Electronic Notes in Theoretical Computer Science 64 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume64.html 63 pages

A Declarative Debugging System for Lazy Functional Logic Programs

Rafael Caballero [1,2] Mario Rodríguez-Artalejo [1,3]

Dep. Sistemas Informáticos y Programación
Universidad Complutense Madrid
Madrid, Spain

Abstract

We present a declarative debugger for lazy functional logic programs with polymorphic type discipline. Whenever a computed answer is considered wrong by the user (error symptom), the debugger locates a program fragment (function defining rule) responsible for the error. The notions of *symptom* and *error* have a declarative meaning w.r.t. to an intended program semantics. Debugging is performed by searching in a computation tree which is a logical representation of the computation. Following a known technique, our tool is based on a program transformation: transformed programs return computation trees along with the results expected by source programs. Our transformation is provably correct 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. Case studies and extensions are planned as future work.

1 Introduction

The impact of declarative languages on practical applications is inhibited by many known factors, including the lack of *debugging tools*, whose construction is recognized as difficult for lazy functional languages. As argued in [29], such debuggers are needed, and much of interest can still be learned from their construction and use. Debugging tools for lazy functional logic languages [11] are even harder to construct.

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

[2] Email: rafa@sip.ucm.es

[3] Email: mario@sip.ucm.es

A promising approach is *declarative debugging*, which starts from a computation considered incorrect by the user (error symptom) and locates a program fragment responsible for the error. In the case of (constraint) logic programs, error symptoms can be either *wrong* or *missing* computed answers [26,13,6,17,28]. Declarative debugging has been also adapted to lazy functional programming [21,22,23,27,18,20,25] and combined functional logic programming [19]. All these approaches use a *computation tree* (CT) [19] as logical representation of the computation. Each node in a CT represents the result of a computation step, which must follow from the results of its children nodes by some logical inference. Diagnosis proceeds by traversing the CT, asking questions to an external oracle (generally the user) until a so-called *buggy node* is found, whose result is erroneous, but whose children have all correct results. The user does not need to understand the computation operationally. Any buggy node represents an erroneous computation step, an the debugger can display the program fragment responsible for it. From an explanatory point of view, declarative debugging can be described as consisting of two stages, namely CT *generation* and CT *navigation* [22].

We present a declarative debugger of wrong answers for lazy functional logic programs with polymorphic type discipline. Following a known idea [22,20,25], we use a program transformation for CT generation. We give a careful specification of the transformation, we show its advantages w.r.t. previous related ones, and we describe some new techniques which allow to avoid redundant questions to the oracle during the navigation phase.The debugger has been implemented as part of the $\mathcal{TOY}$ system [14]; a prototype version can be downloaded from `http://titan.sip.ucm.es/toy/`. Case studies and extensions of the debugger are planned as future work.

A known extension of declarative debugging is *abstract diagnosis* [3,1], leading to equivalent bottom-up and top-down diagnosis methods which do not require error symptoms to be given in advance. In order to be effectively implemented, abstract diagnosis uses abstract interpretation techniques to build a finite abstraction of the intended program semantics. These methods are outside the scope of this paper.

The rest of the paper is organized as follows: Section 2 recalls preliminary notions and previous results about functional logic programming and declarative debugging. Section 3 summarizes our new contributions w.r.t. previous related works. Our approaches to CT generation and navigation, with detailed explanations of the new contributions, are presented in Sections 4 and 5, respectively. Conclusions and plans for future work are summarized in Section 6. Detailed proofs of the main results are included in the Appendix A, while Appendix B includes some simple debugging sessions.

2 Preliminaries

Functional Logic Programming (FLP for short) aims at the integration of the best features of current functional and logic languages; see [11] for a survey. This paper deals with declarative debugging for lazy FLP languages such as Curry or $\mathcal{TOY}$ [12,14], which include pure LP and lazy FP programs as particular cases. 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 [9], but using the concrete syntax of $\mathcal{TOY}$ for program examples.

2.1 Types, Expressions and Substitutions

2.1.1 Types and Signatures

We assume a countable set $TVar$ of *type variables* $\alpha,\ \beta,\ \dots$ and a countable ranked alphabet $TC = \bigcup_{n\in\mathbb{N}} TC^n$ of *type constructors* C. Types $\tau \in Type$ have the syntax

$$\tau ::= \alpha\ \ (\alpha \in TVar) \mid (C\ \tau_1 \dots \tau_n)\ \ (C \in TC^n) \mid (\tau \rightarrow \tau') \mid (\tau_1, \dots, \tau_n)$$

By convention, $C\ \overline{\tau}_n$ abbreviates $(C\ \tau_1 \dots \tau_n)$, "$\rightarrow$" 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 $n, k \geq 0, \alpha_1, \dots, \alpha_k$ are pairwise different, τ_i are datatypes, and $tvar(\tau_i) \subseteq \{\alpha_1,\dots,\ \alpha_k\}$ 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. For any signature Σ, we write $\Sigma_\perp$ for the result of extending Σ with a new data constructor $\perp\ ::\ \alpha$, intended to represent an undefined value that belongs to every type. As notational conventions, we use $c, d \in DC$, $f, g \in FS$ and $h \in DC \cup FS$, and we define the *arity* of $h \in DC^n \cup FS^n$ as $ar(h) = n$.

2.1.2 Expressions and Patterns

In the sequel, we always suppose a given signature Σ, often not made explicit in the notation. Assuming a countable set Var of (data) variables $X, Y, \dots$ disjoint from $TVar$ and Σ, *partial expressions* $e \in Exp_\perp$ have the syntax

$$e ::= \perp\ \mid X \mid h \mid (e\ e') \mid (e_1,\ \dots,\ e_n)$$

where $X \in Var$, $h \in DC \cup FS$. Expressions of the form $(e\,e')$ stand for the application of expression e (acting as a function) to expression e' (acting as an argument), while expressions $(e_1, \ldots, e_n)$ represent tuples with n components. 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 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

$$t ::= \perp \;\mid X \mid c\; t_1 \ldots t_m \mid f\; t_1 \ldots t_m \mid (t_1, \ldots, t_n)$$

where $X \in Var$, $c \in DC^n$, $0 \leq m \leq n$, $f \in FS^n$, $0 \leq m < n$ and t_i partial patterns for all $1 \leq i \leq m$. They represent *approximations* of the values of expressions. Following the spirit of denotational semantics [10], we view $Pat_\perp$ as the set of finite elements of a semantic domain, and we define the *approximation ordering* $\sqsubseteq$ as the least partial ordering over $Pat_\perp$ satisfying the following properties:

- $\perp \sqsubseteq t$, for all $t \in Pat_\perp$.
- $h\;\overline{t}_m \sqsubseteq h\;\overline{s}_m$ whenever these two expressions are patterns and $t_i \sqsubseteq s_i$ for all $1 \leq i \leq m$.
- $(t_1, \ldots, t_n) \sqsubseteq (s_1, \ldots, s_n)$ whenever $t_i, s_i \in Pat_\perp$, $t_i \sqsubseteq s_i$ for all $1 \leq i \leq m$.

$Pat_\perp$, and more generally any partially ordered set (shortly, poset), can be converted into a semantic domain by means of a technique called *ideal completion*; see e.g. [16].

Partial patterns of the form $f\; t_1 \ldots t_m$ with $f \in FS^n$ and $m < n$ serve as a convenient representation of functions as values; see [9]. 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 in a debugging session, as we will see.

2.1.3 Substitutions

A *total substitution* is a mapping $\theta : Var \to Pat$ with a unique extension $\hat{\theta} : Exp \to Exp$, which will be noted also as θ. The set of all substitutions is noted as $Subst$. The set of all the *partial substitutions* $\theta : Var \;\to\; Pat_\perp$ is denoted as $Subst_\perp$ and defined analogously. We define the *domain* $dom(\theta)$ as the set of all variables X s.t. $\theta(X) \neq X$, and the *range* $ran(\theta)$ as $\bigcup_{X \in dom(\theta)} var(\theta(X))$. As usual, $\theta = \{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ stands for the substitution with domain $\{X_1, \ldots, X_n\}$ which satisfies $\theta(X_i) = t_i$ for all $1 \leq i \leq n$. By convention, we write $e\theta$ instead of $\theta(e)$, and $\theta\sigma$ for the composition of θ and σ, such that $e(\theta\sigma) = (e\theta)\sigma$ for any e. For any subset $\mathcal{X} \subseteq dom(\theta)$ we define the *restriction* $\theta\restriction_{\mathcal{X}}$ as the substitution θ' such that $dom(\theta') = \mathcal{X}$ and $\theta'(X) = \theta(X)$ for all $X \in A$. We also define the *disjoint union* $\theta_1 \uplus \theta_2$ of two given substitutions with disjoint domains, as the substitution θ such that $dom(\theta) = dom(\theta_1) \cup dom(\theta_2)$, $\theta(X) = \theta_1(X)$ for all $X \in dom(\theta_1)$, and $\theta(Y) = \theta_2(Y)$ for all $Y \in dom(\theta_2)$.

The identity mapping *id* from Var onto itself is called the *identity substitution*, and any substitution ρ which behaves as a bijective mapping from Var onto itself is called a *renaming*. Two expressions e and e' are called *variants* iff there is some renaming ρ such that $e\rho = e'$. The *subsumption ordering* over Exp is defined by the condition $e \leq e'$ iff $e' = e\theta$ for some $\theta \in Subst$. A similar ordering can be defined over $Exp_\perp$, and extended to work over $Subst_\perp$ by defining $\theta \leq \theta'$ iff $\theta' = \theta\sigma$ for some $\sigma \in Subst_\perp$. For any set of data variables $\mathcal{X}$, we use the notations $\theta \leq \theta'[\mathcal{X}]$ (resp. $\theta \leq \theta'[\setminus\mathcal{X}]$) to indicate that $X\theta' = X\theta\sigma$ holds for some $\sigma \in Subst_\perp$ and all $X \in \mathcal{X}$ (resp. all $X \notin \mathcal{X}$). Another useful notion is the *approximation* ordering over $Subst_\perp$, defined by the condition $\theta \sqsubseteq \theta'$ iff $\theta(X) \sqsubseteq \theta'(X)$, for all $X \in Var$.

Up to this point we have considered *data substitutions*. *Type substitutions* can be defined similarly, as mappings $\theta_t : TVar \rightarrow Type$ with a unique extension $\hat{\theta_t} : Type \rightarrow Type$, noted also as θ_t. The set of all type substitutions is noted as $TSubst$. Most of the concepts and notations presented above for data substitutions (such as domain, range, composition, renaming, etc.) make sense also for type substitutions, and we will freely use them when needed.

2.1.4 Well-typed Expressions

Inspired by Milner's type system [15,4] we now introduce the notion of well-typed expression. We define a *type environment* as any set T of type assumptions $X :: \tau$ for data variables, such that T does not include two different assumptions for the same variable. The *domain* $dom(T)$ and the *range* $ran(T)$ of a type environment are the set of all data variables resp. type variables that occur in T. For any variable $X \in dom(T)$, the unique type τ such that $(X :: \tau) \in T$ is noted as $T(X)$. The notation $(h :: \tau) \in_{var} \Sigma$ is used to indicate that Σ includes the type declaration $h :: \tau$ up to a renaming of type variables.

Type judgements $(\Sigma, T) \vdash_{WT} e :: \tau$ are derived by means of the following *type inference* rules:

VR $(\Sigma, T) \vdash_{WT} X :: \tau$, if $T(X) = \tau$

ID $(\Sigma, T) \vdash_{WT} h :: \tau\sigma_t$,

if $(h :: \tau) \in_{var} \Sigma_\perp,\ \sigma_t \in TSubst$

AP $(\Sigma, T) \vdash_{WT} (e\ e_1) :: \tau$,

if $(\Sigma, T) \vdash_{WT} e :: (\tau_1 \rightarrow \tau)$, $(\Sigma, T) \vdash_{WT} e_1 :: \tau_1$, for some $\tau_1 \in Type$

TP $(\Sigma, T) \vdash_{WT} (e_1, \ldots, e_n) :: (\tau_1, \ldots, \tau_n)$,

if $(\Sigma, T) \vdash_{WT} e_1 :: \tau_1, \ldots, (\Sigma, T) \vdash_{WT} e_n :: \tau_n$

Note that the previous type inference rules can deal with polimorphic types, because the type declarations included in the signature Σ are interpreted as type schemes, as seen in the inference rule **ID**.

We will abbreviate a sequence $(\Sigma, T) \vdash_{WT} e_1 :: \tau_1, \ \ldots, (\Sigma, T) \vdash_{WT} e_n :: \tau_n$ as $(\Sigma, T) \vdash_{WT} \overline{e}_n :: \overline{\tau}_n$, while $(\Sigma, T) \vdash_{WT} a :: \tau, \ (\Sigma, T) \vdash_{WT} b :: \tau$ will be abbreviated as $(\Sigma, T) \vdash_{WT} a :: \tau :: b$.

An expression $e \in Exp_\perp$ is called *well-typed* iff there exist some *type environment* T and some type τ, such that the *type judgement* $T \vdash_{WT} e :: \tau$ can be derived. Expressions that admit more than one type are called *polymorphic*. 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 [9] for more details.

2.2 Programs and Goals

2.2.1 Well-typed Programs

A *well-typed program* P is a set of *well-typed defining rules* for the function symbols in its signature. Defining rules for $f \in FS^n$ with principal type declaration $f \ :: \ \overline{\tau}_n \rightarrow \tau$ have the form

$$(R) \qquad \underbrace{f\ t_1 \ldots t_n}_{\text{left-hand side}} \ \rightarrow \ \underbrace{r}_{\text{right-hand side}} \ \Leftarrow \ \underbrace{C}_{\text{condition}}$$

and must satisfy the following requirements:

(i) $t_1 \ldots t_n$ is a linear sequence of transparent patterns and r is an expression.

(ii) The *condition* C is a sequence of *atomic conditions* $C_1, \ldots, C_k$, where each C_i can be either a *joinability statement* of the form $e == e'$, with $e, e' \in Exp$, or an *approximation statement* of the form $d \rightarrow s$, with $d \in Exp$ and $s \in Pat$.

(iii) Moreover, the condition C must be *admissible* w.r.t. the set of variables $\mathcal{X} =_{def} var(f\ \bar{t}_n)$. By definition, this means that the set of all the approximation statements occurring in C must admit some sequential arrangement, say $d_1 \rightarrow s_1, \cdots, d_m \rightarrow s_m$ $(m \geq 0)$, such that the three properties below hold:
  (a) For all $1 \leq i \leq m$: $var(s_i) \cap \mathcal{X} = \emptyset$
  (b) For all $1 \leq i \leq m$, s_i is linear and for all $1 \leq j \leq m$ with $i \neq j$ $var(s_i) \cap var(s_j) = \emptyset$.
  (c) For all $1 \leq i \leq m, 1 \leq j \leq i$: $var(s_i) \cap var(d_j) = \emptyset$.

(iv) There is some type environment T with domain $var(R)$, which well-types the definining rule in the following sense:
  (a) For all $1 \leq i \leq n$: $(\Sigma, T) \vdash_{WT} t_i \ :: \ \tau_i$.
  (b) $(\Sigma, T) \vdash_{WT} r \ :: \ \tau$.
  (c) For each $(e == e') \in C$ there is some $\mu \in Type$ such that $(\Sigma, T) \vdash_{WT} e \ :: \ \mu \ :: \ e'$.

(d) For each $(d \rightarrow s) \in C$ there is some $\mu \in Type$ such that $(\Sigma, T) \vdash_{WT} d :: \mu :: s$.

In the programming language $\mathcal{TOY}$ [14] program rules are written in a somewhat different way, namely:

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

In this syntax, the condition C of a program rule is split in two parts: one part JC consisting of joinability statements $e == e'$, and another part LD consisting of approximation statements $d \rightarrow s$, which are understood as *local definitions* for the variables occurring in the pattern s. This motivates requirement (iii) above. In fact:

- Items (iii) (a), (iii) (b) require the locally defined variables to be different from each other and away from the variables occurring in the rule's left-hand side, that act as formal parameters.
- Item (iii) (c) ensures that variables defined in local definition number i can be used in local definition number j only if $j > i$. In particular, this means that the local definitions cannot be recursive.

Informally, the intended meaning of a program rule like (R) above is that a call to function f can be reduced to r whenever the actual parameters match the patterns t_i, and both the joinability 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 $d \rightarrow s$ is satisfied by evaluating d to some possibly partial pattern which matches s. A precise formulation of program semantics will be presented in Section 2.3.

2.2.2 A Simple Program

Below we show a simple example program, written in the concrete syntax of the $\mathcal{TOY}$ language. In this syntax, local definitions $d \rightarrow s$ are written as $s \leftarrow d$, and they must appear in a textual order which shows fulfilment of the admissibility requirements explained in Section 2.2.1. $\mathcal{TOY}$ also allows to use infix operators such as : to build expressions such as (X:Xs), which is understood as ((:) X Xs). The signature of the program can be easily inferred from the type declarations included in its text. In particular, the data declarations give complete information about the type constructors and the principal types of the data constructors.

```
% data [A] = [] | A : [A]

head :: [A] → A        tail :: [A] → [A]
head (X:Xs)   → X      tail (X:Xs)   → Xs
```

```
map :: (A → B) → [A] → [B]          twice :: (A → A) → A → A
map F []          → []              twice F X    → F (F X)
map F (X:Xs)      → F X : map F Xs

drop4 :: [A] → [A]                  from :: nat → [nat]
drop4             → twice twice tail from N       → N : from N

data nat = z | suc nat

plus :: nat → nat → nat             times :: nat → nat → nat
plus z Y          → Y               times z Y         → z
plus (suc X) Y    → suc (plus X Y)  times (suc X) Y   → plus (times X Y) X

take :: nat → [A] → [A]                (//) :: A → A → A
take z Xs              → []            X // Y    → X
take (suc N) []        → []            X // Y    → Y
take (suc N) (X:Xs)    → X : take N Xs

data person =   john | mary | peter | paul | sally | molly | rose | tom |
                bob | lisa | alan | dolly | jim | alice

parents :: person → (person,person)
parents peter    → (john,mary)       parents alan    → (paul,rose)
parents paul     → (john,mary)       parents dolly   → (paul,rose)
parents sally    → (john,mary)       parents jim     → (tom,sally)
parents bob      → (peter,molly)     parents alice   → (tom,sally)
parents lisa     → (peter,molly)

ancestor :: person → person
ancestor X →     Y // Z // ancestor Y // ancestor Z
                 where (Y,Z) ← parents X

% data bool = true | false

related :: person → person → bool
related X Y →    true <= ancestor X == ancestor Y
```

The data declarations for the types of lists and boolean values are included merely as comments, since these types are predefined in $\mathcal{TOY}$. Note that the list constructors are noted as [] and : (an infix operator), as in Haskell [24]. The intended meaning of the functions should be clear from their names and definitions. The arity of each function is always the same as 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 last two functions illustrate the use of joinability conditions and local definitions. Moreover, the functions ancestor and (//) are *non-deterministic*, since a call to them with fixed parameters can return more than one result. For instance, ancestor alan can return any of the results paul, rose, john or mary.

Some of the program rules in this example are *incorrect* w.r.t. the intended meaning of the corresponding functions. More precisely, the second rule for times and the single rule for from are wrong; their correct versions should be:

$$\mathsf{times\ (suc\ X)\ Y} \to \mathsf{plus\ (times\ X\ Y)\ Y} \qquad \mathsf{from\ N} \to \mathsf{N : from\ (suc\ N)}$$

In the next section we will give a formal definition of "intended meaning", which is needed to prove mathematical results about the correctness of declarative debugging.

2.2.3 Well-typed Goals

A *well-typed goal* G has the same form as a well-typed condition. In particular, it must satisfy the admissibility requirements explained in Section 2.2.1, but now w.r.t. the empty set of variables. A FLP system is expected to *solve* goals, returning substitutions θ as computed answers. For the simple program from Section 2.2.1, some examples of goals and answers which can be computed by the $\mathcal{TOY}$ system are:

(i) The goal related alan X == true has the computed answer $\{\mathsf{X} \mapsto \mathsf{alice}\}$ (among others).

(ii) The goal take (suc (suc z)) (from X) == Xs has a single computed answer, namely $\{\mathsf{Xs} \mapsto \mathsf{X{:}X{:}[\,]}\}$, which is *wrong* w.r.t. the intended meaning of the program.

(iii) 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 first two solutions computed by $\mathcal{TOY}$ are $\{\mathsf{N} \mapsto \mathsf{z},\ \mathsf{Y} \mapsto \mathsf{z}\}$ (which is *correct*) and $\{\mathsf{N} \mapsto \mathsf{suc\ z},\ \mathsf{Y} \mapsto \mathsf{z}\}$ (which is *wrong*). This is because the buggy function times causes the expression (times (suc z)) to return always the result z. The valid solution $\{\mathsf{N} \mapsto \mathsf{suc\ z},\ \mathsf{Y} \mapsto \mathsf{suc\ X}\}$ expected by the user is in fact a *missing answer*. Diagnosing missing answers is beyond the scope of this paper.

2.3 Program Semantics

2.3.1 The Semantic Calculus SC

In [9], a *rewriting calculus* called *GORC* was used to deduce from a given program P those approximation and joinability statements which should be considered as valid according to P's semantics. Informally, an approximation statement $e \to t$ means that $t \in Pat_\perp$ represents a partially defined value which approximates the value of $e \in Exp_\perp$; while a joinability statement $e == e'$ means that $e \to t$, $e' \to t$ holds for some *total* $t \in Pat$.

In this paper we will use the *Semantic Calculus SC*, a variant of *GORC* which was first proposed in [2] in order to define a logically correct framework for the declarative debugging of wrongs answers in lazy FLP languages. Formally, *SC* consists of the following inference rules:

$$\textbf{BT} \quad e \rightarrow \perp$$

$$\textbf{RR} \quad X \rightarrow X \text{ with } X \in Var$$

$$\textbf{DC} \quad \frac{e_1 \rightarrow t_1 \ \ldots \ e_m \rightarrow t_m}{h\ \overline{e}_m \rightarrow h\ \overline{t}_m} \ h\ \overline{t}_m \in Pat_\perp$$

$$\textbf{JN} \quad \frac{e \rightarrow t \qquad e' \rightarrow t}{e \ == \ e'} \ t \in Pat \text{ (total pattern)}$$

$$\textbf{AR+FA} \quad \frac{e_1 \rightarrow t_1 \ \ldots \ e_n \rightarrow t_n \quad \dfrac{C \quad r \rightarrow s}{\boxed{f\ \overline{t}_n \rightarrow s}} \quad s\ \overline{a}_k \rightarrow t}{f\ \overline{e}_n\ \overline{a}_k \rightarrow t} \ \begin{array}{l}(f\ \overline{t}_n \rightarrow r \Leftarrow C) \in [P]_\perp, \\ t \neq \perp\end{array}$$

In all the SC rules, $e, e_i \in Exp_\perp$ are partial expressions, $t_i, t, s \in Pat_\perp$ are partial patterns and $h \in DC \cup FS$. The notation $[P]_\perp$ in rule $AR + FA$ stands for the set $\{(l \rightarrow r \Leftarrow C)\theta \mid (l \rightarrow r \Leftarrow C) \in P,\ \theta \in Subst_\perp\}$ of partial instances of the rules from P. The labels of the different inference rules have the following intended meanings: BT stands for *Bottom*, RR for *restricted reflexivity*, DC for *decomposition*, JN for *joinability* and $AR + FA$ for *argument reduction + function application*.

Notice that $AR+FA$ is the only SC rule which depends on the given program. It must be understood as the consecutive application of two inference steps, whose separate specification is displayed below:

$$\textbf{AR} \quad \frac{e_1 \rightarrow t_1 \ \ldots \ e_n \rightarrow t_n \quad f\ \overline{t}_n \rightarrow s \quad s\ \overline{a}_k \rightarrow t}{f\ \overline{e}_n\ \overline{a}_k \rightarrow t} \ \begin{array}{l} f \in FS^n \\ t \neq \perp\end{array}$$

$$\textbf{FA} \quad \frac{C \qquad r \rightarrow s}{f\ \overline{t}_n \rightarrow s} \quad (f\ \overline{t}_n \rightarrow r \Leftarrow C) \in [P]_\perp$$

The rule $AR+FA$ formalizes the steps to be performed for computing a *partial pattern* t as approximated value for the function application $f\ \overline{e}_n\ \overline{a}_k$, namely:

(i) Compute suitable *partial patterns* t_i as approximated values for the argument expressions e_i.

(ii) Apply a program rule instance $(f\ \overline{t}_n \rightarrow r \Leftarrow C) \in [P]_\perp$, verify the condition C, and compute a suitable partial pattern s as approximated value for the right-hand side r.

(iii) Compute t as approximated value for $s\ \overline{a}_k$.

Working with partial patterns here allows to specify non-strict semantics with the syntactic simplicity of strict semantics. In the case $k > 0$, f must be a higher-order function which returns a functional value, represented by the pattern s. In the case $k = 0$, the rule $AR + FA$ can be simplified by taking $f\ \overline{t}_n \rightarrow t$ as the conclusion of the FA step, and omitting the premise $s\ \overline{a}_k \rightarrow t$. We will implicitly assume this simplification all along the paper.

Note that SC cannot apply the two inference rules AR and FA independently; they must be always used within a combined $AR + FA$ step. Nevertheless, to think of the FA steps within a given SC proof is helpful, because only such steps depend on program rules. Moreover, the conclusions of FA steps are particularly simple approximation statements of the form $f\ \bar{t}_n \to s$ (with $t_i,\ s \in Pat_\perp$), which will be called *basic facts* in the rest of the paper. Both basic facts and local definitions are approximation statements, but they are used for different purposes. A basic fact $f\ \bar{t}_n \to s$ asserts that the (possibly non-linear) partial pattern s approximates the result of $f\ \bar{t}_n$, a call function call with the exact number of arguments expected by f's arity, and with arguments $t_i \in Pat_\perp$, which represent the partial approximations of f's actual parameters needed to compute s as result.

The other inference rules in SC are easier to understand. In the sequel we use the notation $P \vdash_{SC} \varphi$ is used to indicate that the statement φ can be deduced from the program P using the SC inference rules. For instance, taking as P the simple program from Section 2.2.2, the following SC derivations are possible:

(i) $P \vdash_{SC}$ from X $\to$ X:$\perp$.

(ii) $P \vdash_{SC}$ from X $\to$ X:X:$\perp$.

(iii) $P \vdash_{SC}$ parents alice $\to$ (tom,sally).

(iv) $P \vdash_{SC}$ ancestor alan $\to$ john.

(v) $P \vdash_{SC}$ ancestor alan $\to$ mary.

(vi) $P \vdash_{SC}$ ancestor alice $\to$ john.

(vii) $P \vdash_{SC}$ ancestor alice $\to$ mary.

(viii) $P \vdash_{SC}$ ancestor alan $==$ ancestor alice.

These examples show that the semantics of approximation statements is consistent with their use as local definitions within programs, but different from the meaning of equality. For instance, from X $\to$ X:$\perp$ only means that the partial value X:$\perp$ *approximates* the value of (from X), not that the value of (from X) is *equal* to X:$\perp$. There is a formal relationship between approximation statements and the approximation ordering over $Pat_\perp$ defined in Section 2.1.2. This and other basic properties of SC are stated in the following result, which can be proved by straightforward induction on the structure of SC proofs [4].

Proposition 2.1 *For any given program P:*
(i) *For all $t,\ s \in Pat_\perp$: $P \vdash_{SC} t \to s$ iff $t \sqsupseteq s$.*

[4] The proof of a similar result for first-order programs can be found in [8].

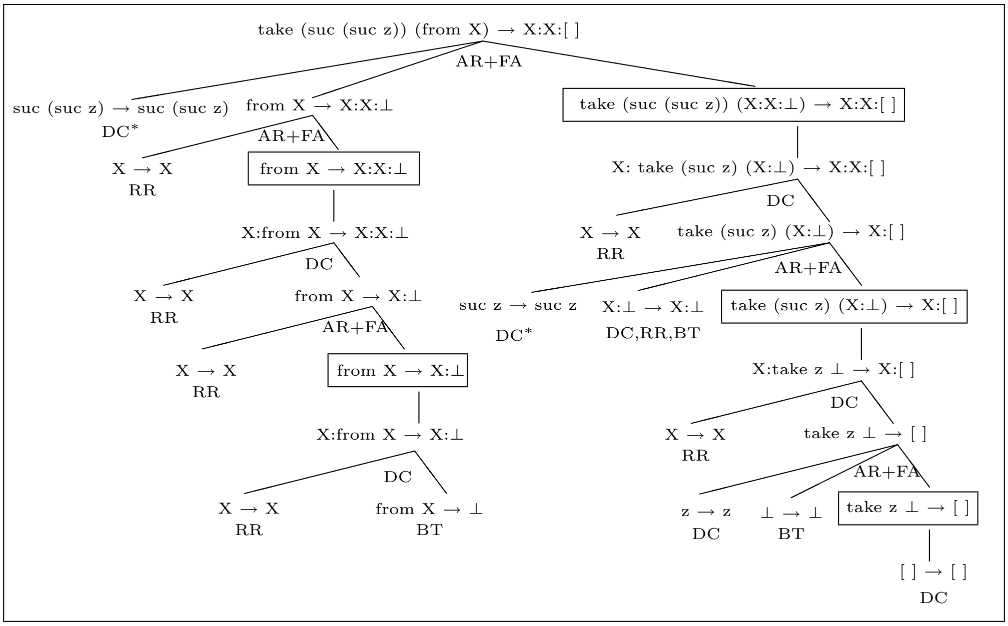


Figure 1: Proof Tree in the semantic calculus SC

(ii) *For all* $e \in Exp_\perp$, t, $s \in Pat_\perp$: *if* $P \vdash_{SC} e \to t$ *and* $t \sqsupseteq s$, *then also* $P \vdash_{SC} e \to s$.

(iii) *For all* $e \in Exp_\perp$, $t \in Pat_\perp$ *and* θ, $\theta' \in Subst_\perp$ *such that* $P \vdash_{SC} e\theta \to t$ *and* $\theta \sqsubseteq \theta'$, *one also has* $P \vdash_{SC} e\theta' \to t$ *with a SC proof of the same size and structure.*

(iv) *For all* $e \in Exp_\perp$, $s \in Pat_\perp$ *such that* $P \vdash_{SC} e \to s$, *one has also* $P \vdash_{SC} e\theta \to s\theta$ *for any* total substitution $\theta \in Subst$.

2.3.2 Proof Trees Witnessing Computed Answers

We have already introduced the notion of computed answer in Section 2.2.3, assuming the existence of some goal solving system. From now on and for the rest of the paper, we will also assume that the goal solving system is sound w.r.t. the semantic calculus SC. More precisely, we assume that $P \vdash_{SC} G\theta$ holds for every substitution θ which is computed as an answer for G by the goal solving system, using program P. Note that θ *must be thought as given in advance* before SC proves $G\theta$. By convention, the notation $P \vdash_{SC} G\theta$ means that $P \vdash_{SC} \varphi\theta$ holds for each single atomic statement φ in G.

Given an atomic goal G, a particular SC deduction proving $P \vdash_{SC} G\theta$ can be always represented using a *proof tree* (briefly PT) with atomic statements attached to its nodes, such that $G\theta$ is attached to the root node and the statement at each node can be inferred form the statements attached to its children by means of some SC inference rule. In the case that G is not atomic, each particular SC deduction proving $P \vdash_{SC} G\theta$ can be represented by a family of proof trees for the different deductions $P \vdash_{SC} \varphi\theta$ corresponding to the single atomic statements in G. By slight abuse of the language, we will speak of a proof tree also in this case.

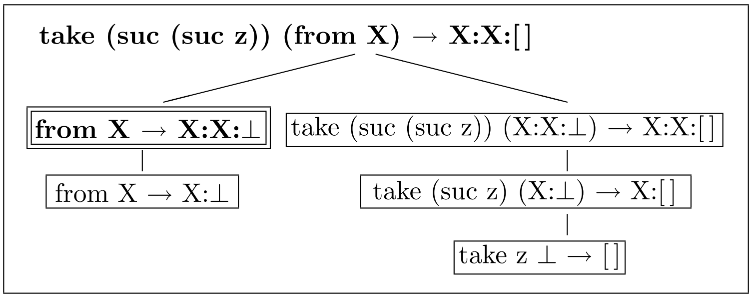


Figure 2: APT corresponding to the PT of Figure 1

As we have mentioned already, $\theta = \{\mathsf{Xs} \mapsto \mathsf{X{:}X{:}[]}\}$ is a computed answer for the goal $G =$ take (suc (suc z)) (from X) == Xs w.r.t. the simple program P form Section 2.2.2. Any proof of $P \vdash_{SC} G\theta$ must include a deduction of $P \vdash_{SC}$ take (suc (suc z)) (from X) $\rightarrow$ X:X:[], which is witnessed by the PT displayed in Fig. 1. Note that the basic facts occurring as conclusions of FA steps are highlighted by displaying them within boxes.

2.3.3 Abbreviated Proof Trees

As we will explain in the next section, our aim is to use proof trees as computation trees for declarative debugging. To this purpose, the only relevant nodes are those which correspond to the conclusion of FA steps. This is because all the other inference rules in SC, being program independent, cannot give rise to incorrect steps. For this reason, we associate to any given proof tree an *abbreviated proof tree* (briefly APT), obtained by removing all those nodes of the PT, except the root, which are not the conclusion of a FA inference. More precisely, the APT corresponding to a given PT is constructed as follows:

- The root of the APT is the root of the given PT.
- For any node already placed in the APT, its children are the closest descendants of the corresponding node in the PT which represent the conclusion of a non-trivial FA step.
- A FA step with conclusion $f\ \bar{t}_n \rightarrow s$ is considered non-trivial iff $s \neq \perp$.

Note that trivial FA steps can be also ignored, because their conclusions are always trivially valid facts of the form $f\ \bar{t}_n \rightarrow \perp$. In every APT, each node is implicitly associated to the program rule instance used by the corresponding FA step, whose conclusion is precisely the basic fact $f\ \bar{t}_n \rightarrow s$ at the node. Note that $t_1, \ldots, t_n$, s are partial patterns which cannot contain any reducible function calls. As a concrete example, Fig. 2 shows the APT obtained from the PT in Fig. 1.

2.3.4 Intended Models

Intended models of logic programs, as used in [6,13], can be represented as sets of atomic formulas belonging to the program's Herbrand base. The *open Herbrand universe* (i.e. the set of terms with variables) gives rise to a more informative semantics [5]. In our FLP setting, a natural analogous to the open Herbrand universe is the set $Pat_\perp$ of all the partial patterns, equipped with the approximation ordering $\sqsubseteq$. Similarly, a natural analogous to the open Herbrand base is the collection of all the basic facts $f\ \bar{t}_n \to s$. Therefore, we can define a *Herbrand interpretation* as a set $\mathcal{I}$ of basic facts fulfilling the following three requirements for all $f \in FS^n$ and arbitrary partial patterns $t, \bar{t}_n$:

(i) $(f\,\bar{t}_n \to\perp)\ \in \mathcal{I}$.

(ii) If $(f\ \bar{t}_n \to s) \in \mathcal{I}$, $t_i \sqsubseteq t'_i$, $s \sqsupseteq s'$ then also $(f\,\bar{t}'_n \to s')\ \in \mathcal{I}$.

(iii) if $(f\ \bar{t}_n \to s)\ \in \mathcal{I}$, and θ is *total* substitution, then $(f\ \bar{t}_n \to s)\theta \in \mathcal{I}$.

This definition of Herbrand interpretation is simpler than the one in [9], where a more general notion of interpretation (under the name *algebra*) is presented. The trade-off for this simpler presentation is to exclude non-Herbrand interpretations from our consideration. In our debugging scheme we will assume that the intended model of a program is a Herbrand interpretation $\mathcal{I}$. Herbrand interpretations can be ordered by set inclusion.

A logically correct program P should conform to its intended interpretation $\mathcal{I}$. In order to formalize this idea, we need some definitions. First, we say that a given approximation or joinability statement φ is *valid* in the Herbrand interpretation $\mathcal{I}$ iff φ can be proved in the calculus $SC_\mathcal{I}$ consisting of the SC rules BT, RR, DC and JN together with the inference rule $FA_\mathcal{I}$ below:

$$\mathbf{FA}_\mathcal{I} \quad \frac{e_1 \to t_1\ \ldots\ e_n \to t_n \quad s\ \bar{a}_k \to t}{f\ \bar{e}_n\,\bar{a}_k \to t} \quad \begin{array}{l} t \text{ pattern}, t \neq\perp, s \text{ pattern} \\ (f\ \bar{t}_n\ \to\ s)\ \in \mathcal{I} \end{array}$$

For instance, assuming the natural intended model $\mathcal{I}$ for the simple program from Section 2.2.2, the following statements are valid in $\mathcal{I}$:

(i) from X $\to$ X:suc X:$\perp$

(ii) take (suc (suc z)) (from X) $\to$ X:suc X:[]

(iii) ancestor alan == ancestor alice

The first of these statements even belongs to $\mathcal{I}$. In general, for every basic fact $f\ \bar{t}_n \to s$, it can be proved that $f\ \bar{t}_n \to s$ is valid in $\mathcal{I}$ iff $(f\ \bar{t}_n \to s) \in \mathcal{I}$.

Next we define the *denotation* of expressions and the notion of *model* of a given program:

- The denotation of e is the set $[\![e]\!]^\mathcal{I} = \{s \in Pat_\perp \mid e \to s \text{ valid in } \mathcal{I}\}$.
- $\mathcal{I}$ is a model of P ($\mathcal{I} \models P$) iff every program rule in P is valid in $\mathcal{I}$.
- A program rule $l \to r\ \Leftarrow C$ is valid in $\mathcal{I}$ ($\mathcal{I} \models l \to r\ \Leftarrow C$) iff for any substitution $\theta \in Subst_\perp$, $\mathcal{I}$ satisfies the rule instance $l\theta \to r\theta\ \Leftarrow C\theta$.
- $\mathcal{I}$ satisfies a rule instance $l' \to r'\ \Leftarrow C'$ iff either $\mathcal{I}$ does not satisfy C' or else $[\![l']\!]^\mathcal{I} \supseteq [\![r']\!]^\mathcal{I}$.

- $\mathcal{I}$ satisfies an instantiated condition $C' = \varphi_1, \dots \varphi_k$ iff for $i = 1 \dots k$, $\mathcal{I}$ satisfies φ_i.
- $\mathcal{I}$ satisfies $d' \rightarrow s' \in C'$, iff $[\![d']\!]^{\mathcal{I}} \supseteq [\![s']\!]^{\mathcal{I}}$. It can be shown that $[\![d']\!]^{\mathcal{I}} \supseteq [\![s']\!]^{\mathcal{I}}$ iff $s' \in [\![d']\!]^{\mathcal{I}}$.
- $\mathcal{I}$ satisfies $l' == r' \in C'$, iff $[\![l']\!]^{\mathcal{I}} \cap [\![r']\!]^{\mathcal{I}} \cap Pat \neq \emptyset$.

The fundamental relationship between programs and models is stated in the following result, which is proved in [9] for a notion of model more general than Herbrand models. A proof for the present formulation can be found in Appendix A.

Theorem 2.2 *Let P be a program and φ any approximation or joinability statement. Then:*
(a) If $P \vdash_{SC} \varphi$ then φ is valid in any Herbrand model of P.
(b) $\mathcal{M}_P = \{f\ \bar{t}_n \rightarrow s \mid P \vdash_{SC} f\ \bar{t}_n \rightarrow s\}$ is the least Herbrand model *of P w.r.t. the inclusion ordering.*
(c) If φ is valid in $\mathcal{M}_P$ then $P \vdash_{SC} \varphi$.

Putting together the previous theorem and the assumed soundness of the goal solving system w.r.t. SC, we immediately obtain:

Proposition 2.3 *Assume a program P and a computed answer θ for a goal G, such that $G\theta$ is not valid in the Herbrand interpretation $\mathcal{I}$. Then, there must be some program rule in P which is not valid in $\mathcal{I}$.*

This proposition predicts the existence of at least one wrong program rule whenever a wrong computed answer is observed. Here, *wrong* must be understood in the precise sense of being *not valid in the intended model.* In the case of our simple program P, $\theta = \{\mathsf{Xs} \mapsto \mathsf{X{:}X{:}[]}\}$ is a wrong computed answer for the goal $G = \mathsf{take\ (suc\ (suc\ z))\ (from\ X)} == \mathsf{Xs}$, because $G\theta$ is not valid in the intended model. By Proposition 2.3, some wrong rule in P must be responsible for the wrong answer. Indeed, the program rule defining the function from is wrong.

Whenever a program rule $l \rightarrow r \Leftarrow C$ is not valid in the intended model $\mathcal{I}$, there must be some substitution $\theta \in Subst_\perp$ such that the rule instance $l\theta \rightarrow r\theta \Leftarrow C\theta$ is not satisfied by $\mathcal{I}$, which means that

(i) $\varphi\theta$ is valid in $\mathcal{I}$ for all $\varphi \in C$.

(ii) $r\theta \rightarrow s$ is valid in $\mathcal{I}$ for some $s \in Pat_\perp$ such that $(l\theta \rightarrow s) \notin \mathcal{I}$.

In our example, the incorrect instance of the rule defining from is the rule itself. Indeed, $\mathsf{N{:}from\ N} \rightarrow \mathsf{N{:}N{:}}\perp$ is valid in $\mathcal{I}$, but $(\mathsf{from\ N} \rightarrow \mathsf{N{:}N{:}}\perp) \notin \mathcal{I}$. This corresponds to item (ii) above, with $\mathsf{N{:}N{:}}\perp$ acting as s.

For the purposes of practical debugging, Proposition 2.3 must be refined to yield an *effective* method which can be used to find an incorrect instance of a program rule, starting from the observation of a wrong computed answer.

In the next section, we show that this can be achieved by using a declarative debugging scheme with APTs acting as computation trees. Effective methods to implement this approach are investigated in the rest of the paper.

2.4 Declarative Debugging

2.4.1 A Generic Declarative Debugging Scheme

The debugging scheme proposed in [19] assumes that any terminated computation can be represented as a finite tree, called *computation tree* (briefly CT). The root of this tree corresponds to the result of the main computation, and each node corresponds to the result of some intermediate subcomputation. Moreover, it is assumed that the result at each node is *determined* by the results of the children nodes. Therefore, every node can be seen as the outcome of a single *computation step*. The debugger works by traversing a given CT (so called CT *navigation*), looking for *erroneous* nodes. Different kinds of programming paradigms and/or errors need different types of trees, as well as different notions of *erroneous*.

A sound debugger should only report bugs that really correspond to wrong computation steps. This consideration leads to ignore erroneous nodes which have some erroneous children, since they do not necessarily correspond to wrong computation steps. Following the terminology of [19], an erroneous node with no erroneous children is called a *buggy node*. In order to avoid unsoundness, the debugging scheme looks only for buggy nodes, asking questions to an *oracle* (generally the user) in order to determine which nodes are erroneous. The following easy result is proved in [19]:

Proposition 2.4 *A finite computation tree has an erroneous node iff it has a buggy node. In particular, a finite computation tree whose root node is erroneous has some buggy node.*

This provides a 'weak' notion of *completeness* for the debugging scheme that is satisfactory in practice. Usually, actual debuggers look only for a topmost buggy node in a computation tree whose root is erroneous. Multiple bugs can be found by reiterated application of the debugger.

2.4.2 Debugging with APTs is Logically Correct

Our debugging system is based on the declarative debugging scheme just recalled. We assume well-typed FLP programs and goals, as described in Section 2.2. We also suppose an intended model for each program, represented as a set of basic facts, as explained in Section 2.3.4. Computations are performed by a goal solving system which must be sound w.r.t. the semantic calculus SC from Section 2.3.1. Whenever a computation obtains an answer substitution θ for a goal G using program P, we assume that an APT witnessing $P \vdash_{SC} G\theta$ is used as computation tree. An APT node is considered erroneous iff the statement attached to it (which is always a basic fact, except perhaps for the root) is not valid in the intended model.

The next theorem guarantees the logical correctness of declarative debugging with APTs:

Theorem 2.5 *Assume a wrong computed answer θ, computed for the goal G using program P, such that $G\theta$ is not valid in the intended model. Consider any APT witnessing $P \vdash_{SC} G\theta$, which must exist due to soundness of the goal solving system w.r.t. SC. Then, declarative debugging using the APT as computation tree has the following two properties:*

(a) Completeness: navigating the APT will find a buggy node.

(b) Soundness: every buggy node in the APT points to an instance of a program rule which is incorrect w.r.t. the intended model.

Proof.
Item (a) follows immediately from Proposition 2.4, provided that the search strategy used to navigate the tree does not miss existing buggy nodes. To prove item (b), assume that the intended model is $\mathcal{I}$, the APT is *apt*, and the PT which has been abbreviated to obtain *apt* is *pt*. Now consider any given buggy node in *apt*. The corresponding node in *pt* must contain a basic fact $f\ \bar{t}_n \to s$ which is not valid in $\mathcal{I}$ and has been inferred as the conclusion of a FA inference step using some instance of a program rule, say $(f\ \bar{t}_n \to r \Leftarrow C) \in [P]_\perp$. Therefore, the children of $f\ \bar{t}_n \to s$ in *pt* correspond to the statement $r \to s$ and all the statements in C. In *apt*, the children of $f\ \bar{t}_n \to s$ are not necessarily these; but since *apt* has been built as the abbreviated form of *pt*, it happens that $r \to s$ and C can be inferred from the children of $f\ \bar{t}_n \to s$ in *apt* by means of SC inferences which are different from FA and therefore correct in every Herbrand interpretation. Moreover, all the children of $f\ \bar{t}_n \to s$ in *apt* are valid in $\mathcal{I}$, because they are the children of a buggy node. With this we can conclude that C and $r \to s$ are valid in $\mathcal{I}$, while $f\ \bar{t}_n \to s$ is not; which means that the program rule instance $(f\ \bar{t}_n \to r \Leftarrow C) \in [P]_\perp$ is incorrect in $\mathcal{I}$. $\square$

This theorem provides an effective version of Proposition 2.3 as well as a logical interpretation of computation trees. To the best of our knowledge, this is missing in other related approaches to declarative debugging of lazy FP and FLP programs [21,22,23,27,18,20,25].

As a concrete example, consider again the PT shown in Fig. 1 and the corresponding APT shown in Fig. 2. As we have said before, PT witnesses the computation of the wrong answer $\theta = \{\mathsf{Xs} \mapsto \mathsf{X{:}X{:}[]}\}$ for the goal

$$G = \mathsf{take\ (suc\ (suc\ z))\ (fromX)} == \mathsf{Xs}$$

using the simple program from Section 2.2.2 [5]. In Fig. 2, the statements at

[5] Strictly speaking, a witnessing PT for this computation should have the joinability state-

erroneous nodes are displayed in bold letters, and the only buggy node appears surrounded by a double box. In this case, the reasoning of Theorem 2.5 leads to the incorrect program rule instance used by the FA step at the buggy node, which is from N $\rightarrow$ N:from N.

In a previous work [2] we have presented a method to extract the APT which witnesses a particular computation from a formal representation of the computation in a lazy narrowing calculus. This theoretical result depends on a particular formalization of narrowing, and does not provide a direct way to implement a debugging tool for existing FLP systems. In the rest of this paper we propose more effective methods for the generation and navigation of APTs, which allow to implement a working debugging tool.

3 Problems and Contributions

In this short 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 [22,20,25] gives rise to transformed programs whose functions return CTs along with the originally expected results. The *abstract machine* approach [21,22,23,27] requires lower 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.

3.1.1 Curried Functions

Roughly, all transformational approaches transform the functions defined in the source program to return pairs (res, ct) consisting of a computed result and a CT. From the viewpoint of types, the transformation of a n-ary function $f \in FS^n$ looks as follows:

$$f \ :: \ \tau_1 \ \rightarrow \ \cdots \ \ \rightarrow \tau_n \ \rightarrow \ \tau \quad \Rightarrow \quad f^{\mathcal{T}} \ :: \ \tau_1^{\mathcal{T}} \ \rightarrow \ \cdots \ \rightarrow \tau_n^{\mathcal{T}} \ \rightarrow \ (\tau^{\mathcal{T}}, \ \mathsf{cTree})$$

where cTree is a datatype for representing CTs, and $\tau_i^{\mathcal{T}}$ resp. $\tau^{\mathcal{T}}$ are suitable transformations of the types τ_i resp. τ. This type transformation amounts to the identity in the case of *datatypes* (i.e., types with no occurrence of the type constructor "$\rightarrow$"), but it becomes relevant in the case of *higher-order* (briefly, HO) types, whose translation involves the type cTree. For instance, the types

ment take (suc (suc z)) (from X) == X:X:[] at the root; but the PT from Fig. 1 represents the interesting part of the deduction.

of the functions plus, drop4 and map from the simple program in Section 2.2.2, whose respective arities are 2, 0 and 2, are translated as shown below. The type of drop4 has the form $(\tau^{\mathcal{T}}, cTree)$ because drop4 has been declared as a nullary function, to be defined by parameterless program rules.

$$\begin{array}{lcl} \mathsf{plus} :: \mathsf{nat} \rightarrow \mathsf{nat} \rightarrow \mathsf{nat} & \Rightarrow & \mathsf{plus}^{\mathcal{T}} :: \mathsf{nat} \rightarrow \mathsf{nat} \rightarrow (\mathsf{nat},\ \mathsf{cTree}) \\ \mathsf{drop4} :: [\mathsf{A}] \rightarrow [\mathsf{A}] & \Rightarrow & \mathsf{drop4}^{\mathcal{T}} :: ([\mathsf{A}] \rightarrow ([\mathsf{A}],\ \mathsf{cTree}),\mathsf{cTree}) \\ \mathsf{map} :: (\mathsf{A} \rightarrow \mathsf{B}) \rightarrow [\mathsf{A}] \rightarrow [\mathsf{B}] & \Rightarrow & \mathsf{map}^{\mathcal{T}} :: (\mathsf{A} \rightarrow (\mathsf{B},\mathsf{cTree})) \rightarrow [\mathsf{A}] \rightarrow ([\mathsf{B}],\mathsf{cTree}) \end{array}$$

As pointed out in [20,25], this 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 ($\mathsf{map}^{\mathcal{T}}$ $\mathsf{drop4}^{\mathcal{T}}$) is ill-typed, because the type of $\mathsf{drop4}^{\mathcal{T}}$ does not match the type expected by $\mathsf{map}^{\mathcal{T}}$ for its first parameter. More generally, the type of the result returned by $f^{\mathcal{T}}$ 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. However, when translating naïvely, ($\mathsf{map}^{\mathcal{T}}$ ($\mathsf{plus}^{\mathcal{T}}$ z)) remains well-typed, while ($\mathsf{map}^{\mathcal{T}}$ $\mathsf{plus}^{\mathcal{T}}$) becomes ill-typed.

As a possible solution to this problem, the authors of [20] suggest to modify the translation in such a way that a curried function of arity $n > 0$ always returns a result of type $(\tau^{\mathcal{T}}, \mathsf{cTree})$ when applied to its first parameter. The type translation of the function plus following this idea yields $\mathsf{plus}^{\mathcal{T}} :: \mathsf{nat} \rightarrow (\mathsf{nat} \rightarrow (\mathsf{nat},\ \mathsf{cTree}),\ \mathsf{cTree})$.

However, as noted in [20], such a transformation would cause transformed programs to compute inefficiently, producing CTs with many useless nodes. Therefore, the authors of [20] wrote: *"An intermediate transformation which only handles currying when necessary is desirable. Whether 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 $f_0^{\mathcal{T}}, \ldots, f_{n-2}^{\mathcal{T}}, f^{\mathcal{T}}$ with respective arities 1, 2, $\ldots n-1$, n, and suitable types. Function $f_m^{\mathcal{T}}$ $(0 \leq m \leq n-2)$ is used to translate occurrences of f applied to m parameters, while $\mathrm{f}^{\mathcal{T}}$ translates occurrences of f applied to $n-1$ parameters. For instance, (map plus) is transformed into ($\mathsf{map}\ ^{\mathcal{T}}$ $\mathsf{plus_0}^{\mathcal{T}}$), using the auxiliary function $\mathsf{plus_0}^{\mathcal{T}} :: \mathsf{nat} \rightarrow (\mathsf{nat} \rightarrow (\mathsf{nat},\mathsf{cTree}),\ \mathsf{cTree})$. As we will see formally in Section 4, the application of a n-ary function f to n or more parameters must be translated with the help of local definitions, a technique already used in [22,20,25].

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 ($\mathsf{twice}^{\mathcal{T}}$ 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 detailed specification of the transformation, dealing both with types and with program rules, is presented in Section 4.

3.1.2 Correctness Results

Our program transformation preserves polymorphic well-typing (module the type transformation $\tau \mapsto \tau^{\mathcal{T}}$) as well as the program semantics formalized in Section 2.3. Under some minimal and natural assumptions about the goal solving system, we also prove that translated programs compute APTs which can be used for logically correct declarative debugging, as we have seen in Section 2.4.2, Theorem 2.5.

These correctness results are presented in Section 4. To the best of our knowledge, previous related papers [22,20,25] give no correctness proof for the program transformation. The author of [25], 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.

3.2 CT Navigation

In order to be a really practical tool, a declarative debugger should keep the number of questions asked to the oracle as small as possible. Our debugger uses a decidable and semantically correct entailment between basic 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 can be avoided. In Section 5 we define the entailment relation, proving its decidability and discussing its use during CT navigation.

4 Generation of CTs by Program Transformation

In this section we present the program transformation used by our debugger and we prove its correctness. Roughly, a program P is converted into a new program $P^{\mathcal{T}}$, where function calls return the same results P would return, but paired with CTs. Formally, $P^{\mathcal{T}}$ is obtained by transforming the signature Σ of P into a new signature $\Sigma^{\mathcal{T}}$, 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 s rl cts) corresponds to a call to the function f with arguments ts and result s, 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 s. Due to lazy evaluation, the main computation may demand only partial approximations of the results of intermediate computations. Therefore, ts and s stand for possibly *partial values*, represented as partial patterns; and (f ts → s) represents the *basic fact* whose validity will be asked to the oracle during debugging, as explained in Section 2.4. As for void, it represents an empty CT, returned by calls to functions which are trusted to be correct (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; other choices are possible. In fact, our current prototype debugger uses more structured representations instead of strings. In particular, values of type rule in our debugging system represent instances of program rules, so that the wrong program rule instances associated to buggy nodes can be presented to the user.

4.2 Transforming Program Signatures

For every n-ary function $f \;::\; \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$ occurring in P, $P^{\mathcal{T}}$ must include an $(m+1)$-ary auxiliary function $f_m^{\mathcal{T}}$ for each $0 \leq m < n-1$, as well as an n-ary function $f^{\mathcal{T}}$, with principal types:

$$
\begin{aligned}
f_m^{\mathcal{T}} &\;::\; \tau_1^{\mathcal{T}} \rightarrow \ldots \rightarrow \tau_{m+1}^{\mathcal{T}} \rightarrow ((\tau_{m+2} \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau)^{\mathcal{T}}, \mathsf{cTree}) \\
f^{\mathcal{T}} &\;::\; \tau_1^{\mathcal{T}} \rightarrow \ldots \rightarrow \tau_n^{\mathcal{T}} \rightarrow \; (\tau^{\mathcal{T}}, \mathsf{cTree})
\end{aligned}
$$

Similarly, for each n-ary data constructor $c \;::\; \tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \tau$ occurring in P, $P^{\mathcal{T}}$ must keep c with the same principal type, and include new $(m+1)$-ary auxiliary functions $c_m^{\mathcal{T}}$ $(0 \leq m < n)$, with principal types:

$$
c_m^{\mathcal{T}} \;::\; \tau_1^{\mathcal{T}} \rightarrow \ldots \rightarrow \tau_{m+1}^{\mathcal{T}} \rightarrow ((\tau_{m+2} \rightarrow \ldots \rightarrow \tau_n \rightarrow \; \tau)^{\mathcal{T}}, \mathsf{cTree})
$$

Note that $c_m^{\mathcal{T}}$ are not data constructors in the translated signature. Defining rules for them will be presented below. The principal types declared above for the function symbols in the transformed signature depend on a *type transformation*. Any type τ in P's signature is transformed into another type $\tau^{\mathcal{T}}$ in $P^{\mathcal{T}}$'s signature, which is recursively defined as follows:

$$
\begin{aligned}
&\alpha^{\mathcal{T}} && = \alpha && (\alpha \in TVar)\\
&(C\ \overline{\tau}_n)^{\mathcal{T}} && = C\ \overline{\tau}_n^{\mathcal{T}} && (C \in TC^n)\\
&(\mu \to \nu)^{\mathcal{T}} && = \mu^{\mathcal{T}} \to (\nu^{\mathcal{T}}, \mathsf{cTree}) &&
\end{aligned}
$$

Observe that $\tau^{\mathcal{T}}$ equals τ whenever τ is a dataype with no occurrences of the higher-order type constructor "$\to$". Since this is the case for the principal types of arguments and results of data constructors c, the auxiliary functions $c_m^{\mathcal{T}}$ can be also declared as

$$
c_m^{\mathcal{T}}\ ::\ \tau_1 \to \ldots \to \tau_{m+1} \to ((\tau_{m+2} \to \ldots \to \tau_n \to \tau)^{\mathcal{T}}, \mathsf{cTree})
$$

In addition to the constructors and functions obtained by transforming those occurring in P's the signature of $P^{\mathcal{T}}$ always includes some additional auxiliary function symbols, which will be introduced in Section 4.4 below.

4.3 Defining Auxiliary Functions

Each auxiliary function $f_m^{\mathcal{T}}$ expects $m+1$ arguments and returns a partial application of $f_{m+1}^{\mathcal{T}}$ paired with a trivial CT. Exceptionally, $f_{n-2}^{\mathcal{T}}$ returns a partial application of $f^{\mathcal{T}}$. The auxiliary functions $c_m^{\mathcal{T}}$ are defined similarly, except that $c_{n-1}^{\mathcal{T}}$ returns a value built with the data constructor c.

$$
\begin{array}{llllll}
f_0^{\mathcal{T}}\, X_1 & \to & (f_1^{\mathcal{T}}\, X_1, \text{void}) & \quad c_0^{\mathcal{T}}\, X_1 & \to & (c_1^{\mathcal{T}}\, X_1, \text{void})\\
f_1^{\mathcal{T}}\, X_1\, X_2 & \to & (f_2^{\mathcal{T}}\, X_1\, X_2, \text{void}) & \quad c_1^{\mathcal{T}}\, X_1\, X_2 & \to & (c_2^{\mathcal{T}}\, X_1\, X_2, \text{void})\\
\ldots & & & \quad \ldots & & \\
f_{n-2}^{\mathcal{T}}\, \overline{X}_{n-1} & \to & (f^{\mathcal{T}}\ \overline{X}_{n-1}, \text{void}) & \quad c_{n-1}^{\mathcal{T}}\, \overline{X}_n & \to & (c\ \overline{X}_n, \text{void})
\end{array}
$$

4.4 Transforming Function Definitions

Each program rule $f\ t_1 \ldots t_n \to r \Leftarrow JC\, where\, LD$ occurring in P is transformed into a corresponding program rule for $f^{\mathcal{T}}$ in $P^{\mathcal{T}}$. We can assume that JC consists of joinability conditions $l_i == r_i$ and LD consists of local definitions $s_j \leftarrow d_j$ written in a textual order which fulfills the admissibility properties required for the conditions of program rules (see Section 2.2.1). Then the transformed program rule is constructed as follows:

$$
\begin{array}{l}
f^{\mathcal{T}}\ t_1^{\mathcal{T}} \ldots t_n^{\mathcal{T}} \rightarrow (R,T) \Leftarrow \ldots LS_i == RS_i \ldots \\
\quad \text{where}\{ \quad \begin{array}{ll}
 & \ldots \\
s_j^{\mathcal{T}} & \leftarrow d_j^{\mathcal{T}}; \\
 & \ldots \\
LS_i & \leftarrow l_i^{\mathcal{T}}; \\
RS_i & \leftarrow r_i^{\mathcal{T}}; \\
 & \ldots \\
R & \leftarrow r^{\mathcal{T}}; \\
T & \leftarrow \text{cNode}\ "f"\ [dVal\ t_1^{\mathcal{T}}, \ldots, dVal\ t_n^{\mathcal{T}}]\ (dVal\ R)\ "f.p"\ (clean\,[\,])\} \downarrow
\end{array}
\end{array}
$$

Some additional explanations are needed at this point:

- $t_l^{\mathcal{T}}$, $s_j^{\mathcal{T}}$, $d_j^{\mathcal{T}}$, $l_i^{\mathcal{T}}$, $r_i^{\mathcal{T}}$ and $r^{\mathcal{T}}$ refer to an expression transformation (defined below) which converts any $e\ ::\ \tau$ of signature Σ into $e^{\mathcal{T}}\ ::\ \tau^{\mathcal{T}}$ of signature $\Sigma^{\mathcal{T}}$.
-R, T, LS_i, RS_i are new fresh variables, and p is an index which represents the position of the program rule, in textual order.
-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 [20,25]. Any call (dVal a) (read: "demanded value of a") returns a representation of 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 $\perp$[6]. Moreover, dVal also renames all the identifiers of auxiliary functions $f_m^{\mathcal{T}}$ resp. $c_m^{\mathcal{T}}$ into f resp. c. In this way, the patterns representing computed results are translated back to the original signature.

The expression transformation $e \mapsto e^{\mathcal{T}}$ 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 $e^{\mathcal{T}}\ ::\ \tau^{\mathcal{T}}$ whenever $e\ ::\ \tau$, we use two auxiliary *application operators*:

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

These are used within $e^{\mathcal{T}}$ at those points where the application of a function from the translated signature (to a number of parameters equal to its arity) is expected to return a value paired with a CT. Applications of higher-order

[6] Because of this replacement of $\perp$ in place of unknown values, the basic facts occurring in proof trees must be understood as *approximation statements* rather than *equalities*.

variables are treated in a similar way. Formally:

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

From the previous specification it is easy to see that the translation $t^{\mathcal{T}}$ of a *pattern* t does not have any occurrences of the auxiliary application operators and is in fact another pattern, from which t can be univocally recovered. Coming back to the construction of translated program rules, we see that the translated expressions $t_l^{\mathcal{T}}$, $s_j^{\mathcal{T}}$, $d_j^{\mathcal{T}}$, $l_i^{\mathcal{T}}$, $r_i^{\mathcal{T}}$ and $r^{\mathcal{T}}$ are intended to ensure well-typing, but seemingly ignore CTs. In particular, the local definition of T renders a CT whose root has complete information about the arguments, result and program rule corresponding to a particular call to function f, but the list of children CTs seems to be empty. In fact this is not the case, because the local definitions $\{\ldots\}$ are further translated into $\{\ldots\}\downarrow$, which means that the normal form obtained by applying the transformation rules AP_0 and AP_1 defined below, with a leftmost-innermost strategy. The notation $e[e_1]$ must be undestood as an expression containing in occurrence of the subexpression e_1 in some context.

- AP_0:

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

- AP_1:

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

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

(i) AP_0 always finds $fun = g^{\mathcal{T}}$, for some nullary function symbol $g \in FS^0$.

(ii) AP_1 always finds $arg = s_m^{\mathcal{T}}$ for some pattern s_m in P's signature; and either fun is a variable, or else $fun = g^{\mathcal{T}}\ s_1^{\mathcal{T}} \ldots s_{m-1}^{\mathcal{T}}$ for some $g \in FS^m$, $m > 0$ and some patterns $s_1, \cdots, s_{m-1}$ in P's signature.

Each application of the AP transformations eliminates the currently leftmost-innermost occurrence of an application operator, while introducing a new local definition for the result R' and the computation tree T' coming from that application, and adding the pair (dVal R', T') to the list of children of T. The innermost strategy ensures that no application operators occur in the new local definition. Since the initial number of application operators is finite, the process is terminating and the normal form always exists. 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, by performing a simple partial evaluation by unfolding w.r.t. 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, the call to the auxiliary function clean is introduced, in order that the execution of (clean lp) *at run time* can build the ultimate list of children CTs. The definition of clean is such that all the pairs (pv,ct) in lp such that pv represents $\perp$ or ct is void are ignored, thus avoiding useless nodes to occur in the final CT. The program rules defining clean and some other auxiliary functions, shown below, must be included in any transformed program.

```
clean :: [(pVal, cTree)] → cTree
clean []                  →   []
clean ((R,T) : Rest)      →   clean Rest <= irrelevant (R,T) == true
clean ((R,T) : Rest)      →   T : clean Rest <= irrelevant (R,T) == false

irrelevant :: (pVal, cTree) → bool
irrelevant (R,T)          →   true <= isBottom R == true
irrelevant (R,T)          →   isVoid T <= isBottom R == false

isBottom :: pVal → bool
isBottom R   →   if R == "_" then true else false

isVoid :: cTree → bool
isVoid void                                     →   true
isVoid (cTree Fun Args Result Rule Children)    →   false
```

Note that the definition of isBottom uses a conditional expression, a language feature which is supported by $\mathcal{TOY}$, although not included in the formal presentation of FLP programs given in Section 2.1.2. This completes the description of the program transformation, except for the behaviour of dVal. This impure function cannot be defined by ordinary program rules, and it must be provided at some lower, implementation dependent level [7]. In our current

[7] Nevertheless, the requirements on dVal's behaviour needed to ensure the semantic correction of transformed programs can be formally specified; see the proof of Theorem 4.3

debugging tool for the FLP language $\mathcal{TOY}$, the fuction dVal is implemented in Prolog, as the rest of the $\mathcal{TOY}$ system. The final form of a transformed program rule is shown below.

$$
\begin{array}{lll}
f^{\mathcal{T}}\ t_1^{\mathcal{T}}\ \ldots\ t_n^{\mathcal{T}} \to (R,T) \Leftarrow \ldots LS_i == RS_i \ldots & & \\
\quad \text{where}\{ & & \ldots \\
 & (R_k, T_k) & \leftarrow call_k^{\mathcal{T}}; \\
 & & \ldots \\
 & s_j^{\mathcal{T}} & \leftarrow w_j^{\mathcal{T}}; \\
 & & \ldots \\
 & LS_i & \leftarrow u_i^{\mathcal{T}}; \\
 & RS_i & \leftarrow v_i^{\mathcal{T}}; \\
 & & \ldots \\
 & R & \leftarrow v^{\mathcal{T}}; \\
 & T & \leftarrow \text{cNode}\ \text{"}f\text{"}\ [dVal\ t_1^{\mathcal{T}}, \ldots, dVal\ t_n^{\mathcal{T}}]\ (dVal\ R)\ \text{"}f.p\text{"} \\
 & & \qquad\qquad (clean\,[\cdots, (dVal\ R_k, T_k), \cdots])\}
\end{array}
$$

Here, the transformed patterns $t_l^{\mathcal{T}}$ and $s_j^{\mathcal{T}}$ are as explained before, while the transformed patterns $w_j^{\mathcal{T}}$, $u_i^{\mathcal{T}}$, $v_i^{\mathcal{T}}$ and $v^{\mathcal{T}}$ are which remains from the transformed expressions $d_j^{\mathcal{T}}$, $l_i^{\mathcal{T}}$, $r_i^{\mathcal{T}}$ and $r^{\mathcal{T}}$ upon termination of the AP transformations. Moreover, the local definitions $(R_k, T_k) \leftarrow call_k^{\mathcal{T}}$ have been created by the AP transformations, applied in leftmost-innermost order. For each k, $call_k^{\mathcal{T}}$ is the transformed form of a function call $call_k$ in the original signature, which must have one of the two following forms:

(i) $call_k = g\ \overline{s}_m$, for some $g \in FS^m$, $m \geq 0$ and some patterns $\overline{s}_m$.

(ii) $call_k = F\ s$, for some variable F and some pattern s.

Note that the possible forms of $call_k$ correspond to the possible forms of fun and arg when the AP transformations are applied, as explained above.

4.5 An Example

Below we show part of the type declarations and program rules resulting from the transformation of the simple program from Section 2.2.2. For the sake of a simpler concrete syntax, we write "f'" instead of "$f^{\mathcal{T}}$" for translated symbols.

```
times' :: nat → nat → (nat, cTree)
times' (suc X) Y   → (R,T)
                     where  (M,T1) ←  times' X Y
                            (N,T2) ←  plus' X M
                                 R ←  N
                                 T ←  cNode "times" [dVal (suc X), dVal Y] (dVal R)
                                      "times.2" (clean [(dVal M,T1), (dVal N,T2)])
```

```
twice' :: (A → (A, cTree)) → A → (A, cTree)
twice' F X   → (R,T)
             where   (Y,T1) ← F X
                     (Z,T2) ← F Y
                          R ← Z
                          T ← cNode "twice" [dVal F, dVal X] (dVal R)
                              "twice.1" (clean [(dVal Y,T1), (dVal Z,T2)])

drop4' :: ([A] → ([A], cTree), cTree)
drop4'   → (R,T)
         where   (F,T1) ← twice' twice0' tail'
                      R ← F
                      T ← cNode "drop4" [] (dVal R)
                          "drop4.1" (clean [(dVal F,T1)]))
```

Note that the transformation of the program rule defining drop4 starts by transforming twice twice tail into $\mathsf{twice}^{\mathcal{T}}$ $\mathsf{twice}_0^{\mathcal{T}}$ @ $\mathsf{tail}^{\mathcal{T}}$, which gives rise to $(F, T_1) \leftarrow \mathsf{twice}^{\mathcal{T}}\ \mathsf{twice}_0^{\mathcal{T}}\ \mathsf{tail}^{\mathcal{T}}$ by application of an AP transformation. Careful examination of this example is left as an exercise for the reader.

4.6 Transforming Goals

The debugging process can be started whenever some answer θ computed for a goal G is considered erroneous by the user. For the sake of a simpler presentation, we will assume that G includes no local definitions. This is no serious limitation in practice. In order to build a suitable CT for the navigation phase, an auxiliary function definition

$$\mathsf{sol}\ \overline{X_n} = \mathsf{true} \Leftarrow G$$

is considered, whose translation is automatically added to the transformed program. Here, $\overline{X_n}$ are the variables occurring in G. Due to the assumption that G includes no local definitions, all these variables are allowed to occur as formal parameters of sol. Using the answer substitution θ which has been already computed by the goal solving system, the debugger can build the transformed goal

$$\mathsf{sol}^{\mathcal{T}}\ \overline{X_n}\theta^{\mathcal{T}} == (\mathsf{true},\ \mathsf{Tree})$$

As we will prove in Section 4.8, solving this goal with the transformed program leads to a solution which binds no variables in $\overline{X_n}\theta^{\mathcal{T}}$ and binds Tree to an APT witnessing $P \vdash_{SC} G\theta$. According to Theorem 2.5, navigating this APT leads to some buggy node which points to an incorrect instance of program rule in P.

In the case of our simple program from Section 2.2.2, a user could decide to activate the debugger after observing the wrong computed answer $\theta = \{\mathsf{Xs} \mapsto \mathsf{X{:}X{:}[\,]}\}$ for the goal

$$\mathsf{take\ (suc\ (suc\ z))\ (from\ X) == Xs}$$

In this situation, the debugger would use the transformed program to solve the goal

$$\mathsf{sol}^{\mathcal{T}}\ \mathsf{X\ (X{:}X{:}[\,]) == (true, Tree)}$$

This would bind the variable Tree to an APT essentially equivalent to the one shown in Fig. 2 [8], and debugging would proceed by navigating this APT.

4.7 Program Flattening

As a convenient technical device for proving some of the results in the coming section, we introduce another program transformation, called *flattening*. Intuitively, flattening a program means to eliminate nested function calls both in the right-hand sides and in the conditions of function defining rules. This can be done by introducing new local definitions.

The idea of flattening is not a new one. It played an important rôle in the operational semantics of K-LEAF, a pioneering functional logic language [7]. In our present context, flattening becomes important because of its close relationship to the transformation of program rules described above in Section 4.4. In fact, *flattening* a program rule for $f \in FS^n$ whose transformed form is as shown at the end of Section 4.4 yields, by the definition, the following:

$$\begin{array}{l} f\ t_1 \ldots t_n \rightarrow R \Leftarrow \ldots LS_i == RS_i \ldots \\ \quad \text{where}\{ \quad \begin{array}{ll} & \ldots \\ R_k & \leftarrow call_k; \\ & \ldots \\ s_j & \leftarrow w_j; \\ & \ldots \\ LS_i & \leftarrow u_i; \\ RS_i & \leftarrow v_i; \\ & \ldots \\ R & \leftarrow v \end{array} \\ \qquad\quad \} \end{array}$$

Flattening a whole program P is defined as the result of flattening one by one all the function defining rules belonging to P, which yields an intuitively equivalent program P_F with the same signature, called the *flat form* of P. For instance, the flat form of our simple program P from section 2.2.2 contains, among others, the program rules shown below. Their correspondence with the transformed program rules in $P^{\mathcal{T}}$ shown in Section 4.4 should be obvious.

[8] Due to the presence of the auxiliary function sol, the APT computed for Tree will be not formally identical to the APT from Fig. 2.

```
times :: nat → nat → nat
times (suc X) Y    → R
                where   M   ← times X Y
                        N   ← plus X M
                        R   ← N

twice:: (A → A) → A → A
twice F X    → R
          where   Y   ← F X
                  Z   ← F Y
                  R   ← Z

drop4 :: [A] → [A]
drop4    → R
      where   F   ← twice twice tail
              R   ← F
```

Note that programs in flat form only use flat function calls of the form $(f\, t_1 \ldots t_n)$, with $f \in FS^n$ and $t_1, \ldots, t_n$ patterns. Therefore proofs built in the semantic calculus SC do not need the inference rule AR when the program and the statement to be deduced are flat. This is because all the arguments of function calls met in the course of such a proof will necessarily be patterns. Let FSC be the variant of SC consisting of all the inference rules specified in Section 2.3.1, but with FA in place of $AR + FA$. The following result guarantees that the semantics of functions, as specified by the calculus SC, is preserved by flattening.

Theorem 4.1 *For every program P, for all $f \in FS^n$, and for all partial patterns $\overline{t_n}, s \in Pat_\perp$: $P \vdash_{SC} f\ \bar{t}_n \to s$ holds iff $P_F \vdash_{FSC} f\ \bar{t}_n \to s$. Moreover, the same witnessing APT can be chosen for both deductions.*

Proof Idea

This follows from a more general result which relates a SC deduction of the form $P \vdash_{SC} e \to s$ (with $e \in Exp_\perp$, $s \in Pat_\perp$) to a corresponding FSC deduction using the flat form of e. Building the proof relies on the recursive definition of a flattening transformation of expressions and program rules. In fact, this can be used as an alternative way to define the program rule transformation presented in Section 4.4. Details are left outside of the scope of this paper. □

4.8 Correctness Results

Now we are ready to present the three main results about the correctness of our program transformation, whose proofs are given in Appendix A. The first result concerns the type discipline. It guarantees that the debugger does not need to perform any type checking/inference before entering the CT generation phase, which proceeds as explained in Section 4.6.

Theorem 4.2 *The transformation $P^{\mathcal{T}}$ of a well-typed program P is always well-typed.*

The second result says that the semantics of any transformed function $f^{\mathcal{T}}$ in a transformed program $P^{\mathcal{T}}$ is the same as the semantics of f in the original program P, except that calls to $f^{\mathcal{T}}$ also return APTs, represented as values of type cTree.

Theorem 4.3 *Consider any n-ary function f and arbitrary partial patterns $\overline{t_n}$, t in the signature of a program P.*

(i) *Assume $P \vdash_{SC} f\ \bar{t}_n \rightarrow t$ and let apt be a witnessing APT for this deduction. Then $P^{\mathcal{T}} \vdash_{FSC^{\mathcal{T}}} f^{\mathcal{T}}\ \overline{t_n^{\mathcal{T}}} \rightarrow (t^{\mathcal{T}}, ct)$, where ct :: cTree is a total pattern which represents apt.*

(ii) *Assume $P^{\mathcal{T}} \vdash_{FSC^{\mathcal{T}}} f^{\mathcal{T}}\ \overline{t_n^{\mathcal{T}}} \rightarrow (t^{\mathcal{T}}, ct)$. Then $P \vdash_{SC} f\ \bar{t}_n \rightarrow t$.*

Proof Idea.
Due to Theorem 4.1, the SC deduction $P \vdash_{SC} f\ \bar{t}_n \rightarrow t$ can be replaced by the FSC deduction $P_F \vdash_{FSC} f\ \bar{t}_n \rightarrow t$ in the statement of the theorem. Intuitively, this makes the result plausible, due to the close correspondence between flat program rules and transformed program rules. The notation $FSC^{\mathcal{T}}$ refers to a variant of the flat semantic calculus FSC, which must be used for deductions with transformed programs. $FSC^{\mathcal{T}}$ consists of the inference rules of SC but with FA in place of $AR+FA$ and with the addition of special metarules which formalize the behaviour of the impure function dVal. Full details are given in Appendix A. □

Our last result shows that the goal transformation described in Section 4.6 is indeed suitable to generate correct APTs. Before presenting the theorem, we formalize certain assumptions about the undelying goal solving system. The theorem holds for every goal solving system which satisfies these assumptions.

Definition 4.4

(a) A *goal solving system GS* is assumed to produce an ordered sequence of computed answers θ_i for a given program P and a goal G. Each computed answer θ_i is assumed to be a substitution of patterns for variables occurring in G. We write $G \Vdash_{GS,P} \theta$ to indicate that θ is one of the answers for G computed by GS with program P. Similarly, we write $G \Vdash^{1st}_{GS,P} \theta$ to indicate that θ is the first answer for G computed by GS using program P.

(b) Given a goal solving system GS, we say
  (b.1) GS is *stable* iff for every program P and every goal G without local

definitions: if $G \Vdash_{GS,P} \theta$ then $sol\ \overline{X}_n\theta == true \Vdash^{1st}_{GS,P_{sol}} id$, where $P_{sol} = P \uplus \{sol\ \overline{X}_n \to true \Leftarrow G\}$, with a new n-ary function symbol sol and $\overline{X}_n = var(G)$.

(b.2) GS is *sound* iff for every program P and goal G, if $G \Vdash_{GS,P} \theta$ then $P \vdash_{SC} G\theta$.

(b.3) GS is *weakly complete* iff for every program P, for any $p :: \overline{\tau}_n \to bool$ and for all patterns $\overline{t}_n$ in $P's$ signature: If $p\ \overline{t}_n == true \Vdash^{1st}_{GS,P} id$, and apt is the APT for $P \vdash_{SC} p\ \overline{t}_n \to true$ witnessing the previous computation (which exists by soundness) and $P^{\mathcal{T}} \Vdash FSC^{\mathcal{T}}\ p^{\mathcal{T}}\ \overline{t^{\mathcal{T}}}_n \to (true, ct)$ where ct represents apt, then $p^{\mathcal{T}}\ \overline{t^{\mathcal{T}}}_n == (true, T) \Vdash^{1st}_{GS,P^{\mathcal{T}}} \{T \mapsto ct\}$.

(b.4) GS is *reasonable* iff GS is stable, sound and weakly complete.

The items of the previous definition are intended as minimal requirements that should be fulfilled by goal solving systems based on lazy narrowing strategies. Weak completeness is a sensible assumption because of Theorem 4.3 (i), and stability can be guarenteed by treating all the variables occurring in $\overline{X}_n\theta$ as constants when solving a goal $sol\ \overline{X}_n\theta == true$.

We believe that the goal solving system underlying $\mathcal{TOY}$ [14] is reasonable in the technical sense of Definition 4.4 but presently we do not intend to support our belief by a mathematical proof. It would be a very hard task, as any formal correctness proof for a complex software system.

Now we are in a position to state:

Theorem 4.5 *Let G be a goal with variables $\overline{X_n}$ and without local definitions. Assume that θ has been computed as an answer for G using program P. Consider the program P_{sol} obtained by adding to P the new auxiliary function definition* $\mathsf{sol}\ \overline{X_n} = \mathsf{true} \Leftarrow G$*. If the goal solving system is reasonable, solving the transformed goal* $\mathsf{sol}^{\mathcal{T}}\ \overline{X_n}\theta^{\mathcal{T}} == (\mathsf{true}, \mathsf{Tree})$ *with the transformed program $P^{\mathcal{T}}_{sol}$ succeeds. Moreover, the first computed answer binds no variables in $\overline{X_n}\theta^{\mathcal{T}}$ and binds* Tree *to an APT wittnessing $P \vdash_{SC} G\theta$.*

A proof of this theorem can be found in Appendix A. Although the result holds for any computed answer θ, its interest for debugging is restricted to the case that θ is seen by the user as a wrong computed answer. In this case, the debugger can find an incorrect program rule by navigating the APT, as explained in Section 4.6.

5 Navigating the CTs by Oracle Querying

In this Section we present a technique used by our debugger to avoid redundant questions to the oracle during the navigation phase. We also present a simple example of debugging session. More examples can be found in Appendix B.

Once the CT associated to a wrong answer has been built (as described in

Section 4.6), navigation performs a top-down traversal, asking the oracle about the validity of the *basic facts* associated to the visited nodes (except for 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, as we have seen in Section 2.3.4. 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 in Section 2.1.2. By definition, 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

$$t_1\theta \sqsubseteq s_1,\ \ldots\ , t_n\theta \sqsubseteq s_n, s \sqsubseteq t\theta$$

Due to Proposition 2.1 item (i), we can also write these conditions as:

$$s_1 \to t_1\theta,\ \ldots\ , s_n \to t_n\theta, t\theta \to s$$

Entailment between basic fact can be decided by means of the next algorithm.

Algorithm

Let $f\ \overline{t}_n \to t$ and $f\ \overline{s}_n \to s$ be two basic facts which share no common variables. In order to decide whether $f\ \overline{t}_n \to t\ \succeq\ f\ \overline{s}_n \to s$ we define a system of transformations, somewhat similar to those used in Martelli and Montanari's unification algorithm. The transformations are applied to a multiset S of approximation statements $a \to b$, with $a, b \in Pat_\perp$, together with a set of variables W. Both are represented together in the form: $S \Box W$, which we will call a *configuration* from now on.

We say that *$S\theta$ holds*, with $\theta \in Subst$, iff for all $s \to t \in S, t\theta \sqsubseteq s\theta$. The *set of solutions* of a configuration $S \Box W$ is defined as the set of total substitutions over variables in W for which all the approximation statements in S do hold, i.e.: $Sol(S \Box W)\ =\ \{\theta \in Subst \mid dom(\theta) \subseteq W, ran(\theta) \subseteq Pat, S\theta \text{ holds } \}$.

The purpose of the algorithm is to find some solution for the initial configuration $S_0 \Box W_0$ with $S_0 = s_1 \to t_1, \ldots, s_n \to t_n,\ t \to s$ and $W_0 = var(f\ \overline{t}_n \to t)$, i.e. we indicate that only variables in $f\ \overline{t}_n \to t$ can be instantiated. At each step of the algorithm a configuration $S_i \Box W_i$ is transformed into a new one $S_{i+1} \Box W_{i+1}$ producing a substitution θ_{i+1}. This is done by applying some (non-deterministically) selected transformation rule to any (non-deterministically) selected element $a \to b$ of S_i. Such step can be represented as

$$\underbrace{a \to b, S}_{S_i} \Box W_i \quad \vdash_{\theta_i} \quad S_{i+1} \Box W_{i+1}$$

For the sake of simplicity, sometimes we will write a configuration $S_i \Box W_i$ as K_i. The transformation rules are presented below.

Transformation Rules

In the following we assume $X, Y \in Var$ with $X \in W$; $a_k, b_k, t \in Pat_\perp$; $s \in Pat$; and $h \in DC \cup FS$. Moreover, X_k represent new, fresh variables.

$$\begin{array}{lll}
R1\ Y \to Y,\ S\Box W & \vdash_{id} & S\Box W \\
R2\ t \to \perp,\ S\Box W & \vdash_{id} & S\Box W \\
R3\ h\ \overline{a}_m \to h\ \overline{b}_m,\ S\Box W & \vdash_{id} & \ldots, a_k \to b_k, \ldots S\Box W \\
R4\ s \to X,\ S\Box W & \vdash_{\{X \mapsto s\}} & S\{X \mapsto s\}\Box W \\
R5\ X \to Y,\ S\Box W & \vdash_{\{X \mapsto Y\}} & S\{X \mapsto Y\}\Box W \\
R6\ X \to h\ \overline{a}_m,\ S\Box W & \vdash_{\{X \mapsto h\ \overline{X}_m\}} & \ldots, X_k \to a_k, \ldots\ S\{X \mapsto h\ \overline{X}_m\}\Box W, \overline{X}_m
\end{array}$$

The algorithm finishes when a configuration is reached s.t. no transformation can be applied. Next theorem ensures that such configuration always exists, as well as its relationship with the entailment. The proof can be found in Appendix A.

Theorem 5.1 *The algorithm described above always stops in some configuration $S_j \Box W_j$ which cannot be further transformed. Moreover, the initial entailment $f\ \overline{t}_n \to t\ \succeq\ f\ \overline{s}_n \to s$ holds iff $S_j = \emptyset$.*

Now, the interest of the entailment for declarative debugging is justified by the next result.

Theorem 5.2 *Entailment between basic facts is a decidable preorder. Moreover, any intended model given as a Herbrand interpretation $\mathcal{I}$ is closed under entailment, i.e. 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.

The fact that Herbrand interpretations are closed under entailment is a straightforward consequence from the definition of the entailment relation and conditions (ii), (iii) in the definition of Herbrand interpretation (see Section 2.3.4). The definition of entailment also implies easily that $\succeq$ is a reflexive and transitive relation, and thus a preorder. In order to prove that $\succeq$ is decidable, let us consider two arbitrary basic facts $f\ \overline{t}_n \to t$, $f\ \overline{s}_n \to s$ and choose any renaming ρ such that $(f\ \overline{t}_n \to t)\rho$ and $f\ \overline{s}_n \to s$ share no variables. By definition of entailment, (a) and (b) below are equivalent:

(a) $f\ \overline{t}_n \to t\ \succeq\ f\ \overline{s}_n \to s$

(b) $(f\ \overline{t}_n \to t)\rho\ \succeq\ f\ \overline{s}_n \to s$

Finally, Theorem 5.1 ensures that (b) can be decided by applying the algorithm described above to the initial configuration $S_0 \Box W_0$, where:

$$S_0 = s_1 \to t_1\rho, \ldots, s_n \to t_n\rho,\; t\rho \to s \qquad W_0 = var((f\ \bar{t}_n \to t)\rho)$$

$\Box$

Thanks to Theorem 5.2 an oracle 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, 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. 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. In both cases, a question to the oracle can be avoided.

Our debugger has been implemented as part of the $\mathcal{TOY}$ system. A prototype version can be downloaded from http://titan.sip.ucm.es/toy/. Here we show a debugging session for a program which contains the wrong definition of the function times already discussed in Section 2.2.2, along with correct definitions of the functions head, tail, map and from. 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. The questions asked by the debugger and the answers given by the user are as follows:

```
Consider the following facts:
1: from X → X:suc X:⊥
2: map (times (suc z)) (X:suc X:⊥) → ⊥:z:⊥
3: tail (⊥:z:⊥) → z:⊥
4: head (z:⊥) → z
Are all of them valid? ([y]es / [n]o) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: map (times (suc z)) (suc X:⊥) → z:⊥
Are all of them valid? ([y]es / [n]o) / [a]bort) n

Consider the following facts:
1: times (suc z) (suc X) → z
Are all of them valid? ([y]es / [n]o) / [a]bort) n

Consider the following facts:
1: times z (suc X) → z
2: plus z z → z
Are all of them valid? ([y]es / [n]o) / [a]bort) y

Rule number 2 of the function times is wrong.
Wrong instance: times (suc z) (suc X) → (plus (times z (suc X)) z)
```

As shown by this example, our current prototype debugger searches the CT top-down, using a strategy whose aim is to avoid redundant questions and to give freedom to the oracle. At any point during the search, the current node contains an invalid statement (initially, this is true because the root of the CT corresponds to an error symptom detected by the user). The debugger builds the list L of the basic facts attached to the children of the current node. If some member of L entails a fact known to be invalid from some previous oracle answer, the debugger moves to the corresponding child and continues with the same strategy. Otherwise, the debugger displays the list L for the oracle's consideration. If the oracle regards all the facts in L as valid, then the current node is buggy, and the debugger shows its associated program rule instance (which can be computed from the CT) as responsible for the bug. Otherwise, the oracle must choose some erroneous fact in the list. The debugger adds this fact to its store of invalid facts, moves to the corresponding child node, and continues with the same strategy.

In the simple example shown above, the entailment relation is not helpful, but in more involved cases it can reduce the number of questions asked to the oracle. Note that the particular search strategy we have described is such that all the answers provided by the oracle are negative, except for the last question. This might not be the case in other alternative strategies, which we have not yet investigated. 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 FLP languages [22,20,25]. We have given a new, more formal specification of this technique, which avoids type errors related to the use of curried functions and preserves both well-typing and program semantics (as formalized in [9,2]), independently of the narrowing strategy chosen as goal solving mechanism. A prototype implementation of our debugger for the functional logic language $\mathcal{TOY}$ [14] is available. Our implementation uses a semantically correct algorithm to detect and avoid redundant questions to the oracle, thus reducing the complexity of debugging.

In order to improve the practical usefulness of our results, we have started a cooperation with Herbert Kuchen and Wolfgang Lux, to include a similar debugger as a tool within the Curry [12] implementation developed at Münster University. Hopefully, this will eventually help to evaluate the debugger on practical applications. We also plan to implement and evaluate alternative search strategies for the navigation phase. As more substantial research work, we plan to investigate and implement extensions of the debugger, to support constraint-based computations as well as the diagnosis of missing answers.

Acknowledgements
We are grateful to Herbert Kuchen, Paco López and Wolfgang Lux, who have 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] M. Alpuente, J. Correa and M. Falaschi. *A Debugging Scheme for Functional Logic Programs*. Proc. WFLP'2001, Kiel, Germany, September 13–15, 2001.

[2] R. Caballero, F.J. López-Fraguas and M. Rodríguez-Artalejo. *Theoretical Foundations for the Declarative Debugging of Lazy Functional Logic Programs*. In Proc. FLOPS'01, Springer LNCS 2024, 170–184, 2001.

[3] M. Comini, G. Levi, M.C. Meo and G. Vitello. *Abstract Diagnosis*. J. of Logic Programming 39, 43–93, 1999.

[4] L. Damas and R. Milner. *Principal Type Schemes for Functional Programs*. Proc. ACM Symp. on Principles of Programming Languages (POPL'82), ACM Press, pp. 207–212, 1982.

[5] M. Falaschi, G. Levi, M. Martelli and C. Palamidessi. *A Model-theoretic Reconstruction of the Operational Semantics of Logic Programs*. Information and Computation 102(1). pp. 86-113, 1993.

[6] G. Ferrand. *Error Diagnosis in Logic Programming, an Adaptation of E.Y. Shapiro's Method*. The Journal of Logic Programming 4(3), 177-198, 1987.

[7] E. Giovannetti, G. Levi, C. Moiso and C.Palamidessi. *Kernel-LEAF: A Logic plus Functional Language*. Journal of Computer and System Science 42(2), pp. 139–185, 1991.

[8] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas and M. Rodríguez-Artalejo. *An Approach to Declarative Programming Based on a Rewriting Logic*. The Journal of Logic Programming 40(1), pp. 47–87, 1999.

[9] J.C. González-Moreno, M.T. Hortalá-González and M. Rodríguez-Artalejo. *Polymorphic Types in Functional Logic Programming*. FLOPS'99 special issue of the Journal of Functional and Logic Programming, 2001. See http://danae.uni-muenster.de/lehre/kuchen/JFLP

[10] C.A. Gunter and D. Scott. *Semantic Domains*. In J.van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Elsevier and The MIT Press, Vol. B, Chapter 6, pp. 633–674, 1990.

[11] 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.

[12] M. Hanus (ed.). *Curry: An Integrated Functional Logic Language*. Version 0.7, February 2, 2000.
Available at http://www.informatik.uni-kiel.de/curry/.

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

[14] 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://titan.sip.ucm.es/toy.

[15] R. Milner. *A Theory of Type Polymorphism in Programming*. Journal of Computer and Systems Sciences, 17, pp. 348–375, 1978.

[16] B. Möller. *On the Algebraic Specification of Infinite Objects - Ordered and Continuous Models of Algebraic Types*. Acta Informatica 22, pp. 537–578, 1985.

[17] L. Naish. *Declarative Diagnossing of Missing Answers*. New Generation Computing, 10, 255–385, 1991.

[18] L. Naish. *Declarative Debugging of Lazy Functional Programs*. Australian Computer Science Communications, 15(1):287–294, 1993.

[19] L. Naish. *A Declarative Debugging Scheme*. Journal of Functional and Logic Programming, 1997-3.

[20] L. Naish and T. Barbour. *Towards a Portable Lazy Functional Declarative Debugger*. Australian Computer Science Communications, 18(1):401–408, 1996.

[21] H. Nilsson, P. Fritzson. *Algorithmic Debugging of Lazy Funcional Languages*. The Journal of Functional Programming, 4(3):337-370, 1994.

[22] H. Nilsson and J. Sparud. *The Evaluation Dependence Tree as a basis for Lazy Functional Debugging*. Automated Software Engineering, 4(2):121–150, 1997.

[23] H. Nilsson. *Declarative Debugging for Lazy Functional Languages*. Ph.D. Thesis. Dissertation No. 530. Univ. Linköping, Sweden. 1998.

[24] S.L. Peyton Jones (ed.), J. Hughes (ed.), L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M.P. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman and P. Wadler. *Report on the programming language Haskell 98: a non-strict, purely functional language*. Available at http://www.haskell.org/definition, February 1999.

[25] B. Pope. *Buddha. A Declarative Debugger for Haskell*. Honours Thesis, Department of Computer Science, University of Melbourne, Australia, June 1998.

[26] E.Y. Shapiro. *Algorithmic Program Debugging*. The MIT Press, Cambridge, Mass., 1982.

[27] J. Sparud. *Tracing and Debuggging Lazy Functional Computations*. PhD Thesis. Department of Computer Science, Chalmers Universitity of Technology. Göteborg, Sweden, 1999.

[28] 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.

[29] P. Wadler. *Why no one uses Functional Languages*. SIGPLAN Notices 33(8), 23–27, 1998.

7 Appendix A: Proofs of the main results

7.1 Proofs of Results from Section 2

Proof of Theorem 2.2
This theorem is also based on some auxiliary lemmata.

Lemma 7.1 *For any given Herbrand interpretation $\mathcal{I}$, the analogous of Proposition 2.1 holds for the calculus $SC_{\mathcal{I}}$, i.e.:*

(i) *For all $t,\ s \in Pat_{\perp}$: $t \rightarrow s$ is valid in $\mathcal{I}$ iff $t \sqsupseteq s$.*
(ii) *For all $e \in Exp_{\perp}$, $t,\ s \in Pat_{\perp}$: if $e \rightarrow t$ is valid in $\mathcal{I}$ and $t \sqsupseteq s$, then $e \rightarrow s$ is also valid in $\mathcal{I}$.*
(iii) *For all $e \in Exp_{\perp}$, $t \in Pat_{\perp}$ and $\theta,\ \theta' \in Subst_{\perp}$ such that $e\theta \rightarrow t$ is valid in $\mathcal{I}$ and $\theta \sqsubseteq \theta'$, the statement $e\theta' \rightarrow t$ is also valid in $\mathcal{I}$, with a $SC_{\mathcal{I}}$ proof of the same size and structure.*
(iv) *For all $e \in Exp_{\perp}$, $s \in Pat_{\perp}$ such that $e \rightarrow s$ is valid in $\mathcal{I}$, the statement $e\theta \rightarrow s\theta$ is also valid in $\mathcal{I}$ for every* total *substitution $\theta \in Subst$.*

Proof Idea. This can be proved by straightforward induction on the size of $SC_{\mathcal{I}}$ derivations, similarly to Proposition 2.1. □

Lemma 7.2 *Assume a Herbrand interpretation $\mathcal{I}$, a partial expression $e \in Exp_{\perp}$ and a partial pattern $t \in Pat_{\perp}$. Then $[\![e]\!]^{\mathcal{I}} \supseteq [\![t]\!]^{\mathcal{I}}$ iff $t \in [\![e]\!]^{\mathcal{I}}$.*

Proof. Assume $[\![e]\!]^{\mathcal{I}} \supseteq [\![t]\!]^{\mathcal{I}}$. By Lemma 7.1, $t \rightarrow t$ is valid in $\mathcal{I}$. Then $t \in [\![t]\!]^{\mathcal{I}}$ and therefore $t \in [\![e]\!]^{\mathcal{I}}$. Conversely, suppose that $t \in [\![e]\!]^{\mathcal{I}}$. Then $e \rightarrow t$ is valid in $\mathcal{I}$, and for all $s \in [\![t]\!]^{\mathcal{I}}$, $t \rightarrow s$ is also valid in $\mathcal{I}$. By Lemma 7.1 it follows that e $\rightarrow s$ is valid in $\mathcal{I}$ for all $s \in [\![t]\!]^{\mathcal{I}}$, which means $[\![e]\!]^{\mathcal{I}} \supseteq [\![t]\!]^{\mathcal{I}}$. □

Lemma 7.3 *Let $\mathcal{I}$ a Herbrand interpretation and $f\ \bar{t}_n \rightarrow s$ a basic fact. Then $f\ \bar{t}_n \rightarrow s$ is valid in $\mathcal{I}$ iff $(f\ \bar{t}_n \rightarrow s) \in \mathcal{I}$.*

Proof. If $s = \perp$ the result holds because $f\,\bar{t}_n \rightarrow \perp$ belongs to every Herbrand interpretation and $f\,\bar{t}_n \rightarrow \perp$ is valid in $\mathcal{I}$ due to the $SC_{\mathcal{I}}$ rule BT. In the rest of the proof we assume that s is not $\perp$.

If $(f\ \bar{t}_n \rightarrow s) \in \mathcal{I}$ then $f\ \bar{t}_n \rightarrow s$ is valid in $\mathcal{I}$, as witnessed by the following $SC_{\mathcal{I}}$ derivation, ending with a $FA_{\mathcal{I}}$ step:

$$\frac{t_1 \rightarrow t_1\ \ldots\ t_n \rightarrow t_n \quad s \rightarrow s}{f\ \bar{t}_n \ \rightarrow s}\ (f\ \bar{t}_n\ \rightarrow\ s) \in \mathcal{I}$$

The derivation can be completed because Lemma 7.1 ensures that all the premises $t_i \rightarrow t_i$ and $s \rightarrow s$ are valid in $\mathcal{I}$.

Conversely, if $f\ \bar{t}_n \rightarrow s$ is valid in $\mathcal{I}$, there is a $SC_{\mathcal{I}}$ proof of $f\ \bar{t}_n \rightarrow s$, which must end with a $FA_\mathcal{I}$ step and have the following form:

$$\frac{t_1 \rightarrow t'_1 \ \ldots \ t_n \rightarrow t'_n \quad s' \rightarrow s}{f\ \overline{t}_n \ \rightarrow s} \ (f\ \overline{t'}_n \ \rightarrow \ s') \in \mathcal{I}$$

By Lemma 7.1 we can conclude that $t'_1 \sqsubseteq t_1,\ \ldots,\ t'_n \sqsubseteq t_n$ and $s \sqsubseteq s'$. Since $(f\ \overline{t'}_n \ \rightarrow \ s') \in \mathcal{I}$, item (ii) from the definition of Herbrand interpretation in Section 2.3.4 implies $(f\ \overline{t}_n \rightarrow s) \in \mathcal{I}$. □

Lemma 7.4 *Let P any program and $\mathcal{M}_P = \{f\ \overline{t}_n \rightarrow s \mid P \vdash_{SC} f\ \overline{t}_n \rightarrow s\}$. Then $\mathcal{M}_P$ is a Herbrand interpretation.*

Proof. M_P must satisfy the three conditions of Herbrand interpretations:

(i) $(f\,\overline{t}_n \rightarrow \perp) \ \in \mathcal{M}_P$.
This property holds since $f\,\overline{t}_n \rightarrow \perp$ can be proved in SC by means of the BT rule.

(ii) If $(f\ \overline{t}_n \rightarrow s) \in \mathcal{M}_P$, $t_i \sqsubseteq t'_i, s \sqsupseteq s'$ then $(f\,\overline{t'}_n \rightarrow s') \ \in \mathcal{M}_P$.
This follows immediately from the definition of $\mathcal{M}_P$ and Proposition 2.1.

(iii) If $(f\ \overline{t}_n \rightarrow s) \ \in \mathcal{M}_P$ and $\theta \in Subst$ is a *total substitution*, then $(f\ \overline{t}_n \rightarrow s)\theta \in \mathcal{M}_P$. This is also a straigthforward consequence of Proposition 2.1 and the construction of $\mathcal{M}_P$.

□

We are now ready to prove claims (a), (c) and (b) of Theorem 2.2, in this order.

(a) Let φ be a statement such that $P \vdash_{SC} \varphi$ and assume that $\mathcal{I}$ is a Herbrand model of P. Let T be the proof tree of φ in SC. We will build a proof tree T' of φ in $SC_\mathcal{I}$, showing that φ is valid in $\mathcal{I}$. This is done by using induction on the depth of T.

Basis: (depth(T)= 0). Then φ is the only node of T and corresponds either to a BT, DC or to a RR inference. Since these rules are also present in $SC_\mathcal{I}$ we can take $T' = T$.

Inductive step: (depth(T)= n, $n > 0$). We distinguish different cases depending on the SC rule applied at the root of T.

DC: In this case φ must have the form $h\ \overline{e}_m \rightarrow h\ \overline{t}_m$ and the inference step at the root of T must be:

$$\frac{e_1 \rightarrow t_1 \ \ldots \ e_m \rightarrow t_m}{h\ \overline{e}_m \rightarrow h\ \overline{t}_m}$$

Since rule DC also exists in $SC_\mathcal{I}$ we can build T' with the same root as T and with the same children at the root. Moreover by the induction hypothesis the $e_i \rightarrow t_i$ are valid in $\mathcal{I}$ and therefore there exist proof trees in $SC_\mathcal{I}$ that will complete the construction of T'.

JN: In this case φ has the form $e == e'$ and the inference step at the root of T must be:

$$\frac{e \to t \qquad e' \to t}{e \;==\; e'}$$

where t is a total pattern. Since rule JN also exists in $SC_{\mathcal{I}}$ we can build T' with the same root as T and with the same children at the root. Morover by the induction hypothesis the $e \to t$, $e' \to t$ are valid in $\mathcal{I}$ and therefore there exist proof trees in $SC_{\mathcal{I}}$ that will complete the construction of T'.

AR + FA: In this case φ has the form $f\ \overline{e}_n\ \overline{a}_m \to t$ and the SC inference at the root of T must be:

$$\frac{e_1 \to t_1\ \ldots\ e_n \to t_n \quad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t}_n \to s}} \quad s\ \overline{a}_m \to t}{f\ \overline{e}_n\ \overline{a}_m \to t}$$

Then we build T' by using rule $FA_{\mathcal{I}}$ at the root:

$$\frac{e_1 \to t_1\ \ldots\ e_n \to t_n \quad s\ \overline{a}_m \to t}{f\ \overline{e}_n \overline{a}_m \to t}\ (f\ \overline{t}_n\ \to\ s) \in \mathcal{I}$$

and completing the proof tree by means of $SC_{\mathcal{I}}$ proof trees for the statements $e_i\ \to\ t_i$ and $s\ \overline{a}_m\ \to\ t$, which exist by induction hypothesis, since all these statements have proof trees of depth less than n in SC. However, we still have to check that the conditions required by $FA_{\mathcal{I}}$ are satisfied. First t is actually a pattern different from $\perp$ because this condition is also required by rule $AR + FA$. To see that $f\ \overline{t}_n\ \to\ s$ is in $\mathcal{I}$ we observe that $f\ t_1 \ldots t_n\ \to\ t \Leftarrow C$ is an instance of some rewrite rule belonging to P. Moreover, $\mathcal{I}$ satisfies C by induction hypothesis. Since $\mathcal{I}$ is a model of P, we can conclude that $[\![f\ t_1 \ldots t_n]\!]^{\mathcal{I}} \supseteq [\![r]\!]^{\mathcal{I}}$. By induction hypothesis we also know that $r \to s$ is valid in $\mathcal{I}$. It follows that $s \in [\![r]\!]^{\mathcal{I}} \subseteq [\![ft_1 \ldots t_n]\!]^{\mathcal{I}}$ and hence $s \in [\![ft_1 \ldots t_n]\!]^{\mathcal{I}}$, as we needed.

(c) $\mathcal{M}_P$ is an Herbrand interpretation as shown in Lemma 7.4. Assume that φ is valid in $\mathcal{M}_P$ with proof tree T in $SC_{\mathcal{M}_P}$. Then we show by induction on $depth(T)$ that we can build a proof tree T' for φ in SC.

Basis ($depth(T) = 0$). The only possible inferences applied at the root of T are BT, RR or DC. Since these 3 rules belong also to SC we can take $T' = T$.

Inductive Step ($depth(T) = n, n > 0$). Then either DC, JN or $FA_{\mathcal{M}_P}$ has been applied at the root of T. In the DC and JN cases, the same inference can be applied at the root of T' and by induction hypothesis SC proof trees T_i exist all the children. This completes the desired proof tree T'.

In the $FA_{\mathcal{M}_P}$ case the root inference of T has the form

$$\frac{e_1 \to t_1 \ \ldots \ e_n \to t_n \quad s\ \overline{a}_m \to t}{f\ \overline{e}_n\,\overline{a}_m \to t}\ t \text{ pattern},\ t \neq \perp,\ (f\ \overline{t}_n\ \to\ s) \in \mathcal{M}_P$$

Since $(f\ \overline{t}_n\ \to\ s)\ \in \mathcal{M}_P$, then there exists a SC proof tree for $f\ \overline{t}_n\ \to\ s$. Such a proof tree must have a $AR + FA$ inference at the root:

$$\frac{t_1 \to t'_1 \ \ldots \ t_n \to t'_n \quad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t'}_n \to s}}}{f\ \overline{t}_n \to s}\ (f\ \overline{t'}_n \to r \Leftarrow C) \in [P]_\perp,\ s \text{ pattern},\ s \neq \perp$$

Hence, the statements $t_i \to t'_i$, C and $r \to s$ have proof trees in SC. Then the tree T' is built by using a FA inference at the root:

$$\frac{e_1 \to t'_1 \ \ldots \ e_n \to t'_n \quad \dfrac{C \quad r \to s}{\boxed{f\ \overline{t'}_n \to s}} \quad s\ \overline{a}_m \to t}{f\ \overline{e}_n\ \overline{a}_m \to t}\ \begin{array}{l} s \text{ pattern} \\ f\ \overline{t'}_n \to r \Leftarrow C \in [P]_\perp, \\ t \text{ pattern},\ t \neq \perp \end{array}$$

To complete T' we only need to show that the statements $e_i\ \to\ t'_i$ have SC proofs. This follows easily from Proposition 2.1, since each $e_i \to t_i$ has a SC proof by induction hypothesis, and $t_i \to t'_i$ have also SC proofs.

(b) The fact that $\mathcal{M}_P$ is included in any Herbrand model of P follows from Lemma 7.3, the construction of $\mathcal{M}_P$ and item (a) of this theorem. Moreover, we already know by Lemma 7.4 that $\mathcal{M}_P$ is a Herbrand interpretation. In order to show that $\mathcal{M}_P$ is a model of P, we must prove that $\mathcal{M}_P$ satisfies every program rule instance $(f\ \overline{t}_n \to r \Leftarrow C) \in [P]_\perp$. This is trivially true if $\mathcal{M}_P$ does not satisfy C. Assuming that $\mathcal{M}_P$ does satisfy C, we have to prove that $[\![r]\!]^{\mathcal{M}_P}\ \subseteq\ [\![f\ \overline{t}_n]\!]^{\mathcal{M}_P}$. This means that any $t\ \in\ Pat_\perp$ such that $r\ \to\ t$ is valid in $\mathcal{M}_P$ must verify that $f\ \overline{t}_n \to t$ is also valid in $\mathcal{M}_P$. By item (c) of this theorem and the construction of $\mathcal{M}_P$, it suffices to prove that $P \vdash_{SC} f\ \overline{t}_n \to t$ under the assumption that $P\ \vdash_{SC}\ r\ \to\ t$. If $t\ =\ \perp$ this is trivially true. Otherwise we build the following SC proof tree with an $AR + FA$ inference at the root:

$$\frac{t_1 \to t_1 \ \ldots \ t_n \to t_n \quad \dfrac{C \quad r \to t}{\boxed{f\ \overline{t}_n \to t}}}{f\ \overline{t}_n \to t}\ f\ \overline{t}_n \to r \Leftarrow C \in [P]_\perp,\ t \neq \perp$$

Note that this proof tree can be completed, since:

- $P \vdash_{SC} t_i \to t_i$ holds by Proposition 2.1.
- Each statement φ' in C is valid in $\mathcal{M}_P$, and thus fulfils $P \vdash_{SC} \varphi'$, by item (c) of this theorem.
- $P \vdash_{SC} r \to t$ is assumed to hold.

□

7.2 Proofs of Results from Section 4

Proof of Theorem 4.2

In what follows, we use the following notations:

- $\Sigma^{\mathcal{T}}$ stands for a transformed signature, defined as explained in Section 4.2
- $T^{\mathcal{T}}$ stands for a transformed type environment, defined by the condition $T^{\mathcal{T}}(X) = T(X)^{\mathcal{T}}$ for all $X \in Var$.
- $\theta^{\mathcal{T}}$ stands for a transformed type substitution, defined by the condition $\theta^{\mathcal{T}}(\alpha) = \theta(\alpha)^{\mathcal{T}}$ for all $\alpha \in TVar$.

Now we present some auxiliary lemmata.

Lemma 7.5 *Let $T^{\mathcal{T}}$ be a type environment and $a_1^{\mathcal{T}} \ldots a_k^{\mathcal{T}}$, $b^{\mathcal{T}}$ expressions such that*
1) $(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} b^{\mathcal{T}} :: (\tau_1 \to \ldots \to \tau_k \to \tau)^{\mathcal{T}}$
2) $(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} a_i^{\mathcal{T}} :: \tau_i^{\mathcal{T}}$ for every $i = 1, \ldots, k$.
Then $(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} (\ldots((b^{\mathcal{T}} \ @ \ a_1^{\mathcal{T}}) \ @ \ a_2^{\mathcal{T}})\ldots) \ @ \ a_k^{\mathcal{T}} :: \ \tau^{\mathcal{T}}$.

Proof

Induction on $k \geq 0$.

Basis: $k = 0$. Then $(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} b^{\mathcal{T}} :: \tau^{\mathcal{T}}$, and the result holds.

Inductive Case: $k > 0$. Let $e = (\ldots((b^{\mathcal{T}} \ @ \ a_1^{\mathcal{T}}) \ @ \ a_2^{\mathcal{T}})\ldots) \ @ \ a_{k-1}^{\mathcal{T}}$. By I.H.:

$$(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} e :: (\tau_k \to \tau)^{\mathcal{T}} = \tau_k^{\mathcal{T}} \to (\tau^{\mathcal{T}}, cTree)$$

Now, since $@ :: (\alpha \to (\beta, cTree)) \to \alpha \to \beta$, it is clear that $(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} e \ @ \ a_k^{\mathcal{T}} :: \tau^{\mathcal{T}}$. □

Lemma 7.6 *For any type τ and any type substitution θ: $\tau^{\mathcal{T}}\theta^{\mathcal{T}} = (\tau\theta)^{\mathcal{T}}$.*

Proof

Induction on the structure of τ.

Basis: $\tau = \alpha \in TVar$. Then $\alpha^{\mathcal{T}}\theta^{\mathcal{T}} = \alpha\theta^{\mathcal{T}} = (\alpha\theta)^{\mathcal{T}}$ (by def. of $\theta^{\mathcal{T}}$).

Inductive Case. Two possibilities:

(a) $\tau = c\ \tau_1 \ldots \tau_n$, for some $c \in DC^n$. Then:

$$\begin{array}{lcl}
(c\ \tau_1 \ldots \tau_n)^{\mathcal{T}} \theta^{\mathcal{T}} & = & \\
(c\, \tau_1^{\mathcal{T}} \ldots \tau_n^{\mathcal{T}}) \theta^{\mathcal{T}} & = & \\
c\ (\tau_1^{\mathcal{T}} \theta^{\mathcal{T}}) \ldots (\tau_n^{\mathcal{T}} \theta^{\mathcal{T}}) & = & \text{(by I.H.)} \\
c\ (\tau_1 \theta)^{\mathcal{T}} \ldots (\tau_n \theta)^{\mathcal{T}} & = & \\
(c\ (\tau_1 \theta) \ldots (\tau_n \theta))^{\mathcal{T}} & = & \\
((c\ \tau_1 \ldots \tau_n)\theta)^{\mathcal{T}} & &
\end{array}$$

(b) $\tau = \mu \rightarrow \nu$

$$\begin{array}{lcl}
(\mu \rightarrow \nu)\theta^{\mathcal{T}} & = & \\
(\mu^{\mathcal{T}} \rightarrow (\nu^{\mathcal{T}}, cTree))\theta^{\mathcal{T}} & = & \\
\mu^{\mathcal{T}} \theta^{\mathcal{T}} \rightarrow (\nu^{\mathcal{T}} \theta^{\mathcal{T}}, cTree) & = & \text{(by I.H.)} \\
(\mu\theta)^{\mathcal{T}} \rightarrow ((\nu\theta)^{\mathcal{T}}, cTree) & = & \\
(\mu\theta \rightarrow \nu\theta)^{\mathcal{T}} & = & \\
((\mu \rightarrow \nu)\theta)^{\mathcal{T}} & &
\end{array}$$

$\square$

Lemma 7.7 *The expression transformation $e \mapsto e^{\mathcal{T}}$ defined in Section 4.4 transforms any well typed expression $(\Sigma, T) \vdash_{WT} e :: \tau$ into a well typed expression $(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} e^{\mathcal{T}} :: \tau^{\mathcal{T}}$.*

Proof
We distinguish the same six cases as in the definition of $e^{\mathcal{T}}$ given in Section 4.4.

1. $e = X\ a_1 \ldots a_k,\ X \in Var, k \geq 0.$
In this case, $e^{\mathcal{T}} = (\ldots((X \,@\, a_1^{\mathcal{T}}) \,@\, a_2^{\mathcal{T}}) \ldots) \,@\, a_k^{\mathcal{T}}$. Since $(\Sigma, T) \vdash_{WT} e :: \tau$, $(\Sigma, T) \vdash_{WT} a_i :: \tau_i,\ 0 \leq i \leq k$ and $T(X) = \tau_1 \rightarrow \ldots\ \rightarrow \tau_k \rightarrow \tau$. By I.H. $(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} a_i^{\mathcal{T}} :: \tau_i^{\mathcal{T}},\ 0 \leq i \leq\ k$. Applying Lemma 7.5 with $b^{\mathcal{T}} = X$ the result $(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} e^{\mathcal{T}} :: \tau^{\mathcal{T}}$ is obtained.

2. $e = c\ e_1 \ldots e_m\ (c \in DC^n, m < n, n > 0)$.
Assume that the principal type of c in Σ is $c :: \mu_1 \rightarrow \ldots \rightarrow \mu_n \rightarrow \mu$. Since $(\Sigma, T) \vdash_{WT} e :: \tau$, there must be some $\theta \in TSubst$ such that

$$(\Sigma, T) \vdash_{WT} e_i :: \mu_i\theta\ (1 \leq i \leq m), \quad \tau = (\mu_{m+1} \rightarrow \ldots \rightarrow \mu_n \rightarrow \mu)\theta$$

By I.H. and Lemma 7.6 we obtain

$$(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} e_i^{\mathcal{T}} :: \mu_i^{\mathcal{T}} \theta^{\mathcal{T}}\ (1 \leq i \leq m)$$

On the other hand, $e^{\mathcal{T}} = c_m^{\mathcal{T}}\ e_1^{\mathcal{T}} \ldots e_m^{\mathcal{T}}$ and the principle type of $c_m^{\mathcal{T}}$ in $\Sigma^{\mathcal{T}}$ is

$$c_m^{\mathcal{T}} ::\ \mu_1^{\mathcal{T}} \to \ldots \to \mu_{m+1}^{\mathcal{T}} \to ((\mu_{m+2} \to \ldots \to \mu_n \to \mu)^{\mathcal{T}}, cTree)$$

Therefore, we can deduce:

$$(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} e^{\mathcal{T}} :: \mu_{m+1}^{\mathcal{T}}\theta^{\mathcal{T}} \to ((\mu_{m+2} \to \ldots \to \mu_m \to \mu)^{\mathcal{T}}\theta^{\mathcal{T}},\ cTree)$$

which is the same as $(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} e^{\mathcal{T}} :: \tau^{\mathcal{T}}$, because

$$\begin{array}{ll}
\tau^{\mathcal{T}} & = \\
((\mu_{m+1} \to \ldots \to \mu)\theta)^{\mathcal{T}} & = \ \text{Lemma 7.6} \\
(\mu_{m+1} \to \ldots \to \mu)^{\mathcal{T}}\theta^{\mathcal{T}} & = \\
(\mu_{m+1}^{\mathcal{T}} \to ((\mu_{m+2} \to \ldots \to \mu_n)^{\mathcal{T}},\ cTree))\theta^{\mathcal{T}} & = \\
\mu_{m+1}^{\mathcal{T}}\theta^{\mathcal{T}} \to ((\mu_{m+2} \to \ldots \to \mu_n)^{\mathcal{T}}\theta^{\mathcal{T}},\ cTree) &
\end{array}$$

3. $e =\ c\ e_1 \ldots e_n\ (c \in DC^n, n \geq 0)$.
Assume that the principal type of c in Σ is as in case **2**.
Since $(\Sigma, T) \vdash_{WT}\ e :: \tau$, there must be some $\theta \in TSubst$ such that

$$(\Sigma, T) \vdash_{WT} e_i :: \mu_i\theta\ (1 \leq i \leq n), \quad \tau = \mu\theta$$

By I.H. and Lemma 7.6 we obtain

$$(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT}\ e_i^{\mathcal{T}} :: \mu_i^{\mathcal{T}}\theta^{\mathcal{T}}\ (1 \leq i \leq n)$$

Since the μ_i are the principal types of a data constructor's arguments they must be datatypes, so that $\mu_i^{\mathcal{T}} = \mu_i$. Moreover $e^{\mathcal{T}} = c\ e_1^{\mathcal{T}} \ldots e_n^{\mathcal{T}}$, and the principal type declaration of c in $\Sigma^{\mathcal{T}}$ is the same as in Σ. Therefore we can deduce $(\Sigma^{\mathcal{T}},\ T^{\mathcal{T}}) \vdash_{WT}\ e :: \mu\theta^{\mathcal{T}}$. Since μ is also a datatype, Lemma 7.6 ensures that $\mu\theta^{\mathcal{T}} = \mu^{\mathcal{T}}\theta^{\mathcal{T}} = (\mu\theta)^{\mathcal{T}} = \tau^{\mathcal{T}}$ and we are ready.

4. $e =\ f\ a_1 \ldots a_k\ (f \in FS^0, k \geq 0)$.
In this case

$$e^{\mathcal{T}} = (\ldots(((@_0\ f^{\mathcal{T}})@\ a_1^{\mathcal{T}})\ @\ a_2^{\mathcal{T}})\ @\ \ldots)\ @\ a_k^{\mathcal{T}}$$

Assume that the principal types of f and $f^{\mathcal{T}}$ in their respective signatures are $f\ ::\ \mu$ and $f^{\mathcal{T}}\ ::\ (\mu^{\mathcal{T}}, cTree)$. Since $(\Sigma, T) \vdash_{WT}\ e\ ::\ \tau$, there must be some $\theta \in TSubst$ such that

$$\mu\theta = \tau_1 \to \ldots \tau_k \to \tau, \quad (\Sigma, T) \vdash_{WT}\ a_i :: \tau_i,\ (1 \leq i\ \leq k)$$

By I.H. we can obtain: $(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT}\ a_i^{\mathcal{T}}\ ::\ \tau_i^{\mathcal{T}}$. Moreover, using principal types $f^{\mathcal{T}} :: (\mu^{\mathcal{T}}, cTree)$ and $@_0 :: (\beta, cTree) \to \beta$ it is easy to deduce:

$$(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} @_0 f^{\mathcal{T}}\ ::\ \mu^{\mathcal{T}}\theta^{\mathcal{T}} = (\mu\theta)^{\mathcal{T}}$$

where the last equality holds by Lemma 7.6. Now:

$$(\mu\theta)^{\mathcal{T}} = (\tau_1 \to \ldots \to \tau_k \to \tau)^{\mathcal{T}}$$

and Lemma 7.5 can be applied (with $b = f$) to obtain

$$(\Sigma, T) \vdash_{WT} (\ldots(((@_0\ f^{\mathcal{T}}) @\ a_1^{\mathcal{T}})\ @\ a_2^{\mathcal{T}})\ @\ \ldots)\ @\ a_k^{\mathcal{T}} :: \tau^{\mathcal{T}}$$

which is the same as $(\Sigma, T) \vdash_{WT} e^{\mathcal{T}} :: \tau^{\mathcal{T}}$.

5. $e = f\ e_1 \ldots e_m$ $(f \in FS^n, n > 0, m < n-1)$.
Analogous to case **2**.

6. $e = f\ e_1 \ldots e_{n-1}\, a_1 \ldots a_k$ $(f \in FS^n, n > 0, k \geq 0)$.
In this case

$$e^{\mathcal{T}} = (\ldots((f^{\mathcal{T}}\ e_1^{\mathcal{T}} \ldots e_{n-1}^{\mathcal{T}})\ @\ a_1^{\mathcal{T}})\ @\ a_2^{\mathcal{T}})\ @\ \ldots)\ @\ a_k^{\mathcal{T}}$$

The principal types of f and $f^{\mathcal{T}}$ in their respective signatures must be of the form

$$f :: \mu_1 \to\ \ldots \to \mu_n \to \mu, \quad f^{\mathcal{T}} :: \mu_1^{\mathcal{T}} \to \ldots \to \mu_n^{\mathcal{T}} \to (\mu^{\mathcal{T}}, cTree)$$

Since $(\Sigma, T) \vdash_{WT} e :: \tau$, there must be some $\theta \in TSubst$ such that

$$(\mu_1 \to\ \ldots \to \mu_n \to \mu)\theta = \tau_1 \to\ \ldots \to \tau_{n-1} \to \nu_1 \to \ldots \to \nu_k \to \tau$$

with

$$(\Sigma, T) \vdash_{WT} e_i :: \tau_i\ (1 \leq i \leq n-1), \quad (\Sigma, T) \vdash_{WT} a_j :: \nu_j\ (1 \leq j \leq k)$$

By I.H. we obtain:

$$(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} e_i^{\mathcal{T}} :: \tau_i^{\mathcal{T}}\ (1 \leq i \leq n-1), \quad (\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} a_j^{\mathcal{T}} :: \nu_j^{\mathcal{T}}\ (1 \leq j \leq k)$$

Using the principal type of $f^{\mathcal{T}}$ in $\Sigma^{\mathcal{T}}$ as well as Lemma 7.6 we can deduce:

$$\begin{array}{rll} (\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} f^{\mathcal{T}}\ e_1^{\mathcal{T}} \ldots e_{n-1}^{\mathcal{T}} :: & \mu_n^{\mathcal{T}}\theta^{\mathcal{T}} \to (\mu^{\mathcal{T}}\theta^{\mathcal{T}}, cTree) & = \text{Lemma 7.6} \\ & \mu_n\theta^{\mathcal{T}} \to (\mu\theta^{\mathcal{T}}, cTree) & = \\ & ((\mu_n \to \mu)\theta)^{\mathcal{T}} & = \\ & (\nu_1 \to \ldots \to \nu_k \to \nu)^{\mathcal{T}} & \end{array}$$

Now we can apply Lemma 7.5 to obtain

$$(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} (\ldots((f^{\mathcal{T}}\ e_1^{\mathcal{T}} \ldots e_{n-1}^{\mathcal{T}})\ @\ a_1^{\mathcal{T}})\ @\ a_2^{\mathcal{T}})\ @\ \ldots)\ @\ a_k^{\mathcal{T}} :: \tau^{\mathcal{T}}$$

which is the same as $(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} e^{\mathcal{T}} :: \tau^{\mathcal{T}}$. □

Lemma 7.8 *Any simple application of the transformation rules AP_0 or AP_1 defined in Section 4.4 preservers well-typing. More precisely: if a partially transformed program rule is well-typed w.r.t. a type environment $T_i^{\mathcal{T}}$, then the new program rule obtained by one single application of AP_0 or AP_1 is also well-typed w.r.t. some type environment $T_{i+1}^{\mathcal{T}}$ with type assumptions for some new variables.*

Proof.

Both transformation rules AP_0, AP_1 transform only the part of the rule corresponding to the local declarations, and hence we only need to check that this part is well-typed.

• Rule AP_0 transforms
$\{\ldots;\, p \leftarrow e[@_0\ fun]; \ldots T \leftarrow \text{cNode} \ldots (clean\ lp)\}$ into
$\{\ldots;\, (R', T') \leftarrow fun;\, p \leftarrow e[R']; \ldots T \leftarrow \text{cNode} \ldots (clean\ (lp{+\!+}[(\text{dVal}\ \ R', T')]))\}$
There must exist a type environment $T_i^{\mathcal{T}}$ such that:

(1) $(\Sigma^{\mathcal{T}}, T_i^{\mathcal{T}}) \vdash_{WT} p\ ::\ \tau\ ::\ e[@_0\ fun]$, with τ' the type used for $(@_0\ fun)$ in the proof.
(2) $(\Sigma^{\mathcal{T}}, T_i^{\mathcal{T}}) \vdash_{WT} T\ ::\ cTree\ ::\ (\text{cNode} \ldots (clean\ (lp\,{+}{+}[(\text{dVal}\ \ R', T')])))$, with $[(pVal, cTree)]$ the type used for lp in the proof.

Then we define $T_{i+1}^{\mathcal{T}}$ by extending $T_i^{\mathcal{T}}$ with suitable types for the new variables R' and T':

$$T_{i+1}^{\mathcal{T}} = T_i^{\mathcal{T}}\ \oplus\ \{R' :: \tau', T' :: cTree\}$$

Since τ' is the type of $(@_0\ fun)$ in (1), then:

$$(\Sigma^{\mathcal{T}}, T_i^{\mathcal{T}}) \vdash_{WT} @_0\ ::\ (\tau', cTree) \rightarrow \tau', \quad (\Sigma^{\mathcal{T}}, T_i^{\mathcal{T}}) \vdash_{WT} fun\ ::\ (\tau', cTree)$$

Then obviously

$$(\Sigma^{\mathcal{T}}, T_{i+1}) \vdash_{WT} (R', T')\ ::\ (\tau', cTree) :: fun, \quad (\Sigma^{\mathcal{T}}, T_{i+1}) \vdash_{WT} p\ ::\ \tau :: e[R']$$

Finally, since $dVal\ ::\ A \rightarrow pVal$, $(+\!+) :: [A] \rightarrow [A] \rightarrow [A] \in \Sigma^{\mathcal{T}}$, and using the types for T, lp in (2)

$$(\Sigma^{\mathcal{T}}, T_{i+1}) \vdash_{WT} (lp\,{+}{+}[(\text{dVal}\ \ R', T')])) :: [(pVal, cTree)]$$

and hence:

$$(\Sigma^{\mathcal{T}}, T_{i+1}) \vdash_{WT} (\text{cNode} \ldots (clean\ (lp\,{+}{+}[(\text{dVal}\ \ R', T')]))) :: cTree$$

as expected.
• Rule AP_1: The proof is very similar to the case of rule AP_0.

$\square$

Now we are ready to prove Theorem 4.2:
In $P^{\mathcal{T}}$ there are new functions such as clean, dVal, functions for partial applications and constructors, as well as functions coming from the transformation of functions in P (see Section 4.4. We must prove that all these of functions are well-typed.

• Function dVal is a primitive and therefore we only can assume that it is well-typed. Checking that clean is well-typed is straightforward from its definition.

• Auxiliary functions $f_0^{\mathcal{T}}, \ldots, f_{n-2}^{\mathcal{T}}$ for $f \in FS^n$. Two cases:

(1) $f_m^{\mathcal{T}}$, $m < n-2$
The only rule for

$$f_m^{\mathcal{T}} \ :: \ \tau_1^{\mathcal{T}} \to \ldots \to \tau_{m+1}^{\mathcal{T}} \to ((\tau_{m+2} \to \ldots \to \tau_n \to \tau)^{\mathcal{T}}, \, cTree)$$

is $f_m^{\mathcal{T}} \ \overline{X}_{m+1} \ \to \ (f_{m+1}^{\mathcal{T}} \ \overline{X}_{m+1}, \, void)$. We define a new type environment:

$$T^{\mathcal{T}} = \{X_1 :: \tau_1^{\mathcal{T}}, \ldots, X_{m+1} :: \tau_{m+1}^{\mathcal{T}}\}$$

which ensures:
a) $(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} \overline{X}_{m+1} :: \overline{\tau^{\mathcal{T}}}_{m+1}$.
b) $(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} f_{m+1}^{\mathcal{T}} \ \overline{X}_{m+1} :: \tau_{m+2}^{\mathcal{T}} \to ((\tau_{m+3} \to \ldots \to \tau_n \to \tau)^{\mathcal{T}}, \, cTree) =$
$((\tau_{m+2} \to \tau_{m+3} \to \ldots \to \tau_n \to \tau)^{\mathcal{T}}, \, cTree)$.
c) $(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \ \vdash_{WT} \ void :: cTree$

(2) $f_{n-2}^{\mathcal{T}}$
The only rule for $f_{n-2}^{\mathcal{T}} \ :: \ \tau_1^{\mathcal{T}} \to \ldots \to \tau_{n-1}^{\mathcal{T}} \to ((\tau_n \to \tau)^{\mathcal{T}}, \, cTree)$ is $f_{n-2}^{\mathcal{T}} \ \overline{X}_{n-1} \to (f^{\mathcal{T}} \ \overline{X}_{n-1}, \, void)$, and by defining the same type environment T as above and since $f^{\mathcal{T}} :: \tau_1^{\mathcal{T}} \to \ldots \to \tau_n^{\mathcal{T}} \to (\tau^{\mathcal{T}}, \, cTree)$, the result can be checked as in the previous case.

• Auxiliary functions $c_0^{\mathcal{T}}, \ldots, c_{n-1}^{\mathcal{T}}$ for $c \in DC^n$. Analogous to the previous case.

• Transformed functions.
The well-typedness of transformed functions can be checked in two steps:

1) Assuming a well-type program rule in P:

$$(R) \quad f \ t_1 \ \ldots \ t_n \to r \Leftarrow \ldots l_i == r_i \ldots \quad \text{where}\{\ldots \ s_j \leftarrow d_j; \ldots\}$$

For a function with principal type declaration $f : \tau_1 \to \tau_n \to \tau$, the transformed function has principal type $f^{\mathcal{T}} : \tau_1^{\mathcal{T}} \to \ldots \to \tau_n^{\mathcal{T}} \to (\tau^{\mathcal{T}}, cTree)$ and the transformed program rule looks initially as follows, before starting to apply the transformations AP_0 and AP_1 to the local definitions:

$$
\begin{array}{lll}
(R_0^{\mathcal{T}})\ f^{\mathcal{T}}\ t_1^{\mathcal{T}}\ \ldots\ t_n^{\mathcal{T}} \rightarrow (R,T) \Leftarrow \ldots CL_i == CR_i \ldots & & \\
\quad \text{where}\{ & & \ldots \\
 & s_j^{\mathcal{T}} & \leftarrow d_j^{\mathcal{T}}; \\
 & & \ldots \\
 & CL_i & \leftarrow l_i^{\mathcal{T}}; \\
 & CR_i & \leftarrow r_i^{\mathcal{T}}; \\
 & & \ldots \\
 & R & \leftarrow r^{\mathcal{T}}; \\
 & T & \leftarrow \text{cNode}\ \text{"}f\text{"}\ [dVal\, t_1^{\mathcal{T}}, \ldots dVal\, t_n^{\mathcal{T}}]\ (dVal\ R)\ \text{"}f.j\text{"}\ (clean\,[\,])\}
\end{array}
$$

In order to prove that the previous partially transformed program rule is well-typed, we consider a type environment T which well-types the original program rule in the sense defined in Section 2.2.1. We define $T_0^{\mathcal{T}}$ in the following way:

$T_0^{\mathcal{T}}(X) = (T(X))^{\mathcal{T}}$ for all $X \in dom(T)$.
$T_0^{\mathcal{T}}(R) = \tau^{\mathcal{T}}$ with τ s.t. $(\Sigma, T) \vdash_{WT} r :: \tau$.
$T_0^{\mathcal{T}}(T) = cTree$.
$T_0^{\mathcal{T}}(CL_i) = T_0^{\mathcal{T}}(CR_i) = \nu_i^{\mathcal{T}}$ with ν_i s.t. $(\Sigma, T) \vdash_{WT} l_i :: \nu_i :: r_i$.

Now:
- Since (R) well-typed, $(\Sigma, T) \vdash_{WT} t_i :: \tau_i$. By Lemma 7.7,

$$(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} t_i^{\mathcal{T}} :: \tau_i^{\mathcal{T}}$$

- $(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} (R,T) :: (\tau^{\mathcal{T}}, cTree)$.

- $(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} CL_i :: \nu_i^{\mathcal{T}} :: CR_i$.

- Since (R) well-typed, $(\Sigma, T) \vdash_{WT} s_j :: \upsilon :: d_j$. By Lemma 7.7,

$$(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} s_j^{\mathcal{T}} :: \upsilon^{\mathcal{T}} :: d_j^{\mathcal{T}}$$

- Also, by Lemma 7.7, and the construction of $T^{\mathcal{T}}$:

$$
\begin{array}{c}
(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} CL_i :: \nu_i^{\mathcal{T}} :: l_i^{\mathcal{T}} \\
(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} CR_i :: \nu_i^{\mathcal{T}} :: r_i^{\mathcal{T}} \\
(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} R :: \tau^{\mathcal{T}} :: r^{\mathcal{T}}
\end{array}
$$

- Finally is easy to check from the signature of $cNode$, $dVal$ and $clean$ that:

$$
\begin{array}{l}
(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} \\
\qquad (\text{cNode}\ \text{"}f\text{"}\ [dVal\, t_1^{\mathcal{T}}, \ldots,\ dVal\, t_n^{\mathcal{T}}]\ (dVal\ R)\ \text{"}f.j\text{"}\ (clean\,[\,])) :: cTree
\end{array}
$$

and. by definition

$$(\Sigma^{\mathcal{T}}, T^{\mathcal{T}}) \vdash_{WT} T : cTree$$

Therefore the initial transformation is well-typed.

2) The consecutive application of rules AP_0 and AP_1 transform a well-typed rule into a new well-typed rule. This follows from Lemma 7.8.
Since each rule AP_0 and AP_1 reduces the number either the number of either $@_0$ or $@$, and the number of these symbols is finite, the process will end in a well-typed rule.

□

Proof of Theorem 4.3

The proof of this theorem depends on the specification of the semantic calculus $FSC^{\mathcal{T}}$ used for deductions with transformed programs. This is presented below:

Definition 7.9 The calculus $FSC^{\mathcal{T}}$ consists of the rules of FSC (i.e. rules BT, RR, DC, JN and FA of the SC described in Section 2.3.1) plus two new rule schemes $dVal$ and SFA (meaning *Suspended Function Application*) defined as follows:

- $(dVal)$ $\quad dval\ t^{\mathcal{T}} \ \rightarrow\ \ulcorner t \urcorner$

  where:
  - t can be any pattern in the original signature.
  - $t^{\mathcal{T}}$ is the transformation of pattern t as described in Section 4.2.
  - $\ulcorner t \urcorner$ is the representation of t as string.
- (SFA) $\quad call^{\mathcal{T}} \ \rightarrow\ (\bot, \bot)$

  Here $call^{\mathcal{T}}$ can be any partial expression which has one of the forms $g^{\mathcal{T}}\ \overline{s^{\mathcal{T}}}_m$ (with $g \in FS^m$, $m \geq 0$) or $(F\ s^{\mathcal{T}})\rho^{\mathcal{T}}$ (F variable, s pattern, $\rho \in Subst_{\bot}$).

Before starting the proof we observe that the SC deduction $P\ \vdash_{SC}\ f\ \overline{t_n} \rightarrow t$ can be replaced by $P_F\ \vdash_{FSC}\ f\ \overline{t_n} \rightarrow t$ due to the semantic correctness of flattening (Theorem 4.1). Now we are ready to prove items (i) and (ii) of the theorem.

(i) Assume a FSC proof tree T witnessing $P_F\ \vdash_{FSC}\ f\ \bar{t}_n \rightarrow t$ with associated APT apt. Reasoning by induction on the structure of T we show that it is possible to find a total $ct :: cTree$ representing apt such that

$$P^{\mathcal{T}}\ \vdash_{FSC^{\mathcal{T}}}\ f^{\mathcal{T}}\ \overline{t^{\mathcal{T}}}_n \rightarrow (t^{\mathcal{T}}, ct)$$

Since $t \neq \bot$, the inference step at the root of T must be a FA step using one of the defining rules for f in P_F instantiated by some $\rho \in Subst_{\bot}$. Let us consider this program rule $rl_F \in P_F$ along with the corresponding program rules $rl \in P$ and $rl^{\mathcal{T}} \in P^{\mathcal{T}}$ respectively. In the sequel we assume that $rl^{\mathcal{T}}$ and rl_F have the forms shown at the end of Section 4.4 and in Section 4.7, respectively, except that the left-hand sides are now assumed to be $f^{\mathcal{T}}\ p_1^{\mathcal{T}}\ \ldots\ p_n^{\mathcal{T}}$ and $f\ p_1\ \ldots\ p_n$, respectively. In the reasonings below, we make implicit use of Proposition 2.1 and Lemma 7.6 at several places.

The substitution ρ must be such that the inference step at the root of T as well as the rest of T succeed. Therefore we can assume:

(1) $\overline{p}_n\rho = \overline{t}_n$.

(2) $P_F\ \vdash_{FSC}\ R\rho \to t$ i.e. $t \sqsubseteq R\rho$ (since t, $R\rho$ patterns).

(3) $P_F\ \vdash_{FSC}\ (call_k \to R_k)\rho$ for each condition $R_k \leftarrow call_k$ in rl_F (proved by subtrees T_k of T with smaller size than T).

(4)$P_F\ \vdash_{FSC}\ (s_j \leftarrow w_j)\rho$ i.e. $s_j\rho \sqsubseteq w_j\rho$ for each condition $s_j \leftarrow w_j$ in rl_F (since $s_j\rho, w_j\rho$ are patterns).

(5) $P_F\ \vdash_{FSC}\ (LS_i \leftarrow u_i)\rho$ and $P_F\ \vdash_{FSC}\ (RS_i \leftarrow v_i)\rho$ i.e. $LS_i\rho \sqsubseteq u_i\rho$ and $RS_i\rho \sqsubseteq v_i\rho$ (since $LS_i\rho$, $u_i\rho$, $RS_i\rho$, $v_i\rho$ are patterns) for all conditions $LS_i \leftarrow u_i$, $RS_i \leftarrow v_i$ in rl_F.

(6) $P_F\ \vdash_{FSC}\ (R \leftarrow v)\rho$ i.e. $R\rho \sqsubseteq v\rho$, since $R\rho, v\rho$ are patterns.

Now we look for a corresponding $\rho^{\mathcal{T}} \in Subst_\perp^{\mathcal{T}}$ defined in such a way that we can build a $FSC^{\mathcal{T}}$ proof tree T' witnessing $P^{\mathcal{T}}\ \vdash_{FSC^{\mathcal{T}}}\ f^{\mathcal{T}}\ \overline{t_n^{\mathcal{T}}} \to (t^{\mathcal{T}}, ct)$, so that the inference step at the root of T' will be a FA inference using the $\rho^{\mathcal{T}}$-instance of the program rule $rl^{\mathcal{T}}$. As a partial definition of $\rho^{\mathcal{T}}$ we assume: $\rho^{\mathcal{T}}(X) = \rho(X)^{\mathcal{T}}$ for all $X \in dom(\rho)$. The effect of $\rho^{\mathcal{T}}$ over those variables of $rl^{\mathcal{T}}$ which do not appear in rl_F (namely T and the various T_k) will be defined later. Presently, the partial definition of $\rho^{\mathcal{T}}$ allows us to draw some conclusions from items (1) – (6) above:

(1′) $\overline{p}_n^{\mathcal{T}}\rho^{\mathcal{T}} = (\overline{p}_n\rho)^{\mathcal{T}} = \overline{t^{\mathcal{T}}}_n$.

(2′) $t^{\mathcal{T}} \sqsubseteq R\rho^{\mathcal{T}}$ i.e. $P^{\mathcal{T}}\ \vdash_{FSC^{\mathcal{T}}}\ R\rho^{\mathcal{T}} \to t^{\mathcal{T}}$ (since $t^{\mathcal{T}}$, $R\rho^{\mathcal{T}}$ patterns).

(3′) Each condition $(R_k, T_k) \leftarrow call_k^{\mathcal{T}}$ in $rl^{\mathcal{T}}$ corresponds to $R_k \leftarrow call_k$ in rl_F. Here we can distinguish three cases:

(3.1′) $R_k\rho \neq \perp$ and $call_k = g\,\overline{s}_m$ for some $g \in FS^m$, $m \geq 0$ and some patterns $\overline{s}_m$. Then $call_k^{\mathcal{T}}\rho^{\mathcal{T}} = g^{\mathcal{T}}\ \overline{s}_m^{\mathcal{T}}\rho^{\mathcal{T}} = g^{\mathcal{T}}\ (\overline{s}_m\rho)^{\mathcal{T}}$ and by I.H. applied to $P_F\ \vdash_{FSC}\ g\,\overline{s}_m\rho \to R_k\rho$ we can assume $P^{\mathcal{T}}\ \vdash_{FSC^{\mathcal{T}}}\ call_k^{\mathcal{T}}\rho^{\mathcal{T}} \to (R_k\rho^{\mathcal{T}}, ct_k)$ where $ct_k :: cTree$ represents apt_k, the APT extracted from T_k.

((3.2′) $R_k\rho \neq\perp$ and $call_k = F\ s$ with F variable. Since $F\rho$ and $s\rho$ are patterns, $P_F\ \vdash_{FSC}\ (F\ s \to R_k)\rho$, and $R_k\rho \neq\perp$, it follows that $F\rho$ must be a rigid pattern. We consider different subcases according to the form of $F\rho$:

(3.2.1′) $F\rho = c\ \overline{s}_m$, $m \geq 0$, $ar(c) = m+1$, $c \in DC$.
Since $P_F\ \vdash_{FSC}\ F\rho\ s\rho \to R_k\rho$, $s\rho$ must be a pattern s.t. $R_k\rho \sqsubseteq c\ \overline{s}_m\ s\rho$. Moreover:
$call_k^{\mathcal{T}}\rho^{\mathcal{T}} = (F\ s^{\mathcal{T}})\rho^{\mathcal{T}} = ((c\ \overline{s}_m)^{\mathcal{T}})\ (s^{\mathcal{T}}\rho^{\mathcal{T}}) = (\overline{c^{\mathcal{T}}}_m\ \overline{s^{\mathcal{T}}}_m)\ (s^{\mathcal{T}}\rho^{\mathcal{T}})$. The last step holds because of the definition of the transformation $^{\mathcal{T}}$. $P^{\mathcal{T}}$ includes the program rule $\overline{c^{\mathcal{T}}}_m\ \overline{X}_{m+1} \to\ (c\ \overline{X}_{m+1}, void)$. Using this rule and $R_k\rho^{\mathcal{T}} \sqsubseteq c\ \overline{s^{\mathcal{T}}}_m\ (s^{\mathcal{T}}\rho^{\mathcal{T}})$, we can derive:

$$P^{\mathcal{T}}\ \vdash_{FSC^{\mathcal{T}}}\ c_m^{\mathcal{T}}\ \overline{s^{\mathcal{T}}}_m\ (s^{\mathcal{T}}\rho^{\mathcal{T}}) \to (R_k\rho^{\mathcal{T}}, void)$$

i.e.

$$P^{\mathcal{T}}\ \vdash_{FSC^{\mathcal{T}}}\ call_k^{\mathcal{T}}\ \rho^{\mathcal{T}}\ \to (R_k\rho^{\mathcal{T}}, void)$$

(3.2.2′) $F\rho = c\ \overline{s}_m$, $m \geq 0\ ar(c) > m+1$, $c \in DC$. Analogous to the previous case but using the $P^{\mathcal{T}}$ rule $c_m^{\mathcal{T}}\ \overline{X}_{m+1} \to (c_{m+1}^{\mathcal{T}}, void)$.

(3.2.3′) $F\rho = g\ \overline{s}_m$, $m \geq 0\ ar(g) = m+1$, $g \in FS$. In this case:

$$call_k^{\mathcal{T}}\ \rho^{\mathcal{T}}\ =\ (F\ s^{\mathcal{T}})\rho^{\mathcal{T}}\ =\ (g\ \overline{s}_m)^{\mathcal{T}}(s^{\mathcal{T}}\rho^{\mathcal{T}})\ =\ g^{\mathcal{T}}\ \overline{s^{\mathcal{T}}}_m\ (s^{\mathcal{T}}\rho^{\mathcal{T}})$$

By I.H. applied to $P_F\ \vdash_{FSC}\ g\ \overline{s}_m\ (s\rho) \to R_k\rho$ we arrive to the same conclusion as in case (3.1′), namely: $P^{\mathcal{T}}\ \vdash_{FSC^{\mathcal{T}}}\ call_k^{\mathcal{T}}\ \to (R_k\rho^{\mathcal{T}}, ct_k)$ where $ct_k :: cTree$ represents the APT extracted from T_k.

(3.2.4′) $F\rho = g\ \overline{s}_m$, $m \geq 0$, $ar(g) > m+1$, $g \in FS$. Similarly to (3.2.1′), since $P_F\ \vdash_{FSC}\ (F\rho)\ (s\rho)\ \to R_k\rho$, $s\rho$ must be s.t. $R_k\rho \sqsubseteq g\ \overline{s}_m\ (s\rho)$. Moreover $call_k^{\mathcal{T}}\ \rho^{\mathcal{T}} = (F\ s^{\mathcal{T}})\rho^{\mathcal{T}} = (g\ \overline{s}_m)^{\mathcal{T}}\ (s^{\mathcal{T}}\rho^{\mathcal{T}}) = g_m^{\mathcal{T}}\ \overline{s^{\mathcal{T}}}_m\ (s^{\mathcal{T}}\rho^{\mathcal{T}})$, and $P^{\mathcal{T}}$ includes one of the two following defining rules:
(R1) $g_m^{\mathcal{T}}\ \overline{X}_{m+1} \to (g^{\mathcal{T}}\ \overline{X}_{m+1}, void)$ if $m+2 < ar(g)$.
(R2) $g_m^{\mathcal{T}}\ \overline{X}_{m+1} \to (g^{\mathcal{T}}\ \overline{X}_{m+1}, void)$ if $m+2 = ar(g)$.
In the case $m+2 < arg(g)$ we have:
- $P^{\mathcal{T}}\ \vdash_{FSC^{\mathcal{T}}}\ call_k^{\mathcal{T}}\ \rho^{\mathcal{T}} \to (g_{m+1}^{\mathcal{T}}\ \overline{s_m^{\mathcal{T}}}\ (s^{\mathcal{T}}\rho^{\mathcal{T}}), void)$ using (R1).
- $R_k\rho^{\mathcal{T}} \sqsubseteq ((g\ \overline{s}_m)(s\rho))^{\mathcal{T}} = g_{m+1}^{\mathcal{T}}\ \overline{s^{\mathcal{T}}}_m\ (s^{\mathcal{T}}\rho^{\mathcal{T}})$.

Similarly, in the case $m+2 = arg(g)$:
- $P^{\mathcal{T}}\ \vdash_{FSC^{\mathcal{T}}}\ call_k^{\mathcal{T}}\ \rho^{\mathcal{T}} \to (g^{\mathcal{T}}\ \overline{s_m^{\mathcal{T}}}\ (s^{\mathcal{T}}\rho^{\mathcal{T}}), void)$ using (R2).
- $R_k\rho^{\mathcal{T}} \sqsubseteq ((g\ \overline{s}_m)(s\rho))^{\mathcal{T}} = g^{\mathcal{T}}\ \overline{s^{\mathcal{T}}}_m\ (s^{\mathcal{T}}\rho^{\mathcal{T}})$.

In both cases, we can conclude that $P^{\mathcal{T}}\ \vdash_{FSC^{\mathcal{T}}}\ call_k^{\mathcal{T}}\rho^{\mathcal{T}}\ \to (R_k\rho^{\mathcal{T}}, void)$.

(3.3′) $R_k\rho = \perp$. In this case T_k must reduce to one single step, applying the SC inference BT, and we can establish no definite conclusion about $call_k\rho$, except that it must have one of the following forms:
($*$) $g\ \overline{s}_m$, with $g \in FS_m$, $m \geq 0$; $\overline{s}_m$ patterns.
($**$) $(F\ s)\rho$, which might be even flexible if $F\rho = F$.
In both cases, we can use the special $FSC^{\mathcal{T}}$-inference SFA to derive: $P^{\mathcal{T}}\ \vdash_{FSC^{\mathcal{T}}}\ call_k^{\mathcal{T}} \to (\perp, \perp)$.

(4′) $(s_j\rho)^{\mathcal{T}} \sqsubseteq (w_j\rho)^{\mathcal{T}}$ i.e. $s_j^{\mathcal{T}}\rho^{\mathcal{T}} \sqsubseteq w_j^{\mathcal{T}}\rho^{\mathcal{T}}$ i.e. $P^{\mathcal{T}} \vdash_{FSC^{\mathcal{T}}} (s_j^{\mathcal{T}} \leftarrow w_j^{\mathcal{T}})\rho^{\mathcal{T}}$ for each condition $s_j^{\mathcal{T}} \leftarrow w_j^{\mathcal{T}}$ in $rl^{\mathcal{T}}$ (since $s_j^{\mathcal{T}}\rho^{\mathcal{T}}, w_j^{\mathcal{T}}\rho^{\mathcal{T}}$ are patterns).

(5′) $(LS_i\rho)^{\mathcal{T}} \sqsubseteq (u_i\rho)^{\mathcal{T}}$ i.e. $LS_i\rho^{\mathcal{T}} \sqsubseteq u_i^{\mathcal{T}}\rho^{\mathcal{T}}$ i.e. $P^{\mathcal{T}} \vdash_{FSC^{\mathcal{T}}} (LS_i \leftarrow u_i^{\mathcal{T}})\rho^{\mathcal{T}}$ and $(RS_i\rho)^{\mathcal{T}} \sqsubseteq (v_i\rho)^{\mathcal{T}}$ i.e. $RS_i\rho^{\mathcal{T}} \sqsubseteq v_i^{\mathcal{T}}\rho^{\mathcal{T}}$ i.e. $P^{\mathcal{T}} \vdash_{FSC^{\mathcal{T}}} (RS_i \leftarrow v_i^{\mathcal{T}})\rho^{\mathcal{T}}$,
for each condition $LS_i \leftarrow u_i^{\mathcal{T}}$, $RS_i \leftarrow v_i^{\mathcal{T}}$ in $rl^{\mathcal{T}}$, ($LS_i\rho^{\mathcal{T}}$, $u_i^{\mathcal{T}}\rho^{\mathcal{T}}$, $RS_i^{\mathcal{T}}\rho^{\mathcal{T}}$, $v_i^{\mathcal{T}}\rho^{\mathcal{T}}$ are patterns).

(6′) $(R\rho)^{\mathcal{T}} \sqsubseteq (v\rho)^{\mathcal{T}}$ i.e. $R\rho^{\mathcal{T}} \sqsubseteq v^{\mathcal{T}}\rho^{\mathcal{T}}$ i.e. $P^{\mathcal{T}} \vdash_{FSC^{\mathcal{T}}} (R \leftarrow v^{\mathcal{T}})\rho$, ($R\rho^{\mathcal{T}}, v^{\mathcal{T}}\rho^{\mathcal{T}}$ are patterns).

At this point we can complete the definition of $\rho^{\mathcal{T}}$ by requiring:
- $\rho^{\mathcal{T}}(T_k) = ct_k$, for all those k corresponding to case (3.1′) or case (3.2.3′).
- $\rho(T_k) = void$, for all those k corresponding to some of the cases (3.2.1′), (3.2.2′), (3.2.4′).
- $\rho^{\mathcal{T}}(T_k) = \perp$, for all those k corresponding to case (3.3′).
- $\rho(T) =$ cNode "f" $[\ulcorner t_1^{\mathcal{T}}\urcorner, \dots, \ulcorner t_n^{\mathcal{T}}\urcorner]$ $\ulcorner R\rho^{\mathcal{T}}\urcorner$ "f.ind" $[\dots ct_k \dots]$

where:
- $\ulcorner t_i^{\mathcal{T}}\urcorner$ $(1 \le i \le n)$, $\ulcorner R\rho^{\mathcal{T}}\urcorner$ are the representations of the patterns $t_i^{\mathcal{T}}$ $(1 \le i \le n)$, $R\rho^{\mathcal{T}}$ as strings.
- "f" resp. "f.ind" are the strings which represent the symbol f and the symbol f followed by the index member of the program rule rl (among the program rules for f, taken in textual order).
- $[\dots ct_k \dots]$ is the list of all those ct_k corresponding to cases (3.1′), (3.2.3′).

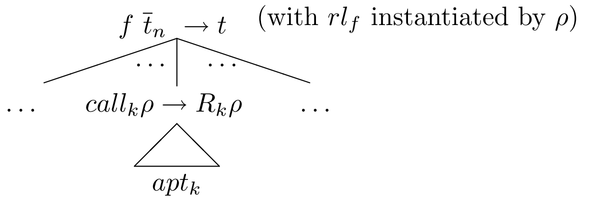
Let $ct = \rho^{\mathcal{T}}(T)$. We claim that $P^{\mathcal{T}} \vdash_{FSC^{\mathcal{T}}} f^{\mathcal{T}}\ \overline{t^{\mathcal{T}}}_n \rightarrow (t^{\mathcal{T}}, ct)$ and that $ct :: cTree$ represents *apt*, the APT extracted from the FSC proof tree T which proved $P_F \vdash_{FSC} f\ \bar{t}_n \rightarrow t$. To justify the claim we build a $FSC^{\mathcal{T}}$ proof tree T' whose last step is a FA inference using the $\rho^{\mathcal{T}}$-instance of the program rule $rl^{\mathcal{T}} \in P^{\mathcal{T}}$. Items $(1') - (6')$ show that the instantiated rule can be applied, and that all the conditions occurring as premises of FA, except the last one, can be proved by means of $FSC^{\mathcal{T}}$ derivations. The last condition is:

$$(T \leftarrow \text{cNode}\ "f"[dVal\ p_1^{\mathcal{T}}, \dots, dVal\ p_n^{\mathcal{T}}]\ (dVal\ R)$$
$$"f.ind"\ (clean[\dots(dVal\ R_k, T_k)\dots]))\rho^{\mathcal{T}}$$

This can be also derived from $P^{\mathcal{T}}$ in $FSC^{\mathcal{T}}$, because:
- $T\rho^{\mathcal{T}}$ is ct, as defined above.
- For all $i \le i \le n$, $dVal\ (p_i^{\mathcal{T}}\rho^{\mathcal{T}}) = dVal\ t_i^{\mathcal{T}}$, and $P^{\mathcal{T}} \vdash_{FSC^{\mathcal{T}}} dVal\ t_i^{\mathcal{T}} \rightarrow \ulcorner t_i^{\mathcal{T}}\urcorner$ by the special $FSC^{\mathcal{T}}$-inference ($dVal$).
- $P^{\mathcal{T}} \vdash_{FSC^{\mathcal{T}}} dVal\ R\rho^{\mathcal{T}} \rightarrow \ulcorner R\rho^{\mathcal{T}}\urcorner$ also because of ($dVal$).
- $P^{\mathcal{T}} \vdash_{FSC^{\mathcal{T}}} clean\ [\dots,(dVal\ R_k, T_k)\rho^{\mathcal{T}},\dots] \rightarrow [\dots, ct_k, \dots]$,

where $[\dots, ct_k, \dots]$ stands for the list of those ct_k corresponding to cases (3.1′), (3.2.3′). This follows from the definition of *clean*, because those k which correspond to other cases are such that either $R_k\rho^{\mathcal{T}} = \perp$ (and then $P^{\mathcal{T}} \vdash_{FSC^{\mathcal{T}}} dVal\ R_k\rho^{\mathcal{T}} \rightarrow \ulcorner \perp \urcorner$, $P^{\mathcal{T}} \vdash_{FSC^{\mathcal{T}}} isBottom\ \ulcorner \perp \urcorner \rightarrow true$) or

$\rho^{\mathcal{T}}(T_k) = void$.
Finally, observe that ct indeed represents apt, because apt is the APT extracted from T, and therefore its structure is:



Note that the children are the APT's apt_k corresponding to the uppermost FA steps in T different from the root step and corresponding to function calls which return a value different from $\perp$. These FA inferences correspond to the conditions $(R_K \leftarrow call_k)\rho$ in $rl_F\rho$, but excluding some cases:
- Those k such that $R_k\rho = \perp$, i.e. case $(3.3')$.
- Those k such that $R_k\rho \neq \perp$ but $(R_k \leftarrow call_k)\rho$ has been proved (within T) without applying inference FA, i.e. cases $(3.2.1')$, $(3.2.2')$ and $(3.2.4')$.

The remaining cases are just $(3.1')$ and $(3.2.3')$, for which we know that apt_k is represented by $ct_k :: cTree$. Therefore, ct really represents apt.

(ii)

Assume that T' is a $FSC^{\mathcal{T}}$ proof tree witnessing $P^{\mathcal{T}} \vdash_{FSC^{\mathcal{T}}} f^{\mathcal{T}}\ \overline{t_n^{\mathcal{T}}} \rightarrow (t^{\mathcal{T}}, ct)$. Reasoning by induction on the structure of T' we can build a FSC proof tree T witnessing $P_F \vdash_{FSC} f\ \overline{t_n} \rightarrow t$. A detailed proof would be similar to that of item (i). The intuitive idea is as follows:
- All the steps in T' having to do with the computation of values of type $cTtree$ can be ignored. In particular we can ignore all the applications of the $FSC^{\mathcal{T}}$ rule $(dVal)$, as well as all the applications of the $FSC^{\mathcal{T}}$ rule FA corresponding to the application of the auxiliary functions *clean*, *irrelevant*, *isVoid* and *isBottom*.
All the FA steps in T' dealing with the application of $g^{\mathcal{T}}$ for some $g \in FS$ (maybe f itself) can be converted into corresponding FA steps in T, using a corresponding instance of the program rule. This idea works because of the clear one-to-one correspondence between the program rules of $P^{\mathcal{T}}$ and P_F.
More formally, the inductive reasoning works because each FA step in T' uses a program rule instance whose conditions $(R_k, T_k) \leftarrow call_k$ have instances of the form $g^{\mathcal{T}}\ \overline{s^{\mathcal{T}}}_m \rightarrow (s^{\mathcal{T}}, ct)$ (for some $g \in FS^m$), such that $P^{\mathcal{T}} \vdash_{FSC^{\mathcal{T}}} g^{\mathcal{T}}\ \overline{s^{\mathcal{T}}}_m \rightarrow (s^{\mathcal{T}}, ct)$, with $FSC^{\mathcal{T}}$ proof trees of smaller size than T' included as parts of T'. $\square$

Proof of Theorem 4.5
In order to prove the theorem we assume:
(1) $G \Vdash_{GS,P} \theta$.

And we reason as follows:

(2) $sol\ \overline{X}_n\theta == true \ \Vdash^{1st}_{GS,Psol} id$, by (1) and stability of GS.

(3) $P_{sol} \vdash_{SC} \ sol\ \overline{X}_n\theta \to true$ with APT apt which can be extracted from the computation (2). By (2) and soundness of GS.

(4) $P^{\mathcal{T}}_{sol} \ \vdash_{FSC^{\mathcal{T}}} \ sol^{\mathcal{T}}\ \overline{X}_n\theta^{\mathcal{T}} \to (true, ct)$ where $ct :: cTree$ represents apt. This holds by (3) and the semantic correctness of the program transformation (Theorem 4.3, item (i)).

(5) $sol^{\mathcal{T}}\ \overline{X}_n\theta^{\mathcal{T}} == (true, T) \ \Vdash^{1st}_{GS,P^{\mathcal{T}}_{sol}} \ \{T \mapsto ct\}$. By (2), (3), (4) and weak completeness of GS.

Note that ct represents apt, an APT witnessing (3). Due to the definition of sol in P_{sol}, apt serves also as witness of $P \ \vdash_{SC} \ G\theta$. $\square$

7.3 Proofs of Results from Section 5

Proof of Theorem 5.1

First we present two auxiliary lemmata.

Lemma 7.10 *Let $S_i \Box W_i$ be a configuration obtained after i steps of the algorithm described in Section 5, and $(s \to t) \in S_i$.*
Then $var(s) \cap W_i = \emptyset \ \lor \ var(t) \cap W_i = \emptyset$.

Proof

We reason by induction on i.
Basis. The result holds for the first configuration because of the construction of S_0 and W_0, since the two initial basic facts have no common variables.

Inductive step.
Consider the i-th step of the algorithm ($i \geq 1$): $S_{i-1} \Box W_{i-1} \ \vdash_{\theta_i} \ S_i \Box W_i$. Let $s \to t$ be any approximation statement in S_i. We can distinguish two cases:

- $s \to t = (a \to b)\theta_i$ for some $a \to b$ in S_{i-1}. By I.H. either a or b (or both) share no variables with W_{i-1}. Assume that $var(a) \cap W_{i-1} = \emptyset$ (analogous for b). Then, since $dom(\theta_i) \subseteq W_{i-1}$, $s = a\theta_i = a$ and since W_i coincides with W_{i-1} except for the possible addition of some new fresh variables, $var(s) \cap W_i = \emptyset$.

- $s \to t$ is some of the new approximation statements introduced either by rule R3 or R6. In the case of $R3$, each $a_i \to b_i$ must fulfill the result, applying I.H. to $h\ \overline{a}_m \to h\ \overline{b}_m$. The case of R6 it is clear from I.H. that $var(h\ \overline{a}_m) \cap W_{i-1} = \emptyset$ and therefore also $var(a_i) \cap W_i = \emptyset$ for all $1 \geq i \geq m$. $\square$

Lemma 7.11 *Let $S_i \Box W_i \vdash_{\theta_{i+1}} S_{i+1} \Box W_{i+1}$ be some step of the algorithm described in Sect. 5. Then $Sol(S_i \Box W_i) = (\theta_{i+1} Sol(S_{i+1} \Box W_{i+1}))\upharpoonright_{W_i}$.*

Proof

The Lemma can be proved by examining the transformation rule applied at the given step. In the case of rules R1, R2 and R3, $\theta_{i+1} = id$, $W_{i+1} = W_i$ and the result follows from the definition of the approximation ordering $\sqsubseteq$, as given in Section 2.1. Rules R4, R5 and R6 are considered below.

R4 $Sol(s \to X, S\Box W) = \{X \mapsto s\}Sol(S\{X \mapsto s\}\Box W)$
a) $Sol(s \to X, S\Box W) \subseteq \{X \mapsto s\}Sol(S\{X \mapsto s\}\Box W)$
Let $\theta \in Sol(s \to X, S\Box W)$. Then:
- $X\theta = s\theta$, since $X\theta \sqsubseteq s\theta$ and θ is a total substitution.
- $S\theta$ holds
- $\theta = \{X \mapsto s\}\theta$, because for any $Y \in Var$, if $Y \neq X$ then $Y\{X \mapsto s\}\theta = Y\theta$ and if $Y = X$ then $Y\{X \mapsto s\}\theta = s\theta = X\theta = Y\theta$.
Therefore $S\{X \mapsto s\}\theta = S\theta$ holds, and hence $\theta \in Sol(S\{X \mapsto s\}\Box W)$.
Finally, considering again that $\theta = \{X \mapsto s\}\theta$, the expected result $\theta \in \{X \mapsto s\}Sol(S\{X \mapsto s\}\Box W)$ is obtained.

b) $\{X \mapsto s\}Sol(S\{X \mapsto s\}\Box W) \subseteq Sol(s \to X, S\Box W)$
Any element in $\{X \mapsto s\}Sol(S\{X \mapsto s\}\Box W)$ must be of the form $\{X \mapsto s\}\theta$, with $\theta \in Sol(S\{X \mapsto s\}\Box W)$. Then:
- $S\{X \mapsto s\}\theta$ holds.
- $s\{X \mapsto s\}\theta = X\{X \mapsto s\}\theta$. To check this, note that $X \notin var(s)$, because $X \in W$ implies $var(s) \cap W = \emptyset$, by Lemma 7.10. Then $s\{X \mapsto s\}\theta = s\theta = X\{X \mapsto s\}\theta$.

Hence $\{X \mapsto s\}\theta \in Sol(s \to X, S\Box W)$.

R5 $Sol(X \to Y, S\Box W) = \{X \mapsto Y\}Sol(S\{X \mapsto Y\}\Box W)$
Analogous to the previous case.

R6 $Sol(X \to h\,\overline{a}_m, S\Box W) =$
$(\{X \mapsto h\,\overline{X}_m\}Sol(\ldots, X_k \to a_k, \ldots, S\{X \mapsto h\,\overline{X}_m\}\Box W, \overline{X}_m)\restriction_W$
a) $Sol(X \to h\,\overline{a}_m, S\Box W) \subseteq$
$(\{X \mapsto h\,\overline{X}_m\}Sol(\ldots, X_k \to a_k, \ldots, S\{X \mapsto h\,\overline{X}_m\}\Box W, \overline{X}_m)\restriction_W$
Let $\theta \in Sol(X \to h\,\overline{a}_m, S\Box W)$. Then:
- $X\theta = h\;\overline{t}_m$ with $a_k\theta \sqsubseteq t_k$.
- $S\theta$ holds.
Consider the total substitution $\rho =_{def} \theta \uplus \{X_1 \mapsto t_1, \ldots, X_m \mapsto t_m\}$. Then :
- $X_k\rho = t_k$, $a_k\rho = a_k\theta$ and therefore $a_k\rho \sqsubseteq X_k\rho$.
- $S\{X \mapsto h\,\overline{X}_m\}\rho = S\theta$ holds.
Hence $\rho \in Sol(\ldots, X_k \to a_k, \ldots, S\{X \mapsto h\,\overline{X}_m\}\Box W, \overline{X}_m)$.
Moreover $\{X \mapsto h\,\overline{X}_m\}\rho\restriction_W = \theta$.
Therefore $\theta \in (\{X \mapsto h\,\overline{X}_m\}Sol(\ldots, X_k \to a_k, \ldots, S\{X \mapsto h\,\overline{X}_m\}\Box W, \overline{X}_m)\restriction_W$.

b) $(\{X \mapsto h\,\overline{X}_m\}Sol(\ldots, X_k \to a_k, \ldots, S\{X \mapsto h\,\overline{X}_m\}\Box W, \overline{X}_m)\restriction_W \subseteq$
$Sol(X \to h\,\overline{a}_m, S\Box W)$
Any member of $(\{X \mapsto h\,\overline{X}_m\}Sol(\ldots, X_k \to a_k, \ldots, S\{X \mapsto h\,\overline{X}_m\}\Box W, \overline{X}_m)\restriction_W$ must be of the form $\theta = (\{X \mapsto h\,\overline{X}_m\}\rho)\restriction_W$ with
$\rho \in Sol(\ldots, X_k \to a_k, \ldots, S\{X \mapsto h\,\overline{X}_m\}\Box W, \overline{X}_m)$ s.t.:

(1) For each k, $a_k\rho \sqsubseteq X_k\rho$ holds.
(2) $S\{X \mapsto h\,\overline{X}_m\}\rho =$ holds.
By Lemma 7.10, and since $X \in W$, $var(a_k) \cap W = \emptyset$.
Also, $var(a_k) \cap (W, \overline{X}_m) = \emptyset$, since $\overline{X}_m$ are fresh variables.
Therefore $a_k\rho \sqsubseteq X_k\rho$ iff $a_k \sqsubseteq X_k\rho$. Calling b_k to each $X_k\rho$:
(3) $a_k\rho \sqsubseteq X_k\rho$ iff $a_k \sqsubseteq b_k$.
This means that θ must be of the form $\theta = \{X \mapsto h\,\bar{b}_m\} \uplus \ \rho\restriction_{(W-\{X\})}$.
Then:
- $(X \to h\,\overline{a}_m)\theta = h\,\bar{b}_m \to h\,\overline{a}_m$ which holds iff for each k $\ a_k \sqsubseteq b_k$ which is true by (3) and (1).
- $S\theta = S(\{X \mapsto h\,\bar{b}_m\} \uplus \ \rho\restriction_{(W-\{X\})}) = S\{X \mapsto h\,\overline{X}_m\}\rho$ which holds by (2). $\square$

Now, Theorem 5.1 can be proved as follows:

To prove that the algorithm is terminating we define a well founded lexicographic order between configurations.
We say that $K_i < K_j$ (with $K_i = S_i \square W_i$, $K_j = S_j \square W_j$) iff
a) $\parallel K_i \parallel_1 \ < \ \parallel K_j \parallel_1$, or
b) $\parallel K_i \parallel_1 \ = \ \parallel K_j \parallel_1$ and $\parallel K_i \parallel_2 \ < \ \parallel K_j \parallel_2$, or
c) $\parallel K_i \parallel_1 \ = \ \parallel K_j \parallel_1$ and $\parallel K_i \parallel_2 \ = \ \parallel K_j \parallel_2$, and $\parallel K_i \parallel_3 \ < \ \parallel K_j \parallel_3$.
where:

- $\parallel S \square W \parallel_1$ = number of occurrences of rigid patterns $h\,\overline{a}_m$, $m \geq 0$ in some $(s \to t) \in S$ s.t.:
  a) If $h\,\overline{a}_m$ is part of s then $var(t) \cap W \neq \emptyset$.
  b) If $h\,\overline{a}_m$ is part of t then $var(s) \cap W \neq \emptyset$.

- $\parallel S \square W \parallel_2$ = number of occurrences of symbols $h \in DC \cup FS$ in S.

- $\parallel S \square W \parallel_3$ = size of S (as a multiset, counting repetitions).

Now it suffices to check that at each step of the algorithm $K_{i+1} \ < \ K_i$. This can be done by examining the transformation rule applied at this step, as well as the selected $(s \to t) \in S_i$:

R1 Then $\parallel K_{i+1} \parallel_1 \ = \ \parallel K_i \parallel_1$, $\parallel K_{i+1} \parallel_2 \ = \ \parallel K_i \parallel_2$, and $\parallel K_{i+1} \parallel_3 \ < \ \parallel K_i \parallel_3$.

R2 Then $\parallel K_{i+1} \parallel_1 \ = \ \parallel K_i \parallel_1$, and either
- $\parallel K_{i+1} \parallel_2 \ = \ \parallel K_i \parallel_2$, $\parallel K_{i+1} \parallel_3 \ < \ \parallel K_i \parallel_3$ (if $t \in Var$)
or
- $\parallel K_{i+1} \parallel_2 \ < \ \parallel K_i \parallel_2$ (if t is not a variable).

R3 Either $\parallel K_{i+1} \parallel_1 \ < \ \parallel K_i \parallel_1$, or $\parallel K_{i+1} \parallel_1 \ = \ \parallel K_i \parallel_1$ and $\parallel K_{i+1} \parallel_2 \ < \ \parallel K_i \parallel_2$

(since symbol h is removed in S_{i+1}).

R4 The algorithm step is, in this case, of the form:

$$\underbrace{s \to X,\ S\Box W}_{K_i} \vdash_{\{X \mapsto s\}} \underbrace{S\{X \mapsto s\}\Box W}_{K_{i+1}},\ X \neq s, X \in W$$

If s is a variable then $\| K_{i+1} \|_1 = \| K_i \|_1$, $\| K_{i+1} \|_2 = \| K_i \|_2$, and $\| K_{i+1} \|_3 < \| K_i \|_3$. If s is not a variable then it is a rigid pattern. Then, since $X \in W$, Lemma 7.10 ensures that

$$\| S\{X \mapsto s\}\Box W \|_1 = \| S\Box W \|_1 < \| s \to X,\ S\Box W \|_1$$

i.e. $\| K_{i+1} \|_1 < \| K_i \|_1$.

R5 Then $\| K_{i+1} \|_1 = \| K_i \|_1$, $\| K_{i+1} \|_2 = \| K_i \|_2$, and $\| K_{i+1} \|_3 < \| K_i \|_3$.

R6 Analogously to R4 when s is not a variable: $\| K_{i+1} \|_1 < \| K_i \|_1$.

Next we prove that if $S_j \neq \emptyset$ then $Sol(S_0\Box W_0) = \emptyset$ and hence there is no substitution θ that solves the system and the entailment does not hold. By Lemma 7.11 it is enough to show that $Sol(S_j\Box W_j) = \emptyset$. Since no rule transformation can be applied to this configuration, at least one of the cases below must hold. Notice that in every case the system cannot be solved.
a) $\perp \to s \in S_j$, $s \neq \perp$. Then there exists no total substitution θ such that $s\theta \sqsubseteq \perp$.
b) $h\,\overline{a}_m \to g\,\overline{b}_l$ with either $h \neq g$ or $m \neq l$. Obvious.
c) $h\,\overline{a}_m \to X$, $h\,\overline{a}_m$ not total, $X \in W_j$. Then there is no total substitution θ s.t. $X\theta \subseteq (h\ \overline{a}_m)\theta$.
d) $X \to Y$, $X \neq Y$, $X \notin W_j$, $Y \notin W_j$. Straightforward, from the requirement of $dom(\theta) \subseteq W_j$ in every solution.
e) $X \to h\,\overline{a}_m$, $X \notin W_j$. As the previous case.

Finally, if $S_j = \emptyset$ then
$Sol(\emptyset\Box W_j) = Subst_{W_j}$ where $Subst_{W_j} = \{\theta \in Subst \mid dom(\theta) \subseteq W_j\}$.
We consider the substitution $\theta = \theta_1\theta_2\ldots\theta_j$.
By Lemma 7.11, $Sol(S_0\Box W_0) = (\theta Subst_{W_j}) \upharpoonright_{W_0}$.
Since $id \in Subst_{W_j}$, $\theta id \upharpoonright_{W_0} = \theta \upharpoonright_{W_0} \in Sol(S_0\Box W_0)$, and hence $\theta \upharpoonright_{W_0}$ can be used to prove the entailment between both basic facts. $\Box$

8 Appendix B: Some simple Examples

8.1 Example 1

This is based on an example for logic programming debugging presented in [6]:

```
rev :: [A] → [A]
rev [ ]          →   [ ]
rev (X:Xs)       →   app (rev Xs) (X:[ ])

app :: [A] → [A] → [A]
app [ ] Y        →   Y
app (X:Xs) Y     →   app Xs Y
```

The rule app.2 is erroneous and the goal

$$\mathsf{rev\ (U{:}V{:}[\,])} == \mathsf{R}$$

yields the wrong answer $\{\mathsf{R} \mapsto \mathsf{U{:}[\,]}\}$. This is the debugging session in $\mathcal{TOY}$:

```
Consider the following facts:
1: rev (U:V:[ ]) → U:[ ]
Are all of them valid? ([y]es / [n]o) / [a]bort) n

Consider the following facts:
1: rev (V:[ ]) → V:[ ]
2: app (V:[ ]) (U:[ ]) → U:[ ]
Are all of them valid? ([y]es / [n]o) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: app [ ] (U:[ ]) → U:[ ]
Are all of them valid? ([y]es / [n]o) / [a]bort) y

Rule number 2 of the function app is wrong.
Wrong instance: app (V:[ ]) (U:[ ]) → app [ ] (U:[ ])
```

8.2 Example 2

This example shows how the *insertion sort* algorithm can be programmed in $\mathcal{TOY}$, taking advantage of the possibility of defining *non-deterministic* functions.

```
insertSort :: [A] → [A]
insertSort []          → []
insertSort (X:Xs)    → insert X (insertSort Xs)

% non-deterministic function
insert :: A → [A] → [A]
insert X []          → X:[]
insert X (Y:Ys)    → X:Y:Ys ⇐ X ≤ Y == true
insert X (Y:Ys)    → insert X Ys ⇐ X ≤ Y == false
```

The right hand side of the rule insert.3 should be Y:insert X Ys. Function $\leq$ can be considered predefined and hence correct. The goal

$$\text{insertSort (2:1:[ ])} == \text{R}$$

renders the incorrect answer $\{\text{R} \mapsto \text{(2:[ ])}\}$. The debugging session in $\mathcal{TOY}$ is as follows:

```
Consider the following facts:
1: insertSort (2:1:[ ]) → 2:[ ]
Are all of them valid? ([y]es / [n]o) / [a]bort) n

Consider the following facts:
1: insertSort [1] → 2:[ ]
2: insert 2 [1] → 2:[ ]
Are all of them valid? ([y]es / [n]o) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: insert 2 [ ] → 2:[ ]
Are all of them valid? ([y]es / [n]o) / [a]bort) y

Rule number 3 of the function insert is wrong.
Wrong instance: insert 2 (1:[ ]) → insert 2 [ ] ⇐ 2 ≤ 1 == false
```

8.3 Example 3

Next example is a Haskell-like program computing the frontier of a given tree T. Function frontier is expected to traverse the leaves of T from left to right, collecting them in a list. Trees are represented by constructors leaf/1 and node/2.

```
data tree A = node (tree A) (tree A) | leaf A

frontier :: tree A → [A]
frontier Tree → appendFT Tree [ ]
```

```
appendFT :: (tree A) → [A] → [A]
appendFT (leaf X)              → (X:)
appendFT (node Left Right)     → appendFT Right . appendFT Left

(.) :: (B → C) → (A → B) → A → C
(F . G) X → F (G X)
```

The auxiliary function appendFT is intended to append the frontier of a given tree to a given list. However rule appendFT.2 is wrong, its right hand side should be appendFT Left . appendFT Right (swapping Left and Right). For example, the goal:

$$\text{frontier (node (leaf 0) (leaf 1))} == \text{Xs}$$

computes the wrong answer $\{\text{Xs} \mapsto 1{:}0{:}[\,]\}$ (instead of $\{\text{Xs} \mapsto 0{:}1{:}[\,]\}$). The debugging session in this case looks as follows:

```
Consider the following facts:
1: appendFT (node (leaf 0) (leaf 1)) → (1:).(0:)
2: (1:).(0:[] → 1:0:[]
Are all of them valid? ([y]es / [n]o) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: appendFT (leaf 0) → (0:)
2: appendFT (leaf 1) → (1:)
Are all of them valid? ([y]es / [n]o) / [a]bort) y
Rule number 2 of the function appendFT is wrong.
Wrong instance:
 appendFT (node (leaf 0) (leaf 1)) → appendFT (leaf 1) .appendFT (leaf 0)
```

The buggy function appendFT is higher-order, since it returns functions as results. Therefore the debugger asks the oracle about basic facts whose right-hand sides can be *higher order patterns* as

$$\text{appendFT (node (leaf 0) (leaf 1))} \to (1{:}).(0{:})$$

These questions make sense in our framework, and are crucial to detect the bug in this case.

8.4 Example 4

The last example is intented to compute the infinite list of all the prime numbers, by applying the classical *sieve of Erathostenes* method.

```
primes :: [int]
primes   →   sieve (from 2)
```

```
from:: int → [int]
from N    →    N:from (N+1)

sieve:: [int] → [int]
sieve (X:Xs)    →    X:filter (notDiv X) (sieve Xs)

filter :: (A → bool) → [A] → [A]
filter P []          →    []
filter P (X:Xs)      →    if P X then (X:filter P Xs)
                          else filter P Xs

notDiv :: int → int → bool
notDiv X Y    →    mod X Y > 0

take :: int → [A] → [A]
take N []          →    []
take N (X:Xs)      →    if N > 0 then (X:take (N-1) Xs)
                        else []
```

However, due to the mistake in rule notDiv.1 (its right-hand side should be mod Y X > 0) the goal

$$\text{take 3 primes} == \text{R}$$

yields the incorrect answer $\{\text{R} \mapsto (2{:}3{:}4{:}[\,])\}$. The debugging session locates the wrong answer as follows:

```
Consider the following facts:
1: primes → 2:3:4:5:_
2: take 3 (2:3:4:5:_) → 2:3:4:[]
Are all of them valid? ([y]es / [n]o) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Consider the following facts:
1: from 2 → 2:3:4:5:_
2: sieve (2:3:4:5:_) → 2:3:4:5:_
Are all of them valid? ([y]es / [n]o) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: sieve (3:4:5:_) → 3:4:5:_
2: filter (notDiv 2) (3:4:5:_) → 3:4:5:_
Are all of them valid? ([y]es / [n]o) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.
```

```
Consider the following facts:
1: notDiv 2 3 → true
2: filter (notDiv 2) (4:5:_) → 4:5:_
Are all of them valid? ([y]es / [n]o) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 2.

Consider the following facts:
1: notDiv 2 4 → true
2: filter (notDiv 2) (5:_) → 5:_
Are all of them valid? ([y]es / [n]o) / [a]bort) n
Enter the number of a non-valid fact followed by a fullstop: 1.

Rule number 1 of the function notDiv is wrong.
Wrong instance: notDiv 2 4 → (mod 2 4) > 0
```

Notice the occurrence of symbol _ (representing $\perp$) in many basic facts of the session, approximating results of subcomputations that were not needed.