partially evaluated w.r.t. a source program to obtain "good quality" target code.

In this work we use partial evaluation to generate target code. Partial evaluation is especially useful when applied to interpreters; in this setting the static input is typically the object program being interpreted, while the actual call to the object program is dynamic. Partial evaluation can then produce a more efficient, specialized version of the interpreter, which can be viewed as a compiled version of the object program [34].

In our work we have used the LOGEN system [13]. Much like MIXTUS, LOGEN can handle many non-declarative aspects of Prolog. LOGEN also supports *partially static* data by allowing the user to declare custom "binding types." More details on the LOGEN system can be found elsewhere [13]. Unlike MIXTUS, LOGEN is a so-called *offline* partial evaluator, i.e., specialization is divided into two phases: (i) A *binding-time analysis* (BTA for short) phase which, given a program and an approximation of the input available for specialization, approximates all values within the program and generates annotations that steer (or control) the specialization process. (ii) A (simplified) *specialization phase*, which is guided by the result of the BTA.

Because of the preliminary BTA, the specialization process itself can be performed very efficiently, with predictable results (which is important for our application). Moreover, due to its simplicity it is much easier to establish correctness of the specialization process.

Finally, while our work is motivated by provably correct code generation, we believe our approach to be useful to develop "ordinary" compilers for domain specific languages in general [7].

19 Horn Logical Semantics

The denotational semantics of a language $\mathcal{L}$ has three components: (i) *syntax specification:* maps sentences of $\mathcal{L}$ to parse trees; it is commonly specified as a grammar in the BNF format; (ii) *semantic algebra:* represents the mathematical objects whose elements are used for expressing the meaning of a program written in the language $\mathcal{L}$; these mathematical objects typically are sets or domains (partially ordered sets, lattices, etc.) along with associated operations to manipulate the elements of the sets; (iii) *valuation functions:* these are functions mapping parse trees to elements of the semantic algebras.

Traditional denotational definitions express syntax as BNF grammars, and the semantic algebras and valuation functions using λ-calculus. In Horn Logical semantics, Horn-clauses (or pure Prolog) and constraints[4] are used instead to specify all the components of the denotational semantics of programming languages [5]. There are three major advantages of using Horn clauses and constraints for coding denotational semantics.

First, the syntax specification *trivially and naturally* yields an executable parser. The BNF specification of a language $\mathcal{L}$ can be quite easily transformed to a *Definite Clause Grammar* (DCG) [21]. The syntax specification[5] written in the DCG notation serves as a parser for $\mathcal{L}$. This parser can be used to parse programs written in $\mathcal{L}$ and obtain their parse trees (or syntax trees). Thus, the syntactic BNF specification of a language is easily turned into *executable syntax* (i.e., a parser). Note that the syntax of even context sensitive languages can be specified using DCGs [5].

[4]Constraints may be used, for example, to specify semantics of languages for real-time systems [6].

[5]A grammar coded as a DCG is syntax specifi cation in the sense that various operational semantics of logic programming (standard Prolog order, tabled execution, etc.) can be used for execution during actual parsing. Different operational semantics will result in different parsing algorithms (e.g., Prolog in recursive descent parsing with backtracking, tabled execution in chart parsing, etc.).

Second, the semantic algebra and valuation functions of $\mathcal{L}$ can also be coded in Horn-clause Logic. Since Horn-clause Logic or pure Prolog is a declarative programming notation, just like the λ-calculus, the mathematical properties of denotational semantics are preserved. Since both the syntax and semantic part of the denotational specification are expressed as logic programs, they are both executable. These syntax and semantic specifications can be loaded in a logic programming system and executed, given a program written in $\mathcal{L}$. This provides us with an interpreter for the language $\mathcal{L}$. In other words, the *denotation*[6] of a program written in $\mathcal{L}$ is executable. This executable denotation can also be used for many applications, including automated generation of compiled code.

Third, non-deterministic[7] semantics can be given to a language w.r.t. resources (e.g., time, space, battery power) consumed during execution. For example, some operations in the semantic algebra may be specified in multiple ways (say in software or in hardware) with each type of specification resulting in different resource consumption. Given a program and bounds on the resources that can be consumed, only some of the many possible semantics may be viable for that program. Resource bounded partial evaluation [2] can be used to formalize resource conscious compilation (e.g., energy aware compilation) [23] via Horn Logical semantics.

Horn-logical semantics can also be used for automatic verification and consistency checking [5, 6]. We do not elaborate any further since we are not concerned with verification in this work.

The disadvantage of Horn logical semantics is that it is not denotational in the strict sense of the word because the semantics given for looping constructs is not compositional. The fix operator used to give compositional semantics of looping constructs in λ-calculus cannot be naturally coded in Horn logic due to lack of higher order functions. This, for example, precludes the use of structural induction to prove properties of programs. However, note that even though the semantics is not truly compositional, it is declarative, and thus the fix-point of the logic program representing the semantics can be computed via the standard T_P operator [43]. Structural/fix-point induction can then be performed over this T_P operator to prove properties of programs. Note that even in the traditional λ-calculus approach, the declarative meaning of the fix operator (defined as computing the limit of a series of functions) is given outside the operational framework of the λ-calculus, just as the computation of the fix(T_P) in logic programming is outside the operational framework of Horn Clause logic. For partial evaluation, the operational definition of fix, i.e., fix(F) = F(fix F), is used.

In [5] we show how both the syntax and semantics of a simple imperative language (a simple subset of Pascal whose grammar is shown in Figure 1) can be given in Horn Logic. The Horn logical semantics, viewed operationally, automatically yields an interpreter. Given a program P,

[6]We refer to the denotation of a program under the Horn-logical semantics as its *Horn logical denotation.*

[7]Non-deterministic in the logic programming sense.

```
Program ::= C.
C ::= C1;C2 |
     loop while B C end while |
     if B then C1 else C2 endif |
     I := E
E ::= N | Identifier | E1 + E2 |
     E1 - E2 | E1 * E2 | (E)
B ::= E1 = E2 | E1 > E2 | E1 < E2
N ::= 0 | 1 | 2 | ...  | 9
Identifier ::= w | x | y | z
```

Figure 1: BNF grammar

the interpreter can be partially evaluated w.r.t. P to obtain P's compiled code.
A program and its corresponding code generated via partial evaluation using the LOGEN system [13] is shown below. The specialization time is insignificant (i.e., less than 10 ms). Note that the semantics is written under the assumption that the program takes exactly two inputs (found in variables `x` and `y`) and produces exactly one output (placed in variable `z`). *The definitions of the semantic algebra operations are removed, so that unfolding during partial evaluation will stop when a semantic algebra operation is encountered.* The semantic algebra operations are also shown below.

```
 z = 1;             main(A, B, C) :-        while_eval__1(A, B) :-
 w = x;             initialize_store(D),      access(w, A, C),
 loop while w > 0   update(x, A, D, E),       ( C>0 ->
   z = z * y ;      update(y, B, E, F),          access(z, A, D),
   w = w - 1        update(z, 1, F, G),          access(y, A, E),
 end while.         access(x, G, H),             F is D*E,
                     update(w, H, G, I),          update(z, F, A, G),
                     while_eval__1(I, J),         access(w, G, H),
                     K=J,                         I is H-1,
                     access(z, K, C).             update(w, I, G, J),
                                                  while_eval__1(J, B),
                                                 ;   B=A ).
SEMANTIC ALGEBRA:
 initialize_store([(x,0),(y,0),(z,0),(w,0)]).
 access(Id,[(Id,Val)|_ ],Val).    update(Id,NV,[(Id,_)|R],[(Id,NV)|R]).
 access(Id,[_|R],Val) :-          update(Id,NewV,[P|R],[P|R1]) :-
               access(Id,R,Val).                update(Id,NewV,R,R1).
```

Notice that in the program that results from partial evaluation, only a series of memory `access`, memory `update`, arithmetic and comparison operations are left, that correspond to `load`, `store`, arithmetic, and comparison operations of a machine language. The while-loop, whose meaning was expressed using recursion, will partially evaluate to a *tail-recursive* program. These tail-recursive calls are easily converted to iterative structures using jumps in the target code.

Though the compiled code generated is in Prolog syntax, it looks a lot like machine code. A few simple transformation steps will produce actual machine code. These transformations include replacing variable names by register/memory locations, replacing a Prolog function call by a jump (using a goto) to the code for that function, etc. The code generation process is provably correct, since target code is obtained automatically via partial evaluation. Of course, we need to ensure that the partial evaluator works correctly. However, this needs to be done only once. Note that once we prove the correctness of the partial evaluator, compiled code for programs written in any language can be generated as long as the Horn-logical semantics of the language is given.

It is easy to see that valuation predicate for an iterative structure will always be tail-recursive. This is because the operational meaning of a looping construct can be given by first iterating through the body of the loop once, and then recursively re-processing the loop after the state has been appropriately changed to reflect the new values of the loop control parameters. The valuation predicate for expressing this operational meaning will be inherently tail recursive.

Note also that if a predicate definition is tail recursive, a folding/unfolding based partial evaluation of the predicate will preserve its tail-recursiveness. This allows us to replace a tail recursive call with a simple jump while producing the final assembly code. The fact that tail-recursiveness is preserved follows from the fact that folding/unfolding based partial evaluation can be viewed as algebraic simplification, given the definitions of various predicates. Thus, given a tail recursive definition, the calls in its body will be expanded in-place during partial evaluation. Expanding a tail-recursive call will result in either the tail-recursion being eliminated or being replaced again by its definition. Since the original definition is tail-recursive, the unfolded definition will stay tail recursive. (A formal proof via structural induction can be given [22] but is omitted due to lack of space.)

20 Definite Clause Semantics

Note that in the code generated, the `update` and `access` operations are parameterized on the memory store (i.e., they take an input store and produce an output store). Of course, real machine instructions are not parameterized on store. This store parameter can be (syntactically) eliminated by using the DCG notation for expressing the valuation predicates as well.

All valuation predicates take a store argument as input, modify it per the semantics of the command under consideration and produce the modified store as output [5]. Because the semantic rules are stated declaratively, the store argument “weaves” through the semantic sub-predicates called in the rule. This suggests that we can express the semantic rules in the DCG notation. Thus, we can view the semantic rules as computing the *difference* between the output and the input stores. This difference reflects the effect of the command whose semantics is being given. Expressed in the DCG notation, the store argument is (syntactically) hidden away. For example, in the DCG notation the valuation predicate

```
command(comb(C1, C2), Store, Outstore) :-
          command(C1, Store, Nstore),
          command(C2, Nstore, Outstore).
```

is written as:

```
command(comb(C1, C2)) --> command(C1), command(C2).
```

In terms of difference structures, this rules states that the difference of stores produced by `C1; C2` is the “sum” of differences of stores produced by the command `C1` and `C2`. The rest of the semantic predicates can be rewritten in this DCG notation in a similar way.

```
main(U,V,A) -->
      update(x,U),
      update(y,V),
      update(z,1),
      access(x,F),
      update(w,F),
      while_eval__1,
      access(z,A).
while_eval__1 -->
     (access(w,C),
      {0<C} ->
         access(z,D),
         access(y,E),
         {F is D*E},
         update(z,F),
         access(w,H),
         {I is H-1},
         update(w,I),
         while_eval__1
       ; []).
```

```
main:            while:
 store x U        load w C
 store y V        skipgtz C
 store z 1        jump else
 load x F         load z D
 store w F        load y E
 jump while       mul D E F
end:              store z F
 load z W         load w H
                  sub1 H I
                  store w I
                  jump while
                 else:
                  noop
                  jump end
```

Figure 2: Partially evaluated semantics and its assembly code

Expressed in the DCG notation, the semantic rules become more intuitively obvious. In fact, these rules have more natural reading; they can be read as simple rewrite rules. Additionally, now we can partially evaluate this DCG w.r.t. an input program, and obtain compiled code that has the store argument syntactically hidden. The result of partially evaluating this DCG-formatted semantics is shown to the left in Figure 2. Notice that the store argument weaving through the generated code shown in the original partially evaluated code is hidden away. Notice also that the basic operations (such as comparisons, arithmetic, etc.) that appear in the target code are placed in braces in definite clause semantics, so that the two store arguments are not added during expansion to Prolog. The constructs appearing within braces can be regarded as the "terminal" symbols in this semantic evaluation, similar to terminal symbols appearing in square brackets in the syntax specification. In fact, the operations enclosed within braces are the primitive operations left in the residual target code after partial evaluation. Note, however, that these braces can be eliminated by putting wrappers around the primitive operations; these wrappers will have two redundant store arguments that are identical, per the requirements of the DCG notation. Note also that since the LOGEN partial evaluator is oblivious of the DCG notation, the final generated code was cast into the DCG notation manually.

Now that the store argument that was threading through the code has been eliminated, the access/update instructions can be replaced by load/store instructions, tail recursive call can be replaced by a jump, etc., to yield proper assembly code. The assembly code that results in shown to the right in figure 2. We assume that inputs will be found in registers `U` and `V`, and the output will be placed in register `W`. Note that `x, y, z, w` refer to the memory locations allocated for the respective variables. Uppercase letters denote registers. The instruction `load x Y` moves the value of memory location `x` into register `Y`, likewise `store x Y` moves the value of register Y in memory location `x` (on a modern microprocessor, both `load` and `store` will be replaced by the `mov` instruction); the instruction `jump label` performs an unconditional jump, `mul D E F` multiplies the operands `D` and `E` and puts the result in register `F`, `sub1 A B` subtracts 1 from register A and puts the result in register B, while `skipgtz C` instruction realizes a conditional expression (it checks if register `C` is greater than zero, and if so, skips the immediately following instruction).

Note that we have claimed the semantics (e.g., the one given in section 20) to be denotational. However, there are two problems: (i) First, we use the `(p->q;r)` construct of logic programming which has a hidden cut, which means that the semantics predicates are not even declarative. (ii) second, the semantics is not truly compositional, because the semantics of the while command is given in terms of the while command itself. This non-compositionality means that structural induction cannot be applied.

W.r.t. (i) note that the condition in the `->` always involves a relational operator with ground arguments (e.g., `Bval = true`). The negation of such relational expressions can always be computed and the clause expanded to eliminate the cut. Thus, a clause of the form

```
        p(..)  :- (Bval = true -> q(...); r(...))
```

can be re-written as

```
        p(..)  :- Bval = true, q(...).
        p(..)  :- Bval = false, r(...).
```

Note that this does not adversely affect the quality of code produced via partial evaluation.

W.r.t. (ii), as noted earlier, program properties can still be proved via structural induction on the T_P operator, where P represents the Horn logical semantic definition.

Another issue that needs to be addressed is the ease of proving a partial evaluator correct given that a partial evaluator such as LOGEN [13] or Mixtus [51] are complex pieces of software. However, as already mentioned, because of the offline approach the actual specialization phase of LOGEN is quite straightforward and should be much easier to prove correct. Also, because of the predictability of the offline approach, it should also be possible to formally establish that the output of LOGEN corresponds to proper target code.[8]

Note that because partial evaluation is done until only the calls to the semantic algebra operation remain, the person defining the semantics can control the type of code generated by suitably defining the semantic algebra. Thus, for example, one can first define the semantics of a language in terms of semantic algebra operations that correspond to operations in an abstract machine. Abstract machine code for a program can be generated by partial evaluation w.r.t. this semantics. This code can be further refined by giving a lower level semantics for abstract machine code programs. Partial evaluation w.r.t. this lower level semantics will yield the lower level (native) code.

21 Continuation Semantics

So far we have modeled only direct semantics [20] using Horn logic. It is well known that direct semantics cannot naturally model exception mechanisms and `goto` statements of imperative programming languages. To express such constructs naturally, one has to resort to continuation semantics. We next show how continuation semantics can be naturally expressed in Horn Clause logics using the DCG notation. In the definite clause continuation semantics, semantics of constructs is given in terms of the *differences of parse trees* (i.e., difference of the input parse tree and the continuation's parse tree) [22]. Each semantic predicate thus relates an individual con-

[8]E.g., for looping constructs, the unfolding of the (tail) recursive call has to be done only once through the recursive call to obtain proper target code.

struct (difference of two parse trees) to a fragment of the store (difference of two stores). Thus, semantic rules are of the form:

```
    command(C1, C2, Program, S1, S2) :- ...
```

where the difference of C1 and C2 (say ΔC) represents the command whose semantics is being given, and the difference of S1 and S2 represents the store which reflects the incremental change (ΔS) brought about to the store by the command ΔC. Note that the `Program` parameter is needed to carry the mapping between labels and the corresponding command. Each semantic rule thus is a stand alone rule relating the difference of command lists, ΔC, to difference of stores, ΔS. *If we view a program as a sequence of difference of command lists then its semantics can simply be obtained by "summing" the difference of stores for each command.* That is, if we view a program P as consisting of sequence of commands:

$P = \Delta C_1 + \Delta C_2 + \ldots + \Delta C_n$

then its semantics S is viewed as a "sum" of the corresponding differences of stores:

$S = \Delta S_1 \oplus \Delta S_2 \oplus \ldots \oplus \Delta S_n$

and the continuation semantics simply maps each ΔC_i to the corresponding ΔS_i. Note that $\oplus$ is a non-commutative operator, and its exact definition depends on how the store is modeled. Additionally, continuation semantics allow for cleaner, more intuitive declarative semantics for imperative constructs such as exceptions, catch/throw, goto, etc. [20].

Finally, note that the above continuation semantics rules can also be written in the DCG notation causing the arguments `S1` and `S2` to become syntactically hidden:

```
    command(C1, C2, Program) --> ...
```

Below, we give the continuation semantics of the subset of Pascal considered earlier after extending it with statement labels and a `goto` statement. Note that the syntax trees are now represented as a list of commands. Each command is represented in the syntax tree as a pair, whose first element is a label (possibly null) and the second element is the command itself. Only the valuation functions for commands are shown (those for expressions, etc., are similar to the one shown earlier).

```
prog_eval([], _, _, 0) --> []
prog_eval(CommList, Val_x, Val_y, Output) -->
   update(x, Val_x), update(y, Val_y),
   command_eval(CommList,cont([],[]), CommList), access(z, Output).

command_eval([],[],_Program) --> [].
command_eval([],cont(CommList,Cont),Program)-->
   command_eval(CommList,Cont,Program).
command_eval([Comm|CommList],Cont,Program)-->
```

```
   comm_eval(Comm,CommList,Cont,NCommList,NCont,Program),
   command_eval(NCommList,NCont,Program).

comm_eval([(_,abort)|_],_Comm,_Cont,[],[],_Program) --> [].
comm_eval((Label,while(B,LoopBody)),OldRest,OldCont,[],[],Program)
     --> bool_while_eval(B,LoopBody,
             cont([(Label,while(B,LoopBody))|OldRest],
                    OldCont), OldRest,OldCont,Program).
comm_eval((_,ce(B,C1,C2)),OldRest,OldCont,[],[],Program) -->
   bool_eval(B,C1,cont(OldRest,OldCont),C2,cont(OldRest,OldCont),Program).
comm_eval((_,ce(B,C1)),OldRest,OldCont,[],[],Program) -->
   bool_eval(B,C1,cont(OldRest,OldCont),OldRest,OldCont,Program).
comm_eval((_,jmp(ID)),_OldRest,_OldCont,JumpList,cont([],[]),Program)-->
   {find_label(ID,Program,JumpList)}.
comm_eval((_,assign(id(I), E)),OldRest,OldCont,OldRest,OldCont,_Program)
    --> expr(E, Val), update(I, Val).

bool_while_eval(Cond,C1,C1Cont,C2,C2Cont,Program) -->
   bool_eval(Cond,C1,C1Cont,C2,C2Cont,Program).
bool_eval(greater(E1, E2),C1,C1Cont,C2,C2Cont,Program)
    --> expr(E1, Eval1), expr(E2, Eval2),
          ({Eval1 > Eval2} -> command_eval(C1,C1Cont,Program) ;
                              command_eval(C2,C2Cont,Program)).
  /*the code for lesser(E1,E2) and equal(E1,E2) is very similar*/
```

The code above is self-explanatory. Semantic predicates pass command continuations as arguments. The code for `find_label/3` predicate is not shown. It looks for the program segment that is a target of a `goto` and changes the current continuation to that part of the code.

Consider the program shown below to the left in Figure 3. In this program segment, control jumps from outside the loop to inside via the goto statement. The result of partially evaluating the interpreter (after removing the definitions of semantic algebra operations) obtained from the semantics w.r.t. this program (containing a `goto`) is shown in the figure 3 to the right. Figures 4 shows another instance of a program involving goto's and the code generated by the LOGEN partial evaluator by specialization of the definite clause continuation semantics shown above.

Note that a Horn logical continuation semantics can be given for any imperative language in such a way that its partial evaluation w.r.t. a program will yield target code in terms of access/update operation. This follows from the fact that programs written in imperative languages

```
//source code
z = 1;
w = x;
goto label;
loop while w > 0
   z = z * y ;
   label: w = w - 1
endloop while;
z = 8;
z = 7.
```

```
//generated code
interpreter(A, B, C) -->
   update(x, A),
   update(y, B),
   update(z, 1),
   access(x, D),
   update(w, D),
   access(w, E),
   {F is E-1},
   update(w, F),
   fix1,
   access(z, C).
```

```
fix1 -->
   ( access(w, A),
      {0<A} ->
      access(z, B),
      access(y, C),
      {D is B*C},
      update(z, D),
      access(w, E),
      {F is E-1},
      update(w, F),
      fix1
   ;  update(z, 8),
      update(z, 7)
   ).
```

Figure 3: Example with a jump from outside to inside a while loop

consist of a series of commands executed under a control that is explicitly supplied by the programmer. Control is required to be specified to a degree that the continuation of each command can be uniquely determined. Each command (possibly) modifies the store. Continuation semantics of a command is based on modeling the change brought about to the store by the continuation of this command. Looking at the structure of the continuation semantics shown above, one notes that programs are represented as lists of commands. The continuation of each command may be the (syntactically) next command or it might be some other command explicitly specified by a control construct (such as a goto or a loop). The continuation is modeled in the semantics explicitly, and can be explicitly set depending on the control construct. The semantics rule for each individual command computes the changes made to the store as well as the new continuation. Thus, as long as the control of an imperative language is such that *the continuation of each command can be explicitly determined*, its Horn logical continuation semantics can be written in the DCG syntax. Further, since the semantics is executable, given a program written in the imperative language, it can be executed under this semantics. The execution can be viewed as unfolding the top-level call, until all goals are solved. If the definitions of the semantic algebra operations are removed, then the top-level call can be simplified via unfolding (partial evaluation) to a resolvent which only contains calls to the semantic algebra operations; this resolvent will correspond to the target code of the program.

It should also be noted that the LOGEN system allows users to control the partial evaluation process via annotations. Annotations are generated by the BTA and then can be modified manually. This feature of the LOGEN system gives considerable control of the partial evalua-

```
//source code
z = 1;
w = x;
loop while w > 0
   z = z * y ;
   w = w - 1;
   goto label
endloop while;
label: z = 8
z = 7.
```

```
//generated code
interpreter(A, B, C) -->
   update(x, A), update(y, B),
   update(z, 1),
   access(x, D), update(w, D),
   ( access(w, E),
      {0<E} ->
      access(z, F), access(y, G),
      {H is F*G},
      update(z, H),
      access(w, I),
      {J is I-1},
      update(w, J),
      update(z, 8), update(z, 7)
   ;  update(z, 8), update(z, 7)
   ),
   access(z, C).
```

Figure 4: Example with a jump from inside to outside a while loop

tion process—and hence of the code generation process—to the user. The interpreter has to be annotated only once by the user, to ensure that good quality code will be generated.

22 A Case Study in SCR

We have applied our approach to a number of practical applications. These include generating code for parallelizing compilers in a provably correct manner [5], generating code for controllers specified in Ada [12] and for domain specific languages [7] in a provably correct manner, and most recently generating code in a provably correct manner for the Software Cost Reduction (SCR) framework.

The SCR (Software Cost Reduction) requirements method is a software development methodology introduced in the 80s [8] for engineering reliable software systems. The target domain for SCR is real-time embedded systems. SCR has been applied to a number of practical systems, including avionics system (the A-7 Operational flight Program), a submarine communication system, and safety-critical components of a nuclear power plant [9].

We have developed the Horn logical continuation semantics for the complete SCR language. This Horn logical semantics immediately provides us with an interpreter on which the program above can be executed. Further, the interpreter was partially evaluated and compiled code was obtained. The time taken to obtain compile code using definite clause continuation semantics of SCR was an order of magnitude faster than a program transformation based strategy described in [15] that uses the APTS system [18], and more than 40 times faster than a strategy that associates C code as attributes of parse tree nodes and synthesizes the overall code from it [15].

23 Related Work

Considerable work has been done on manually or semi-mechanically proving compilers correct. Most of these efforts are based on taking a specific compiler and showing its implementation to be correct. A number of tools (e.g., a theorem prover) may be used to semi-mechanize the proof. Example of such efforts range from McCarthy's work in 1967 [16] to more recent ones [3]. As mentioned earlier, these approaches are either manual or semi-mechanical, requiring human intervention, and therefore not completely reliable enough for engineering high-assurance systems. "Verifying Compilers" have also been considered as one of the grand challenge for computing research [10], although the emphasis in [10] is more on developing a compiler that can verify the assertions inserted in programs (of course, such a compiler has to be proven correct first).

Considerable work has also been done on generating compilers automatically from language semantics [20]. However, because the syntax is specified as a (non-executable) BNF and semantics is specified using λ-calculus, the automatic generation process is very cumbersome. The approach outlined in this work falls in this class, except that it uses Horn logical semantics which, we believe and experience suggests, can be manipulated more efficiently. Also, because Horn logical semantics has more of an operational flavor, code generation via partial evaluation can be done quite efficiently.

Considerable work has also been done in using term rewriting systems for transforming source code to target code. In fact, this approach has been applied by researchers at NRL to automatically generate C code from SCR specification using the APTS [18] program transformation system. As noted earlier, the time taken is considerably more than in our approach. Other approaches that fall in this category include the HATS system [24] that use tree rewriting to accomplish transformations. Other transformation based approaches are mentioned in [15].

Recently, Pnueli et al have taken the approach of verifying a given run of the compiler rather than a compiler itself [19]. This removes the burden of maintaining the compiler's correctness proof; instead each run is proved correct by establishing a refinement relationship. However, this approach is limited to very simple languages. As the authors themselves mention, their approach “seems to work in all cases that the source and target programs each consist of a repeated execution of a single loop body ..,” and as such is limited. For such simple languages, we believe that a Horn logical semantics based solution will perform much better and will be far easier to develop. Development of the refinement relation is also not a trivial task. For general programs and general languages, it is unlikely that the approach will work.

Note that considerable work has been done in partially evaluating meta-interpreters for declarative languages, in order to eliminate the interpretation overhead (see, for example, [17, 1]). However, in this work our goal is to generate assembly-like target code for imperative languages.

24 Conclusions

In this work we presented an approach based on formal semantics, Horn logic, and partial evaluation for obtaining provably correct compiled code. We showed that not only the syntax specification, but also the semantic specification can be coded in the DCG notation. We also showed that continuation semantics of an imperative language can also be coded in Horn clause logic. We applied our approach to a real world language—the SCR language for specifying real-time embedded system. The complete syntax and semantic specification for SCR was developed and used for automatically generating code for SCR specifications. Our method produces executable code considerably faster than other transformation based methods for automatically generating code for SCR specifications.

Acknowledgments

We are grateful to Constance Heitmeyer and Elizabeth Leonard of the Naval Research Labs for providing us with the BNF grammar of SCR and the safety injection program as well as for discussions, and to the anonymous referees. Work in Part IV was done jointly with Qian Wang and Gopal Gupta of the University of Texas at Dallas, who were partially supported by NSF grants CCR 9900320, INT 9904063, and EIA 0130847, by the Department of Education and the Environmental Protection Agency.

Part V

References

References

[1] A. Brogi and S. Contiero. Specializing Meta-Level Compositions of Logic Programs. Proceedings LOPSTR'96, J. Gallagher. Springer-Verlag, LNCS 1207.

[2] S. Debray. Resource bounded partial evaluation. PEPM 1997. pp. 179-192.

[3] A. Dold, T. Gaul, W. Zimmermann Mechanized Verification of Compiler Backends Proc. Software Tools for Technology Transfer, Denmark, 1998.

[4] S. R. Faulk. State Determination in Hard-Embedded Systems. Ph.D. Thesis, Univ. of NC, Chapel Hill, NC, 1989.

[5] G. Gupta "Horn Logic Denotations and Their Applications," *The Logic Programming Paradigm: A 25 year perspective*. Springer Verlag. 1999:127-160.

[6] G. Gupta, E. Pontelli. A Constraint-based Denotational Approach to Specification and Verification of Real-time Systems. In *Proc. IEEE Real-time Systems Symposium,* pp. 230-239. Dec. 1997.

[7] G. Gupta, E. Pontelli. A Logic Programming Framework for Specification and Implementation of Domain Specific Languages. In *Essays in Honor of Robert Kowalski*, 2003, Springer Verlag LNAI,

[8] K. L. Henninger. Specifying software requirements for complex systems: New techniques and their application. IEEE Trans. on Software Engg. 5(1):2-13.

[9] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated Consistency Checking of Requirements Specifications. ACM TOSEM 5(3). 1996.

[10] C. A. R. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. J.ACM, 50(1):63-69. Jan 2003.

[11] N. Jones. Introduction to Partial Evaluation. In *ACM Computing Surveys*. 28(3):480-503.

[12] L. King, G. Gupta, E. Pontelli. Verification of BART Controller. In *High Integrity Software*, Kluwer Academic, 2001.

[13] M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline specialisation in Prolog using a hand-written compiler generator. *Theory and Practice of Logic Programming*, 4(1):139–191, 2004.

[14] M. Leuschel, B. Martens, and D. De Schreye. Controlling Generalization and Polyvariance in Partial Deduction of Normal Logic Programs. ACM Transactions on Programming Languages and Systems (TOPLAS), 20(1):208-258.

[15] E. I. Leonard and C. L. Heitmeyer. Program Synthesis from Requirements Specifications Using APTS. Kluwer Academic Publishers, 2002.

[16] J. McCarthy and J. Painter. Correctness of a Compiler for Arithmetic Expressions. MIT AI Lab Memo, 1967.

[17] S. Owen. Issues in the Partial Evaluation of Meta-Interpreters. Proceedings Meta'88. MIT Press. pp. 319–339. 1989.

[18] R. Paige. Viewing a Program Transformation System at Work. *Proc. Programming Language Implementation and Logic Programming*, Springer, LNCS 844. 1994.

[19] A. Pnueli, M. Siegel, E. Singerman. Translation Validation. *Proc TACAS'98*, Springer Verlag LNCS, 1998.

[20] D. Schmidt. *Denotational Semantics: a Methodology for Language Development*. W.C. Brown Publishers, 1986.

[21] L. Sterling & S. Shapiro. The Art of Prolog. MIT Press, '94.

[22] Q. Wang, G. Gupta, M. Leuschel. Horn Logical Continuation Semantics. UT Dallas Technical Report. 2004.

[23] Q. Wang, G. Gupta. Resource Bounded Compilation via Constrained Partial Evaluation. UTD Technical Report. Forthcoming.

[24] V. L. Winter. Program Transformation in HATS. *Software Transformation Systems Workshop*, '99.

[25] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[26] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M. Hermenegildo, J. Maluszynski, and G. Puebla. On the Role of Semantic Approximations in Validation and Diagnosis of Constraint Logic Programs. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 155–170, Linköping, Sweden, May 1997. U. of Linköping Press.

[27] F. Bueno, P. López-García, and M. Hermenegildo. Multivariant Non-Failure Analysis via Standard Abstract Interpretation. In *7th International Symposium on Functional and Logic Programming (FLOPS 2004)*, number 2998 in LNCS, pages 100–116, Heidelberg, Germany, April 2004. Springer-Verlag.

[28] Michael Codish and Cohavit Taboch. A semantic basic for the termination analysis of logic programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.

[29] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. *Tree Automata Techniques and Applications*. http://www.grappa.univ-lille3.fr/tata, 1999.

[30] S.J. Craig, John P. Gallagher, M. Leuschel, and Kim S. Henriksen. Fully automatic binding time analysis for Prolog. In Sandro Etalle, editor, *Pre-Proceedings, 14th International Workshop on Logic-Based Program Synthesis and Transformation, LOPSTR 2004, Verona, August 2004*, pages 61–70, 2004.

[31] S.K. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

[32] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[33] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[34] Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971.

[35] Maurizio Gabbrielli and Roberto Giacobazzi. Goal independency and call patterns in the analysis of logic programs. In *Proceedings of the 1994 ACM Symposium on Applied Computing, SAC 1994*, pages 394 – 399, 1994.

[36] Maurizio Gabbrielli, Giorgio Levi, and Maria Chiara Meo. Resultants semantics for Prolog. *Journal of Logic and Computation*, 6(4):491–521, 1996.

[37] J. Gallagher and G. Puebla. Abstract Interpretation over Non-Deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In Shriram Krishnamurthi and C. R. Ramakrishnan, editors, *Fourth International Symposium on Practical BAspects of Declarative Languages (PADL'02)*, LNCS, pages 243–261, January 2002.

[38] J. P. Gallagher. A Program Transformation for Backwards Analysis of Logic Programs. In M. Bruynooghe, editor, *Proceedings of the International Symposium on Logic Based Program Synthesis and Transformation (LOPSTR 2003)*, volume 3018 of *LNCS*, pages 92–105, 2003.

[39] J. P. Gallagher and D.A. de Waal. Fast and precise regular approximation of logic programs. In P. Van Hentenryck, editor, *Proceedings of the International Conference on Logic Programming (ICLP'94), Santa Margherita Ligure, Italy*. MIT Press, 1994.

[40] J. P. Gallagher and K. S. Henriksen. Abstract domains based on regular types. In V. Lifschitz and B. Demoen, editors, *Proceedings of the International Conference on Logic Programming (ICLP'2004)*, volume 3132 of *Springer-Verlag Lecture Notes in Computer Science*, pages 27–42, 2004.

[41] S. Genaim and M. Codish. Inferring termination conditions of logic programs by backwards analysis. In *International Conference on Logic for Programming, Artificial intelligence and reasoning*, volume 2250 of *Springer Lecture Notes in Artificial Intelligence*, pages 681–690, 2001.

[42] Andy King and Lunjin Lu. A backward analysis for constraint logic programs. *Theory and Practice of Logic Programming*, 2(4-5):514–547, 2002.

[43] J. W. Lloyd. *Logic Programming*. Springer-Verlag, 1987.

[44] P. López-García, F. Bueno, and M. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, LNCS. Springer-Verlag, August 2004.

[45] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[46] L. Naish. A three-valued declarative debugging scheme. In *8th Workshop on Logic Programming Environments*, July 1997. ICLP Post-Conference Workshop.

[47] G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Constraint Logic Programs. In P. Deransart, M. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming*, number 1870 in LNCS, pages 23–61. Springer-Verlag, September 2000.

[48] G. Puebla, F. Bueno, and M. Hermenegildo. Combined Static and Dynamic Assertion-Based Debugging of Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation (LOPSTR'99)*, number 1817 in LNCS, pages 273–292. Springer-Verlag, 2000.

[49] G. Puebla and M. Hermenegildo. Implementation of Multiple Specialization in Logic Programs. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 77–87. ACM Press, June 1995.

[50] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.

[51] D. Sahlin. The mixtus approach to the automatic evaluation of full prolog. In *Proceedings of the North American Conference on Logic Programming*, pages 377–398. MIT Press, October 1990.

[52] C. Vaucheret and F. Bueno. More precise yet efficient type inference for logic programs. In *International Static Analysis Symposium*, number 2477 in LNCS, pages 102–116. Springer-Verlag, September 2002.