2. An Assertion Language for Constraint Logic Programs

Germán Puebla, Francisco Bueno, and Manuel Hermenegildo

School of Computer Science
Technical University of Madrid, Spain
email: {german,bueno,herme}@fi.upm.es

In an advanced program development environment, such as that discussed in the introduction of this book, several tools may coexist which handle both the program and information on the program in different ways. Also, these tools may interact among themselves and with the user. Thus, the different tools and the user need some way to communicate. It is our design principle that such communication be performed in terms of *assertions*. Assertions are syntactic objects which allow expressing properties of programs. Several assertion languages have been used in the past in different contexts, mainly related to program debugging. In this chapter we propose a general language of assertions which is used in different tools for validation and debugging of constraint logic programs in the context of the DiSCiPl project. The assertion language proposed is parametric w.r.t. the particular constraint domain and properties of interest being used in each different tool. The language proposed is quite general in that it poses few restrictions on the kind of properties which may be expressed. We believe the assertion language we propose is of practical relevance and appropriate for the different uses required in the tools considered.

2.1 Introduction

Assertions are linguistic constructions which allow expressing properties of programs. Assertions have been used in the past in different contexts and for different purposes related to program development:

Run-time checking: This is one of the traditional uses of assertions and has been applied extensively in the context of imperative programming languages. The user adds assertions to a program which express conditions about the program which should hold at run-time. Otherwise the program is *incorrect*. A usual example is to check that the value of a variable remains within a given range at a given program point. If assertions are found not to hold an error message is flagged. Note that in this context, assertions express properties about the run-time behaviour of the program which *should hold* if the program is correct (see [2.30] for an application in Constraint Logic Programming (CLP) [2.19]).

Replacing the oracle: In declarative debugging [2.27], the existence of an *oracle* (normally the user) which is capable of answering questions about the intended behaviour of the program is assumed. In this context, the user may write assertions which express properties which should hold if the program were correct [2.11, 2.12, 2.2]. If it is possible to answer the questions posed by the declarative debugger just by using the information given as assertions, then there is no need to ask the oracle (the user). This makes systems more practical because the burden on the user is reduced. Note that here again, assertions are used to express properties which *should hold* for the program to be correct.

Compile-time checking: In this context, the user may write assertions which express properties about the program which are intended to be checked at compile-time. The result of such checking may indicate either that the assertions actually hold and the program is *validated* w.r.t. the assertions or that the assertions do not hold, and then the program is *incorrect* w.r.t. the assertions. Again, these are properties which *should hold*, i.e., otherwise a bug exists in the program. An example of this kind of assertions are type declarations (e.g., [2.18, 2.28], functional languages, etc.), which have been shown to be useful in debugging. Generally, and in order to be able to check these properties at compile-time, the expressible properties are restricted in such a way that compile-time checking can always determine whether the assertions hold or not.

Providing information to the optimizer: Assertions have also been proposed as a means of providing information to an optimizer in order to perform additional optimizations during code generation. Such assertions can be provided by the user (e.g., [2.28], which also implements checking) or automatically generated, generally by means of static program analysis. In this context, assertions do not express properties which should hold for the program, but rather properties which *do hold* for the program at hand. Note that if the program is not correct, the properties which hold may not coincide with the properties which should hold.

General communication with the compiler: In a setting where there is both a static inference system, such as an abstract interpreter [2.9, 2.14], and an optimizer, assertions have also been proposed as a means of allowing the user to provide additional information to the analyzer [2.4], which it can use both to increase the precision of the information it infers and/or to perform additional optimizations during code generation [2.31, 2.29, 2.22, 2.20]. Examples are assertions which state information on *entry points* to a program module, assertions which describe properties of built-ins [2.17], assertions which provide some (but not all [2.4]) type declarations in the context of a type *inferencing* system, etc. Also, assertions can be used to represent analysis output in a user-friendly way and to communicate different modules of the compiler which deal with analysis information [2.4]. In this context, assertions express both properties which should hold and properties which *do hold* for the program in hand.

Program documentation: Assertions have also been used to document programs and to automatically generate manuals (as inspired by the "literate programming" style [2.21, 2.7]). These assertions are usually written by the user but they can also be automatically generated. Some examples are Javadoc [2.13], in the context of imperative languages, and LPdoc in the context of CLP [2.15]. In this application, assertions may express both properties which *do hold* or which *should hold* for the program in hand.

In addition to the classification given above, made according to the context in which assertions are used, assertions can be classified according to many other criteria. For example, as mentioned above, in some cases the assertions express properties which should hold (intended properties) while in others the assertions express properties which actually hold (actual properties) for the program. Also, it can be noted that in some cases it is the user who provides assertions to the tool whereas in other ones the assertions are generated by the tool.

Independently of these and other classifications, our aim is to design an assertion language which, in the context of CLP, *can be used for all the purposes mentioned above*, and hopefully new ones which may result from synergistic interactions due to the integration. The main application in which we will be using the assertion language within this book will be, of course, program debugging, but we expect the language to be useful in several other tasks.

There is a clear trade-off between the expressive power of a language of assertions and the difficulty in dealing with it. A reasonable overall objective when designing an assertion language is to try to maximize the expressive power of the language while at the same time keeping it amenable to automatic reasoning. More concretely, in the context of the DiSCiPl project, different tools for program development and debugging co-exist in the programming environment. In particular, Chapter 3 presents a preprocessor which performs combined compile-time and run-time checking of assertions, inference of assertions based on abstract interpretation, a form of diagnosis (location of errors), and, though not discussed there, also automatic documentation generation from assertions. The system presented in Chapter 3.7 also performs compile-time and run-time checking of assertions. Chapter 4 presents a system which allows locating errors in programs and uses assertions restricted to regular types. Finally, assertions could be used to replace the oracle in the declarative debugger presented in Chapter 5 (and even have some potential uses in the context of the visualizers described in later chapters).

We would like the assertion language to allow expressing any property which is of interest for any of the debugging (and validation) tools in the environment. Also, we would like the assertion language to be independent of the particular CLP platform in which it is applied and the constraint domains supported. Thus, we choose not to restrict too much beforehand the kind of properties which can be expressed with our assertions. A fundamental motivation behind this choice is the frequent availability in our target debugging environments of tools which can handle quite rich properties, through techniques such as approximations and abstract interpretation [2.9].

Clearly, not all tools will be capable of dealing with *all* properties expressible in our assertion language. However, rather than having different assertion languages for each tool, we propose the use of the same assertion language for all of them. This facilitates communication among the different tools and enables easy reuse of information, i.e., once a property has been stated there is no need to repeat it for the different tools. Each tool should then only make use of the part of the information given as assertions which the tool *understands* and should deal *safely* with the part of the information it does not understand.

Informally, a particular tool understands a given assertion if the tool can evaluate the assertion in the appropriate context and this evaluation has a chance of yielding true or false (i.e., it is not the case that it will always return "don't know"). For example, a program analysis for groundness of the computed answers typically understands an assertion stating that in all answers to a given predicate the second argument is ground. It also may understand an assertion stating that the same argument is a free variable (in the sense that it may be able to prove that the assertion is false). However, it will not understand an assertion which states that all calls to a given predicate must have a list as first argument: first, the tool is not able to reason about calls to predicates; second, it is not able to reason about "types."

We will present assertions which are able to capture "contexts" of the operational semantics as well as of the declarative semantics of CLP programs. Properties about the program execution states, of the computed answers, the correct answers, and of the computations themselves can all be expressed in our assertions. A preliminary version of the assertion language we present here appeared in [2.25].

In our assertion language, assertions are always instances of some *assertion schema* together with a reference to which part of the program (predicate or program point) the assertion refers to and, depending on the schema used, one or two *logic formulae*. Whereas the assertion language has a fixed set of assertion schemas, the user has a high degree of freedom for defining the logic formulae for the properties considered of interest. Thus, the whole assertion language is determined by a set of assertion schemas and the way in which "logic formulae" can be built. Intuitively, the logic formulae in the assertions are used to say things such as "X is a list of integers," "Y is ground," "$p(X)$ does not fail," etc. The (schemas of the) assertions specify which are the X's, Y's, and $p(X)$'s of which the previous things are said.

The structure of this chapter is the following. The role of assertions in program validation and debugging is further discussed in Section 2.2. Sections 2.3 through 2.6 present several assertion schemas available in our language. In more detail, Section 2.3 presents a series of assertion schemas which allow expressing properties related to execution states. Section 2.4 presents the syntax we use for logic formulae and also discusses some general issues on the evaluation of such formulae. While the assertion schemas presented in Section 2.3 allow expressing properties related to execution states and are thus operational, Section 2.5 introduces an assertion schema related to declarative properties of programs. In Section 2.6 we present assertion schemas which allow reasoning about completeness of the set of answers of a program. Section 2.7 shows how, independently of the schema used, the assertion language is made more expressive by adding a flag to each assertion which we refer to as an assertion "status." Section 2.8 presents yet another assertion schema which is conceptually different from all the ones seen in the previous sections. It allows expressing properties about whole computations of predicates rather than just states. In Section 2.9 we present how to define "property predicates", i.e., the predicates which are used as atomic formulae. These predicates are the building blocks from which logic formulae, and thus assertions, are written. Section 2.10 summarizes the assertion syntax and introduces some extensions to the assertion language which make the task of assertion writing easier. Finally, Section 2.11 discusses some issues about the proposed assertion language and concludes.

2.2 Assertions in Program Validation and Debugging

When reasoning about whether a certain program behaves as indicated by a set of assertions, it is often useful to restrict the discussion to a set of *valid* initial queries. This is because when we design a program not only do we have an expectation of what the program must compute but also we expect the program to be used by calling only certain predicates, and with some restricted class of input data. Thus, informally, a program is correct when it behaves according to the user's intention for any input data satisfying certain preconditions. We refer to such input data as *valid* input data, and to the corresponding queries as *valid queries*. The `entry` assertions which will be presented in Section 2.3.4 are a means for providing (a description of) the valid queries to the program. In what follows we assume that program debugging and validation is always performed w.r.t. a given set of (descriptions of) valid queries.

Assertion-based program validation and debugging aims at *automatically* reasoning about program correctness by checking whether assertions hold or not for a given program and a (set of) valid initial queries. In order to perform such reasoning automatically, some *inference system* is required which is capable of determining whether the assertions hold or not. Most existing assertion-based systems are designed with a fixed inference system in mind (for example, a particular type inferencing algorithm). Depending on the capabilities of such inference system, the assertion language is defined in such a way that every assertion expressible in the assertion language can be automatically determined to hold or not. In the design of the present assertion language we depart from most existing assertion languages in that we do not assume the existence of any fixed inference system. The main reason for this is the availability of a growing number of practical different static analyses for (constraint) logic programs which can perform the task of inference systems. We admit the possibility that different tools use different inference systems and, also, the same tool may make use of several inference systems. Thus, we cannot assume that the given assertions can always be proved nor disproved by any particular inference system. This on one hand allows having a very flexible assertion language since it is not limited to some kinds of properties. On the other hand, each tool must know how to safely deal with those assertions for which its inference system(s) cannot determine whether they hold or not. Though plenty of different inference systems may exist, we make a distinction between static inference systems and dynamic inference systems. The former systems are capable of reasoning about the program behaviour (and thus of the truth value of the assertions) without actually having to run the program, whereas the latter systems do run the program and check the assertions which are triggered by the execution of the program. In the design of the assertion language we have taken both kinds of inference systems into account, providing means to deal with the program properties expressed by the assertions either in dynamic or in static systems. We postpone the discussion on how this can be done, and how to safely deal with properties which a particular inference system does not understand until after (part of) the assertion language has been presented.

Very often, the properties of a program which we are interested in expressing by means of assertions are related to the run-time behaviour of the program. For this, we need to consider the *operational* semantics of the program. The operational semantics of a program is in terms of its *derivations* which are sequences of reductions between *execution states*. An execution state $\langle G \mid \theta \rangle$ consists of the current goal G and the current constraint store (or *store* for short) θ which contains information on the values of variables. The way in which a state is transformed into another one is determined by the operational semantics and the program code. The assertions presented in sections 2.3 and 2.8 refer to the operational behaviour of the program.

One of the advantages of CLP is that in addition to the operational semantics, programs also have a *declarative* meaning or semantics which is independent of the particular details on how the program is executed. Our assertion language also has assertions specifically designed for expressing properties related to such declarative semantics. These assertions are presented in Section 2.5.

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.

In this chapter we will present a repertoire of *assertion schemas*. Such schemas can be seen as templates which when properly instantiated define in a simple and clear way the required formulae app_A and sat_A. The choice of one schema or another greatly determines what the applicability contexts of the assertion may be. Thus, the use of assertion schemas on one hand makes the task of assertion writing easier, but on the other hand somewhat limits the freedom in describing in which contexts assertions are applicable. However, we argue that the proposed repertoire of schemas is flexible enough for the purposes for which the assertion language has been designed and we accept this limited freedom in return for the clarity of the resulting assertion language.[1]

2.3 Assertion Schemas for Execution 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. Typically, a program can be seen as composed of a set of *predicates* (also known as *procedures*). Alternatively, a program can be seen, at a finer-grained level, as composed of a set of *program points*. Thus, we first introduce several assertion schemas 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 ones as *program-point* assertions. Though a simple program transformation technique can be used to express program-point assertions in terms of predicate assertions, we maintain program-point 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

[1] A formal discussion on applicability contexts and schemas can be found in [2.26].

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.

Example 2.3.1. We illustrate the use of assertions about execution states with an example. Figure 2.1 presents a Ciao [2.3] program which implements the *quicksort* algorithm together with a series of both predicate and program-point assertions which express properties which the user expects to hold for the program.[2] Two predicate assertions are given for `qsort/2` (`A1` and `A2`) and another two for `partition/4` (`A4` and `A5`). There is also a program-point assertion (`A3`). The meaning of the assertions in this example is explained in detail in subsequent sections. 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.

2.3.1 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 account for probably one of the most common sorts of program properties which we may be interested in expressing in relation with program predicates. 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 (to be discussed in Section 2.4), 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." Referring to our notions of applicability and satisfiability sets, the resulting assertion can be interpreted as "the assertion is applicable in those execution states which correspond to success states of a computation of *Pred*, and its satisfiability set has those states in which *Postcond* holds."

[2] Both for convenience, i.e., so that the assertions concerning a predicate appear near its definition in the program text, and for historical reasons, i.e., mode declarations in Prolog or entry and trust declarations in PLAI [2.4, 2.23], we write predicate assertions as directives, which appear within the program text. Depending on the tool different alternatives may be used, including for example separate files or incremental addition of assertions in an interactive graphical environment.

```
:- calls qsort(L,R) : list(L).                                  % A1
:- success qsort(L,R) : list(L,int) => list(R,int).             % A2

qsort([X|L],R) :-
        check(number(X)),                                       % A3
        partition(L,X,L1,L2),
        qsort(L2,R2),
        qsort(L1,R1),
        append(R1,[X|R2],R).
qsort([],[]).

:- calls partition(L,X,L1,L2) : list(L).                        % A4
:- success partition(L,X,L1,L2)
         : ( list(L), ground(X) ) => ( list(L1), list(L2) ). % A5

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

Fig. 2.1. Some Assertions about Execution States

Example 2.3.2. 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 at 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).

2.3.2 Adding Preconditions to the Success Schema

The `success` schema can be used when the applicability set of an assertion is the set of success states for a given predicate. However, it is often useful to consider more restricted applicability sets. A classical example is to only consider 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." Alternatively, it can be interpreted as "the assertion is applicable to those execution states which correspond to success states of a computation of *Pred* which was originated by a calling state in which *Precond* holds, and its satisfiability set has those states in which *Postcond* holds." 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 2.3.3. The following assertion (`A2` in Figure 2.1) 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. Note that the program in Figure 2.1 can be used to sort a list of integers but also to sort a list of, say, floating point numbers. Thus, we cannot require in general that the result of ordering a list be a list of integers. This is why we add as precondition that the list to be sorted is indeed a list of integers.

2.3.3 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" [2.24]). 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." Alternatively, the resulting assertion can be interpreted as "the assertion is applicable in those execution states which correspond to calling states to *Pred*, and its satisfiability set has those states in which *Precond* holds."

Example 2.3.4. The following assertion (`A1` in Figure 2.1) 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).
```

2.3.4 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. This is for example the case also in traditional preconditions of a program. 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*." Alternatively, it can be interpreted as "the assertion is applicable in those execution states which correspond to initial queries to *Pred*, and the satisfiability set has those states in which *Precond* holds."

Example 2.3.5. 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) : numlist(L).
```

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` schemas 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) : numlist(L).`' 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 2.3.6. 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.

The name `entry` is used for historic reasons. Entry declarations have long been used (see for example [2.4]) in order to improve the accuracy of goal-dependent analyses since they allow providing a description of the *initial* calls to the program. Goal-dependent analyses may obtain results which are *specialized* (restricted) to a given context, which allows them to provide in general better (stronger) results than goal-independent analyses.

2.3.5 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. Thus, alternatively it can be interpreted as "the assertion is applicable in those execution states originated at the program point in which the assertion appears and its satisfiability set has those states in which *Cond* holds."

Example 2.3.7. 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 changing 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 checking. One possible such definition for run-time checking is presented in Chapter 3.

2.4 Logic Formulae about Execution States

As we have seen, schemas 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. In this section we present how such formulae are defined in our assertion language, and discuss how they should be evaluated.

We allow conjunctions and disjunctions in the formulae, and choose to write them down, for simplicity, in the usual CLP syntax. Thus, logic formulae about execution states can be:

- An atom of the form $p(t_1, \ldots, t_n)$ with $n \geq 0$, where p/n is a *property predicate*. How to define these predicates is explained in Section 2.9.
- An expression of the form $(F1,\ F2)$ where $F1$ and $F2$ are logic formulae about execution states and, as usual in CLP, the comma should be interpreted as conjunction.
- An expression of the form $(F1;\ F2)$ where $F1$ and $F2$ are logic formulae about execution states and, as usual in CLP, the semicolon should be interpreted as disjunction.

Such formulae have to be evaluated as part of the evaluation of an assertion. Evaluation of an assertion can be seen as composed of three steps. First, an appropriate inference system[3] 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

[3] As we will see in Chapter 3, even the underlying logic system may be used as inference system for evaluating the formulae.

has been applied on the assertion, and thus on AF, so that it refers to the corresponding variables of θ. We denote by $eval(AF, \theta, P, IS)$ the result of the evaluation of AF in θ by IS w.r.t. the definitions of property predicates P (the definition of property predicates in discussed in Section 2.9). The inference system must be correct in the sense that if $eval(AF, \theta, P, IS) = true$ then AF must actually hold in θ and if $eval(AF, \theta, P, IS) = false$ then AF must actually do not hold in θ. However, we also allow incompleteness of IS, i.e., $eval(AF, \theta, P, IS)$ 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 $eval(AF, \theta, P, IS) = AF$ 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 $eval(AF, \theta, P, IS)$ 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 [2.26] for a formal presentation of correctness and completeness w.r.t. these kinds of assertions).

In order to compute the value of $eval(p(t_1, \ldots, t_n), \theta, P, IS)$ 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. $false$). Since from the beginning we allow incompleteness of IS, it is important that the user can indicate property predicates which are *provable* and *disprovable* in a given inference system, together with the corresponding sufficient conditions. Our assertion language also provides means to do this, as explained in Section 2.9.

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 knowlegde 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 to validate the program w.r.t. the assertions [2.5], 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 preprocessor presented in Chapter 3, using abstract interpretation to compute the approximations.

2.5 An Assertion Schema for Declarative Semantics

As already mentioned, one of the main features of CLP is the existence of a *declarative semantics* which allows concentrating on *what* the program computes and not on *how* it should be computed. This feature is exploited for example in declarative debugging. Those tools which are concerned with the declarative semantics require the use of dedicated assertions. This is why we also have in our assertion language assertion schemas which refer to declarative semantics.

Consider the case of CLP(D), where D is the domain of values. For example, in classical logic programming D is the Herbrand Universe. In CLP($\Re$), D is the set of real numbers and of ground terms (for example lists) containing real numbers. The declarative semantics in CLP(D) associates a *meaning* to each program P, denoted $[\![P]\!]$, which corresponds to the *least D-model* of P. $[\![P]\!]$ is a set of D-atoms, where a D-atom is an expression $p(d_1, \ldots, d_n)$ with $n \geq 0$ such that p is an n-ary predicate symbol and $d_i \in D$. $[\![P]\!]$ coincides with $\mathrm{lfp}(T_P) = \bigcup_{i=0}^{\infty} T_P^i(\emptyset)$ where T_P is the immediate consequence operator.

We now introduce an assertion schema which allows stating properties which should hold in the least D-model $[\![P]\!]$ of a program. Otherwise the program is incorrect. This can be done using the schema:

```
:- inmodel Pred => Cond.
```

where *Pred* is a predicate descriptor and *Cond* is a logical formula about D-atoms. It should be interpreted as "in any D-atom $p(d_1, \ldots, d_n) \in [\![P]\!]$ whose predicate symbol coincides with that of *Pred*, *Cond* should hold." Alternatively, it can be interpreted as "the applicability set of the assertion has those D-atoms in $[\![P]\!]$ whose predicate symbol is that of *Pred* and the satisfiability set is the set of D-atoms whose predicate symbol is that of *Pred* which satisfy the property *Cond*."[4]

Example 2.5.1. The following assertion states that the result of ordering a list by means of the predicate `qsort` should be a list:

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

if we determine that the D-atom `qsort(a,a)` $\in [\![P]\!]$ then the program is not correct w.r.t. the above assertion A. This is because in `qsort(a,a)` A is applicable. However, $list(a)$ does not hold and thus `qsort(a,a)` is not in the satisfiability set of A.

As seen in the example above, this kind of assertions allows reasoning about (partial) correctness of programs. This is why we call them *correctness* declarative assertions. Note that they are apparently very similar to the `success` assertions presented in Section 2.3.1 since every success state of a predicate is in the declarative semantics of the program. In fact, a program which is correct w.r.t. an assertion '`:- inmodel` *Pred* `=>` *Cond*' is also correct w.r.t. the assertion '`:- success` *Pred* `=>` *Cond*' due to correctness of the operational semantics (but not vice versa due to possible incompleteness of the operational semantics). A further difference between `inmodel` and `success` assertions is that in `inmodel` it is not possible to add preconditions since the declarative semantics does not capture calls to predicates. In addition, depending on the semantics used, the logic formula used in *Cond* of `inmodel` assertions are not allowed to refer to the instantiation state of arguments, for example using `var/1`, whereas this is completely legal in `success` assertions.[5]

2.6 Assertion Schemas for Program Completeness

As seen above, there is a similarity between `success` and `inmodel` assertions in that they both express properties about the *answers* of predicates. More precisely, `success` assertions express properties of the *computed answers*

[4] Any D-atom p with the predicate symbol of $Pred$ satisfies $p = \mu(Pred)$ for a substitution μ. Strictly speaking, it is $\mu(Cond)$ which is satisfied on p. For simplicity of the presentation, we have prefered not to clatter it with these technical details.

[5] However, this could be remedied by using a more powerful declarative semantics (e.g. [2.1]), in which for example using `var/1` would make perfec sense.

of predicates, i.e., those generated by the operational semantics, whereas `inmodel` assertions refer to *correct answers*, i.e., those which are in the declarative semantics of the program (its least D-model). When considering answers to predicates, one particular aspect to reason about is *correctness* of the program, which corresponds to answering the question: Are all the actual answers of the program in the set of intended answers? Conversely, another aspect we can reason about is the well known concept of *completeness* of the program, which corresponds to answering the question: Are all the intended answers of the program in the set of actual answers? In other words, a program is complete when it does not fail to produce any expected answer. Clearly, we would like our program to be both correct and complete w.r.t. our intention. This corresponds to the classical notion of *total correctness*, as opposed to the previous notion of correctness, which is also known as *partial correctness*.

Though not explicitely mentioned, all the assertions presented in the previous sections allow reasoning about (partial) correctness of programs w.r.t. assertions, i.e. the may allow detecting that a program is not partially correct w.r.t. the assertion or validating the program w.r.t. the assertion. However, they are of no use in order to reason about completeness of programs. This is why we now introduce another kind of assertions which are variations of the `inmodel` and `success` assertions. They can be distinguished from the previous ones because the arrow (`=>`) now points in the reverse direction, i.e., `<=`. For example, an assertion of the form:

`:- inmodel` *Pred* `<=` *Cond*.

(note the reversed direction in the arrow) should be interpreted as "any D-atom of the form $p(d_1, \ldots, d_n)$ whose predicate symbol is the same as that in *Pred* and on which *Cond* holds should be in $[\![P]\!]$."

Example 2.6.1. The following assertion where the symbol `==` stands for term identity states that the pair $\langle$`[2,1]`,`[1,2]`$\rangle$ is an expected solution of `qsort`.:

```
:- inmodel qsort(L,R) <= (L == [2,1], R == [1,2]).
```

If we can determine that $qsort([2,1],[1,2]) \notin [\![P]\!]$ then P is incomplete w.r.t the above assertion. This is an indication that the existing code for `qsort` does not allow determining that one (in this case the only) result of ordering `[2,1]` is `[1,2]`. It is thus an indicator that the current version of the program is not complete w.r.t. the above assertion. This is clearly the symptom of an error, but it can be the case that the program sorts correctly lists of length different from two, in which case the error cannot be detected automatically using the (partial correctess) assertions seen in the previous sections.

We can also write completeness assertions for operational semantics using the following schema (optional "fields" appear in square brackets):

`:- success` *Pred* [`:` *Precond*] `<=` *Postcond*.

which should be interpreted as "any call to predicate *Pred* which on the calling state satisfies *Precond* must have as success states at least all those states which satisfy *Postcond*."

Example 2.6.2. Consider the following program which aims at improving the previous version of `qsort` by stopping recursion when the list has length 1 since any such list is already sorted. We add to the program a completeness success assertion:

```
:- success qsort(L,R): (L==[], var(R)) <= R == [].
qsort([X,Y|L],R) :-
        partition([Y|L],X,L1,L2),
        qsort(L2,R2),
        qsort(L1,R1),
        append(R1,[X|R2],R).
qsort([A],[A]).
```

which requires that the results of sorting the empty list include the empty list, provided that the second argument satisfies `var/1` which holds iff it is a free, unconstrained variable at the call. The precondition `var(X)` is needed since, for example, the call '`qsort([],[1])`' has no success state.

The assertion above can be used to detect that the program is not complete since we have forgotten the clause '`qsort([],[]).`', thus a call such as '`qsort([],L).`' fails without producing any answer.

2.7 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 an actual property. The difference with status `true` introduced below is that this information is given by the user and it may not be possible to infer it automatically.

– 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 Chapter 3) 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.

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 2.3.5 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`). See Section 2.10.1 for syntactic details. Again, if we want to execute a program with program-point assertions we can simply define the predicate corresponding to the status so that it always succeeds. For example, if the status is `true` we then define the predicate `true/1` (resp., `trust/1`, `checked/1`, `false/1`) as "`true(_)..`"

Example 2.7.1. Figure 2.2 presents the same program as in Figure 2.1 but rather than with `check` assertions, with both predicate and program-point `true` assertions which express analysis results. The results have been generated by CiaoPP [2.6, 2.16] using goal-dependent mode analysis. Predicate and/or program-point assertions may be generated according to the user's choice. Program-point assertions contain information for each program point and are literals of the `true/1` predicate. Regarding predicate assertions, for conciseness, compound (`pred`) predicate assertions are usually produced by the analyzer. Compound assertions will be introduced in Section 2.10.2.

```
:- entry qsort(L,R) : ground(L).
:- true pred qsort(A,B) : ground(A) => ( ground(A), ground(B) ).

qsort([X|L],R) :-
        true((ground([L,X]),var(L1),var(L2),var(R1),var(R2))),
        partition(L,X,L1,L2),
        true((ground([L,L1,L2,X]),var(R1),var(R2))),
        qsort(L2,R2),
        true((ground([L,L1,L2,R2,X]),var(R1))),
        qsort(L1,R1),
        true(ground([L,L1,L2,R1,R2,X])),
        append(R1,[X|R2],R),
        true(ground([L,L1,L2,R,R1,R2,X])).
qsort([],[]).

:- true pred partition(A,B,C,D)
         : ( ground(A), ground(B), var(C), var(D) )
        => ( ground(A), ground(B), ground(C), ground(D) ).

partition([],B,[],[]).
partition([E|R],C,[E|Left1],Right) :-
        true((ground([C,E,R]),var(Left1),var(Right))),
        E<C, !,
        true((ground([C,E,R]),var(Left1),var(Right))),
        partition(R,C,Left1,Right),
        true(ground([C,E,Left1,R,Right])).
partition([E|R],C,Left,[E|Right1]) :-
        true((ground([C,E,R]),var(Left),var(Right1))),
        E>=C,
        true((ground([C,E,R]),var(Left),var(Right1))),
        partition(R,C,Left,Right1),
        true(ground([C,E,Left,R,Right1])).
```

Fig. 2.2. Analysis results expressed as assertions

Though both `true` and `trust` assertions refer to properties of the actual program it is important to see that they are not equivalent. As alredy mentioned, `true` assertions are generated by analysis and are automatically provable, whereas `trust` assertions are provided by the user and often they are not provable either because part of the program is not available or because analysis is not powerful enough. In any case, analysis is instructed to *trust* such assertions.

The status `trust` receives this name for historical reasons. In [2.4] `trust` declarations were already used in order to provide the analyzer with additional information so that it could improve its results. Note that a `trust` assertion for a predicate p may not only improve the analysis information for the predicate p (if the information it contains is better than that generated by analysis) but also it may allow improving the analysis information on other predicates which depend on p.

Example 2.7.2. Consider the following program in which the definition of predicate `r/2` is not available but where there is a `trust` assertion which states that upon success of `r(A,B)`, `B` is a list provided that `A` was a list on call:

```
:- entry p(X,Y).
:- trust success r(A,B) : list(A) => list(B).

p(X,Y):- q(X), r(X,Y).

q([])
q([_|Xs]):- q(Xs).
```

Assume we are using a goal-dependent type analyzer. The `entry` assertion for predicate `p/2` informs the analyzer that such predicate can be subject to initial queries. In addition, the analyzer assumes that the set of existing `entry` assertions cover all possible initial queries. Since there is no `entry` for predicates `q` nor `r`, the analyzer assumes that these predicates cannot be called in initial queries. Without the `trust` assertion, typically analysis would infer the following call and success types for predicates `p` and `r`:

```
:- true pred p(A,B) : (term(A),term(B)) => (list(A),term(B)).
:- true pred r(A,B) : (list(A),term(B)) => (list(A),term(B)).
```

where the type `term` represents the set of all possible terms. However, by exploiting the information in the `trust` assertion analysis can conclude that:

```
:- true pred p(A,B) : (term(A),term(B)) => (list(A),list(B)).
:- true pred r(A,B) : (list(A),term(B)) => (list(A),list(B)).
```

Note that not only the information on `r` has been improved, but also that of `p` since it depends on `r`.

It is important to mention that even though `trust` assertions are trusted by the analyzer to improve its information unless they are incompatible with the information generated by the analyzer (see [2.4]), they may also be subject to run-time checking. The translation scheme for assertions with the status `trust` is exactly the same as the one given in Chapter 3 for assertions with the status `check`. It should be an option whether only `check` assertions or both `check` and `trust` assertions should be checked at run-time.

A similar situation happens with `entry` assertions (presented in Section 2.3.4). If analysis is goal-dependent, it assumes that the entries hold, and thus all the information generated is correct under this assumption, but such information may become incorrect if the run-time queries do not conform to the existing `entry` assertions. Thus, in order to guarantee that the results of static checking and/or program optimization based on analysis results are sound we may also check entries at run-time. As in the case of `trust` assertions, it should be an option of the compiler whether to perform run-time checking of `entry` assertions or not.[6]

2.8 An Assertion Schema for Computations

Though the assertions already presented for operational semantics, declarative semantics, and for reasoning about completeness are very useful, there are many other interesting properties of programs which cannot be expressed using the presented assertion schemas. This is why we introduce yet another assertion schema named `comp`, which relates to *computations*, where by computation we mean the (ordered) execution tree of all derivations of a goal from a calling state.

The `comp` schema is, in the same way as `success` and `calls` schemas, associated to predicates and is inherently operational. The `success` and `calls` schemas 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 schemas. 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: non-failure, termination, determinacy, non-suspension, non-floundering, 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 2.8.1. 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 unconstraint variable at the calling state should produce

[6] The introduction of run-time tests into the original program is analogous to that performed for `calls` assertions but is only applied to initial calls to the program (via a transformation of the program which renames apart the internal calls).

at least one solution in finite time (the property of the computation `succeeds` will be further discussed in Section 2.9.3):

```
:- 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.

2.8.1 Logic Formulae about Computations

Similarly to logic formulae about execution states, logic formulae about computations can be conjunctions and/or disjunctions of formulae, where the atomic formulae are property predicates (about computations). As before, conjunctions and disjunctions are written in CLP syntax (i.e., are commas and semicolons, respectively).

As in the case of logic formulae about execution states, given a `comp` assertion for *Pred* with logic formula *Comp-prop* on computations and an execution state for a goal of the predicate of *Pred* in calling store θ, we first apply a renaming on the assertion which relates the variables in *Pred* with the variables in θ. Then we evaluate each atomic formula AFC in *Comp-prop* and the evaluation of *Comp-prop* is obtained by composing the values obtained for each AFC and simplifying the resulting expression. As before, we denote by $eval(AFC, \theta, P, IS)$ the evaluation of AFC in θ by IS w.r.t. the definition of properties P. Note that, in general, properties of the computation cannot be decided by looking at the store θ alone, as it is the case with properties of execution states. Thus, IS may need to reconstruct (part of) the computation of a particular instance of $Pred$, according to the calling store θ, in order to decide whether AFC holds or not. We assume that P contains, in addition to the definition of the property predicates, the program under consideration, so that reconstructing the computations is possible. Since the necessary part of the computation required may be an infinite object, it is possible that the process of reconstructing such computation does not terminate, in which case $eval(AFC, \theta, P, IS)$ will not terminate either. Thus, we admit that the evaluation of an atom of a property of the computation AFC, in addition to returning *true*, *false* or AFC (if IS cannot decide the property), may also not terminate, precisely in those cases in which the execution of $\langle Pred \mid \theta\rangle$ does not terminate either. We argue that this is admissible since the evaluation of the property still does not introduce non-termination, in the sense that if the program execution terminates the evaluation of properties will also terminate.

Example 2.8.2. Consider again the `comp` assertion in Example 2.8.1. Consider also that during dynamic checking of such assertion we reach an execution state of the form $\langle$`qsort(X,Y)` :: *Goal* $\mid \theta\rangle$, i.e., `qsort(X,Y)` is the first literal to solve and *Goal* is a (possibly empty) conjunction of literals, where θ indicates that `X` takes the value `[1,5,3]` and `Y` is an unconstrained free variable. In such state our `comp` assertion is applicable since $eval($`list([1,5,3],int)`$, \theta, P, IS)$ and $eval($`var(Y)`$), \theta, P, IS)$, where P must contain at least the definition of the parametric property `list/2` (and of `var/1` if it were not a builtin predicate), can be proved to take the value *true* using an appropriate IS. In order to see whether the assertion is satisfiable we have to compute the value of $eval($`succeeds(qsort([1,5,3],Y))`$, \theta, P, IS)$, where P must contain at least the definition of `succeeds/1` and also of `qsort/2` and of all other predicates it uses, in this case `partition/4` and `append/3`. In our example the computation which has to be reconstructed is $\langle$`qsort([1,5,3],Y)` $\mid \theta\rangle$, which is finite. Thus, checking the property should terminate.

2.9 Defining Property Predicates

All our assertion schemas are parameterized on logic formulae for expressing the particular properties of the execution states (in `calls` and `success` assertions), of the correct answers (in `inmodel` assertions), and of the computations (in `comp` assertions). Atoms in such formulae are of the predicates which we call *property predicates* (or *properties* for short when the context is clear enough). In this section we discuss how to define such property predicates. We have not presented the property predicates allowed in our assertion language yet because the assertion language is parametric w.r.t. the set of property predicates of interest. Thus, rather than having a fixed set of such predicates, we allow users to define their own properties in a very flexible way.

Since we have assumed that our source language is a logic and/or constraint logic programming language, in which it is natural to define predicates, it also seems natural to use the underlying CLP language to define the property predicates. This design decision has very important implications: (1) The user does not need to learn a new language for defining property predicates since the same language used for writing programs can be used. (2) This makes the assertion language extremely expressive since the user can define almost any predicate property that is considered of interest when dealing with a particular program. (3) With a little run-time support, atomic logic formulae in assertions can be evaluated by simply executing them on the underlying CLP system. This can be seen as taking the underlying CLP system as an inference system.[7] (4) Though the assertion language may remain decidable for run-time checking under some sensible restrictions on the property predicates used (such as that their execution always terminates), there

[7] The use of *executable* properties which can be checked dynamically, i.e., by executing the code defining them, has also been proposed in the context of declarative debugging [2.11].

is little hope that the assertion language remains decidable for compile-time checking given any fixed set of static inference systems available, as we allow users to define their own predicates. This is why systems designed with particular inference systems in mind generally only allow using a predefined set of property predicates or they have a very restricted language for defining new property predicates, which is a restriction that we want to lift.

2.9.1 Declaring Property Predicates

Since the set of property predicates is not fixed in our assertion language, in order to be able to perform some syntactic checking on the assertions given for a program, i.e., to check whether they are consistently written, we require that all predicates which can be used as property predicates are declared as such using an assertion of the form:

`:-` [*IS_List*] `prop` *Pred-Spec*`.` or `:-` [*IS_List*] `cprop` *Pred-Spec*`.`

where *Pred-Spec* is a term of the form p/n, where p is a predicate symbol and n its arity. They indicate, respectively, that the predicate can be used in logic formula about states or about computations. Also, the (optional) field *IS_List* contains a list of the inference systems in which *Pred* is provable. We consider that it is not required to indicate whether the underlying CLP system can be used to prove the property or not: if a predicate property is defined in the CLP source language, such definition is assumed to be exact and thus the underlying CLP system can be used to decide whether the property holds or not. As further discussed in Chapter 3, we impose some restrictions on the code which defines property predicates. Thus, we also assume that the definition in source code of any property predicate for which a `prop` or `cprop` declaration exists satisfies such restrictions.

All property predicates which may appear in the logic formulae of the assertions given for the program must be declared, independently of whether a definition as source code is given for them or not. In particular, we should declare as property predicates those ones for which there is an inference system which partially captures (or is complete for) them and which we intend to use in assertions.

Example 2.9.1. Consider the property predicate about states `ground/1`. An argument satisfies this property if it is bound to a Herbrand term without variables. This property is often provable using static inference systems which reason about variable groundness. Consider now that in our tool there are two different inference systems called *sharing* and *def* which are capable of proving the property `ground/1`. This should be indicated with a declaration like '`:- [sharing, def] prop ground/1.`'. In addition, it is likely that a builtin (or library) predicate with the same name and meaning exists in the CLP system. In that case, the property can be decided during dynamic checking using the underlying CLP system. If it is not a builtin we could write a definition of such predicate in CLP, as further discussed below.

2.9.2 Defining Property Predicates for Execution States

In this section we discuss by means of examples several issues related to the definition of property predicates about execution states by means of CLP programs. We start with an example.

Example 2.9.2. Consider the following definition of the property predicate `list/1` by means of a regular program [2.32, 2.10]:

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

The above definition can be used to dynamically decide whether a term is of type `list` or not. This case is also interesting because if an inference system which captures regular types (which are further discussed in this book in Chapter 4) is available, then it may be able to prove such property statically by using the code above, which the inference system can safely handle as a type declaration. This should be indicated with the assertion '`:- regtype prop list/1.`' assuming that `regtype` is the name of the inference system for regular types. This is an example of how users can define new regular types in our assertion language by means of a (regular) CLP program.

A distinguishing feature of (constraint) logic programming w.r.t other programming paradigms such as functional or imperative programming is that, in a given execution state, the values of certain program variables may be (partially) undefined. This is a consequence of the existence of "logical variables" whose value may be further instantiated during forward execution. This feature has to be taken into account when defining properties of execution states.

Example 2.9.3. In the definition of property `list` in Example 2.9.2 above it is not obvious which one of the following two possibilities we mean exactly: "the argument is *instantiated* to a list" (let us indicate this property with the property predicate `inst_to_list`), or "if any part of the argument is instantiated, this instantiation must be compatible with the argument being a list" (we will associate this property with the property predicate `compat_with_list`). For example, `inst_to_list` should be true for the terms `[]`, `[1,2]`, and `[X,Y]`, but should not for `X` and `[a|X]`. In turn, `compat_with_list` should be true for `[]`, `X`, `[1,2]`, and `[a|X]`, but should not be for `[a|1]` and `a`.

We refer to properties such as `inst_to_list` above as *instantiation properties* and to those such as `compat_with_list` as *compatibility properties* (corresponding to the traditional notions of "instantiation types" and "compatibility types"), and to the corresponding property predicates as instantiation and compatibility property predicates.

It turns out that both of these notions are quite useful in practice. Consider for example a definition of the well known predicate `append`:

```
append([],L,L).
append([X|Xs],L,[X|NL]):- append(Xs,L,NL).
```

For this predicate we probably would like to use `compat_with_list` to state that in all calls to `append` all three arguments must be compatible with lists in an assertion like:

```
:- calls append(A,B,C) :
   (compat_with_list(A), compat_with_list(B), compat_with_list(C)).
```

With this assertion, no error will be flagged for a call to `append` such as `append([2],L,R)`, since `L` can be instantiated to a list later on the execution, but a call `append([],a,R)` would indeed flag an error.

On the other hand, we probably would like to also use `inst_to_list` to describe the type of calls for which `qsort` has been designed, i.e., those in which the first argument must indeed be a list. This was done for example in assertion `A1` of Figure 2.1:

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

i.e., here we clearly wanted `list(L)` to mean `inst_to_list(L)`.

Since both kinds of properties are properties of interest, one possibility is to define, in each case, two distinct property predicates, one of each kind. This will force us to define both `inst_to_list` and `compat_with_list` in different ways.

Example 2.9.4. A possible definition of `inst_to_list` is the following:

```
:- prop inst_to_list/1.

inst_to_list(X) :-
       nonvar(X), inst_to_list_aux(X).

inst_to_list_aux([]).
inst_to_list_aux([_|T]) :- inst_to_list(T).
```

However, one would like that the more natural definition of `list` of Example 2.9.2 could be used both as an instantiation and as a compatibility property, by simply instructing the system to handle the definition of the property in the appropriate way. We introduce a mechanism in our assertion language for this purpose. Properties about execution states are interpreted by default as instantiation properties. Thus, writing:

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

has the desired effect. If we are interested in using the definition of a predicate *Property* as a compatibility property, we should indicate it in the formula as `compat(`*Property*`)` and it will automatically be interpreted as: "*Property* holds in the current store or it can be made to hold by adding bindings (or constraints) to the current store." Thus, writing:

```
:- calls append(A,B,C)
        : ( compat(list(A)), compat(list(B)), compat(list(C)) ).
```

also has the desired effect.

This allows to define property predicates as compatibility properties, and use them as either an instantiation or a compatibility property on states. However, note that if the definition of a state property predicate *Prop* contains certain impure builtin predicates which explicitly (e.g., `nonvar`, `var`) or implicitly (e.g., `integer`, `atom`, `>`) perform some degree of instantiation checking, it may not be correct to use *Prop* as a compatibility property.

On the other hand, note that the definition of `list` would not behave as expected either if used as an instantiation property as is. This is because if we execute it in a store which is not sufficiently instantiated, execution will succeed and add the necessary bindings or constraints for the property to hold rather than failing, which is what the instantiation property should do. On the contrary, `list` behaves as expected of a compatibility property. Nonetheless, we have preferred to consider properties such as `list` instantiation properties by default. This is because we prefer to put the burden of interpreting the definition of properties as instantiation properties on the inference system, rather than putting such burden on the users by forcing them to write the instantiation property explicitly. We argue that this is a natural choice, since in most cases writing down the definition of the compatibility property is easier, but (at least in our experience) in a large number of logic formulae about execution states users are interested in the instantiation property rather than in the compatibility property.

2.9.3 Defining Property Predicates for Computations

Since existing (constraint) logic programming systems have meta programming facilities which allow having atoms (calls to predicates) appearing as arguments of calls to other predicates, it is in principle possible to use the underlying programming language in order to also define property predicates for computations. However, this is not as easy as defining property predicates for execution states.

Example 2.9.5. Consider the property of the computation `succeeds(`*Pred*`)` which is to be interpreted as "the computation of *Pred* in the current store produces at least one solution in finite time." This property, which when appears in a `comp` assertion has no arguments, has to be defined in CLP as a predicate with one argument (which receives the goal on whose computation we aim at checking the property). The following definition could be used:

```
succeeds(Goal):- call(Goal).
```

Although defining the above predicate property in CLP has not been very difficult, in general, defining a property of a computation in source code requires to actually perform the computation (which in the example above is done using the `call` builtin predicate). This is possible since checking the property does not need to observe intermediate states of the computation of `qsort(A,B)` (with the values of `A` and `B` according to the store) and it suffices to check whether the call succeeds or not. For other properties which may need observing internal states of the execution we may need to program a meta-interpreter. This has the disadvantage that the property being defined may end up being rather obscure.

An additional difficulty in expressing property predicates for computations as CLP predicates is that many of the interesting properties of computations we may want to write are by nature undecidable. Thus, it is just not possible to provide an effective definition of such properties which can be executed during dynamic checking. A good example of this is trying to define a predicate `terminates/1` which decides whether computation of a predicate (universally) terminates or not. Clearly, it is not possible to provide a general definition of such predicate. Thus, `comp` assertions are often not amenable to dynamic assertion checking, since it is difficult to write the properties involved in such a way that they can be executed efficiently and non trivial results obtained. However, we still include assertions about properties of computations in our language for several reasons: (1) when considering static assertion checking, it is often the case that there are analysis which obtain useful results on properties which are undecidable in general. This is because such analyses use sufficient conditions which guarantee that the properties hold of the program. For example, a good number of termination analyses exist which can prove termination of a high percentage of terminating programs, though there will always be terminating programs which cannot be proved to terminate. (2) In other cases, approximate definitions in source code can often be provided for dynamic and/or static checking of properties of the computation. This is discussed in the following section. (3) They are also useful for expressing the results of static analyses which infer properties of computations such as termination, non-failure, determinacy,... (4) They can be used for expressing properties which we do not aim at checking but rather they are interesting for documentation purposes. An example of this is the Ciao reference manual [2.3] where, for example, the assertion '`:- trust comp write/1 + iso.`' indicates that the implementation of predicate `write/1` behaves according to the ISO Prolog specification.

2.9.4 Approximating Property Predicates

As already mentioned, in our assertion language we do not require an exact definition of every predicate property used. This is useful for at least two reasons. One reason is that there are a good number of interesting properties which are not decidable and for which it is just not possible to provide exact definitions using the source CLP language. As discussed above, this is often the case when trying to define property predicates about computations.

Another reason is that sometimes the user prefers not to provide an exact definition because even though it is possible, it is a hard or tedious task. However, it is very simple to provide an approximate but still useful definition. The user may decide to provide an exact definition at a later point in time if so desired.

Our assertion language, rather than restricting the set of property predicates to those for which an exact definition in the source language is provided or simply returning AF as the result of $eval(AF, \theta, P, IS)$ if IS does not capture the property AF, it allows providing *approximations* of such property predicates AF. Such approximations provide sufficient conditions for returning *true* (proving AF) or *false* (disproving AF). These approximations are given using assertions of the form:

`:- proves` *Pred* `:` *Cond*`.` or `:- disproves` *Pred* `:` *Cond*`.`

where *Pred* is a property predicate descriptor and *Cond* a logic formula about states or about computations. They indicate that given a current store θ if we can prove that *Cond* holds in θ (using any inference system) then we have also proved that *Pred* holds in θ, if a `proves` assertion is given, or that *Pred* does not hold in θ, if a `disproves` assertion is given.

Example 2.9.6. Consider a static inference system called `simple_stat` which is capable of determining that at certain program points some arguments are bound to integer numbers. We can use the following declarations:

```
:- simple_stat prop integer/1.

:- proves ground(X) : integer(X).
:- disproves var(X) : integer(X).
```

in order to indicate that (1) `simple_stat` can prove the state property `integer/1` and (2) the fact that we can establish that some argument is an integer number can be used to guarantee that such argument is ground and also to prove that such argument is definitely not a free variable. Note that the `proves` and `disproves` assertions are independent of any inference system. This means that they would also be useful if some other inference system were able to prove the property `integer/1`. Thus, though the predicate properties `ground` and `var` are not directly inferred by `simple_stat` we can use them in assertions and still be able to either prove or disprove such assertions, in this case statically, using `simple_stat`.

2.10 Syntax of and Extensions to the Assertion Language

In this section we provide a summary of the syntax of assertions. We then introduce another predicate assertion schema which can be used in addition to the ones introduced previously. It can be seen as syntactic sugar for a set of predicate assertions. Finally we comment on some other syntactic sugar which facilitates the writing of assertions.

2.10.1 Syntax of the Assertion Language

We now summarize the syntax of the assertions presented with the following two formal grammars. The first one defines the syntax of program assertions, from the non-terminal *program-assert*:

program-assert	::=	*predicate-assert*
		| *prog-point-assert*
predicate-assert	::=	`:-` *stat-flag pred-assert* `.`
		| `:-` *entry* `.`
pred-assert	::=	`calls` *pred-cond*
		| `success` *pred-cond direction state-log-formula*
		| `inmodel` *pred-desc direction state-log-formula*
		| `comp` *pred-cond* `+` *comp-log-formula*
entry	::=	`entry` *pred-cond*
pred-cond	::=	*pred-desc*
		| *pred-desc* : *state-log-formula*
pred-desc	::=	*Pred-name*
		| *Pred-name*(*args*)
args	::=	*Var*
		| *Var*, *args*
state-log-formula	::=	(*state-log-formula* , *state-log-formula*)
		| (*state-log-formula* ; *state-log-formula*)
		| `compat`(*State-prop*)
		| *State-prop*
comp-log-formula	::=	*comp-log-formula* , *comp-log-formula*
		| *comp-log-formula* ; *comp-log-formula*
		| *Comp-prop*
stat-flag	::=	*status*
		| ϵ
status	::=	`check`
		| `true`
		| `checked`
		| `trust`
		| `false`
direction	::=	`=>`
		| `<=`
prog-point-assert	::=	*status*(*state-log-formula*)

There are some non-terminals in the grammar which are not defined. This is because they are constraint-domain and/or platform dependent. They can be easily distinguished in the previous grammar because their name starts with a capital letter:

Pred-name As we are interested in having an assertion language which looks homogeneous with the CLP language used, we admit as *Pred-name* any valid name for a predicate in the underlying CLP language. Usually, non-empty strings of characters which start with a lower-case letter.

Var It corresponds to the syntax for variables in the CLP language. Usually, non-empty strings of characters which start with a capital letter. As mentioned before, it is assumed that all variables in the same predicate description are distinct.

State-prop An atom of a `prop` property predicate.

Comp-prop An atom of a `cprop` property predicate.

The following grammar defines the syntax of assertions for declaring property predicates, from the non-terminal *prop-assert*:

prop-assert	::=	*prop-exp*
		| *approx-exp*
prop-exp	::=	`:-` *is-flag prop pred-spec* `.`
prop	::=	`prop`
		| `cprop`
is-flag	::=	`[` *is-idlist* `]`
		| *Is-id*
		| ϵ
is-idlist	::=	*Is-id* `,` *is-idlist*
		| *Is-id*
pred-spec	::=	*Pred-name*/*Number*
approx-assert	::=	`:-` *approx approx-exp* `.`
approx	::=	`proves`
		| `disproves`
approx-exp	::=	*State-prop* `:` *state-log-formula*
		| *Comp-prop* `:` *comp-log-formula*

The new non-terminals in the grammar are as follows:

Is-id A constant of the language which uniquely identifies an inference system in the debugging system being used.

Number A number (which is meant to be the arity of a predicate).

2.10.2 Grouping Assertions: Compound Assertions

The motivation for introducing compound assertions is twofold. First, when more than one `success` (resp. `comp`) assertion is given by the user for the same predicate, in the user's mind this set is usually meant to cover all the

Field	Translation if given	Otherwise
`=>` *Postcond*	`success` *Pred* : *Precond* `=>` *Postcond*	∅
`+` *Comp-prop*	`comp` *Pred* : *Precond* `+` *Comp-prop*	∅

Table 2.1. Transforming compound into basic assertions.

different uses of the predicate. In such cases, the disjunction of the preconditions in all the `success` (resp. `comp` assertions) is often a description of the possible calls to the predicate. However, the user would have to explicitly write down a `calls` assertion to express this. It would be desirable to have the `calls` assertion be automatically generated in such cases for the set of assertions, rather than having to add it manually. Compound assertions allow this. Second, a disadvantage of the assertion schemas presented in sections 2.3 and 2.8 is that it is often the case that in order to express a series of properties of a predicate, several of them need to be written.

Each compound assertion is translated into one, two, or even three basic predicate assertions, depending on how many of the fields in the compound assertion are given. Compound assertions are built using the `pred` schema, which has the form:[8]

`:- pred` *Pred* [`:` *Precond*] [`=>` *Postcond*] [`+` *Comp-prop*]`.`

Example 2.10.1. The following assertion indicates that whenever we call `qsort` with the first argument being a list, the computation should terminate and if the computation succeeds, on termination the second argument should also be a list.

```
:- pred qsort(L,R) : list(L) => list(R) + terminates.
```

in addition, if this is the only `pred` assertion given for predicate `qsort`, then it also indicates that all calls to `qsort` should have a list in the first argument.

Table 2.1 presents how a compound assertion is translated into basic `success` and `comp` assertions. Generation of `calls` assertions from compound assertions is more involved, as the set of all compound assertions for one predicate must cover all possible calls to that predicate. Thus, if the set of compound assertions for a predicate *Pred* is $\{A_1, \ldots, A_n\}$, let $A_i =$*Pred* : C_i [`=>` S_i] [`+` $Comp_i$], then the (only) `calls` assertion which is generated is

`:- calls` *Pred*: $\vee_{i=1}^{n} C_i$.

Example 2.10.2. Consider the two following compound assertions for predicate `qsort`:

[8] Note that the syntax grammar presented previously does not incclude this extension.

```
:- pred qsort(A,B) : numlist(A) => numlist(B) + terminates.
:- pred qsort(A,B) : intlist(A) => intlist(B) + terminates.
```

The `calls` basic assertion which would be generated is:

```
:- calls pred qsort(A,B) : (numlist(A) ; intlist(A)).
```

Note that when compound assertions are used, a `calls` assertion is always implicitly generated. If we do not want the `calls` assertion to be generated (for example because the set of assertions available does not cover all possible uses of the predicate) basic `success` or `comp` assertions rather than compound (`pred`) assertions should be used.

2.10.3 Some Additional Syntactic Sugar

There are a number of syntactic sugar conventions in the assertion language which can be added to facilitate the writing of assertions. We mention here, by means of examples, some of the most interesting ones.[9]

Abridged syntax. With this syntax it is not necessary to explicitly mention which argument of the predicate descriptor the logic formulae refer to. The different arguments are identified by position. Individual logic formulae are separated by * and they refer to the predicate arguments by order. For example:

```
:- calls qsort(A,B) : list * term.
:- calls qsort/2 : list * term.
```

are both equivalent to:

```
:- calls qsort(A,B) : ( list(A), term(B) ).
```

When there is the need for associating two or more properties to the same argument then the following syntax may be used:

```
:- calls qsort/2 : { list, ground } * term.
```

which is equivalent to:

```
:- calls qsort(A,B) : ( list(A), ground(A), term(B) ).
```

The abridged syntax can be mixed with the normal syntax in a given assertion, provided that each "field" of the assertion is written using only one syntax. For example:

```
:- success qsort(A,B) : list * term => list(B).
```

[9] Note that the syntax grammar presented previously does not incorporate these extensions.

Compatible properties. In some cases, the programmer wants to specify compatibility properties of some arguments both at the call and success states of the predicate. To avoid repeating the properties, the syntax of the following example can be used:

```
:- pred qsort(A,B) :: ( list(A), list(B) ).
```

which is equivalent to:

```
:- pred qsort(A,B) : ( compat(list(A)), compat(list(B)) ) =>
                     ( compat(list(A)), compat(list(B)) ).
```

This kind of writing can also be "in-lined" into the predicate arguments. For example, the following assertion is equivalent to the two ones above:

```
:- pred qsort(list,list).
```

Modes. They allow specifying in a compact way several properties which refer to one argument. Thus, modes can be seen as property macros. For example, provided that the following mode definition exists:

```
:- modedef out(X) : var(X) => ground(X).
```

then, instead of the first assertion below, the second one could be written, which is equivalent:

```
:- pred qsort(A,B) : var(B) => ground(B).
:- pred qsort(A,out(B)).
```

Our assertion language generalizes the classical concept of modes, allowing users to define their own. For example, the classical Prolog modes "+" (i.e., the corresponding argument is a non-variable on input) and "-" (the argument is a variable on input) can be expressed in our language by defining them as:

```
:- modedef '+'(X) : nonvar(X).
:- modedef '-'(X) : var(X).
```

Mode syntax can be mixed with any other syntax, as in the first assertion below (which is equivalent to the second one):

```
:- pred qsort(list,out(B)) => list(B).
:- pred qsort(A,B) :: list(A) : var(B) => ( list(B), ground(B) ).
```

Also, "meta-"modes can be defined which allow writing assertions in a very compact way. The previous assertion could be written as follows, by using a different mode definition, i.e.:

```
:- modedef out(X,P) : var(X) => ( P(X), ground(X) ).
:- pred qsort(list,out(list)).
```

where `P(X)` is a higher-order notation used in Ciao which stands for applying the property predicate `P`, whatever value it has (in this example `list`) to the argument `X`.

2.11 Discussion

In this chapter we have presented an assertion language which should be of interest in several different tools and for different purposes: compile-time checking, replacing the oracle in declarative debugging, run-time checking, providing information to the optimizer, general communication with the compiler, and automatic documentation. In such a situation it is difficult to restrict beforehand the properties of a program which we may be interested in expressing by means of assertions. Also, we cannot assume the existence of a particular inference system in every tool other than the underlying CLP system.

An assertion language can be seen as composed of: (1) a set of assertion schemas, (2) a syntax to build logic formulae for such schemas, and (3) a syntax to define predicates for the atomic logic formulae. In addition to the assertion language we need some inference system, capable of evaluating the assertions for a program. Often, assertion languages are designed bottom-up, in the sense that once the inference system to be used has been decided, the three components mentioned are defined so that the assertion language remains *decidable*, in the sense that any assertion expressible in the assertion language can be either proved to hold or not to hold in a program using the available inference system. This allows debugging systems based on this approach to reject programs which have not been validated w.r.t. the given assertions. In contrast, the design of our assertion language is top-down in the sense that it is not induced by any particular inference system. We do not assume a fixed set of property predicates but rather we provide the means for defining new property predicates. This can be done by providing exact descriptions of properties as CLP predicates and also by using assertions which indicate that a (possibly built-in) predicate property can be proved by a given inference system. We also use assertions to provide sufficient conditions for proving and disproving predicate properties when such exact definition is not available. As a result of this, on one hand our assertion language is very flexible, and on the other hand we have to lift the assumption that any assertion is decidable in the system. In other words, we have to live with the possible undecidability of any logic formula and thus of any assertion. Thus, in the tools which use the proposed assertion language we have to be able to deal safely with approximations [2.5]. We argue that lifting the decidability assumption opens the door to very interesting possibilities and that still very useful results can be obtained by combining static and dynamic checking of the assertions.

Even though the properties given in assertions may not be decidable, it is our view that assertions should be checked as much as possible at compile-time via static analysis. The system should be able to make conservative approximations in the cases in which precise information cannot be inferred (and some assertions may remain unproven). This is the approach taken in Chapter 3. Note, however, that if the properties allowed in assertions are not decidable the approach to the treatment of "don't know" during compile-time checking has to be weaker than the one used for "strong" debugging systems: in our case the program cannot be rejected. The case that the analysis is not capable either to prove nor disprove an assertion may be because we do not have an accurate enough inference system or simply because the assertion is just not statically decidable. In this we follow the spirit of [2.4, 2.8]. However, we do not rule out the definition and use of a decidable debugging system, e.g., based on types, if so desired.

Acknowledgements

The proposed assertion language is based on previous work by several DiSCiPl project members and numerous discussions within the project. However, any errors or omissions are only the fault of its authors. The authors would like to thank in particular Jan Małuszyński, Wlodek Drabent, and Pierre Deransart for many interesting discussions on assertions. Also Abder Aggoun, Helmut Simonis, Eric Vetillard and Claude Lai for their feedback on the kind of assertions needed in different state-of-the-art debugging tools.

This work has been partially supported by the European ESPRIT LTR project # 22532 "DiSCiPl" and Spanish CICYT projects TIC99-1151 EDIPIA and TIC97-1640-CE.

References

2.1 A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-semantics approach: Theory and applications. *Journal of Logic Programming*, 19&20, 1994.
2.2 J. Boye, W. Drabent, and J. Małuszyński. Declarative diagnosis of constraint programs: an assertion-based approach. In *Proc. of the 3rd. Int'l Workshop on Automated Debugging–AADEBUG'97*, pages 123–141, Linköping, Sweden, May 1997. U. of Linköping Press.
2.3 F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla. The Ciao Prolog System. Reference Manual. The Ciao System Documentation Series–TR CLIP3/97.1, School of Computer Science, Technical University of Madrid (UPM), August 1997.
2.4 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.
2.5 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.
2.6 F. Bueno, P. López-García, G. Puebla, and M. Hermenegildo. The Ciao Prolog Preprocessor. Technical Report CLIP8/95.0.7.20, Technical University of Madrid (UPM), Facultad de Informática, 28660 Boadilla del Monte, Madrid, Spain, November 1999.
2.7 D. Cordes and M. Brown. The Literate Programming Paradigm. *IEEE Computer Magazine*, June 1991.

2.8 P. Cousot. Types as Abstract Interpretations. In *Symposium on Principles of Programming Languages*, pages 316–331. ACM Press, January 1997.
2.9 P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
2.10 P.W. Dart and J. Zobel. A regular type language for logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 157–187. MIT Press, 1992.
2.11 W. Drabent, S. Nadjm-Tehrani, and J. Małuszyński. The Use of Assertions in Algorithmic Debugging. In *Proceedings of the Intl. Conf. on Fifth Generation Computer Systems*, pages 573–581, 1988.
2.12 W. Drabent, S. Nadjm-Tehrani, and J. Maluszynski. Algorithmic debugging with assertions. In H. Abramson and M.H.Rogers, editors, *Meta-programming in Logic Programming*, pages 501–522. MIT Press, 1989.
2.13 Lisa Friendly. The Design of Distributed Hyperlink Program Documentation. In *Int'l. WS on Hypermedia Design*, Workshops in Computing. Springer, June 1996. Available from `http://java.sun.com/docs/javadoc-paper.html`.
2.14 M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, 1996.
2.15 M. Hermenegildo. A Documentation Generator for Logic Programming Systems. In *ICLP'99 Workshop on Logic Programming Environments*, pages 80–97. N.M. State University, December 1999.
2.16 M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 International Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.
2.17 M. Hermenegildo, G. Puebla, and F. Bueno. Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging. In K. R. Apt, V. Marek, M. Truszczynski, and D. S. Warren, editors, *The Logic Programming Paradigm: a 25–Year Perspective*, pages 161–192. Springer-Verlag, July 1999.
2.18 P. Hill and J. Lloyd. *The Goedel Programming Language*. MIT Press, Cambridge MA, 1994.
2.19 J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
2.20 A. Kelly, A. Macdonald, K. Marriott, P. Stuckey, and R. Yap. Effectiveness of optimizing compilation for CLP(R). In *Proceedings of Joint International Conference and Symposium on Logic Programming*, pages 37–51. MIT Press, 1996.
2.21 D. Knuth. Literate programming. *Computer Journal*, 27:97–111, 1984.
2.22 K. Marriott and P. Stuckey. The 3 R's of Optimizing Constraint Logic Programs: Refinement, Removal, and Reordering. In *19th. Annual ACM Conf. on Principles of Programming Languages*. ACM, 1992.
2.23 K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.
2.24 L. Naish. A three-valued declarative debugging scheme. In *8th Workshop on Logic Programming Environments*, July 1997. ICLP Post-Conference Workshop.

2.25 G. Puebla, F. Bueno, and M. Hermenegildo. An Assertion Language for Debugging of Constraint Logic Programs. In *Proceedings of the ILPS'97 Workshop on Tools and Environments for (Constraint) Logic Programming*, October 1997.
2.26 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)*, LNCS. Springer-Verlag, 2000. To appear.
2.27 E. Shapiro. *Algorithmic Program Debugging*. ACM Distiguished Dissertation. MIT Press, 1982.
2.28 Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury: an efficient purely declarative logic programming language. *JLP*, 29(1–3), October 1996.
2.29 P. Van Roy and A.M. Despain. High-Performace Logic Programming with the Aquarius Prolog Compiler. *IEEE Computer Magazine*, pages 54–68, January 1992.
2.30 E. Vetillard. *Utilisation de Declarations en Programmation Logique avec Constraintes*. PhD thesis, U. of Aix-Marseilles II, 1994.
2.31 R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.
2.32 E. Yardeni and E. Shapiro. A Type System for Logic Programs. *Concurrent Prolog: Collected Papers*, pages 211–244, 1987.