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

(Extended version)

Tech. Rep. SIC-4-09, 2009

Francisco J. López-Fraguas[1], Stephan Merz[2], and Juan Rodríguez-Hortalá[1]

[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

A key aspect of research on any programming language paradigm is providing precise theoretical foundations, usually in the form of some mathematical theory for the (denotational, operational) semantics of the language. This is specially true in the case of *declarative* languages, that frequently are designed with an underlying corresponding theory or formalism in mind (e.g., Horn logic for logic programming, λ-calculus or term rewriting for functional programming). Fully formalizing the (meta)theory of a programming language or paradigm is a further step that can be done in the development of 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 the next future of programming languages research and technology. Just a couple of reasons: formalizations help to clarify overlooked aspects or overcome common misunderstandings, to discover some pitfalls, or even to provide new insights; moreover, formalized metatheories are a direct vehicle to mechanized reasoning about programs, and therefore a support for software development tools like certifying compilers or certified program transformations.

In this paper we address the problem of formalizing the semantics of functional logic programming (FLP), a well established paradigm aiming at integrating the best features of logic and functional languages (see [9] for a recent survey on FLP). In modern FLP languages such as Curry [10] or Toy [14] programs are constructor based rewrite systems that may be non-terminating and non-confluent. 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 *CRWL*[3] was proposed in [7, 8] to accomodate this view of non-determinism, and 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

* 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".

form $\mathcal{P} \vdash e \twoheadrightarrow t$, 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 [16] or functional programming [6].

The remainder of the paper is organized as follows: Sect. 2 contains some preliminaries about the *CRWL* framework and the Isabelle system, 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.

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 *Exp* of *expressions* is inductively defined as

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

where $X \in \mathcal{V}$, $h \in CS^n \cup FS^n$ and $e_1, \ldots, e_n \in Exp$. The set *CTerm* of *constructed terms* (or *c-terms*) is defined like *Exp*, but with h restricted to CS^n (so $CTerm \subseteq Exp$). The intended meaning is that *Exp* stands for evaluable expressions, i.e., expressions that can contain function symbols, while *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 $var(e)$. We will frequently use *one-hole contexts*, defined as

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

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

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

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

A CRWL-*program* (or simply a *program*) is a set of rewrite rules of the form $f(\overline{t}) \to e$ where $f \in FS^n$, $e \in Exp$ and $\overline{t}$ is a linear n-tuple of c-terms, where linearity means that each variable occurs only once in $\overline{t}$. Notice that we allow e to contain *extra variables*, i.e., variables not occurring in $\overline{t}$. *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 accomodate non-strictness at the semantic level, we enlarge Σ with a new constant constructor symbol $\bot$. The sets $Exp_\bot$, $CTerm_\bot$, $Subst_\bot$, $CSubst_\bot$ 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

$$\bot \sqsubseteq e \quad \text{and} \quad e \sqsubseteq e' \Rightarrow \mathcal{C}[e] \sqsubseteq \mathcal{C}[e'] \text{ for all } e, e' \in Exp_\bot, \mathcal{C} \in Cntxt$$

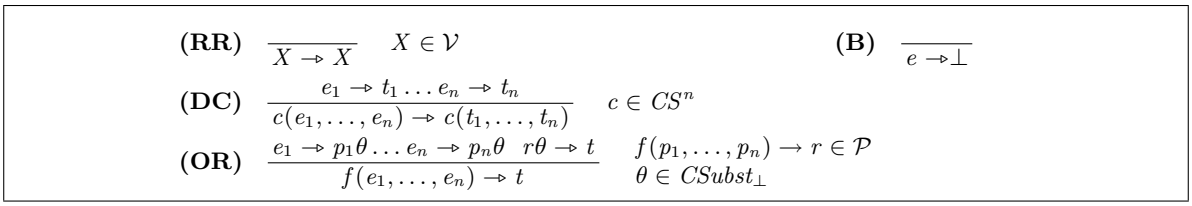

Fig. 1. Rules of *CRWL*

This partial ordering can be extended to substitutions: given $\theta, \sigma \in Subst_\bot$ 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 $\mathcal{P} \vdash e \twoheadrightarrow t$, with $e \in Exp_\bot$ and $t \in CTerm_\bot$, 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 $CSubst_\bot\ \theta$) 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 θ 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 $e \in Exp_\bot$ is defined as $[\![e]\!]^{\mathcal{P}} = \{t \in CTerm_\bot \mid \mathcal{P} \vdash_{CRWL} e \twoheadrightarrow t\}$.

2.3 Isabelle/HOL

Isabelle [15] is a generic framework for mechanizing logics. The syntax of the target logic is encoded in the simply typed λ-calculus. The logic's proof system (axioms and proof rules) is represented in natural deduction style via sequents

$$\bigwedge x.\ [\![P_1;\ \ldots;\ P_n]\!] \Longrightarrow Q$$

where P_i and Q are propositions written in the syntax of the target logic that may contain the universally quantified variables x. The Isabelle kernel provides elementary functions for combining sequents, and these functions are the only means to derive theorems. Assuming a correct implementation of the kernel functions and the soundness of the proof system and its encoding, this design ensures that only actual theorems can be derived. Isabelle also comes with automatic proof methods, including first-order reasoners, a rewriting engine, and a decision procedure for linear arithmetic, which can be instantiated for different logics. Because the results of these proof methods are ultimately certified by applications of the kernel functions, they are not critical for correctness.

Users rarely encode new logics from scratch: most applications are encoded in predefined Isabelle object logics, which provide a rich library of definitions and theorems and instantiate the generic proof machinery for immediate use. Traditionally, proofs consisted of sequential applications of tactics that reduce the statement of a theorem to zero or more subproblems, until no open goals were left. This low-level style of reasoning has been largely replaced by a structured declarative proof language called Isar in which a user can write a proof in a language resembling standard mathematical prose, and freely mix forward and backward reasoning. Although Isar proofs are more verbose, they can be easier to read and to maintain, and we use them throughout our encoding of *CRWL*.

Isabelle/HOL is the encoding in Isabelle of higher-order logic (following Church's Simple Theory of Types). It has been used for numerous verification projects including formalized mathematics, programming language semantics, and security protocols, to name just a few examples. We give a brief introduction to

some of the features of Isabelle/HOL that we use in the following; extensive documentation is available from the Isabelle Web site.[4] In the following, we tacitly refer to Isabelle/HOL when speaking of Isabelle.

Types in Isabelle. Type constructors include *bool*, the type `'a` $\Rightarrow$ `'b` of total functions with arguments of type `'a` and results of type `'b`, and the product type `'a` $*$ `'b`. The function arrow is right-associative: `'a` $\Rightarrow$ `'b` $\Rightarrow$ `'c` is parsed as `'a` $\Rightarrow$ (`'b` $\Rightarrow$ `'c`). Function application is written as juxtaposition of the function and its argument(s). Functional values are written as λ-terms. The function and product type constructors are polymorphic: `'a` and `'b` are type variables that can be instantiated by arbitrary types. Type inference is usually implicit; type declarations are written in the form $v :: ty$ where v is the entity to be declared and ty is a type.

Sets are identified with their characteristic predicates: the type `'a` *set* is synonymous in Isabelle with the type `'a` $\Rightarrow$ *bool*. Isabelle comes with a facility for defining inductive data types. For example,

datatype *nat* =
  Zero ("0")
| *Suc nat*

defines the Peano numbers as an inductive data type with a 0-ary constructor *Zero* (written 0) and a 1-ary constructor *Suc*. In order to ensure that an inductive data type is well-formed, all occurrences of the type being defined (such as *nat* in the example) as arguments of the constructors must be positive.

Inductive type constructors can be polymorphic as well. For example, the type `'a` *option* is defined as a data type with a 0-ary constructor *None* and a 1-ary constructor *Some* `'a`. It can be understood as augmenting type `'a` by an additional "undefined" value. Isabelle also defines function

$$the :: \texttt{'a}\ option \Rightarrow \texttt{'a}$$

such that $the(Some\ x) = x$, for any x of type `'a`. The option type is used to represent partial functions in a logic of total functions: type `'a` $\rightharpoonup$ `'b` is a synonym for the type `'a` $\Rightarrow$ (`'b` *option*); a function of such a type returns *None* for an argument outside the domain of the partial function, and *Some* y (for an appropriate value y) otherwise.

Locales. An Isabelle *locale* is a module whose interface consists of a signature (in the form of a number of operators) and a number of assumptions (logical formulas). For example, a locale of partial orders would introduce a constant

$$leq :: \texttt{'a} \Rightarrow \texttt{'a} \Rightarrow bool$$

and assumptions stating reflexivity, anti-symmetry, and transitivity of *leq*. Locales serve to structure logical theories: inside a locale, the operators making up its signature are considered as constants, and the assumptions as axioms. Properties of these and further derived operators can be proved. For example, one could define a strict variant *less* of *leq* and prove further transitivity properties involving *less* and *leq*. These generic developments can be reused in *interpretations*, which instantiate the locale by concrete operators. For example, the partial order locale could be interpreted by instantiating *leq* by the standard ordering $\leq$ on natural numbers. At this point, the user must prove that the locale assumptions are indeed satisfied by the instance. If they are, all derived operators and facts are available to the interpretation.

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.

[4] `http://isabelle.in.tum.de/`

```
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"
```

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 ϑ will be the set of elements from `varId` for which ϑ 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.

```
lemma subsCompAp :
  "(apSubst ϑ (apSubst σ e)) = (apSubst (substComp ϑ σ) e)"
```

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 `ordap` (with concrete infix syntax $\sqsubseteq$) models the approximation order.

```
inductive
  ordap :: "exp ⇒ exp ⇒ bool" ("_ ⊑ _" [51,51] 50)
where
  B: "perp ⊑ e"
| V: "Var x ⊑ Var x"
| Ap: "⟦ size es = size es' ; ALL i < size es. es!i ⊑ es'!i ⟧
      ⟹ Ap h es ⊑ Ap h es'"
```

Rule `B` asserts that `perp ⊑ e` holds for every `e`; rule `V` is needed for $\sqsubseteq$ to be reflexive; finally rule `Ap` ensures closedness under Σ-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.

```
lemma ordapRefl : "e ⊑ e"
lemma ordapTrans :
  assumes "e1 ⊑  e2" and "e2 ⊑ e3"
  shows "e1 ⊑  e3"
lemma ordapAntisym :
  assumes "e1 ⊑ e2" and "e2 ⊑ e1"
  shows "e1 = e2"
definition ordap_less ("_ ⊏ _" [51,51] 50) where
  "e ⊏ e' ≡ e ⊑ e' ∧ e ≠ e'"
interpretation exp : order [ordap ordap_less]
```

Contexts are represented as the datatype `cntxt`, defined as follows:

```
datatype cntxt = Hole | Cperp | CVar varId
               | CAp signat "cntxt list"
```

Note that `cntxt` cannot follow the inductive structure of *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 `CAp` will be a context and the others will be expressions. Then our contexts are defined as expressions with possible some holes inside. Therefore the datatype `cntxt` represents contexts with any number of holes, even zero holes, and the function `apCon :: "exp ⇒ cntxt ⇒ 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 `numHoles :: "cntxt ⇒ nat"` that returns the numbers of holes in a context. Using it we can define define predicates `oneHole` and `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 inductive structure of the definition of *CRWL* as it is presented in Fig. 1.

```
inductive
  clto :: "program ⇒ exp ⇒ exp ⇒ bool"  ("_ ⊢ _ → _" [100,50,50] 38)
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

Using the elements defined in the previous section, Isabelle is already able to prove some simple lemmas automatically, like the following reflexivity of *CRWL* for values.

```
lemma ctermRefl : assumes "cterm t" shows "prog ⊢ t → t"
using assms
by (induct t rule: cterm.induct, auto)
```

Or the property that every value computed by *CRWL* is a partial c-term.

```
lemma ctermVals : assumes "P ⊢ e → t" shows "cterm t"
using assms
by (induct arbitrary: e, auto simp add: list_all_length list_ball_code)
```

And even more complex properties, like the following relating two inductive notions like `clto` and `ordap`.

```
lemma lessCTermOrdapL : assumes "P ⊢ t → s" and "cterm t" shows "s ⊑ t"
using assms
by (induct, auto)
```

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 hypothesis, therefore we will have one case for each *CRWL* rule. The first three cases are proved automatically, although the case for `RR` needs some tuning of the simplifier.

```
using assms
proof induct
  case B thus ?case by auto
next
  case (RR prog v) thus ?case by (simp add: CSubsCterm ctermRefl del: apSubst.simps)
next
  case (DC es ts prog c) thus ?case by auto
next
```

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 `apSubst`. In the proof for that subgoal we used lemma `CSubsComp` to ensure that the c-substitution μ used for parameter passing composed with the c-substitution ϑ in the hypothesis yields another c-substitution, and lemma `subsCompAp` to guarantee the correct behaviour of the composition for those c-substitutions.

```
  case (OR f ps r prog μ es t)
    from OR have "prog ⊢ (Ap (fs f) (map (apSubst ϑ) es)) → (apSubst ϑ t)"
    proof -
      from OR (2) and OR (7) and CSubsComp have "csubst (substComp ϑ μ)" by simp
```

```
      moreover
       from OR (3) have "size (map (apSubst ϑ) es) = size(ps)" by simp
      moreover
       from OR  have "ALL i< size (map (apSubst ϑ) es). clto prog ((map (apSubst ϑ) es)
! i) (apSubst (substComp ϑ μ) (ps ! i))"
       proof -
         from OR (4) and OR (7) have "ALL i < size es. clto prog (apSubst ϑ (es ! i))
(apSubst ϑ (apSubst μ (ps ! i)))" by simp
         with subsCompAp have "ALL i < size es. clto prog (apSubst ϑ (es ! i)) (apSubst
(substComp ϑ μ) (ps ! i))" by simp
         thus ?thesis by simp
       qed
      moreover
       from OR have "clto prog (apSubst (substComp ϑ μ) r) (apSubst ϑ t)"
       proof -
         from OR (6) and OR (7) have "clto prog (apSubst ϑ (apSubst μ r)) (apSubst ϑ t)"
by simp
         with subsCompAp show ?thesis by simp
       qed
      ultimately show ?thesis using OR (1) by blast
     qed
     thus ?case by simp
qed
```

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 *CRWL* wrt. c-terms, expressed by lemma `ctermRefl`, which allows us to reduce to itself any expression `Xϑ` coming from a premise `X → X`.

The second property that we address is the *polarity of* 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.

```
theorem crwlPolarity :
  assumes "prog ⊢ e → t" and "e ⊑ e'" and "t' ⊑ t"
  shows "prog ⊢ e' → t'"
using assms proof (induct arbitrary: e' t')
```

The idea of the proof is to construct a *CRWL*-proof for the conclusion from the *CRWL*-proof of the hypothesis, hence it is natural to proceed by induction on the structure of this proof (method `induct`). The qualifier `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. These proofs make use of the method `ordap.cases`, which performs a case distinction analysis based on the alternatives provided by the definition of `ordap`.

```
lemma ordapPerp: assumes "e ⊑ perp" shows "e = perp"
using assms
by (rule ordap.cases, auto)
```

```
lemma ordapVar: assumes "Var v ⊑ e" shows "e = Var v"
using assms
by (rule ordap.cases, auto)

lemma ordapVar_converse: assumes "e ⊑ Var v" shows "e = perp ∨ e = Var v"
using assms
by (rule ordap.cases, auto)

lemma ordapAp:
   assumes "Ap h es ⊑ e'"
   shows "∃es'. e' = Ap h es' ∧ size es = size es'
                ∧ (ALL i < size es. es!i ⊑ es'!i)"
using assms
by (rule ordap.cases, auto)

lemma  ordapAp_converse:
   assumes "e' ⊑ Ap h es"
   shows "e' = perp ∨ (∃es'. e' = Ap h es' ∧ size es = size es'
                             ∧ (ALL i < size es. es'!i ⊑ es!i))"
using assms
by (rule ordap.cases, auto)
```

The inductive proof for Thm. `crwlPolarity` again considers each *CRWL* rule in turn. In the case for `B` we have `t = perp`, hence we just have to apply `ordapPerp` to get `t' = perp`, and then use the *CRWL* rule `B`.

```
case B thus ?case
    proof -
      from B and ordapPerp have "t' = perp" by simp
      thus ?thesis by auto
    qed
next
```

Regarding `RR`, as then `t = Var v`, by `ordapVar_converse` we get that either `t' = perp` or `t' = Var v`. The first case is trivial, and in the latter we just have to apply `ordapVar` getting `e' = Var v`, which is enough for Isabelle to finish the proof automatically.

```
  case (RR prog v e' t') thus ?case
    proof -
      from RR and ordapVar_converse have "t' = perp ∨ t' = (Var v)" by simp
     moreover
      {
       assume "t' = perp" hence ?thesis by auto
      }
     moreover
      {
       assume t'Var : "t' = (Var v)"
       from RR and ordapVar have "e' = (Var v)" by simp
       with t'Var have ?thesis by auto
      }
     ultimately show ?thesis by auto
    qed
next
```

The case of DC is more complicated. Again we obtain two cases for t' = 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.

```
  case (DC es ts prog c e' t') thus ?case
    proof -
      from DC and ordapAp_converse have "(t' = perp) ∨ (∃ ts' . (t' = Ap (cs c) ts') ∧
(size ts = size ts') ∧ (ALL i < (size ts). (ts' ! i) ⊑ (ts ! i)))" by simp
     moreover
      {
       assume "t' = perp" hence ?thesis by auto
      }
     moreover
      {
       assume t'ApC :"∃ ts' . (t' = Ap (cs c) ts') ∧ (size ts = size ts') ∧ (ALL i <
(size ts). (ts' ! i) ⊑ (ts ! i))"
       then obtain ts' where t'Shape : "(t' = Ap (cs c) ts') ∧ (size ts = size ts') ∧
(ALL i < (size ts). (ts' ! i) ⊑ (ts ! i))" ..
       from DC (3) and ordapAp have "∃ es' . (e' = Ap (cs c) es') ∧ (size es = size
es') ∧ (ALL i < (size es). (es ! i) ⊑ (es' ! i))" by simp
       then obtain es' where e'Shape : "(e' = Ap (cs c) es') ∧ (size es = size es') ∧
(ALL i < (size es). (es ! i) ⊑ (es' ! i))" ..
       with t'Shape and DC (1) have size's : "size es' = size ts'" by simp
       from size's and e'Shape and t'Shape and DC (2) have "ALL i < (size es). prog ⊢
(es' ! i) → (ts' ! i)" by auto
       with size's and e'Shape and t'Shape have ?thesis by auto
      }
     ultimately show ?thesis by auto
    qed
next
```

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.

```
  case (OR f ps r prog ϑ es t e' t') thus ?case
    proof -
      from OR (7) and ordapAp have "∃ es' . (e' = Ap (fs f) es') ∧ (size es = size
es') ∧ (ALL i < (size es). (es ! i) ⊑ (es' ! i))" by simp
      then obtain es' where e'Shape : "(e' = Ap (fs f) es') ∧ (size es = size es') ∧
(ALL i < (size es). (es ! i) ⊑ (es' ! i))" ..
      with OR (4) and ordapRefl have prems' : "ALL i < (size es). prog ⊢ (es' ! i) →
apSubst ϑ (ps ! i)" by simp
      from OR (3) and e'Shape have sizes' : "size es' = size ps" by simp
      from OR (6) OR (8) and ordapRefl have "prog ⊢ apSubst ϑ r → t'" by simp
      with OR (1) and OR (2) and sizes' and prems' and e'Shape show ?thesis by auto
    qed
qed
```

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.

```
theorem compCRWL :
   assumes "oneHole xC"
   shows "den P (apCon e xC) = (⋃t∈den P e. den P (apCon t xC))"
```

We have proved the two set inclusions separately as auxiliary lemmas.

```
lemma compCRWL1 :
  assumes "P ⊢ a → t" "a = apCon e xC" and "oneHole xC"
  shows "∃ s. P ⊢ e → s ∧ P ⊢ apCon s xC → t"
using assms unfolding oneHole_def
proof (induct arbitrary: e xC)

lemma compCRWL2 :
  assumes "P ⊢ a → t" "a = apCon s xC" "oneHole xC" and "P ⊢ e → s"
  shows "P ⊢ apCon e xC → t"
using assms unfolding oneHole_def
proof (induct arbitrary: s xC e)
```

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, as are the following versions of lemmas `compCRWL1` and `compCRWL2`, for the particular case of having a hole as context.

```
lemma compCRWLHole1 :
  assumes "P ⊢ apCon e xC → t" and "xC = Hole"
  shows "∃ t'. P  ⊢ e → t' ∧ P ⊢ apCon t' xC → t"
proof -
  from assms have derHole : "P ⊢ e → t" by auto
  with ctermRefl and ctermVals have "P ⊢ t → t" by simp
  with derHole and assms show ?thesis by auto
qed

lemma compCRWLHole2 :
  assumes "P ⊢ apCon s Hole → t" and "P ⊢ e → s"
  shows "P ⊢ apCon e Hole → t"
proof -
  from assms have derHole : "P ⊢ s → t" by auto
  from assms and ctermVals[of "P" "e" "s"] have "cterm s" by simp
  with derHole and lessCTermOrdap have "t ⊑ s" by simp
  with ordapRefl crwlPolarity[of "P" "e" "s" "e" "t"] and assms have "P ⊢ e → t" by
simp
  with assms show ?thesis by simp
qed
```

The only new result we have used in that proof is Lemma `lessCTermOrdap`, that relates reduction of c-terms with its relation in the approximation order.

```
lemma lessCTermOrdap :
  assumes "cterm t"
```

```
  shows "(P ⊢ t → s) = (s ⊑ t)"
using assms
by (auto simp add: lessCTermOrdapL lessCTermOrdapR)
```

To prove it we use the left-to-right of Lemma `lessCTermOrdapL` at the beginning of this section, and the right-to-left implication of Lemma `lessCTermOrdapR`, which can be easily proved by using the polarity of *CRWL*.

```
lemma lessCTermOrdapR :
  assumes "s ⊑ t" and "cterm t"
  shows "P ⊢ t → s"
using assms
proof -
  assume less : "s ⊑ t"
  from assms and ctermRefl have "P ⊢ t → t" by simp
  with less ordapRefl and crwlPolarity[of "P" "t" "t" "t" "s"] show ?thesis by simp
qed
```

Finally, with lemmas `compCRWLHole1` and `compCRWLHole2` at hand the proof for Thm. `compCRWL` is straightforward.

```
using assms
proof -
  have "den P (apCon e xC) ⊆ (⋃t∈den P e. den P (apCon t xC))"
  using assms
  proof -
    have "∀ t ∈ den P (apCon e xC). t ∈ (⋃s∈den P e. den P (apCon s xC))"
    proof -
     {
      fix t
      assume "t ∈ den P (apCon e xC)"
      hence "P ⊢ apCon e xC → t" unfolding den_def by simp
      with compCRWL1 have "∃ s. P  ⊢ e → s ∧ P ⊢ (apCon s xC) → t" using assms by
simp
      then obtain s where "P  ⊢ e → s ∧ P ⊢ (apCon s xC) → t" by auto
      hence "t ∈ (⋃s∈den P e. den P (apCon s xC))" unfolding den_def by auto
     }
      thus ?thesis by auto
    qed
    with subset_eq[of "den P (apCon e xC)" "⋃t∈den P e. den P (apCon t xC)"] show
?thesis by simp
  qed
 moreover
  have "den P (apCon e xC) ⊇ (⋃t∈den P e. den P (apCon t xC))"
  using assms
  proof -
    have "∀ t ∈ (⋃s∈den P e. den P (apCon s xC)). t ∈ den P (apCon e xC)"
    proof -
     {
      fix t
      assume  "t ∈ (⋃s∈den P e. den P (apCon s xC))"
      hence "∃ s ∈ den P e. (t ∈ den P (apCon s xC))" by simp
      then obtain s where "(s ∈ den P e) ∧ (t ∈ den P (apCon s xC))" by auto
```

```
      hence "P ⊢ e → s ∧ P ⊢ (apCon s xC) → t" unfolding den_def by simp
      with compCRWL2[of "P" _ "t" "s" "xC" "e"] have "P ⊢ (apCon e xC) → t" using assms
by simp
      hence "t ∈ den P (apCon e xC)" unfolding den_def by simp
     }
      thus ?thesis by auto
    qed
    with subset_eq[of "⋃t∈den P e. den P (apCon t xC)" "den P (apCon e xC)"] show
?thesis by simp
  qed
ultimately
  show ?thesis using subset_antisym[of "den P (apCon e xC)" "⋃t∈den P e. den P (apCon
t xC)"] by simp
qed
```

Again, while Thm. `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 it 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 $\mathcal{P} = \{f(X, X) \to a\}$. There is a *CRWL*-proof for the statement $\mathcal{P} \vdash f(a, b) \twoheadrightarrow a$ 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 independancy 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`, 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.

We will end this section with the following result about the replaceability of expressions in one-hole contexts, and easy consequence of the compositionality of *CRWL*.

```
theorem cntxtReplace : "(den P e1 = den P e2) =  (∀ xC. oneHole xC --> (den P (apCon e1
xC) = den P (apCon e2 xC)))"
proof -
  {
  have "(den P e1 = den P e2) --> (∀ xC. oneHole xC --> (den P (apCon e1 xC) = den P
(apCon e2 xC)))"
  by (auto simp add: compCRWL)
  }
 moreover
  {
  have "(∀ xC.  oneHole xC --> (den P (apCon e1 xC) = den P (apCon e2 xC))) --> (den P
e1 = den P e2)"
```

```
  proof
    assume h1 : "∀ xC.  oneHole xC --> (den P (apCon e1 xC) = den P (apCon e2 xC))"
    hence "den P (apCon e1 Hole) = den P (apCon e2 Hole)"
    proof -
      have "oneHole Hole" unfolding oneHole_def by simp
      with h1 show ?thesis by blast
    qed
    thus "den P e1 = den P e2" by simp
  qed
  }
 ultimately show ?thesis by auto
qed
```

5 Conclusions

This paper presented a formalization of the essentials of *CRWL* [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 it is apparent looking at the Isabelle code.

Our formalization is generic with respect to syntax, in the sense that a previously given signature and program is not assumed, and includes important auxiliary notions like substitutions or contexts. This is in contrast to previous work [4, 5] where some work about formalization of CRWL was reported, but focused on formalizing the semantics of each concrete program, as a way of proving concrete program properties. 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. Of course, such general properties hold for each concrete program, but nothing similar was achieved in the above mentioned previous works. 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 [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 [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 [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 [13, 12] of how the combination of a denotational and operational perspective is useful for general semantic reasoning in FLP.

References

1. E. Albert, M. Hanus, F. Huch, J. Oliver, and G. Vidal. Operational semantics for declarative multi-paradigm languages. *Journal of Symbolic Computation*, 40(1):795–829, 2005.
2. B. E. Aydemir, A. Bohannon, M. Fairbairn, J. N. Foster, B. C. Pierce, P. Sewell, D. Vytiniotis, G. Washburn, S. Weirich, and S. Zdancewic. Mechanized metatheory for the masses: The PoplMark challenge. In J. Hurd and T. F. Melham, editors, *TPHOLs*, volume 3603 of *Lecture Notes in Computer Science*, pages 50–65. Springer, 2005.
3. F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, United Kingdom, 1998.
4. J. Cleva, J. Leach, and F. López-Fraguas. A logic programming approach to the verification of functional-logic programs. In *Proc. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming, PPDP'04*, pages 9–19. ACM, 2004.

5. J. Cleva and I. Pita. Verification of CRWL programs with rewriting logic. *J. Universal Computer Science*, 12(11):1594–1617, 2006.
6. M. de Mol, M. C. J. D. van Eekelen, and M. J. Plasmeijer. Theorem proving for functional programmers. In T. Arts and M. Mohnen, editors, *Implementation of Functional Languages, 13th International Workshop, IFL 2002*, volume 2312 of *Lecture Notes in Computer Science*, pages 55–71. Springer, 2001.
7. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. A rewriting logic for declarative programming. In *Proc. European Symposium on Programming (ESOP'96)*, pages 156–172. Springer LNCS 1058, 1996.
8. J. C. González-Moreno, T. Hortalá-González, F. López-Fraguas, and M. Rodríguez-Artalejo. An approach to declarative programming based on a rewriting logic. *Journal of Logic Programming*, 40(1):47–87, 1999.
9. M. Hanus. Multi-paradigm declarative languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*, pages 45–75. Springer LNCS 4670, 2007.
10. M. Hanus (ed.). Curry: An integrated functional logic language (version 0.8.2). Available at *http://www.informatik.uni-kiel.de/~curry/report.html*, March 2006.
11. H. Hussmann. *Non-Determinism in Algebraic Specifications and Algebraic Programs*. Birkhäuser Verlag, 1993.
12. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. A simple rewrite notion for call-time choice semantics. In *Proc. Principles and Practice of Declarative Programming*, pages 197–208. ACM Press, 2007.
13. F. López-Fraguas, J. Rodríguez-Hortalá, and J. Sánchez-Hernández. Rewriting and call-time choice: the HO case. In *Proc. 9th International Symposium on Functional and Logic Programming (FLOPS'08)*, volume 4989 of *LNCS*, pages 147–162. Springer, 2008.
14. F. López-Fraguas and J. Sánchez-Hernández. $\mathcal{TOY}$: A multiparadigm declarative system. In *Proc. Rewriting Techniques and Applications (RTA'99)*, pages 244–247. Springer LNCS 1631, 1999.
15. L. C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer Verlag, Berlin, Heidelberg, 1994. See also the Isabelle home page at `http://isabelle.in.tum.de/`.
16. R. F. Stärk. The theoretical foundations of lptp (a logic program theorem prover). *J. Log. Program.*, 36(3):241–269, 1998.