Elimination of Extra Variables in Functional Logic Programs

Javier de Dios Castro [2]

Telefónica I+D
Madrid, Spain

Francisco J. López-Fraguas [1,3]

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

Abstract

Programs in modern functional logic languages are rewrite systems following the constructor discipline but where confluence and termination are not required, thus defining possibly non strict and non-deterministic functions. While in practice and in some theoretical papers rewrite rules can contain extra variables in right hand sides, some other works and techniques do not consider such possibility. We address in this paper the question of whether extra variables can be eliminated in such kind of functional logic programs, proving the soundness and completeness of an easy solution that takes advantage of the possibility of non-confluence. Although the focus of the paper is mainly theoretical, we give some first steps towards the practical usability of the technique.

Key words: Functional logic programs, extra variables, program transformation

1 Introduction

Declarative languages use rules of different kinds –clauses, equations, ...– to define their basic constructs –predicates, functions, ...–. In some cases, right-hand sides of rules are allowed to contain *extra* variables, i.e. variables not appearing in left-hand sides.

[1] Author partially supported by the Spanish projects TIN2005-09207-C03-03 'MERIT-FORMS' and S-0505/TIC/0407 'PROMESAS-CAM'.
[2] Email: `jdc254@tid.es`
[3] Email: `fraguas@sip.ucm.es`

Logic programs are plenty of extra variables, in most of the cases to express intermediate results, and occasionally to express incomplete information or search. Typically, during computation such extra variables get bound via unification. On the opposite side, functional programs are completely free [4] of extra variables, which make no sense in the paradigm. Intermediate results are simply expressed by nesting of function application.

Functional logic languages (see [12,19] for surveys) aim to combine the best of both paradigms: non-determinism and implicit search from logic programming, functional notation, lazy evaluation, higher order (HO) features and types from functional programming. Functional application and nesting avoid most of the uses of extra variables of logic programs, but there are nevertheless nice specifications in functional logic programming where extra variables are useful, as in the following definition of the property 'Ys is a sublist of Xs':

```
sublist(Xs,Ys) → ifthen(eq(Us++Ys++Vs,Xs),true)
```

Here `++` is list concatenation, `eq` is the equality function and the function `ifthen` is defined by the rule `ifthen(true,Y) → Y`. Notice that `Us, Vs` are extra variables in the rule for `sublist`; notice also that, despite of the presence of extra variables, the program is still confluent, since `sublist(Xs,Ys)` can only return the value `true`.

Modern functional logic languages, like Curry [14] or Toy [6], use also some kind of rewrite rules as programs, that can possibly be non-terminating (nothing new with respect to lazy functional languages) and *non-confluent*, thus defining possibly non-strict non-deterministic functions. The latter is a distinctive feature of such a family of functional logic languages, in which we focus on, and for which we use *FLP* as generic denomination.

Non-determinism may come from overlapping rules for a function, as happens with the following non-deterministic constant (0-ary function) `coin`:

```
coin → 0          coin → 1
```

This kind of non-deterministic functions will be extensively used in this paper.

Non-determinism may also come from the unrestricted use of extra variables in right-hand sides, like in the following variant of the `sublist` program, where a non-deterministic function is defined, instead of a predicate:

```
aSublist(Xs) → ifthen(eq(Us++Ys++Vs,Xs),Ys),
```

Notice that now `Ys,Us,Vs` are extra variables, and that the program is not confluent anymore.

Our interest in elimination of extra variables is twofold: first, we wanted to clarify theoretically the intuitive fact that, with the aid of non-deterministic functions, extra variables are in some sense unnecessary; second, many works in the FLP field only consider programs without extra variables. This happens,

[4] We do not consider local variables introduced by *let* declarations in functional programs as truly extra variables in the same sense of logic programs or rewrite systems.

just to mention a few, with relevant works about operational issues [4], about techniques like partial evaluation [2], or about failure [16].

Elimination of extra variables has deserved some attention in the fields of logic programming [18,3], or conditional term rewriting systems [13,17]. None of these works cover our FLP programs (in particular, confluence is required in all of them), and the involved techniques are more complex than the method to be proposed here, while it is uncertain that they are indeed more effective when applicable to the same situation. This can be seen as a nice example of how widening the view (dropping confluence) sometimes easies the solution.

The rest of the paper is organized as follows: in the next section we give some introductory ideas and discussions; Sect. 3 is the core of the paper, with the precise description of our method and the results of soundness and completeness, using as formal setting CRWL [8,9], a well-known semantic framework for FLP; Sect. 4 addresses more practical issues and gives some experimental results; finally, Sect. 5 summarizes some conclusions, discusses some limitations and points to possible future work. Omitted proofs and additional details, in particular about practical issues, can be found in [7].

2 An introductory example

Consider the following FLP program P for detecting if a natural number (represented by the constructors `z` and `s`) is a perfect square, defining previously some basic functions:

```
add(z,Y)     → Y                times(z,Y)     → z
add(s(X),Y)  → s(add(X,Y))      times(s(X),Y)  → add(Y,times(X,Y))
eq(z,z)      → true             eq(z,s(Y))     → false
eq(s(X),z)   → false            eq(s(X),s(Y))  → eq(X,Y)
 % Similar rules for equality of boolean values
ifthen(true,Y) → Y
pfSquare(X) → ifthen(eq(times(Y,Y),X),true)
```

Notice that the rule for `pfSquare` contains an extra variable `Y`. Therefore P is not a functional program, but fits into the field of functional logic programming. Disregarding concrete syntax [5], P is executable in existing FLP systems like Curry [14] or Toy [6].

To evaluate, for instance, the expression `pfSquare(4)` (we write 4 to abbreviate the term `s(s(s(s(z))))`) we can simply rewrite it producing the new expression `ifthen(eq(times(Y,Y),4),true)`. Now, rewriting is not enough be-

[5] In particular, we use first order syntax throughout the paper, while Curry or Toy use HO curried notation.

cause of the occurrences of the variable `Y`; this is why FLP uses some kind of *narrowing* (see [12]) instead rewriting as operational procedure. In the example, after some failed attempts, narrowing produces the binding `Y/s(s(z))` and the value `true` for the expression.

Our purpose, concreted to this example, is to transform P into a new program P' semantically equivalent to P –in a sense to be made precise in the next section– but having no extra variables. To obtain such P' is quite easy: we replace extra variables by a non-deterministic constant (i.e. 0-ary function) `genNat` whose range of values is the set of all natural numbers. Thus P' is obtained by adding the following definition of `genNat` to P:

```
genNat → z              genNat → s(genNat)
```

and replacing the rule defining `pfSquare` by

```
pfSquare(X) → pfAux(X,genNat)

pfAux(X,Y) → ifthen(eq(times(Y,Y),X),true)
```

It is remarkable the simplicity and locality of the transformation: only the rules with extra variables must be transformed. The transformation becomes even simpler if one uses local definitions, typical of functional programming and also supported by Curry or Toy.[6] In this case the rule for `pfSquare` can be:

```
pfSquare(X) → ifthen(eq(times(Y,Y),X),true) where Y = genNat
```

This contrasts with existing methods for eliminating extra variables in logic programs or some kind of rewrite systems [18,3,13], which are much more complex and do not respect locality by the addition of new arguments to more functions or predicates than those having extra variables.

In our case, if things are so easy is because of the possibility of defining non-deterministic functions offered by FLP. Notice nevertheless that the behavior of non-determinism can be rather subtle, specially when combined with the non-strict semantics (lazy evaluation, in operational terms) considered in FLP. Our transformation is a good example of such subtleties: it is correct under the *call-time choice* regime [15,9] followed in Curry or Toy, but would be unsound under *run-time choice* regime[7]. Figure 1 shows, for the program P' and the expression `pfSquare(s(s(s(z))))`, a possible reduction sequence using run-time choice and returning the value `true`, which is wrong. We underline the redex reduced at each step and write $\rightarrow^*$ to indicate a sequence of reduction steps.

Notice that with call-time choice all the occurrences of `genNat` should be shared and therefore the previous reduction is not legal. Notice also that in the

[6] But not present in the theoretical CRWL framework to be considered in the next section.
[7] Using standard terminology of functional programming, call-time choice with non-strict semantics can be operationally described as *call by need with sharing*, while run-time choice proceeds *without* sharing, what is very different in absence of confluence.

$$\underline{\texttt{pfSquare(3)}} \to \underline{\texttt{pfAux(3,genNat)}} \to$$
$$\texttt{ifthen(eq(times(}\underline{\texttt{genNat}}\texttt{,genNat),3),true)} \to$$
$$\texttt{ifthen(eq(}\underline{\texttt{times(s(genNat),genNat)}}\texttt{,3),true)} \to$$
$$\texttt{ifthen(eq(add(}\underline{\texttt{genNat}}\texttt{,times(genNat,genNat)),3),true)} \to^*$$
$$\texttt{ifthen(eq(}\underline{\texttt{add(3,times(genNat,genNat))}}\texttt{,3),true)} \to^*$$
$$\texttt{ifthen(}\underline{\texttt{eq(s(s(s(times(genNat,genNat)))),3)}}\texttt{,true)} \to^*$$
$$\texttt{ifthen(eq(times(}\underline{\texttt{genNat}}\texttt{,genNat),z),true)} \to$$
$$\texttt{ifthen(eq(}\underline{\texttt{times(z,genNat)}}\texttt{,z),true)} \to$$
$$\texttt{ifthen(}\underline{\texttt{eq(z,z)}}\texttt{,true)} \to \underline{\texttt{ifthen(true,true)}} \to \texttt{true}$$

Fig. 1. A wrong computation using run-time choice

original program P, call-time choice and run-time choice make no difference from the point of view of results, since P is confluent: in both options the reduction of `pfSquare(3)` fails.

For this kind of reasons we feel that, although the transformation is simple, the underlying semantics is complex enough as to demand a formal proof of its correctness, what is done in the next section.

3 Variable elimination: the method and the results

We give here precise definitions and proofs about the program transformation for extra variable elimination. We must start from a formal setting for FLP with a sound treatment of the combination of non-strict semantics and non-determinism with call-time choice, since it is a key issue in our approach.

3.1 The formal setting: CRWL

Since we are interested in ensuring semantic equivalence of the transformed program with respect to the original one, our choice has been the CRWL framework [8,9]. An alternative could have been the approach of [1], but it is two much operationally biased for our purposes.

The *Constructor-based ReWriting Logic* (CRWL) was introduced in [8,9] as a foundation for functional logic programming with lazy non-deterministic functions, covering declarative semantics –logical and model-theoretical– and also operational semantics in the form of a lazy narrowing calculus. We mainly use the logical face of CRWL.

As semantics for non-determinism, CRWL's choice is angelic non-determinism with call-time choice for non-strict functions. Angelic non-determinism means that the results of all the possible computations, due to the different non-determinist choices, are collected by the semantics [8] , even if the possibility of infinite computations is not excluded. Call-time choice means that given a function call $f(e_1, \ldots, e_n)$, one chooses some fixed (possibly partial) value for each of the actual parameters before applying the rewrite rule that defines f. An alternative description, closer to implementations, is the following: call by need is used to reduce $f(e_1, \ldots, e_n)$ but all the multiple occurrences of e_i's that might be introduced in the reduction are shared.

CRWL is widely accepted as solid semantic foundation for FLP, and has been successfully extended in several ways to cover different aspects of FLP like HO, algebraic and polymorphic types or constraints (see [19] for a survey), but we stick here to the original version, which is first order and untyped. In the last section we discuss these limitations which, by the way, are common in most papers about FLP.

We review now the essential notions about CRWL needed for this paper.

3.1.1 CRWL basic concepts

We assume two disjoint sets of constructor symbols $c, d, \ldots \in CS$ and function symbols $f, g, \ldots \in FS$, each symbol having a fixed arity ≥ 0. Constructor terms (c-terms for short) $t, s, \ldots \in CTerm_\Sigma$ are defined as usual, using constructor symbols from CS and variables taken from a countable set $\mathcal{V}$. The set $CTerm_\perp$ of partial c-terms is obtained by allowing also the use of $\perp$ as constant (0-ary) constructor symbol. $\perp$ represents the undefined value, needed to express partial values in the semantics. Expressions $e, l, r, \ldots \in Expr$ are made of variables, constructor symbols and function symbols. Partial expressions from $Expr_\perp$ use also $\perp$. C-terms and expressions are called *ground* if they do not contain variables.

C-substitutions are mappings $\sigma : \mathcal{V} \rightarrow CTerm$ which extend naturally to $\sigma : Expr_\perp \rightarrow Expr_\perp$. Analogously, we define partial C-substitutions as $\sigma : \mathcal{V} \rightarrow CTerm_\perp$. A substitution is *grounding* if $X\sigma$ is ground for all $X \in \mathcal{V}$.

The *approximation ordering* between partial expressions $e \sqsubseteq e'$ is the least partial ordering verifying $\perp \sqsubseteq e$ for all $e \in Expr_\perp$, and such that constructor and function symbols are monotonic.

A *CRWL-program* P is any set of left linear constructor based rewrite rules, i.e. rules of the form $f(t_1, \ldots, t_n) \rightarrow e$, where $f \in FS$ has arity $n \geq 0$, $(t_1, \ldots, t_n)$ is a linear tuple –i.e., no variable occurs twice in it– of c-terms, and e is any expression. Notice that there is no restriction about different rules for the same f, or about variables in e. The set of extra variables of a rule $l \rightarrow e$ is $EVar(l \rightarrow r) = var(r) \backslash var(l)$.

The *set of partial c-instances* of the rules of P is defined as

$$[P]_\perp = \{(l \rightarrow r)\sigma \mid (l \rightarrow r) \in P, \sigma \in CSubst_\perp\}$$

[8] But not by each individual computation, which is only responsible of producing a single value. Different values are usually obtained in different computations, by means of backtracking.

$$\textbf{(BT)}\ \frac{}{e \to \perp}\ e \in \mathit{Exp}_\perp \qquad \textbf{(RR)}\ \frac{}{X \to X}\ X \in \mathcal{V}$$

$$\textbf{(CS)}\ \frac{e_1 \to t_1\ ...\ e_n \to t_n}{c(e_1, ..., e_n) \to c(t_1, ..., t_n)}\quad c \in CS^n,\ t_i \in \mathit{CTerm}_\perp,\ e_i \in \mathit{Exp}_\perp$$

$$\textbf{(FR)}\ \frac{e_1 \to t_1\ ...\ e_n \to t_n \quad e \to t}{f(e_1, ..., e_n) \to t}\quad \text{if } f(t_1, ..., t_n) \to e \in [P]_\perp$$

Fig. 2. The *CRWL* proof calculus

It will play a role when expressing call-time choice below. Notice that if $l \to r \in [P]_\perp$, then also $(l \to r)\sigma \in [P]_\perp$, for any $\sigma \in \mathit{CSubst}_\perp$.

Remark: the original CRWL as presented in [8,9] considered conditional rules of the form $f(t_1, \ldots, t_n) \to e \Leftarrow C_1, \ldots, C_m$, where each C_i is a joinability condition $e \bowtie e'$, expressing that e and e' can be reduced to unifiable c-terms. Joinability conditions were introduced in [8,9] to improve the operational behavior of strict equality, a secondary subject for the aims of this paper. On the other hand, it is an easy fact that conditions can be replaced by the use of the function ifthen, as in the example of Sect. 2. For all these reasons we consider only unconditional rules.

3.1.2 The CRWL proof calculus

CRWL determines the logical semantics of a program P by means of a proof calculus able to derive reduction or approximation statements of the form $e \to t$ ($e \in \mathit{Expr}_\perp, t \in \mathit{CTerm}_\perp$), where 'reduction' includes the possibility of applying rewriting rules from P respecting call-time choice, or replacing some subexpression by $\perp$. Figure 2 shows the rules for CRWL-derivability. BT stands for *Bottom*, RR for *Restricted Reflexivity*, DC for *Decomposition* and OR for *Outer Reduction* The rule BT, in combination with the use of partial instances in OR expresses non-strict semantics. The use of C-instances instead of general instances in OR reflects call-time choice.

As usual with such kind of logical calculi, a derivation for a statement $e \to t$ can be represented by a *proof tree* with the conclusion written at the bottom. We write $P \vdash_{CRWL} \varphi$ to express that the statement φ has a CRWL-derivation from P. We speak sometimes of P-derivations to make explicit which program is used for the derivation. Notice that the CRWL-calculus *is not* an operational procedure for executing programs, but a way of describing the logic of programs.

The following lemmata, expressing that CRWL-derivability is closed under C-substitutions and monotonic, are slight variations of known results about

CRWL.

Lemma 3.1 *Let P be a CRWL-program, $e \in Expr_\perp$ and $t \in CTerm_\perp$. If $P \vdash_{CRWL} e \to t$, then $P \vdash_{CRWL} e\sigma \to t\sigma$, for any $\sigma \in CSubst_\perp$. Furthermore, both derivations can be made as to have the same number of* (OR) *steps.*

Lemma 3.2 *Let P be a CRWL-program, $e, e' \in Expr_\perp$, and $t \in CTerm_\perp$. If $P \vdash_{CRWL} e \to t$ and $e \sqsubseteq e'$, then $P \vdash_{CRWL} e' \to t$.*

3.2 Non deterministic generators can replace extra variables

As illustrated in Sect. 2, extra variables are to be replaced by a non-deterministic constant able to generate all ground c-terms. We start by defining the notion of generator, which depends on the signature; we assume a fixed set CS of constructor symbols.

Definition 3.3 (Grounding generator) A user defined constant (0-ary function) *gen* is called a *grounding generator* if $R_{gen} \vdash_{CRWL} gen \to t$ for any ground $t \in CTerm_\perp$, where R_{gen} is the set of rules defining *gen* (possibly plus some auxiliary functions used in the definition of *gen*).

Notice that if a program P contains a grounding generator, then also $P \vdash_{CRWL} gen \to t$ for any ground $t \in CTerm_\perp$.

It is not difficult to construct grounding generators:

Lemma 3.4 *The function gen defined by the rules:*

$gen \to a$ *% for each constant constructor a*

$gen \to c(gen, \ldots, gen)$ *% for each constructor c of arity $n > 0$*

is a grounding generator.

We now define the program transformation to eliminate extra variables, which is parameterized by a given grounding generator.

Definition 3.5 (Transformed program) Let P be a CRWL-program, and *gen* a grounding generator. The transformed program P' results of adding to P the definition of *gen* and replacing in P every rule $f(t_1, \ldots, t_n) \to r$ with extra variables

$$Var(r) - Var(f(t_1, \ldots, t_n)) = \{X_1, \ldots, X_m\} \neq \emptyset$$

by the two rules

$$f(t_1, \ldots, t_n) \to f'(t_1, \ldots, t_n, gen, \ldots, gen)$$

$$f'(t_1, \ldots, t_n, X_1, \ldots, X_m) \to r$$

where f' is a new function symbol (one for each rule with extra variables).

It is clear that there is no extra variable in P'. The transformation could be generalized, without affecting the validity of the results below, by allowing to use different generators for the different occurrences of gen in the new rule for f.

The following theorem, which is the main result of the paper, establishes to which extent the transformed P' and the original P are equivalent with respect to declarative semantics, i.e. with respect to CRWL-derivability.

Theorem 3.6 *Let P be a CRWL-program, P' its transformed program, $e \in Expr_\perp$ an expression not using the auxiliary symbols of P' and $t \in CTerm_\perp$ a ground c-term. Then $P \vdash_{CRWL} e \rightarrow t \Leftrightarrow P' \vdash_{CRWL} e \rightarrow t$.*

The $\Leftarrow$ part can be understood as correctness of the transformation (for the class of considered reductions) and the $\Rightarrow$ part as completeness. Notice that we cannot expect to drop the groundness condition imposed over t.[9] The simplest example is given by a program P with only one rule $f \rightarrow X$, for which P' would be the definition of gen plus the rules $f \rightarrow f'(gen)$ and $f'(X) \rightarrow X$. It is clear that $P \vdash_{CRWL} f \rightarrow X$ but not $P' \vdash_{CRWL} f \rightarrow X$; what we have instead is $P' \vdash_{CRWL} f \rightarrow t$ for each ground instance of X.

To prove the theorem we need some auxiliary results about the use of variables in CRWL-derivations. The first one establishes that for proving ground statements $e \rightarrow t$ one can write CRWL-derivations without variables at all, even if the program P has extra variables. This does not preclude the existence of other CRWL-derivations with variables for the same statement.

Lemma 3.7 *Let P be a CRWL-program, $e \in Expr_\perp$, and $t \in CTerm_\perp$. If $P \vdash_{CRWL} e \rightarrow t$ and $Var(e \rightarrow t) = \emptyset$, then $e \rightarrow t$ has a CRWL-derivation without variables.*

The following generalization of the previous lemma is not strictly necessary in the rest of the paper, but has some interest in itself.

Lemma 3.8 *Let P be a CRWL-program, $e \in Expr_\perp$, and $t \in CTerm_\perp$. If $P \vdash_{CRWL} e \rightarrow t$, then there is a CRWL-derivation of $e \rightarrow t$ which only uses variables from $e \rightarrow t$.*

In the following lemma, very close already to Theorem 3.6, we show that the transformed program is semantically equivalent to the original one for ground statements.

Lemma 3.9 *Let P be a CRWL-program, P' its transformed program, $e \in Expr_\perp$ an expression not using the auxiliary symbols of P', and $t \in CTerm_\perp$. If $Var(e \rightarrow t) = \emptyset$ then $P \vdash_{CRWL} e \rightarrow t \Leftrightarrow P' \vdash_{CRWL} e \rightarrow t$.*

The previous lemma is already a quite strong equivalence result. It is enough to guarantee, for instance, that in the example of Sect. 2, the transformed program gives `pfSquare(4) -> true`, if it is the case (as happens in-

[9] More precisely, groundness is needed for completeness. It is not difficult to see that correctness holds for all reductions $e \rightarrow t$, as far as e does not use the auxiliary symbols from P'.

deed) that the original program does. Theorem 3.6 improves slightly the result by dropping the condition $var(e) = \emptyset$. Its proof is now easy:

Proof. *(Theorem 3.6)*
$\Rightarrow$) Assume $P \vdash_{CRWL} e \rightarrow t$ with t ground. Let $Var(e) = \{X_1, \ldots, X_n\}$ and consider the substitution $\sigma = \{X_1/\perp, \ldots, X_n/\perp\} \in CSubst_\perp$.

By Lemma 3.1, $P \vdash_{CRWL} e\sigma \rightarrow t\sigma$, i.e., $P \vdash_{CRWL} e\sigma \rightarrow t$, since t is ground.

By Lemma 3.9, $P' \vdash_{CRWL} e\sigma \rightarrow t$, since $e\sigma \rightarrow t$ is ground.

By Lemma 3.2, $P' \vdash_{CRWL} e \rightarrow t$ since $e\sigma \sqsubseteq e$.

$\Leftarrow$) Similar. □

4 Generators in practice

Although our aim in this paper is mainly theoretical, in this section we shortly discuss some practical issues concerning the use of grounding generators and give the results of a small set of experiments. We are not formal in this section; instead we explain things through examples. We use Toy [6] as reference language, but most if not all the comments are also valid for Curry [14].

As a first remark, since FLP languages are typed, one should program one generator for each (data constructed) type. Apart from being much more efficient than a ‘universal’ generator, this prevents ill-typedness. We make some remarks about primitive types and polymorphism in the next section.

Second, not every grounding generator could be used in practice, at least with the most common implementations of FLP languages which perform depth first search of the computation space. Complete implementations like [20] do not present that problem.

Example 4.1 Consider a data type definition `data t = a | b | c(t,t)`. The grounding generator given in Lemma 3 would be

```
gen -> a        gen -> b        gen -> c(gen,gen)
```

The evaluation of `gen`, under depth first search, produces by backtracking the sequence of values `a, b, c(a,a),c(a,b),c(a,c(a,a)),...`. The value `c(b,b)`, among infinitely many others, would never appear, thus destroying the theoretical completeness of the use of generators. This situation is not really new, since depth first search itself destroys the theoretical completeness of the operational procedures of FLP systems. But at least with generators we can do better with little effort.

4.1 Programming fair generators

To program generators that are complete even under depth first search, we can proceed by *iterative deepening*, i.e., by levels of increasing complexity, provided that each level is finite and that the union of all of them covers the whole universe of data values of the given type.

```
gen: generates values of type t

 gen          ->  gen1(z)         gen2(z)       ->  a

 gen1(N)      ->  gen2(N)         gen2(z)       ->  b

 gen1(N)      ->  gen1(s(N))      gen2(s(N))    ->  c(gen2(N),gen3(N))

 gen3(N)      ->  gen2(N)         gen2(s(N))    ->  c(gen4(N),gen2(N))

 gen3(N)      ->  gen4(N)           % using gen4 here avoids repetitions

 gen4(s(N))   ->  gen3(N)
```

Fig. 3. A fair generator for the datatype `t`

Perhaps the simplest choice is to collect at each level all the terms of a given depth. Not only is simple: in our experiments we have found that this choice behaves much better than other criteria for determining levels, like the size of a term, either absolute (number of symbols) or with some kind of weighting.

To program a generator for increasing depth is not difficult. Figure 3 contains a fair generator for the datatype `t` of the example above. We use natural numbers as auxiliary data type; primitive integers could be used instead. A family of auxiliary functions is required to achieve fairness and avoid repetitions.

4.2 Some results

Next, we show a comparison between original programs with extra variables and the transformed programs after eliminating them. For the comparisons we give runtimes and also the number of reductions to head normal form, which is more independent of the actual implementation and computer. These programs are developed in the system TOY [6] and their source code can be found at `http://gpd.sip.ucm.es/fraguas/prole06/examples.toy`.

Table 4 corresponds to some functions over natural numbers: `pfSquare` (see Sect. 2), `even`, which is similar but using addition instead of multiplication, and `divisible`, which recognizes if a natural number is divisible, defined by the rule `divisible(X) -> ifthen(eq(times(s(s(Y)),s(s(Z))),X)`, which contains two extra variables. In all cases we have tried with different natural numbers, one of them producing a failed reduction, which requires to explore the whole search space.

The columns marked with $^{(*)}$ indicate that the reduction finitely fails. As it is apparent, the performance of the transformed program does not differ too much from the original.

Table 5 corresponds to some functions operating over lists of different

Program		N = 100		N = 997		N = 4900	
		ms	hnf	ms	hnf	ms	hnf
even	Original Program	40	957	3366$^{(*)}$	10974	40676	46557
	Transformed Program	70	1008	6130$^{(*)}$	11972	80107	49008
pfSq	Original Program	50	1642	62080$^{(*)}$	972584	7290	275352
	Transformed Program	50	2073	83619$^{(*)}$	1455641	8263	394563
div	Original Program	110	1156	64926$^{(*)}$	1019129	2943	56356
	Transformed Program	100	1205	87375$^{(*)}$	1508596	3083	58805

Fig. 4. Some results for natural numbers

Program		N = 10		N = 30		N = 50	
		ms	hnf	ms	hnf	ms	hnf
sublist	Original Program	10	461	141	2561	609	6261
	Transformed Program	16	776	78	4696	187	11816
last	Original Program	10	558	31	3458	630	8758
	Transformed Program	16	1129	78	8399	234	22469
intersec	Original Program	10	964	155	13554	547	53744
	Transformed Program	30	1624	312	25004	1375	102064

Fig. 5. Some results for lists

datatypes. We have tried with different depths, where the depth of a list depends on its length and also on the depth of its elements. The results are still acceptable. A bit surprising fact, for which we do not have any good explanation, is that for `sublist` the system reports greater number of reductions to hnf but also better runtimes.

For this comparative we have run TOY v.2.2.3 (with Sicstus Prolog 3.11.1.) in a Pentium 4 (3.4 Ghz) with 1Gb of RAM, under Windows XP.

5 Conclusions and Future Work

We have presented a technique for eliminating extra variables in functional logic programs, i.e. variables present in right-hand sides but not in left-hand sides of rules for function definitions. Therefore, extra variables are not necessary, at least in a certain theoretical sense. Compared to the case of logic programs or more restricted classes of rewrite systems, things are easier in our setting due to the possibility of writing non-confluent programs offered by FLP languages like Curry or Toy: occurrences of extra variables are replaced

–in a convenient way as to respect call-time choice semantics adopted by such languages– by invocations to a non-deterministic constant (0-ary function) able to generate all ground constructor terms. Any generating function fulfilling such property can be used, which leaves room to experimentation. The experiments we have made so far indicate that a simple but good option, at least for systems like Toy or Curry using depth first search, is to operate by iteratively increasing the depth of the generated term.

Since generating functions have an associated infinite search space, our method determines in many cases an infinite search space, even if the original definition did not. But due to laziness this is not always the case, as the examples of Table 4 show. The precise relation of our method with respect to finite failure is an interesting subject of future research.

We have proved the adequateness of our proposal in the formal setting of the CRWL framework, which is commonly considered a solid semantic basis for FLP. We have considered a first order, untyped version of CRWL, two limitations that we briefly discuss now:

- Taking into account (constructor based) types involves several aspects: a first one, easily addressable, is that a different generating function should be defined for each type, as was mentioned in Sect. 4. Much more difficult –and not solved in this paper– is the question of polymorphism: to eliminate an extra variable with a polymorphic type, the type should be a parameter of the generating function, to be instantiated during runtime. But runtime managing of types is not available in current FLP languages, and is definitely out of the scope of this paper. Primitive types like real numbers also cause obvious difficulties, but the same stands for the mere use of extra variables with these types, since typically narrowing cannot be used for them, because basic operations (like could be +) are not governed by rewrite rules; these situations rather fall into the field of *constraint* programming.
- Adding HO functions does not change things substantially, except when an extra variable is HO. This is allowed, but certainly rather unfrequent in practice, in the system Toy, based in HO extensions of CRWL [10]. According to the *intensional view* of functions in HO-CRWL, HO variables could be in principle eliminated by a function generating *HO-patterns*. Apart from known subtle problems with respect to types [11], new problems arise, since such generators typically do not accept principal types, and therefore must be implemented as builtins and left outside of the type discipline.

From the practical point of view, some improvement of our methods is desirable, by providing a larger class of generators, identifying their appropriateness to certain classes of problems, or by combining the use of generators with other techniques, for instance based on some kind of partial evaluation.

We have recently known of an independent and contemporaneous work [5] where a program transformation similar to ours for the elimination of extra variables in functional logic programs is proposed. There are nevertheless several differences to be pointed out. One is the lack in [5] of any indication to practical results. But the main one is the formal framework considered in each work: [5] proves the completeness of the transformation with respect to ordinary rewriting, while soundness is proved for a variation of rewriting considering explicit substitutions in order to reflect call-time choice, since otherwise the transformation is unsound, as we pointed out in Sect. 2. We find this change of formal setting a bit unsatisfactory, because completeness is not really proved for call-time choice, which can obtain less results than ordinary rewriting (run-time choice). Furthermore we think that the use of CRWL allows a simpler proof.

We end with a final comment about an interesting potential side-product of elimination of extra variables. We can distinguish two situations where the need of narrowing as operational procedure for FLP appears: the first one is when the initial expression to evaluate has variables, even if the program does not use extra variables; the second one is when, as intermediate step, a rule with extra variables is used, introducing expressions with variables, even if the initial one had not. Although the first appears quite frequently when illustrating narrowing in papers, the main source of expressions with variables to be narrowed is the second one. With the elimination of extra variables, these expressions will not occur, thus opening the door to more efficient implementations of FLP replacing the burden of narrowing and search (*unification + rewriting + backtracking*) by the simpler combination *rewriting + backtracking*.

References

[1] E. Albert, M. Hanus, F. Huch, J. Oliver, G. Vidal. *Operational Semantics for Declarative Multi-Paradigm Languages*, Journal of Symbolic Computation 40(1), pp. 795–829,2005.

[2] M. Alpuente, M. Hanus, S. Lucas and G. Vidal. *Specialization of Functional Logic Programs Based on Needed Narrowing*, Theory and Practice of Logic Programming 5(3), pp. 273-303, 2005.

[3] J. Álvez and P. Lucio. *Elimination of Local Variables from Definite Logic Programs*, Electronic Notes in Theoretical Computer Science Volume 137, Issue 1 , pp. 5–24, July 2005.

[4] S. Antoy, R. Echahed, and M.Hanus. *A Needed Narrowing Strategy*, Journal of the ACM 74 (4), pp. 776–822, 2000.

[5] S. Antoy, and M.Hanus. *Overlapping Rules and Logic Variables in Functional Logic Programs*, to appear in Proc. Int. Conf. on Logic Programming ICLP'06.

[6] R. Caballero, J. Sanchez (eds). *$\mathcal{TOY}$: A Multiparadigm Declarative Language. Version 2.2.3.* Tech. Rep. UCM, July 2006. http://toy.sourceforge.net.

[7] Javier de Dios. *Eliminación de variables extra en programación lógico-funcional*, Master Thesis DSIP-UCM, May 2005, (in Spanish). Available at http://gpd.sip.ucm.es/fraguas/prole06/jdios3ciclo.pdf.

[8] J.C. González-Moreno, M.T. Hortalá-González, F.J. López-Fraguas and M. Rodríguez-Artalejo. *A Rewriting Logic for Declarative Programming*. Proc. European Symp. on Programming (ESOP'96), Springer LNCS 1058, pp. 156–172, 1996.

[9] 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*. Journal of Logic Programming 40(1), pp. 47–87, 1999.

[10] J.C. González-Moreno, M.T. Hortalá-González and M. Rodríguez-Artalejo. *A Higher Order Rewriting Logic for Functional Logic Programming*. Proc. Int. Conf. on Logic Programming, The MIT Press, pp. 153–167, 1997.

[11] 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.

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

[13] M. Hanus. *On Extra Variables in (Equational) Logic Programming*, Proc. 12th International Conference on Logic Programming, MIT Press, pp. 665-679, 1995.

[14] M. Hanus (ed.). *Curry: an Integrated Functional Logic Language*, Version 0.8, April 15, 2003. http://www-i2.informatik.uni-kiel.de/∼curry/.

[15] H. Hussmann. *Non-determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.

[16] F.J. López-Fraguas and J. Sánchez-Hernández. *A Proof Theoretic Approach to Failure in Functional Logic Programming*. Theory and Practice of Logic Programming 4(1), pp. 41–74, 2004.

[17] E. Ohlebusch. *Transforming Conditional Rewrite Systems with Extra Variables into Unconditional Systems*, Proc. 6th Int. Conf. on Logic for Programming and Automated Reasoning, Springer LNAI 1705, pp. 111–130, 1999.

[18] M. Proietti and A. Pettorossi. *Unfolding - definition - folding, in this order, for avoiding unnecessary variables in logic programs*, Theoretical Computer Science 142, pp. 89–124, 1995.

[19] M. Rodríguez-Artalejo. *Functional and Constraint Logic Programming*. In H. Comon, C. Marché and R. Treinen (eds.), *Constraints in Computational Logics, Theory and Applications*, Revised Lectures of the International Summer School CCL'99, Springer LNCS 2002, Chapter 5, pp. 202–270, 2001.

[20] A. Tolmach, S. Antoy and M. Nita. *Implementing Functional Logic Languages Using Multiple Threads and Stores*, Proc. of the Ninth International Conference on Functional Programming (ICFP 2004), ACM Press, pp. 90–102, 2004.