URL: http://www.elsevier.nl/locate/entcs/volume86.html 20 pages

Cost-Sensitive Diagnosis of Declarative Programs

D. Ballis[b] M. Falaschi[b] C. Ferri[a] J. Hernández-Orallo[a] M.J. Ramírez-Quintana[a]

[a] *DSIC, Universidad Politécnica de Valencia, Camino de Vera s/n, Apdo. 22012, 46071 Valencia, Spain. Email: {cferri,jorallo,mramirez}@dsic.upv.es.*

[b] *Dip. Matematica e Informatica, Via delle Scienze 206, 33100 Udine, Italy. Email: {demis,falaschi}@dimi.uniud.it.*

Abstract

Diagnosis methods in debugging aim at detecting bugs of a program, either by comparing it with a correct specification or by the help of an oracle (typically, the user herself). Debugging techniques for declarative programs usually exploit the semantical properties of programs (and specifications) and generally try to detect one or more "buggy" rules. In this way, rules are split apart in an absolute way: either they are correct or not. However, in many situations, not every error has the same consequences, an issue that is ignored by classical debugging frameworks. In this paper, we generalise debugging by considering a cost function, i.e. a function that assigns different cost values to each kind of error and different benefit values to each kind of correct response. The problem is now redefined as assigning a real-valued probability and cost to each rule, by considering each rule more or less "guilty" of the overall error and cost of the program. This makes possible to *rank* rules rather than only separate them between right and wrong. Our debugging method is also different from classical approaches in that it is probabilistic, i.e. we use a set of ground examples to approximate these rankings.

Key words: Declarative Debugging, Cost Matrices, Software Testing, Cost-sensitive Evaluation.

1 Introduction

Debugging has always been an undesired stage in software development. Much research in the area of software development has been devoted to avoid errors

[1] This work has been partially supported by CICYT under grant TIC2001-2705-C03-01 and Acción Integrada Hispano-Italiana HI2000-0161.

but, in the general framework of programming, the errors are still there and, hence, must be detected and corrected.

The problem of debugging can be presented in several ways. In the worst case, we only know that the program does not work correctly, and little can be done automatically. Sometimes we are given several wrong (and optionally right) computations (e.g. from the profiler). In other situations, we take for granted that we have an oracle (generally the user) whom we can ask about the correct computation of any example. In the best case, we have a formal specification and the program to be corrected.

Even in the best case (we know the specification) the diagnosis of which parts of the program are wrong is not trivial. Successful techniques in the diagnosis of bugs are usually associated with the type of programming language used. Programming languages based on rules and, especially, declarative languages allow the use of powerful (formal) techniques to address the problem. Several approaches have been developed for logic programming [3], functional programming [8] and logic and functional integrated languages [1,2], in order to debug programs according to different observable properties, e.g., correct answers, computed answers, call patterns, finite failure, etc. Furthermore, some of these methods are also able to diagnose lost answers (the completeness problem).

However, many formal techniques derived so far are based on the idea of comparing specification and program, also assuming that all the predicates or functions implemented by the program are also defined by the specification, to which they can be compared, or assuming that auxiliary functions are correct.

So we would like to cope with large programs, where the specification of the auxiliary functions is not known, and taking into account the program usage distribution and the costs of each kind of error.

Let us illustrate the kind of problems we want to address with an example.

Example 1.1 Consider a taxi company that wants to use a program for sending the appropriate vehicle type (car, van, minibus or special service) depending on the number of people in the group that has made the call. The maximum capacities of each vehicle are 4 for cars, 8 for vans and 12 for minibusses. The specification I given by the company is then as follows [2]:

$$\texttt{taxi(N)} \rightarrow \texttt{car} \Leftarrow \texttt{N} \leq 4$$
$$\texttt{taxi(N)} \rightarrow \texttt{van} \Leftarrow 4 < \texttt{N} \leq 8$$
$$\texttt{taxi(N)} \rightarrow \texttt{minibus} \Leftarrow 8 < \texttt{N} \leq 12$$
$$\texttt{taxi(N)} \rightarrow \texttt{special} \Leftarrow 12 < \texttt{N}$$

A programmer just implements this problem as the following program R_1:

[2] In the example, $<$ and $\leq$ are the usual predefined boolean functions modelling inequality relations among naturals.

$$
\begin{aligned}
&r1 : \mathtt{cap(car)} \to \mathtt{4}\\
&r2 : \mathtt{cap(van)} \to \mathtt{9}\\
&r3 : \mathtt{cap(minibus)} \to \mathtt{12}\\
&r4 : \mathtt{interval(X,N,Z)} \to (\mathtt{X} < \mathtt{N} \leq \mathtt{Z})\\
&r5 : \mathtt{taxi(N)} \to \mathtt{car} \ \Leftarrow\ \mathtt{interval(0,N,cap(car))}\\
&r6 : \mathtt{taxi(N)} \to \mathtt{van} \ \Leftarrow\ \mathtt{interval(cap(car),N,cap(van))}\\
&r7 : \mathtt{taxi(N)} \to \mathtt{minibus} \ \Leftarrow\ \mathtt{interval(cap(van),N,cap(minibus))}\\
&r8 : \mathtt{taxi(N)} \to \mathtt{special} \ \Leftarrow\ \mathtt{cap(minibus)} < \mathtt{N}
\end{aligned}
$$

As can be seen, the programmer has made a mistake on the capacity of the van. However, the semantics of the auxiliary function `cap` measuring the“capacity” of a vehicle is not in the specification and, consequently, classical approaches cannot be applied unless assuming that the auxiliary functions are correct, which in this case it is not true.

Nonetheless, the previous example can also be used to show that even detecting errors is not sufficient for improving the quality of a program.

Example 1.2 Consider the alternative program R_2:

$$
\begin{aligned}
&r1 : \mathtt{cap(car)} \to \mathtt{3}\\
&r2 : \mathtt{cap(van)} \to \mathtt{7}\\
&r3 : \mathtt{cap(minibus)} \to \mathtt{12}\\
&r4 : \mathtt{interval(X,N,Z)} \to (\mathtt{X} < \mathtt{N} \leq \mathtt{Z})\\
&r5 : \mathtt{taxi(N)} \to \mathtt{car} \ \Leftarrow\ \mathtt{interval(0,N,cap(car))}\\
&r6 : \mathtt{taxi(N)} \to \mathtt{van} \ \Leftarrow\ \mathtt{interval(cap(car),N,cap(van))}\\
&r7 : \mathtt{taxi(N)} \to \mathtt{minibus} \ \Leftarrow\ \mathtt{interval(cap(van),N,cap(minibus))}\\
&r8 : \mathtt{taxi(N)} \to \mathtt{special} \ \Leftarrow\ \mathtt{N} > \mathtt{cap(minibus)}
\end{aligned}
$$

Apparently, this program is more buggy than R_1 since rules $r1$ and $r2$ seem to be wrong. However, in the context where this program is going to be used, it is likely that R_2 is better than R_1. But why?

The issue here is that in most situations, not every error has the same consequences. For instance, a wrong medical diagnosis or treatment can have different costs and dangers depending on which kind of mistake has been done. Obviously, costs of each error are problem dependent, but it is not common the case that they are uniform for a single problem. Note that it might even be cheaper not to correct a limited bug and invest that time in correcting other costlier bugs.

For the previous example, the cost of each error could be represented as a “cost matrix” and its values could have been estimated by the company[3]:

		I			
		car	*van*	*minibus*	*special*
	car	0	200	200	200
R	*van*	4	0	200	200
	minibus	10	6	0	200
	special	30	15	5	0

Obviously the cost of sending a car when a van is needed is much higher than the contrary situation, since in the first case the vehicle is not big enough to carry all the people. For this particular matrix and a non-biased distribution of examples, the program R_2 is better than the program R_1, even though apparently it has more buggy rules. Nonetheless a different distribution of examples (e.g. if groups of 4 people are much more common than groups of 9 people) could turn R_1 into a better program. Consequently, in order to know which rules are more problematic or costlier, we need to know:

- The cost of each kind of error given by a cost function.
- The distribution of cases that the program has to work with.

The previous example also shows that if we do not have the specification of the main function and all the auxiliary functions, it is impossible in general to talk about correct and incorrect rules. In these cases, it is only sensible to talk about rules that participate more or less in the program errors or costs, giving a rate rather than a sharp classification between good and bad rules. Another common situation is during the development of the software product when there is not enough time to correct all the errors before the date of the purchase of the product. In this case, it could be very useful to have a rank of errors in order to correct the most important ones first.

Taking into account the previous considerations, we are going to devise a new debugging schema (for decreasing programs) with the following requirements:

- It should be able to work with samples of examples, with oracles or with a correct specification, without the need of correct specifications for auxiliary functions.
- It should consider a user-defined or predefined cost function in order to perform cost-sensitive debugging.
- The previously obtained costs will be used to rank the rules from more costly to less costly rather than strictly separating between buggy or non-buggy

[3] This is usually a simplification, since the cost may also depend on the size, the age or even the extra number of people.

rules.

- It should be possible to use a tunable trade-off between efficiency of the method and the quality of its ranking.

The paper is organised as follows. In Section 2, we give some notation and we formally introduce the cost-sensitive diagnosis problem. Section 3 deals with an estimation of the rule error probability. Section 4 is mainly devoted to the illustration of the Cost-Sensitive diagnosis framework. Scalability and applicability of the approach are discussed in Section 5. Section 6 concludes the paper.

2 Notation and Problem Statement

In this work, we will concentrate on declarative languages, although most of the ideas introduced are valid for other rule-based languages. First, let us briefly introduce some notation and next we will state the problem more formally.

2.1 Notation

Throughout this paper, V will denote a countably infinite set of variables and Σ is a *signature* denoting a set of function symbols, each of which has a fixed associated arity. $\mathcal{T}(\Sigma \cup V)$ and $\mathcal{T}(\Sigma)$ denote the non-ground term algebra and the term algebra built on $\Sigma \cup V$ and Σ, respectively. By $Var(t)$ we intend the set of variables occurring in term t. Given an equation e, we will denote the left-hand side (resp. the right-hand side) of e by $lhs(e)$ (resp. by $rhs(e)$).

A *conditional term rewriting system* (CTRS for short) is a pair (Σ, R), where R is a finite set of rewrite rule schemes of the form $(\lambda \to \rho \Leftarrow C)$, λ, $\rho \in \mathcal{T}(\Sigma \cup V)$, $\lambda \notin V$ and $Var(\rho) \subseteq Var(\lambda)$. The condition C is a (possibly empty) sequence $e_1, \ldots, e_n$, $n \geq 0$, of equations.

We will often write just R instead of (Σ, R). Given a CTRS (Σ, R), we assume that the signature Σ is partitioned into two disjoint sets $\Sigma = \mathcal{C} \uplus \mathcal{F}$, where $\mathcal{F} = \{f \mid (f(t_1, \ldots, t_n) \to r \Leftarrow C) \in R\}$ and $\mathcal{C} = \Sigma \setminus \mathcal{F}$. Symbols in $\mathcal{C}$ are called *constructors* and symbols in $\mathcal{F}$ are called *defined functions*. Elements in $\mathcal{T}(\mathcal{C})$ are called *values* (or ground constructor terms). In the remainder of this paper a *program* is a CTRS. A term t is a *normal form* w.r.t. a program R, if there is no term s such that $t \to_R s$, where $\to_R$ denotes the usual conditional rewrite relation [6]. Let $nf_R(t)$ denote the set of normal forms of the term t w.r.t. R and $cnf_R(t) = \{s | s \in nf_R(t) \text{ and } s \in \mathcal{T}(\mathcal{C})\}$. Moreover, let $nf_{R,r}(t) = \{s | s \in nf_R(t) \text{ and } r \text{ occurs in } t \to_R^* s\}$ and $cnf_{R,r}(t) = \{s | s \in nf_{R,r}(t) \text{ and } s \in \mathcal{T}(\mathcal{C})\}$. A rule $(\lambda \to \rho \Leftarrow t_1 = t_1', \ldots, t_n = t_n')$ is *decreasing* if, for each substitution σ, terms $\sigma(t_i)$, $\sigma(t_i')$, $i = 1, \ldots, n$, are smaller than $\sigma(\lambda)$ w.r.t. a termination ordering [5]. A CTRS R is called *decreasing*, whenever all the rules in R are decreasing.

A CTRS R is *confluent* if, for each term s such that $s \to_R^* t_1$ and $s \to_R^* t_2$, there exists a term u such that $t_1 \to_R^* u$ and $t_2 \to_R^* u$. If R is confluent then the normal form of a term t is unique. A *sample* S is a set of examples of the form $\langle e, n \rangle$ where e is a ground equation and $n \in I\!N$ is a unique identifier. For the sake of simplicity, we will often denote the example $\langle e, n \rangle$ by e_n or e.

2.2 A Cost-Sensitive Debugging Schema

A cost-sensitive debugging problem is defined as follows:

- a program R to be debugged;
- one main function definition f, selected from R;
- a correctness reference, either an intensional specification I or a sample S, or an oracle O;
- a cost function defined from $\mathcal{T}(\mathcal{C}) \times \mathcal{T}(\mathcal{C})$ to the set of real numbers;
- a probability distribution function p of ground terms that have f as outermost function symbol.

Let us define more precisely some of the previous components. First of all, the program R is assumed to be decreasing in order to ensure effectiveness of our debugging framework. However, R could be non-confluent. In that case, we assume the following worst-case error situation: if one of all the normal forms computed by R is not correct w.r.t. the correctness reference, we consider program R incorrect.

Secondly, f is the main function definition to which the cost measures and errors refer to. In the case that more than one function definition f must be taken into account, we could debug them separately.

Thirdly, our cost-sensitive diagnosis framework exploits a correctness reference representation based on a sample S, which could be generated according to distribution p. Example generation could be done by means of a generative grammar method, which is able to automatically yield a set of ground equations satisfying a given probability distribution requirement, as we will discuss in section 5. However, correctness reference can also be represented by the following alternative forms: an intensional program I (a specification, which at least defines function f), an oracle O (an entity that can be queried about the correct normal form of any well-constructed term whose outermost symbol is f). Our framework can deal with all these three representations, since both I and O can be used to randomly generate a sample S, according to a given distribution p. Hereinfater, we will work with S.

Fourthly, we have a function $ECF_f : \mathcal{T}(\mathcal{C}) \times \mathcal{T}(\mathcal{C}) \to I\!R$, called the Error Cost Function, such that the first argument of this function is intended to be one of the normal forms computed by the program R for a term $f(t_1, \ldots, t_n)$ and the second one is intended to be the correct normal form. In the particular case that the set of computed values is a set of constants (nominal type), the Error Cost Function can be expressed as a bidimensional matrix, which makes its understanding easier, as seen in the introduction.

As we will see, in many cases ECF_f and p are not known. We will also discuss reasonable assumptions (symmetric cost function and universal distribution) for these situations later.

2.3 Kinds of Error

First of all, we assume that our sample S fulfils the following properties: (i) for each $l = r \in S$, $r \in \mathcal{T}(\mathcal{C})$, (ii) there are not two equations e and e' such that $lhs(e) = lhs(e')$ and $rhs(e) \neq rhs(e')$. The reason why we require (i) concerns the fact that generally programmers are interested in computing values. So, we will rank rules w.r.t. values. Besides, (ii) is needed in order to force a good behaviour of our correctness reference: we do not want that a ground function call can calculate more than one value. In particular, (ii) implies a *confluent* intensional specification I.

For this work we will only consider correctness errors.

Definition 2.1 [Correctness Error] The program R has a *correctness error* on example e, denoted by $\Box_e$, when there exists a term $t \in cnf_R(lhs(e))$ such that $t \neq rhs(e)$.

For simplicity, the error function ECF_f is defined only for values (ground constructor terms) plus the additional element $\perp$ representing all the non-constructor normal forms. More precisely, the first argument of ECF_f has to be a value or $\perp$, while the second argument is constrained to be a value.

3 Estimating the Error Probability

Consider a program R that is not correct w.r.t. some specification. In some cases there is clearly one rule that can be modified to give a correct program. In other cases, a program cannot be corrected by changing only one rule and the possibilities of correction (and hence the number of rules that might be altered) become huge.

Consequently, we are interested in a rating of rules that assigns higher values to rules that are more likely to be guilty of program failure. More precisely, we want to derive the probability that given an error, the cause is a particular rule. If we enumerate the rules of a program from r_1 to r_n and, as we defined in Section 2.3, we denote by $\Box$ a mismatch between the normal form computed by R and the correct normal form (correctness error), then we would like to obtain the probability that given a correctness error, a particular rule is involved, i.e. $P(r_i|\Box)$.

This probability could be computed directly or, as will be shown later, it could be obtained by the help of Bayes theorem, i.e.:

$$P(r_i|\Box) = \frac{P(r_i) \cdot P(\Box|r_i)}{P(\Box)}.$$

So, we only have to obtain the following probabilities: $P(r_i)$, $P(\Box|r_i)$ and

$P(\Box)$. In the sequel, let $card(L)$ be the cardinality of a sample L. The probability of error $P(\Box)$ is easy to be estimated by using the sample S. Let us denote by $E \subseteq S$ the sample which gives rise to a correctness error using R. Formally, $E = \{e \in S \mid \exists\, t \in cnf_R(lhs(e)) \text{ and } t \neq rhs(e)\}$. As we said, if R is not confluent it is sufficient that one normal form is different from the correct value to consider that to be an error. Consequently:

$$P(\Box) \approx \frac{card(E)}{card(S)}$$

i.e., the number of cases that are wrongly covered by the program w.r.t. the size of the sample.

The probability that a rule is used, $P(r_i)$, is also easy to be estimated. Let us denote by $U_i \subseteq S$ the following sample: $U_i = \{e \in S \mid \exists\, t \in cnf_{R,r_i}(lhs(e))\}$. If an example uses more than once rule r_i, r_i is only reckoned once. Then we have:

$$P(r_i) \approx \frac{card(U_i)}{card(S)}$$

Finally, we have to approximate the probability that there is an error whenever a rule r_i is used, that is, $P(\Box|r_i)$. Let us denote by $E_i \subseteq U_i$ the sample from U_i rising a correctness error w.r.t. R. Then,

$$P(\Box|r_i) \approx \frac{card(E_i)}{card(U_i)}.$$

If we take a look at the previous formulae, it is straightforward noting that:

$$P(r_i|\Box) \approx \frac{card(E_i)}{card(E)}$$

which is consistent with the definition of $P(r_i|\Box)$ as "probability that given an error, the cause is a particular rule". Moreover, since $card(E)$ is the same for all the rules, we would have that $P(r_i|\Box)$ only depends on $card(E_i)$.

However, consider the following example:

$$\begin{aligned}
&r_1 : \texttt{f}(\texttt{X}, []) \rightarrow \texttt{0}\\
&r_2 : \texttt{f}(\texttt{true}, [\texttt{A}|\texttt{B}]) \rightarrow \texttt{sumlist}([\texttt{A}|\texttt{B}])\\
&r_3 : \texttt{f}(\texttt{false}, [\texttt{A}|\texttt{B}]) \rightarrow \texttt{sizelist}([\texttt{A}|\texttt{B}])\\
&r_4 : \texttt{sumlist}([\texttt{A}|\texttt{B}]) \rightarrow \texttt{A} + \texttt{sumlist}[\texttt{B}]\\
&r_5 : \texttt{sumlist}([]) \rightarrow \texttt{0}\\
&r_6 : \texttt{sizelist}([\texttt{A}|\texttt{B}]) \rightarrow \texttt{0} + \texttt{sizelist}[\texttt{B}]\\
&r_7 : \texttt{sizelist}([]) \rightarrow \texttt{0}
\end{aligned}$$

and $S = \{\langle f(true, [3,2,5]) = 10, 1\rangle, \langle f(false, []) = 0, 2\rangle, \langle f(false, [2]) = 1, 3\rangle\}$. With this we would have that $P(r_6|\Box) = 1$. This is so easily detected because f is implemented in a "divide-and-conquer" fashion, and examples are distributed in such a way that even auxiliary functions not given in the specification can be debugged. However, this may not be the case for programs where the recursive clauses are deeply interrelated (we would also need the definition of the auxiliary functions in these cases).

When there are few examples, it could happen that a faulty rule never rises a correctness error, because it might not appear in any example rewrite sequence. Therefore it may not be detected as incorrect. In this situation, the use of some smoothing techniques, such as Laplace smoothing or the m-estimate, for correcting error rule probabilities could be helpful.

3.1 Smoothing

Whenever a rule has not been used to derive any example, its probability of error is 0. Due to this lack of information, we are not able to evaluate whether that rule is faulty or not. The reason is that the previous probability estimates are based solely on relative frequencies. The usual solution to this problem is to employ a frequency correction method called *smoothing*.

Laplace correction is the simplest form of smoothing of relative frequencies and is defined as:

$$P' \approx \frac{N_f + 1}{N_T + M}$$

where N_f is the number of favourable cases, N_T is the number of total cases and M is the number of possibilities. Note that if we have not seen any case, $P' = 1/M$, which is quite reasonable.

The probability of error $P(\Box)$ does not need smoothing, because it is the same for all the rules, and we can assume it is different from zero (i.e., there is at least an error). But $P(r_i)$ and $P(\Box|r_i)$ can be redefined as follows.

First, the probability $P(r_i)$, that a rule is used to prove some examples, becomes

$$P'(r_i) \approx \frac{card(U_i) + 1}{card(S) + card(R)}$$

where $card(R)$ represents the number of rules of program R. Note that, without any information, the probability that a rule is used would be $1/card(R)$.

Finally, the probability that there is an error when a rule is used, $P(\Box|r_i)$ is smoothed considering the fact that a rule may be correct or faulty. So, we obtain

$$P'(\Box|r_i) \approx \frac{card(E_i) + 1}{card(U_i) + 2}$$

The new values computed so far allow to provide a smoothed version of $P(r_i|\Box)$ as follows:

$$P'(r_i|\Box) = \frac{P'(r_i) \cdot P'(\Box|r_i)}{P(\Box)}$$

and, for a sufficient sample size, we might also consider not to smooth $P(\Box|r_i)$.

$$P''(r_i|\Box) = \frac{P'(r_i) \cdot P(\Box|r_i)}{P(\Box)}$$

Now consider the following example.

lhs	rhs_S	rhs_R	r_1	r_2
$even(0)$	$true$	$true$	1	0
$even(s(0))$	$false$	$true$	1	1
$even(s(s(0)))$	$true$	$true$	1	1
$even(s(s(s(0))))$	$false$	$true$	1	1
$P(r_i)$	–	–	4/4	3/4
$P(\Box \vert r_i)$	–	–	2/4	2/3
$P'(r_i)$	–	–	5/6	4/6
$P'(\Box \vert r_i)$	–	–	3/6	3/5
$P(\Box)$	–	–	1/2	1/2
$P(r_i \vert \Box)$	–	–	1	1
$P'(r_i \vert \Box)$	–	–	5/6	4/5
$P''(r_i \vert \Box)$	–	–	5/6	8/9

Table 1
Smoothing values for Example 3.1.

Example 3.1 Let R be te following wrong program

$$r_1 : \texttt{even(0)} \rightarrow \texttt{true}$$

$$r_2 : \texttt{even(s(X))} \rightarrow \texttt{even(X)}$$

and S be the sample

$$S = \{ \langle even(0) = true, 1\rangle, \langle even(s(0)) = false, 1\rangle, \langle even(s(s(0))) = true, 1\rangle,$$
$$\langle even(s(s(s(0)))) = false, 1\rangle\}.$$

Then, we get $P(r_1|\Box) = 1$ and $P(r_2|\Box) = 1$. Now, let us smooth our probabilities.

By looking at Table 1, $P''(r_i|\Box)$ gives a good result but just contrary to $P'(r_i|\Box)$. This shows that smoothing can alter the orderings and should be employed carefully, since could introduce rough probability approximations.

3.2 Advantages and Caveats of using $P(r_i|\Box)$

Let us study the kinds of problems where the previous approach works. In some cases the rating is clearly intuitive, for instance, when we have non-recursive function definitions with a fine granularity of the rules or recursive definitions where the error is in the base case. Consider, e.g., the following wrong program for the function *even*:

$$r_1 : \texttt{even(0)} \rightarrow \texttt{false}$$

$$r_2 : \texttt{even(s(s(X)))} \rightarrow \texttt{even(X)}$$

Here r_1 is clearly given the highest $P(r_i|\Box)$ whenever the sample contains "even(0) = true". r_2 must necessarily be lower.

However, there are some cases where there is a tie between the error probabilities of two or more rules. One can imagine to use $P(\Box|r_i)$ in order to untie, but this does not always work properly. For instance, let us consider the following example that computes the product of two natural numbers by the use of an auxiliary function *multaux*:

$$r1 : \texttt{mult(0,y)} \rightarrow \texttt{0}$$

$$r2 : \texttt{mult(s(X),Y)} \rightarrow \texttt{s(multaux(X,Y,Y))}$$

$$r3 : \texttt{multaux(X,Y,0)} \rightarrow \texttt{mult(X,Y)}$$

$$r4 : \texttt{multaux(X,Y,s(Z))} \rightarrow \texttt{s(multaux(X,Y,Z))}$$

In this case, if the sample S just contains examples for *mult* but not for *multaux* (probably because we have a specification of *mult* using *add*, as usual) then we will have $P(\Box|r_2) = P(\Box|r_3) = P(\Box|r_4) = 1$. However, $P(r_2|\Box)$ $= P(r_3|\Box) = 1$. We cannot in general distinguish between r_2 and r_3, because whenever r_2 is used r_3 is also used and vice versa. If we had examples for *multaux*, we would have that $P(\Box|r_2) = 1 > P(\Box|r_3) = P(\Box|r_4)$, clarifying that since *multaux* seems to be correct, the error can only be found in r_2.

The following result about incorrectness detection can be proven.

Theorem 3.2 *Let S be a sample and R be a program not rising correctness error w.r.t. S. Consider a program R' which differs from R just by one rule r_i. Then, $\exists\, e' \in S$ such that $cnf_R(lhs(e')) \subset cnf_{R'}(lhs(e'))$. Then, the diagnosis method assigns the greatest $P(r_i|\Box)$ to rule r_i.*

Proof. Since there is (at least) $e' \in S$ such that $cnf_R(lhs(e')) \subset cnf_{R'}(lhs(e'))$, we have that there exits (at least) a term $t \in cnf_{R'}(lhs(e'))$ and $t \notin cnf_R(lhs(e'))$. Since R is correct w.r.t. S, then $rhs(e') \in cnf_R(lhs(e'))$ and $t \neq rhs(e')$. Thus, R' rises a correctness error on example e'. Clearly, there may be more than one $e' \in S$ on which R' gives rise to a correctness error. So, let $E \subseteq S$ be the sample of such examples. Since R' only differs from R by one rule r_i and R does not give rise to any correctness error, we have that $E_i = E$. Hence,

$$P(r_i|\Box) = \frac{card(E_i)}{card(E)} = 1.$$

Consequently, any other rule of R' must have less or equal probability. And this proves the claim.

□

Finally, the previous measure detects faulty rules and it is able to *rank* them. However, even though the distribution of examples is taken into account (the probabilities are estimated from this distribution), sometimes we obtain rule rankings containing ties. Therefore we are sometimes not allowed to localise error sources with a sufficient precision degree. In the next section, we will try to refine the ranking process by considering rule costs.

4 A Cost-Sensitive Diagnosis

In this section, we define a cost-sensitive diagnosis for functional logic programs. For this purpose we require a cost function $ECF_f : \mathcal{T}(\mathcal{C}_\perp) \times \mathcal{T}(\mathcal{C}) \to I\!R$ defined for the outputs of function f.

By using this function we could compute the cost per example assigned to each rule, i.e. which proportion of the overall cost of the program is blamed to each rule.

Definition 4.1 [Rule Correctness Cost] The *rule correctness cost*, denoted by $Cost_{r_i}$ is computed in the following way:

$$Cost_{r_i} = \frac{\sum_{e \in S} Cost_{r_i}(e)}{card(S)}$$

where $Cost_{r_i}(e)$ is defined as follows:

$$Cost_{r_i}(e) = \sum_{t \in cnf_{R,r_i}(lhs(e))} ECF_f(t, rhs(e))$$

Example 4.2 Consider the following program R and sample S:

```
r1 : even(0) → true
r2 : even(s(X)) → even(X)
```

$$S = \{\ \langle even(0) = true, 1\rangle, \langle even(s(0)) = false, 2\rangle,$$
$$\langle even(s(s(0))) = true, 3\rangle, \langle even(s(s(s(0)))) = false, 4\rangle\}.$$

And consider the ECF_{even} defined by the following cost matrix.

		I	
		$false$	$true$
R	$false$	-1	1
	$true$	1	-1

Consequently, using this cost matrix, we have that r_2 is costlier than r_1.

However, there are two important questions to be answered here. What is the relation between the *Cost* measure and the error probability? And secondly, which cost matrix should be used in the case we are not given one? The first question is answered by the following proposition, which states that the rule correctness cost method is a "generalisation" of error probability method.

Proposition 4.3 *Let R be a confluent CTRS, r_i be a rule belonging to R and S be a sample. Consider the cost function $ECF_f(X,Y) = \kappa$ if $X \neq Y$ and 0 otherwise, where $\kappa \in I\!R, \kappa > 0$. Then, $Cost_{r_i} = \kappa \cdot P(r_i|\Box) \cdot P(\Box)$.*

Proof. Let $e \in S$. Consider the rule correctness cost per example $Cost_{r_i}(e)$ which is defined as

$$(1) \qquad Cost_{r_i}(e) = \sum_{t \in cnf_{R,r_i}(lhs(e))} ECF_f(t, rhs(e)).$$

Since R is confluent, the normal form of a given term (whenever it exists) is unique. So, Equation 1 becomes

$$(2) \quad Cost_{r_i}(e) = ECF_f(t, rhs(e)) = \begin{cases} \kappa \text{ if } t \neq rhs(e), t \in \mathcal{T}(\mathcal{C}), \kappa \in I\!R, \kappa > 0 \\ 0 \text{ otherwise.} \end{cases}$$

where t is the unique normal form of $lhs(e)$. By summing each rule correctness cost per example, we get the following relation:

$$(3) \qquad card(E_i) = \frac{1}{\kappa} \sum_{e \in S} Cost_{r_i}(e)$$

As $E \subseteq S$, $card(S) = card(E) + c$, where $c \geq 0$ is a natural constant. By exploiting relations above and probability definitions given in Section 3, we finally get the desired result.

$$Cost_{r_i} = \frac{\sum_{e \in S} Cost_{r_i}(e)}{card(S)} = \kappa \cdot \frac{card(E_i)}{card(S)} = \kappa \cdot \frac{card(E_i)}{card(E)} \cdot \frac{card(E)}{card(S)} = \kappa \cdot P(r_i|\Box) \cdot P(\Box).$$

So, our claim is proven. $\Box$

The interesting thing (and the answer to the second question) comes when we use cost functions which do not subsume error probabilities.

As we have seen in the previous example, instead of the cost function that does not take into account hits (i.e. correct constructor normal forms), we prefer the symmetrical cost function $ECF_f(X,Y) = 1$ if $X \neq Y$ and -1 otherwise (we assign 0 when X is not a value). The rationale of this choice is based on the idea that not only errors have to be taken into account but also hits. Hence, a very useful part of a program may have participated in an error but should be less blamed for that than a less useful part of a program that has also participated in the error. According to all the machinery we set up, a by-default modus operandi could be as follows.

Modus Operandi

- First of all, discard all the incompleteness errors. The true equations corresponding to these errors (lhs is the normal form generated by R and rhs is the correct normal form) are output on a new set E_Δ.
- Compute the $p(r_i|\Box)$ for all the rules and give a first ranking of rules.
- In case of tie use $Cost_{r_i}$ with the symmetrical cost function to refine the ranking.

Let us see in the taxi assignment example of Section 1, whether this rule of thumb works in practice.

Example 4.4 Consider the following program.

$$
\begin{aligned}
&r_1 : \texttt{cap(car)} \rightarrow \texttt{4}\\
&r_2 : \texttt{cap(van)} \rightarrow \texttt{9}\\
&r_3 : \texttt{cap(minibus)} \rightarrow \texttt{12}\\
&r_4 : \texttt{interval(X,N,Z)} \rightarrow (\texttt{X} < \texttt{N} \leq \texttt{Z})\\
&r_5 : \texttt{taxi(N)} \rightarrow \texttt{car} \Leftarrow \texttt{interval(0,N,cap(car))}\\
&r_6 : \texttt{taxi(N)} \rightarrow \texttt{van} \Leftarrow \texttt{interval(cap(car),N,cap(van))}\\
&r_7 : \texttt{taxi(N)} \rightarrow \texttt{minibus} \Leftarrow \texttt{interval(cap(van),N,cap(minibus))}\\
&r_8 : \texttt{taxi(N)} \rightarrow \texttt{special} \Leftarrow \texttt{cap(minibus)} < \texttt{N}
\end{aligned}
$$

and the specification I shown in the introduction. Let us use the following sample:

$$
\begin{aligned}
S = \{ &\langle taxi(1) = car, 1\rangle, \langle taxi(1) = car, 2\rangle, \langle taxi(2) = car, 3\rangle,\\
&\langle taxi(3) = car, 4\rangle, \langle taxi(5) = van, 5\rangle, \langle taxi(7) = van, 6\rangle,\\
&\langle taxi(9) = minibus, 7\rangle, \langle taxi(11) = minibus, 8\rangle, \langle taxi(20) = special, 9\rangle\}.
\end{aligned}
$$

Note that repeated examples exist and they should be taken into account. By applying the probability estimation of Section 3 and the symmetrical cost function, the following probabilities and costs are computed.

	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8
$P(r_i\|\Box)$	1	1	0	1	0	1	0	0
$Cost_{r_i}$	$-\frac{5}{9}$	$-\frac{2}{9}$	$-\frac{5}{9}$	$-\frac{8}{9}$	$-\frac{4}{9}$	$-\frac{1}{9}$	$-\frac{1}{9}$	$-\frac{1}{9}$

We note there is a tie among the probabilities of rules r_1, r_2, r_4 and r_6. This rule ranking can be refined by taking into account rule correctness costs. Indeed, we have that r_6 is rated first and r_2 is rated second.

5 Generation of Examples, Scalability and Applications

In order to approximate in an appropriate way the ratings introduced before, we need a representative sample of examples. If we are given the sample S we use it directly. However, if we have a specification I or an oracle O then we need to obtain random examples from it. A first idea in these latter cases is to generate terms by using the fix-point of the immediate consequence operator until iteration k, i.e. ${T_R}^k$. However, the problem of this approach is that the number of examples cannot be told in advance with k and, more importantly, the complexity of the ${T_R}^k$ procedure highly depends on the size and complexity of the program. Consequently, it seems more reasonable to generate some set or multiset of terms (as *lhs* of examples) and use them to obtain (jointly with I or O) the *rhs* of the examples. If we are given a probability distribution over the terms or the examples, this is fairly clear, we only need to generate them with an appropriate generative grammar. However, which distribution should we assume if we are not given one? If the main function for which we want to generate examples is of type or sort $\mathcal{U}$, we have to determine a distribution on all the ground terms of this type. Since there may be infinite possible terms, a uniform distribution is not a good solution for this.

A way to obtain a representative sample with less assumptions but still feasible is the use of a universal distribution over the terms, which is defined as the distribution that gives more probability to short terms and less probability to large terms [7]. This is also beneficial for computing normal forms, since the size of the examples will not be, in general, too large, and for many programs, this would also mean relatively short rewriting chains.

Definition A.1. The Universal Distribution over a type or sort $\mathcal{U}$ is defined as:

$$P(s) = b^{-Inf(s)}, Type(s) = \mathcal{U}$$

where $Type(s)$ is the type of term s and $Inf(s)$ is the information needed to code s using a generative grammar for all the possible values of type $\mathcal{U}$. The base b must be a positive real number strictly greater than 1. Usually $b = 2$.

A generative grammar is any context free or context sensitive grammar with at least one possible rewriting alternative in all the rules. For generating functional terms we just need a context free grammar. The overall information needed to choose between the possible rewrites in this grammar is what is measured by $Inf(s)$. Let us define $Inf(s)$ in a more formal way.

Let us denote first with $\mathcal{F}_{\mathcal{U}}$ the possible function symbols of type $\mathcal{U}$. Let us denote with $head(s)$ the outermost function symbol of term s and $s|_i$, $i = 1..n$, the n arguments of the outermost function symbol of s, with $n = arity(head(s))$.

If we want to build a *ground* term of type $\mathcal{U}$, i.e. an element of $\mathcal{T}(\Sigma_{\mathcal{U}})$, we have to tell from all the possible functors the one to choose for the outer position and then to generate terms for each argument of the function. According to this, we can define recursively the information of a term as follows:

Definition A.2. The information of a term s of type $\mathcal{U}$ is defined as:

$$Inf(s) = Inf_0(s) + Inf_1(s)$$
$$Inf_0(s) = log_2(card(\mathcal{F}_{Type(head(s))})) + log_2(head(s))$$
$$Inf_1(s) = \sum_{t=s|_n, n=1..arity(head(s))} Inf(t)$$

The previous definition takes into account the bits required to select from all the possible functors, the bits required to select/separate all the possible arguments and the bits to code all the arguments.

In order to obtain examples under this distribution, we just need to consider the types of the set of functions $\mathcal{F}$ and introduce the function symbols using a specific local distribution. As will be shown below we need a local distribution that assures the generation is finite. For the generation, we use empty positions, that will be subsequently filled with terms. In order to ensure termination of the generation process, after an empty position has been filled by a functor, the number of new open empty positions should be probabilistically lower than 1. For instance, it could be equal to 1/2.

Let us consider that for a term s of type $\mathcal{U}$ we have k different functors $f_i \in \mathcal{F}_{\mathcal{U}}$. We have to assign a local probability $p'(f_i)$ to each of these functors in such a way that the following equation holds:

$$\forall f_i \in \mathcal{F}_{\mathcal{U}} \sum_{i=1..k} p(f_i) \cdot arity(f_i) = 1/b$$

which ensures that the number of new open empty positions should be probabilistically equal to 1/b. Let us define $k' = card(\{arity(f_i) \neq 0\})$. The following assignment of probabilities is designed with this purpose. First we can define the probabilities for the functors with arity different from 0:

$$p'(f_i) = \frac{1}{b \cdot k' \cdot arity(f_i)} \quad \text{if } arity(f_i) \neq 0$$

and then the probability for the functors with arity equal to 0 (the remainder of probability uniformly distributed):

$$p'(f_i) = 1 - \frac{\sum_{f_i, arity(f_i)>0} p'(f_i)}{k - k'} \quad \text{if } arity(f_i) = 0$$

And from here, we can define the probability of a term as follows:

$$p(s) = p'(head(s)) \cdot \prod_{t=s|_n, n=1..arity(head(s))} p(t)$$

As a consequence we can state the following theorem:

Theorem 2. A probabilistic generative grammar using probability $p(f_i)$ ensures termination. More formally, it is a probability distribution for the terms of type $\mathcal{U}$, i.e.

$$\forall \mathcal{U} \sum_{t \in \mathcal{T}(\Sigma_{\mathcal{U}})} p(t) \leq 1 \quad (1)$$

Proof. The proof is based on the fact that probability of creating gaps is less than 1. Then,

$$\sum_{i=1..k} p'(f_i) \cdot arity(f_i) = \sum_{i=1..k'} \frac{1}{b \cdot k' \cdot arity(f_i)} arity(f_i) = \sum_{i=1..k'} \frac{1}{b \cdot k'} = 1/b$$

since $b > 1$. □

Finally, the universal distribution can be approximated by the following procedure to generate the terms. We denote an empty position (a gap still to be generated) by the symbol '_'.

```
function GenerateTermFromType($\mathcal{U}$) : s
Select one function $f$ from $\mathcal{F}_{\mathcal{U}}$ randomly according to probability $p(f_i)$.
Let $s = f(\_, \_, ..., \_)$
end function

function GenerateTerm(s) : s
for each still empty position $\pi$ in $s$ do
  $t$= GenerateTermFromType(Type($s|_\pi$))
  $s$= $s[t]_\pi$
end for
end function
```

Using these procedures, each *lhs* of an example is just obtained by using a call to "Generate($f(\_, \_, ..., \_)$)" where f is the main function. This procedure can be executed as many times as desired. There is a high probability that repeated terms may be generated. These can be simply removed or ignored, depending on whether we want a real distribution (e.g. when costs are taken into account) or just some detection (without ranking) of bugs.

Finally, the *rhs* of each term t is computed asking to the oracle or,if we have a specification I, just by computing $nf_I(t)$.

An interesting thing about the previous method of generating terms is that the base b on $p(f_i)$ can be modified in order to be less steep. Values closer (but still greater) to 1 will give a distribution more similar to a uniform distribution and large terms will be more likely to be generated. Values greater or equal than 2 will give a high probability to short terms.

Let us see a final example. Given the type $I\!N$, we want to generate terms for it according to the universal distribution. If we choose $b = 2$ we have that:

$$p(0) = 1/2$$

$$p(s^n(0)) =_{n>0} p(s) \cdot p(s^{n-1}(0)) = \frac{1}{2^{n+1}}$$

Note that half of the probability is given to term 0, since $p(0) = 1/2$. On the other hand, if we choose $b = 1.1$ we would have that:

$$p(0) = 1 - \frac{1}{1.1 \cdot 1 \cdot 1} = 0.091$$

which shows that this distribution is less steep and the probability of bigger terms is higher.

It may be interesting to discuss whether short examples are more likely to include extreme or strange values (as it is usually recommended by techniques such as "Extreme Value Analysis"). We believe that long examples are precisely those which are inside the bulk of "usual" examples, the word "usual" understood as not peculiar. Note that base cases, cases with extreme values, exceptional cases, etc. of a single function are precisely those which can be reached with a few rewriting steps in an overwhelming majority of applications (of course you can make up a very special program that can only have a special behaviour after a thousand calls to the same function, but this is not a good practice or it is not a very common example). In this way we would have that for many programs, if we generate a sufficiently large sample, almost all the small peculiar cases would be considered.

Regarding scalability, the generation of a sample of n terms according to the universal distribution can be considered in $\mathcal{O}(n)$. It is the computation of the *rhs* of each of these terms that may be the most computationally expensive part. As we have said, in many situations, short terms will have (in general) shorter derivations than large terms, and this will affect positively on the possibility of generating large amounts of examples for many applications. It is reasonable to think that larger programs will require more examples, although it is problem-dependent to know if the relation is sublinear, linear or exponential. The good thing of our method is that we can begin to estimate probabilities in an incremental way and we can stop when the cardinalities of each of the rule probabilities and costs are large enough to have good approximations. In other words, for each particular program, we can tune a specific compromise between time complexity and precision of the estimation of probabilities and costs.

With respect to applicability, the first area where our approach is especially beneficial is in those cases where we have a relevant cost function or matrix. For instance, diagnosis programs (medical diagnosis, failure detection, etc.) are clear examples where our approach can take advantage of the cost information. Just as an example, it is more important to detect an error that alerts that a system is not working (when it is) than the reverse situation. A second area of application is the development of programs for which we are given many use cases as a specification or we can generate many of these. There is no need of a full specification in these cases. A third area of application is prototyping, when we want to detect the most evident errors first in order to show some basic functionality. Additionally to these three specific areas, we think that our approach could also be beneficial in general, depending possibly of the kind of program to be debugged.

Finally, an alternative way of generating examples could be to use the universal distribution for the *derivations*. This would be similar to assigning probabilities to a definitional tree. Hence, terms would be more probable the shorter their program derivation. Nonetheless, we think that this approach would be less efficient (we have to consider both the *lhs* and the *rhs* wrt. the program) and more difficult to implement than the previous approach.

6 Conclusions

In this paper, we have shown that the analysis of rule use depending on the distribution of examples can also be used for detecting bugs and, more importantly, to priorise them. Developing a debugging ranking of rules must also take into account the context and distribution where the program is being used. Some cases handled by a program are much more frequent than others. This means that errors in frequent cases are corrected first than errors in less frequent cases or even cases that almost never occur. This information can be given by a probability distribution and hence used by our diagnosis framework when constructing the sample (instead of the general, by-default universal distribution). Moreover, this information can be complemented with the cost function, because it may also be the case that rare situations may have more cost than common situations.

These considerations are, to the best of our knowledge, quite new in the typical discourse of program debugging, but are well-known in what is called cost-sensitive learning. Other techniques, such as ROC analysis [4] could also be applied for diagnosis.

The efficiency of our method is proportional to the number of examples used. The bigger the sample the more precise the estimation of probabilities and costs but the efficiency decreases (linearly).

Another advantage of this framework, inherited from its simplicity, is that it can also be applied to other declarative and even non-declarative rule-based languages, provided the rules have enough granularity.

One way to advance in some complex cases (especially recursive ones) could be to explore rule dependencies. Another issue to be explored is that the probabilities and costs could be computed exactly and not approximated, when we are given the specification. This would turn this method into a purely semantical one instead of a statistical one. As future work, we also plan to extend our cost analysis in order to deal with completeness errors and nonterminating programs.

Finally, in this work we have only considered detection and ranking of bugs, but not their correction. In many cases, inductive techniques (i.e. learning) should be used for this. Systems that are able to work with cost functions would be more coherent and could be coupled with this approach.

Acknowledgements

We would like to thank the WFLP'03 anonymous referees for their helpful comments which have helped to significantly improve this paper.

References

[1] M. Alpuente, F. J. Correa, and M. Falaschi. Debugging Scheme of Functional Logic Programs. In M. Hanus, editor, *Proc. of International Workshop on Functional and (Constraint) Logic Programming, WFLP'01*, volume 64 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2002.

[2] R. Caballero-Roldán, F.J. López-Fraguas, and M. Rodríquez Artalejo. Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. In *Fifth International Symposium on Functional and Logic Programming*, volume 2024 of *LNCS*, pages 170–184. Springer-Verlag, 2001.

[3] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.

[4] J.A. Hanley and B.J. McNeil. The meaning and use of the area under a receiver operating characteristic (roc) curve. *Radiology, 143*, pages 29–36, 1982.

[5] S. Kaplan. Simplifying Conditional Term Rewriting Systems: Unification, Termination and Confluence. *Journal of Symbolic Computation*, 4:295–334, 1987.

[6] J.W. Klop. Term Rewriting Systems. In *Handbook of Logic in Computer Science*, volume I, pages 1–112. Oxford University Press, 1992.

[7] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and its Applications.* 2nd Ed. Springer-Verlag, 1997.

[8] H. Nilsson and P. Fritzson. Algorithmic debugging for lazy functional languages. *Journal of Functional Programming*, 4(1):337–370, July 1994.