Redex2Coq: towards a theory of decidability of Redex's reduction semantics

Mallku Soldevila
FAMAF, UNC and CONICET (Argentina)

Rodrigo Ribeiro
DECOM, UFOP (Brazil)

Beta Ziliani
FAMAF, UNC and Manas.Tech (Argentina)

Abstract

We propose the first step in the development of a tool to automate the translation of Redex models into a semantically equivalent model in Coq, and to provide tactics to help in the certification of fundamental properties of such models.

The work is based on a model of Redex's semantics developed by Klein *et al.* In this iteration, we were able to code in Coq a primitive recursive definition of the matching algorithm of Redex, and prove its correctness with respect to the original specification. The main challenge was to find the right generalization of the original algorithm (and its specification), and to find the proper well-founded relation to prove its termination.

Additionally, we also adequate some parts of our mechanization to prepare it for the future inclusion of Redex features absent in Klein *et al.*, such as the Kleene's closure operator.

2012 ACM Subject Classification Theory of computation → Formal languages and automata theory; Theory of computation → Rewrite systems; Theory of computation → Interactive proof systems

Keywords and phrases Coq, PLT Redex, Reduction semantics

Digital Object Identifier 10.4230/LIPIcs...

Funding *Mallku Soldevila*: CONICET (Argentina)

1 Introduction

Redex [7] is a DSL built on top of the Racket programming language, which allows for the mechanization of reduction semantics models and formal systems. It includes a variety of tools for testing the models, including: unit testing; random testing of properties; and a stepper for step-by-step reduction sequences. Given its tookit, Redex has been successfully used for the mechanization of large semantics models of real programming languages (*e.g.*, JavaScript [9, 16]; Python [17]; Scheme [13]; and Lua [21, 20, 19]).

The approach of Redex to semantics engineering involves a lightweight development of models that focuses on a quick transition between specification of models and testing of their properties. These virtues of Redex enable it as a useful tool with which to perform the first steps of a formalization effort. Nonetheless, when a given model seems to be thoroughly tested and mature, one still might need to prove its desired properties, since no amount of testing can guarantee the absence of errors [4].

Redex does not offer tools for formal verification of a given model, and there are no automatic tools to export the model into some proof assistant. Hence, for verification purposes, a given model must be written again entirely into a proof assistant. Besides being a time-consuming process, another downside is that the translation into the proof assistant may be guided just by an intuitive understanding of the behavior of the mechanization in Redex. Intuitive understanding that could differ from the actual behavior of the model in Redex. This is so, since the tool implements a particular meaning of reduction semantics with evaluation contexts, offering an expressive language to the user that includes several features, useful to express concepts like context-dependent syntactic rules. The actual semantics of this language may not coincide with what the researcher understands [5].

We propose to build a tool to automatically translate a given model in Redex into an equivalent model in Coq. The interpretation of the resulting model is done through a shallow embedding in Coq of Redex's actual semantics. In that regard, we note that there exist already several implementations of some of the concepts of reduction semantics with evaluation contexts (see §5). However, they are not specific to Redex's semantics, and therefore miss some crucial concepts, such as its support for evaluation contexts and its Kleene's closure operator.

Summary of the contributions.

In this work we present a first step into the development of a tool to automate the translation of a Redex model into a semantically equivalent model in Coq, and to provide automation to the proof of essential properties of such models. The present work is heavily based on RedexK, the model of Redex's semantics developed by [5]. In summary:

- We mechanize RedexK in Coq. In the process, we develop a proof of termination for the matching algorithm, which enables its mechanization into Coq as a regular primitive recursion.
- We modify RedexK to prepare it for the future addition of features, like the Kleene's closure operator, and the development of tactics to decide about properties of reduction semantics models.
- We prove soundness properties of the matching algorithm with respect to its specification.
- We prove the correspondence of our algorithm with respect to the original proposal present in RedexK.

The reader is invited to download the accompanying source code from

`https://github.com/Mallku2/redex2coq`

The remainder of this paper is structured as follows: §2 presents a brief introduction to reduction semantics, as presented in Redex; §3 offers a general overview of our mechanization in Coq; §4 presents the main soundness results proved within our mechanization; §5 discuss about related work from the literature of the area; finally, §6 summarizes the results presented in this paper and discusses future venues of research enabled by this first iteration of our tool.

2 Redex

In this section, we present a brief introduction to Redex's main concepts, limiting our attention to the concepts that are relevant to our tool in this first iteration of the development. As a running example, we show how to mechanize in Redex a fragment of λ-calculus with normal order call-by-value reduction. For a better introduction to these topics, the reader can consult [7, 10] and the original paper on which our mechanization is based [5].

Redex can be viewed as a particular implementation of Reduction Semantics with Evaluation Contexts (RS), in which semantical aspects of computations are described as relations over syntactic elements (terms) of the language.

As a simple introductory example, Figure 1 shows part of a specification for a call-by-value λ-calculus. The grammar of the language is defined with the first command,

```
(define-language lambda
[e ::= x (e e) v] [v ::= (λ x e)] [x ::= variable-not-otherwise-mentioned] [E ::= hole (E e) (v E)])
(reduction-relation lambda #:domain e
 [--> (in-hole E ((λ x e) v)) (in-hole E (substitute e x v)) beta_contraction])
(define-metafunction lambda
  fv : e -> (x ...)
  [(fv x) (x)]
  [(fv (e_1 e_2)) (x_1 ... x_2 ...) (where (x_1 ...) (fv e_1)) (where (x_2 ...) (fv e_2))]
```

Figure 1 Definition of a language in Redex.

`define−language`. The language called `lambda` contains non-terminals `e` (representing any λ-term), `v` (values; in this case only λ-abstractions), `x` (variables; defined with pattern `variable−not−otherwise−mentioned`, meaning the symbols that are not used as literals elsewhere in the language) and `E` (evaluation contexts, to be explained below). The right-hand side of the productions of each non-terminal are shown to the right of the ::= symbol.

The productions of non-terminal `E` indicate that an evaluation context could be a single hole, or a context of the form E' `e`, where E' is another evaluation context; or a context of the form v E'. Note that the consequence of this definition is that we are imposing normal-order reduction.

The reduction relation is defined with the keyword `reduction−relation`. It defines a relation between terms (`e`), from the previously defined `lambda` language, consisting of a single contraction, `beta_contraction`. This rule explains two things: how β-contractions are done; and the order in which those contractions can occur, effectively imposing the order of evaluation. The rule states that if a term can be decomposed into context `E` and an abstraction application ((λ `x` `e`) `v`) (pattern (`in−hole` `E` ((λ `x` `e`) `v`))), then, the original term reduces to the phrase resulting from plugging the result of substituting `x` by `v` in `e` into the context `E` (pattern `in−hole` `E` (`substitute` `e` `x` `v`)).

As an example, consider the term ((λ `w` `w`) (λ `y` `y`)) (λ `z` `z`). In order to match the left-hand side of the rule, it decomposes the term into context `E` = `hole` (λ `z` `z`), matching `x` with `w`, `e` with `w`, and `v` with (λ `y` `y`). The result is the term (λ `y` `y`) (λ `z` `z`).

We won't delve into the details of the `substitute` meta-function, but it will be useful to explain one of its components: the list of *free variables* of a term, `fv`, partially shown in Figure 1. This meta-function is defined using the `define−metafunction` keyword. The signature of the function, `fv :  e` → (`x` ...) , states that `fv` receives a λ-term, and returns a list of 0 or more variables (pattern `x` ... , to be explained below). After the signature, we have 2 equations explaining which are the free variables: in a term that is a single variable `x` or an application e_1 e_2. For reasons of space, we do not show equations referring to the cases where the term under consideration is a λ-abstraction.

The pattern `p` ... is called the *Kleene's closure* of a pattern, and expresses the idea of "zero or more terms" that match a given pattern `p`. For example, the first `where` clause of the second equation imposes a condition that holds only when the expression `fv` e_1 matches the pattern x_1 ..., meaning that `fv` e_1 must evaluate to a list of 0 or more variables. Redex bind that list with x_1 ..., and we can use this pattern to refer to this list. In particular, in this case we return x_1 ... followed by the variables resulting from evaluating `fv` e_2 (that is, x_2 ...). As a last comment, it is possible to express context-dependent restrictions by using specific indexes: for example, pattern (`x_1` `x_1`) only matches a list of two equal variables; and pattern (`x_!_` `x_!_`) only matches a list of two different variables.

```
Inductive term := lit_term : lit → term
                | list_term_c : list_term → term
                | contxt_term : contxt → term
  with list_term := nil_term_c : list_term | cons_term_c : term → list_term → list_term
  with contxt := hole_contxt_c : contxt | list_contxt_c : list_contxt → contxt
  with list_contxt := hd_contxt : contxt → list_term → list_contxt
                    | tail_contxt : term → list_contxt → list_contxt.
```

Figure 2 Language of terms.

3 Expressing Redex in Coq

In this section, we introduce the main ideas behind our implementation in Coq. Later, in §4, we describe the main soundness properties that we mechanized.

To introduce the simpler parts of the mechanization, we will show listings of our source code together with some natural language explanation. The more complex portions of the mechanization (like the matching/decomposition algorithm), will be described more abstractly.

3.1 Language of terms and patterns

We begin the presentation by introducing our mechanized version of the language of terms and patterns. We ask for some reasonable decidability properties about the language that we use to describe a given reduction semantics model. These standard properties will be useful to develop our mechanization in its present version, and more so in the prospective future of the development.

3.1.1 Symbols

The module type `Symbols` abstracts the atomic elements of the language of terms and patterns: literals (`lit`), non-terminals (`nonterm`), and *pattern variables* (`var`, also sub-indexes used in the patterns). We require that these types are also instances of the `stdpp`'s typeclass `EqDecision` [23]. Details can be found in file `patterns_terms.v`.

3.1.2 Terms

In RedexK, terms are classified according to their structure, or if they act as a context or not. According to their structure, terms are classified as atomic literals or with a binary-tree structure. In our case, we will generalize the notion of "terms with structure". One of the most prominent features absent in RedexK is the Kleene's closure operator. In order to be able to include this feature in a future iteration of our model, we begin by generalizing the notion of structured terms. We will allow them to be lists of 0 or more terms. Non-empty lists can also be considered as binary trees, but where the right sub-tree of a given node is always a list. We will enforce that shape through types.

The language of terms is presented in Figure 2. A term consisting of a literal is built with constructor `lit_term`, while structured terms are captured and enforced through a type, `list_term`. Structured terms can be an empty list, built with `nil_term_c`, or a list with one term as its head, and some list as its tail, using constructor `cons_term_c`. Finally, we define an injection into terms, `list_term_c`.

```
Inductive pat := lit_pat : lit → pat | hole_pat : pat
| list_pat_c : list_pat → pat | name_pat : var → pat → pat
| nt_pat : nonterm → pat | inhole_pat : pat → pat → pat
with list_pat := nil_pat_c : list_pat | cons_pat_c : pat → list_pat → list_pat.
```

Figure 3 Language of patterns.

The other kind of terms considered in RedexK are contexts. Contexts include information about where to find the hole, to help the algorithms of decomposition and plugging. That information consists in a path from the root of the term (seen as a tree) to the leaf that contains the hole. To that end, RedexK defines a notion of context that, if it is not just a single hole, it contains a *tag* indicating where to look for the hole: either into the left or the right sub-tree of the context. We preserve the same idea, adapted to our presentation of structured terms.

We introduce the type `contxt`, to represent and enforce through types the notion of contexts. These contexts can be just a single hole (`hole_contxt_c`) or a list of terms with some position marked with a hole. In order to guarantee the presence of a hole into this last kind of contexts, we introduce the type `list_contxt`. These contexts can point into the first position of a given list (`hd_contxt`) or the tail (`tail_contxt`). Finally, we have the injections from `list_contxt` into `contxt` (`list_contxt_c`), and from `contxt` into `term` (`contxt_term`). These injections, naturally, are used later as coercions.

3.1.3 Patterns

As mentioned in §2, Redex offers a language of patterns with enough expressive power to state context-dependent restrictions. We mechanize the same language of patterns as presented in RedexK, with the required change to accommodate our generalization done to structured terms, as explained in the previous sub-section. The language of patterns is presented in Figure 3.

Pattern `lit_pat` l matches only a single literal l. Pattern `hole_pat` matches a context that is just a single hole. In order to describe the new category of structured terms that we presented in the previous subsection, we add a new category of patterns enforced through type `list_pat`. From this category of patterns, pattern `nil_pat_c` matches a list of 0 terms, while pattern `cons_pat_c` p_{hd} p_{tl} matches a list of terms, whose first term matches pattern p_{hd}, and whose tail matches the pattern p_{tl}. Finally, we have a injection from this category of patterns into the type `pat`: `list_pat_c`.

Context-dependent restrictions are imposed through pattern `name_pat` x p. This pattern matches a term t that, in turn, must match pattern p. As a result, the pattern `name_pat` x p introduces a context-dependent restriction in the form of a *binding*, that assigns *pattern variable* x to term t. Data-structures to keep track of this information will be introduced later, but for the moment, just consider that during matching some structures are used to keep track of all of this context-dependent restrictions that have the form of a binding between a pattern variable and a term. If, at the moment of introducing the binding to x, there exist another binding for the same variable but with respect to a term different than t, the whole matching fails.

Pattern `nt_pat` e matches a term t, if there exist a production from non-terminal e, whose right-hand-side is a pattern p that matches term t.

Finally, pattern `inhole_pat` p_c p_h matches some term t, if t can be decomposed between some context C, that matches pattern p_c, and some term t', that matches pattern p_h. It should be possible to plug t' into context C, recovering the original term t. Note that the information contained in the tag of each kind of non-empty context, that indicates where to find the hole, helps in this process: at each step the process looks, either, into the head of the context or into its tail.

3.1.4 Decidability of predicates about terms and patterns

We want to put particular emphasis on the development of tools to recognize the decidability of predicates about terms and patterns. This could serve as a good foundation for the future development of tactics to help the user automate as much as possible the process of proving arbitrary statements about the user's reduction semantics models.

As a natural consequence of our first assumptions about the atomic elements of the languages of terms and patterns, presented in §3.1.1, we can also prove decidability results about definitional equalities among terms and patterns. Another straightforward consequence involves the decidability of definitional equalities between values of the many data-structures involved in the process of matching. Future efforts will be put in developing further this minimal theory about decidability (see §6).

3.1.5 Grammars

The notion of grammar in Redex, as presented in §2, is modeled in RedexK as a finite mapping between non-terminals and sets of patterns. Our intention is not to force some particular representation for grammars, beyond the previous description. As a first step, we axiomatize some assumptions about grammars through a module type. We begin by defining a production of the grammar, simply, as a pair inhabiting `nonterm * pat`, and we define a `productions` type as a list of type production. We also ask for the existence of computational type `grammar`, a constructor for grammars (inhabiting `productions` → `grammar`), the possibility of testing *membership* of a production with respect to a grammar, and to be possible to *remove* a production from a grammar (`remove_prod`). We ask for some notion of *length* of grammars, and that `remove_prod` actually affects that length in the expected way. This will be useful to guarantee the termination property of the matching algorithm (see §3.2.1). Finally, we ask for some reasonable decidability properties for these types and operations: decidability of definitional equalities among values of the previous types, and, naturally, for the testing of membership of a production with respect to a given grammar.

Abstracting these previous types and properties in a module type (`Grammar`), could serve in the future when developing further our theory of decidability for the notion of RS implemented in Redex. As a simple example, separating the type `productions` from the actual definition of the type `grammar`, allows for the encapsulation of properties in the type `grammar` itself, that specifies something about the inhabitants of `productions`. Some decidability results depend on a grammar whose productions are restricted in some particular way.[1]

For this first iteration, we provide an instantiation of the previous module type with a grammar implemented using a list of productions. Here, the type `grammar` does not impose new properties over the type `productions`. We also provide a minimal theory to reason

[1] For example, while the general language intersection problem for context-free grammars (CFG) is non-decidable, the intersection problem between a regular CFG and a non-recursive CFG is decidable [14].

about *grammars as lists*, that helps in proving the required termination and soundness properties of the matching algorithm. This is required since our previous axiomatization of grammars, through module type `Grammar`, is not strong enough to prove every desired property of our algorithm. A goal for a next iteration would be to take advantage of the experience with this development, and strengthen our axiomatization of grammars.

3.2 Matching and decomposition

The first challenge that we encounter when trying to mechanize RedexK, was finding a primitive recursive algorithm to express matching and decomposition. The original algorithm from RedexK is not a primitive recursion, for reasons that will be clear below. However, the theory developed in the paper, to check the soundness of this algorithm and to characterize the inputs over which it actually converges to a result, helped us to recapture the matching and decomposition process as a *well-founded recursion*.

3.2.1 Well-founded relation over the domain of matching/decomposition

In Coq, a well-founded recursion is presented as a primitive recursion over the evidence of *accessibility* of a given element (from the domain of the well-founded recursion), with respect to a given *well-founded relation* R. That is, it is a primitive recursion over the proof of a statement that asserts that, from a given actual parameter x over which we are evaluating a function call, there is only a finite quantity of elements which are *smaller* than x, according to relation R. These smaller elements are the ones over which the recursive calls can be evaluated. In other words: R does not contain infinite decreasing chains, and, hence, the number of recursive calls is always finite. Such relation R is called well-founded.

The actual steps of matching/decomposition will be presented in detail below. But, for the moment, in pursuing a well-founded recursive definition for the matching/decomposition process, let us observe that, for a given grammar G, pattern p and term t, the matching/decomposition of t with p involves, either:

1. Steps where the input term t is *decomposed* or *consumed*.
2. Steps where there is no input consumption, but, either:
   a. The pattern p is decomposed or consumed.
   b. The productions of the grammar G are considered, searching for a suitable pattern that allows matching to proceed.

Step 1 corresponds, for example, to the case where t is a list of terms of the form `cons_term_c` t_{hd} t_{tl}, and p is a list of patterns of the form `cons_pat_c` p_{hd} p_{tl}. Here, the root of each tree (t and p) match, and the next step involves checking if t_{hd} matches pattern p_{hd}, and if t_{tl} matches p_{tl}.

Step 2a corresponds, for example, to the case where pattern p has the form `name_pat` x p': as described in §3.1.3, the next step in matching/decomposition involves checking if pattern p' matches t. Here, the step does not involve consumption of input term t, but it does involve a recursive call to matching/decomposition over a proper sub-pattern of p.

Finally, step 2b corresponds to the case of pattern `nt_pat` n, which implies looking for productions of n in G that match t. Here, there is no reduction of terms and this process does not neccesarily imply the reduction of patterns.

If not because for the pattern `nt_pat`, it could be easily argued that the process previously described is indeed an algorithm. Now, if we do take into account `nt_pat` patterns, termination in the general case does no longer holds. In particular, non-termination could be observed with a *left-recursive* grammar G and a given non-terminal n that witnesses the left-recursion of G. Matching pattern `nt_pat` n, following the described process, could get stuck repeating the step of searching into the productions of n, without any consumption of input: from pattern `nt_pat` n we could reach to the same pattern `nt_pat` n.

Indeed, the described matching algorithm does not deal with left-recursion, as is argued in [5]. There, the property of left-recursion is captured by providing a relation $\rightarrow_G$ that order patterns as they appear during the previously described phase of the matching process, when the input term is not being consumed, but there is decomposition of a pattern and/or searching into the grammar, looking for a proper production to continue the matching. Then, a left-recursive grammar would be one that makes the chains of the previous relation to contain a repeated pattern. In particular, during matching, we could begin with a pattern `nt_pat` n and reach the same pattern without consuming input, repeating this process over and over again.

Then, if, for a non left-recursive grammar G and non-terminal n from G, it is the case that $p \not\rightarrow_G^+ p$ for any pattern p (where $\rightarrow_G^+$ is the transitive closure of $\rightarrow_G$), it must be the case that also `nt_pat` $n \not\rightarrow_G^+$ `nt_pat` n. This means that, when searching for productions of n in G, and as long as the matching/decomposition is in the stage captured by $\rightarrow_G$, (*i.e.*, no consumption of input), it should be possible to *discard* the productions from the grammar G being tested.

The previous observation helps us argue that, provided that G is non left-recursive, when the matching process enters the stage of non-consumption of input, this phase will eventually finalize: either, the pattern under consideration is totally decomposed and/or we run out of productions from G. In what follows, we will assume *only* non-left-recursive grammars. This does not impose a limitation over our model of Redex, since it only allows such kind of grammars.

We will exploit the previous observations to build a well-founded relation over the domain of our matching/decomposition function. The technique that we will use will consist in, first, modeling each phase in isolation through a particular relation. There will be a relation $<_{\mathtt{t}}$: `term` $\rightarrow$ `term` $\rightarrow$ `Prop` explaining what happens to the input when it is being consumed, and a relation $<_{\mathtt{p\times g}}$: `pat` $\times$ `grammar` $\rightarrow$ `pat` $\times$ `grammar` $\rightarrow$ `Prop`, explaining what happens to the pattern and the grammar when there is no consumption of input. We will also prove the well-foundedness of each relation. The final well-founded relation for the matching/decomposition function will be the *lexicographic product* of the previous relations, a well-known method to build new well-founded relations out of other such relations [15]. We will parameterize this relation by the original grammar, to be able to recover the original productions when needed (see §3.2.4 for details). For a given grammar g, we will denote this last relation with $<^{g}_{\mathtt{t\times p\times g}}$. Note that its type will be `term` $\times$ `pat` $\times$ `grammar` $\rightarrow$ `term` $\times$ `pat` $\times$ `grammar` $\rightarrow$ `Prop`.

For a tuple (t, p, G) to be related with another *smaller* tuple (t', p', G'), according to $<^{g}_{\mathtt{t\times p\times g}}$, it must happen that $t' <_{\mathtt{t}} t \vee (t' = t \wedge (p', G') <_{\mathtt{p\times g}} (p, G))$. This expresses the situations where there is actual progress in the matching/decomposition algorithm towards a result: either there is consumption of input or the phase of production searching and decomposition of the pattern progresses towards its completion. Note that this definition shows that the lexicographic product is a more general relation, that contains chains of tuples that do not necessarily model what happens during matching and decomposition: if $t' <_{\mathtt{t}} t$, then $(t', p', G') <^{g}_{\mathtt{t\times p\times g}} (t, p, G)$, for some grammar g, regardless of what (p', G') and (p, G) actually are. Later, when presenting the relations that form this lexicographic product, we will also specify which are the actual chains that we will consider when modeling the process of matching and decomposition. We will refer to these last kind of chains as the *chains of*

$$(p_c, G) <_{\mathtt{p\times g}} (\mathtt{inhole\_pat}\ \ p_c\ \ p_h, G) \qquad (p_h, G) <_{\mathtt{p\times g}} (\mathtt{inhole\_pat}\ \ p_c\ \ p_h, G)$$

$$(p, G) <_{\mathtt{p\times g}} (\mathtt{name\_pat}\ \ x\ \ p, G) \qquad \frac{p \in G(n) \qquad G' = G \setminus (n, p)}{(p, G') <_{\mathtt{p\times g}} (\mathtt{nt\_pat}\ \ n, G)}$$

Figure 4 Consumption of pattern and productions.

interest.

In particular, this means that we will define a more general relation, that is simpler to define and to work with, but that still retains the desired properties: it will be well-founded and will contain the chains of interest, besides other meaningless chains.

3.2.2 Input consumption

We define the relation $<_{\mathtt{t}}$ to be exactly $<_{\mathsf{subt}}$, where $<_{\mathsf{subt}}$ will denote the relation `subterm_rel : term → term → Prop`, that links a term with each of its sub-terms. This describes an order that coincides with that in which the input is consumed, for the actual specification of matching and decomposition. This does not avoid for more exotic patterns, that could be introduced in the future, to have a different behavior on input consumption. Hence, the distinction between what constitutes a relation like $<_{\mathtt{t}}$ and what simply is $<_{\mathsf{subt}}$.

3.2.3 Pattern and production consumption

The specification of $<_{\mathtt{p\times g}}$, shown in Figure 4, matches the cases 2a and 2b described in §3.2.1. Recall that, in this case, the algorithm entered a phase where the pattern is being decomposed or productions from some non-terminal are being tested, to see if matching/decomposition can continue.

Matching a term t with a pattern of the form `inhole_pat` p_c p_h, means trying to decompose the term between some context that matches pattern p_c, and some sub-term of t that matches pattern p_h. In doing so, the first step involves a decomposition process (to be specified later in §3.2.5), that begins working over the whole term t, and with respect to just the sub-pattern p_c. Hence, this step does not involve input consumption, but it does involve considering a reduced pattern: p_c. We just capture this simple fact through $<_{\mathtt{p\times g}}$, by stating that $(p_c, G) <_{\mathtt{p\times g}} (\mathtt{inhole\_pat}\ \ p_c\ \ p_h, G)$ holds, for any grammar G. Note that we preserve the grammar.

In the particular case that p_c matches `hole_contxt_c`, then there is no actual decomposition of the term t. This means that, when looking for said sub-term of t that matches pattern p_h, we will still being considering the whole input term t. Again, we just capture this simple fact by stating that $(p_h, G) <_{\mathtt{p\times g}} (\mathtt{inhole\_pat}\ \ p_c\ \ p_h, G)$ holds, for any grammar G.

The case for the pattern `name_pat` x p can be explained on the same basis as with the previous cases.

Finally, the last case refers to the pattern `nt_pat` n: it involves considering each production of non-terminal n in G. Here it is assumed that G contains the correct set of productions that remain to be tested (an invariant property about G through our algorithm). Then, we continue the process considering a grammar G' that contains every production from G, except for (n, p): the already considered production of non-terminal n with right-hand-side p. We denote it stating that G' equals the expression $G \setminus (n, p)$.

$$\frac{p \in G'(n) \qquad G \vdash t : p_{G' \setminus (n,p)} \mid b}{G \vdash t : (\texttt{nt\_pat}\ \ n)_{G'} \mid \oslash}$$

$$\frac{G \vdash t_{hd} : (p_{hd})_G \mid b_{hd} \qquad G \vdash t_{tl} : (p_{tl})_G \mid b_{tl}}{G \vdash \texttt{cons\_term\_c}\ \ t_{hd}\ \ t_{tl} : (\texttt{cons\_pat\_c}\ \ p_{hd}\ \ p_{tl})_{G'} \mid b_{hd} \sqcup b_{tl}}$$

$$\frac{G \vdash t = C[\![t_h]\!] : (p_c)_{G'} \mid b_c \qquad t_h \ <_{\mathsf{subt}}\ t \qquad G \vdash t_h : (p_h)_G \mid b_h}{G \vdash t : (\texttt{inhole\_pat}\ \ p_c\ \ p_h)_{G'} \mid b_c \sqcup b_h}$$

Figure 5 Generalized specification of matching.

3.2.4 Specification of matching

We now explain our specification for matching and decomposition, which is a slight generalization from that of RedexK [5]. In the original specification, the judgment about matching has the form $G \vdash t : p \mid b$, stating that pattern t matches pattern p, under the productions from grammar G, producing the bindings b (which could be an empty set of bindings, denoted with $\oslash$). A seemingly obvious fact is that the non-terminals that may appear on pattern p will be interpreted in terms of the productions from G. In our presentation, we relax this assumption, and allow the non-terminals to be interpreted in terms of some arbitrary grammar G', which in practice will be a subset of G.

Therefore, our judgment is of the form $G \vdash t : p_{G'} \mid b$, with the particular difference that, *initially*, we interpret the non-terminals from p with grammar G'. Only when input consumption begins, we restore the original grammar G. Figure 5 presents a simplified fragment of our formal system. Following a top-down order, the first rule applies when a term t matches a pattern `nt_pat` n, when the non-terminals of this pattern (in this case, just n) are *initially* interpreted in terms of the productions of G': then, that matching is successful if there exist some $p \in G'(n)$, such that t matches p, when its non-terminals are *initially* interpreted under the productions from the grammar $G' \setminus (n, p)$. Recall that this means that this last grammar will be used as long as there is no input consumption, or there is no other appearance of a pattern `nt_pat`. Again, we are following the chains from $<_{\mathtt{p\times g}}$. Also, the non-left-recursivity of the grammars being considered guarantee that this replacement of the grammars is semantics-preserving: we will not need another production from n, as long as there is no input consumption. Finally, note that this match does not produce bindings.

The second rule can be understood in terms of the previously introduced concepts. Note that, for each recursive proof of matching over sub-terms and sub-patterns, we *re-install* the original grammar G. We denote with $\sqcup$ the union of bindings, which is undefined if the same name is bound to different terms.

The last case in Figure 5 refers to the matching of a term t with a pattern of the form `inhole_pat` p_c p_h. This operation is successful when we can decompose term t between some context that matches pattern p_c, and some sub-term, that matches pattern p_h. In order to fully formalize what this matching means, we need to explain what *decomposition* means. RedexK specifies this notion through another formal system, whose adaptation to our work we present in the following sub-section. The original system allows us to build proofs for judgments of the form $G \vdash t = C[\![t']\!] : p \mid b$, meaning that we can decompose term t,

$$\frac{G \vdash t_{hd} = C[\![t'_{hd}]\!] : (p_{hd})_G \mid b_{hd} \qquad G \vdash t_{tl} : (p_{tl})_G \mid b_{tl}}{G \vdash \texttt{cons\_term\_c}\ \ t_{hd}\ \ t_{tl} = (\texttt{hd\_contxt}\ \ C\ \ t_{tl})[\![t'_{hd}]\!] : (\texttt{cons\_pat\_c}\ \ p_{hd}\ \ p_{tl})_{G'} \mid b_{hd} \sqcup b_{tl}}$$

$$\frac{G \vdash t = C_c[\![t_c]\!] : (p_c)_{G'} \mid b_c \qquad t_c \ <_{\mathsf{subt}}\ t \qquad G \vdash t_c = C_h[\![t_h]\!] : (p_h)_G \mid b_h}{G \vdash t = (C_c ++ C_h)[\![t_h]\!] : (\texttt{inhole\_pat}\ \ p_c\ \ p_h)_{G'} \mid b_c \sqcup b_h}$$

Figure 6 Generalized specification of decomposition.

between some context C, that matches pattern p, and some sub-term t'. The decomposition produces bindings b, and the non-terminals from pattern p are interpreted through the productions present in grammar G. In our case, we modify this judgment by generalizing it in the same way done for the matching judgment: $G \vdash t = C[\![t']\!] : p_{G'} \mid b$, including the possible interpretation of non-terminals in p, initially, using grammar G'.

Returning to the case about `inhole_pat` patterns in Figure 5, note that our intention is to distinguish the case where the decomposition step actually consumes some portion from t (shown in the rule), from the case where it does not (not shown in Figure 5). The first situation (described in the rule for `inhole_pat`) means that context C is not simply a hole, and t_h is an actual proper sub-term of t: *i.e.*, $t_h \ <_{\mathsf{subt}}\ t$. Also, note that the decomposition is proved interpreting (initially) the non-terminals from p_c with production from the arbitrary grammar G' ($(p_c)_{G'}$). And the proof of the matching between t_h and p_h is done interpreting the non-terminals of this last pattern with productions from the original grammar G ($(p_c)_G$). On the contrary, when the decomposition step does not consume some input (pattern p_c matches against a hole, and the resulting term t_h is exactly t), the proof of the matching between t_h and p_h is done considering the arbitrary grammar G'.

3.2.5 Specification of decomposition

The final part of the specification concerns the decomposition judgment required for the `inhole_pat` pattern. We already mentioned what it does and how it is generalized; we proceed to explain the relevant rules listed in Figure 6.

The first rule explains the decomposition of a list of terms `cons_term_c` t_{hd} t_{tl}, between a context that matches a list of patterns `cons_pat_c` p_{hd} p_{tl}, and some sub-term. In the particular case of the first rule, the hole of the resulting context is pointing to somewhere in the head of the list of terms. This information is indicated by the constructor of the resulting context: `hd_contxt` C t_{tl}, where C is some context that must match pattern p_{hd}, as indicated in the premise of the inference rule. Note that the whole premise is stating that the decomposition occurs in the head of the list of terms (t_{hd}), and the resulting sub-term is t'_{hd}. Then, the side-condition from the inference rule states that the tail of the original input term, t_{tl}, must match the tail of the list of patterns p_{tl}. Finally, note that in the decomposition through sub-pattern p_{hd}, and the matching sub-pattern p_{tl}, the non-terminals of these patterns are interpreted in terms of productions from the original grammar, G.

With respect to the remaining rule, the case of the `inhole_pat` pattern, it handles the matching of pattern `inhole_pat (inhole_pat` p_c p_h`)` $p_{h'}$ with some term t. The semantics of this case involves a first step of decomposition of t between some context that matches sub-pattern `inhole_pat` p_c p_h, and some sub-term that matches sub-pattern $p_{h'}$.

```
Definition binding := var * term.
Inductive decom_ev : term → Set :=
  | empty_d_ev : forall (t : term), decom_ev t
  | nonempty_d_ev : forall t (c : contxt) subt,
      {subt = t ∧ c = hole_contxt_c} + {subterm_rel subt t} → decom_ev t.
Inductive mtch_ev : term → Set :=
  mtch_pair : forall t, decom_ev t → list binding → mtch_ev t.
```

Figure 7 Mechanization of decomposition and matching results.

In the rule shown in Figure 6, we are describing what it means, in this situations, that first step of decomposing t in terms of a context that matches pattern `inhole_pat` p_c p_h. Since the whole pattern must match some context, it means that, both, p_c and p_h, are patterns describing contexts. Note that we distinguish the case where p_c produces an empty context, from the case where it does not (not shown in Figure 6). This distinction allows us to recognize whether we should interpret non-terminals from patterns through the original grammar G or the arbitrary grammar G'.

The last piece of complexity of the rule for the `inhole_pat` pattern resides in the actual context that results from the decomposition. Here, the authors of RedexK, expressed this context as the result of plugging one of the obtained contexts within the other, denoted with the expression $C_c ++ C_h$: this represents the context obtained by plugging context C_h within the hole of context C_c, following the information contained in the constructor of this last context to find its actual hole. For reasons of space we elude this definition, though it presents no surprises.

3.2.6 Matching and decomposition algorithm

We close this section presenting a simplified description of the matching and decomposition algorithm adapted for its mechanization in Coq. We remind the reader that this algorithm is just a modification of the one proposed for RedexK [5].

The previous specification of the algorithm cannot be used directly to derive an actual effective procedure to compute matching and decomposition. In particular, the rules for decomposition of lists of terms (second and third rules from Figure 6) do not suggest effective meanings to determine whether to decompose on the head, and match on the tail, or vice versa. To solve this issue, and the complexity problem that could arise from trying to naively perform both kind of decomposition simultaneously, the algorithm developed for RedexK performs matching and decomposition simultaneously, sharing intermediate results.

Supporting data-structures.

In Figure 7 we show some of the implemented data-structures used to represent the results returned by RedexK's algorithm. The result of a matching/decomposition of a term t (with some given pattern) will be represented through a value of type `mtch_ev` t. Making the type dependent on t is done for soundness checking.

For reasons of brevity, when presenting the algorithm we will avoid the actual concrete syntax from our mechanization. A value of type `mtch_ev` t will be denoted as (d, b), where d is a value of type `decom_ev` t (explained below), and b is a list of bindings (also shown in Figure 7). For a value of the list type `mtch_powset_ev` t, we will denote it decorating it with its dependence on the value t: $[(d, b), ...]_t$

$$
\begin{aligned}
\mathsf{M}_{\mathsf{ev\_gen}}(g_1,(t,p,g_2),M_{ap}) = [&(d,b) \mid d \in \mathsf{select}(t_{hd}, d_{hd}, t_{tl}, d_{tl}, t, sub),\\
& sub : \mathsf{subterms}\ t\ t_{hd}\ t_{tl}, \quad b = b_{hd} \sqcup b_{tl},\\
& (d_{hd}, b_{hd})_{t_{hd}} \in M_{ap}(tp_{hd}, lt_{hd}), \quad (d_{tl}, b_{tl})_{t_{tl}} \in M_{ap}(tp_{tl}, lt_{tl}),\\
& lt_{hd} : tp_{hd} <^{g_1}_{\mathtt{t\times p\times g}} tp_{cons}, \quad lt_{tl} : tp_{tl} <^{g_1}_{\mathtt{t\times p\times g}} tp_{cons},\\
& tp_{cons} = (t,p,g_2), \quad tp_{hd} = (t_{hd}, p_{hd}, g_1), \quad tp_{tl} = (t_{tl}, p_{tl}, g_1)]_t\\
& \mathit{with}\ t = \mathbf{cons}\ t_{hd}\ t_{tl} \quad p = \mathbf{cons}\ p_{hd}\ p_{tl}\\
\mathsf{M}_{\mathsf{ev\_gen}}(g_1,(t,p,g_2),M_{ap}) = [&(d,b) \mid d = \mathsf{combine}\ (t, C, t_c, ev, d_h),\\
& b = b_c \sqcup b_h, \quad (d_h, b_h)_{t_c} \in M_{ap}(tp_h,\ lt_h),\\
& lt_h : tp_h <^{g_1}_{\mathtt{t\times p\times g}} tp_{inhole}, \quad tp_h = (t_c, p_h, g_h),\\
& g_h \text{ according to Figure 5},\\
& ((C, t_c)^{ev}_t, b_c)_t \in M_{ap}(tp_c, lt_c), \quad lt_c : tp_c <^{g_1}_{\mathtt{t\times p\times g}} tp_{inhole},\\
& tp_{inhole} = (t,p,g_2), \quad tp_c = (t, p_c, g_2)]_t\\
& \mathit{with}\ p = \textbf{in-hole}\ p_c\ p_h
\end{aligned}
$$

Figure 8 Generator function for the matching and decomposition algorithm.

Values inhabiting type `decom_ev` t represent a decomposition of a given term t, between a context and a sub-term. We include in the value some evidence of soundness of the decomposition: a sub-term $subt$ extracted in the decomposition is either t itself, or a proper sub-term of t .

Since a value of type `mtch_ev` t could represent a single match or a single decomposition, following [5] we distinguish an actual match using an empty decomposition `empty_d_ev` t. Otherwise, a decomposition is represented through the value `nonempty_d_ev` t C $subt$ ev, for context C, sub-term $subt$ and soundness evidence ev. We denote such values as $(C, subt)^{ev}_t$.

Matching and decomposition algorithm as a least-fixed-point.

We capture the intended matching/decomposition algorithm as the least fixed-point of a *generator function* or *functional* of the following type:

```
forall (g1 : grammar) (tpg1 : (term * pat * grammar)),
  (forall tpg2 : (term * pat * grammar),
    matching_tuple_order g1 tpg2 tpg1 → list (mtch_ev (fst tpg2)))
→ list (mtch_ev (fst tpg1))
```

The family of generator functions $\mathsf{M}_{\mathsf{ev\_gen}}$ of this type is parameterized over grammars and tuples of terms and patterns. Also, these functions receive a candidate of matching/decomposition that they will *improve*: they will construct the result by optionally calling the candidate over tuples that are provably smaller that the given tuple `tpg1`, according to the well-founded order (`matching_tuple_order g1 tpg2 tpg1`, see §3.2.1). Hence, $\mathsf{M}_{\mathsf{ev\_gen}}$ will build a function that performs the matching indicated in `tpg1`, using, if necessary, a candidate function that is able to perform matching for tuples smaller than `tpg1`.

Figure 8 shows 2 of the equations that capture $\mathsf{M}_{\mathsf{ev\_gen}}$. For reasons of space, we describe terms and patterns avoiding the more verbose concrete syntax of our mechanization. The first equation explains the matching and/or decomposition of a list of terms (**cons** t_{hd} t_{tl}) with a list of patterns (**cons** p_{hd} p_{tl}). We describe by comprehension the list of results. Note that, to explain this case, we need to consider the approximation function M_{ap} that $\mathsf{M}_{\mathsf{ev\_gen}}$ receives as its last parameter. We begin by using M_{ap} to compute matching and decomposition for *smaller* tuples: $tp_{hd} = (t_{hd}, p_{hd}, g_1)$ and $tp_{tl} = (t_{tl}, p_{tl}, g_1)$. Note that, given that these tuples represent a matching/decomposition over a proper sub-term of the input term, we consider the original grammar g_1 (first parameter of $\mathsf{M}_{\mathsf{ev\_gen}}$). In order to be able to fully evaluate M_{ap}, we need to build proofs lt_{hd} and lt_{tl} of type $tp_{hd} <^{g_1}_{\mathtt{t\times p\times g}} tp_{cons}$ and $tp_{tl} <^{g_1}_{\mathtt{t\times p\times g}} tp_{cons}$, respectively, where tp_{cons} is the original tuple over which we evaluate $\mathsf{M}_{\mathsf{ev\_gen}}$. Then, for each value of type `mtch_ev` t_{hd} and `mtch_ev` t_{tl} of the results obtained from evaluating M_{ap}, the algorithm queries if they are decompositions or not, and if it is possible to combine these results, using the helper function select.

The original select helper function from RedexK receives as parameters t_{hd}, d_{hd}, t_{tl} and d_{tl}. It analyses d_{hd} and d_{tl}: if none of them represent actual decompositions, then the whole operation will be considered just a matching of the original list of terms and select must build an *empty* decomposition of the proper type to represent this. If only d_{hd} is a decomposition, then the whole operation is interpreted as a decomposition of the original list of terms on the head of the list. In that case, select builds a value of type `decom_ev` (**cons** t_{hd} t_{tl}).

The remaining equation, that of the **in-hole** pattern, can be understood on the same basis as the previous one, requiring only some explanation the auxiliary function combine: it helps in deciding if the result is a decomposition against pattern **in-hole**, or if it is just a match against said pattern, depending on whether d_h is a decomposition or not.

Finally, we define the desired matching/decomposition algorithm, M_{ev}, as the least fixed-point of the previous generator function. For reasons of space we do not show its definition, but it presents no surprises. The resulting implementation can be seen on file ./`match_impl.v`.

3.3 Semantics for context-sensitive reduction rules

The last component of RedexK consists in a semantics for context-sensitive reduction rules, with which we define semantics relations in Redex. The proposed semantics makes use of the introduced notion of matching, to define a new formal system that explains what it means for a given term to be *reduced*, following a given semantics rule. We have mechanized the previous formal system, though, for reasons of space, we do not introduce it here in detail. The reader is invited to look at the mechanization of this formal system, in module ./reduction.v.

3.4 Extra material

In the `README.md` file of the repository the interested reader will find the correspondence between the source code and this paper. Additionally, besides from the results shown here, we included a mechanization of a lambda-calculus with normal-order reduction similar to the one presented in §2. It serves mainly to showcase the actual capabilities of Redex that are mechanized in the present version of the tool, and how to invoke them to implement a reduction-semantics model. We note that the performance of the matching/decomposition algorithm is subpar, an issue we plan to tackle in a future iteration of the tool.

4 Soundness and completeness of matching

```
Theorem completeness_M_ev : ∀ G1 G2 p t sub_t b C,
 (G1 |− t : p, G2 | b   → In (mtch_pair t (empty_d_ev t) b) (M_ev G1 (t, (p, G2))))
 ∧
 (G1 |− t1 = C [ t2 ] : p , G2 | b → ∃ (ev_decom : {sub_t = t} + {subterm_rel sub_t t}),
    In (mtch_pair t (nonempty_d_ev t C sub_t ev_decom) b) (M_ev G1 (t, (p, G2)))).

Theorem from_orig : ∀ G t p b,
   non_left_recursive_grammar →
   G |− t : p | b → G |− t : p, G | b
  with from_orig_decomp : ∀ G C t1 t2 p b,
   non_left_recursive_grammar →
   G |− t1 = C [ t2 ] : p | b → G |− t1 = C [ t2 ] : p , G | b.
```

Figure 9 The statement of completeness of M_{ev} and completeness of our formal systems, in Coq.

In the original paper of RedexK they prove the correspondence between the algorithm and its specification. In our mechanization we reproduced this result, for the least-fixed-point of $\mathsf{M}_{\mathsf{ev\_gen}}\ g\ (t,\ p,\ g')$ and our extended definition of matching (§3.2.4). In what follows, $\mathsf{M}_{\mathsf{ev}}\ g\ (t,\ p,\ g')$ represents the least-fixed-point of $\mathsf{M}_{\mathsf{ev\_gen}}\ g\ (t,\ p,\ g')$. Naturally, for a given grammar g, the original intention of matching and decomposition corresponds to $\mathsf{M}_{\mathsf{ev}}\ g\ (t,\ p,\ g)$. We show the statement of completeness of the algorithm in Figure 9. Note that we represent and manipulate results returned from M_{ev} through Coq's standard library implementation of lists. Also, the shape of the tuples of terms, patterns and grammars, is the result of the way in which we build our lexicographic product: the product between a relation with domain `term`, and a relation with domain `pat` × `grammar`. Completeness can be proved by *rule induction* on the evidences of match and decomposition.

The converse, the soundness property, is not shown, but it is the expected converse of the completeness statement. The proof present no surprises: since we have a well-founded recursion over the tuples from `term` × `pat` × `grammar`, we also have an induction principle to reason over them.

We also verified the correspondence between our specifications and the original formal systems from the paper. We can't do it for the general case: we followed the proposal of the authors of RedexK, explained in §3.2, and only consider those grammars that are *non-left recursive*. In Coq, we name this predicate `non_left_recursive_grammar` (see file `wf_rel.v`).

We show in Figure 9 the completeness result mapping our formal systems with the original ones from RedexK. Note the hypothesis `non_left_recursive_grammar`, and how we consider the same grammar `G` for interpreting the non-terminals.

For the converse, soundness, we need to restrict the result to those grammars `G'` over which we begin interpreting the non-terminals to be *smaller or equal* (`gleq`) to the original one `G` (see ./`verification/match_spec_equiv.v`).

5 Related work

CoLoR [3] is a mechanization in Coq of the theory of well-founded rewriting relations over the set of first-order terms, applied to the automatic verification of termination certificates. It presents a formalization of several fundamental concepts of rewriting theory, and the mechanization of several results and techniques used by termination provers. Its notion of terms includes first-order terms with symbols of fixed and varyadic arity, strings, and simply typed lambda terms. CoLoR does not implement a notion of a language of patterns offering support for context-sensitive restrictions, something that is ubiquitous in a Redex mechanization, to define semantics rules, formal systems and meta-functions over the terms of a given language. Also, Redex is not focused on well-founded rewritting relations, but, rather, in arbitrary relations over terms of a language. In order to use CoLoR to *explain* Redex, it would require a considerable amount of work, extending and/or modifying CoLoR, to be able to encode the semantics of Redex's language of patterns.

Sieczkowski et. al present in [18] an implementation in Coq of the technique of *refocusing*, with which it is possible to extract abstract machines from a specification of a reduction semantics. The derivation method is proved correct, in Coq, and the final product is a generic framework that can be used to obtain interpreters (in terms of abstract machines), from a given reduction semantics that satisfies certain characteristics. In order to characterize a reduction semantics that can be *automatically refocused* (*i.e.*, transformed into a traditional abstract machine), the authors provide an axiomatization capturing the sufficient conditions. Hence, the focus is put in allowing the representation of certain class of reduction semantics (in particular, deterministic models for which refocusing is possible), rather than allowing for the mechanization of arbitrary models (even non-deterministic semantics), as is the case with Redex. Nonetheless, future development of our tool could take advantage of this library, since testing of Redex's models that are proved to be deterministic could make use of an optimization as refocusing, to extract interpreters that run efficiently, in comparison with the expensive computation model of reduction semantics.

Matching logic is a formalism used to specify logical systems and their properties. It is mechanized in Coq in [2], including its syntax, semantics, formal system and the corresponding soundness result. At its heart, matching logic has a notion of patterns and pattern matching. Redex could be explained as a matching logic, with formulas that represent Redex's patterns to capture languages and relations, and whose model refer to the terms (or structures containing terms) that match against these patterns. While this representation of Redex could be of interest for the purpose of studying the underlying semantics of Redex, this is not satisfactory for the purpose of providing the user with a direct explanation in Coq of their mechanization in Redex.

6 Conclusion

We adapted RedexK [5] to be able to mechanize it into Coq. In particular, we obtained a primitive recursive expression of its matching algorithm; we introduced modifications to its language of terms and patterns, to better adapt it to the future inclusion of features of Redex absent in RedexK; we reproduced the soundness results shown in [5], but adapted to our mechanization, while also verifying the expected correspondence between our adapted formal systems, that capture matching and decomposition, and the originals from the cited work.

A natural next step in our development could consist in the addition of automatic routines to transpile a Redex model into an equivalent model in Coq. Also, extending the language with capabilities of Redex absent in RedexK would be of vital importance to allow our tool to be of practical use. Our proposed modification for the language of patterns and terms, already implemented in this first iteration, enables us to easily include the Kleene's closure operator. This could be a reasonable next step in increasing the set of Redex's features captured by our mechanization.

References

1. Hendrik Pieter Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co., New York, N.Y., 1981.
2. Péter Bereczky, Xiaohong Chen, Dániel Horpácsi, Lucas Peña, and Jan Tušil. Mechanizing matching logic in coq. In Vlad Rusu, editor, Proceedings of the Sixth *Working Formal Methods Symposium,* "Al. I. Cuza University", Iasi, Romania, 19-20 September, 2022, volume 369 of *Electronic Proceedings in Theoretical Computer Science*, pages 17–36. Open Publishing Association, 2022. `doi:10.4204/EPTCS.369.2`.
3. Frédéric Blanqui and Adam Koprowski. Color: a coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science*, 21(4):827–859, 2011. `doi:10.1017/S0960129511000120`.
4. Brown PLT Group. Mechanized lambdajs. `https://blog.brownplt.org/2012/06/04/lambdajs-coq.html`, 2012.
5. Steven Jaconette Casey Klein, Jay McCarthy and Robert Bruce Findler. A semantics for context-sensitive reduction semantics. In *APLAS'11*, 2011.
6. Adam Chlipala. *Certified Programming with Dependent Types.* The MIT Press, 2019.
7. M. Felleisen, R. B. Finlder, and M. Flatt. *Semantics Engineering with PLT Redex.* The MIT Press, 2009.
8. M. Felleisen and R. Hieb. The revised report on the syntactic theories of sequential control and state. *TCS*, 103, 1992.
9. A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *ECOOP '10*, 2010.
10. Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: On the effectiveness of lightweight mechanization. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 285–296, New York, NY, USA, 2012. ACM. URL: `http://doi.acm.org/10.1145/2103656.2103691`, `doi:10.1145/2103656.2103691`.
11. P. J. Landin. Correspondence between algol 60 and church's lambda-notation: part i. *Communications of the ACM*, 8:89–101, 1965.
12. Peter J. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6:308–320, 1964.
13. Jacob Matthews and Robert Bruce Findler. An operational semantics for scheme. *Journal of Functional Programming*, 2007.
14. Mark-Jan Nederhof and Giorgio Satta. The language intersection problem for non-recursive context-free grammars. *Inf. Comput.*, 192(2):172–184, August 2004. `doi:10.1016/j.ic.2004.03.004`.
15. Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *J. Symb. Comput.*, 2(4):325–355, dec 1986.
16. J. G. Politz, M. J. Carroll, B. S. Lerner, J. Pombrio, and S. Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In *DLS '12*, 2012.
17. J. G. Politz, A. Martinez, M. Milano, S. Warren, D. Patterson, J. Li, A. Chitipothu, and S. Krishnamurthi. Python: The full monty: A tested semantics for the Python programming language. In *OOPSLA '13*, 2013.
18. Filip Sieczkowski, Małlgorzata Biernacka, and Dariusz Biernacki. Automating derivations of abstract machines from reduction semantics: A generic formalization of refocusing in coq. In *Proceedings of the 22nd International Conference on Implementation and Application of Functional Languages*, IFL'10, page 72–88, Berlin, Heidelberg, 2010. Springer-Verlag.
19. M. Soldevila, B. Ziliani, and B. Silvestre. From specification to testing: Semantics engineering for lua 5.2. *Journal of Automated Reasoning*, 2022.
20. Mallku Soldevila, Beta Ziliani, and Daniel Fridlender. Understanding Lua's garbage collection: Towards a formalized static analyzer. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming*, PPDP 2020, 2020.

21 Mallku Soldevila, Beta Ziliani, Bruno Silvestre, Daniel Fridlender, and Fabio Mascarenhas. Decoding Lua: Formal semantics for the developer and the semanticist. In *Proceedings of the 13th ACM SIGPLAN Dynamic Languages Symposium*, DLS 2017, 2017.

22 Joseph E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, Cambridge, MA, USA, 1977.

23 The Coq-std++ Team. An extended "standard library" for Coq, 2020. Available online at `https://gitlab.mpi-sws.org/iris/stdpp`.