*Electronic Notes in Theoretical Computer Science 57 (2001)*

*URL:* [*http://www.elsevier.nl/locate/entcs/volume57.html*](http://www.elsevier.nl/locate/entcs/volume57.html) *24 pages*

*Declarative Debugging of Functional Logic Programs* [*1*](#_bookmark0)

# *M. Alpuente* [*2*](#_bookmark0)

*Departamento de Sistemas Inform aticos y Computacion-DSIC Technical University of Valencia*

*Camino de Vera s/n, 46022 Valencia, Spain.*

# *F. J. Correa* [*3*](#_bookmark0)

*Departamento de Inform atica y Sistemas{DIS*

*University EAFIT, Carrera 49 7 Sur 50, 3300 Medell n, Colombia.*

# *M. Falaschi* [*4*](#_bookmark0)

*Dipartimento di Matematica e Informatica University of Udine, Via delle Scienze 206, 33100 Udine, Italy.*

*Abstract*

*We present a general framework for the declarative debugging of functional logic programs, which is valid both for eager as well as lazy programs. We associate to our programs a semantics based on a (continuous) immediate consequence operator which models computed answers. Then we show that, given the intended speci ca- tion of a program P , it is possible to check the correctness of P by a single step of the immediate consequence operator.*

*We also present a more e ective methodology which is based on abstract inter- pretation. By approximating the intended speci cation of the success set we derive a nitely terminating debugging method, which can be used statically. Our fra- mework is parametric w.r.t. to the chosen approximation of the success set. We present one speci c example of approximation. We provide an implementation of our debugging system which shows experimentally on a wide set of benchmarks that we are able to nd some common errors in the user programs.*

*c 2001 Published by Elsevier Science B. V. Open access under* [*CC BY-NC-ND license.*](http://creativecommons.org/licenses/by-nc-nd/3.0/)

# *1 Introduction*

*Declarative programming is supported both by functional and logic program-* ming. However, each of these programming styles has di erent advantages

*w.r.t. practical applications. Functional languages provide sophisticated ab-* straction facilities, module systems and clean solutions for integrating I/O into declarative programming as well as for eÆcient program execution. Logic languages allow for computing with partial information and provide built-in search facilities which have strong applications for knowledge-based systems and operations research. However, recent results show that the advantages of these styles can be eÆciently and usefully combined into a single language. Modern functional logic languages o er features from both styles. The ope- rational semantics of integrated languages is usually based on narrowing , a combination of uni cation for parameter passing and reduction as evaluation mechanism which subsumes rewriting and SLD-resolution. Essentially, nar- rowing consists of the instantiation of goal variables, followed by a reduction step on the instantiated goal. Narrowing is complete in the sense of functio- nal programming (computation of normal forms) as well as logic programming (computation of answers). Due to the huge search space of unrestricted narro- wing, steadily improved strategies have been proposed (see [[29]](#_bookmark42) for a survey.) How to debug functional logic programs is an important practical problem which has hardly been addressed in the previous literature. Only a few func- tional logic languages are equipped with a debugging tool (e.g. ALF [[28],](#_bookmark41) Babel [[39]](#_bookmark52) and Curry [[31](#_bookmark44)]). However, these debuggers consist of tracers which are based on suitable extended box models which help display the execution [[30](#_bookmark43)[,5].](#_bookmark18) Due to the complexity of the operational semantics of (functional) lo- gic programs, the information obtained by tracing the execution is diÆcult to understand. The functional logic programming language NUE-Prolog is en- dowed with a declarative debugger [[40](#_bookmark53)] which works in the style proposed by Shapiro [[43],](#_bookmark56) that is, an oracle (typically the user) is supposed to endow the debugger with error symptoms, as well as to correctly answer oracle questions driven by proof trees aimed at locating the actual source of errors. A similar declarative debugger for the functional logic language Escher is proposed in [[36](#_bookmark48)]. Following the generic scheme which is based on proof trees of [[41](#_bookmark54)], a pro- cedure for the declarative debugging of wrong answers is given in [[12]](#_bookmark25) for lazy functional logic languages. The methodology in [[12](#_bookmark25)] includes a formalization of computation trees which is precise enough to prove the logical correctness

*for the debugger and which also helps simplify oracle questions.*

*In the case of pure logic programming,* [*[17]*](#_bookmark30) *has de ned a declarative fra-*

*1 This work has been partially supported by CICYT under grant TIC2001-2705-C03-01, by Accion Integrada Hispano-Italiana HI2000-0161 and by Generalitat Valenciana under grant GV01-424.*

*2 Email:* [*alpuente@dsic.upv.es*](mailto:alpuente@dsic.upv.es)

*3 Email:* [*fcorrea@eafit.edu.co*](mailto:fcorrea@eafit.edu.co)

*4 Email:* [*falaschi@dimi.uniud.it*](mailto:falaschi@dimi.uniud.it)

*mework for debugging which extends the methodology in* [*[25,43]*](#_bookmark56) *to diagnosis*

*w.r.t. computed answers. The framework does not require the determina-* tion the symptoms in advance and is goal independent {it is driven by a set of most general atomic goals. It is based on using the immediate conse- quences operator TP to identify program bugs and has the advantage of giving a symptom{independent diagnosis method [[17,16].](#_bookmark29)

*In this paper, one of the contributions is to develop a declarative diagnosis* method w.r.t. computed answers which generalizes the ideas of [[17]](#_bookmark30) to the diagnosis of functional logic programs. The conditions which we impose on the programs which we consider allow us to de ne a framework for declarative debugging which works for both eager (call{by{value) narrowing as well as for lazy (call{by{name) narrowing. We associate a (continuous) immediate consequence operator to our programs. Then we show that, given the intended speci cation I of a program R, we can check the correctness of R by a single step of this operator. We illustrate this through examples. We discuss the use of our work for both bottom-up as well as top-down abstract debugging of mixed functional logic code.

*We also present a novel, eÆcient methodology which is based on abstract* interpretation. We proceed by approximating the intended speci cation of the success set. Following an idea inspired in [[17](#_bookmark30)[,16,11],](#_bookmark24) we use over and under speci cations I+ and I to correctly over- (resp. under-) approximate the intended semantics. We then use these two sets respectively for the functions in the premises and the consequence of the immediate consequence operator, and by a simple static test we can determine whether some of the clauses are wrong.

*1.1 Plan of the paper*

*The rest of the paper is organized as follows. Section* [*2*](#_bookmark1) *brie y presents some* preliminary de nitions and notations. Section [3](#_bookmark4) rst formulates a novel, gene- ric immediate consequence operator T ' for functional logic program R which is parametric w.r.t. the narrowing strategy ' which can be either eager or lazy. We then de ne a xpoint semantics based on T ' which correctly models the answers computed by a narrower which uses the narrowing strategy '. In the case of the eager strategy, it is enough to introduce a attening transforma- tion which eliminates nesting calls and allows goals to run in the semantics by standard uni cation. However, the lazy strategy is more involved and we need to introduce two kinds of equality in the de nition of T ': the strict equality which models the equality on data terms, and the non{strict = which holds even if the arguments are both unde ned or partially de ned, similarly to [[27].](#_bookmark40) We also formulate a semantics O'(R) and we show the correspondence with the xpoint semantics. In section [4,](#_bookmark7) we introduce the necessary general noti- ons of incorrectness and insuÆciency symptoms and uncovered calls. Section

*R*

*R*

*R*

[*5*](#_bookmark8) *provides an abstract semantics which correctly approximates the xpoint*

*semantics of R. In Section* [*6,*](#_bookmark11) *we present our method of abstract diagnosis* and illustrate its use through examples. In Section [7](#_bookmark12), we present an experi- mental evaluation of the method on a set of benchmarks. Section [8](#_bookmark13) concludes and discusses some related work.

# *2 Preliminaries*

*We brie y summarize some known results about rewrite systems* [*[6,34]*](#_bookmark47) *and* functional logic programming (see [[29,33](#_bookmark46)] for extensive surveys). Throughout this paper, V will denote a countably in nite set of variables and denotes a set of function symbols, or signature, each of which has a xed associated arity. ( [ V ) and ( ) denote the non-ground word (or term) algebra and the word algebra built on [ V and , respectively. ( ) is usually called the Herbrand universe (H) over and it will be denoted by H. B denotes the Herbrand base, namely the set of all ground equations which can be built with the elements of H. A -equation s = t is a pair of terms s; t 2 ( [ V ). Terms are viewed as labelled trees in the usual way. Positions are repre- sented by sequences of natural numbers denoting an access path in a term,

*where denotes the empty sequence.*

*O (t) denotes the set of nonvariable*

*positions of a term t. tju is the subterm at the position u of t. t[r]u is the* term t with the subterm at the position u replaced with r. These notions extend to sequences of equations in a natural way. For instance, the nonva- riable position set of a sequence of equations g (e1;::: ; en) can be de ned as follows: O(g) = fi:u j u 2 O(ei);i = 1;::: ; ng. By V ar(s) we denote the set of variables occurring in the syntactic object s, while [s] denotes the set of ground instances of s. A fresh variable is a variable that appears nowhere else. The symbol denotes a nite sequence of symbols. Identity of syntactic objects is denoted by .

*e*

*Let Eqn denote the set of possibly existentially quanti ed nite sets of* equations over terms [[14].](#_bookmark27) We write E E0 if E0 logically implies E. Thus Eqn is a lattice ordered by with bottom element true and top element f ail. The elements of Eqn are regarded as (quanti ed) conjunctions of equations and treated modulo logical equivalence. An equation set is solved if it is either f ail or it has the form 9y1 ::: 9ym: fx1 = t1;::: ; xn = tng, where each xi is a distinct variable not occurring in any of the terms ti and each yi occurs in some tj. Any set of equations E can be transformed into an equivalent one, solve(E), which is solved. We restrict our interest to the set of idempotent substitutions over ( [ V ), which is denoted by Sub. The empty substitution is denoted by . There is a natural isomorphism between substitutions = fx1=t1;::: ; xn=tng and unquanti ed equation sets = fx1 = t1;::: ; xn = tng. A substitution = fx1=t1;::: ; xn=tng is a uni er of an

*b*

*equation set E i b) E. We let mgu(E) denote the most general uni er of*

*the unquanti ed equation set E. We write mgu(fs1 = t1;::: ; sn = tng; fs0 =*

*1*

*t0 ;::: ; s0 = t0 g) to denote the most general uni er of the set of equations*

*1 n n*

*fs1 = s0 ; t1 = t0 ;::: ; sn = s0 ; tn = t0 g. We write j s to denote the restriction*

*1 1 n n*

*of the substitution to the set of variables in the syntactic object s.*

*A conditional term rewriting system (CTRS for short) is a pair ( ; R),* where R is a nite set of reduction (or rewrite) rule schemes of the form ( ! ( C), , 2 ( [ V ), 62 V and V ar( ) V ar( ). The condition C is a (possibly empty) sequence e1;::: ; en, n 0, of equations which we handle as a set (conjunction) when we nd it convenient. Variables in C that do not occur in are called extra-variables. We will often write just R instead of ( ; R). If a rewrite rule has no condition, we write ! . For CTRS R, r << R denotes that r is a new variant of a rule in R such that r contains only fresh variables, i.e. contains no variable previously met during computation (standardized apart). Given a CTRS h ; Ri, we assume

*that the signature is partitioned into two disjoint sets = C ] D, where* D = ff j (f (t~) ! r ( C) 2 Rg and C = n D. Symbols in C are called

*constructors and symbols in D are called de ned functions.*

*A rewrite step is the application of a rewrite rule to an expression. A* term s conditionally rewrites to a term t, s !R t, if there exist u 2 O(s), ( ! ( s1 = t1;::: ; sn = tn) 2 R, and substitution such that sju ,

*t s[ ]u, and for all i 2 f1;::: ; ng there exists a term wi such that si ! wi*

*R*

*and ti !*

*R*

*wi, where !*

*is the transitive and re exive closure of !R. When*

*no confusion can arise, we omit the subscript R. A term s is a normal form,* if there is no term t with s !R t. The program R is said to be canonical if the binary one-step rewriting relation !R de ned by R is noetherian and con uent [[34](#_bookmark47)].

*R*

*An equational Horn theory E consists of a nite set of equational Horn* clauses of the form ( = ) ( C. An equational goal is a sequence of equations ( C, i.e. an equational Horn clause with no head. We usually leave out the ( symbol when we write goals. A goal of the form ( x = y, with x; y 2 *V* is called a trivial goal. A Horn equational theory E , satisfying 62 V and V ar( ) V ar( ) for each clause ( = ) ( C, can be viewed as a CTRS R, where the rules are the heads (implicitly oriented from left to right) and the conditions are the respective bodies.

*Each equational Horn theory E generates a smallest congruence relation*

*=E called E -equality on the set of terms ( [ V ) (the least equational theory* which contains all logic consequences of E under the entailment relation j= obeying the axioms Eq [5](#_bookmark2) of the equality for E ). E is a presentation or axio- matization of =E . In abuse of notation, we sometimes speak of the equational theory E to denote the theory axiomatized by E . We will denote by H=E the

*nest partition ( )= =E induced by =E over the set of ground terms ( ).*

*H=E is usually called the initial algebra of E* [*[19*](#_bookmark32)*]. Satis ability in H=E is called* E -unif iability, that is, given a set of equations E, E is E -uni able i there

*5 The set Eq of equality axioms for a given program R are: re exivity (x = x (), symmetry (x = y ( y = x), transitivity (x = z ( x = y; y = z) and f -substitutivity (f (x1 ;:::; xn) = f (y1;:::; yn) ( x1 = y1;::: ; xn = yn, for f =n 2 ).*

*exists a substitution such that E j= E* [*[19].*](#_bookmark32) *The substitution is called* an E -uni er of E. Let =R be the re exive, symmetric, and transitive closure of !R. If E is the set of (conditional) equations corresponding to R, then

*=E and =R coincide. Via this correspondence, the notion of R-uni cation is*

*implicitly de ned. We say that R [W ] if there is a substitution such* that =R [W ], i.e. x =R x for all x 2 W .

*2.1 Functional Logic Programming*

*Functional logic languages are extensions of functional languages with prin-* ciples derived from logic programming [[42](#_bookmark55)]. The computation mechanism of functional logic languages is based on narrowing , a generalization of term re- writing where uni cation replaces matching: both the rewrite rule and the term to be rewritten can be instantiated. Under the narrowing mechanism, functional programs behave like logic programs: narrowing solves equations by computing solutions with respect to a given CTRS, which is henceforth called the \program".

*De nition 2.1 (Narrowing) Let R be a program and g be an equational* goal. We say that g conditionally narrows into g0 if there exists a position u 2 O(g), a standardized apart variant r ( ! ( C) of a rewrite rule in R, and a substitution such that: is the most general uni er of gju and ,

*and g0 (C; g[ ]*

*) . We write g [u;r; ] g0 or simply g g0. The relation is*

*u ; ; ;*

*called (unrestricted or full) conditional narrowing.*

*A narrowing derivation for g in R is de ned by g g0 i 9 ;::: ; 9 : g*

*1*

*; 1 n ;*

*::: n*

*;*

*g0 and = 1*

*::: n*

*. We say that the derivation has length n. If n = 0,*

*then = . The extension of a CTRS R with the rewrite rules for dealing* with the equality is denoted by R+. In the case of unrestricted narrowing, R+ denotes R[ fx = x ! trueg, x 2 V . This allows us to treat syntactical uni cation as a narrowing step, by using the rule (x = x ! true) to compute mgu's. Then s = t ; true holds i = mgu(fs = tg).

*We use the symbol > as a generic notation for sequences of the form* true;::: ; true. A successful derivation (or refutation) for g in R+ is a narro-

*wing derivation g >, where*

*;*

*j V ar(g)*

*is the computed answer substitution*

*(cas).*

*The narrowing mechanism is a powerful tool for constructing complete E -* uni cation algorithms for useful classes of equational theories. In this context, completeness means that, for every solution to a given set of equations, a more general solution can be found by narrowing. Formally, a narrowing algorithm is complete for (a class of) CTRS's if it generates a solution at least as general as any that satis es the query (it generates a complete set of E -uni ers).

*2.2 Complete Narrowing Strategies*

*Since unrestricted narrowing has quite a large search space, several strategies* to control the selection of redexes have been developed. A narrowing strategy (or position constraint) is any well-de ned criterion which obtains a smaller search space by permitting narrowing to reduce only some chosen positions. A narrowing strategy ' can be formalized as a mapping that assigns a subset '(g) of O(g) to every goal g (di erent from >) such that, for all u 2 '(g), the goal g is narrowable at position u. An important property of a narrowing strategy ' is completeness, meaning that the narrowing constrained by ' is still complete. There is an inherited tradeo coming from functional programming, between the bene ts of outside-in evaluation of orthogonal, nonterminating rules and those of inner or eager evaluation with terminating, non orthogonal rules. A survey of results about the completeness of narrowing strategies can be found in [[4,21](#_bookmark34)[,22,29].](#_bookmark42) To simplify our notation, we let IR' denote the class of CTRS's which satisfy the conditions for the completeness of the narrowing strategy '.

*We let inn(g) (resp. out(g)) denote the narrowing strategy which assigns* the position p of the leftmost-innermost (resp. leftmost-outermost) narrowing redex of g to the goal g. [6](#_bookmark3) We formulate a conditional narrower with strategy ', ' 2 finn; outg, as the smallest relation ;' satisfying

*u = '(g) ^ ( ! ( C) << R'+ ^ = mgu(fgju = g)* g ;' (C; g[ ]u)

*For ' 2 finn; outg, R'+ = R[ fEq'g, where Eq' are the rules which model*

*the equality on data terms:*

*c =' c ! true % c=0 2 C*

*c(x1;::: ; xn) =' c(y1;::: ; yn) ! (x1 =' y1) ^ ::: ^ (xn =' yn) % c=n 2 C Here =' is the standard equality = of terms whenever ' = inn, while for*

*the case when ' is out, and nonterminating rules are considered, we need to*

*distinguish the standard (non-strict) equality =, which is de ned on partially* determined or in nite data structures from the strict equality , which is only de ned on nite and completely determined data structures, and which gives to equality the weak meaning of identity of nite objects (e.g., see [[39]).](#_bookmark52) We also assume that equations in g and C have the form s = t whenever we consider ' = inn, whereas the equations have the form s t when we consider ' = out. Note that a non{strict equation like f (a) = g(a) is not an acceptable goal when ' = out.

*6 An innermost term is an operation applied to constructor terms, i.e., a term of the form f (d1;::: ; dk ), where f 2 F and for all i = 1;::: ; k, di 2 (C [ V ). The leftmost-innermost position of g is the leftmost position of g which points to an innermost subterm. A position p is leftmost-outermost in a set of positions O if there is no p0 2 O with p0 pre x of p, or p0 = q:i:q0 and p = q:j:q00 and i < j.*

*Innermost narrowing is the foundation of several functional logic program-* ming languages like SLOG [[26],](#_bookmark39) LPG [[7,8]](#_bookmark21) and (a subset of) ALF [[28].](#_bookmark41) Inner- most narrowing corresponds to the eager evaluation strategies in functional programming. Modern functional logic languages like Curry [[31](#_bookmark44)], Escher [[35](#_bookmark49)] and Toy [[13](#_bookmark26)] are based on lazy evaluation principles, which delay the eva- luation of function arguments until their values are needed to compute some result. This avoids unnecessary computations and allows one to deal with in nite data structures [[29].](#_bookmark42) Needed narrowing is a complete lazy narrowing strategy which is optimal w.r.t. the length of the derivations and the number of computed solutions in inductively sequential programs. Informally, inductive sequentiality amounts to the existence of discriminating left-hand sides, i.e. typical functional programs. Needed narrowing can be easily and eÆciently implemented by translating de nitional trees into case expressions as propo- sed in [[32],](#_bookmark45) which also proves that there is a strong equivalence of needed narrowing derivations in the original program and leftmost-outermost nar- rowing derivations in the transformed program. A similar transformation is presented in [[44](#_bookmark57)], where inductively sequential programs are translated to uni- form form, which has only at rules with pairwise non-subuni able left-hand sides, where the strong equivalence between needed narrowing and leftmost- outermost narrowing derivations also holds.

# *3 Denotation of a Functional Logic Program*

*A Herbrand interpretation I for a program R is a set of ground equations,* with the understanding that s = t is true w.r.t. I i s = t 2 I. A Her- brand interpretation satis es a program clause i , for each ground instance

*= ( C of the clause, we have that = 2 I whenever C I. A Her-*

*brand E-interpretation for R is a Herbrand interpretation for R obeying the* equality axioms for R. A Herbrand model for R is a Herbrand interpretation for R which satis es each program clause in R. A Herbrand E-model for R is a Herbrand model for R which satis es the equality axioms for R. The inter- section of all Herbrand E-models for R is also a Herbrand E-model for R (the least Herbrand E-model), and it was proposed as the declarative semantics for positive programs [[33](#_bookmark46)]. This semantics is known to be isomorphic to the initial algebra H=E of the program, and in the following will be denoted by M(R).

*For canonical programs, M(R) is equivalent to the operational semantics* given by the ground success set, i.e. the set of all ground equations s = t such that s and t have a common R-normal form, and to the xpoint semantics given by the least xpoint TR"! of the following transformation TR (imme- diate consequence operator), which is continuous on the complete lattice of

*Herbrand interpretations ordered by set inclusion* [*[33*](#_bookmark46)*].*

*TR(I) = ft = t 2 B g [ fe 2 B j ( ! ( C) 2 [R];*

*fe[ ]ug[ C I; u 2 O(e); eju = g*

*Informally, TR(I) contains the set of all ground instances of the re exivity* axiom and the set of all ground equations that can be `constructed' from elements of the Herbrand interpretation I by replacing one occurrence of the right-hand side of the head of a rule in R by the corresponding left-hand side. In order to formulate a semantics for functional logic programs modeling computed answers, the usual Herbrand base has to be extended to the set of all (possibly) non-ground equations modulo variance [[23](#_bookmark36)[,24].](#_bookmark37) HV denotes the

*V -Herbrand universe which allows variables in its elements, and is de ned as*

*( [ V )=. For the sake of simplicity, the elements of HV have the same* representation as the elements of ( [ V ) and are also called terms. BV denotes the V -Herbrand base, namely, the set of all equations s = t modulo variance, where s; t 2 HV . Note that the standard Herbrand base B is equal to [BV ]. The preorder on ( [V ) induces an order relation on ( [V )= (and therefore on HV ). The ordering on HV induces an ordering on BV , namely s0 = t0 s = t if s0 s and t0 t. The power set of BV is a complete lattice under set inclusion.

*In the following, we introduce a semantics Fca(R) for program R such* that the computed answer substitutions of any (possibly conjunctive) goal g can be derived from Fca(R) by uni cation of the equations in the goal with the equations in the denotation. We assume that the equations in the denotation are renamed apart. Equations in the goal have to be attened

*'*

*'*

*rst, i.e. subterms have to be unnested so that the term structure is directly*

*accessible to uni cation.*

*De nition 3.1 ( at goal w.r.t. ') A at equation is an equation of the* form f (d1;::: ; dn) = d or d1 =' d2, where d; d1;::: ; dn 2 (C [ V ) are con- structor terms. A at goal is a set of at equations.

*For the outermost strategy, ' = out, the only non{strict equations in a*

*at goal are of the form f (d~) = x. These equalities are treated di erently*

*from those originally present in the bodies of program rules and in the goal* (denoted by ). In particular, the clauses for = must allow the elimination of f (a) = x, whenever f (a) would not have been selected by narrowing (i.e., when its value is not required to reduce g(f (a)).

*Any sequence of equations E can be transformed into an equivalent one,* f lat'(E), which is at. The attening procedures for equation sets which produce at goals w.r.t. inn and out, respectively, can be found in [[10](#_bookmark22)[,27].](#_bookmark40)

*It is known that the xpoint semantics allows for the reconstruction of the* top down operational semantics and allows for the (bottom-up) computation of a model which is completely independent from the goal.

*3.0.1 Fixpoint Semantics*

*Now we are ready to introduce a new, generic immediate consequence operator* T ' which models computed answers w.r.t. '. For any program R, we denote by R the set of identical equations f (x1;::: ; xn) = f (x1;::: ; xn), for each

*R*

*function symbol f =n 2 D. We let ='*

*R*

*denote the set of the identical equations*

*c(x1;::: ; xn) =' c(x1;::: ; xn) for the constructor symbols c=n occurring in R* only. As we will see, these functional re exivity axioms play an important role in de ning the xpoint semantics of R.

*In non{strict languages, if the compositional character of meaning has* to be preserved in presence of in nite data structures and partial functions, then non-normalizable terms, which may occur as subterms within normali- zable expressions, also have to be assigned a denotation. Such a denotation is bound to the class of all partial results of the in nite computation along with the usual approximation ordering on them [[27,39](#_bookmark52)] or, equivalently, the in nite data structure de ned as the least upper bound of this class. Following [[27](#_bookmark40)[,39],](#_bookmark52) we introduce a fresh constant symbol ? into to represent the value of expressions which would otherwise be unde ned.

*De nition 3.2 Let I be a Herbrand interpretation and R 2 IR'. Then,*

*T '(I) = R [ =' [ fe 2 B j ( ! ( C) << R';*

*R R* *V*

*fl = rg[ C0 I;*

*mgu(f lat'(C); C0) = ;*

*mgu(f = rjug ) = ; u 2 O '(r);* e = (l = r[ ]u) g:

*where R' = R if ' = inn, while R' = R[ ff (t~) ! ? j f =n 2 Dg if ' = out.*

*The following proposition allows us to de ne the xpoint semantics.*

*Proposition 3.3 The T '*

*R*

*operator is continuous on the complete lattice of*

*Herbrand interpretations, ' 2 finn; outg. The least xpoint lf p(T ') = T ' " !.*

*R R*

*De nition 3.4 The least xpoint semantics of a program R in IR' is de ned*

*as F'*

*R*

*(R) = lf p(T '), ' 2 finn; outg.*

*De nition 3.5 We let Fca(R) denote fl = r 2 lf p(T ') j r does not contain*

*' R*

*any de ned function symbol f =n 2 Dg, ' 2 finn; outg.*

*Theorem 3.6 (strong soundness and strong completeness)*

*Let R be a program in IR' and g be a (non{trivial) goal according to '. Then*

*is an answer substitution computed by ;' for g in R i there exist g0 (e1;::: ; en) << Fca(R) such that = mgu(f lat'(g); g0)j V ar(g).*

*'*

*Example 3.7 Let us consider the program R = fg(x) ! 0; f(0) ! 0; f(s(x))*

*! f(x)g. According to De nition* [*3.4,*](#_bookmark5) *Fca (R) = f0 = 0; s(x) = s(x); g(x) = 0; f(0) = 0; f(s(0)) = 0;::: ; f(sn (0)) = 0;:: :g. Given the goal g (y =* f(z)), innermost narrowing computes the answers ffy=0; z=0g; fy=0; z=s(0)g;

*inn*

*::: ; fy=0; z=sn(0)g;:: :g in R, which exactly coincides with the set of substi-* tutions computed by unifying the at goal f(z) = y with the equations in Fca (R).

*inn*

*Example 3.8 Now consider the program R = ffrom(x) ! [xjfrom(s(x))];* first([xjy]) ! xg. According to De nition [3.4,](#_bookmark5) Fca (R) = fs(x) s(x); from(x) = ?; from(x) = [xj?];:::; from(x) = [xj[s(x)j ::: [sn (x)j?]]];::: ; first(x) = ?; first([xjy]) = xg, with n 2 !. Given the goal g (first (from(s(x))) z), outermost narrowing only computes the answer fz=s(x)g in R, which is also the only substitution which can be computed by unifying the at goal (first(y) = w; from(s(x)) = y; w z) in Fca (R).

*out*

*out*

*According to Theorem* [*3.6,*](#_bookmark6) *Fca(R) can be used to simulate the execution* for any (non{trivial) goal g, that is, Fca(R) can be viewed as a (possibly in nite) set of `unit' clauses, and the computed answer substitutions for g in R can be determined by `executing' f lat(g) in the program Fca(R) by standard uni cation, as if the equality symbol were an ordinary predicate. In the following, we show the relation between the semantics Fca(R) and a novel operational \computed answer" semantics Oca(R) which correctly models the behavior of single equations, which we introduce in the following.

*'*

*'*

*'*

*'*

*'*

*3.0.2 Computed Answer Semantics*

*The operational success set semantics Oca(R) of a program R w.r.t. narrowing* semantics ' is de ned by considering the answers computed for \most general calls", as shown by the following de nition.

*'*

*De nition 3.9 Let R be a program in IR'. Then,*

*Oca(R) = =' [ f(f (x ;::: ;x ) = x*

*) j (f (x ;::: ;x ) =* *x*

*) >*

*' R* *1*

*n n+1*

*1 n '*

*n+1 ;'*

*where f =n 2 D, xn+1 and xi are distinct variables, for i = 1;::: ;n g.*

*The equivalence between the operational and the least xpoint semantics* is established by the following theorem.

*Theorem 3.10 If R 2 IRinn, then Oca (R) = Fca (R).*

*inn*

*inn*

*If R 2 IRout, then Oca (R) = fl = r 2 Fca (R) j ? does not occur in rg*

*out out*

*The following theorem relates the non-ground semantics Oca(R) to the*

*'*

*standard least Herbrand E-model semantics M(R).*

*Theorem 3.11 Let R be a program in IR' and r be the transitive-symmetric* closure of relation r under replacement (the f -substitutivity property). Then, M(R) = [Oca(R)].

*'*

# *4 Diagnosis of declarative programs*

*We now introduce some basic de nitions on the diagnosis of declarative pro-* grams [[17](#_bookmark30)]. As operational semantics we consider the set of computed answer

*substitutions.*

*De nition 4.1 Let I be the speci cation of the intended computed answer* semantics for R.

*(i) R is partially correct w.r.t. I, if Oca(R) I.*

*'*

*(ii) R is complete w.r.t. I, if I Oca(R).*

*'*

*(iii) R is totally correct w.r.t. I, if Oca(R) = I.*

*'*

*If a program contains errors, these are signalled by corresponding sym-* ptoms.

*De nition 4.2 Let I be the speci cation of the intended computed answer* semantics for R.

*(i) An incorrectness symptom is an equation e such that e 2 Oca(R) and*

*'*

*e 62 I.*

*(ii) An incompleteness symptom is an equation e such that e 2 I and e* *62*

*Oca(R).*

*'*

*In case of errors, in order to determine the faulty rules, we give the following* de nitions.

*De nition 4.3 Let I be the speci cation of the intended xpoint semantics*

*for R. If there exists an equation e 2 T '*

*frg*

*(I) and e 62 I , then the rule r 2 R*

*is incorrect on e.*

*Therefore, the incorrectness of rule r is signalled by a simple transformation* of the intended semantics I.

*De nition 4.4 Let I be the speci cation of the intended xpoint semantics* for R. An equation e is uncovered if e 2 I and e 62 T '(I).

*R*

*By the above de nition, an equation e is uncovered if it cannot be derived* by any program rule using the intended xpoint semantics.

*Proposition 4.5 If there are no incorrect rules in R w.r.t the speci cation of* the intended xpoint semantics, then R is partially correct w.r.t. the intended computed answer semantics.

*We now consider a bottom-up abstract debugger for a strict language.*

*Hence, in the rest of this paper, we x ' = inn.*

# *5 Abstract success set*

*The theory of abstract interpretation* [*[18*](#_bookmark31)*] provides a formal framework for de-* veloping advanced data- ow analysis tools. Abstract interpretation formalizes the idea of `approximate computation' in which computation is performed with descriptions of data rather than with the data itself. The semantics operators are then replaced by abstract operators which are shown to `safely' approxi- mate the standard ones. In this section, starting from the xpoint semantics

*in Section* [*3,*](#_bookmark4) *we develop an abstract semantics which approximates the obser-* vable behavior of the program and is adequate for modular data- ow analysis, such as the analysis of unsatis ability of equation sets or any analysis which is based on the program success set. We assume the framework of abstract interpretation for analysis of equational unsatis ability as de ned in [[2].](#_bookmark15) Ano- ther approach to constructing an abstract term rewriting system is followed in [[9].](#_bookmark23) We think that another approximation of the xpoint semantics given in the previous section which is di erent from the one that we describe in this section can be characterized by following an approach similar to that in [[9].](#_bookmark23) We rst recall the abstract domains and the associated abstract operators.

*Then we describe a novel abstract immediate consequence operator T ]*

*R*

*which*

*is able to approximate the operator TR, and the abstract xpoint semantics* F](R). In the following, we denote the abstract analog of a concrete object O by O].

*5.1 Abstract Domains and Operators*

*A description is the association of an abstract domain (D; ) (a poset) with a* concrete domain (E; ) (a poset). When E = Eqn or E = Sub, the description is called an equation description or a substitution description, respectively. The correspondence between the abstract and concrete domain is established through a `concretization' function : D ! 2E. We say that d approximates e, written d / e, i e 2 (d). The approximation relation can be lifted to relations and cross products as usual [[2].](#_bookmark15)

*Abstract substitutions are introduced for the purpose of describing the* computed answer substitutions for a given goal. Abstract equations and ab- stract substitutions correspond, in our approach, to abstract program de- notations and abstract observable properties, respectively. The domains for equations and substitutions are as follows.

*De nition 5.1 (abstract Herbrand universe) Let ] be an irreducible sym-*

*bol, where ] 62 . Let H]*

*V*

*= ( ( [ V [ f]g); ) be the domain of terms over*

*the signature augmented by ], where the partial order is de ned as follows:*

*(a) 8t 2 H] :] t and t t and*

*V*

*(b) 8s1;::: ;s ; s0 ;::: ; s0 2 H] ; 8f =n 2 . s0 s*

*^ ::: ^ s0 s )*

*n 1 n* *V*

*1 1 n n*

*f (s0 ;::: ; s0 ) f (s1;::: ; sn):*

*1 n*

*This order can be extended to equations: s0 = t0 s = t i s0 s and* t0 t and to sets of equations S; S0:

*1) S0 S i 8e0 2 S0:9e 2 S such that e0 e. Note that S0 ftrueg ) S0 ftrueg.*

*2) S0 v S i (S0 S) and (S S0 implies S0 S).*

*Intuitively, S0 v S means that either S0 contains less information than S,* or if they have the same information, then S0 expresses it using fewer elements.

*Roughly speaking, the special symbol ] introduced in the abstract domains* represents any concrete term. The behaviour of the symbol ] from a program- ming viewpoint resembles that of an \anonymous" variable in Prolog. From the viewpoint of logic, ] stands for an existentially quanti ed variable [[2,37](#_bookmark50)[,38].](#_bookmark51) De ne [[S]] = S0, where the n-tuple of occurrences of ] in S is replaced by an n-tuple of existentially quanti ed fresh variables in S0.

*De nition 5.2 An abstract substitution is a set of the form fx1=t1;::: ; xn=tng* where, for each i = 1;::: ; n, xi is a distinct variable in V not occurring in any of the terms t1;::: ; tn and ti 2 ( [ V [ f]g). The ordering on ab- stract substitutions is given by logical implication: let ; 2 Sub], i [[ ]] ) [[ ]].

*b b*

*The descriptions for terms, substitutions and equations are as follows.*

*De nition 5.3 Let HV*

*= ( ( [ V ); ) and H]*

*= ( ( [ V [ f]g); ).*

*The term description is hH] ; ; H*

*V*

*V*

*V*

*i where : H]*

*! 2HV is de ned by:*

*(t0) = ft 2 HV jt0 tg.*

*V*

*Let Eqn be the set of nite sets of equations over ( [V ) and Eqn] be the* set of nite sets of equations over ( [ V [ f]g). The equation description is h(Eqn]; v); ; (Eqn; )i, where : Eqn] ! 2Eqn is de ned by: (g0) = fg *2*

*Eqnjg0 v g and g is unquanti ed g.*

*Let Sub be the set of substitutions over ( [ V ) and Sub] be the set* of substitutions over ( [ V [ f]g). The substitution description h(Sub];

*); ; (Sub; )i, where : Sub] ! 2Sub is de ned by: ( ) = f 2 Subj g.*

*In order to perform computations over the abstract domains, we have to* de ne the notion of abstract uni cation. The abstract most general uni er for our method is very simple and, roughly speaking, it boils down to computing a solved form of an equation set with (possibly) existentially quanti ed variables. We de ne the abstract most general uni er for an equation set S0 2 Eqn] as follows. First replace all occurrences of ] in S0 by existentially quanti ed fresh variables. Then take a solved form of the resulting quanti ed equation set and nally replace the existentially quanti ed variables again by ]. Formally:

*]*

*let 9y ::: y :S = solve([[S0]]) and = fy =];::: ;y =]g. Then mdgu (S0) = S .*

*1*

*n*

*1*

*n*

*The fact that 8 2 unif ([[S]]): mgu](S) justi es our use of `most general'.*

*The safety of the abstract uni cation algorithm has been proven in* [*[2*](#_bookmark15)*].*

*Our analysis is based on a form of simpli ed (abstract) program which* always terminates and in which the query can be executed eÆciently. Our notion of abstract program is parametric with respect to a loop-check. Two di erent instances can be found in [[2,3].](#_bookmark16)

*De nition 5.4 A loop-check is a graph GR associated with a program R, i.e.* a relation consisting of a set of pairs of terms, such that:

*(1) the transitive closure G+ is decidable and*

*R*

*Æ*

*(2) Let t = t0 be a function which assigns to a term t some node t0 in GR.*

*If there is an in nite sequence:*

*h( G0; 0i ; h( G1; 1i ; :::*

*Æ Æ*

*then 9i 0: hti; tii 2 G+; where ti = eju i; e 2 Gi and u 2 O (e): (we refer*

*R*

*Æ Æ*

*to hti; tii as a `cycle' of GR.)*

*A program is abstracted by simplifying the right-hand side and the body* of each clause. This de nition is given inductively on the structure of terms and equations. The main idea is that terms whose corresponding nodes in GR have a cycle are drastically simpli ed by replacing them by ]. We use this de nition in a iterative manner. We rst abstract a concrete rule r obtaining r] (we select a rule with direct recursion if any; otherwise we choose any rule in the program). Then we replace r by r] in R, and recompute the loop{check before proceeding to abstract the next rule. Each concrete rule is considered only once in the abstraction process.

*De nition 5.5 (abstract rule) Let R be a program and let r = ( ! (* C) 2 R. Let GR be a loop-check for R. We de ne the abstraction of r as follows: r] = ( ! sh( ; GR) ( sh(C; GR)) where the shell sh(x; G) of an expression x according to a loop-check G is de ned inductively

*8>x if x 2 V*

*> Æ Æ +*

*f (sh(t1; G);::: ; sh(tk; G)) if x f (t1;::: ; tk) and hx; xi 26 G*

*<*

*>*

*sh(x; G) =*

*sh(l; G) = sh(r; G) if x (l = r)*

*sh(e1; G);::: ; sh(en; G) if x e1;::: ; en*

*>*

*>:] otherwise*

*We can now formalize the abstract semantics.*

*5.2 Bottom-up Abstract Semantics*

*We de ne an abstract xpoint semantics in terms of the least xpoint of a*

*continuous transformation T ]*

*R*

*based on abstract uni cation and the operation*

*of abstraction of a program. The idea is to provide a nitely computable* approximation of the concrete denotation of the program R. In the following, we de ne the abstract transformation T ] .

*R*

*De nition 5.6 (abstract Herbrand base, abstract Herbrand inter-*

*pretation) The abstract Herbrand base of equations B]*

*V*

*is de ned as the set*

*of equations over the abstract Herbrand universe H] . An abstract Herbrand* interpretation is any element of 2B] . A partial order ] on abstract interpre- tations can be de ned in a way similar to the order on interpretations.

*V*

*V*

*We can show that the set of abstract interpretations is a complete lattice*

*w.r.t. ]. An abstract trivial equation is an equation ] = X, X = ] or ] = ].*

*De nition 5.7 Let R be a program, GR be a loop-check for R and R] be the* abstraction of R using GR where we also drop any abstract trivial equation

*from the body of the rules. Let I be an abstract interpretation. Then,*

*T ] (I) = R [ =inn [ fe 2 B]*

*R R* *V*

*j ( ! ( C) << R];*

*fl = rg[ C0 I;*

*mgu](f lat(C); C0) = ;*

*mgu](f = (rju)g ) = ;*

*u 2 O (r); e = (l = r[ ]u) g:*

*Proposition 5.8 The T ]*

*R*

*operator is continuous on the complete lattice of*

*abstract interpretations.*

*We can de ne F](R) and Fca](R) in a way similar to the concrete con-* structions of F'(R) and Fca(R), as in Section [3.](#_bookmark4)

*'*

*De nition 5.9 (abstract least xpoint semantics) The abstract least x-* point semantics of a program R is F](R) = lf p(T ] ). Let Fca](R) = fl = r 2 F](R) j r does not contain any de ned function symbol f =n 2 Dg

*R*

*The following theorem states that F](R) and Fca](R) are nitely compu-* table.

*Theorem 5.10 There exists a nite positive number k such that F](R) =* T ] " k.

*R*

*From a semantics viewpoint, given a program R, the xpoint semantics* F(R) (resp. Fca(R)) is approximated by the corresponding abstract xpoint semantics F](R) (resp. Fca](R)). That is, we can compute an abstract appro- ximation of the concrete semantics in a nite number of steps. The correctness of the abstract xpoint semantics with respect to the concrete semantics is proved by the following:

*Theorem 5.11 There exists a nite positive number k such that T ]*

*R*

*" k /*

*TR " !.*

*Corollary 5.12 F](R) / F(R) and Fca](R) / Fca(R).*

*The semantics Fca](R) collects goal-independent information about suc-* cess patterns of a given program. The relation between the abstract xpoint and the concrete operational semantics (computed answer substitutions) is given by the following theorem. Roughly speaking, given a goal ( G, we obtain a description of the set of the computed answers of ( G by abstract uni cation of the equations in f lat(G) with equations in the approximated semantics Fca](R).

*Theorem 5.13 (strong completeness) Let R be a program in IR' and* ( g be a non{trivial goal. If is a computed answer substitution for ( g in R, then there exists g0 e1;::: ; em << Fca](R) such that 0 = mgu](f lat(g); g0) and 0 .

*Example 5.14 Let us consider the program R = fg(0) ! 0; g(c(x)) !*

*Semantic Property Requirement*

|  |  |
| --- | --- |
| *R is partially correct w.r.t. (I+; I )* | *6 9e:e 2 TR(I ) and e 62 I+* |
| *R is complete w.r.t. (I+; I )* | *6 9e:e 2 I and e 62 TR(I+)* |
| *R is incorrect w.r.t. (I+; I )* | *9e:e 2 TR(I ) and e 62 I+* |
| *R is incomplete w.r.t. (I+; I )* | *9e:e 2 I and e 62 TR(I+)* |

*Table 1*

*SuÆcient Conditions for Correctness and Completeness*

*c(g(x)); f(0) ! 0; f(c(x)) ! x ( g(c(x)) = c(x)g. Let us consider the loop-*

*Æ*

*check* [*[3]*](#_bookmark16) *GR = fhg(x); g(x)ig, and let t= t0 be the (partial) function which,*

*given a graph, assigns to a term t some node t0 in the graph such that t0 uni es* with t, if one such node t0 exists.

*Then, the abstraction of the program R is*

*R] = fg(0) ! 0; g(c(x)) ! c(]); f(0) ! 0; f(c(x)) ! x ( g(c(x)) = c(x)g.*

*F(R) = f0 = 0; g(x) = g(x); f(x) = f(x); c(x) = c(x); g(0) = 0; f(0) = 0; g(c(x)) = c(g(x));:::; g(cn (x)) = cn(g(x));::: ; g(c(0)) = c(0);::: ;*

*g(cn(0)) = cn (0);::: ; f(c(0)) = 0;::: ; f(cn (0)) = cn 1 (0);:: :g and the corre-* sponding xpoint abstract semantics is the nite set

*F](R) = f0 = 0; g(x) = g(x); f(x) = f(x); c(x) = c(x); g(0) = 0; f(0) =*

*0; g(c(x)) = c(]); f(c(x)) = xg, and*

*Fca](R) = f 0 = 0; c(x) = c(x); g(0) = 0; f(0) = 0; g(c(x)) = c(]); f(c(x)) =*

*xg which approximates the program success set.*

*Given a goal ( g ( f(g(x)) = y, innermost conditional narrowing* computes the in nite set of substitutions ffx=0; y=0g; fx=c(0); y=0g; fx=c2 (0); y=c(0)g;::: ; fx=cn (0); y=cn 1 (0)gg. The abstract substitutions returned by abstract uni cation of the equations in the attened goal ( f(z) = y; g(x) = z with Fca](R) are ffx=0; y=0g; fx=c(x0); y=]gg, which approximate the com- puted answers of ( g.

# *6 Abstract diagnosis*

*An eÆcient debugger can be based on the notion of over-approximation and* under-approximation for the intended xpoint semantics that we have intro- duced. The basic idea is to consider two sets to verify partial correctness: I+ which overapproximates the intended semantics I (that is, I (I+)) and I which underapproximates I (that is, I (I)). We can then use such sets as shown in Table [1.](#_bookmark10)

*Proposition 6.1 If there exists an equation e such that e 62 I+ and e* *2*

*Tfrg(I), then the rule r 2 R is incorrect on e.*

*Proposition 6.2 If there exists an equation e such that e 62 TR(I+) and* e 2 I, then the equation e is uncovered.

*In the following, by abuse we let I denote the program that speci es the* intended semantics. In the following, we consider I+ = lf p(T ]), i.e. we consider the abstract success set that we have de ned in previous section as overapproximation of the success set of a program. We can consider any of the sets de ned in the works of [[11,16](#_bookmark29)] as underapproximation of I. Alternatively, we can simply take the set which results from a nite number of iterations of the TI function (the concrete operator). Let us illustrate this method.

*I*

*Example 6.3 Let us consider a program R = fg(0) ! 0; f(0) ! 0; g(c(x))*

*! g(x); f(c(x)) ! x ( g(c(x)) = c(x)g which is incorrect when we consider* as intended speci cation the following program

*I = fg(0) ! 0; f(0) ! 0; g(c(x)) ! c(g(x)); f(c(x)) ! g(x)g.* Then, by using the loop check in Example [5.14](#_bookmark9) we have

*I] = fg(0) ! 0; f(0) ! 0; g(c(x)) ! c(]); f(c(x)) ! g(x)g.*

*After three iterations of the TI operator, we get:*

*I = f0 = 0; c(x) = c(x); f(x) = f(x); g(x) = g(x); f(0) = 0; g(0) = 0; f(c(x)) = g(x); g(c(x)) = c(g(x)); f(c(0)) = 0; g(c(0)) = c(0); f(c2(x)) =*

*c(g(x)); g(c2 (x)) = c2 (g(x)); f(c2(0)) = c(0); f(c3 (x)) = c2 (g(x)); g(c2 (0)) =*

*c2(0); g(c3 (x)) = c3 (g(x))g.*

*After two iterations of the T ]*

*I*

*operator, we get the xpoint:*

*I+ = F](I) = lf p(T ]) = f0 = 0; c(x) = c(x); f(x) = f(x); g(x) = g(x); f(0) = 0; g(0) = 0; f(c(x)) = g(x); g(c(x)) = c(]); f(c(0)) =*

*I*

*0; f(c2 (x)) = c(])g. And*

*TR(I) = f0 = 0; c(x) = c(x); f(x) = f(x); g(x) = g(x); f(0) = 0; g(0) =*

*0; g(c(x)) = g(x); f(c(0)) = 0; f(c2 (x)) = g(x); g(c(0)) = c(0); g(c2 (x)) =*

*c(g(x)); f(c2 (0)) = c(0); f(c3 (x)) = c(g(x)); g(c2 (0)) = c2 (0); f(c3 (0)) =*

*c2(0); f(c4 (x)) = c2(g(x)); g(c3 (x)) = c2 (g(x)); g(c3 (0)) = c3 (0); g(c4 (x)) =*

*c3(g(x))g.*

*Now we can derive that g(c(x)) = g(x) 2 TR(I), while g(c(x)) = g(x) 62* I+. Hence, the corresponding rule g(c(x)) ! g(x) is wrong.

# *7 The System for Declarative Debugging Buggy*

*The basic rules presented so far have been implemented by a prototype system* Buggy [[1],](#_bookmark14) which is available at

[*http://www.dsic.upv.es/users/elp/soft.html*](http://www.dsic.upv.es/users/elp/soft.html)

*It includes a parser for a conditional functional logic language, whose seman-* tics is based on innermost (basic) narrowing, which is a generalization of basic narrowing such that R does not need to be completely de ned, which is quite a restrictive condition which is necessary for the completeness of innermost narrowing [[10](#_bookmark22)[,33].](#_bookmark46) The implementation also includes a module for computing an abstraction of the program based on a given loop-check, an automatic de- bugger which requires that the user indicate some parameters, such as the number n of iterations for approximating the success set from the bottom.

*Then the errors are automatically found by the debugger and the user has to* indicate the corrections to be made on the wrong rules. The Buggy system is written in SICStus Prolog v3.8.1 and the complete implementation consists of about 300 clauses (1260 lines of code). The debugger is expressed by *147*

*clauses (including the user interface and the code needed to handle the repre-*

*sentation), the parser and other utilities are expressed by 65 clauses and basic* narrowing is implemented by 88 clauses. Language syntax follows mainly that of a generic conditional functional logic language, with conditional basic nar- rowing.

*The Buggy main screen allows the user to choose between several alternatives.*

*(i) File options: It contains the classical options for opening, loading and* saving a le, as well as cleaning and exiting the system.

*(ii) Edit: it is possible to load, edit, and visualize the program to be de-* bugged on the screen, as well as its intended semantics. The program is incrementally parsed while it is loaded.

*(iii) Debug: the debugger starts debugging the input program w.r.t. the in-* tended semantics. The user has to say how many iterations to apply for approximating from below the intended semantics. The errors are shown one by one to the user who is required to propose the corrections which are in turn tested. The nal correct program is shown to the user who can save it.

*(iv) Help: contains additional information about the system.*

*We have tested our debugging methodology over several benchmarks. We* have considered programs such as append for computing the concatenation of two input lists, last which returns the last element of a list, knapsack which returns a set of elements of the input list whose weight sum is equal to an input integer value, fibonacci which computes the Fibonacci numbers, fact which computes the factorial of a positive number, sort which uses the insertion sort for ordering an input list of integers. For all these programs by using the intended semantics we detected the errors which were inserted in the program. The nal programs passed the tests of correctness and completeness.

*Let us illustrate the power of our debugging system when functional logic* programs are used as speci cation of the intended semantics. The idea goes back to the origins of declarative programming, considering declarative speci-

*cations as programs for rapid prototyping* [*[15*](#_bookmark28)*]. In our case, we go one step*

*further, because the intended speci cation can then be automatically abstrac-* ted and can be used to automatically debug the nal eÆcient program.

*The following program should order a list of integers using the insertion sort.*

*sort([X]) ! []:*

*sort([XjXs]) ! Ys ( sort(Xs) = Zs;*

*insert(X; Zs; Ys) = true:*

*insert(X; []; [X]) ! true:*

*insert(X; [YjYs]; [YjZs]) ! true ( X > Y = true;*

*insert(X; Ys; Zs) = true: insert(X; [YjYs]; [X; YjYs]) ! true ( X =< Y = true:*

*The intended speci cation I is given by the (quite) ineÆcient program which*

*uses the naive sorting algorithm which computes the permutations of the input* list.

*sort(Xs) ! Ys ( perm(Xs) = Ys; ord(Ys) = true: ord([]) ! true:*

*ord([X]) ! true:*

*ord([X; YjXs]) ! true ( X =< Y = true; ord([YjXs]) = true: perm(Xs) ! [ZjZs] ( select(Z; Xs; Ys) = true;*

*perm(Ys) = Zs:*

*perm([]) ! []:*

*select(X; [XjXs]; Xs) ! true:*

*select(X; [YjXs]; [YjZs]) ! true ( select(X; Xs; Zs) = true:*

*The overapproximation I+ is given by the following set of equations:*

*fsort([A j B]) = []]; sort([]) = []; select(X; [YjZ]; [Yj]]) = true;*

*select(X; [XjY]; Y) = true; perm([]) = []; perm([X j Y]) = []j]];*

*ord([X]) = true; ord([]) = true; select(X; Y; Z) = select(X; Y; Z);*

*X =< Y = X =< Y; [XjY] = [XjY]; [] = []; ord(X) = ord(X); true =*

*true; perm(X) = perm(X); sort(X) = sort(X)g*

*Now the system detects that the program rule sort([X]) ! [] is wrong.* If the programmer replaces it with the correct equation sort([X]) ! [X], the program becomes correct and complete according to our conditions.

# *8 Conclusions*

*We have presented an approach to declarative debugging of functional logic* programs w.r.t. the set of computed answers. We have de ned a declarative debugging method which has similarities to others which have been proposed

*in the literature (such as* [*[11*](#_bookmark24)[*,16]),*](#_bookmark29) *but which is original w.r.t. the de nition* and use of the semantic equations for making the error diagnosis and is useful for both eager as well as for lazy languages. We have presented a novel x- point semantics for functional logic programs, which is parametric w.r.t. the narrowing strategy. This semantics characterizes the set of computed answers in a bottom-up manner. Thus, it is a suitable basis for data ow analysis ba- sed on abstract interpretation. We present one example of abstraction of this

*xpoint semantics which yields an approximated nite (goal-independent) de-*

*scription of the success patterns of the program and which can then be used* in combination with our debugging equations to obtain an eÆcient and ter- minating debugging system. In this paper, we have discussed the successful experiments which have been performed with a prototypical implementation of our debugging system which is publicly available. We believe that it is pos- sible to extend our system in several ways; for instance, by integrating other data ow analysis for approximating term rewriting systems [[9].](#_bookmark23) Another ex- tension can be done by studying the relation and integration with assertion based methods for declarative debugging [[20](#_bookmark33)].

# *References*

*[1] M. Alpuente, F. J. Correa, M. Falaschi, and S. Marson. The Debugging System buggy. Technical report, UPV, 2001. Available at URL:* [*http://www.dsic.upv.es/users/elp/papers.html.*](http://www.dsic.upv.es/users/elp/papers.html)

*[2] M. Alpuente, M. Falaschi, and F. Manzo. Analyses of Unsatis ability for Equational Logic Programming. Journal of Logic Programming, 22(3):221{252,* *1995.*

*[3] M. Alpuente, M. Falaschi, and G. Vidal. A Compositional Semantic Basis for the Analysis of Equational Horn Programs. Theoretical Computer Science,* *165(1):97{131,* *1996.*

*[4] S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. Journal* *of the ACM, 47(4):776{822,* *2000.*

*[5] P. Arenas and A. Gil. A debugging model for lazy functional languages.* *Technical Report DIA 94/6, Universidad Complutense de Madrid, April 1994.*

*[6] F. Baader and T. Nipkow. Term Rewriting and All That. Cambridge University* *Press, 1998.*

*[7] D. Bert and R. Echahed. Design and implementation of a generic, logic and functional programming language. In Proc. of First European Symp. on* *Programming, ESOP'86, pages 119{132. Springer LNCS 213, 1986.*

*[8] D. Bert and R. Echahed. On the Operational Semantics of the Algebraic and Logic Programming Language LPG. In Recent Trends in Data Type Speci cations, pages 132{152. Springer LNCS 906, 1995.*

*[9] D. Bert, R. Echahed, and B.M. stvold. Abstract Rewriting. In Proc. of Third Int'l Workshop on Static Analysis, WSA'93, pages 178{192. Springer LNCS* *724, 1993.*

*[10] P. Bosco, E. Giovannetti, and C. Moiso. Narrowing vs. SLD-resolution.* *Theoretical Computer Science, 59:3{23,* *1988.*

*[11] F. Bueno, P. Deransart, W. Drabent, G. Ferrand, M Hermenegildo,*

*J. Maluszyn ski, and G. Puebla. On the role of semantic approximations in validation and diagnosis of constraint logic programs. In Proc. of the 3rd. Int'l Workshop on Automated Debugging-AADEBUG'97, pages 155{170. U. of* *Linkoping Press, 1997.*

*[12] R. Caballero-Rold an, F.J. Lopez-Fraguas, and M. Rodr quez Artalejo. Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs. In Fifth International Symposium on Functional and Logic Programming, Lecture Notes in Computer Science. Springer-Verlag, Berlin,* *2001. To appear.*

*[13] R. Caballero-Roldan, F.J. L opez-Fraguas, and J. Sanchez-Hern andez. User's* *manual for Toy. Technical Report SIP-5797, UCM, Madrid (Spain), 1997.*

*[14] M. Codish, M. Falaschi, and K. Marriott. Suspension Analysis for Concurrent Logic Programs. In K. Furukawa, editor, Proc. of Eighth Int'l Conf. on Logic* *Programming, pages 331{345. The MIT Press, Cambridge, MA, 1991.*

*[15] M. Comini, R. Gori, and G. Levi. Logic programs as speci cations in the inductive veri cation of logic programs. In Proceeding of Appia- Gulp-Prode'00, Joint Conference on Declarative Programming, 2000. URL:* [*http://nutella.di.unipi.it/~agp00/AccptList.html.*](http://nutella.di.unipi.it/~agp00/AccptList.html)

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

*[17] M. Comini, G. Levi, and G. Vitiello. Declarative Diagnosis Revisited. In John W. Lloyd, editor, Proceedings of the 1995 Int'l Symposium on Logic* *Programming, pages 275{287. The MIT Press, 1995.*

*[18] P. Cousot and R. Cousot. Abstract Interpretation: A Uni ed Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Proc. of Fourth ACM Symp. on Principles of Programming Languages, pages* *238{252,* *1977.*

*[19] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, Handbook of Theoretical Computer Science, volume B: Formal Models* *and Semantics, pages 243{320. Elsevier, Amsterdam, 1990.*

*[20] W. Drabent, S. Nadjim-Tehrani, and J. Maluszynski. The use of assertions in algorithmic debugging. In Proceedings of the 1988 Iinternational Conference on Fifth Generation Computer Systems, pages 573{581, Tokyo, Japan, December 1988.*

*[21] R. Echahed. On completeness of narrowing strategies. In Proc. of CAAP'88,* *pages 89{101. Springer LNCS 299, 1988.*

*[22] R. Echahed. Uniform narrowing strategies. In Proc. of ICALP'92, pages 259{*

*275. Springer LNCS 632, 1992.*

*[23] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. Theoretical Computer Science,* *69(3):289{318,* *1989.*

*[24] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. Information* *and Computation, 103(1):86{113,* *1993.*

*[25] G. Ferrand. Error Diagnosis in Logic Programming, and Adaptation of* *E.Y.Shapiro's Method. Journal of Logic Programming, 4(3):177{198,* *1987.*

*[26] L. Fribourg. SLOG: a logic programming language interpreter based on clausal superposition and rewriting. In Proc. of Second IEEE Int'l Symp. on Logic* *Programming, pages 172{185. IEEE, New York, 1985.*

*[27] E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel Leaf: A Logic plus Functional Language. Journal of Computer and System Sciences, 42:363{* *377, 1991.*

*[28] M. Hanus. Compiling Logic Programs with Equality. In Proc. of 2nd Int'l Workshop on Programming Language Implementation and Logic Programming,* *pages 387{401. Springer LNCS 456, 1990.*

*[29] M. Hanus. The Integration of Functions into Logic Programming: From Theory* *to Practice. Journal of Logic Programming, 19&20:583{628,* *1994.*

*[30] M. Hanus and B. Josephs. A debugging model for functional logic programs. In M. Bruynooghe and J. Penjam, editors, Proc. of 5th Int'l Symp. on Programming Language Implementation and Logic Programming, volume* *714*

*of Lecture Notes in Computer Science, pages 28{43. Springer, 1993.*

*[31] M. Hanus, H. Kuchen, and J.J. Moreno-Navarro. Curry: A Truly Functional Logic Language. In Proc. ILPS'95 Workshop on Visions for the Future of Logic* *Programming, pages 95{107, 1995.*

*[32] M. Hanus and C. Prehofer. Higher-Order Narrowing with De nitional Trees. In Proc. Seventh Int'l Conf. on Rewriting Techniques and Applications, RTA'96,* *pages 138{152. Springer LNCS 1103, 1996.*

*[33] S. Holldobler. Foundations of Equational Logic Programming. Springer LNAI* *353, 1989.*

*[34] J.W. Klop. Term Rewriting Systems. In S. Abramsky, D. Gabbay, and*

*T. Maibaum, editors, Handbook of Logic in Computer Science, volume I, pages 1{112. Oxford University Press, 1992.*

*[35] J. W. Lloyd. Programming in an integrated functional and logic language.* *Journal of Functional and Logic Programming, 3, 1999.*

*[36] J.W. Lloyd. Declarative Programming in Escher. Technical Report CSTR-95-* *013, Computer Science Department, University of Bristol, 1995.*

*[37] M.J. Maher. Complete Axiomatizations of the Algebras of Finite, Rational and In nite Trees. In Proc. of Third IEEE Symp. on Logic In Computer Science,* *pages 348{357. Computer Science Press, New York, 1988.*

*[38] M.J. Maher. On Parameterized Substitutions. Technical Report RC 16042,* *IBM - T.J. Watson Research Center, Yorktown Heights, NY, 1990.*

*[39] J.J. Moreno-Navarro and M. Rodr guez-Artalejo. Logic Programming with Functions and Predicates: The language Babel. Journal of Logic Programming,* *12(3):191{224,* *1992.*

*[40] L. Naish and T. Barbour. Towards a portable lazy functional declarative* *debugger. Australian Computer Science Communications, 18(1):401{408, 1996.*

*[41] Lee Naish. A declarative debugging scheme. Journal of Functional and Logic* *Programming, 1997(3), April 1997.*

*[42] U.S. Reddy. Narrowing as the Operational Semantics of Functional Languages. In Proc. of Second IEEE Int'l Symp. on Logic Programming, pages 138{151.* *IEEE, New York, 1985.*

*[43] E. Y. Shaphiro. Algorithmic Program Debugging. The MIT Press, Cambridge,* *Massachusetts, 1982. ACM Distinguished Dissertation.*

*[44] F. Zartmann. Denotational Abstract Interpretation of Functional Logic Programs. In P. Van Hentenryck, editor, Proc. of the 4th Int'l Static Analysis Symposium, SAS'97, pages 141{159. Springer LNCS 1302, 1997.*