A Declarative Debugger of Wrong Answers for Lazy Functional Logic Programs

Rafael Caballero and Mario Rodríguez-Artalejo *

e-mail: {rafa,mario}@sip.ucm.es
Departamento de Sistemas Informáticos y Programación,
Universidad Complutense de Madrid

Abstract. We describe the design of a declarative debugger for diagnosing wrong answers in lazy functional logic programs with polymorphic type discipline. Following a known technique, the debugger is based on a program transformation: transformed programs return the computation trees needed for debugging, in addition to the results expected by source programs. Thanks to a careful specification of the translation, we are able to prove its correctness w.r.t. well-typing and program semantics. As additional improvements w.r.t. related approaches, we solve a previously open problem concerning the use of curried functions, and we provide a correct method for avoiding redundant questions to the user during debugging. A prototype implementation of the debugger is available. Extensions of the debugger to deal with missing answers and constraints are planned as future work.

1 Introduction

Declarative debugging, first proposed in [14], is a general technique used to detect bugs on the basis of the intended meaning of the source program. It is particularly suitable when debugging declarative programs, since it allows disregarding operational details.

The key idea is to build a *computation tree* (CT) [8] representing the erroneous computation we intend to debug. Each node in the CT represents the outcome of a computation step and must be determined by the results at its children nodes. This tree is then traversed, looking for nodes erroneous w.r.t. the intended meaning of the program. To determine whether a node is erroneous, the debugger asks an external oracle (generally the user). However, detecting erroneous nodes is not enough: a result in a node can be erroneous due to erroneous children. An erroneous node with no erroneous children is called a *buggy node* following [8]. The debugger reports buggy nodes as sources of bugs in the program, since they produce an erroneous result from valid (i.e. non erroneous) incoming values.

From an explanatory point of view, declarative debugging can be seen as a two stage process. Following the terminology from [15], the two stages are called CT *generation* and CT *navigation*, respectively.

This paper describes the design of a declarative debugger of wrong answers in lazy functional logic programs with polymorphic type discipline. Following a

* Work partially supported by the Spanish CICYT (project CICYT-TIC98-0445-C03-02/97 "TREND")

known idea ([15, 11, 9, 13]), we use a program transformation for CT generation. We give a careful specification of the transformation and we show its advantages w.r.t. previous related ones. A prototype of the debugger for the lazy functional logic language $\mathcal{TOY}$ [6] has been implemented. The paper also describes some new techniques we have developed for avoiding redundant questions to the user during the navigation phase.

The rest of the paper is organized as follows: Sect. 2 collects some preliminaries, and Sect. 3 summarizes the contributions of our work w.r.t. previous related papers. Our approach to CT generation and navigation, with detailed explanations of the new contributions, are presented in Sect. 4 and 5, respectively. Our conclusions are in Sect. 6. Due to lack of space, proofs have been limited to brief sketches. Detailed proofs will be given in a full version of the paper.

2 Preliminaries

Functional Logic Programming (FLP for short) aims at the integration of the best features of current functional and logic languages; see [4] for a survey. This paper deals with declarative debugging for lazy FLP languages such as Curry or $\mathcal{TOY}$ [5, 6]. In this section we recall the basic facts about syntax, type discipline and declarative semantics for lazy FLP programs. We follow the formalization given in [3], but we use the concrete syntax of $\mathcal{TOY}$ for program examples.

2.1 Types, Expressions and Substitutions

We assume a countable set $TVar$ of *type variables* α, β, ... and a countable ranked alphabet $TC = \bigcup_{n\in\mathbb{N}} TC^n$ of *type constructors* C. The set $Type$ of valid types $\tau \in Type$ is built as $\tau ::= \alpha \quad (\alpha \in TVar) \mid (C\ \tau_1 \ldots \tau_n) \quad (C \in TC^n) \mid (\tau \rightarrow \tau')$. By convention, $C\ \overline{\tau}_n$ abbreviates $(C\ \tau_1 \ldots \tau_n)$, "$\rightarrow$" associates to the right, $\overline{\tau}_n \rightarrow \tau$ abbreviates $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \tau$, and the set of type variables occurring in τ is written $tvar(\tau)$. A type τ is called *monomorphic* iff $tvar(\tau) = \emptyset$, and *polymorphic* otherwise. A type without any occurrence of "$\rightarrow$" is called a *datatype*.

A *polymorphic signature* over TC is a triple $\Sigma = \langle TC, DC, FS\rangle$, where $DC = \bigcup_{n\in\mathbb{N}} DC^n$ and $FS = \bigcup_{n\in\mathbb{N}} FS^n$ are ranked sets of *data constructors* resp. *defined function symbols*. Each n-ary $c \in DC^n$ comes with a principal type declaration $c :: \overline{\tau}_n \rightarrow C\ \overline{\alpha}_k$, where $n, k \geq 0, \alpha_1, \ldots, \alpha_k$ are pairwise different, τ_i are datatypes, and $\mathrm{tvar}(\tau_i) \subseteq \{\alpha_1, \ldots, \alpha_k\}$ for all $1 \leq i \leq n$ (so-called *transparency property*). Also, every n-ary $f \in FS^n$ comes with a principal type declaration $f :: \overline{\tau}_n \rightarrow \tau$, where τ_i, τ are arbitrary types. In practice, each FLP program P has a signature which corresponds to the type declarations occurring in P. In the rest of this section we assume some fixed signature Σ, not made explicit in the notation.

Assuming a countable set Var of variables, disjoint from Σ, *partial expressions* $e \in Exp_\perp$ have the syntax $e ::= \perp \mid X \mid h \mid (e\,e')$ where $X \in Var,\ h \in DC \cup FS$. Expressions of the form $(e\,e')$ stand for the application of e (acting as a function) to e' (acting as an argument). As usual, we assume that application associates to the left and thus $(e_0\, e_1 \ldots e_n)$ abbreviates $((\ldots(e_0\, e_1)\ldots)\, e_n)$. The special symbol $\perp$ (read *bottom*) represents an undefined value. The set of data variables occurring in e is written $var(e)$. An expression e is called *closed* iff $var(e) = \emptyset$, and *open* otherwise. Moreover, e is called *linear* iff every $X \in var(e)$ has one single occurrence in e. *Partial patterns* $t \in Pat_\perp \subset Exp_\perp$ are built as $t ::= \perp \mid X \mid c\, t_1\, \ldots\, t_m \mid f\, t_1\, \ldots\, t_m$ where $c \in DC^n$, $0 \leq m \leq n$ and $f \in FS^n$, $0 \leq m < n$. Expressions and patterns without any occurrence of $\perp$ are called *total*. We write *Exp* and *Pat* for the sets of total expressions and patterns, respectively. Actually, the symbol $\perp$ never occurs in a program's text; but it may occur during debugging, as we will see.

An expression $e \in Exp_\perp$ is called *well-typed* iff there exist some *type environment* T (a set of type assumptions $X :: \tau$ for the variables occurring in e) and some type τ, such that the *type judgement* $T \vdash_{WT} e :: \tau$ can be derived by means of the type inference rules from Milner's type system. A well-typed expression always admits a so-called *principal type* (PT) that is more general than any other. A pattern whose PT determines the PTs of its subpatterns is called *transparent*. See [3] for more details.

Total substitutions are mappings $\theta : Var \rightarrow Pat$ with a unique extension $\hat{\theta} : Exp \rightarrow Exp$, which will be noted also as θ. The set of all substitutions is denoted as *Subst*. The set $Subst_\perp$ of all the *partial substitutions* $\theta : \mathcal{V} \rightarrow Pat_\perp$ is defined analogously. We write $e\theta$ for the result of applying the substitution θ to the expression e. As usual, $\theta = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ stands for the substitution that satisfies $X_i\theta \equiv t_i$, with $1 \leq i \leq n$ and $Y\theta \equiv Y$ for all $Y \in \mathcal{V}\backslash\{X_1, \ldots, X_n\}$. *Type substitutions*, mapping type variables to types, can be defined similarly.

2.2 Programs and Goals

A *well-typed program* P is a set of *defining rules* for the function symbols in its signature. Defining rules for $f \in FS^n$ have the form

$$(R) \quad \underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} \;\rightarrow\; \underbrace{r}_{\text{right-hand side}} \;\Leftarrow\; \underbrace{C}_{\text{condition}} \;\text{where}\; \underbrace{LD}_{\text{local definitions}}$$

satisfying the following requirements:

(i) $t_1 \ldots t_n$ is a linear sequence of transparent patterns and r any expression.
(ii) C is a sequence of conditions $e_1 == e'_1, \ldots e_k == e'_k$, where e_i, e'_i are expressions.
(iii) LD is a sequence of statements $s_1 \leftarrow d_1, \ldots, s_m \leftarrow d_m$, where d_i are expressions and s_i are transparent linear patterns. This is intended as a local, non-recursive definition of values for new variables occurring in $\overline{s}_m$. Therefore, we require that any variable in s_i must not occur in $t_1, \ldots t_n, s_1, \ldots s_{j-1}, s_{j+1}, \ldots s_m$, and can occur in d_j only if $j > i$.
(iv) There is some type environment T which well-types the left-hand and right-hand sides of the rule according to the principal type of f. Moreover, T must well-type each statement occurring in C or LD, by deriving a common type for its two sides.

```
                                        from :: nat -> [nat]
data nat        = z | suc nat           from N       = N:from (suc N)

head :: [A] -> A                        twice :: (A -> A) -> A -> A
head (X:Xs)     = X                     twice F X    = F (F X)

tail :: [A] -> [A]                      drop4 :: [A] -> [A]
tail (X:Xs)     = Xs                    drop4        = twice twice tail

plus :: nat -> nat -> nat               map :: (A -> B) -> [A] -> [B]
plus z Y        = Y                     map F []     = []
plus (suc X) Y  = suc (plus X Y)        map F (X:Xs) = (F X : map F Xs)

times :: nat -> nat -> nat
times z Y       = z
times (suc X) Y = plus X (times X Y) % Should be plus Y (times X Y)
```

Figure 2.2: An erroneous $\mathcal{TOY}$ program

Informally, the intended meaning of a rule like (R) is that a call to function f can be reduced to r whenever the actual parameters match the patterns t_i, and the conditions and local definitions are satisfied. A condition $e == e'$ is satisfied by evaluating e and e' to some common total pattern. A local definition $s \leftarrow d$ is satisfied by evaluating d to some possibly partial pattern which matches s. In the concrete syntax of $\mathcal{TOY}$, the symbol "=" is used in place of "$\rightarrow$" and "$\leftarrow$" within defining rules. Fig. 2.2 shows a small $\mathcal{TOY}$ program. In addition to `z` and `suc`, the signature includes also the predefined list constructors `[]` and `(:)` (used in infix notation). The defining rules, having neither conditions nor local definitions, are hopefully self-explanatory. The arity of each function equals the number of formal parameters in its rules. In particular, `drop4` (a function which eliminates the first four elements of a given list) has arity 0, in spite of its type. The second rule for `times` is *incorrect* w.r.t. the intended meaning of `times` as the multiplication operation.

Formally, the semantics of any program can be specified by means of the rewriting calculus GORC from [3]. There, conditions $e == e'$ are written as $e \bowtie e'$ and called *joinability statements*, while local definitions $s \leftarrow d$ are written as $d \rightarrow s$ and called *approximation statements*. In our previous paper [1] we have presented a simple variant of GORC (the *semantic calculus* SC) which can be used to build *Abbreviated Proof Trees* (APTs) [1]. Results in [1] ensure the correctness of declarative debugging using APTs as CTs. Nodes in APTs include *basic statements* of the form $f\,\overline{t_n} \rightarrow t$ (with $f \in FS^n$, t_i, t patterns), which become the questions asked to the oracle during the navigation phase of debugging. In this context, $f\,\overline{t_n} \rightarrow t$ is not interpreted as a local definition. Rather, it claims that t approximates the result returned by the function call $f\,\overline{t_n}$. In the sequel, we write $P \vdash \varphi$ to indicate that the statement φ can be proved from

[1] In [1] joinability statements were not considered for the sake of simplicity, and approximation statements were not used to represent local definitions in defining rules.

program P in the semantic calculus SC. As in [1], we also assume an *intended model* $\mathcal{I}$ represented as the set of basic statements which are viewed as valid by the oracle.

We consider *initial goals* of the form $G = e_1 == e'_1, \ldots, e_k == e'_k$. As in logic programming, goals can include *logic variables* that may become bound to patterns when the goal is solved. A solution for G is any total substitution θ such that $P \vdash G\theta$. A solution θ is *valid* iff $G\theta$ is valid in the intended model, and *erroneous* otherwise. Considering the program in Fig. 2.2, the goal `head (tail (map (times N) (from X))) == Y` asks for the second element of the infinite list that contains the product of `N` by the consecutive natural numbers starting at `X`. The $\mathcal{TOY}$ system first computes the valid solution $\theta = \{$`N` $\mapsto$ `z, Y` $\mapsto$ `z`$\}$, and next the erroneous solution $\theta' = \{$`N` $\mapsto$ `suc z, Y` $\mapsto$ `z`$\}$. The valid solution expected by the user ($\{$`N` $\mapsto$ `suc z, Y` $\mapsto$ `suc X`$\}$) is in fact a *missing answer*. Debugging missing answers is beyond the scope of this paper.

3 Problems and Contributions

In this section we summarize the main contributions of this paper to the two stages of declarative debugging, namely CT generation and CT navigation.

3.1 CT Generation

In the context of lazy FP and FLP, two main ways of constructing CT's have been proposed. The *program transformation* approach [15, 11, 9, 13] gives rise to transformed programs whose functions return CTs along with the originally expected results. The *abstract machine* approach [10, 15, 11, 12] requires low level modifications of the language implementation. Although the second approach can result in a better performance, we have adopted the first one because we find it more portable and better suited to a formal correctness analysis. With respect to other papers based in the transformational approach, we present two main contributions, described below.

Curried Functions: Roughly, all transformational approaches transform the functions defined in the source program to return pairs of type $(\tau, cTree)$, τ being the type of the originally expected result and `cTree` a datatype for representing CTs. Moreover, the types of functions acting as parameters of HO functions must be transformed accordingly. From the viewpoint of types, a n-ary curried function f is transformed into f' as follows:

$$f \ :: \ \tau_1 \ \rightarrow \ \cdots \ \rightarrow \tau_n \ \rightarrow \ \tau \quad \Rightarrow \quad f' \ :: \ \tau'_1 \ \rightarrow \ \cdots \ \rightarrow \tau'_n \ \rightarrow \ (\tau', \ cTree)$$

As explained in Sect. 2, n corresponds to the number of parameters expected by the rewrite rules in f's definition, but τ can be also a HO type or a variable. For instance, the types of the functions `plus`, `drop4` and `map` from Fig. 2.2 (with respective arities 2, 0 and 2) are translated as follows:

```
plus::nat → nat → nat      ⇒ plus'::nat → nat → (nat, cTree)
drop4::[A] → [A]           ⇒ drop4'::([A] → ([A], cTree),cTree)
map::(A → B) → [A] → [B] ⇒ map'::(A → (B,cTree)) → [A] → ([B],cTree)
```

As pointed out in [9, 13], the previous approach can lead to type errors when curried functions are used to compute results which are taken as parameters by other functions. For instance, `(map drop4)` is well-typed, but the naïve translation `(map' drop4')` is ill-typed, because the type of `drop4'` does not match the type expected by `map'` for its first parameter. More generally, the type of the result returned by f' when applied to m arguments depends on the relation between m and f's arity n. For example, `(map (plus z))` and `(map plus)` are both well-typed; when translating naïvely, `(map' (plus' z))` remains well-typed, but `(map' plus')` becomes ill-typed.

As a possible solution to this problem, the authors of [9] suggest to modify the translation in such a way that a curried function of arity $n > 0$ always returns a result of type $(\tau, cTree)$ when applied to its first parameter. According to this idea, `plus` would become `plus' :: nat → (nat → (nat, cTree), cTree)`.

However, as noted in [9], such a transformation would cause transformed programs to compute inefficiently, producing CTs with many useless nodes. Therefore, the authors of [9] wrote: *"An intermediate transformation which only handles currying when necessary is desirable. Whatever this can be done without detailed analysis of the program is under investigation"*. Our program transformation solves this problem by translating a curried function f of arity n, into n curried functions $f'_0, \ldots, f'_{n-2}, f'$ with respective arities 1, 2, $\ldots n-1$, n, and proper types. Function f'_m $(m \leq n-2)$ is used to translate occurrences of f applied to m parameters, while f' translates occurrences of f applied to $n-1$ or more parameters. For instance, `(map plus)` translates into (`map'` $\texttt{plus}_0{}'$), using the auxiliary function $\texttt{plus}_0{}'$ `:: nat → (nat → (nat,cTree), cTree)`.

We provide a similar solution to deal with partial application of curried data constructors, which can also cause type errors in the naïve approach (think of `(twice' suc)`, as an example). As far as we know, the difficulties with curried constructors have not been addressed previously. Our approach certainly increases the number of functions in transformed programs, but the extra functions are used only when needed, and inefficient CTs with useless nodes can be avoided. A full specification of the translation is presented in Sect. 4.

Correctness: Our program transformation is provably correct: it preserves polymorphic well-typing (modulo a type transformation) and program semantics (as formalized in [3, 1]). As we will see in Sect. 4, this correctness result holds independently of the narrowing strategy chosen as goal solving mechanism. To the best of our knowledge, previous related papers [15, 11, 9, 13] give no correctness proof for the program transformation. The author of [13], who is aware of the problem, just relies on intuition for the semantic correctness. He mentions the need of a formalized semantics for a rigorous proof. As for type correctness, it is closely related to the treatment of curried functions, which was deficient in previous approaches.

CT Navigation: One important aspect w.r.t. the practical use of declarative debuggers is the number of questions asked to the oracle, which should be as reduced as possible. Our debugger uses a decidable and semantically correct entailment between facts to maintain a consistent and non-redundant store of facts known from previously answered questions. Redundant questions whose answer is entailed by stored facts are avoided. Details are explained in Sect. 5.

4 Generation of CTs by Program Transformation

In this section we present the program transformation used by our debugger and we discuss its correctness. Roughly, a program P is converted into a new program P', where function calls return the same results P would return, but paired with CTs. Formally, P' is obtained by transforming the signature Σ of P into a new signature Σ', introducing definitions for certain auxiliary functions, and transforming the function definitions included in P. Let us consider these issues one by one.

4.1 Representing Computation Trees

A transformed program always includes the constructors of the datatype cTree, used to represent CTs and defined as follows:

```
data cTree              = void | cNode funId [arg] res rule [cTree]
type arg, res           = pVal
type funId, pVal, rule = string
```

A CT of the form `(cNode f ts t rl cts)` corresponds to a call to the function `f` with arguments `ts` and result `t`, where `rl` indicates the function rule used to evaluate the call, and the list `cts` consists of the children CTs corresponding to all the function calls (in the local definitions, right-hand side and conditions of `rl`) whose activation was needed in order to obtain `t`. Due to lazy evaluation, the main computation may demand only partial approximations of the results of intermediate computations. Therefore, `ts` and `t` stand for possibly *partial values*, represented as patterns; and `(f ts → t)` represents the *basic fact* whose validity will be asked to the oracle during debugging, as explained in Sect. 2. As for `void`, it represents an empty CT, returned by calls to functions which are trusted to be not buggy (in particular, data constructors and the auxiliary functions introduced by the translation). Finally, the definition of `arg`, `res`, `funId`, `pVal` and `rule` as synonyms of the type of character strings is just a simple representation, currently used by our prototype debugger.

4.2 Transforming Program Signatures

For every n-ary function $f \ :: \ \tau_1 \to \ldots \to \tau_n \to \tau$ occurring in P, P' must include an $(m+1)$-ary auxiliary function f'_m for each $0 \leq m < n-1$, as well as an n-ary function f', with principal types:

$$f'_m \ :: \ \tau'_1 \to \ldots \to \tau'_{m+1} \to ((\tau_{m+2} \to \ldots \to \tau_n \to \tau)', \ cTree) \quad \text{for } 0 \leq m < n-1$$
$$f' \ :: \ \tau'_1 \to \ldots \to \tau'_n \to (\tau', \ cTree)$$

Similarly, for each n-ary constructor $c \ :: \ \tau_1 \to \ldots \to \tau_n \to \tau$ occurring in P, P' must keep c with the same principal type, and include new $(m+1)$-ary auxiliary functions c'_m $(0 \leq m < n)$, with principal types:

$$c'_m \ :: \ \tau'_1 \to \ldots \to \tau'_{m+1} \to ((\tau_{m+2} \to \ldots \to \tau_n \to \tau)', \ cTree) \qquad \text{for } 0 \leq m < n$$

The type declarations above depend on a *type transformation* which converts any type τ in P's signature into a transformed type τ'. This is defined recursively:

$$\alpha' = \alpha \ (\alpha \in TVar) \qquad (C \ \overline{\tau}_n)' = C \ \overline{\tau}'_n \ (C \in TC^n) \qquad (\mu \to \nu)' = \mu' \to (\nu', cTree)$$

Finally, P' always includes the auxiliary functions `dVal` and `clean`, whose types and definitions will be described below.

4.3 Defining Auxiliary Functions

Each auxiliary function f'_m expects $m+1$ arguments and returns a partial application of f'_{m+1} paired with a trivial CT. Exceptionally, f'_{n-2} returns a partial application of f'. The auxiliary functions c'_m are defined similarly, except that c'_{n-1} returns a value built with the data constructor c.

$$f'_0\,X_1 = (f'_1\,X_1, void) \quad f'_1\,X_1\,X_2 = (f'_2\,X_1\,X_2, void) \ldots f'_{n-2}\,\overline{X}_{n-1} = (f'\;\overline{X}_{n-1}, void)$$
$$c'_0\,X_1 = (c'_1\,X_1, void) \quad c'_1\,X_1\,X_2 = (c'_2\,X_1\,X_2, void) \ldots\; c'_{n-1}\,\overline{X}_{n-1} = (c\;\overline{X}_{n-1}, void)$$

4.4 Transforming Function Definitions

Each defining rule $f\ t_1\ \ldots\ t_n \rightarrow r \Leftarrow C\,where\,LD$ occurring in P is transformed into a corresponding defining rule for f' in P'. Assuming that LD consists of local definitions $s_j \leftarrow d_j$ and C consists of conditions $l_i == r_i$, the transformed defining rule is constructed as

$$\begin{array}{lll}
f'\ t'_1 \ldots t'_n \rightarrow (R,T) \Leftarrow \ldots CL_i == CR_i \ldots & & \\
\quad \text{where}\{ & & \ldots \\
 & s'_j & \leftarrow d'_j; \\
 & & \ldots \\
 & CL_i & \leftarrow l'_i; \\
 & CR_i & \leftarrow r'_i; \\
 & & \ldots \\
 & R & \leftarrow r'; \\
 & T & \leftarrow \text{cNode}\ "f"\ [dVal\,t'_1, \ldots dVal\,t'_n]\ (dVal\ R)\ "f.j"\ (clean\,[\,])\} \downarrow
\end{array}$$

Some additional explanations are needed at this point:

- t'_k, s'_j, d'_j, l'_i, r'_i and r' refer to an expression transformation (defined below) which converts any $e\ ::\ \tau$ of signature Σ into $e'\ ::\ \tau'$ of signature Σ'.
- R, T, CL_i, CR_i are new fresh variables, and j is the number of the rule.
-The notation $\{\ldots\}\downarrow$ refers to a transformation of the local definitions explained below.
-`dVal :: A` $\rightarrow$ `pVal` is an auxiliary impure function without declarative meaning, very similar to `dirt` in [9, 13]. Any call `(dVal a)` (read: "demanded value of `a`") returns a string representing the partial approximation of `a`'s value which was needed to complete the top level computation. The debugger's implementation can compute this from the internal structure representing `a` at the end of the main computation, replacing all occurrences of suspended function calls by `"_"`, which represents the undefined value $\perp$ [2]. Moreover, `dVal` also renames all the identifiers of auxiliary functions f'_m resp. c'_m into f resp. c. In this way, the results computed by the transformed program are translated back to the original signature.

The expression transformation $e \mapsto e'$ is defined by recursion on e's syntactic structure. The idea is to transform the (possibly partial) applications of functions and constructors within e, using functions from the transformed signature. In order to ensure $e'\ ::\ \tau'$ whenever $e\ ::\ \tau$, we use two auxiliary *application operators*:

[2] Because of this replacement of $\perp$ in place of unknown values, nodes in proof trees ultimately include *approximation statements* rather than *equalities*

$$@_0 \ :: (\beta,\texttt{cTree}) \rightarrow \beta \qquad\qquad (@) \ :: (\alpha \rightarrow (\beta,\texttt{cTree})) \rightarrow \alpha \rightarrow \beta$$
$$@_0 \ \texttt{F} \rightarrow \texttt{R where } \{(\texttt{R},\texttt{T}) \leftarrow \texttt{F}\} \qquad \texttt{F} \ @ \ \texttt{X} \rightarrow \texttt{R where } \{(\texttt{R},\texttt{T}) \leftarrow \texttt{F X}\}$$

These are used within e' at those points where the application of a function from the translated signature is expected to return a value paired with a CT. Applications of higher-order variables are treated in a similar way. Formally:

$$\begin{array}{ll}
(X\ a_1\ \ldots\ a_k)' = (\ldots(X @ a'_1)@\ \ldots)@a'_k & (X \in Var,\ k \geq 0)\\
(c\ e_1\ \ldots\ e_m)' \ = c_m\, e'_1 \ldots e'_m & (c \in DC^n,\ m < n, n > 0)\\
(c\ e_1\ \ldots\ e_n)' \ = c\, e'_1 \ldots e'_n & (c \in DC^n, n \geq 0)\\
(f\ a_1\ \ldots\ a_k)' \ = (\ldots((@_0\, f)@\, a'_1)@\ \ldots)@a'_k & (f \in FS^0,\ k \geq 0)\\
(f\ e_1\ \ldots\ e_m)' \ = f_m\, e'_1 \ldots e'_m & (f \in FS^n, n > 0, m < n-1)\\
(f\ e_1\ \ldots\ e_{n-1}\, a_1 \ldots\ a_k)' = (\ldots((f\ e'_1 \ldots e'_{n-1})\, @\, a'_1)\ @\ \ldots)@a'_k & \\
\qquad\qquad\qquad\qquad\qquad\qquad (f \in FS^n, n > 0, k \geq 0) &
\end{array}$$

Looking back to the construction of translated defining rules, we see that the translated expressions t'_k, s'_j, d'_j, l'_i, r'_i and r' are intended to ensure well-typing, but ignore CTs. In particular, the local definition of T renders a CT whose root has the proper form, but whose children are not yet defined. In order to complete the translation, the translated local definitions $\{\ldots\}$ are transformed into $\{\ldots\}\downarrow$, which means the normal form obtained by applying the transformation rules AP_0 and AP_1 defined below, with a leftmost-innermost strategy.

- AP_0:

$\{\ldots;\ t \leftarrow e[@_0\ g];\ldots T \leftarrow \text{cNode}\ldots(clean\ lp)\} \qquad \longrightarrow$
$\{\ldots;\ (R',T') \leftarrow g;\ t \leftarrow e[R'];\ldots T \leftarrow \text{cNode}\ldots(clean\ (lp +\!\!+ [(\text{dVal}\ \ R',T')]))\}$

- AP_1:

$\{\ldots;\ t \leftarrow e[a\ @\ s];\ldots T \leftarrow \text{cNode}\ldots(clean\ lp)\} \qquad \longrightarrow$
$\{\ldots;\ (R',T') \leftarrow a\, s;\ t \leftarrow e[R'];\ldots T \leftarrow \text{cNode}\ldots(clean\ (lp +\!\!+ [(\text{dVal}\ \ R',T')]))\}$

In both transformations, ``++'' stands for the list concatenation function. R' and T' must be chosen as new fresh variables, and t is the pattern in the lefthand side of a local definition whose righthand side includes a leftmost-innermost occurrence of an application operator $(@_0\ g)$ or $(a\ @\ s)$ in some context. Because of the leftmost-innermost strategy, we can claim:

- AP_0 always finds a nullary function symbol in place of g.
- AP_1 always finds a pattern in place of s, and either a variable or a pattern of the form $g\, t_1 \ldots t_{n-1}$ (with $g \in FS'^n$, $n > 0$) in place of a.

Each application of the AP transformations eliminates the currently leftmost-innermost occurrence of an application operator, while introducing a local definition for the result R' and CT T' coming from that application, and adding the pair $(\text{dVal}\ \ R',T')$ to the list of children of T. When the AP transformations terminate, no application operators remain. Therefore, $@_0$ and $@$ do not occur in transformed programs. All the occurrences of ``++'' within the righthand side of T's local definition can be removed, according to the usual definition of list concatenation. This leads to a list `lp :: [(pVal, cTree)]` including as many CTs as application operators did occur in the local definitions, each of them paired with a partial result. Finally, `(clean lp)` builds the ultimate list of children CTs, by ignoring those pairs `(pv,ct)` in `lp` such that `ct` is void or `pv` represents $\perp$. The auxiliary function `clean :: [(pVal, cTree)]` $\rightarrow$ `cTree` (whose simple definition is omitted here) must be included in any transformed program.

4.5 An Example

Transforming the $\mathcal{TOY}$ program from Fig. 2.2 would produce, among others, the function definitions shown below. The concrete syntax of $\mathcal{TOY}$ is used here.

```
twice'      :: (A -> (A, cTree)) -> A -> (A, cTree)
twice' F X  = (R,T)
    where {(R1,T1) = F X;
           (R2,T2) = F R1;
           R       = R2;
           T       = cNode "twice" [dVal F, dVal X] (dVal R) "twice.1"
                             (clean [(dVal R1,T1), (dVal R2,T2)]) }

drop4' :: ([nat] -> ([nat], cTree), cTree)
drop4'  = (R,T)
    where {(R1,T1) = twice' twice_0' tail';
            R      = R1;
            T      = cNode "drop4" [] (dVal R) "drop4.1"
                             (clean [(dVal R1,T1)])}
```

4.6 Transforming Goals

The debugging process can be started whenever some goal G is solved with an erroneous answer θ. In order to build a suitable CT for the goal, a new function definition `solution` $\overline{X_n}$ `= true` $\Leftarrow G$ is considered, whose translation is added to the transformed program. Here, $\overline{X_n}$ are the variables occurring in G. Since θ is a solution for G, the goal `solution` $\overline{X_n}\theta$ `== (true, Tree)` can be solved by the translated program, without instantiating any free variable in $\overline{X_n}\theta$, and instantiating `Tree` to a CT with erroneous root. The navigation phase of the debugger proceeds with this CT.

4.7 Correctness Results

We can prove three main results about the correctness of our program transformation. The first result concerns the type discipline. Thanks to it, the debugger does not need to perform any type checking/inference before entering the CT generation phase.

Theorem 1 The translation P' of a well-typed program P is always well-typed.

Proof Idea. Assuming an expression e in P's signature and a type environment T such that $T \vdash_{WT} e :: \tau$, one can prove $T' \vdash_{WT} e' :: \tau'$ for $T' =_{def} \{X_i :: \tau_i' \mid (X_i :: \tau_i) \in T\}$. Using this result, and reasoning by induction on the number of AP steps involved in the transformation of defining rules, it can be proved that any well-typed defining rule for a function f in P is transformed into a well-typed defining rule for f' in P'. On the other hand, the defining rules for auxiliary functions occurring in P' are obviously well-typed. $\square$

The second result says that the translation preserves the semantics of source programs, enhanced by the additional computation of CTs. Recall the notation $P \vdash f\ \overline{t_n} \to t$ explained in Sect. 2, which can be used also for transformed programs. Intuitively, a basic fact $f\ \overline{t_n} \to t$ asserts that the function call $f\ \overline{t_n}$

can return a partial result t according to the semantics of program P. Note also that the translation of a pattern t, following the definition from Subsection 4.4, is always a pattern t' from which t can be univocally recovered.

Theorem 2 For any n-ary function f and arbitrary partial patterns $\overline{t_n}$, t in the signature of a program P, it holds:

1. If $P' \vdash f' \overline{t'_n} \to (t', ct)$ then $P \vdash f \overline{t_n} \to t$.
2. If $P \vdash f \overline{t_n} \to t$ then there is some pattern ct in P''s signature, which represents an abbreviated proof tree proving $P \vdash f \overline{t_n} \to t$ in the sense of [1], and such that $P' \vdash f' \overline{t'_n} \to (t', ct)$.

Proof Idea. Abbreviated Proof Trees (APTs) were introduced in [1] to formalize proofs of basic facts in a semantic calculus SC. Due to the form of the inference rules of SC, it can be checked that an APT T which proves $P \vdash f \overline{t_n} \to t$ must be built by using some particular instance of defining rule for f of the form $f \overline{t_n} \to r \Leftarrow C\ where\ LD$. Moreover, the root of T must contain $f \overline{t_n} \to t$, and the children must be a list of APTs corresponding to SC proofs which justify the local definitions in LD, the conditions in C and the statement $r \to t$. Moreover, a careful analysis of SC's inference rules shows that these children CTs must correspond to function applications which return a partial result different from $\perp$, taken in leftmost-innermost order. The definition of transformed defining rules in P' has been designed to build precisely such CTs. □

Finally, as a consequence of the previous result, the generated CT provides a correct basis for the navigation phase:

Theorem 3 Assume a program P, a wrong answer θ for a goal G, and a CT T obtained by running a transformed program, as explained in Sect. 4.6. Then T has at least one buggy node, which includes an indication of a semantically incorrect instance of a defining rule from P.

Proof. Because of Theorem 2, T is an APT. The Correctness Theorem from [1] ensures the following: *any* APT whose root is erroneous always includes some buggy node whose associated program rule instance is not valid in the intended model. □

We would like to stress the fact that Theorem 3 holds independently of the narrowing strategy implemented by the system used to run the transformed program. Although a particular narrowing calculus was proposed in [1] to formalize the construction of APTs, *all* APTs (in particular, those computed by transformed programs) are correct CTs for debugging. Of course, the narrowing strategy affects the order in which eventual wrong answers are computed.

5 Navigating the CTs by Oracle Querying

Once the CT associated to a wrong answer has been built (as described in Subsection 4.6), navigation performs a top-down traversal, asking the oracle about the validity of the *basic facts* associated to the visited nodes (except the root, which is known to be erroneous in advance). For the sake of practical usefulness, it is important to ensure that questions asked to the oracle are as few and as simple as possible. The second condition - simplicity - comes along with our choice of APTs as CTs, since basic facts are the minimal pieces of information needed to characterize the intended model of a program (see [1]). To reduce the number of questions, the only possibility considered in related papers is to avoid asking repeated questions. As an improvement, we present an *entailment* relation between basic facts, and we show that it can be used to avoid redundant questions which can be deduced from previous answers.

Our notion of entailment is based on the *approximation ordering* $\sqsubseteq$, defined as the least partial order over the set $Pat_\perp$ of partial patterns which satisfies:

- $\perp \sqsubseteq t$, t any partial pattern.
- $X \sqsubseteq X$, X any variable.
- $h\ \overline{t}_n \sqsubseteq h\ \overline{s}_n$, if $t_1 \sqsubseteq s_1\ \ldots\ t_n \sqsubseteq s_n$, with $h \in DC^m$, $n \leq m$ or $h \in FS^m$, $n < m$.

This ordering has a natural semantic interpretation: $t \sqsubseteq t'$ means that t' has at least so much information as t. For instance: `(z : ⊥)` $\sqsubseteq$ `(z : suc z : ⊥)`. Using $\sqsubseteq$, we define: A basic fact $f\ \overline{t}_n \to t$ *entails* another basic fact $f\ \overline{s}_n \to s$ (written as $f\ \overline{t}_n \to t\ \succeq\ f\ \overline{s}_n \to s$) iff there is some total substitution $\theta \in Subst$ such that $t_1\theta \sqsubseteq s_1, \ \ldots\ , t_n\theta \sqsubseteq s_n, s \sqsubseteq t\theta$.

The following result justifies the interest of entailment, since the answers given by any oracle in declarative debugging are assumed to be valid in the intended model of the program.

Theorem 4 Entailment between basic facts is a decidable preorder (i.e, reflexive and transitive relation). Moreover, for any set $\mathcal{I}$ of basic statements which represents an intended model: *if* $f\ \overline{t}_n \to t\ \succeq\ f\ \overline{s}_n \to s$ *and* $f\ \overline{t}_n \to t\ \in \mathcal{I}$ *then* $f\ \overline{s}_n \to s \in \mathcal{I}$.

Proof Idea To prove decidability of entailment, we have defined a system of transformations, somewhat similar to those used in Martelli and Montanari's unification algorithm. These transformations compute the needed substitution θ whenever entailment holds, and they fail otherwise. The fact that entailment is a preorder is easy to check. The last claim of the Theorem follows easily from the formal definition of intended model given in [1]. $\square$

Thanks to Theorem 4, a question Q entailed by a fact already known to be valid because of some previous answer, must be valid. For instance, if we already know that `from X → X:(suc X):[]` is valid then other basic facts entailed by this one, such as `from z → z:⊥` and `from (suc Y) → (suc Y):(suc (suc Y)):[]`, must also be valid. Dually, a question Q which entails a fact known to be invalid because of some previous answer, must be invalid. In both cases, a question to the oracle can be avoided. For instance, if we know from a previous answer that `from z → (suc z):⊥` is not valid, then other basic facts that entail this one, such as `from X → (suc X):(suc (suc X)):[]` must be also invalid.

The debugger has been implemented as part of the $\mathcal{TOY}$ system. A prototype version can be downloaded from `http://titan.sip.ucm.es/toy/debug.tar.gz`. The implementation of entailment is still ongoing work. Currently the debugger just avoids asking a question which is a *variant* of some previous one.

Here we show a debugging session for our example program of Fig. 2.2. The user activates the debugger because the incorrect answer {`N↦ suc z, Y↦ z`} has been computed for the goal `head (tail ( map (times N) (from X))) == Y`:

```
Is (from X → (X:(suc X):⊥)) valid? y
Is (map (times (suc z)) (X:(suc X):⊥) → (⊥:z:⊥)) valid? n
Is (map (times (suc z)) ((suc X):⊥)) → (z:⊥)) valid? n
Is (times (suc z) (suc X) → z) valid? n
Is (times z (suc X) → z) valid? y
Is (plus z z → z) valid? y
Rule number 2 of the function times is wrong.
Wrong instance: times (suc z) (suc X) → (plus z (times z (suc X)))
```

The debugger finds out an incorrect program rule after asking six questions to the user. Notice that in our implementation not only the number of the wrong rule is displayed, but also the incorrect *instance* used during the computation. This instance can be sometimes helpful when looking for the bug in the rule. Our implementation also avoids to ask questions about predefined functions (e.g. arithmetic operations), since they are trusted to be correct. Allowing the user to annotate certain functions to be trusted as correct is a simple albeit useful extension, not yet implemented.

6 Conclusions and Future Work

Program transformation is a known approach to the implementation of declarative debugging of wrong answers in lazy functional (and logic) languages [15, 11, 9, 13]. We have given a new, more formal specification of this technique, which brings two main advantages w.r.t. previous related work. Firstly, our approach avoids type errors related to the use of curried functions, which were not handled properly up to now. Secondly, our program transformation is provably correct: it preserves polymorphic well-typing (modulo a type transformation) and program semantics (as formalized in our [3, 1]). This correctness result holds independently of the narrowing strategy chosen as goal solving mechanism. A prototype implementation of our debugger for the functional logic language $\mathcal{TOY}$ [6] has been implemented. In a debugging session, the debugger navigates through a computation tree, previously generated by execution of a transformed program. In order to alleviate the complexity of the navigation phase, we have developed a semantically correct algorithm which detects and avoids redundant questions to the oracle.

In the near future, we plan to transfer our prototype debugger to the language Curry [5]. We also plan to investigate and implement extensions of the debugger, to deal with missing answers and constraint-based computations. Most existing work on these issues, as e.g. [7, 16], refers to LP and CLP languages. Eventually, we would like to compare our approach with *abstract diagnosis* techniques [2], in order to evaluate their respective pros and cons.

Acknowledgements We are grateful to Paco López, who has followed our work with interest and provided useful advice. We are also very indebted to Mercedes Abengózar for her great help with implementation work.

References

1. R. Caballero, F.J. López-Fraguas and M. Rodríguez-Artalejo. *Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs.* In Proceedings of the 5th International Symposium on Functional and Logic Programming, Springer LNCS 2024, 170–184, 2001.
2. M. Comini, G. Levi, M.C. Meo and G. Vitello. *Abstract Diagnosis.* J. of Logic Programming 39, 43–93, 1999.
3. J.C. González-Moreno, M.T. Hortalá-González and M. Rodríguez-Artalejo. *Polymorphic Types in Functional Logic Programming.* To appear in the FLOPS'1999 special issue of the Journal of Functional and Logic Programming, 2001. Preliminary version published in the Proceedings of the 5th International Symposium on Functional and Logic Programming, Springer LNCS 1722, 1–20, 1999.
4. M. Hanus. *The Integration of Functions into Logic Programming: A Survey.* J. of Logic Programming 19-20. Special issue "*Ten Years of Logic Programming*", 583–628, 1994.
5. M. Hanus (ed.), *Curry: An Integrated Functional Logic Language*, Version 0.7, February 2, 2000. Available at `http://www.informatik.uni-kiel.de/~curry/`.
6. F.J. López-Fraguas, J. Sánchez-Hernández. *$\mathcal{TOY}$: A Multiparadigm Declarative System*, in Proc. RTA'99, Springer LNCS 1631, pp 244–247, 1999. Available at `http://mozart.sip.ucm.es/toy`.
7. L. Naish. *Declarative Diagnossing of Missing Answers.* New Generation Computing, 10, 255–385, 1991.
8. L. Naish. *A Declarative Debugging Scheme.* Journal of Functional and Logic Programming, 1997-3.
9. L. Naish, and T. Barbour. *Towards a Portable Lazy Functional Declarative Debugger.* Australian Computer Science Communications, 18(1):401–408, 1996.
10. H. Nilsson, P. Fritzson. *Algorithmic Debugging of Lazy Funcional Languages.* The Journal of Functional Programming, 4(3):337-370, 1994.
11. H. Nilsson and J. Sparud. *The Evaluation Dependence Tree as a basis for Lazy Functional Debugging.* Automated Software Engineering, 4(2):121–150, 1997.
12. H. Nilsson. *Declarative Debugging for Lazy Functional Languages.* Ph.D. Thesis. Dissertation No. 530. Univ. Linköping, Sweden. 1998.
13. B. Pope. *Buddha. A Declarative Debugger for Haskell.* Honours Thesis, Department of Computer Science, University of Melbourne, Australia, June 1998.
14. E.Y. Shapiro. *Algorithmic Program Debugging.* The MIT Press, Cambridge, Mass., 1982.
15. J. Sparud and H. Nilsson. *The Architecture of a Debugger for Lazy Functional Languages.* In M. Ducassé, editor, Proceedings of AADEBUG'95, Saint-Malo, France, 1995.
16. A. Tessier and G. Ferrand. *Declarative Diagnosis in the CLP Scheme.* In P. Deransart, M. Hermenegildo and J. Małuszynski (Eds.), *Analysis and Visualization Tools for Constraint Programming*, Chapter 5, pp. 151–174. Springer LNCS 1870, 2000.