Electronic Notes in Theoretical Computer Science 188 (2007) 3–19 

[www.elsevier.com/locate/entcs](http://www.elsevier.com/locate/entcs)

Extra Variables Can Be Eliminated from Functional Logic Programs

# Javier de Dios Castro[2](#_bookmark0)

*Telef´onica I+D Madrid, Spain*

# Francisco J. L´opez-Fraguas[1](#_bookmark0) *,*[3](#_bookmark0)

*Dep. Sistemas Inform´aticos y Computacio´n 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.

*Keywords:* Functional logic programs, extra variables, program transformation

# 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.

Logic programs are plenty of extra variables, in most of the cases to express intermediate results, and occasionally to express incomplete information or search.

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](mailto:jdc254@tid.es)

3 Email: [fraguas@sip.ucm.es](mailto:fraguas@sip.ucm.es)

1571-0661 © 2007 Elsevier B.V. Open access under [CC BY-NC-ND license.](http://creativecommons.org/licenses/by-nc-nd/3.0/)

doi:10.1016/j.entcs.2006.05.049

Typically, during computation such extra variables get bound via unification. On the opposite side, functional programs are completely free [4](#_bookmark1) 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](#_bookmark31),[19](#_bookmark38)] 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](#_bookmark32)] or Toy [[6](#_bookmark25)] use also some kind of rewrite rules as programs, that can be non-terminating 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 and 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 clar- ify 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, just to mention a few, with relevant works about operational issues [[4](#_bookmark22)], about techniques like partial evaluation [[2](#_bookmark20)], or about failure [[16](#_bookmark33)].

Elimination of extra variables has deserved some attention in the fields of logic programming [[18](#_bookmark37),[3](#_bookmark23)], or conditional term rewriting systems [[13](#_bookmark34),[17](#_bookmark35)]. None of these works cover our FLP programs (in particular, confluence is required in all of them),

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.

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](#_bookmark27),[9](#_bookmark28)], 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. Additional details, in particular about practical issues, can be found in [[7](#_bookmark26)].

# An introductory example

Consider the following FLP program *P* for detecting if a natural number (repre- sented 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](#_bookmark2) , *P* is executable in existing FLP systems like Curry

[[14](#_bookmark32)] or Toy [[6](#_bookmark25)].

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 because of the occurrences of the variable Y; this is why FLP uses some kind of *narrowing* (see [[12](#_bookmark31)]) 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–

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

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](#_bookmark3) 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](#_bookmark37),[3](#_bookmark23),[13](#_bookmark34)], 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 trans- formation is a good example of such subtleties: it is correct under the *call-time choice* regime [[15](#_bookmark36),[9](#_bookmark28)] followed in Curry or Toy, but would be unsound under *run- time choice* regime [7](#_bookmark4) . Figure [1](#_bookmark5) 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 →∗ 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 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.

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.

pfSquare(3) → pfAux(3,genNat) → ifthen(eq(times(genNat,genNat),3),true) → ifthen(eq(times(s(genNat),genNat),3),true) → ifthen(eq(add(genNat,times(genNat,genNat)),3),true) →∗ ifthen(eq(add(3,times(genNat,genNat)),3),true) →∗ ifthen(eq(s(s(s(times(genNat,genNat)))),3),true) →∗ ifthen(eq(times(genNat,genNat),z),true) → ifthen(eq(times(z,genNat),z),true) → ifthen(eq(z,z),true) → ifthen(true,true) → true

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

# 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.

* 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](#_bookmark27),[9](#_bookmark28)]. An alternative could have been the approach of [[1](#_bookmark21)], but it is two much operationally biased for our purposes.

The *Constructor-based ReWriting Logic* (CRWL) was introduced in [[8](#_bookmark27),[9](#_bookmark28)] 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](#_bookmark6) , even if the possibility of infinite computa- tions is not excluded. Call-time choice means that given a function call *f* (*e*1*,... , en*), 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*,... , en*) but

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.

all the multiple occurrences of *ei*’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](#_bookmark38)] 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.

* + 1. *CRWL basic concepts*

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

C-substitutions are mappings *σ* : V → *CTerm* which extend naturally to *σ* : *Expr*⊥ → *Expr*⊥. Analogously, we define partial C-substitutions as *σ* : V → *CTerm*⊥. A substitution is *grounding* if *Xσ* is ground for all *X* ∈ V.

The *approximation ordering* between partial expressions *e* ± *e*' is the least partial ordering verifying ⊥ ± *e* for all *e* ∈ *Expr*⊥, 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*,... , tn*) → *e*, where *f* ∈ *FS* has arity *n* ≥ 0, (*t*1*,... , tn*) 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* → *e* is *EV ar*(*l* → *r*) = *var*(*r*)\*var*(*l*).

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

[*P* ]⊥ = {(*l* → *r*)*σ* | (*l* → *r*) ∈ *P, σ* ∈ *CSubst*⊥}

It will play a role when expressing call-time choice below. Notice that if *l* → *r* ∈

[*P* ]⊥, then also (*l* → *r*)*σ* ∈ [*P* ]⊥, for any *σ* ∈ *CSubst*⊥.

Remark: the original CRWL as presented in [[8](#_bookmark27),[9](#_bookmark28)] considered conditional rules of the form *f* (*t*1*,... , tn*) → *e* ⇐ *C*1*,... , Cm*, where each *Ci* is a joinability condition *e da e*', expressing that *e* and *e*' can be reduced to unifiable c-terms. Joinability conditions were introduced in [[8](#_bookmark27),[9](#_bookmark28)] to improve the operational behavior of strict equality, a secondary subject for the aims of this paper. On the other hand, it is an

(BT)

*e* ∈ *Exp*⊥

(RR)

*X* ∈V

*e* →⊥ *X* → *X*

(CS)

(FR)

*e*1 → *t*1 *... en* → *tn*

*c*(*e*1*, ..., en*) → *c*(*t*1*, ..., tn*) *e*1 → *t*1 *... en* → *tn e* → *t*

*f* (*e*1*, ..., en*) → *t*

*c* ∈ *CSn, ti* ∈ *CTerm*⊥*, ei* ∈ *Exp*⊥

if *f* (*t*1*, ..., tn*) → *e* ∈ [*P* ]⊥

Fig. 2. The *CRWL* proof calculus

easy fact that conditions can be replaced by the use of the function *if then*, as in the example of Sect. 2. For all these reasons we consider only unconditional rules.

* + 1. *The CRWL proof calculus*

CRWL determines the logical semantics of a program *P* by means of a proof cal- culus able to derive reduction or approximation statements of the form *e* → *t* (*e* ∈ *Expr*⊥*,t* ∈ *CTerm*⊥), where ‘reduction’ includes the possibility of applying rewriting rules from *P* respecting call-time choice, or replacing some subexpression by ⊥. Figure [2](#_bookmark7) 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* → *t* can be represented by a *proof tree* with the conclusion written at the bottom. We write *P* ▶*CRW L ϕ* 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* ∈ *Expr*⊥ *and t* ∈ *CTerm*⊥*. If P* ▶*CRW L e* → *t, then P* ▶*CRW L eσ* → *tσ, for any σ* ∈ *CSubst*⊥*. Furthermore, both deriva- tions can be made as to have the same number of* (OR) *steps.*

Lemma 3.2 *Let P be a CRWL-program, e, e*' ∈ *Expr*⊥*, and t* ∈ *CTerm*⊥*. If*

*P* ▶*CRW L e* → *t and e* ± *e*'*, then P* ▶*CRW L e*' → *t.*

* 1. *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 *Rgen* ▶*CRW L gen* → *t* for any ground *t* ∈ *CTerm*⊥, where *Rgen* 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* ▶*CRW L*

*gen* → *t* for any ground *t* ∈ *CTerm*⊥.

It is not difficult to construct grounding generators:

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

*gen* → *a % for each constant constructor a*

*gen* → *c*(*gen,... , gen*) *% for each constructor c of arity n >* 0

*is a grounding generator.*

Proof. The fact that *Rgen* ▶*CRW L gen* → *t* for any ground *t* ∈ *CTerm*⊥ follows from an easy induction over the size of *t*.

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 defi- nition of *gen* and replacing in *P* every rule *f* (*t*1*,... , tn*) → *r* with extra variables

*V ar*(*r*) − *V ar*(*f* (*t*1*,... , tn*)) = {*X*1*,... , Xm*} /= ∅

by the two rules

*f* (*t*1*,... , tn*) → *f* '(*t*1*,... , tn, gen,... , gen*) *f* '(*t*1*,... , tn, X*1*,... , Xm*) → *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 according to Def.* [*3.5*](#_bookmark10)*, e* ∈ *Expr*⊥ *an expression not using the auxiliary symbols of P* ' *and t* ∈ *CTerm*⊥ *a ground c-term. Then P* ▶*CRW L e* → *t* ⇔ *P* ' ▶*CRW L e* → *t.*

The ⇐ part can be understood as correctness of the transformation (for the class of considered reductions) and the ⇒ part as completeness. Notice that we

cannot expect to drop the groundness condition imposed over *t*. [9](#_bookmark13) The simplest example is given by a program *P* with only one rule *f* → *X*, for which *P* ' would be the definition of *gen* plus the rules *f* → *f* '(*gen*) and *f* '(*X*) → *X*. It is clear that *P* ▶*CRW L f* → *X* but not *P* ' ▶*CRW L f* → *X*; what we have instead is *P* ' ▶*CRW L f* → *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* → *t* one can write CRWL-derivations without variables at all, even if the program *P* has extra variables. This does not preclude the existence, for the same statement, of other CRWL-derivations with variables.

Lemma 3.7 *Let P be a CRWL-program, e* ∈ *Expr*⊥*, and t* ∈ *CTerm*⊥*. If P* ▶*CRW L e* → *t and V ar*(*e* → *t*)= ∅*, then e* → *t has a CRWL-derivation without variables.*

Proof. We reason by induction on a complexity measure of the CRWL-derivation for *e* → *t*, defined as |*e* → *t*| = (*n, k*), where *n* is the number of OR steps in the derivation and *k* is the total number of steps. Complexity measures are ordered by the lexicographic ordering .

The base case corresponds to the minimum complexity (0,1). The derivation must be of the form B: *e* →⊥ or of the form DC: *c* → *c*, for a constructor constant

*c*. In both cases the result is trivial. Notice that the case RR: *X* → *X* is excluded since *V ar*(*e* → *t*)= ∅.

For the inductive case, we distinguish two subcases according to the CRWL-rule used in the last inference step.

*e*1 → *t*1 *... en* → *tn*

* DC:

*c*(*e*1*,... , en*) → *c*(*t*1*,... , tn*)

In this case the result is immediate by the induction hypothesis, since |*ei* →

*ti*| |*c*(*e*1*,... , en*) → *c*(*t*1*,... , tn*)| for each 1 ≤ *i* ≤ *n*.

*e*1 → *t*1 *... en* → *tn r* → *t*

* OR:

*f* (*e*1*,... , en*) → *t*

for *f* (*t*1*,... , tn*) → *r* ∈ [*P* ]⊥.

Let *σ* ∈ *CSubst*⊥ be any grounding substitution for *V ar*({*t*1*,... , tn, r*}) (we could even assume *σ* ∈ *CSubst*).

By Lemma [3.1](#_bookmark8) we know that *P* ▶*CRW L eiσ* → *tiσ* (1 ≤ *i* ≤ *n*) and *P* ▶*CRW L rσ* → *tσ*, where the number of OR steps for *eiσ* → *tiσ* and *rσ* → *tσ* does not change w.r.t. *ei* → *ti* and *r* → *t*. Thus, |*eiσ* → *tiσ*| and |*rσ* → *tσ*| are all

|*f* (*e*1*,... , en*) → *t*| , since the derivation for *f* (*e*1*,... , en*) → *t* has one additional OR step. To apply induction hypothesis it remains to see that *eiσ* → *tiσ* and *rσ* → *tσ*, are all ground, but this is easy, since *ei,t* are ground (and therefore *eiσ* = *ei* and *tσ* = *t*) and *tiσ* and *rσ* are ground by construction of *σ*. Hence, by induction hypothesis, all *ei* → *tiσ* and *rσ* → *t* have CRWL-derivations without

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

variables.

Finally, just noticing that *f* (*t*1*σ,... , tnσ*) → *rσ* ∈ [*P* ]⊥, we can build a CRWL- derivation

*e*1 → *t*1*σ ... en* → *tnσ rσ* → *t*

OR:

*f* (*e*1*,... , en*) → *t*

which is free of variables.

Notice that a straightforward induction over the structure (or the size) of CRWL- derivations would not have worked, because nothing ensures that in the case of outer reduction the arrows *ei* → *ti* in the premises are ground. We have needed to ground them by means of an auxiliary substitution *σ*, but then the induction hypothesis would not have been applicable.

The following generalization of the previous lemma is not really necessary in the rest of the paper, but has some interest in itself. Since the result is not strictly needed, we are less formal in the proof.

Lemma 3.8 *Let P be a CRWL-program, e* ∈ *Expr*⊥*, and t* ∈ *CTerm*⊥*. If P* ▶*CRW L e* → *t, then there is a CRWL-derivation of e* → *t which uses only variables from e* → *t.*

Proof. *(Sketch)*. We can replace the variables of *e* → *t* by new constants and apply the previous lemma, obtaining a derivation without variables. It suffices to replace back in this derivation the introduced new constants by the original variables.

In the following lemma, very close already to Theorem [3.6](#_bookmark11), we show that the transformed program is semantically equivalent to the original one for ground state- ments.

Lemma 3.9 *Let P be a CRWL-program, P* ' *its transformed program, e* ∈ *Expr*⊥ *an expression not using the auxiliary symbols of P* '*, and t* ∈ *CTerm*⊥*. If V ar*(*e* → *t*)= ∅ *then P* ▶*CRW L e* → *t* ⇔ *P* ' ▶*CRW L e* → *t.*

Proof. ⇒) Since *V ar*(*e* → *t*)= ∅, Lemma [3.7](#_bookmark12) ensures that *e* → *t* has a *P* -derivation without variables. We reason by induction on the structure of such *P* -derivation.

The base cases, where the derivation consists of an application of B or DC for constants, are trivial. As is Lemma [3.7](#_bookmark12), the rule RR must be ignored.

We consider two inductive cases according to the CRWL-rule used in the last inference step. The case of DC is straightforward using the induction hypothesis.

The involved case is OR, because the program rules change from *P* to *P* '. Assume then a *P* -derivation of the form:

OR: *e*1 → *t*1*σ ... en* → *tnσ rσ* → *t*

*f* (*e*1*,... , en*) → *t*

where *f* (*t*1*,... , tn*) → *r* ∈ *P* , *σ* ∈ *CSubst*⊥, and *tiσ* (1 ≤ *i* ≤ *n*), *rσ* are ground (remember that the whole derivation is ground). As induction hypothesis we have then that *P* ' ▶*CRW L ei* → *tiσ* (1 ≤ *i* ≤ *n*) and *P* ' ▶*CRW L rσ* → *t*.

We must build a *P* '-derivation also for *f* (*e*1*,... , en*) → *t*.

If the rule of *P* , *f* (*t*1*,... , tn*) → *r*, does not have extra variables, the same rule is alsoa rule of *P* '. This fact, together with the induction hypothesis, gives directly a *P* '-derivation. Otherwise, let *X*1*,... , Xm* be the extra variables of *r*. Since *P* ' contains the definition of a grounding generator *gen*, we have *P* ' ▶*CRW L gen* → *Xiσ* (1 ≤ *i* ≤ *m*). It is easy to see that *P* ' ▶*CRW L tiσ* → *tiσ* (1 ≤ *i* ≤ *n*), and then we can build the following *P* '-derivation:

*t*1*σ* → *t*1*σ ... tnσ* → *tnσ gen* → *X*1*σ ... gen* → *Xmσ rσ* → *t*

OR:

*f* '(*t*1*,... , tn, gen,... gen*) → *t*

Finally, since *f* (*t*1*σ,... , tnσ*) → *f* '(*t*1*σ,... , tnσ, gen,... gen*) ∈ [*P* ']⊥, we can build the desired *P* '-derivation:

*e*1 → *t*1*σ ... en* → *tnσ f* '(*t*1*σ,... , tnσ, gen,... gen*) → *t*

OR:

*f* (*e*1*,... , en*) → *t*

⇐) As before, by Lemma [3.7](#_bookmark12) we can start from a *P* '-derivation for *e* → *t* without variables [10](#_bookmark15) . We reason by induction on the size of that derivation.

We focus in the interesting inductive case, when the *P* '-derivation ends with an OR step that uses a program rule of *P* ' coming from a rule in *P* with extra variables. Assume that this *P* -rule is *f* (*t*1*,... , tn*) → *r* and the corresponding rules in *P* ' are *f* (*t*1*,... , tn*) → *f* '(*t*1*,... , tn, gen,... , gen*) and *f* '(*t*1*,... , tn, X*1*,... , Xm*) → *r*. The *P* '-derivation will take the form

*e*1 → *t*1*σ ... en* → *tnσ f* '(*t*1*σ,... , tnσ, gen,... , gen*) → *t*

OR:

*f* (*e*1*,... , en*) → *t*

and we must build a *P* -derivation for the same statement.

The *P* '-derivation for *f* '(*t*1*σ,... , tnσ, gen,... , gen*) → *t* that appears as premise above must use the only *P* '-rule for *f* ', having thus the form:

*t*1*σ* → *t*1*σ ... tnσ* → *tnσ gen* → *X*1*σ ... gen* → *Xmσ rσ* → *t*

OR:

*f* '(*t*1*σ,... , tnσ, gen,... , gen*) → *t*

The induction hypothesis says, in particular, that *P* ▶*CRW L ei* → *tiσ* and *P* ▶*CRW L*

*rσ* → *t*, with which we can build the desired *P* -derivation:

*e*1 → *t*1*σ ... en* → *tnσ rσ* → *t*

OR:

*f* (*e*1*,... , en*) → *t*

Notice that, for applying the induction hypothesis, we have used the fact that neither *ei* (since they are subexpressions of the initial expression) nor *r* (since it comes from a rule of *P* ) contain auxiliary symbols from *P* '.

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 indeed) that the original

10 In this case, this could also be proved without Lemma [3.7](#_bookmark12) using the fact that *P* ' has no extra variables.

program does. Theorem [3.6](#_bookmark11) improves slightly the result by dropping the condition

*var*(*e*)= ∅. Its proof is now easy:

Proof. *(Theorem* [*3.6*](#_bookmark11)*)*

⇒) Assume *P* ▶*CRW L e* → *t* with *t* ground. Let *V ar*(*e*) = {*X*1*,... , Xn*} and consider the substitution *σ* = {*X*1*/* ⊥*,... , Xn/* ⊥} ∈ *CSubst*⊥.

By Lemma [3.1](#_bookmark8), *P* ▶*CRW L eσ* → *tσ*, i.e., *P* ▶*CRW L eσ* → *t*, since *t* is ground. By Lemma [3.9](#_bookmark14), *P* ' ▶*CRW L eσ* → *t*, since *eσ* → *t* is ground.

By Lemma [3.2](#_bookmark9), *P* ' ▶*CRW L e* → *t* since *eσ* ± *e*.

⇐) Similar.

# 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](#_bookmark25)] as reference language, but most if not all the comments are also valid for Curry [[14](#_bookmark32)].

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](#_bookmark39)] 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.

* 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.

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

|  |  |  |
| --- | --- | --- |
| -- gen: *generates values o*  -- gen1(N): *generates values o*  -- gen2(N): *generates values o*  -- gen3(N): *generates values o*  -- gen4(N): *generates values o* | | |
| gen | -> | gen1(z) |
| gen1(N) | -> | gen2(N) |
| gen1(N) | -> | gen1(s(N)) |
| gen3(N) | -> | gen2(N) |
| gen3(N) | -> | gen4(N) |
| gen4(s(N)) | -> | gen3(N) |

|  |  |  |
| --- | --- | --- |
| gen2(z) | -> | a |
| gen2(z) | -> | b |
| gen2(s(N)) | -> | c(gen2(N),gen3(N)) |
| gen2(s(N)) | -> | c(gen4(N),gen2(N)) |

Fig. 3. A fair generator for the datatype t

*f type* t*, of any depth*

*f depth* ≥ *N f depth* = *N f depth* ≤ *N*

*f depth < N*

*% using* gen4 *here avoids repetitions*

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](#_bookmark17) 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.

* 1. *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 run-times and also the number of reductions to head normal form, which is more independent of the actual implementation and computer. These programs are de- veloped in the system TOY [[6](#_bookmark25)] and their source code can be found at the address <http://gpd.sip.ucm.es/fraguas/prole06/examples.toy>.

Table [4](#_bookmark18) 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.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 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

Table [5](#_bookmark19) corresponds to some functions operating over lists of different 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.

# 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](#_bookmark18) 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](#_bookmark16). 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](#_bookmark29)]. 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](#_bookmark30)], 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](#_bookmark24)] 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](#_bookmark24)] of any indication to practical results.

But the main one is the formal framework considered in each work: [[5](#_bookmark24)] 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. A´lvez 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*, Proc. Int. Conf. on Logic Programming (ICLP’06), Springer LNCS 4079, pp. 87–101, 2006.
6. R. Caballero, J. Sanchez (eds). T OY*: A Multiparadigm Declarative Language. Version 2.2.3*. Tech. Rep. UCM, July 2006. [http://toy.sourceforge.net](http://toy.sourceforge.net/) .
7. Javier de Dios. *Eliminaci´on de variables extra en programaci´on l´ogico-funcional*, Master Thesis DSIP- UCM, May 2005. Available at <http://gpd.sip.ucm.es/fraguas/prole06/jdios3ciclo.pdf>.
8. J.C. Gonz´alez-Moreno, M.T. Hortal´a-Gonz´alez, F.J. L´opez-Fraguas and M. Rodr´ıguez-Artalejo. *A Rewriting Logic for Declarative Programming*. Proc. European Symp. on Programming (ESOP’96), Springer LNCS 1058, pp. 156–172, 1996.
9. J.C. Gonz´alez-Moreno, M.T. Hortal´a-Gonz´alez, F.J. L´opez-Fraguas and M. Rodr´ıguez-Artalejo. *An Approach to Declarative Programming Based on a Rewriting Logic*. Journal of Logic Programming 40(1), pp. 47–87, 1999.
10. J.C. Gonz´alez-Moreno, M.T. Hortal´a-Gonz´alez and M. Rodr´ıguez-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. Gonz´alez-Moreno, M.T. Hortal´a-Gonz´alez and M. Rodr´ıguez-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/](http://www-i2.informatik.uni-kiel.de/~curry/).

1. H. Hussmann. *Non-determinism in Algebraic Specifications and Algebraic Programs*. Birkha¨user Verlag, 1993.
2. F.J. L´opez-Fraguas and J. S´anchez-Hern´andez. *A Proof Theoretic Approach to Failure in Functional Logic Programming*. Theory and Practice of Logic Programming 4(1), pp. 41–74, 2004.
3. 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.
4. 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.
5. M. Rodr´ıguez-Artalejo. *Functional and Constraint Logic Programming*. In H. Comon, C. March´e 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.

1. 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.