Declarative Error Diagnosis of GAPLog Programs

Fredrik Eklund
Department of Computer and Information Science
Linköping University

Acknowledgements

First, I would like to express my deeply felt gratitude towards my advisor Włodek Drabent. Without his neverending enthusiasm and support this thesis would never have been written. I am very much indebted to him.

My advisory group has also consisted of Jan Małuszyński and Krzysztof Kuchciński. I am very grateful to them too for the time they have spent on me.

Furthermore, I would like to thank the rest of the members of the Logic Programming Group. I have greatly enjoyed the atmosphere, the heated arguments at our instructive seminars, and the warm friendship.

All over the department there are those that I perceive as my friends, and I am very happy that I got to know all of you.

This work was partially supported by ESPRIT project DiSCiPl (LTR project 22532).

Finally, I thank Anna for making life the most wonderful of adventures.

Contents

Chapter 1

Introduction

1.1 GAPLog

GAPLog [15] is a functional logic programming language, and it is an extension of Prolog. GAPLog programs may use functions written in other languages but still have a declarative semantics. This semantics associates each procedure of a GAPLog program with a relation over some domain. The set of such relations provides the model of the program. GAPLog is described in more detail in sec. 3.

Example GAPLog and Prolog programs for computing factorials are shown in figure 1.1. The GAPLog formulation is more straightforward than the Prolog version. This is because Prolog can only handle arithmetics with special predefined predicates, like *is*/2 in the example. More importantly, Prolog is dependent on that a function call is ground when reached by the execution. This is not the case for GAPLog.

```
fac(0, 1).                fac(0, 1).
fac(X, X*Y) :-            fac(X, W) :-
   fac(X-1, Y).               Y is X-1,
                              fac(Y, Z),
                              W is X*Z.
```

Figure 1.1: fac/2 in GAPLog and in Prolog

Function calls generate *constraints* in GAPLog. GAPLog constraints are equations. These are not evaluated until all their arguments are ground, but this could happen at any time during the execution, or not at all. This computation strategy of GAPLog makes it hard to debug programs by tracing the execution, since when a constraint is eventually executed it can be hard to associate it with the construct in the program and the context from which it originates.

1.2 Declarative error diagnosis

An advantage of logic programming languages is that they facilitate declarative programming. They make it possible, at least to a certain extent, to separate the declarative, logical semantics (WHAT is computed) from the operational semantics (HOW it is computed). In principle, all the reasoning concerning the correctness of programs (in other words, concerning program results) can be done on the level of the declarative semantics, thus abstracting from any details of computation. The latter are usually substantially more complicated that the declarative semantics.

This advantage is however lost when debugging has to be done on the level of the operational semantics. This happens in the case of usual debugging tools, which are various versions of tracers. They force the programmer to think in terms of the sequences of the computation steps. Thus it is important to develop debugging methods based on declarative semantics. Such approaches to localizing errors in programs are called declarative diagnosis.

We will assume that the user has a clear idea about the results that should be computed by the program. An error occurs when the program computes something that the programmer did not intend, or when it fails to compute something that he wanted.

The idea behind declarative error diagnosis is to collect information about what the program is intended to do, and compare this with what it actually does. By reasoning from this, a diagnoser can find errors. The information needed can be found by asking for example the user, a formal specification, or an older (correct) version of the program. The entities that provide the diagnoser with information are with a common term referred to as the *oracle*.

1.2.1 Incorrectness error diagnosis

Incorrectness error diagnosis concerns the case when a computation has resulted in an incorrect answer. The principal idea of the solution to this problem is to inspect the *proof tree* constructed for this answer. A proof tree consists of instances of program clauses. Each node is labelled by an atom. A node with all its children corresponds to an instance of a program clause. The root corresponds to the computed answer. To find the erroneous clause the diagnoser traverses the proof tree. At each node it asks the oracle about the validity of the corresponding atom. With the aid of the answers, the diagnoser is able to identify the erroneous clause. It has been shown [25] that the proof tree for an incorrect answer must include an instance of a program clause such that all body atoms are valid, and the head is not valid. This implies that the clause produces incorrect answers from correct ones. Thus, finding an incorrect clause amounts to traversing the proof tree to find a node representing an invalid atom whose all children represent valid atoms.

1.2.2 Insufficiency error diagnosis

Insufficiency error diagnosis concerns the case when a program fails to compute some expected result. The objective for insufficiency diagnosis is to scrutinize the attempt to construct a proof for an atom which incorrectly fails. The reason

for the error is located to one of the procedures (all clauses defining a predicate) in the computation.

1.3 Thesis Overview

The rest of this thesis is organized as follows. Chapter 2 provides a brief overview to related work on debugging in the fields of logic programming, functional logic programming, and constraint logic programming. Chapter 3 contains technical background necessary for the rest of the thesis. It is assumed that the reader is familiar with the basics of logic programming.

In chapters 4 and 5 algorithms are constructed for error diagnosis of incorrect and missing answers. Chapter 6 provides some means to simplify user interaction. Finally, a summary is given in chapter 7.

The appendices contain transcripts from debugging sessions with a system that implements the error diagnosis algorithms given in this thesis.

Chapter 2

Related work

2.1 Debugging of logic programming languages

The very first implementations of Prolog had, naturally, crude tracing facilities as an aid for debugging. Those displayed the actual execution of the programs. Matching the trace information with the program to locate errors was difficult mainly due to backtracking. The problem with backtracking is that it is a non-local transfer of control, which is hard to follow. To overcome the worst deficiencies of the original tracers an execution model was presented by Lawrence Byrd [2]. His box model of execution has since then been the basis of most Prolog debuggers. The box model is based on the operational semantics of Prolog, but the trace information displayed does not exactly conform to the actual execution. This is because the trace displayed is augmented with information which makes it easier for a human to follow the actual execution.

Informally, a box represents all clauses defining a predicate, called a *procedure*. A box has four ports through which control can enter and leave it. First, the *call* port which is entered when the predicate is first called. Second, the *exit* port which is used when the call to the predicate has succeeded. Third, control enters the box through the *redo* port upon backtracking. Fourth, the box is left through the *fail* port when the last alternative for the predicate has been tried and failed.

The problem with this approach is that it implies a discrepancy between design and debugging methodologies. It is considered to be an advantage with the logic programming paradigm that design can be done declaratively. Unfortunately, the box model focuses on the behaviour of the program. Hence, we would like to design and program with the declarative semantics in mind, but we are forced to debug in the environment of the operational semantics.

Tracers based on the box model of Byrd only display information about the current atom to be resolved. This is a small amount of information, and to perform any real debugging the user have to follow the execution trace in the code.

Since Byrd presented his box model more refined versions have been proposed, e.g. [27, 6, 22, 23], together with debuggers. Common to these is that they use more ports in a box to be able to display the execution at a more fine-grained level, and with better context information. Tracer information using an

extended model typically is more informative, because the context information is better. The precision in finding errors is sometimes better because of the more fine-grained view of program execution. However, they are all operational by nature.

Probably the most ambitious debugger developed for logic programming is the so called Transparent Prolog Machine (TPM) [7]. The TPM displays a graphical view of the execution in terms of augmented and/or-trees. They can be used to display information at two levels of precision; a low precision level which display the call structure, and a high precision level which goes into minute detail, down to attempted unifications.

2.2 Declarative error diagnosis

In 1982 Ehud Y. Shapiro presented his PhD thesis [25] on *Algorithmic Debugging*. This was the first work in the area of declarative error diagnosis. The fundamental difference between tracing and declarative error diagnosis is that the latter is based on the declarative semantics of the language, whereas the former is based on the operational semantics. This has far-reaching impact on the way error locating is performed.

The underlying idea is that the diagnoser finds errors in the program guided by knowledge about intended properties of the program. The knowledge is supplied by an *oracle*. The oracle could be a specification of some kind, but is usually synonymous with the user of the diagnoser.

An important contribution by Pereira [21] is a means to simplify the user interaction. In Shapiro's approach, the user was supposed to be able to provide correct instances of an atom when diagnosing insufficiency. In the worst case he would have to provide all of them. In Pereira's approach the user is only required to determine whether a set contains all correct instances of an atom or not.

Lloyd [13, 14] and Ferrand [8] has provided soundness and completeness results for declarative error diagnosers. Lloyd's treatment is in the context of a logic programming language with coroutines. The diagnoser, however, is in the Shapiro style for diagnosing missing answers.

As shown clearly in Naish's comprehensive survey on insufficiency diagnosis [18], coroutining does not mix very well with Pereira's approach. In the survey Naish elaborates on different implementation ideas and philosophies, and applicability of these to different computation rules.

An important issue concerns mechanizing of the oracle. Shapiro's debugger featured memorization of questions asked, and their answers, to ensure that the user was never asked the same question twice. This is an obvious mechanization, but he also proposed some others. It is important that the user is relieved as much as possible of answering questions. Further work (e.g. [5]) investigates mechanizing of the oracle more thoroughly. The user is supplied with means to instruct the diagnoser of intended properties of the program so that it can answer some questions by itself.

Recently the framework of s-semantics has been applied to declarative error diagnosis [4, 3].

The concept of declarative error diagnosis was first proposed as a general approach to debugging [25]. The presentation, however, was carried out in the

paradigm of logic programming. Most of the interest for the approach has also come from the corresponding scientific community, but it has been applied to other programming paradigms, e.g. imperative programming [24] and functional programming [20].

There are very few implementations of declarative error diagnosers which are well spread. The debugging environment NUDE [16] for NU-Prolog has as one of its parts a declarative error diagnoser. It is in the Shapiro tradition, where the user is supposed to provide the diagnoser with correct instances of atoms when diagnosing missing answers.

2.3 Functional logic programming languages

In functional logic programming we have a problem similar to that which was the reason for Byrd's box model, i.e. transfer of control is non-local and non-linear. Most functional logic programming languages are more or less experimental. Not all of them are equipped with a debugging tool, but most of those who are have debuggers with tracers based on refined box models, e.g. ALF [9]. There is also an operational debugger for GAPLog [10] based on a refined box model. It provides the user with a context of the execution, to aid debugging, in a similar manner as refined box models for Prolog do.

NUE-Prolog, by Naish [17], is the only functional logic programming language for which a declarative error diagnoser has been developed [19]. The insufficiency diagnosis is in the Shapiro-style. NUE-Prolog defines the functions in the language, as opposed to GAPLog, which use external definitions.

2.4 Constraint logic programming languages

A subject closely related to debugging of functional logic programs is debugging of constraint logic programs [11]. So far, this is a quite unresearched field, but recently research have started in this field too [12, 26].

Chapter 3

Preliminaries

3.1 GAPLog

GAPLog [15] is a functional logic programming language with the following properties:

- The functional component of the language is not fixed. Functions in a program are imported from an external program, written in a functional or procedural language. These functions are viewed as black boxes. From the logic program's point of view they are considered to be side-effect free.
- The declarative semantics is achieved from viewing the functional expressions as (possibly infinite) sets of equations.
- The operational semantics is based on an incomplete equational unification, called S-unification.

3.1.1 Syntax of GAPLog

We assume three alphabets to be given:

- Σ, the set of *functor symbols*,
- Π, the set of *predicate symbols*,
- $\mathcal{V}$, the set of *variables*.

Terms, atoms, clauses, and *goals* are constructed in the usual way. There is one exception though. The head of a clause must not be of the form $t_1 \doteq t_2$. That is, it must not be an *equational atom*. The reason for this restriction is that equality should be defined exclusively by external procedures. This will be further discussed in section 3.1.2.

The connection between the syntax and the external procedures is made by dividing the set Σ into two disjoint sets: one of *defined symbols*, denoted Σ_D, and one of *constructors*, denoted Σ_C. Terms built solely from constructors and variables are called *constructor terms*. The role of the defined symbols is to denote the external functional procedures. The role of the constructor terms is

to denote elements of the domain of such procedures. Consequently, we define a *function call* to be a term of the form:

$$f(s_1, \ldots, s_n)$$

where f is a defined symbol, each s_i is a constructor term, and the arity of f (i.e. n) agrees with the number of arguments expected by the external procedure associated with f. Moreover, when f is a defined symbol, any term of the form $f(t_1, \ldots, t_n)$ will be called a *functional term.*

In a similar way we call an atom for a *functional atom* if it contains a functional term, and consequently for a *constructor atom* if it contains only constructor terms.

3.1.2 Declarative semantics of GAPLog

The description of the declarative semantics follows closely the presentation by Małuszyński *et al.* [15].

Assumption: Every external procedure returns a ground (i.e. variable-free) constructor term when given ground constructor terms as input.

This means that the external procedures are assumed to compute total functions on ground constructor terms. It follows that each external procedure induces an infinite set of ground equations of the form:

$$f(s_1, \ldots, s_n) \doteq s$$

where the procedure associated with f gives s as output when given $s_1, \ldots, s_n$ as input. The set of all such equations (over all external procedures) is denoted $\mathcal{E}$.

This construction allows us to view any amalgamated program P as an equational logic program, namely:

$$P \cup \mathcal{E}$$

The problem of assigning a declarative semantics to amalgamated programs has thus been reduced to assigning such a semantics to equational logic programs. For our purposes only a few declarative notions are needed. One is the equivalence relation $=_{\mathcal{E}}$ induced by $\mathcal{E}$. It is defined on the set of terms by:

$$t =_{\mathcal{E}} u \iff \mathcal{E} \models t \doteq u$$

That is $t =_{\mathcal{E}} u$ iff every interpretation (of our alphabet) which interprets equality as identity and satisfies all equations in $\mathcal{E}$, also satisfies $t =_{\mathcal{E}} u$. It is easily seen that $\mathcal{E}$ forms a so called *canonical term-rewriting system.* Thus the external procedures may be used to decide whether $t =_{\mathcal{E}} u$ or not. This is done by successively replacing those subterms of t and u which occur as left-hand sides of equations in $\mathcal{E}$ by the corresponding right-hand sides. The two terms obtained when no more replacements are possible are then compared. If they are identical, $t =_{\mathcal{E}} u$ is established. Otherwise the contrary holds.

The relation $=_{\mathcal{E}}$ is extended to atoms in the following way:

$p(\overline{t}) =_{\mathcal{E}} q(\overline{s})$ iff $p \equiv q$ and $\overline{t} =_{\mathcal{E}} \overline{s}$

The relation $=_{\mathcal{E}}$ extends to substitutions by:

$$\sigma =_{\mathcal{E}} \theta \Longleftrightarrow \textit{for each variable } x, x\sigma =_{\mathcal{E}} x\theta$$

The declarative notion of a *correct answer* for a program P and a goal G is also of importance. By this is meant a substitution σ such that

$$P \cup \mathcal{E} \models G\sigma$$

Our main concern will be to compute all correct answers. The operational semantics presented in the next section will reduce this problem to finding all $\mathcal{E}$-unifiers for sets U of equations. These substitutions σ which satisfy:

$$\forall t \doteq u \in U : t\sigma =_{\mathcal{E}} u\sigma$$

The set of all $\mathcal{E}$-unifiers for U may be represented by a *complete* set of $\mathcal{E}$-unifiers for U, that is, a set S of substitutions such that θ is an $\mathcal{E}$-unifier for U iff $\theta =_{\mathcal{E}} \sigma\gamma$ for some $\sigma \in S$ and some γ.

Definition 1 *For each atom A there is an atom A', where each ground functional term in A have been replaced by a ground constructor term, such that $A =_{\mathcal{E}} A'$. The atom A' is called the* evaluated instance *of A and is denoted $e(A)$.*

We will use a certain kind of interpretations for programs. By an abuse of terminology we will call them Herbrand interpretations (actually, when restricted to constructed terms they are Herbrand interpretations).

Definition 2 (Herbrand interpretation) *An* Herbrand interpretation *for a language which contains constructors Σ_C, defined symbols Σ_D, and predicate symbols Π is an interpretation I such that:*

- *The domain of I is the Herbrand universe U_{Σ_C}*
- *for every constant $c \in \Sigma_C$, its interpretation c_I is defined to be c itself;*
- *for every n-ary functor symbol $f \in \Sigma_C$, its interpretation f_I is defined as the mapping*
$$f_I(t_1, \ldots, t_n) := f(t_1, \ldots, t_n);$$
- *the predicate symbol $\doteq$ is interpreted as equality.*

A Herbrand interpretation I is uniquely determined by a set of atoms J containing atoms of the form $p(t_1, \ldots, t_n)$ and $f(t_1, \ldots, t_n) \doteq s$ (where p is not $\doteq$, f is a defined functor, $s, t_1, \ldots, t_n$ are ground constructor terms, and $n \geq 0$) such that $J = \{p(t_1, \ldots, t_n) \mid (t_1, \ldots, t_n) \in p_I\} \cup \{f(t_1, \ldots, t_n) \doteq s \mid f_I(t_1, \ldots, t_n) = s\}$. As is usually done [14], a Herbrand interpretation will be identified with the corresponding set of atoms.

Let I be an Herbrand interpretation. Let $\mathcal{E}_I$ be the set of those atoms of I whose predicate symbol is $\doteq$. The relation $=_I$ is defined to be $=_{\mathcal{E}_I}$. Obviously, $t =_I u$ iff $I \models t \doteq u$. We will use an extension of $=_I$ to atoms analogical to that of $=_{\mathcal{E}}$.

Definition 3 (Herbrand model) *Let P be a GAPLog program and $\mathcal{E}$ be the set of equations describing the external functions used in P. By $M_{P,\mathcal{E}}$ we mean the Herbrand interpretation such that:*

- *A ground constructor atom A is in $M_{P,\mathcal{E}}$ iff $P \cup \mathcal{E} \models A$, and*
- *a ground equation $f(t_1, \ldots, t_n) \doteq s$ is in $M_{P,\mathcal{E}}$ iff $\mathcal{E} \models f(t_1, \ldots, t_n) \doteq s$.*

Proposition 1 *$M_{P,\mathcal{E}}$ is the least Herbrand model of P and E.*

Proof Obviously, $M_{P,\mathcal{E}}$ is a model of $\mathcal{E}$. To show that $M_{P,\mathcal{E}}$ is a model of P it is sufficient to show that each ground instance $H \leftarrow B$ of a clause of P is true in $M_{P,\mathcal{E}}$. Assume $M_{P,\mathcal{E}} \models B$. Then $P \cup \mathcal{E} \models B$. Obviously, $P \cup \mathcal{E} \models H \leftarrow B$. So we have $P \cup \mathcal{E} \models H$ and $M_{P,\mathcal{E}} \models H$.

If I is a Herbrand model of $P \cup \mathcal{E}$ then, by the definition above, each atom of $M_{P,\mathcal{E}}$ is in I. Thus $M_{P,\mathcal{E}}$ is the least Herbrand model w.r.t. $\subseteq$. □

In this work we are not interested in a fixpoint characterization of least Herbrand models.

3.1.3 Operational semantics of GAPLog

The presentation of the operational semantics of GAPLog follows closely the presentation of Małuszyński *et al* [15].

The unification algorithm of GAPLog is called *S-unification* [15]. It successively transforms a set U of equations to *normal form*, or fails in some cases when it does not have an $\mathcal{E}$-unifier.

Normal form is defined as follows [1]:

Definition 4 (Normal form) *A finite set N of equations is in* normal form *iff*

- $N = \{X_1 \doteq s_1, \ldots, X_n \doteq s_n, c_1 \doteq d_1, \ldots, c_m \doteq d_m\}, (n, m \geq 0)$
- *each X_i is a variable which occurs only once in N, and*
- *each s_i and c_i is a constructor term, and*
- *each d_i is a non-ground functional term, that only occurs once in N.*

Equations of the form $c \doteq d$, where c is a constructor term, and d is a non-ground functional term, are called *primitive constraints.*

If the resulting set of S-unification is also in *solved form* it can be transformed to a most general $\mathcal{E}$-unifier σ_U for the set U. A set of equations is in *solved form* iff:

- the left hand sides of all equations in the set are distinct variables, and
- the equations can be ordered as $X_1 \doteq t_1, \ldots, X_n \doteq t_n$, so that if a variable X_i occurs in t_j then $i < j$.

For any set of equations $\{X_1 \doteq t_1, \ldots, X_n \doteq t_n\}$ in solved form, the corresponding $\mathcal{E}$-unifier is the resulting substitution of $\{X_1/t_1\}\{X_2/t_2\}\cdots\{X_n/t_n\}$.

Algorithm 1 (S-unification) *Initially, let $U_0 = U$ and $i = 0$. Repeatedly do the following: select any equation $u \doteq t$ in U_i such that one of Rules 1 to 9 applies. If no such equation exists, stop with U_i as result. Otherwise perform the corresponding action, that is: stop with failure or construct U_{i+1} from U_i and increment i.*

1. *u and t are identical. Remove $u \doteq t$.*
2. *u is a variable which occurs elsewhere in U_i and t is a constructor term distinct from u. If u occurs in t then stop with failure, otherwise replace all other occurrences of u by t (leaving $u \doteq t$ unchanged).*
3. *u is a non-variable and t is a variable. Replace $u \doteq t$ by $t \doteq u$.*
4. *u is of the form $c_1(u_1, \ldots, u_m)$ and t is of the form $c_2(t_1, \ldots, t_n)$, where c_1 and c_2 are constructors. If c_1/m and c_2/n are distinct then stop with failure, otherwise replace $u \doteq t$ by $u_1 \doteq t_1, \ldots, u_m \doteq t_n$.*
5. *u and t are both function calls. Let z be a variable not in $var(U_i) \cup var(U)$. Replace $u \doteq t$ by the two equations $z \doteq u$ and $z \doteq t$.*
6. *u or t has a function call d as a proper subterm. Let z be a variable not in $var(U_i) \cup var(U)$. Replace all occurrences of d in U_i by z, then add the equation $z \doteq d$.*
7. *u is a function call and t is a constructor term. Replace $u \doteq t$ by $t \doteq u$.*
8. *u is a constructor term and t is a function call which occurs elsewhere in U_i. Replace all other occurrences of t by u and leave $u \doteq t$ unchanged.*
9. *t is a ground function call. Replace $u \doteq t$ by $u \doteq s$, where $t \doteq s$ is in $\mathcal{E}$.*

Definition 5 *A* constraint *is a (possibly existentially quantified) conjunction of primitive constraints.*

In the following no distinction will be made between a set of primitive constraints and the conjunction of its elements.

Definition 6 (Constrained atom) *For an atom A and a constraint c, the pair $c[]A$ is called a* constrained atom.

Definition 7 (Constrained query) *For a query (conjunction of atoms) Q and a constraint c, the pair $c[]Q$ is called a constrained query.*

The following definition is slightly changed from the definition in Maluszynski et al [15], because the insufficiency error diagnosis algorithm needs the S-SLD-resolution to handle constrained initial queries.

Definition 8 (S-SLD-resolution) *Let P be a program, $\mathcal{E}$ an equational theory (describing the external procedures), Q_0 a query, C a set of primitive constraints, and $\mathcal{R}$ a computation rule.* A S-SLD-derivation *is a (finite or infinite) sequence of triples $\langle Q_0, \theta_0, C_0; Q_1, \theta_1, C_1; \ldots \rangle$ such that:*

- $\theta_0 = \epsilon$, $C_0 = C$

- *If Q_i, θ_i, C_i is not the last triple in the sequence, $Q_i = A_1, \ldots, A_n$, and $\mathcal{R}(Q_i) = A_j$ then there exists a (standardized apart) clause $B_0 \leftarrow B_1, \ldots, B_n (n \geq 0)$ of P such that:*

  1. *S-unification of $\{B_0 \doteq A_j\} \cup C_i$ does not fail and produces the set U of equations. Let C' be the the set of primitive constraints in U.*
  2. *$C_{i+1} = C'$; θ_{i+1} is the substitution corresponding to $U - C'$.*
  3. *$Q_{i+1} = (A_1, \ldots, A_{j-1}, B_1, \ldots, B_n, A_{j+1}, \ldots, A_m)\theta_{i+1}$.*

If the derivation is of the form:

$$\langle Q_0, \theta_0, C_0; \ldots; \Box, \theta_n, \emptyset \rangle$$

where $\Box$ denotes the empty sequence of atoms, then $\theta_1 \cdots \theta_n$ is said to be the *computed substitution* for the derivation, and $\theta_1 \cdots \theta_n \mid_{Vars(Q_0)}$ (the restriction of $\theta_0 \cdots \theta_n$ to the variables of Q_0) is said to be an *$\mathcal{R}$-computed answer substitution* for P and Q_0 with C.

If the derivation is of the form:

$$\langle Q_0, \theta_0, C_0; \ldots; \Box, \theta_n, C_n \rangle$$

where $\Box$ denotes the empty sequence of atoms, the pair $(\theta_0 \cdots \theta_n \mid_{Vars(Q_0)}, \exists_{-Vars(Q_0)} C_n)$ (where $\exists_{-Vars(Q_0)} C_n$ means that all variables of C_n not occurring in Q_0 are quantified) is said to be an *$\mathcal{R}$-computed constrained answer* for P and Q_0 with C.

S-SLD-resolution is typically implemented by using a program where all clauses have been *flattened*. Flattening of a clause $H \leftarrow B_1, \ldots, B_n$ where $n \geq 0$ can be achieved as follows [15]:

1. For each $B_i, (i = 1..n)$, apply rule 6 of S-unification to $\{v \doteq B_i\}$, where v is a new variable, as many times as possible. This yields a set $\{v \doteq B_i', z_1 \doteq d_1, \ldots, z_n \doteq d_n\}$. Let $C_i = \{z_1 \doteq d_1, \ldots, z_n \doteq d_n\}$.
2. Then apply rule 6 as many times as possible to $\{v \doteq H\}$, where v is a new variable. This yields a set $\{v \doteq H', z_1 \doteq d_1, \ldots, z_n \doteq d_n\}$. Let $C_0 = \{z_1 \doteq d_1, \ldots, z_n \doteq d_n\}$.
3. The flattened clause is $H' \leftarrow C_0, B_1', C_1, \ldots, B_n', C_n$.

Hence, a flattened clause is a clause where all function calls have been removed from inside the atoms in the clause and made explicit in the body.

3.2 Declarative error diagnosis

The idea behind declarative diagnosis is that the programmer in some sense has in mind an intended model I for his program. This means that I is intended to be the least Herbrand model of the (correct) program. Error diagnosis is done by comparing I with declarative properties of the program.

Definition 9 (Incorrect program) *A program P is* incorrect *if $M_{P,\mathcal{E}} - I \neq \emptyset$.*

That is, a program is incorrect if it computes results which are not in the intended model.

Definition 10 (Insufficient program) *A program P is* insufficient *if $I - M_{P,\mathcal{E}} \neq \emptyset$.*

That is, the program does not compute everything in the intended model.

3.2.1 Declarative errors

To be able to express ourselves precisely concerning error diagnosis we must define what an error is. The first kind of error that we will define correspond to *incorrect answers*. That is, a query produced an answer that was not expected. We describe this kind of error symptom with the following definition:

Definition 11 (Incorrectness symptom) *For a program P with intended model I and a query Q with computed answer θ, $Q\theta$ is an* incorrectness symptom *iff $I \not\models Q\theta$.*

In other words, for an atomic query A, $A\theta$ is an incorrectness symptom if it has a ground instance $A\theta\sigma$ whose evaluated instance A' is not in I. Note that $A\theta\sigma =_I A'$, i.e. that functions are evaluated according to the intended model.

Definition 12 (Incorrectness symptom) *For a program P with intended model I and a constrained query $c[]Q$ with computed answer substitution θ and remaining constraint c', $Q\theta$ is an* incorrectness symptom *iff $I \not\models c' \rightarrow c\theta, Q\theta$.*

The result of incorrectness diagnosis is an *incorrect clause*, which is a clause that contains an error.

Definition 13 (Incorrect clause) *A clause $H \leftarrow B$ is called an* incorrect clause *with respect to the intended model I if there exists a substitution θ such that $I \models B\theta$, but $I \not\models H\theta$.*

Theorem 1 *If there is an incorrectness symptom, then there is an incorrect clause or an error in an external procedure.*

The second kind of error which we define corresponds to *missing answers*. That is, a query failed to produce an answer that was expected.

Definition 14 (Insufficiency symptom) *An atom A called is an* insufficiency symptom *for the program P and an intended model I iff there exists a substitution θ such that:*

- *$A\theta$ is ground,*
- *$I \models A\theta$,*
- *the query A results in a finite set of answer substitutions $\{\theta_1, \ldots, \theta_m\}$, $(m \geq 0)$, for P, and;*
- *for any substitution σ, $A\theta \neq_I A\theta_i\sigma$ for $i = 1, \ldots, m$.*

Definition 15 (Insufficiency symptom) *A constrained atom $c[]A$ is an* insufficiency symptom *for the program P and an intended model I iff there exists a substitution θ such that*

- $A\theta$ *is ground,*
- $I \models A$,
- $I \models c\theta$,
- *the constrained query* $c[]A$ *results in a finite set of constrained answers* $\{(\theta_1, c_1), \ldots, (\theta_m, c_m)\}$, $(m \geq 0)$, *for* P, *and*
- $A\theta \neq_I A\theta_i\sigma$ *for* $i = 1, \ldots, m$ *and any substitution* σ *such that* $I \models c_i\sigma$.

The result of insufficiency diagnosis is a *not completely covered atom*, which is a representative of a procedure which fails to produce all expected answers.

Definition 16 (Not completely covered atom) *For a program* P *with intended model* I, *an atom* A *is called* not completely covered *if it has a ground instance* $A\theta$ *such that* $I \models A\theta$, *but no clause of* P *has an instance* $H \leftarrow B$ *such that* $H =_I A\theta$ *and* $I \models \exists(B)$.

The definition for *not completely covered constrained atom* uses def. 16.

Definition 17 (Not completely covered constrained atom) *For a GAPLog program* P *with intended model* I, *a constrained atom* $c[]A$ *is called* not completely covered *if there exists a substitution* θ *such that* $I \models c\theta$ *and* $A\theta$ *is not completely covered with respect to* P *and* I.

So $c[]A$ is not completely covered if there exists a substitution σ such that $A\sigma$ is ground, $I \models c\sigma \wedge A\sigma$, and no clause of P has an instance $H \leftarrow B$ such that $H =_I A\sigma$ and $I \models \exists(B)$.

An *incomplete procedure* is the set of clauses that fails to define a not completely covered atom, and thus is the reason for an insufficiency symptom.

Definition 18 (Incomplete procedure) *The procedure for an atom* A *is called an* incomplete procedure *if* A *is not completely covered.*

Theorem 2 *If there is an insufficiency symptom, then there is an incomplete procedure or an error in an external procedure.*

3.2.2 Oracle questions

Shapiro [25] introduced the idea of having an *oracle* in a debugging system. The oracle's task is to answer questions about intended properties of a program. With the aid of the answers the system would be able to find errors. This is not as farfetched an idea as it might sound. Usually, the oracle is represented by the person who is performing the debugging, but it could also be a formal specification, an older version of the same program, or something similar.

In Shapiro's original work a so called *ground oracle* is used, an oracle that can only answer questions about ground atoms. Soon it was noticed that this is an unnecessary restriction, and non-ground oracles started to be used.

Crucial to the declarative error diagnosis technique are the questions posed to the oracle. It is important that the user understands them thoroughly to be able to answer them correctly. We deal with three kinds of questions, which come in two versions, constrained and unconstrained. The versions without

constraints are essentially variations of classical ones which have been described in numerous papers. The versions with constraints are new, and motivated by the differences between logic programming and GAPLog.

We introduce an example that will be used to explain the questions posed to the oracle. Fig. 3.1 shows a graph with four nodes and some weighted edges between them. Further assume a predicate *distance*/3 for the relation over nodes that are connected, and the total weight on the edges on a path between them. For example, $distance(a, b, 1)$ is in the intended model for *distance*/3, but so is $distance(a, b, 4)$ because there is a path with a loop from a to b, with total weight 4.

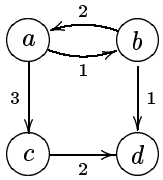

Figure 3.1: A graph

This thesis conforms to the convention that the (initial) goal to a logic program is called a query, and that debugging algorithms ask *questions* (about program semantics).

First, we define questions that are used during incorrectness diagnosis.

Definition 19 (Universal questions) *The question "Is the atom A valid in the intended model I?" is called an* universal question for unconstrained atoms. *The answer to it is* yes *iff $I \models A$.*

The question "Is the constrained atom $c \square A$ valid in the intended model I?" is called an universal question for constrained atoms. *The answer to it is* yes *iff $I \models c \rightarrow A$.*

Example 1 *In our example, the answer to the question*

Is $distance(a, b, 4)$ valid?

is yes*, because there is a path from a to b, with total weight 4.*

Example 2 *For the question*

Is $distance(a, N, D)$ valid?

the answer is no*, because even though there is a path from node a to all the other nodes, the distances can only take on specific values. Moreover, for the atom to be valid in I, all ground instances of it must be true in I, and e.g. there are no nodes labelled with numbers.*

Example 3 *The answer to the question*

Is $D > 0, 1 \doteq D \bmod 2 \square distance(a, b, D)$ valid?

is no*. An example of an instance that is not true in I is when D is bound to 5. Then the constraint is true, but there is no path from a to b with weight 5.*

Example 4 *The question*

$$\textit{Is } D > 0, 1 \doteq D \bmod 3 \square \textit{distance}(a, b, D) \textit{ valid?}$$

renders the answer yes, *because all paths from a to b are of length $3k+1$, where k is an integer greater than or equal to 0.*

The following questions are used during insufficiency diagnosis. Actually, *existential questions* are special cases of *completeness questions.*

Definition 20 (Existential questions) *The question "Is the atomic formula A satisfiable in the intended model I?" is called an* existential question for unconstrained atoms. *The answer to it is* yes *iff $I \models \exists A$. I.e. there exists a θ such that $I \models A\theta$.*

The question " Is the constrained atom $c \square A$ satisfiable in the intended model I?" is called an existential question for constrained atoms. *The answer to it is* yes *iff $I \models \exists(A \wedge c)$. I.e. there exists a θ such that $I \models A\theta \wedge c\theta$.*

Example 5 *We see that for the question*

$$\textit{Is } \textit{distance}(a, c, D) \textit{ satisfiable?}$$

the answer is yes, *since any substitution that binds D to the weight of one of the paths from a to c will do.*

Example 6 *The answer is* no *to the question*

$$\textit{Is } \textit{distance}(c, a, D) \textit{ satisfiable?}$$

since there is no path from c to a.

Example 7 *There are infinitely many paths from a to c, but none of them has a weight less than 3, so the answer to the question*

$$\textit{Is } D < 3 \square \textit{distance}(a, c, D) \textit{ satisfiable?}$$

is no.

Example 8 *There exist paths from a to c that have total weight more than 3, so the answer to the question*

$$\textit{Is } D > 3 \square \textit{distance}(a, c, D) \textit{ satisfiable?}$$

is yes.

Definition 21 (Completeness questions) *The question "Do the atoms in the set S cover all ground instances of the atomic formula A in the intended model I?" is called a* completeness question for unconstrained atoms. *The answer to it is yes iff for any θ such that $I \models A\theta$ and $A\theta$ is ground, there exists a $B \in S$ and a substitution σ such that $A\theta =_I B\sigma$.*

The question "Do the constrained atoms in the set S cover all ground instances of the constrained atom $c \square A$ in the intended model I?" is called a completeness question for constrained atoms. *The answer to it is yes iff for any substitution θ such that $I \models c\theta \wedge A\theta$ and $A\theta$ is ground there exists a $d \square B$ in S and a substitution σ such that $I \models d\sigma$ and $A\theta =_I B\sigma$.*

Example 9 *For the question*

> *Does* $\{distance(c, d, 2)\}$ *cover* $distance(c, d, W)$*?*

the answer is yes*. It can easily be seen that there is only one path from* c *to* d*, and that its weight is 2.*

Example 10 *That the answer to the question*

> *Does* $\{distance(d, c, 2)\}$ *cover* $distance(d, c, W)$*?*

is yes *can be perceived as surprising, but since there are no paths from* d *to* c*, any set covers* $distance(d, c, W)$*.*

Example 11 *Since there are infinitely many paths from* a *to* d*, because of the loop between* a *and* b*, the answer to the question*

> *Does* $\{distance(a, d, 5)\}$ *cover* $distance(a, d, D)$*?*

is no*.*

Example 12 *There are infinitely many solutions to the general query* $distance(N, M, W)$*, but there are only two paths shorter than 2, so the answer to the question*

> *Does* $\{distance(a, b, 1), distance(b, d, 1)\}$ *cover all ground instances of* $W < 2 \square distance(N, M, W)$*?*

is yes*.*

Example 13 *Likewise, there are infinitely many true instances of* $distance(a, b, D)$ *in* I*, where the values for* D *are in* $\{3 * k + 1 | k$ *is a natural number*$\}$*. The answer for the question*

> *Does* $\{D \doteq 3 * K + 1 \square distance(a, b, D)\}$ *cover all instances of* $distance(a, b, D)$*?*

is yes*.*

Chapter 4

Incorrectness diagnosis

The actual difference between Prolog and GAPLog when it comes to diagnosing incorrectness lies in the questions posed to the oracle. The questions should be formulated in such a way that the oracle can take the constraints into account, but this does not really change the algorithm for locating errors. It is also necessary to be aware of that errors in external procedures can mislead the diagnoser unless care is taken to avoid this (see section 4.1).

The execution of a succeeding goal in GAPLog can, just like for Prolog, be viewed as the construction of a proof tree for it. The difference is that we have to take into account external procedures and the possibility of obtaining constrained answers.

4.1 Incorrectness due to external procedures

To diagnose incorrectness of an ordinary logic program a diagnoser constructs a proof tree for a query which succeeds with an incorrect answer, and then asks the oracle if the atoms labelling the nodes are valid in the intended model (see sec. 3.2). Section 4.2 provides definitions and algorithms for incorrectness diagnosis of GAPLog programs, while this section contains a discussion motivating their formulation.

Figure 4.1 displays a program $ef/1$, which is intended to check if both a number and its Fibonacci number are even. The intended semantics of the function fib corresponds to the set of atoms

$$\left\{ fib(N) \doteq M \;\middle|\; \begin{array}{ll} M = 0 & \text{if } N = 0 \\ M = 1 & \text{if } N = 1 \\ M = fib(N-2) + fib(N-1) & \text{otherwise} \end{array} \right\}$$

The relation denoted by the predicate $ef/1$ in the intended model turns out to be quite simple. It is the set of atoms $\{ef(N) \mid N = 6k, k \geq 0\}$.

If everything is correct in the program and the external procedures, then $\leftarrow ef(4)$ fails, since the Fibonacci number of 4 is 3. Assume that the Fibonacci procedure has been incorrectly implemented, so that it computes the Fibonacci number of n to be the Fibonacci number of $n + 2$ instead. In this case the Fibonacci number of 4 is computed to be 8, and $\leftarrow ef(4)$ succeeds.

```
ef(N) :-
   even(N),
   even(fib(N)).

even(0).
even(N) :-
   N > 0,
   even(N-2).
```

Figure 4.1: ef/1

Let the nodes of the proof tree be labelled with instances of atoms in the clauses' bodies. Constructing the proof tree for $\leftarrow ef(4)$ with this approach yields the proof tree in fig. 4.2.

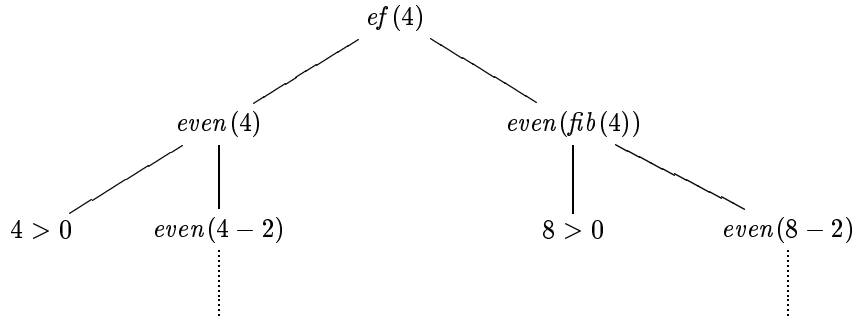

Figure 4.2: Proof tree of $\leftarrow ef(4)$, labelled with instances of body atoms

If the diagnosis is performed with the proof tree of fig. 4.2 as basis, the following dialogue could ensue between the diagnoser and the oracle.

> Is $ef(4)$ valid? No.
>
> Is $even(4)$ valid? Yes.
>
> Is $even(fib(4))$ valid? No.
>
> Is $even(8-2)$ valid? Yes.

Consequently, the diagnoser concludes that the rule for $even/1$ is wrong, even though it is perfectly correct. This is because at the crucial node labelled $even(fib(4))$ we could not know that an incorrect value of fib was computed. It is reasonable to assume that the actual computed values of the external procedures must be taken into account. Besides, any function calls originating from clause heads will not be present in a proof tree constructed in such a way.

This understanding leads to trying another approach to constructing proof trees. Labelling the proof tree for the query $\leftarrow ef(4)$ with evaluated instances

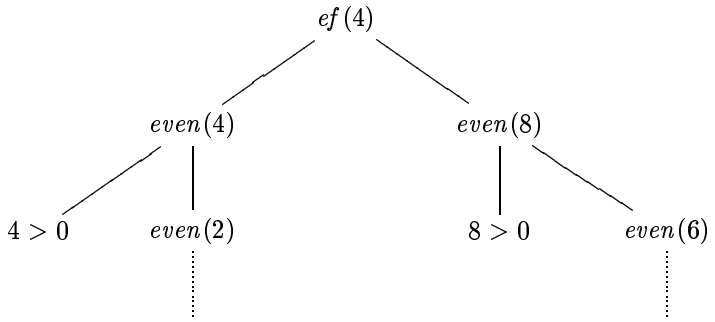

Figure 4.3: Proof tree of $\leftarrow ef(4)$, labelled with evaluated instances of atoms

of atoms yields the proof tree in fig. 4.3. Since evaluated instances are used, the values of the function calls will be visible in the proof tree.

Using it for diagnosing yields the following questions and answers from the diagnoser and the oracle respectively.

Is $ef(4)$ valid? No.

Is $even(4)$ valid? Yes.

Is $even(8)$ valid? Yes.

From these answers the error diagnoser draws the conclusion that the clause of $ef/1$ is incorrect, and reports this to the user. The clause, however, is correct, the error is in the external procedure for *fib*. With this approach the error can not be related to the function call since it is not visible in the nodes, only its result.

This realization leads to the reasonable conclusion that sometimes the oracle has to answer questions about function calls and their respective results. In fig. 4.4 the proof tree labelled with evaluated instances of atoms has nodes augmented with the corresponding function calls and their results.

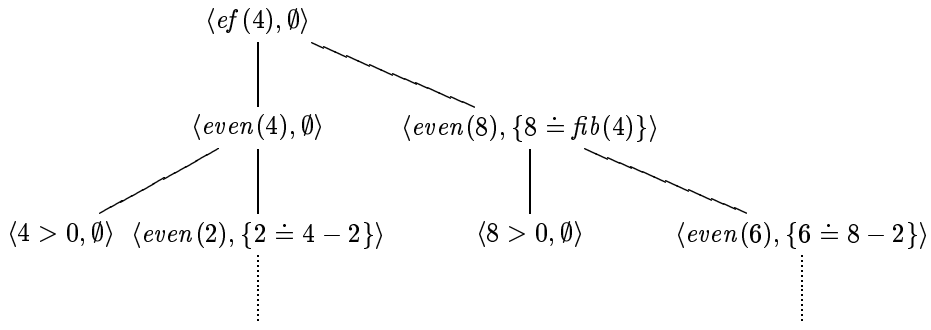

Figure 4.4: Proof tree of $\leftarrow ef(4)$, labelled with evaluated instances of atoms, and function calls with values

For the diagnoser to find out that the error is in the external procedure for computing the Fibonacci function the oracle has to answer questions about all computed constraints originating from the suspected erroneous clause.

Is $ef(4)$ valid? No.

Is $even(4)$ valid? Yes.

Is $even(8)$ valid? Yes.

Is $8 =_I fib(4)$? No.

After the three first questions the diagnoser can conclude that the clause that define the $ef/1$-predicate is the clause where the error appears, but to locate the error to the *fib*-function the last question is needed.

The drawback with this way of constructing proof trees is that no distinction is made between functional terms in head and body atoms. If the distinction is not made, the oracle might have to answer more questions than necessary. To illustrate this the program in fig. 4.5 is introduced.

```
p(X, Y) :-
     q(f(X), Y).

q(X, g(X)).
```

Figure 4.5: p/2

Assume that the GAPLog program and the external procedure implementing g are correct, but that there is an error in the external procedure for f. Further assume that $p(a, b)$, where a and b are arbitrary constants, is an error symptom. The proof tree, as outlined above, for $p(a, b)$ is displayed in fig. 4.6.

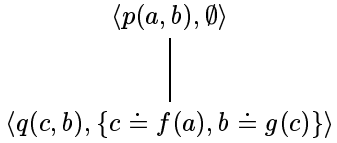

Figure 4.6: Proof tree

If it is used for error diagnosis a dialogue similar to the following would ensue.

Is $p(a, b)$ valid? No.

Is $q(c, b)$ valid? Yes.

Is $b =_I g(c)$? Yes.

Is $c =_I f(a)$? No.

In the same way as above, the two first questions are enough to be able to conclude that the error appears in the clause defining p/2. Hence, the question about g is unnecessary, since it can not give any useful information. However, the actual error can be pinpointed to the external procedure for f, though. With this in mind, a definition of proof trees is provided, in which function calls and their values are associated with the clause instances from which they stem, rather than with an atom.

4.2 Locating the error

In this section an algorithm is presented for locating incorrectness errors in the presence of possible errors in external procedures.

Definition 22 (Proof tree) *A tree T is a* proof tree *for a constrained atom $D[]A$, a program P and a set $\mathcal{E}$ of equations describing the external functions used in P iff*

- *each node is labelled with $\langle A', C' \rangle$, where A' is an atom without function calls, and C' is a set of primitive constraints.*
- *each node N has children $N_1, \ldots, N_n, (n \geq 0)$ iff there exists a substitution θ and a clause, in P, which can be renamed and flattened to $A_0 \leftarrow C_0, A_1, C_1, \ldots, A_n, C_n$, such that*
  - *N is labelled with $\langle A_0\theta, \bigcup_{i=0}^{n} C_i\theta \rangle$, and*
  - *each N_i is labelled with $A_i\theta$ and a set of constraints, and*
  - *all the ground constraints in $C_0\theta, \ldots, C_n\theta$ are true in $\mathcal{E}$.*
- *the root is labelled with $\langle A, C \rangle$.*
- *D is the set of non-ground primitive constraints in the labels of the nodes in T.*

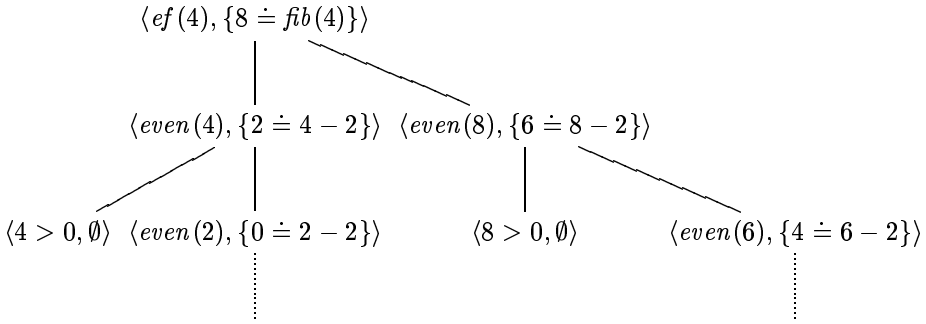

Figure 4.7: Proof tree according to def. 22 for $\leftarrow ef(4)$

The following is the final algorithm for diagnosing incorrectness errors in GAPLog programs. Given an incorrectness error symptom $C[]A]$ it can be used to find the erroneous clause.

Algorithm 2 *1. Construct the proof tree for an incorrectness error symptom* $C[]A$.

2. Traverse the proof tree and ask the oracle for nodes $\langle A', C' \rangle$ *about the validity of the constrained atoms* $\exists_{-Vars(A)} C[]A'$.

3. If such a constrained atom is not valid in the intended model, then continue to search for the error in the node's subtrees. If it is valid continue to search for the error in its siblings.

4. If such a constrained atom is not valid in the intended model, but all its n children's' constrained atoms $\exists_{-Vars(A)} C[]A'_i$ *are valid, then ask the oracle if the ground primitive constraints in* C' *are true. If they are, output* $A' \leftarrow A'_1, \ldots, A'_n$ *as an instance of an erroneous clause. Otherwise one of the external procedures used in the clause is incorrect and is the reason for the error symptom.*

The appendices of this thesis provide examples of using a debugger with an implementation of this algorithm.

Chapter 5

Insufficiency diagnosis

In Shapiro's approach [25] to insufficiency diagnosis the oracle is supposed to provide the diagnoser with true instances of atoms. In the worst case all instances of a given atom. This can of course be quite tedious work. In the approach of Pereira [21] all answers to an atomic query are computed by the diagnoser, and the oracle has to decide whether anything is missing. Even though this can be hard, it is easier than providing all correct answers. It is also possible to simulate a Shapiro-style debugger with a Pereira-style debugger, but the converse is not. Because of these reasons we decided to develop insufficiency diagnosis for GAPLog following Pereira's approach. Furthermore, it is quite straightforward to construct a Shapiro-style debugger for GAPLog, since the debuggers using that style are insensitive to the computation rule.

The result of the insufficiency diagnosis algorithm is a not completely covered atom (def. 16), i. e. an atom that has a ground instance which should succeed, but the bodies of all clauses whose head the instance matches correctly fail. To correct this error one usually has to add another clause, or generalize existing clauses. In the case of GAPLog, constraints need to be considered.

The algorithm presented here is an extension of an algorithm by Drabent et al [5]. First the auxiliary notion of search forest is defined. The definition of search forest assumes that GAPLog uses the left-most selection rule.

Definition 23 (Search forest) *For a program P and a query $\leftarrow C[]A$ the* search forest *is defined as follows:*

There is a tree for each clause $H \leftarrow B_1, \ldots, B_n (n > 0)$ of P, such that S-unification of $\{A \doteq H\} \cup C$ does not fail, and hence produces a set U of equations. Let C' be the set of primitive constraints in U, and let θ be the substitution corresponding to $U - C'$. Then

> *(B_1, θ, C') is the root of the corresponding tree*

and for each node (B_i, γ, D) of the tree, where program P gives $\sigma_1, \ldots, \sigma_m (m \geq 0)$ as computed answer substitutions and $D'_1, \ldots, D'_m$ as corresponding constraints to the query $D[]B_i\gamma$,

> *then $(B_{i+1}, \theta\sigma_j, D'_j)$ for each $j = 1, \ldots, m$ is a child of this node if $i < n$*
>
> *and $(\Box, \theta\sigma_j, D'_j)$ for each $j = 1, \ldots, m$ is a child of this node if $i = n$.*

Note that (B, θ, D) is a node in the forest iff $B\theta$ is a selected goal on the top level in the computation for $C[]A$ and D is the set of constraints when it is selected. Note also that $(\Box, \ldots, \emptyset)$ leaves correspond to successes of $\leftarrow C[]A$, and, consequently, $(B, \ldots, \ldots), B \neq \Box$ leaves to failures. If $(\Box, \theta, \emptyset)$ is a leaf in the forest, then the goal $\leftarrow C[]A$ succeeds with computed answer substitution $\theta_{|variables(A)}$. Leaves like $(\Box, \ldots, D)$ where $D \neq \emptyset$ correspond to *don't know* answers for $\leftarrow C[]A$. For a given A the search forest is unique up to renaming.

5.1 A Preliminary Algorithm for GAPLog

We will now give an insufficiency diagnosis algorithm for GAPLog programs, under the assumption that any external function used in the computation is correct. In sec. 5.2 we will deal with the case when we get insufficiency due to errors in external functions.

1. For an insufficiency symptom $C[]A$, construct the search forest.
2. For a *leaf* $(B, \theta, D), B \neq \Box$, of the search forest, ask if the constrained atom $D[]B\theta$ is satisfiable in the intended model. If the answer is YES, the algorithm is recursively called with $(B\theta, D)$.
3. For an *internal node* (B, θ, D), of the search forest, with children $(B', \theta\sigma_1, D_1)$, $(B', \theta\sigma_2, D_2)$, ..., $(B', \theta\sigma_n, D_n)$, ask the completeness question about the constrained atom $D[]B\theta$ and the set $\{D_1[]A_1, \ldots, D_n[]A_n\}$, where $A_1, \ldots, A_n$ are the evaluated instances of $B\theta\sigma_1, \ldots, B\theta\sigma_n$. If the answer is NO, the algorithm is recursively called with $D[]B\theta$.
4. If all questions about the nodes in the forest for $C[]A$ have been asked and no recursive call have been made on any of them, then $C[]A$ is a not completely covered atom.

5.2 Errors in external procedures

As is the case with incorrectness diagnosis, errors in external procedures must be dealt with in insufficiency diagnosis too. To begin with, we divide the problem into two subproblems. One for the case when the functional terms can be computed at call, and one when they can not be computed until later.

5.2.1 Unconstrained queries

We will use the same example program as for incorrectness diagnosis, the $ef/1$ program (fig. 4.1), and assume the same situation as we did then, that the GAPLog program is correct, but the Fibonacci procedure computes the Fibonacci number of n to be the Fibonacci number of $n+2$. E.g. $fib(6)$ will be 21, instead of 8. This leads to failure of $\leftarrow ef(6)$, even though it should succeed.

To diagnose insufficiency the diagnoser constructs a search forest for the query, and then asks the oracle existential and incompleteness questions about the nodes. Let the nodes of the search forest be labelled with instances of body atoms in the program, as defined in definition 23. The search forest for $\leftarrow ef(6)$ is depicted in fig. 5.1.

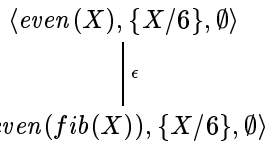

Figure 5.1: Insufficiency search forest for $\leftarrow ef(6)$, with leaves labelled by body atoms

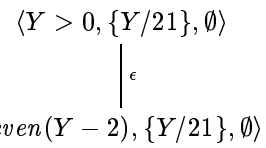

Figure 5.2: Insufficiency search forest for $\leftarrow even(21)$, with leaves labelled with body atoms

If the diagnosis is performed with the search forests of figures 5.1 and 5.2 as basis, the following dialogue could ensue between the diagnoser and the oracle.

> Is $even(fib(6))$ satisfiable? Yes. (This question arises from the search forest for $\leftarrow ef(6)$, fig. 5.1.)
>
> Is $even(21-2)$ satisfiable? No. (This question arises from the search forest for $\leftarrow even(21)$, fig. 5.2.)

From these answers the diagnoser concludes that $even(fib(6))$ is not completely covered. According to the intended model $fib(6)$ is 8, and $even(8)$ is true. Moreover, $even(8) \leftarrow 8 > 0, even(8-2)$ is an instance of a clause of the program and its body is true in the intended model. Hence, $even(fib(6))$ is completely covered and the conclusion is wrong.

Similarly as for incorrectness diagnosis we can choose to label the nodes of search forests with evaluated instances of atoms instead of instantiated program atoms. This of course leads to a different result.

Labelling the nodes of a search forest for the query $\leftarrow ef(6)$ with evaluated instances of atoms yields the search forest in figure 5.3. Using it for diagnosing the query yields the following question and answer from the diagnoser and the oracle respectively.

> Is $even(21)$ satisfiable? No.

From this answer the error diagnoser claims that $ef(6)$ is not completely covered. This is not true, because the body of the clause instance $ef(6) \leftarrow even(6), even(fib(6))$ is true in the intended model. To solve this we must ask questions about the results of the function calls used in the computation.

One solution is to pose the questions to the oracle about atoms in evaluated form, and to ask the oracle about the results of executing external procedures, similarly as for incorrectness diagnosis. Another is to ask the questions about instantiated, but not evaluated, atoms.

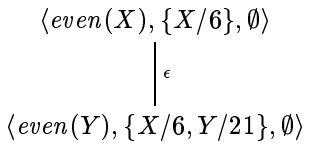

Figure 5.3: Insufficiency search forest for $\leftarrow ef(6)$

5.2.2 Constrained queries

The next problem to handle is delayed constraints. We will use the program in fig. 5.4. The predicate $fib/2$ is describes the relation between natural numbers and their Fibonacci numbers. The program computes it in a roundabout way, though.

```
fib(X, fib(Y)) :-
   id(X, Y).

id(X, X).
```

Figure 5.4: fib/2

Assume that the GAPLog code is correct, but the Fibonacci procedure has the same error as in the previous section. If we call the program with $fib(6, 8)$ we get failure, which we should not. Hence, we have an insufficiency error symptom. The small search forest in fig. 5.5 is generated for $\leftarrow \emptyset[]fib(6, 8)$.

$\langle id(X, Y), \{X/6\}, \{8 \doteq fib(Y)\}\rangle$

Figure 5.5: Insufficiency search forest with delayed constraint

If we naively use the insufficiency algorithm 5.1 the following dialogue would ensue:

> Is id(6, A) and the constraint set $\{8 \doteq fib(A)\}$ satisfiable? Yes.

The answer to that question is *yes* because for $A = 6$ both $8 \doteq fib(A)$ and $id(6, A)$ are true in the intended model.

The diagnoser would from this short dialogue draw the conclusion that $8 \doteq fib(A)[]id(6, A)$ is not completely covered. This is not correct, the error is in an external procedure for the function fib.

5.2.3 The Insufficiency Algorithm for GAPLog with erroneous function calls

1. For an insufficiency error symptom $C[]A$, construct its search forest.

2. Ask the oracle about the results of all function calls computed during S-unification of $C[]A$ with the clause heads.. An incorrect result of a function call implies that the corresponding external procedure is the reason for insufficiency.

3. For a *leaf* $(B, \theta, D), B \neq \Box$ of the search forest, the existential question is asked about the instance of $D[]B\theta$. If the answer is YES, the algorithm is recursively called with $(B\theta, D)$.

4. For an *internal node* (B, θ, D), of the search forest, with children $(B', \theta\sigma_1, D_1)$, $(B', \theta\sigma_2, D_2)$, ..., $(B', \theta\sigma_n, D_n)$, ask the completeness question about the constrained atom $D[]B\theta$ and the set $\{D_1[]A_1, \ldots, D_n[]A_n\}$, where $A_1, \ldots, A_n$ are the evaluated instances of $B\theta\sigma_1, \ldots, B\theta\sigma_n$. If the answer is NO, the algorithm is recursively called with $D[]B\theta$.

5. If all questions about the nodes in the forest for $C[]A$ have been asked and no recursive call have been made on any of them, then $C[]A$ is a not completely covered atom.

Appendix B provide examples of using an implementation of the algorithm above.

Chapter 6

Relevant constraints

As already pointed out, one of the main difficulties in adapting to GAPLog the classical declarative diagnosis techniques is the treatment of constraints. One of the main impacts they have is on the formulation of questions to the oracle.

The incorrectness diagnoser asks universal questions which contain all the constraints of a proof tree. An existential question asked by the insufficiency diagnoser contains all the constraints of the current computation state. The situation with the completeness questions is similar, however they refer to more than one computation states. In all these cases the number of constraints contained in a question can be very big. So it is important to be able to simplify the questions whenever possible.

Below we describe how to do this by means of projection of constraints. Additionally, for the questions used in the insufficiency diagnosis, we discuss removing the constraints that do not participate in (the relevant fragment of) the computation.

Constraint projection

For a constraint C and an atom A, we will denote the formula $\exists_{-Vars(A)}C$ by $C|A$. This formula will be called the *projection* of C onto A. Notice that $\forall A \leftarrow C$ is equivalent to $\forall A \leftarrow (C|A)$. Thus the universal question about $C[]A$ is equivalent to the universal question about $(C|A)[]A$. Also whenever a constrained atom $C[]A$ appears in an existential or completeness question, it can be replaced by $(C|A)[]A$. This follows from the following fact: An atom B is an instance $A\theta$ of A such that $I \models C\theta$ iff B is an instance $A\sigma$ of A such that $I \models (C|A)\sigma$.

So in the oracle questions we can use projections instead of the constraints themselves. This makes it possible to remove some primitive constraints from the questions. To make it precise, consider a constraint $C = C' \cup \{t \doteq f(\ldots)\}$, where t is a constructor term (possibly a variable). If each variable of t occurs neither in C' nor in A then $(C|A)$ and $(C'|A)$ are equivalent.

This property suggests a following algorithm of simplifying constraints in the questions. For a given constrained atom $C[]A$, remove from C a constraint $t \doteq f(\ldots)$ satisfying the condition above. Repeat this until no such constraint exists. By abuse of terminology the process of removing constraints will be called *computing projections* and its result will be called a projection of C on A.

As already stated, computing projections can be used to simplify constraints occurring in all the three kind of oracle questions. (See examples 15 and 16.)

A further possibility for insufficiency diagnosis

In the insufficiency diagnosis there exists a further possibility of removing primitive constraints from oracle questions. The diagnosing algorithm simulates the computation for a given constrained query $C[]A$, which is an insufficiency symptom. Assume that a primitive constraint $t \doteq f(\ldots)$ from C is not evaluated during the computation, because the arguments of f never become ground. (This concerns both the successful and not successful branches of the search tree). Then we can remove this constraint from C and perform debugging starting from $(C - \{t \doteq f(\ldots)\})[]A$ instead. It could be shown that the latter is also an insufficiency symptom.

Thus constraints that never become ground in the computation can be removed from the existential and completeness questions. Such constraints can be found dynamically, by an appropriate analysis at run-time. Below we present a way of finding (some of) them statically. Among the primitive constraints of $C[]A$ we distinguish those that depend on A. Only they may become ground during the execution of the constrained query. Thus the remaining ones can be removed.

The objective of the following definition is to capture all constraints whose right-hand sides can become ground by making some variables in an atom ground.

Definition 24 (Dependent constraint) *Assume a set C of primitive constraints in normal form. Then a constraint $c \in C$ is* dependent *on a query A if each variable in the right hand side of c is either in A or in the left-hand side of another constraint $c' \in C$ which is dependent on $\leftarrow A$.*

The following is an example of how a set of constraints can be simplified by removing those that are not dependent.

Example 14 *Assume that we have a query $p(Y)$ and a set of constraints $C = \{X \doteq f(Z), s(Z) \doteq g(Y), Y \doteq h(X,W)\}$, where $f/1$, $g/1$, and $h/2$ are defined functors, and $s/1$ is a constructor. Then we immediately see that $s(Z) \doteq g(Y)$ is dependent on $\leftarrow p(Y)$, and because of this, so is $X \doteq f(Z)$. On the other hand, $Y \doteq h(X,W)$ is not dependent on $p(Y)$.*

In the following two examples we see how projection and removal of not dependent constraints sn used to simplify a set of constraints..

Example 15 *Assume that we have a query $p(X,Y)$ and a set of constraints $C = \{Z \doteq f(X), s(X,W) \doteq g(Y), s(X,W) \doteq g(V)\}$, where $f/1$ and $g/1$ are defined functors, and $s/2$ is a constructor. Then it follows by projection that $Z \doteq f(X)$ can be removed. Furthermore, $s(X,W) \doteq g(V)$ can be removed because it is not depdendent on $p(X,Y)$. Only $s(X,W) \doteq g(Y)$ remain.*

Example 16 *Assume that we have a query $p(X,Y)$ and a set of constraints $C = \{Z \doteq f(X), s(W) \doteq g(Y), s(W) \doteq g(V)\}$, where $f/1$ and $g/1$ are defined functors, and $s/2$ is a constructor. Then, by projection, $Z \doteq f(X)$ can*

be removed and $s(W) \doteq g(V)$ *can be removed because it is not depdendent on* $p(X, Y)$. *If projection is applied once more, also* $s(X, W) \doteq g(Y)$ *can be removed.*

By example 16 we draw the conclusion that projection and removal of not dependent constraints should not be performed separately, but intertwined.

Chapter 7

Summary

We have shown how the concept of declarative diagnosis can be applied to a functional logic programming language GAPLog. GAPLog is a language that integrates, in a logically sound way, logic programs with functional procedures written in other programming languages.

We provided the necessary generalization of the notions of the intended model, incorrectness and insufficiency symptoms and not completely covered atoms. We generalized the oracle questions. We provided incorrectness and insufficiency diagnosis algorithms. They take into account the possibility that the reason of the error may be an incorrect external procedure. They also consider cases where at the end of the computation not all the delayed external calls have been executed. We treat them similarly to answer constraints in constraint logic programming.

Our insufficiency diagnosis algorithm is an extension of that given in [5]. It does not require the user to provide correct instances of given queries. Such debugging algorithms usually depend on the fixed computation rule of Prolog. In GAPlog however the external procedure calls are delayed; we generalized the algorithm to correctly deal with them.

The diagnosis algorithms were implemented and some debugging experiments were performed (conf. the appendices).

The oracle questions may contain complicated constraints. This makes them difficult to understand and answer by a human. We discussed some means of simplifying them. The first of them is projection. Additionally, for the queries used in the insufficiency diagnosis, we discussed removing the constraints that do not participate in (the relevant fragment of) the computation. Still, as the example in appendix B.1. shows, too complicated constraints in the queries may make the insufficiency diagnosis hardly applicable. This is an important problem for further research. A possible way to cope with it could be application of a "Shapiro-style" algorithm, where the user is required to provide correct instances of given queries.

GAPLog can be seen as a special and rather simplified case of a constraint logic programming language. We expect that many ideas of this paper can be applicable to declarative debugging of constraint logic programs in general. Also we believe that the encountered problems concerning dealing with constraints are representative for constraint logic programming.

Appendix A

A Fibonacci numbers example

The Fibonacci numbers can be recursively defined as

$$fib_n = \begin{cases} n & \text{if } n = 0, 1 \\ fib_{n-2} + fib_{n-1} & \text{otherwise} \end{cases}$$

A direct implementation of that definition results in an unnecessarily inefficient program, due to frequent recomputation of Fibonacci numbers. One of the possible optimizations is to memorize a Fibonacci number as soon as it has been computed, so that it only have to be looked up in a table when it is needed afterwards.

The program below is intended to implement the relation corresponding to the Fibonacci numbers. An accumulator (a programming technique common in logic programming) is used to memorize Fibonacci numbers when they have been computed. Fibonacci numbers can be computed more efficiently, but the accumulator technique is a common tool in logic programming

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% fib/2
%
% The GAPLog predicate fib(+N, -M) computes the N:th
% Fibonacci number and unifies the result with M.

fib(N, M) :-
        fib(N, [], _, M).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% fib/4
%
% The GAPLog goal ?- fib(+N, +Fibs0, -Fibs, -M) computes the
% N:th Fibonacci number and unifies the result with M,
```

```
% provided that Fibs0 is a (possibly empty) list of terms
% fib(K,L), where K is an integer and L the corresponding
% Fibonacci number. Then upon success Fibs contain terms
% fib(K,L) where K is an integer and L the Fibonacci number
% corresponding to K. The second and third arguments
% together form a so called accumulator pair.

fib(0, Fibs, Fibs, 0).

fib(1, Fibs, Fibs, 1).

fib(N, Fibs, Fibs, R) :-
        N > 1,
        member(fib(N, R), Fibs),
        !. % Cut to prevent recomputation of previously computed
           % Fibonacci numbers.

fib(N, Fibs0, [ fib(N, M) | Fibs ], M+K) :- % Incorrectness Error
        N > 1,
        fib(N-2, Fibs0, Fibs1, K),
        fib(N-1, Fibs1, Fibs, M).
```

A.1 Incorrectness diagnosis

The Fibonacci program contains an incorrectness error. The clause head tagged with `Incorrectness Error` above should have read

```
fib(N, Fibs0, [ fib(N, M+K) | Fibs ], M+K)
```

instead. Because of this error incorrect values will be memorized.

The program computes the correct Fibonacci numbers of 0, 1, 2, 3, and 4, but it computes the Fibonacci number of 5 to be 4 instead of 5. Hence, *fib(5,4)* is an error symptom.

The following is a transcript of an error diagnosis session. (Some of the lines have been broken to fit on a row.)

```
?- inc(fib(5,M)).

Is   fib(5,4)   true?   n.

Is   fib(5,[],
           [fib(5,2),fib(4,1),fib(3,1),fib(2,1)],4)   true?   n.

Is   fib(4,[fib(3,1),fib(2,1)],
           [fib(4,1),fib(3,1),fib(2,1)],2)   true?   y.

Is   fib(3,[],[fib(3,1),fib(2,1)],2)   true?   y.

This is an instance of an incorrect clause:
```

```
fib(5, [], [fib(5,2),fib(4,1),fib(3,1),fib(2,1)], 4) :-
        5>1,
        fib(3, [], [fib(3,1),fib(2,1)], 2),
        fib(4, [fib(3,1),fib(2,1)],
               [fib(4,1),fib(3,1),fib(2,1)], 2).
```

Observe the question

```
Is   fib(4,[fib(3,1),fib(2,1)],
           [fib(4,1),fib(3,1),fib(2,1)],2)   true?   y.
```

in the diagnosis session.

It is not immediately obvious that the answer is *yes*. It even might be tempting to answer *no*, because the fourth Fibonacci number is 3, not 2. Nevertheless, given the memorized values, in the second argument, that $fib_2 = 1$ and $fib_3 = 1$, it is correct that fib_4 is 2.

Appendix B

Crypto-arithmetic puzzle example

Crypto-arithmetic puzzles are well-known mathematical pastimes. The most well-known example is the following: assign digits to each letter in the following addition so that it adds up correctly. No digit can be assigned to more than one letter. Usually it is understood that the most significant letter can not be assigned the digit 0.

```
   SEND
+  MORE
-------
  MONEY
```

The program below is intended to implement a solver for generalized crypto-arithmetic puzzles. Puzzles are generalized in the sense that there can be any number of terms of arbitrary size in the addition and the most significant letter of a word might be associated with the digit zero. The program contains some errors that generate both incorrectness and insufficiency error symptoms.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% Puzzle/2
%
% puzzle(+Puzzle, ?Mapping) is true if Puzzle is a general
% crypt-arithmetic puzzle with the mapping Mapping.

puzzle(Puzzle, Mapping) :-
        reverseall(Puzzle, Puzzle1),
        reverse(Puzzle1, [ Sum | Terms ]),
        flatten([ Sum | Terms ], Letters),
        uniqify(Letters, Unique),
        pairs(Unique, Digits, Mapping),
```

```
        compare(Sum, Terms, 0, Mapping),
        choice(Digits, [0,1,2,3,4,5,6,7,8,9]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% compare/4
%
% compare(+Sum, +Terms, +Carry, +Mapping) is true if the sum
% of the terms Terms and the Carry equals Sum. The
% representation is with letters whose numerical binding is
% in Mapping.

compare([], _, _, _). % Error, should be: compare([], _, 0, _).
compare([ Letter | RestSum ], Terms, Carry, Mapping) :-
        lookup(Letter, Mapping,
                  Sum - 10 * (Sum / 10) + Carry), % Errors
        sum(Terms, Mapping, RestTerms, Sum),
        compare(RestSum, RestTerms,
                  (Carry + Sum) / 10, Mapping). % Error

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% sum/4
%
% sum(+ListofLists, +Mapping, ?Terms, ?Value) is true if
% Value is the sum of the bindings in Mapping for each
% element first in every list in ListofLists and Terms is
% the list of the tails of each list in ListofLists.

sum([], _, [], 0).
sum([ [] | Rest ], Mapping, Terms, Value) :-
        sum(Rest, Mapping, Terms, Value).
sum([[ Letter | Letters ] | Terms ], Mapping,
        [ Letters | RestTerms ], Value + Sum) :-
   lookup(Letter, Mapping, Value),
   sum(Terms, Mapping, RestTerms, Sum).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% flatten/2,4
%
% flatten(+ListsOfLists, ?Flattened) is true if Flattened is
% a flat list of all objects in ListsOfLists, *including*
% empty lists.
% NB: Does only handle ground lists.

flatten(List, Flattened) :-
        flatten(List, Flattened, Flattened, Last),
        Last = [].
```

```
flatten([], _, Last, Last).
flatten([ Elem | Rest ], Flat, [ Elem | Last0 ], Last) :-
        \+ functor(Elem, '.', 2),
        flatten(Rest, Flat, Last0, Last).
flatten([ List | Rest ], Flat, Last0, Last) :-
        functor(List, '.', 2),
        flatten(List, Flat, Last0, Last1),
        flatten(Rest, Flat, Last1, Last).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% choice/2
%
% choice(?List1, ?List2) is true if List1 is a permutation
% of some of the elements in List2.

choice([], _).
choice([ Element | Rest ], Elements) :-
        select(Element, Elements, Unused),
        choice(Rest, Unused).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% select/3

select(Selected, [ Selected | Unused ], Unused).
select(Selected, [ Unused0 | Rest ], [ Unused0 | Unused ]) :-
        select(Selected, Rest, Unused).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% reverse/2-3

reverse(List, Reversed) :-
        reverse(List, [], Reversed).

reverse([], Reversed, Reversed).
reverse([ Head | Tail ], Rev0, Rev) :-
        reverse(Tail, [ Head | Rev0 ], Rev).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% reverseall/2

reverseall([], []).
reverseall([ List | Rest ], [ Reversed | RestRev ]) :-
        reverse(List, Reversed),
        reverseall(Rest, RestRev).
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% lookup/3

lookup(A, [ (A, V) | _ ], V).
lookup(A, [ _ | T ], V) :-
        lookup(A, T, V).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% pairs/3
%
% pairs(?List1, ?List2, ?Pairs) is true if Pairs is the list
% of pairs of consecutive elements in List1 and List2.

pairs([], [], []).
pairs([ H1 | T1 ], [ H2 | T2 ], [ (H1, H2) | Pairs ]) :-
        pairs(T1, T2, Pairs).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
% uniqify/2

uniqify(List, Unique) :-
        uniqify(List, [], Unique).
uniqify([], _, []).
uniqify([ Head | Tail ], Seen, Unique) :-
        member_unique(Head, Seen),
        !,
        uniqify(Tail, Seen, Unique).
uniqify([ Head | Tail ], Seen, [ Head | Unique ]) :-
        % \+ member_unique(Head, Seen),
        uniqify(Tail, [ Head | Seen ], Unique).

member_unique(X, [X|_]).
member_unique(X, [H|T]):-
        X \== H,
        member_unique(X, T).
```

B.1 Insufficiency diagnosis

The crypto-arithmetic puzzle

```
  ETT
  ETT
  ETT
  ETT
+ ETT
-----
```

```
FEM
```

where ETT means the number one in Swedish, and FEM means five, has exactly one solution. Due to the errors in the program it is not found. Hence,

$$puzzle([[e,t,t],[e,t,t],[e,t,t],[e,t,t],[e,t,t],[f,e,m]],M)$$

is an insufficiency symptom, and insufficiency error diagnosis can be started from it.

```
?- ins(puzzle([[e,t,t],[e,t,t],[e,t,t],[e,t,t],[e,t,t],[f,e,m]],
                  M)).

Is choice([A,B,C,D],[0,1,2,3,4,5,6,7,8,9]) and the
constraint set [F1 is K-E1, E1 is 10*L, D1 is C1/10,
C1 is Z+P, B1 is P-A1, A1 is 10*Q, Z is Y/10, Y is 0+U, X is U-W,
W is 10*V, V is U/10, U is D+T, T is D+S, S is D+R, R is D+E,
Q is P/10, P is D+0, 0 is D+N, N is D+M, M is D+F, L is K/10,
K is B+J, J is B+I, I is B+H, H is B+G, G is B+0, F is D+0,
E is D+0, A is X+0, B is B1+Z, C is F1+D1] satisfiable?  n.
```

It is not immediately obvious that the response to the question above should be *no*. Our intended model for `choice/2` gives us that the only possible values for the variables `A,B,C` and `D` are integers in the interval 0 to 9. The primitive constraint `V is U / 10` tells us that `V` must be a float, since `/` is floating point division in GAPLog. Hence, `W` and `X` must be floats too, because of the primitive constraints `W is 10*V` and `X is U-W`. Finally, we see that we have a type clash in `A is X+0`, and due to this the constraint is not satisfiable.

The diagnosis dialogue continues with the following question:

```
Does the following set contain all valid instances of the goal:
compare([m,e,f],[[t,t,e],[t,t,e],[t,t,e],[t,t,e],[t,t,e]],0,
          [(m,A),(e,B),(f,C),(t,D)])

answers:
compare([m,e,f],[[t,t,e],[t,t,e],[t,t,e],[t,t,e],[t,t,e]],0,
[(m,A),(e,B),(f,C),(t,D)]), [F1 is K-E1,E1 is 10*L,D1 is
C1/10,C1 is Z+P,B1 is P-A1,A1 is 10*Q,Z is Y/10,Y is 0+U,X
is U-W,W is 10*V,V is U/10,U is D+T,T is D+S,S is D+R,R is
D+E,Q is P/10,P is D+0,0 is D+N,N is D+M,M is D+F,L is
K/10,K is B+J,J is B+I,I is B+H,H is B+G,G is B+0,F is D+0,E
is D+0,A is X+0,B is B1+Z,C is F1+D1]

|: n.
```

In our intended model there is exactly one atom which is an instance of `compare([m,e,f],[[t,t,e],[t,t,e],[t,t,e],[t,t,e],[t,t,e]], 0,`

`[(m,A),(e,B),(f,C),(t,D)])`. The constraint above can be shown to be unsatisfiable, using the same reasoning as in the previous question. Thus, the answer to the question is *no*.

```
Does the following set contain all valid instances of the goal:
compare([e,f],[[t,e],[t,e],[t,e],[t,e],[t,e]],(0+A)/10,
          [(m,B),(e,C),(f,D),(t,E)])
and the constraint set  [L is E+K, K is E+J, J is E+I, I is E+0,
                  H is A-G,G is 10*F,F is A/10,B is H+0,A is E+L]
answers:
compare([e,f],[[t,e],[t,e],[t,e],[t,e],[t,e]],A,
           [(m,B),(e,C),(f,D),(t,E)]),
[F1 is L-E1, E1 is 10*M, D1 is C1/10, C1 is A+Q, B1 is Q-A1,
A1 is 10*R, Z is 0+V, Y is V-X, X is 10*W, W is V/10,
V is E+U, U is E+T, T is E+S, S is E+F, R is Q/10, Q is E+P,
P is E+O, O is E+N, N is E+G, M is L/10, L is C+K, K is C+J,
J is C+I, I is C+H, H is C+0, G is E+0, F is E+0, B is Y+0,
A is Z/10,C is B1+A, D is F1+D1]

|: y.
```

The constraint set in the goal is unsatisfiable for similar reason as above. Consequently any answer contains all valid instances of the goal, since there are none. This reasoning apply to the next question too.

```
Does the following set contain all valid instances of the goal:
sum([[t,t,e],[t,t,e],[t,t,e],[t,t,e],[t,t,e]],
       [(m,A),(e,B),(f,C),(t,D)],E,F)
and the constraint set  [I is F-H,H is 10*G,G is F/10,A is I+0]

answers:
sum([[t,t,e],[t,t,e],[t,t,e],[t,t,e],[t,t,e]],
        [(m,A),(e,B),(f,C),(t,D)],
        [[t,e],[t,e],[t,e],[t,e],[t,e]],
        D+E),
[L is F-K, K is 10*J, J is F/10, I is D+H, H is D+G, G is D+0,
F is D+E, E is D+I, A is L+0]

|: y.

Does the following set contain all valid instances of the goal:
lookup(m,[(m,A),(e,B),(f,C),(t,D)],E-10*(E/10)+0)
answers:
lookup(m,[(m,A),(e,B),(f,C),(t,D)],A),[]

|: y.


This atom is not completely covered:
```

```
compare([m,e,f],[[t,t,e],[t,t,e],[t,t,e],[t,t,e],[t,t,e]],0,
            [(m,B),(e,C),(f,D),(t,E)])
```

The actual error is that integer division has been replaced by floating point division in the clause defining `compare`/4. The corrected clause looks as follows:

```
compare([ Letter | RestSum ], Terms, Carry, Mapping) :-
        lookup(Letter, Mapping,
                  Sum - 10 * (Sum // 10) + Carry), % Error
        sum(Terms, Mapping, RestTerms, Sum),
        compare(RestSum, RestTerms,
                  (Carry + Sum) // 10, Mapping).
```

Notice that the clause is still erroneous.

Obviously, insufficiency diagnosis when such complicated constraints appear is not feasible to perform in this way.

B.2 Insufficiency diagnosis

One way to avoid at least some of the inconveniences with complicated constraints can be to start diagnosis from a ground error symptom. There is only one ground instance of the error symptom in sec. B.1, namely

$puzzle([[e,t,t],[e,t,t],[e,t,t],[e,t,t],[e,t,t],[f,e,m]],[(m,0),(e,1),(f,6),(t,2)])$

The error diagnosis session of sec. B.1 is redone for the ground error symptom, and we see that it is much simpler.

```
?- ins(puzzle([[e,t,t],[e,t,t],[e,t,t],[e,t,t],[e,t,t],[f,e,m]],
                [(m,0),(e,1),(f,6),(t,2)])).

Is compare([m,e,f],[[t,t,e],[t,t,e],[t,t,e],[t,t,e],[t,t,e]],0,
              [(m,0),(e,1),(f,6),(t,2)])   satisfiable?   y.

Is sum([[t,t,e],[t,t,e],[t,t,e],[t,t,e],[t,t,e]],
          [(m,0),(e,1),(f,6),(t,2)],A,B)
and the constraint set
[E is B-D,D is 10*C,C is B/10,0 is E+0] satisfiable?   n.

Does the following set contain all valid instances of the goal:
lookup(m,[(m,0),(e,1),(f,6),(t,2)],A-10*(A/10)+0)
answers:
lookup(m,[(m,0),(e,1),(f,6),(t,2)],0),[]

|: y.

This atom is not completely covered:
compare([m,e,f],[[t,t,e],[t,t,e],[t,t,e],[t,t,e],[t,t,e]],0,
          [(m,0),(e,1),(f,6),(t,2)])
```

Thanks to starting diagnosis with a ground insufficiency symptom the questions became much easier.

B.3 Incorrectness diagnosis

When the erroneous clause has been replaced for the (still erroneous) clause at the end of sec. B.1 the crypto-arithmetic puzzle solver contains an error which can lead to the error symptom

```
puzzle([[e,t,t],[e,t,t],[e,t,t],[e,t,t],[e,t,t],[f,e,m]],
  [(m,0),(e,3),(f,8),(t,6)]).
```

This is an incorrectness error symptom, and we try to find the error.

```
?- inc(puzzle([[e,t,t],[e,t,t],[e,t,t],[e,t,t],[e,t,t],[f,e,m]],
                M)).
Is   puzzle([[e,t,t],[e,t,t],[e,t,t],[e,t,t],[e,t,t],[f,e,m]],
             [(m,0),(e,1),(f,6),(t,2)])   true?   y.
Is   puzzle([[e,t,t],[e,t,t],[e,t,t],[e,t,t],[e,t,t],[f,e,m]],
             [(m,0),(e,3),(f,8),(t,6)])   true?   n.
Is   choice([0,3,8,6],[0,1,2,3,4,5,6,7,8,9])   true?   y.
Is   compare([m,e,f],[[t,t,e],[t,t,e],[t,t,e],[t,t,e],[t,t,e]],0,
             [(m,0),(e,3),(f,8),(t,6)])   true?   n.
Is   compare([e,f],[[t,e],[t,e],[t,e],[t,e],[t,e]],3,
             [(m,0),(e,3),(f,8),(t,6)])   true?   n.
Is   compare([f],[[e],[e],[e],[e],[e]],3,
             [(m,0),(e,3),(f,8),(t,6)])   true?   n.
Is   compare([],[[],[],[],[],[],[]],1,
             [(m,0),(e,3),(f,8),(t,6)])   true?   n.

This is an incorrect clause:
compare([], [[],[],[],[],[],[]], 1, [(m,0),(e,3),(f,8),(t,6)]).
```

The answer to the last question must be *no*, because the carry argument is 1, but the sum argument is empty. The corrected fact looks like:

```
compare([], 0, _, _).
```

B.4 Insufficiency diagnosis

After correcting the previous error there is still an error left in the program. The puzzle

```
   A
   A
   A
   A
   A
   A
   A
   A
   A
   A
```

```
   A
+  A
----
 BCD
```

have got solutions, but the query

```
puzzle([[a],[a],[a],[a],[a],[a],[a],[a],[a],[a],[a],[a],[a],
        [b,c,d]], M)
```

fails.

The diagnosis session is started with one of the four ground insufficiency symptoms. Otherwise the diagnosis becomes as hard as in sec. B.1.

```
?- ins(puzzle([[a],[a],[a],[a],[a],[a],[a],
                    [a],[a],[a],[a],[a],[b,c,d]],
              [(d,8),(c,0),(b,1),(a,9)])).

Is compare([d,c,b], [[a],[a],[a],[a],[a],[a],
                             [a],[a],[a],[a],[a],[a]],
           0, [(d,8),(c,0),(b,1),(a,9)])   satisfiable?   y.

Is compare([c,b], [[],[],[],[],[],[],[],[],[],[],[],[]],
             (0+108)//10, [(d,8),(c,0),(b,1),(a,9)])
satisfiable? y.

Is sum([[],[],[],[],[],[],[],[],[],[],[],[]],
         [(d,8),(c,0),(b,1),(a,9)], A, B)
and the constraint set
[E is B-D, D is 10*C, C is B//10, 0 is E+10]
satisfiable?   n.

Does the following set contain all valid instances of the goal:
lookup(c, [(d,8),(c,0),(b,1),(a,9)], A-10*(A//10)+10)
answers:
lookup(c,[(d,8),(c,0),(b,1),(a,9)],0),[]

|: y.
This atom is not completely covered:
compare([c,b],[[],[],[],[],[],[],[],[],[],[],[],[]], (0+108)//10,
            [(d,8),(c,0),(b,1),(a,9)])

The uncovered atom matches the head of these unused clauses:
compare([c,b], [[],[],[],[],[],[],[],[],[],[],[],[]],
               (0+108)//10, [(d,8),(c,0),(b,1),(a,9)]) :-
        lookup(c, [(d,8),(c,0),(b,1),(a,9)],
               A-10*(A//10)+(0+108)//10),
        sum([[],[],[],[],[],[],[],[],[],[],[],[]],
               [(d,8),(c,0),(b,1),(a,9)], B, A),
        compare([b], B, ((0+108)//10+A)//10,
                [(d,8),(c,0),(b,1),(a,9)]).
```

The corrected version of the clause looks like

```
compare([ Letter | RestSum ], Terms, Carry, Mapping) :-
        lookup(Letter, Mapping,
                  (Carry + Sum) - 10 * ((Carry + Sum) // 10)),
        sum(Terms, Mapping, RestTerms, Sum),
        compare(RestSum, RestTerms,
                 (Carry + Sum) // 10, Mapping).
```

Bibliography

[1] S. Bonnier. Horn Clause Logic with External Procedures: Towards a Theoretical Framework. Licentiate Thesis 197, Department of Computer and Information Science, Linköping University, 1989.

[2] Lawrence Byrd. Understanding the control flow of logic programs. In S-Å. Tärnlund, editor, *Proceedings of the logic programming workshop*, pages 127–138, Debrecen, Hungary, 1980.

[3] Marco Comini, Giorgio Levi, and Giuliana Vitiello. Declarative diagnosis revisited. In John Lloyd, editor, *Logic Programming, Proc. of the 1995 International Symposium*, pages 275–287. MIT Press, 1995.

[4] Marco Comini, Giorgio Levi, and Giuliana Vitiello. Efficient detection of incompleteness errors in the abstract debugging of logic programs. In Mireille Ducassé, editor, *Proc. 2nd International Workshop on Automated and Algoritmic Debugging*. IRISA-CNRS, 1995.

[5] 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.

[6] Marc Eisenstadt. A powerful Prolog trace package. In Tim O'Shea, editor, *Advances in artificial intelligence*, pages 149–158. North-Holland, 1985.

[7] Marc Eisenstadt and Mike Brayshaw. The transparent Prolog machine (TPM): an execution model and graphical debugger for logic programming. *The Journal of Logic Programming*, 5(4):277–342, December 1988.

[8] Gérrand Ferrand. Error diagnosis in logic programming, an adaption of E. Y. Shapiro's method. *The Journal of Logic Programming*, 4(3):177–198, September 1987.

[9] M. Hanus and B. Josephs. A debugging model for functional logic programs. In *Proc. of the 5th International Symposium on Programming Language Implementation and Logic Programming*, pages 28–43. Springer LNCS 714, 1993.

[10] Tim Heyer. Felsökning i GAPLog: En modell för exekvering av logikprogram med fördröjda funktionsanrop. Master's thesis, Linköping University, 1994. Report-no: LiTH-IDA-Ex-9441. The report is in Swedish.

[11] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *The Journal of Logic Programming*, 19 & 20:503–582, May 1994.

[12] François Le Berre and Alexandre Tessier. Declarative Incorrectness Diagnosis in Constraint Logic Programming. In *Joint Conference on Declarative Programming*, 1996.

[13] J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.

[14] John W. Lloyd. *Foundations of Logic Programming.* Springer Verlag, second edition, 1987.

[15] Jan Małuszyński, Staffan Bonnier, Johan Boye, Feliks Kluźniak, Andreas Kågedal, and Ulf Nilsson. Logic programs with external procedures. In Krzysztof Apt, Jaco de Bakker, and Jan Rutten, editors, *Logic programming languages: constraints, functions, and objects.* The MIT Press, 1993.

[16] L. Naish, P. W. Dart, and J. Zobel. The NU-Prolog debugging environment. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 521–536, Lisbon, 1989. The MIT Press.

[17] Lee Naish. Adding equations to nu-prolog. In J. Małuszyński and M. Wirsing, editors, *Proceedings of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming, PLILP91, Passau, Germany*, number 528 in Lecture Notes in Computer Science, pages 15–26. Springer-Verlag, August 1991.

[18] Lee Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 10(3):255–285, 1992.

[19] Lee Naish and Tim Barbour. A declarative debugger for a logical-functional language. Technical Report 30, Dept. of Computer Science, The University of Melbourne, 1994.

[20] Henrik Nilsson and Peter Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 1994.

[21] Luís Moniz Pereira. Rational debugging in logic programming. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science, pages 203–210, London, 1986. Springer-Verlag.

[22] Dave Plummer. Coda: An extended debugger for PROLOG. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 496–511, Seattle, 1988. ALP, IEEE, The MIT Press.

[23] A. Schleiermacher and J. F. H. Winkler. The implemetation of ProTest: a Prolog debugger for a refined box model. *Software - Practice & Experience*, 20(10):985–1006, 1990.

[24] Nahid Shahmehri. *Generalized Algorithmic Debugging.* PhD thesis, Linköping University, 1991.

[25] Ehud Y. Shapiro. *Algorithmic Program Debugging.* ACM Distinguished Dissertations. The MIT Press, 1983.

[26] Alexandre Tessier. Declarative Debugging in Constraint Logic Programming: the Cover Relation. Technical Report 96/09, LIFO, University of Orléans, 1996.

[27] Toshio Yokoi, Shunichi Uchida, and ICOT Third Laboratory. Sequential Inference Machine: SIM - Its programming and operating system. In ICOT, editor, *International conference on fifth generation computer systems*, pages 70–81. North-Holland, 1984.