A Formalization of the Semantics of Functional-Logic Programming in Isabelle*

Francisco J. López-Fraguas¹, Stephan Merz², and Juan Rodríguez-Hortalá¹

¹ Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
fraguas@sip.ucm.es, juanrh@fdi.ucm.es
² INRIA Nancy & LORIA
Stephan.Merz@loria.fr

Abstract. Modern functional-logic programming languages like Toy or Curry feature non-strict non-deterministic functions that behave under call-time choice semantics. A standard formulation for this semantics is the CRWL logic, that specifies a proof calculus for computing the set of possible results for each expression. In this paper we present a formalization of that calculus in the Isabelle/HOL proof assistant. We have proved some basic properties of CRWL: closedness under c-substitutions, polarity and compositionality. We also discuss some insights that have been gained, such as the fact that left linearity of program rules is not needed for any of these results to hold.

1 Introduction

Fully formalizing the (meta)theory of a programming language can be beneficial for developing its foundations. There is an increasing number of researchers (see e.g. [2]) sharing the conviction that the combination formalization+mechanized theorem proving must (and will) play a prominent role in programming languages research and technology. In particular, formalizations help to clarify overlooked aspects, to discover pitfalls, and even to provide new insights; moreover, formalized metatheories lead to mechanized reasoning about programs, giving reliable support to tools like certifying compilers or certified program transformations.

In this paper we formalize the semantics of functional logic programming (FLP), a well established paradigm (see [9]) integrating features of logic and functional languages. In modern FLP languages such as Curry [10] or Toy [14] programs are constructor based rewrite systems that may be non-terminating and non-confuent. Semantically this leads to the presence of non-strict and non-deterministic functions. The semantics adopted for non-determinism is call-time choice [11, 8], informally meaning that in any reduction, all descendants of a given subexpression must share the same value. The semantic framework CRWL³ was proposed in [7, 8] to accomodate this view of non-determinism, and

* This work has been partially supported by the Spanish projects TIN2005-09207-C03-03 (MERIT-FORMS-UCM), S-0505/TIC/0407 (PROMESAS-CAM) and TIN2008-06622-C03-01/TIN (FAST-STAMP).
³ CRWL stands for “Constructor-based ReWriting Logic”.

is nowadays considered the standard semantics of FLP. For the purpose of this paper, the most relevant aspect of CRWL is a proof calculus devised to prove reduction statements of the form $\mathcal{P} \vdash e \rightarrow t$, meaning that $t$ is a possible (partial) value to which $e$ can be reduced using the program $\mathcal{P}$.

We have chosen Isabelle/HOL as concrete logical framework for our formalization. Using such a broadly used system is not only easier, but also more flexible and stable than developing language specific tools like has been done, e.g., for logic programming [15] or functional programming [6].

The remainder of the paper is organized as follows: Sect. 2 contains some preliminaries about the CRWL framework, Sect. 3 presents the Isabelle theories developed to formalize CRWL, and Sect. 4 gives the mechanized proofs of some important properties of CRWL. Finally, Sect. 5 summarizes some conclusions and points to future work.

An extended version of this paper can be found at http://gpd.sip.ucm.es/juanrh/pubs/isabelle-crwl-report.pdf. The Isabelle code underlying the results presented here is available at https://gpd.sip.ucm.es/trac/gpd/wiki/GpdSystems/IsabelleCrwl.

2 Preliminaries

2.1 Constructor-based term rewrite systems

We consider a first-order signature $\Sigma = CS \cup FS$, where $CS$ and $FS$ are two disjoint sets of constructor and defined function symbols respectively, each with associated arity. We write $CS^n$ ($FS^n$ resp.) for the set of constructor (function) symbols of arity $n$. The set $\text{Exp}$ of expressions is inductively defined as

$$\text{Exp} \ni e ::= X \mid h(e_1, \ldots, e_n),$$

where $X \in V$, $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in \text{Exp}$. The set $\text{CTerm}$ of constructed terms (or c-terms) is defined like $\text{Exp}$, but with $h$ restricted to $CS^n$ (so $\text{CTerm} \subseteq \text{Exp}$). The intended meaning is that $\text{Exp}$ stands for evaluate expressions, i.e., expressions that can contain function symbols, while $\text{CTerm}$ stands for data terms representing values. We will write $e, e', \ldots$ for expressions and $t, s, \ldots$ for c-terms. The set of variables occurring in an expression $e$ will be denoted as $\text{var}(e)$. We will frequently use one-hole contexts, defined as

$$\text{Cntxt} \ni C ::= [ ] \mid h(e_1, \ldots, C, \ldots, e_n)$$

for $h \in CS^n \cup FS^n$. The application of a context $C$ to an expression $e$, written $C[e]$, is defined inductively by

$$[ ][e] = e \quad \text{and} \quad h(e_1, \ldots, C, \ldots, e_n)[e] = h(e_1, \ldots, C[e], \ldots, e_n).$$

The set $\text{Subst}$ of substitutions consists of finite mappings $\theta : V \rightarrow \text{Exp}$ (i.e., mappings such that $\theta(X) \neq X$ only for finitely many $X \in V$), which extend naturally to $\theta : \text{Exp} \rightarrow \text{Exp}$. We write $e\theta$ for the application of $\theta$ to $e$, and $\theta\theta'$
for the composition of substitutions, defined by $X(\theta\theta') = (X\theta)\theta'$. The domain of $\theta$ is defined as $\text{dom}(\theta) = \{X \in \mathcal{V} \mid X \theta \neq X\}$. In most cases we will use $c$-substitutions $\theta \in \text{CSubst}$, for which $X \theta \in \text{CTerm}$ for all $X \in \text{dom}(\theta)$.

A CRWL-program (or simply a program) is a set of rewrite rules of the form $f(t) \rightarrow e$ where $f \in \text{FS}_n$, $e \in \text{Exp}$ and $t$ is a linear $n$-tuple of $c$-terms, where linearity means that each variable occurs only once in $t$. Notice that we allow $e$ to contain extra variables, i.e., variables not occurring in $t$. CRWL-programs often allow also conditions in the program rules. However, CRWL-programs with conditions can be transformed into equivalent programs without conditions, therefore we consider only unconditional rules.

### 2.2 The CRWL framework

In order to accommodate non-strictness at the semantic level, we enlarge $\Sigma$ with a new constant constructor symbol $\bot$. The sets $\text{Exp}_\bot$, $\text{CTerm}_\bot$, $\text{Subst}_\bot$, $\text{CSubst}_\bot$ of partial expressions, etc., are defined naturally. Notice that $\bot$ does not appear in programs. Partial expressions are ordered by the approximation ordering $\sqsubseteq$ defined as the least partial ordering satisfying $\bot \sqsubseteq e$ and $e \sqsubseteq e' \Rightarrow C[e] \sqsubseteq C[e']$ for all $e, e' \in \text{Exp}_\bot, C \in \text{Cntxt}$.

This partial ordering can be extended to substitutions: given $\theta, \sigma \in \text{Subst}_\bot$ we say $\theta \sqsubseteq \sigma$ if $X \theta \sqsubseteq X \sigma$ for all $X \in \mathcal{V}$.

The semantics of a program $\mathcal{P}$ is determined in CRWL by means of a proof calculus (see Fig. 1) for deriving reduction statements $\mathcal{P} \vdash e \rightarrow t$, with $e \in \text{Exp}_\bot$ and $t \in \text{CTerm}_\bot$, meaning informally that $t$ is (or approximates) a possible value of $e$, obtained by iterated reduction of $e$ using $\mathcal{P}$ under call-time choice. Rule B (bottom) allows us to avoid the evaluation of any expression, in order to get a non-strict semantics. Rules RR (restricted reflexivity) and DC (decomposition) allow us to reduce any variable to itself, and to decompose the evaluation of an expression whose root symbol is a constructor. Rule OR (outer reduction) expresses that to evaluate a function call we must first evaluate its arguments to get an instance of a program rule, perform parameter passing (by means of a $\text{CSubst}_\bot$ $\theta$) and then reduce the instantiated right-hand side. The use of partial $c$-substitutions in OR is essential to express call-time choice, as only single partial values are used for parameter passing. Notice also that by the effect of $\theta$ in OR.
extra variables in the right-hand side of a rule can be replaced by any c-term, but not by any expression. The CRWL-denotation of an expression $e \in \text{Exp}_\perp$ is defined as $\left[e\right]^P = \{ t \in \text{CTerm}_\perp \mid P \vdash_{CRWL} e \rightarrow t \}$.

3 Formalizing CRWL in Isabelle

3.1 Basic definitions

We describe our formalization of CRWL in Isabelle. The first step is to define elementary types for the syntactic elements.

\begin{verbatim}
datatype signat = fs string | cs string
datatype varId = vi string
datatype exp = perp | Var varId | Ap signat "exp list"
types
  subst = "varId ⇒ exp option"
  rule = "exp * exp"
  program = "rule set"
\end{verbatim}

Signatures are represented by a datatype that provides two constructors cs and fs to distinguish between constructor and function symbols. The type varId is used to represent variable identifiers, which will be employed to define substitutions. Then the datatype exp is naturally defined following the inductive scheme of Exp$_\perp$, therefore with this representation every expression is partial by default.

Substitutions (type subst) are represented as partial functions from variable identifiers to expressions, using Isabelle’s option type. Hence the domain of a substitution $\vartheta$ will be the set of elements from varId for which $\vartheta$ returns some value different from None. Note that this representation does not ensure that domains of substitutions are finite. Our proofs do not rely on this finiteness assumption. Finally we represent a program rule as a pair of expressions, where the first element is considered the left-hand side of the rule and the second the right-hand side, and a program simply as a set of program rules. The set of valid CRWL programs is characterized by a predicate crwlProgram :: "program ⇒ bool" that checks whether the restrictions of left-linearity and constructor discipline are satisfied.

We define a function apSubst :: "subst ⇒ exp ⇒ exp" for applying a substitution to an expression. The composition of substitutions is defined through a function substComp :: "subst ⇒ subst ⇒ subst". The following lemma ensures the correctness of this definition.

\begin{verbatim}
lemma subsCompAp :
  "(apSubst \vartheta (apSubst \sigma e)) = (apSubst (substComp \vartheta \sigma) e)"
\end{verbatim}

Just as ML, the Isabelle type system does not support subtyping, which could have been useful to represent the sets of c-terms and c-substitutions. Instead, we define predicates cterm and csubst characterizing these subtypes. We prove the expected lemmas, such as that the composition of two c-substitutions is a c-substitution, or that the application of a c-substitution to a c-term yields a c-term.
3.2 Approximation order and contexts

Two key notions of CRWL have not yet been formalized: the approximation order \( \sqsubseteq \), which will be used in the formulation of the polarity of CRWL, and the notion of one-hole context, which will be used in the compositionality.

The following inductively defined predicate \( \text{ordap} \) (with concrete infix syntax \( \sqsubseteq \)) models the approximation order.

\[
\text{inductive} \\
\text{ordap} :: \text{exp} \Rightarrow \text{exp} \Rightarrow \text{bool} \\
\text{where} \\
\text{B: } "\perp \sqsubseteq e" \\
\text{V: } "\text{Var x} \sqsubseteq \text{Var x}" \\
\text{Ap: } "[\text{size es = size es'; } \text{ALL } i < \text{size es. es}!i \sqsubseteq es'!i ] \Rightarrow \text{Ap h es} \sqsubseteq \text{Ap h es}'"
\]

Rule B asserts that \( \perp \sqsubseteq e \) holds for every \( e \); rule V is needed for \( \sqsubseteq \) to be reflexive; finally rule Ap ensures closedness under \( \Sigma \)-operations, and thus compatibility with context [3], because \( \sqsubseteq \) is reflexive and transitive, as we will see. The following results state that our formulation of \( \sqsubseteq \) defines a partial order.

\[
\text{lemma ordapRef1 : } "e \sqsubseteq e" \\
\text{lemma ordapTrans:} \\
\text{assumes } "e1 \sqsubseteq e2" \text{ and } "e2 \sqsubseteq e3" \\
\text{shows } "e1 \sqsubseteq e3" \\
\text{lemma ordapAntisym:} \text{assumes } "e1 \sqsubseteq e2" \text{ and } "e2 \sqsubseteq e1" \\
\text{shows } "e1 = e2" \\
\text{definition ordap_less ("_ \sqsubseteq _" [51,51] 50) where} \\
"e \sqsubseteq e' \equiv e \sqsubseteq e' \land e \neq e'" \\
\text{interpretation exp : order [ordap ordap_less]}
\]

Contexts are represented as the datatype \text{cntxt}, defined as follows:

\[
\text{datatyp e} \text{cntxt = Hole | Cperp | CVar varId } \\
\text{| CAp signat "cntxt list"}
\]

Note that \text{cntxt} cannot follow the inductive structure of \text{Cntxt} with precision, because the type system of Isabelle is not expressive enough to allow us to specify that only one of the arguments of \text{CAp} will be a context and the others will be expressions. Then our contexts are defined as expressions with possibly some holes inside. Therefore the datatype \text{cntxt} represents contexts with any number of holes, even zero holes, and the function \text{apCon : "exp \Rightarrow cntxt \Rightarrow exp"} is defined so it puts the argument expression in every hole of the argument context. In order to characterize contexts with just one hole, we define a function \text{numHoles : "cntxt \Rightarrow nat"} that returns the numbers of holes in a context. Using it we can define define predicates \text{oneHole} and \text{noHole} and prove the following lemmas.
lemma noHoleApDontCare :
  assumes "noHole xC"
  shows "apCon e xC = apCon e' xC"

lemma oneHole :
  assumes "oneHole (CAp h xCs)"
  shows "∃ xC yCs zCs. xCs = (yCs @ xC # zCs) ∧ oneHole xC ∧
                     (∀ c ∈ set (yCs @ zCs). noHole c)"

3.3 The CRWL logic in Isabelle/HOL

The CRWL logic has been formalized through the inductive predicate clto with
infix notation "_ " " → _". The rules defining clto faithfully follow the in-
ductive structure of the definition of CRWL as it is presented in Fig. 1.

inductive
  clto :: "program ⇒ exp ⇒ exp ⇒ bool" ("_ " " → _")
where
  B[intro]: "prog " " exp " " perp"
  | RR[intro]: "prog " " Var v " " Var v"
  | DC[intro]: "|size es = size ts;
               ∃i < size es. prog " es!i " ts!i
               ] =⇒ prog " Ap (cs c) es " Ap (cs c) ts"
  | OR[intro]: "|(Ap (fs f) ps, r) ∈ prog ; csubst ϑ ;
               size es = size ps ;
               ∃i < size es. prog " es!i " apSubst ϑ (ps!i);
               prog " apSubst ϑ r " t
               ] =⇒ prog " Ap (fs f) es " t"

Using clto we can easily define the CRWL denotations in Isabelle as follows.

definition den :: "program ⇒ exp ⇒ exp set" where
  "den P e = {t. P " e " t}"

4 Reasoning about CRWL in Isabelle

The first interesting property that we are proving about CRWL expresses that
evaluation is closed under c-substitutions: reductions are preserved when terms
are instantiated by c-substitutions.

theorem crwlClosedCSubst :
  assumes "prog " e " t" and "csubst ϑ"
  shows "prog " apSubst ϑ e " apSubst ϑ t"

The proof of this lemma proceeds by induction on the CRWL-proof of the hy-
pothesis, therefore we will have one case for each CRWL rule. The first three
cases are proved automatically. However, to prove the case for rule OR Isabelle
needs some help from us. We need to prove

  prog " (Ap (fs f) (map (apSubst ϑ) es)) " (apSubst ϑ t)
and then let the simplifier apply the definition of \texttt{apSubst}. In the proof for that subgoal we used lemma \texttt{CSubsComp} to ensure that the c-substitution \( \mu \) used for parameter passing composed with the c-substitution \( \delta \) in the hypothesis yields another c-substitution, and lemma \texttt{subsCompAp} to guarantee the correct behaviour of the composition for those c-substitutions.

Note that for this result to hold no additional hypotheses about the program or the expressions involved are needed. In particular, this implies that the result holds even for programs that do not follow the constructor discipline or that have non left-linear rules. The Isabelle proof clearly shows that the important ingredients are the use of c-substitutions for parameter passing and the reflexivity of \texttt{CRWL} wrt. c-terms, expressed by lemma \texttt{ctermRefl}, which allows us to reduce to itself any expression \( X \delta \) coming from a premise \( X \rightarrow X \).

The second property that we address is the \textit{polarity} of \texttt{CRWL}. This property is formulated by means of the approximation order and roughly says that if we can compute a value for an expression then we can compute a smaller value for a bigger expression. Here we should understand the approximation order as an information order, in the sense that the bigger the expression, the more information it gives, and so more values can be computed from it.

\textbf{Theorem crwlPolarity :}  
assumes "prog |- e \rightarrow t" and "e \sqsubseteq e'" and "t \sqsubseteq t"  
shows "prog |- e' \rightarrow t'"

using assms proof (induct arbitrary: e' t')

The idea of the proof is to construct a \texttt{CRWL}-proof for the conclusion from the \texttt{CRWL}-proof of the hypothesis, hence it is natural to proceed by induction on the structure of this proof (method \texttt{induct}). The qualifier \texttt{arbitrary} is used to generalize the assertion for any expressions \( e' \) and \( t' \). The proof also relies on the following additional lemmas about the approximation order, which were proved automatically by Isabelle.

\textbf{Lemma ordapPerp:} assumes "e \sqsubseteq \texttt{perp}" shows "e = \texttt{perp}"

\textbf{Lemma ordapVar:} assumes "Var v \sqsubseteq e" shows "e = Var v"

\textbf{Lemma ordapVar_converse:}  
assumes "e \sqsubseteq Var v" shows "e = \texttt{perp} \lor e = Var v"

\textbf{Lemma ordapAp:}  
assumes "Ap h es \sqsubseteq e'"  
shows "\exists es'. e' = Ap h es' \land size es = size es' \land (\forall i < size es. es!i \sqsubseteq es'!i)"

\textbf{Lemma ordapAp_converse:}  
assumes "e' \sqsubseteq Ap h es"  
shows "e' = \texttt{perp} \lor (\exists es'. e' = Ap h es' \land size es = size es' \land (\forall i < size es. es'!i \sqsubseteq es!i))"

The inductive proof for theorem \texttt{cwlPolarity} again considers each \texttt{CRWL} rule in turn. In the case for B we have \( t = \texttt{perp} \), hence we just have to apply \texttt{ordapPerp} to get \( t' = \texttt{perp} \), and then use the \texttt{CRWL} rule B. Regarding \texttt{RR}, as
then $t = \text{Var} \ v$, by ordapVar_converse we get that either $t' = \text{perp}$ or $t' = \text{Var} \ v$. The first case is trivial, and in the latter we just have to apply ordapVar getting $e' = \text{Var} \ v$, which is enough for Isabelle to finish the proof automatically. The case of DC is more complicated. Again we obtain two cases for $t' = \text{perp}$ and $t'$ a constructor application, by using lemma ordapAp_converse. While the first case is trivial, the second one requires some involved reasoning over the list of arguments, using the information we get from applying lemma ordapAp. Finally, the proof for OR is similar to the second case of the proof for DC, with a similar manipulation of the list of arguments, and the use of lemma ordapAp to obtain the induction hypothesis for the arguments.

Once again we find that this proof does not require any hypothesis on the linearity or the constructor discipline of the program: this is indeed quite obvious because this property only talks about what happens when we replace some subexpression by perp.

Finally we will tackle the compositionality of CRWL, that says that if we take a context with just one hole and an expression, then the set of values for the expression put it that context will be the union of the set of values for the result of putting each value for the expression in that context.

```isar
theorem compCRWL : assumes "oneHole xC" shows "den P (apCon e xC) = (\bigcup t \in \text{den} P \ e \cdot \text{den} P (\text{apCon} \ t \ xC))"
```

We have proved the two set inclusions separately as auxiliary lemmas compCRWL1 and compCRWL2. The proofs of these lemmas are quite laborious but essentially proceed by induction on the CRWL-proof in their hypothesis, using it to build a CRWL-proof for the statement in the conclusion. In these proofs, Lemma noHoleApDontCare from Subsect. 3.2 is fundamental.

Again, while theorem compCRWL requires the context to have just one hole, it does not assume the linearity or constructor discipline of the program. This came as a surprise to us, and initially made us doubt about the accuracy of our formalization of CRWL. But it turns out that although CRWL is designed to work with CRWL-programs, that fulfil these restrictions, it can also be applied to general programs. For those programs some properties, such as the theorems crwlClosedCSubst, crwlPolarity, and compCRWL still hold, but other fundamental properties do not, in particular the strong adequacy results w.r.t. its operational counterparts of [8, 12, 1]. The point is that for those programs CRWL does not deliver the “intended semantics” anymore. And this is not strange, because that semantics was intended with CRWL-programs in mind. For example, consider the non linear program $P = \{f(X, X) \rightarrow a\}$. There is a CRWL-proof for the statement $P \vdash f(a, b) \rightarrow a$ but this value cannot be computed in any of the operational notions of [8, 12, 1] nor in any implementation of FLP, in which the independence of the matching process of the arguments — derived from left-linearity of program rules — is assumed. It is also not very natural that $f(a, b)$ could yield the value $a$ for the arguments $a$ and $b$ being different values, which implies that the semantics defined by CRWL for non left-linear
programs is pretty odd. But that is not a big problem, because we only care about the properties of CRWL for the kind of programs it has been designed to work with. And if it enjoys some interesting properties for a bigger class of programs that is fine, because that nice properties will be inherited by the class of CRWL-programs.

On the other hand, for programs not following the constructor discipline, we will not even be able to have a matching for an argument of a rule which is not a constructor, because in the rule OR we have to reduce every argument of a function call to a value, which will be a c-term by Lemma ctermVals (see the extended version of this paper), and so could never be an instance of expression containing function symbols. Thus, the rule OR could not be used for program rules not following the constructor discipline.

5 Conclusions

This paper presented a formalization of the essentials of CRWL \cite{7, 8}, a well-known semantic framework for functional logic programming, in the interactive proof assistant Isabelle/HOL. We chose that particular logical framework for its stability and its extensive libraries. The Isar proof language allowed us to structure the proofs so that they become quite elegant and readable, as can be observed by looking at the Isabelle code.

Our formalization is generic with respect to syntax, and includes important auxiliary notions like substitutions or contexts. This is in contrast to previous work \cite{4, 5} that focused on formalizing the semantics of each concrete program. In contrast, our paper focuses on developing the metatheory of the formalism, allowing us to obtain results that are more general and also more powerful: we formally prove essential properties of the paradigm like polarity or compositionality of the CRWL-semantics. We plan to extend our theories so that we will be able to reason about properties of concrete programs by deriving theorems that express verification conditions in the line of those stated in \cite{4, 5}.

While developing the formalization we realized an interesting fact not pointed out before: properties like polarity or compositionality do not depend on the constructor discipline and left-linearity imposed to programs. However, such requirements will certainly play an essential role when extending our work to formally relate the CRWL-semantics with operational semantics like the one developed in \cite{12}, one of our intended subjects of future work. We think that could be interesting in several ways. First of all it would be a further step in the direction of challenge 3 of \cite{2}, “Testing and Animating wrt the Semantics”, because we would end up getting an interpreter of CRWL during the process. We should then also formalize the evaluation strategy for the operational semantics, obtaining an Isabelle proof of its optimality. Finally there are precedents \cite{13, 12} of how the combination of a denotational and operational perspective is useful for general semantic reasoning in FLP.
References