Abstract We propose the first steps in the development of a tool to automate the translation of Redex models into a (hopefully) semantically equivalent model in Coq, and to provide tactics to help in the certification of fundamental properties of such models. The work is heavily based on a model of Redex's semantics developed by Klein et al. By means of a simple generalization of the matching problem in Redex, we obtain an algorithm suitable for its mechanization in Coq, for which we prove its soundness properties and its correspondence with the original solution proposed by Klein et al. In the process, we also adequate some parts of our mechanization to better prepare it for the future inclusion of Redex features absent in the present model, like its Kleene-star operator. Finally, we discuss future avenues of development that are enabled by this work.

Keywords Coq · PLT Redex · Reduction semantics

Redex → Coq: towards a theory of decidability of Redex's reduction semantics

Mallku Soldevila · Rodrigo Ribeiro · Beta Ziliani

the date of receipt and acceptance should be inserted later

1 Introduction

Motivation Redex (Felleisen et al, 2009) is a DSL, built on top of the Racket programming language, that allows for the mechanization of reduction semantics models and formal systems. It also includes a variety of tools for testing them: unit testing, random generators of terms for random testing of properties, stepper for step-by-step reduction sequences. It has been successfully used for the mechanization of large semantics models of real programming languages (*e.g.*, JavaScript Guha et al (2010); Politz et al (2012); Python Politz et al (2013); Scheme Matthews and Findler (2007); and Lua 5.2 Soldevila et al (2017, 2020, 2022)); the development of tools for program analysis (like, again, Soldevila et al (2020), to check for a particular kind of *well-behavedness* of Lua 5.2 programs; Lorenzen and Erdweg (2013), for checking type-soundness of syntactic language extensions that introduces high-level programming concepts). Other, particular uses cases, involve the mechanization of operational semantics for virtual machines specialised for running reactive programs Oeyen et al (2022), or even mechanizing a model of Redex itself, as is done in Felleisen et al (2009).

M. Soldevila
FAMAF, UNC and CONICET (Argentina)
E-mail: mes0107@famaf.unc.edu.ar
ORCID: 0000-0002-8653-8084

B. Ziliani
FAMAF, UNC and Manas.Tech (Argentina)
E-mail: beta@mpi-sws.org
ORCID: 0000-0001-7071-6010

R. Ribeiro
DECOM, UFOP (Brazil)
E-mail: rodrigo.ribeiro@ufop.edu.br
ORCID: 0000-0003-0131-5154

Redex's approach to semantics engineering involves a philosophy about documents that specify semantics models, which can be summarized as "semantic models as software artifacts" (Klein et al, 2012), and 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, it could be of use to actually prove its desired properties, since no amount of testing can guarantee the absence of errors.

As an example, consider the experience with Guha et al (2010), a major step into the development of semantics models for JavaScript, dubbed λ_{JS}. It has been reported that, even after the mechanisation of their semantics model with Redex, and after intensive testing of an interpreter derived from the mechanization against major implementations of JavaScript, other researchers found a missing case in the semantics.[1] The missing case caused certain terms to get *stuck*, breaking a progress property claimed for the model. This called for a revision of the model, in search for any other flaw, but equipped with a proof assistant. To this end, the researchers mechanized λ_{JS} entirely into Coq.

At the moment there is no other way to tackle such task: the 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 (see Casey Klein and Findler (2011) for a development of this issue).

Our proposal, to assist in mitigating the described situation, consists in helping the user with the automatic translation of a given model in Redex, into an equivalent model in Coq. The interpretation, of the resulting model in Coq, will be done through a shallow embedding in this proof assistant of Redex's actual semantics. In that regard, we note that there already exist several implementations of some of the concepts of reduction semantics with evaluation contexts (see §5 for a detailed description of the available options). However, some features of Redex, like its support for evaluation contexts and its semantics for a Kleene's closure of patterns, are particular to the tool. To gain trust about the correspondence between the original model in Redex and its transpiled version into Coq, it may be preferable to have a direct explanation of this last model in terms of Redex's own behavior, avoiding codifying Redex's concepts on top of another model of reduction semantics.

[1] See `https://blog.brownplt.org/2012/06/04/lambdajs-coq.html`

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 the model of Redex's semantics developed by Klein et al. (Casey Klein and Findler, 2011) (from now on, RedexK). 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 Redex's Kleene closure of patterns 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 `github.com/Mallku2/redex-into-coq`.

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 will 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 a fragment of λ-calculus with normal order reduction, in Redex. For a better introduction to these topics, the reader can consult Felleisen et al (2009); Klein et al (2012) and the original paper on which our mechanization is based (Casey Klein and Findler, 2011). Also, its reference manual presents the most up-to-date information about Redex's features.

Redex can be viewed as a particular implementation of the semantics of reduction semantics with evaluation contexts (RS). Reduction semantics (Felleisen et al, 2009) follows the intellectual tradition of providing a theory about the concepts that are expressed by a given language, just in terms of relations over terms of said language. This tradition is embodied in theories like λ-calculus, proposed by Alonzo Church (Barendregt, 1981) as a way to explain and study functions, in terms of rewriting relations. While its capabilities to express computations were already known (Barendregt, 1981), it was rediscovered as a way

```
(define-language lambda
  [e ::= x (e e) v]

  [v ::= (λ x e)]

  [x ::= variable-not-otherwise-mentioned]

  [E ::= hole (E e) (v E)])
```

Fig. 1: Definition of a language in Redex.

to formally describe programming languages later, by researchers like Peter J. Landin (*e.g.*, see Landin (1964, 1965)). These syntactic theories ended up being useful to *explain* several different phenomena and mechanisms present in programming languages, in a concise and abstract way: *e.g.*, evaluation strategies, parameter passing style in function-calls, complex control-flow features and state (Felleisen and Hieb, 1992).[2]

In the context of semantics for programming languages, terms represent actual programs (and, maybe, some semantics elements), and the relations over programs can represent dynamic and static semantics relations. In order to define these structures that contain terms (languages and relations), the user of Redex uses a language of *patterns*. These patterns constitute a highly expressive language, whose semantics is explained specifying which terms match against a given pattern. This formalization of the notion of matching against Redex's patterns is the main focus of Casey Klein and Findler (2011), together with the development of an algorithmic interpretation of this specification. Mechanize this work in Coq, solving problems like finding a primitive recursive implementation of the matching process, constitutes the main work presented in this paper.

As a simple introductory example, consider Figure 1, where it is shown the definition of a grammar that captures terms of a call-by-value λ-calculus, where we impose normal-order evaluation, using *evaluation contexts*. The grammar contains non-terminals `e` (representing any λ term), `v` (representing values; in this case, only λ-abstractions), `x` (representing variables) and `E` (representing evaluation contexts, to be explained below). The right-hand-side of the productions of each non-terminal are shown on the right of the ::= symbol.

Productions of non-terminals `e` and `x` are standard. In the case of non-terminal `x`, the right-hand-side of its only production is defined with a pattern (`variable-not-otherwise-mentioned`) that has a context-sensitive meaning: the terms that *match against* non-terminal `x` (*i.e.*, the terms that can be produced by `x`), are only those that do not match against the remaining non-terminals (*i.e.*, the terms that cannot be produced by the remaining non-terminals).

[2] Note the emphasis put in the word "explain": not every researcher would concur with the idea that a relation over terms is actually explaining said terms (for example, see Stoy (1977), page 9).

```
(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))]

  [(fv (λ x_1 e)) (x_2 ... x_3 ...)

   (where (x_2 ... x_1 x_3 ...) (fv e))]

  ;{x not in (fv e)}
  [(fv (λ x e)) (fv e)])
```

Fig. 2: Definition of a meta-function in Redex: *free variable* in λ terms.

Within the toolbox of RS, the syntactic notion of *contexts* is a useful device to express context-sensitive rules and concise definitions. A *context* is a term with a special position (or positions) denoted with a marker, a *hole*. RS then offers ways to refer to these contexts, to reason over them and to manipulate them, through the operations of *decomposition* of a given term into a context and another sub-term, and *plugging* a given term into the hole of a given context. Decomposition is referred through a special pattern, that expresses the way into which a given term must be decomposed. Plugging is its dual concept, and it is denoted in a similar way, typically being the position where it occurs on a given definition what distinguishes it from a decomposition.

Context themselves may represent the literal context where a given term appears within a program. *Evaluation contexts* are a special category of contexts, used typically in programming languages' semantics, that point into a single position within a program, indicating where we should look for the next redex during a reduction. If we are interested in a deterministic dynamic semantics, we could use evaluation contexts to impose a particular reduction (or evaluation) order: for a well-defined notion of evaluation contexts, it should be possible to decompose every program into a unique evaluation context and a sub-term, that should be a redex (according to the given dynamic semantics).

Returning to Figure 1, 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, for a given λ term of the form $e_1\ e_2$, we will evaluate it in a normal-order fashion: that is, the next redex should be looked into e_2 only if e_1 is already a value; otherwise, we start looking for the redex within e_1.

We can also define a notion of *free variable* in λ terms with a *meta-function* `fv`, whose equations are listed in Figure 2. Note that we can define the (run-time checked) signature of the function, `fv : e` $\rightarrow$ `(x ...)` , which explains 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 4 equations explaining which are the free variable in: a term that is a single variable `x`; an application $\texttt{e}_1\ \texttt{e}_2$; a λ abstraction whose formal parameter ($\texttt{x}_1$) occurs free in its body (`e`); and a λ abstraction whose formal parameter (`x`) does not occur free in its body (`e`). Note that the second and third equation contain side-conditions, in the form of a clause `where`: their semantics dictate that these conditions hold if the expression to the right, matches against the pattern on the left. For example, the first `where` clause of the second equation holds if the expression ($\texttt{fv}\ \texttt{e}_1$) (an evaluation of `fv` over term $\texttt{e}_1$), matches against the pattern ($\texttt{x}_1$ ...), to be explained below.

The definition of `fv` shows a feature of Redex which is particular to it: the *Kleene closure* of a given pattern, which serves to express the idea of "zero or more terms" that match against a given pattern. It is denoted as a pattern followed by ... (*i.e.*, a mathematical ellipsis). In the previous figure, it was used to define the domain of `fv` (a list of "0 or more variables", with pattern (`x ...`)), and in the second and third equation, within the `where` clauses and when expressing the final value of `fv`. For example, as mentioned previously, the first `where` clause of the second equation imposes a condition that holds only when the expression $\texttt{fv}\ \texttt{e}_1$ matches against the pattern $\texttt{x}_1$ ..., meaning that $\texttt{fv}\ \texttt{e}_1$ must evaluate to a list of 0 or more variables. Redex bind that list with $\texttt{x}_1$ ..., and we can use this pattern whenever we want to refer to this list. In particular, the value of `fv` over the abstraction of this second equation, means that we return the variables to which $\texttt{fv}\ \texttt{e}_1$ evaluated ($\texttt{x}_1$ ...), followed by the variables to which $\texttt{fv}\ \texttt{e}_2$ evaluated ($\texttt{x}_2$ ...): that is, $\texttt{x}_1$ ... $\texttt{x}_2$ .... Note that, in the `where` clause of the fourth equation, we are asking for `fv e` to match against the pattern $\texttt{x}_2$ ... $\texttt{x}_1\ \texttt{x}_3$ ..., where $\texttt{x}_1$ is the formal parameter and `e` is the body of the λ abstraction whose free variables we want to obtain. This means that we are forcing the situation where the formal parameter appears in the list of free variables of `e`. In other words, the third equation refers to the case where the formal parameter of the λ abstraction appears free in its body.

The interesting aspect of the previous language of patterns is that it allows us to enforce context-dependent restrictions, through many devices. For example, by repeating sub-patterns, within a given pattern, the user can enforce the repetition of elements into a given list of terms or any other part of a phrase. For example, the pattern (`x_1 x_1`) only matches against a list of 2 equal variables. Also, we can force some sub-terms to be different from the rest, by using the suffix `_!_` after each pattern whose matching term we want to distinguish from the rest. For example, the pattern (`x_!_ x_!_`) only matches against a list of 2 different variables. These devices, to enforce context-dependent rules, can be exploited to define languages, but also any relations over their terms.

Finally, Figure 3 depicts the definition of the compatible closure (with respect to evaluation contexts `E`) of a call-by-value β-contraction, in Redex (note the keyword `reduction—relation`). The single reduction rule shown explain 2 things: how β-contractions are done, using a generic capture-avoiding sub-

```
(define reduction
  (reduction-relation
   lambda
   #:domain e

   [--> (in-hole E ((λ x e) v))
        (in-hole E (substitute e x v))
        beta_reduction]))
```

Fig. 3: Definition of a semantics relation in Redex.

stitution function (`substitute`); and the order in which those contraction can occur, effectively imposing the order of evaluation described by contexts `E`. The rule states that, if a given term can be decomposed between some context `E` and some abstraction application ((λ `x` `e`) `v`) (condition expressed through the pattern (`in-hole` `E` ((λ `x` `e`) `v`))), then, the original term reduces to the phrase resulting from plugging (`substitute` `e` `x` `v`) (*i.e.*, capture-avoiding substitution of the formal parameter `x`, by the value `v`, into the abstractions' body `e`) into the context `E` (which is expressed through the pattern `in-hole` `E` (`substitute` `e` `x` `v`)). Finally, the resulting relation will be the *least* relation that satisfies the given conditions. That is, these definitions can be translated as the usual Coq's inductive relations.

For reasons of space, and to keep our example simple, we eluded the definition of the capture-avoiding substitution function. This can be defined as our previous specification of function `fv` (plus some escaping to Racket code). However, Redex itself provides a general mechanism to get a substitution function *by free*, requiring from us only to indicate the bounding occurrences of variables in the constructions of our language, and their scope. This feature is not included in RedexK, and neither is it considered in our mechanization.

The previous brief introduction to Redex served the purpose of introducing some features with which we will be dealing when working with RedexK. We avoid features that are not covered in said model. Also, not every capability previously described is covered in RedexK, though we need to mention them in order to easily implement our model of λ-calculus: we are talking about the Kleene closure of patterns, used when defining meta-function `fv` (Figure 2); and the pattern `variable-not-otherwise-mentioned`, used to define λ variables (Figure 1).

3 Expressing Redex in Coq.

In this section, we introduce the main ideas behind our implementation in Coq. Later, in §4, we will provide a specification of the obtained algorithm, proofs asserting the correspondence between the algorithm and its specification, and between our specification and the one provided in the original work.

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. In that way, while being faithful to our mechanization, we will avoid the expected complexities of an actual implementation with a dependently-typed language like Coq.

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, but also in the prospective future of the development.

3.1.1 Symbols

We require for the elements of the language of terms and patterns (literals, non-terminals and sub-indexes used in the patterns) to be equipped with a decidable definitional equality. To formalize these properties we take advantage of `stdpp`'s (The Coq-std++ Team, 2020) typeclass `EqDecision`. We abstract all of these assumptions into the module type `Symbols`, shown in Figure 4.

In order to implement an instantiation of a module of type `Symbols`, we ask the user for the type of literals, *pattern variables* (or the sub-indexes of patterns mentioned in §2) and non-terminals of the grammar: computational types `lit`, `var` and `nonterm`, respectively. We also ask for proofs showing that these types are also instances of typeclass `EqDecision`. Naturally, we do not want to burden the user with these proofs. A requirement for our future transpiler from Redex to Coq should be that it must be able to automatically build these proofs, something that is feasible within Coq.

3.1.2 Terms

In the original paper, 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 Redex's Kleene closure of patterns. Such patterns match against (or describe) lists of 0 or more terms. 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.

```
Module Type Symbols.
  (* literals for both, pats and terms *)
  Parameter lit: Set.
  (* names in name_pat *)
  Parameter var: Set.
  (* representation of the non-terms of a given grammar, for patterns nt *)
  Parameter nonterm: Set.

  (* some assumptions to ease the reasoning about decidability *)
  Parameter nonterm_eq_dec : EqDecision nonterm.
  Parameter var_eq_dec : EqDecision var.
  Parameter lit_eq_dec : EqDecision lit.
End Symbols.
```

Fig. 4: A module type capturing assumptions about several atomic elements of the grammar.

```
Inductive term : Set :=
  | lit_term    : lit → term
  | list_term_c : list_term → term
  | contxt_term : contxt → term

  with list_term : Set :=
  | nil_term_c  : list_term
  | cons_term_c : term → list_term → list_term

  with contxt : Set :=
  | hole_contxt_c : contxt
  | list_contxt_c : list_contxt → contxt

  (* hd_contxt and tail_contxt point into a position of a list of terms *)
  with list_contxt : Set :=
  | hd_contxt   : contxt → list_term → list_contxt
  | tail_contxt : term → list_contxt → list_contxt.
```

Fig. 5: Language of terms.

The language of terms is presented in 5. Terms as literals are 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, a some list as its tail, using constructor `cons_term_c`. Finally, we define an injection into terms, `list_term_c`.

The other kind of terms considered in RedexK are contexts. Being a context involves not only the existence of a hole marking some position into a term (as mentioned in §2). It also involves including information describing where to find that marked position, in the context itself, 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, the authors of RedexK defined 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

```
Inductive pat : Set :=
| 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 : Set :=
| nil_pat_c : list_pat
| cons_pat_c : pat → list_pat → list_pat.
```

Fig. 6: Language of patterns.

or the right sub-tree of the context. We preserve the same idea, adapted to our presentation of structured terms: now, a hole could mark the head or the tail of a list, and we add to the contexts' tags indicating that information.

We introduce the type `contxt`, to represent and enforce through types the notion of contexts. These contexts can be just a single hole, built with constructor `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, constructed with `hd_contxt`, or the tail, constructed with `tail_contxt`. Finally, we have 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 even 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 6.

Pattern `lit_pat` l matches only against a single literal l. Pattern `hole_pat` matches against 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 against a list of 0 terms, while pattern `cons_pat_c` p_{hd} p_{tl} matches against a list of terms, whose first term matches against pattern p_{hd}, and whose tail matches against 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 against a term t that, in turn, must match against 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. This could happen if, for example, pattern `name_pat` x p is just a sub-pattern from another pattern, and there is already a sub-pattern of the form `name_pat` x p', where p' already matched against a term different than t.

With the language of patterns we can describe the grammar of our language, as well as specify the many kinds of relations over terms of our language. For example, the following pattern (taken from the grammar of the λ-calculus shown in Figure 1):

```
cons_pat_c (nt_pat e) (cons_pat_c (nt_pat e) nil_pat_c)
```
[3]

would represent the right-hand-side of the production that indicates that a λ term applied to another λ term, is a valid term. As seen, patterns themselves can contain mentions to non-terminals of our grammar: pattern `nt_pat` e matches against a term t, if there exist a production from non-terminal e, whose right-hand-side is a pattern p that matches against term t.

Finally, pattern `inhole_pat` p_c p_h matches against some term t, if t can be decomposed between some context C, that matches against pattern p_c, and some term t', that matches against pattern p_t. 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] For simplicity, we avoid mentioning the injection of this value into type `pat`, through `list_pat_c`.

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. 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 (`new_grammar` : `productions` $\rightarrow$ `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`).[4] 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.[5]

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 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.

3.1.6 Remaining data-structures

We end this sub-section with a brief description of the most important remaining data-structures, needed to implement matching and decomposition:

[4] That is, we ask for the possibility of building a new grammar from a given one, that does not contain some particular production of the later grammar.

[5] For example, while the general language intersection problem for context-free grammars (CFG) is non-decidable, the intersection between a regular CFG and a non-recursive CFG happens to be decidable Nederhof and Satta (2004).

- `binding : var * term`: a representation of a context-dependent restriction, introduced by the pattern `name_pat`, as described in §3.1.3.
- `decom_ev : term → Set`: a dependently-typed representation of a decomposition of a given term `t`, between a context and a sub-term. We make this type dependent on `t`, and include in `decom_ev` some evidence of soundness of the decomposition.
- `mtch_ev : term → Set`: a dependently-typed representation of one result from a matching/decomposition of a given term `t`, against some pattern. It contains an instance of `binding`, and an instance of `decom_ev` depending on `t` itself.

Their actual purpose will be clear in §3.2.6, when introducing the matching/decomposition algorithm. Also, functions to manipulate values of the previous types will be presented as needed.

3.2 Matching and decomposition

The first challenge that we encounter when trying to mechanize RedexK, is that of 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 recursive function calls can be evaluated. In other words: R does not contain infinite decreasing chains, and, hence, the number of recursive function 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 against 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 against which the matching should 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 hd matches against pattern hd', and if tl matches against tl'. In each case, some part of t has been consumed, and the following steps involve considering for matching some proper sub-term of t. Clearly, we can perform only a finite amount of these kind of steps.

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 against 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. Again, we can perform only a finite amount of these kind of steps.

Finally, step 2b corresponds to the case of pattern `nt_pat` n, which implies looking for productions of n in G that match against 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 grammar G *left-recursive* and a given non-terminal n that witnesses the left-recursion of G. Matching against 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, over and over again.

Indeed, the described matching algorithm does not deal with left-recursion, as is argued in Casey Klein and Findler (2011). 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. We mention here said definition:

Definition 1 (Left-recursion Casey Klein and Findler (2011)) A grammar G is left recursive if $p \rightarrow_G^+ p$ for some pattern p, where $\rightarrow_G^+$ is the transitive

(but not reflexive) closure of $\rightarrow_G : \texttt{pat} \times \texttt{pat}$, the least relation satisfying the following conditions:

$\texttt{nt\_pat}\ n \rightarrow_G p$, if $p \in G(n)$
$\texttt{name\_pat}\ x\ p \rightarrow_G \texttt{p}$
$\texttt{inhole\_pat}\ p_c\ p_h \rightarrow_G p_c$
$\texttt{inhole\_pat}\ p_c\ p_h \rightarrow_G p_h$, if pattern p_c matches against $\texttt{hole\_contxt\_c}$

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, it must be the case that also $\texttt{nt\_pat}\ n \not\rightarrow_G^+ \texttt{nt\_pat}\ n$. 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 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 will 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 $<_{\texttt{t}} : \texttt{term} \rightarrow \texttt{term} \rightarrow \texttt{Prop}$ explaining what happens to the input when it is being consumed, and a relation $<_{\texttt{p}\times\texttt{g}} : \texttt{pat} \times \texttt{grammar} \rightarrow \texttt{pat} \times \texttt{grammar} \rightarrow \texttt{Prop}$, 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 (Paulson, 1986). We will parameterize this relation by the original grammar, to be able to recover the original productions when needed.[6] For a given grammar g, we will denote this last relation with $<^g_{\texttt{t}\times\texttt{p}\times\texttt{g}}$. Note that its type is:

$$\texttt{term} \times \texttt{pat} \times \texttt{grammar} \rightarrow \texttt{term} \times \texttt{pat} \times \texttt{grammar} \rightarrow \texttt{Prop}$$

For a tuple (t, p, G) to be related with another *smaller* tuple (t', p', G'), according to $<^g_{\texttt{t}\times\texttt{p}\times\texttt{g}}$, it must happen the following:

$$t' <_{\texttt{t}} t \vee (t' = t \wedge (p', G') <_{\texttt{p}\times\texttt{g}} (p, G))$$

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, however, this definition shows

[6] We will present in §3.2.4 the situations where this is needed.

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 $(t', p', G') <^{g}_{\mathtt{t}\times\mathtt{p}\times\mathtt{g}} (t, p, G)$, 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 *interesting chains* or *chains of interest*.

The previous 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.

For our implementation, we will simply use Coq's standard library implementation of lexicographic product on pairs:

```
slexprod : forall A B : Type, (A → A → Prop) → (B → B → Prop) → A * B → A * B → Prop
```

That is, for a given grammar g, $<^{g}_{\mathtt{t}\times\mathtt{p}\times\mathtt{g}}$ will be defined in terms of `slexprod`. As noted, this relation is well-founded provided we are able to prove the well-foundedness of its composing relations. Hence, the following type is inhabited:

```
forall (A B : Type) (leA : A → A → Prop) (leB : B → B → Prop),
well_founded leA → well_founded leB → well_founded (slexprod A B leA leB)
```

Note that `well_founded le` simply codifies the type stating that the relation `le` is well-founded:

```
well_founded = fun (A : Type) (R : A → A → Prop), forall a : A, Acc R a
```

Where, for a given relation `R` and element `a` in its domain, `Acc R a` is the type of proofs showing that `a` is *accesible* for relation `R`: informally, there is only a finite amount of elements smaller than `a`, according to `R` (see Chlipala (2019), section 7.1, for a more detailed presentation of the concept).

In what follows, we will present our definition for the relations $<_{\mathtt{t}}$ and $<_{\mathtt{p}\times\mathtt{g}}$. Fortunately, they describe simple processes for which it is possible to prove their well-foundedness without resorting to complex arguments.

3.2.2 Input consumption

As stated in the previous sub-section, $<_{\mathtt{t}}$ should model how the input term is *consumed* or *decomposed* during matching and decomposition. This amounts to relate a term t' with another term t, in that order, if from term t we can reach term t' during a recursive evaluation of matching/decomposition of t against some pattern. A reasonable definition for this relation can be, simply, this: $<_{\mathtt{t}} = <_{\mathsf{subt}}$, where $<_{\mathsf{subt}}$ denotes the relation:

```
subterm_rel : term → term → Prop
```

that links a term with each of its sub-terms.[7] That is, $t' <_{\mathtt{t}} t$ if t' is just any sub-term of t. For the actual specification of matching and decomposition, this

[7] In turn, `subterm_rel` is defined for verification purposes of the matching/decomposition algorithm. See §3.2.6.

definition is enough (as we will see when introducing the process in §3.2.6). This does not avoid for more exotic patterns, that could be introduced in the future, to have a different behavior on input consumption, in such a way that some recursive evaluation involves a term that is not an actual sub-term of the original input term.[8] For this first iteration of our tool, we just acknowledge that this could happen in a future version of the language of patterns of Redex. Hence, we will assume the existence of a relation $<_{\mathtt{t}}$, with the purpose already described, and that, for the time being, it is exactly $<_{\mathsf{subt}}$.

While in this sub-section we are concerned with $<_{\mathtt{t}}$, there is still a related issue that also involves our parameterized lexicographic product of relations: recall that, given a grammar g, $(t', p', G') <^{g}_{\mathtt{t}\times\mathtt{p}\times\mathtt{g}} (t, p, G)$ holds if and only if:

$$t' <_{\mathtt{t}} t \vee (t' = t \wedge (p', G') <_{\mathtt{p}\times\mathtt{g}} (p, G))$$

If what it actually holds is $t' <_{\mathtt{t}} t$, then the pair (p', G') can be anything. As we mentioned in the previous sub-section, this means that $<^{g}_{\mathtt{t}\times\mathtt{p}\times\mathtt{g}}$ contains chains that do not necessarily model an actual process of matching/decomposition. In our case, the chains that will be of interest are the ones where, when there is actual input consumption (*i.e.*, $t' <_{\mathtt{t}} t$), then p' is some sub-pattern of p (following rules to be introduced in §3.2.4) and $G' = g$. That is, after a step of input consumption, we *re-install* the original grammar g in the tuple. This is needed since we need to guarantee that, if a pattern of the form `nt_pat` n, for some non-terminal `n`, appears during matching/decomposition, we have at our disposal every production of `n`, for production searching. The only situation where it is guaranteed that we do not have to worry about this situation, is *after* the appearance of pattern `nt_pat` n, and before the next step where input consumption occurs. This is the phase 2b mentioned in §3.2.1: when a pattern like `nt_pat` n appears, the process of production searching begins. And because g is non-left recursive (recall that we only assume such kind of grammars; see §3.2.1), it is guaranteed that we will not need to look for another production of `n`, as long as this phase of the matching/decomposition process continues. This will become more clear when introducing the actual algorithm of matching/decomposition and its specification, beginning in §3.2.4.

3.2.3 Pattern and production consumption

We now turn to the specification of $<_{\mathtt{p}\times\mathtt{g}}$, which explains how evolve the pattern and the grammar (over which we interpret the non-terminals from the pattern), when there is no input consumption. This stage of the matching/decomposition algorithm corresponds to phases 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,

[8] Consider, for example, context-dependent rules such as in the pattern (`x_!_ x_!_`), described in §2. It only matches against a list of 2 different variables: its semantics cannot be explained by considering only recursive evaluations of matching between proper sub-patterns and sub-terms of the input term.

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

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

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

Fig. 7: Consumption of pattern and productions.

to see if matching/decomposition can continue. Note that the evolution of the pattern in this stage of the algorithm is already described in Definition 1, in §3.2.1. Indeed, $<_{\mathtt{p\times g}}$ will be defined just considering the inverse of the relation showed in said definition, plus some particular considerations about grammars.

We present in Figure 7 the definition of $<_{\mathtt{p\times g}}$. Matching a term t against a pattern of the form $\mathtt{inhole\_pat}\ p_c\ p_h$, means trying to decompose the term between some context that matches against pattern pc, and some sub-term of t that matches against 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: pc. 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 against `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 against pattern p_h, we will still being considering the whole input term t: no input consumption occurred as a result of extracting out from t a context that matches against p_c. 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. Note that we also preserve the grammar, and that we do not force this situation to hold only when the pattern pc matches against `hole_contxt_c`. This results in a relation $<_{\mathtt{p\times g}}$ that contains some chains of tuples that do not correspond to the matching/decomposition algorithm. This is not a problem, since it also contain the chains that we need, and the resulting definition is simpler.

The case for the pattern $\mathtt{name\_pat}\ x\ p$ can be explained on the same basis as with the previous cases: matching term t against pattern $\mathtt{name\_pat}\ x\ p$ involves, first, trying to match the whole term t against the sub-pattern p. There is no input consumption involved in this first step, but there is a reduction of the pattern. We also preserve the grammar in this step.

Finally, the last case refers to the pattern $\mathtt{nt\_pat}\ n$: 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, to be justified below). 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 $G \setminus (n, p)$.

The previously mentioned invariant about G will be maintained through the chains of interest of our lexicographic product $<^{g}_{\mathtt{t}\times\mathtt{p}\times\mathtt{g}}$, for a given grammar g over which we begin the matching/decomposition process. Preserving this invariant involves maintaining unaltered the grammar over which we interpret the non-terminals of the pattern (as shown in the first 3 cases of Figure 7) in the absence of pattern decomposition, but allowing $<_{\mathtt{p}\times\mathtt{g}}$ to consider a *smaller* grammar once some production is tested (last case in Figure 7), on the basis of the non-left-recursivity of the grammar being considered. Finally, preserving the invariant also involves reestablishing to the original grammar g, once the matching returns to input consumption (something to be specified §3.2.4).

A final concern about $<_{\mathtt{p}\times\mathtt{g}}$ is related with convincing ourselves that this relation does not contain infinite decreasing chains: *i.e.*, that it is well-founded. Looking again at Figure 7 we observe that, at each step, either the pattern is being reduced, or the grammar considered contains less productions. Hence, for example, a simple proof by a nested induction, first, on the size of the grammar and, at each case, structural induction on the pattern, suffices to show the well-foundedness of $<_{\mathtt{p}\times\mathtt{g}}$.

3.2.4 Specification of matching

We now turn to the task of modifying the original specification for matching and decomposition from RedexK. As we will see, our specification defines a simple generalization of the original problem, as presented in Casey Klein and Findler (2011): here, we will allow for the matching and decomposition algorithm to interpret the non-terminals in the pattern by looking for productions from some arbitrary grammar, not just the original grammar, during some specific phase of the process.

The specification for RedexK consists of 2 mutually inductive formal systems, that help to build proofs for judgments that speak about matching and decomposition. We will begin by presenting the formal system that specifies the notion of matching. Judgments about matching have the form $G \vdash t : p \mid b$, stating that pattern t matches against pattern p, under the productions from grammar G, producing the bindings b (which could be an empty set of bindings, denoted with $\oslash$). The non-terminals that may appear on pattern p will be interpreted in terms of the productions from G. Hence, the formal system that allows us to build proofs for such judgments, explains the semantics of matching against a given pattern.

Here, we will consider a generalization of this problem: our formal system will serve to build proofs for judgments of the form $G \vdash t : p_{G'} \mid b$, stating almost the same as the previous formal system, with the particular difference that, *initially*, we interpret the non-terminals from p looking for their productions in some arbitrary grammar G' (that is what the notation $p_{G'}$ tries to

$$G \vdash \texttt{lit\_term}\ a : (\texttt{lit\_pat}\ a)_{G'} \mid \varnothing \qquad G \vdash \texttt{hole\_contxt\_c} : \texttt{hole\_pat}_{G'} \mid \varnothing$$

$$G \vdash \texttt{nil\_term\_c} : \texttt{nil\_pat\_c}_{G'} \mid \varnothing \qquad \frac{G \vdash t : p_{G'} \mid b}{G \vdash t : (\texttt{name\_pat}\ x\ p)_{G'} \mid b \sqcup \{(x, t)\}}$$

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

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

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

$$\frac{G \vdash t = \texttt{hole\_contxt\_c}[\![t]\!] : (p_c)_{G'} \mid b_c \qquad G \vdash t : (p_h)_{G'} \mid b_h}{G \vdash t : (\texttt{inhole\_pat}\ p_c\ p_h)_{G'} \mid b_c \sqcup b_h}$$

Fig. 8: Modified specification of matching.

suggest). Only when input consumption begins, we will turn to the original grammar G. Figure 8 presents a simplified fragment of our formal system. Following a top-down, left-to-right order, the first rule states that a term of the form `lit_term` a (a literal) only matches against a pattern of the form `lit_pat` a, producing no bindings. Here, the grammars play no role. The second rule and third rules can be understood on the same basis.

The fourth rule explains the way in which a pattern of the form `name_pat` x p introduces context-dependent restrictions, when a given term t successfully matches against it. This implies that sub-pattern p matches against t, producing bindings b, and a new binding (x, t) can be added to b. This is done through the disjoint-union of bindings, denoted with $b \sqcup \{(x, t)\}$. This operation is defined only if there is no binding for x in b, or, if $b(x) = t$. Note that, given that we recursively prove matching for the whole input term t (*i.e.*, no input consumption occurred), we preserve the grammar G'. That is, we are following the chains from the well-founded relation $<_{\texttt{p}\times\texttt{g}}$ (Figure 7). This semantics accounts for the behavior shown in §2, when referring to the sub-terms of a given term, after the matching, through the names presented in the pattern. See, for example, Figure 2, where the pattern being described is used in defining the equations that capture the meta-function `fv` from the λ-calculus.

The fifth rule explains what it means for a term t to match against 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 some arbitrary grammar

G': then, that matching is successful if there exist some $p \in G'(n)$, such that t matches against 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 $<_{\mathtt{p}\times\mathtt{g}}$. 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 sixth rule describes matching of a term that represents a list of terms (`cons_term_c` t_{hd} t_{tl}) against a pattern that also describes a list of terms, through a list of patterns (`cons_pat_c` p_{hd} p_{tl}). We consider this matching partially successful if the head of the list of terms, t_{hd}, matches against the head of the list of patterns, p_{hd}, producing some bindings b_{hd}. Note that, given that this last match is done over an actual sub-term of the original input, we *re-install* the original grammar G, to interpret the non-terminals from p_{hd}. We also ask for the tail of the input list of terms, t_{tl}, to match against the tail of the list of patterns, p_{tl}. Again, this match is done over a sub-term of the input term, hence, we consider the original grammar G. The non-left-recursivity of the grammars being considered does not interfere with the possibility of both, p_{hd} and p_{tl}, include patterns `nt_pat`.

If successful, the previous match produces some bindings b_{tl}. Finally, it will be possible to prove the match between the original list of terms and patterns, if the disjoint union between b_{hd} and b_{tl} is defined. Consider what would happen if p_{hd} and p_{tl} contain `name_pat` patterns that introduce contradictory restrictions: the match should fail. This also shows how these context-dependent restrictions operate, to impose conditions over different parts of a given term. Finally, the cases for contexts `hd_contxt` and `tail_contxt` (contexts in the form of list of terms, with one special hole), are totally analogous to this case.

The last 2 cases in Figure 8 refer to the matching of a term t against 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 against pattern p_c, and some sub-term, that matches against 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, between some context C, that matches against 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 judgments (and the formal system itself), by generalizing them in the same way done for the matching judgments: now, we consider judgments of the form $G \vdash t = C[\![t']\!] : p_{G'} \mid b$, meaning almost the same as the previous decomposition judgment, with the possibility of interpreting the non-terminals

in p, initially, through the productions from some arbitrary grammar G' (that is, $p_{G'}$). We will explain in detail this formal system in the next sub-section.

Returning to the cases about `inhole_pat` patterns, in Figure 8, note that we distinguish the case where the decomposition step actually consumes some portion from t, from the case where it does not. The first situation (described in the first 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 temporally interpreting the non-terminals of this last pattern with productions from the original grammar G ($(p_c)_G$).

The second rule for `inhole_pat` considers the possibility that the initial decomposition did not consume some part of the input term t. That is, p_c matched against a single hole (`hole_pat`). In that case, the decomposition did not produce an actual sub-term of t, and the following match against pattern p_h is done with the whole input term. Hence, $(p_h)_{G'}$. Note that, again, in both cases of `inhole_pat`, the final set of bindings in the judgment is the result of the disjoint union of bindings from the decomposition of the term, and the matching with its sub-term.

3.2.5 Specification of decomposition

The final part of the specification concerns specifically with the process of decomposition. That is, part of the semantics of the `inhole_pat` pattern. As already mentioned, the original specification of this concept comes in the form of a formal system to prove judgments of the form $G \vdash t = C[\![t']\!] : p \mid b$ (explained previously), that we generalize to judgments of the form $G \vdash t = C[\![t']\!] : p_{G'} \mid b$, that we also introduced in the previous sub-section. Figure 9 presents a simplified fragment of the modified formal system. We describe the rules following a top-down order.

The first rule specifies the process of decomposition of a given term t, when the pattern that describes the possible context is just `hole_pat`. In that case, since such context only matches against `hole_contxt_c`, the term t is decomposed between a context that is just a single hole, and t itself as the sub-term. No binding is generated.

The second and third rules explain the decomposition of a list of terms `cons_term_c` t_{hd} t_{tl}, between a context that matches against a list of patterns `cons_pat_c` p_{hd} p_{tl}, and some sub-term. In the second 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 against pattern p_{hd}, as indicated in the premise of the inference rule. Indeed, recall that the context from the decomposition must match against pattern `cons_pat_c` p_{hd} p_{tl}. If this patterns is describing some context whose hole points to somewhere in the head of the list of terms, it must be the case that the sub-pattern p_{hd} matches against some context. Note that the whole premise is stating that the

$$G \vdash t = \texttt{hole\_contxt\_c}[\![t]\!] : \texttt{hole\_pat}_{G'} \mid \oslash$$

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

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

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

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

$$\frac{G \vdash t = \texttt{hole\_contxt\_c}[\![t]\!] : (p_c)_{G'} \mid b_c \qquad G \vdash t = C_h[\![t_h]\!] : (p_h)_{G'} \mid b_h}{G \vdash t = (\texttt{hole\_contxt\_c} +\!+ C_h)[\![t_c]\!] : (\texttt{inhole\_pat}\ \ p_c\ \ p_h)_{G'} \mid b_c \sqcup b_h}$$

$$\frac{G \vdash t = C[\![t']\!] : p_{G'} \mid b}{G \vdash t = C[\![t']\!] : (\texttt{name\_pat}\ \ x\ \ p)_{G'} \mid b \sqcup \{(x, C)\}}$$

Fig. 9: Modified specification of decomposition.

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 against the tail of the list of patterns p_{tl}. Finally, note that in the decomposition through sub-pattern p_{hd}, and the matching against sub-pattern p_{tl}, the non-terminals of these patterns are interpreted in terms of productions from the original grammar, G. This is done since, in each case, we are operating over a proper sub-term of the original input.

The third rule can be explained on the same basis as in the previous case, with the sole difference that, now, the context from the resulting decomposition is pointing to somewhere in the tail of the original list of terms. Note that, in both rules, the resulting bindings are the disjoint union of bindings from the decomposition and the matching step.

The remaining rules can also be understood in similar terms as with the previous rules, the exception being the case of the `inhole_pat` pattern. Note that this situation corresponds to an original pattern of the form:

$$\texttt{inhole\_pat}\ (\texttt{inhole\_pat}\ p_c\ \ p_h)\ \ p_{h'}$$

that we matched against some term t. The semantics of this involves a first step of decomposition of t between some context that matches against sub-pattern `inhole_pat` p_c p_h, and some sub-term that matches against sub-pattern $p_{h'}$. In the rules from Figure 9 for the case of pattern `inhole_pat`, we are describing what it means, in this situations, that first step of decomposing t in terms of a context that matches against pattern `inhole_pat` p_c p_h. Since the whole pattern must match against some context, it means that, both, p_c and p_h, are patterns describing contexts. Now, through the pattern `inhole_pat` p_c p_h we are decomposing again the context, a first part that should match against p_c, and a nested context (to be put within the hole of the previous context) that matches against p_h. This idea is expressed through the premises of both inference rules for the case of the `inhole_pat` pattern. Note that, again, we distinguish the case where p_c produces an empty context, from the case where it does not. The intention being to be able 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 rules 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 the context C_h 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, in Casey Klein and Findler (2011).

Naturally, the actual mechanization is far more complex than what we present here. The intention is to provide the reader with a high-level view of the main ideas behind the mechanization.

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 9) 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 but sharing intermediate results.

Supporting data-structures. In Figure 10 we show some of the implemented data-structures used to represent the results returned by RedexK's algorithm.

```
Definition binding := prod var term.
Definition bindings := list binding.

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 → bindings → mtch_ev t.

Definition mtch_powset_ev (t : term) := list (mtch_ev t).
```

Fig. 10: Mechanization of decomposition and matching results.

The result of a matching/decomposition of a term t (against some given pattern) will be represented through a value of type `mtch_ev` t. Naturally, making the type dependent on t is done for future soundness checking. The algorithm could return several values of this type, each one representing a possible match or a decomposition. We represent this several values through the list type `mtch_powset_ev` t.[9]

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 10). For reasons of brevity, we drop the dependence of the previous value on term t, in the notation used. We will maintain the same notation used so far for bindings. In particular, recall that an empty list of bindings is denoted as $\oslash$. For a value of the list type `mtch_powset_ev` t, we will denote it decorating it with its dependence on the value t: $[(d, b), ...]_t$

Values type `decom_ev` t represent a decomposition of a given term t, between a context and a sub-term. We make the type dependent on t for soundness checking purposes, and we include in the value some evidence of soundness of the decomposition: in particular, evidence showing that a sub-term $subt$ extracted in the decomposition is, either, t itself (proof of type $subt = t$) or a proper sub-term of t (proof of type `subterm_rel` $subt$ t). Recall that `subterm_rel` is our mechanization of relation $<_{\mathsf{subt}}$ (see §3.2.2). Soundness properties about a context c extracted in the decomposition are, either, embedded in the `decom_ev` value itself (c = `hole_contxt_c`, when `subt` = `t`), or they emanate from properties stated through the formal system that captures decomposition (note that this system does not explicitly specify properties about the sub-term extracted in the decomposition, but it does capture the context).

[9] Naturally, this allows for repeated values to occur in the result. This does not affect desired soundness properties.

Since a value of type `mtch_ev` t could represent a single match or a single decomposition, we distinguish an actual match using an empty decomposition `empty_d_ev` t (denoted as $\bullet_t$ or, simply, $\bullet$, when it is clear from context the actual term t upon which the value depends). A value of type `mtch_ev` t that actually represents a decomposition, will contain a value `decom_ev` t of the form `nonempty_d_ev` t C *subt* *ev*, for a proper context C, sub-term *subt* and soundness evidence *ev*. We will denote values constructed this way as $(C, \mathit{subt})_t^{\mathit{ev}}$. For a proof of type $\{\mathit{subt} = t \wedge C = \texttt{hole\_contxt\_c}\} + \{\texttt{subterm\_rel}\ \mathit{subt}\ t\}$, when we can determine the actual disjunct proved we will indicate it with its type. For example, if we know that what it actually holds is `subterm_rel` *subt* t, we will write $(C, \mathit{subt})_t^{\texttt{subterm\_rel}\ \mathit{subt}\ t}$. Also, in this context we will write just $\mathit{subt} = t$, when what it holds is predicate $\mathit{subt} = t \wedge C = \texttt{hole\_contxt\_c}$. Finally, when it is required to simplify the notation, and when it is clear from the context, we will allow us not to include information about dependence on the particular term t that is being considered.

Matching and decomposition algorithm as a least-fixed-point. As is common practice in functional programming, we will capture the intended matching/decomposition algorithm as the least fixed-point of a *generator function* or *functional*. That is, we will provide equations that actually capture a function that receives an *approximation* of our intended algorithm, and uses it to return a better approximation. Provided that we can show that this generator function respects our well-founded relation (described in §3.2.1), through Coq's `Fix` combinator we can get, in return, a function that is *total* over the domain of that relation. Now, looking at `Fix`'s implementation, we see that it defines a process that unfolds our generator function *only* as much as needed to reach to a result, doing primitive recursion over the proof of accessibility (with respect to the provided well-founded relation) of the parameter upon which we are evaluating our generator.[10] So, this process is guaranteed to terminate and it can be shown that it emulates the behavior of the least fixed-point of our generator function.[11]

This fixed-point will be a function that captures so good our intended algorithm, that it cannot be *improved* by our generator function: *i.e.*, it is a fixed-point and, even more, *is* the intended algorithm.

Consider the following function type:

```
forall (g1 : grammar) (tpg1 : matching_tuple),
 (forall tpg2 : matching_tuple, matching_tuple_order g1 tpg2 tpg1
                                →
                                mtch_powset_ev (matching_tuple_term tpg2))
 →
 mtch_powset_ev (matching_tuple_term tpg1)
```

[10] For example, in `https://coq.inria.fr/refman/language/coq-library.html#index-20`

[11] If we can also prove that our generator function does not *distinguish* extensionally equal approximations, we can also get a proof showing that through the `Fix` combinator we get a fixed-point of our generator function (lemma `Fix_eq` from Coq's standard library). The fact that it is also the least fixed-point, could be proved on the basis of the way in which `Fix` operates.

It corresponds to a family of generator functions, $\mathsf{M}_{\mathsf{ev\_gen}}$, parameterized over grammars and tuples of terms and patterns, and that will *improve* candidates of matching/decomposition functions.

In the previous type, `matching_tuple` is defined as the type `term` × `pat` × `grammar` and `matching_tuple_order` is the mechanization of our well-founded relation from §3.2.1. Finally, for a given value *tpg* : `matching_tuple`, `matching_tuple_term` *tpg* : `term` is just the projection of the first component of *tpg*. Note that, for given *G1* : `grammar`, tuple *Tpg1* : `matching_tuple`, then, $\mathsf{M}_{\mathsf{ev\_gen}}$ *G1 Tpg1* has type:

```
(forall tpg2 : matching_tuple, matching_tuple_order G1 tpg2 Tpg1
                               →
                               mtch_powset_ev (matching_tuple_term tpg2))
→
mtch_powset_ev (matching_tuple_term Tpg1)
```

Hence, $\mathsf{M}_{\mathsf{ev\_gen}}$ *G1 Tpg1* will be our intended generator function for grammar *G1* and matching tuple *Tpg1*. It receives as a parameter a function that it will improve: the type of this parameter codifies the idea that it is a function that knows how to compute the matching/decomposition for any tuple that is *smaller* than *Tpg1*, according to `matching_tuple_order` *G1*. Finally, in $\mathsf{M}_{\mathsf{ev\_gen}}$ *G1 Tpg1*, *G1* will represent the original grammar over which we want to compute the matching and decomposition indicated by the tuple *Tpg1*.

Figure 11 shows 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. Also, we will employ the same syntax for terms and patterns, resorting to context for disambiguation. In general the syntax is self-explanatory: for example **hole** represents a hole term or pattern (depending on context), **a** represents a literal term or pattern, etc.

The first 4 equations of Figure 11 can be understood by comparison with the specifications of matching (Figure 8) and decomposition (Figure 9). For example, the first equation explains the matching and decomposition of a term **hole** against a pattern **hole**. Note that the second inference rule of matching specifies that such term matches against such pattern, producing no bindings: this is represented by the pair $(\bullet, \oslash)$, in the result captured in the first equation. Also, the first rule of decomposition specifies that given some term t, it can be decomposed between a single hole (that matches against pattern **hole**) and t itself, producing no bindings. Assuming $t = \textbf{hole}$, the pair $((\textbf{hole}, \textbf{hole})^{\textbf{hole}=\textbf{hole}}, \oslash)$ shown in the result captured in the first equation represents the described situation: the context extracted is a single hole, and the sub-term is $t = \textbf{hole}$ itself. Note that we also indicate the type of the actual proof contained in the piece of evidence of soundness of the decomposition: in this case, a proof showing that the sub-term extracted is actually equal to the original term itself (**hole** = **hole**). In the described pairs we dropped the mention to the actual term whose types depend on, but we do mention it for the whole list type containing the previous pairs. Finally, we do not name the parameters to $\mathsf{M}_{\mathsf{ev\_gen}}$ that are not mentioned in the right-hand side of the equations, using a wildcard "_" instead.

$$
\begin{array}{ll}
\mathsf{M}_{\mathsf{ev\_gen}}(\_, (\mathbf{hole}, \mathbf{hole}, \_), \_) & = [((\mathbf{hole}, \mathbf{hole})^{\mathbf{hole}=\mathbf{hole}}, \oslash), (\bullet, \oslash)]_{\mathbf{hole}} \\
\mathsf{M}_{\mathsf{ev\_gen}}(\_, (t, \mathbf{hole}, \_), \_) & = [((\mathbf{hole}, t)^{t=t}, \oslash)]_t \\
\mathsf{M}_{\mathsf{ev\_gen}}(\_, (\mathbf{a}, \mathbf{a}, \_), \_) & = [(\bullet, \oslash)]_{\mathbf{a}} \\
\mathsf{M}_{\mathsf{ev\_gen}}(\_, (\mathbf{nil}, \mathbf{nil}, \_), \_) & = [(\bullet, \oslash)]_{\mathbf{nil}} \\
\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), \\
& \qquad sub : \mathsf{subterms}\ t\ t_{hd}\ t_{tl} \\
& \qquad b = b_{hd} \sqcup b_{tl}, \\
& \qquad (d_{hd}, b_{hd})_{t_{hd}} \in M_{ap}(tp_{hd}, lt_{hd}), \\
& \qquad (d_{tl}, b_{tl})_{t_{tl}} \in M_{ap}(tp_{tl}, lt_{tl}), \\
& \qquad lt_{hd} : tp_{hd} <^{g_1}_{\mathtt{t\times p\times g}} tp_{cons}, \\
& \qquad lt_{tl} : tp_{tl} <^{g_1}_{\mathtt{t\times p\times g}} tp_{cons}, \\
& \qquad tp_{cons} = (t, p, g_2), \\
& \qquad tp_{hd} = (t_{hd}, p_{hd}, g_1), \\
& \qquad tp_{tl} = (t_{tl}, p_{tl}, g_1)]_t, \\
& \qquad with\ t = \mathbf{cons}\ t_{hd}\ t_{tl} \\
& \qquad\qquad 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), \\
& \qquad b = b_c \sqcup b_h, \\
& \qquad (d_h, b_h)_{t_c} \in M_{ap}(tp_h,\ lt_h), \\
& \qquad lt_h : tp_h <^{g_1}_{\mathtt{t\times p\times g}} tp_{inhole}, \\
& \qquad tp_h = (t_c, p_h, g_h), \\
& \qquad g_h \text{ according to Figure 8}, \\
& \qquad ((C, t_c)^{ev}_t, b_c)_t \in M_{ap}(tp_c, lt_c), \\
& \qquad lt_c : tp_c <^{g_1}_{\mathtt{t\times p\times g}} tp_{inhole}, \\
& \qquad tp_{inhole} = (t, p, g_2), \\
& \qquad tp_c = (t, p_c, g_2)]_t, \\
& \qquad with\ p = \mathbf{in\text{-}hole}\ p_c\ p_h \\
\mathsf{M}_{\mathsf{ev\_gen}}(g_1, (t, p, g_2), M_{ap}) & = [(d, b') \mid b' = \{(x, \mathsf{named}(t, d))\} \sqcup b, \\
& \qquad (d, b) \in M_{ap}(tp_{p'}, lt_{p'})), \\
& \qquad lt_{p'} : tp_{p'} <^{g_1}_{\mathtt{t\times p\times g}} tp_{name}, \\
& \qquad tp_{name} = (t, p, g_2), \\
& \qquad tp_{p'} = (t, p', g_2)]_t, \\
& \qquad with\ p = \mathbf{name}\ x\ p' \\
\mathsf{M}_{\mathsf{ev\_gen}}(g_1, (t, \mathbf{nt}\ n, g_2), M_{ap}) & = [(d, \oslash) \mid (d, b) \in M_{ap}(tp_p, lt_p), \\
& \qquad lt_p : tp_p <^{g_1}_{\mathtt{t\times p\times g}} tp_n, \\
& \qquad tp_n = (t, \mathbf{nt}\ n, g_2), \\
& \qquad tp_p = (t, p, g_2 \setminus (n, p)), \\
& \qquad p \in G(n)]_t
\end{array}
$$

Fig. 11: Generator function for the matching and decomposition algorithm.

The fifth equation explains the matching and/or decomposition of a list of terms (**cons** t_{hd} t_{tl}) against 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 subterm 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 inspect 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 (*i.e.*, they are values of the form $\bullet$), then the whole operation will be considered just a matching of the original list of terms (rule for matching of `cons_term_c`, Figure 8) and select must build an *empty* decomposition of the proper type to represent this: `decom_ev` (**cons** t_{hd} t_{tl}). If only d_{hd} is a decomposition, of the form $(C, t_{hd'})^{ev_{hd}}_{t_{hd}}$, then the whole operation is interpreted as a decomposition of the original list of terms on the head of the list (first rule of decomposition for `cons_term_c`, Figure 9). In that case, select builds a value of type `decom_ev` (**cons** t_{hd} t_{tl}), of the form $(\mathtt{hd\_contxt}\ C\ t_{tl}, t_{hd'})^{ev_{\mathbf{cons}\ t_{hd}\ t_{tl}}}_{\mathbf{cons}\ t_{hd}\ t_{tl}}$. Observe the correspondence between, on the one hand, the context and the sub-term from this decomposition, and, on the other hand, the context and sub-term specified in the first rule of decomposition for `cons_term_c`, Figure 9. Finally, for the mechanization of select to be able to build the required soundness proofs of decomposition (for `decom_ev`), we need to provide to it the original list of terms, and evidence *sub* showing that t_{hd} and t_{tl} are the actual head and tail of the original input term (*sub* : subterms t t_{hd} t_{tl}).

The remaining equations can be understood on the same basis as the previous one, requiring only some explanation the equations for the patterns **in-hole** and **name**: in the first case, the auxiliary function combine 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; in the case of the **name** pattern, the auxiliary function named plays a similar role as the previous one: it helps in deciding if the result is a decomposition or a matching, in order to make the binding refer to the context extracted (case of **name** pattern in Figure 9) or the actual input term (case of **name** pattern in Figure 8), respectively.

Finally, as mentioned at the beginning of this section, we define the desired matching/decomposition algorithm, M_{ev}, as the least fixed-point of the previous generator function. We show in Figure 12 its Coq definition.

```
Definition Mev (g : grammar) (tup : matching_tuple) :
mtch_powset_ev (matching_tuple_term tup) :=
(Fix
 (matching_tuple_order_well_founded g)
 (* dependent range type of the function that we are building *)
 (fun tup : matching_tuple ⇒ mtch_powset_ev (matching_tuple_term tup))
 (* generator function *)
 (Mev_gen g))
tup.
```

Fig. 12: Definition of M_{ev} in Coq.

3.2.7 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 (for example, the semantics rules presented in §2 to capture β-contractions). 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. Its mechanization does not require the development of new concepts or the use of complex tools, and it relies, heavily, on the notion of matching previously presented. The reader is invited to look at the mechanization of this formal system, in module reduction.v of the source code accompanying this paper.

4 Soundness of matching

In the original paper of RedexK it is proved the expected correspondence between the presented algorithm and its specification. In our mechanization we reproduced those results, for the least-fixed-point of $\mathsf{M}_{\mathsf{ev\_gen}}\ g\ (t, p, g')$, for arbitrary g, t, p and g'. Naturally, for a given grammar g, the original intention of matching and decomposition corresponds to the least-fixed-point of $\mathsf{M}_{\mathsf{ev\_gen}}\ g\ (t, p, g)$. In what follows, $M\_ev\ g\ (t, p, g')$ will represent the least-fixed-point of $\mathsf{M}_{\mathsf{ev\_gen}}\ g\ (t, p, g')$.

With respect to the soundness checks of the mechanized version of the matching/decomposition algorithm, we were able to implement a proof of the completeness of the process, with respect to its specification. We show its statement in Figure 13.

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`.

Theorem 1 (Completeness of M_ev)

```
Theorem completeness_M_ev :

forall 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 → exists (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)))).
```

Fig. 13: Mechanization of the proof of completeness of *M_ev*: statement.

Theorem 2 (Soundness of M_ev)

```
Theorem soundness_M_ev :

forall G1 G2 p t sub_t b C,
 (In (mtch_pair t (empty_d_ev t) b)
     (M_ev G1 (t, (p, G2)))          →   G1 |– t : p, G2 | b)
 ∧
 (exists (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)))          →   G1 |– t1 = C [ t2 ] : p , G2 | b).
```

Fig. 14: Mechanization of the proof of soundness of *M_ev*: statement.

```
Definition pat_grammar_evolution_trans :=
  clos_trans_1n (pat * grammar) pat_grammar_evolution.

Definition non_left_recursive_grammar :=
  forall (p : pat) (g1 g2 : grammar),
   not (pat_grammar_evolution_trans (p, g1) (p, g2)).
```

Fig. 15: Non-left-recursive grammars.

The converse, the soundness of the process with respect to its specification, was also mechanized, and we show its statement in Figure 14. The proofs 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. This is useful to prove soundness. Completeness can be proved by *rule induction* on the evidences of match and decomposition.

Lemma 1 (Soundness of replacement of grammars)

```
Lemma non_left_g_rm_sound :

forall g1 g2 t n p (proof_in : prod_in_g (n, p) g2) b,
  non_left_recursive_grammar →

  (g1 |– t : p, remove_prod (n, p) g2 proof_in | b
                        ↔
              g1 |– t : p, g2 | b)

  ∧

  (forall (C : contxt) (subt : term),
    g1 |– t = C[subt] : p, remove_prod (n, p) g2 proof_in | b
                          ↔
               g1 |– t = C[subt] : p, g2 | b).
```

Fig. 16: Mechanization of the proof of soundness of our manipulation of grammars: statement.

We also verified the correspondence between our mechanized specification of matching and decomposition, and the original formal systems from the paper. In doing so, we needed to prove, first, some soundness results about our own formal systems. On the one hand, we needed to verify that our replacement of grammars during the phase of non-consumption of input is actually sound. In doing so, we required an actual formalization of our assumptions about the grammars under consideration: that is, that they are non-left-recursive. We show its definition in Figure 15. Note that `pat_grammar_evolution` is just the mechanized version of the relation $<_{p \times g}$, that models the phase of non consumption of input, during matching/decomposition. The definition shown in Figure 15 is just an adaptation of the proposal of the authors of RedexK.

We make use of `non_left_recursive_grammar` to strengthen our assumptions, when verifying the soundness of our replacement of grammars. Our formal statement of soundness consists in specifying that, if we have already used some production (for example, when matching against a pattern **nt** *n* and after looking for some production of *n* in the grammar under consideration), removing it from the grammar does not impede us to complete a proof of matching or decomposition. We show its statement in Figure 16. In the statement, `prod_in_g` (`n`, `p`) `g2` is a type (of sort `Prop`) that represents proofs showing that (`n`,`p`) is actually a production from `g2`. Finally, `remove_prod` (`n`, `p`) `g2` `proof_in` builds a new grammar that contains the same productions as `g2`, except for (`n`, `p`).

In addition, we needed to verify that our formal system to specify decomposition, actually help us to build proofs about meaningful statements: in particular, if I can build a proof for a judgment like `G |– t = C [t'] : p, G'`, then it must happen that `t'` is a proper sub-term of `t`, or, if `t = t'`, then, the

Lemma 2 (Soundness of decomposition)

```
Lemma subterm_rel_characterization :

forall G p G' t C t' b,
 G |- t = C [t'] : p, G' | b → subterm_rel t' t ∨ (t = t' ∧ C = hole_t).
```

Fig. 17: Mechanization of the proof of soundness of decomposition: statement

Theorem 3 (Completeness of our formal systems)

```
Theorem from_orig :

  forall G t p b,
   non_left_recursive_grammar →
   G |- t : p | b →
   G |- t : p, G | b

  with from_orig_decomp :

   forall G C t1 t2 p b,
   non_left_recursive_grammar →
   G |- t1 = C [ t2 ] : p | b → G |- t1 = C [ t2 ] : p , G | b.
```

Fig. 18: Mechanization of the proof of completeness of our formal systems: statement.

context `C` is simply a hole. The statement of our mechanized proof for this result is shown in Figure 17.

With the obtained tools, we were able to tackle the formal verification of correspondence between our formal system and the original ones from RedexK. Figure 18 shows the statement of our completeness result: under the assumption that grammars are non-left-recursive, and given a judgment that can be proved in one of the original formal systems, we can reproduce the same result within our formal systems. In particular, if we can prove `G |- t : p | b`, that is, a proof of matching under the original formal system, within our formal system we can prove `G |- t : p, G | b`. Note that, as expected, we begin interpreting the non-terminals from the pattern using the same original grammar G. A similar result is obtained for the formal systems specifying decomposition.

Finally, to prove soundness of our specification, with respect to the original formal systems, we need to reduce the spectrum of possible grammars over which we begin interpreting the non-terminals of the pattern. In particular, to reproduce, with the original formal system, a proof about a judgment `G |- t : p, G' | b`, we cannot allow for `G'` to be *any* grammar. We will be limited only to grammars that contain some, or all, of the productions from the original grammar `G`. We formalize this concept through a relation `gleq`,

```
Axiom gleq : grammar → grammar → Prop.
Axiom gleq_refl : forall G, gleq G G.
Axiom gleq_trans : forall G G' G'', gleq G G' → gleq G' G'' → gleq G G''.
Axiom gleq_weakening : forall {G G' p}, gleq G' G → prod_in_g p G' → prod_in_g p G.
Axiom gleq_remove: forall p G pf, gleq (remove_prod p G pf) G.
```

Fig. 19: Mechanization of the proof of soundness of our formal systems: axiomatization of `gleq`.

Theorem 4 (Soundness of our formal systems)

```
Theorem to_orig :

forall G G' t p b,
   gleq G' G →
   G |- t : p, G' | b → G |- t : p | b

  with to_orig_decomp :

   forall G G' C t1 t2 p b,
   gleq G' G →
   G |- t1 = C [ t2 ] : p , G' | b →
   G |- t1 = C [ t2 ] : p | b
```

Fig. 20: Mechanization of the proof of soundness of our formal systems: statement.

whose properties we present in Figure 19. Note that the axioms only ask for `gleq` to be reflexive, transitive and to express the idea that, if we can prove `gleq G' G`, then, every production from `G'` is also in `G`.

With the previous tool we can, finally, tackle the expected prove of the statement showing that, if we can conclude something using our formal system, we can reach to a similar conclusion using the original specification. We show its statement in Figure 20.

A natural consequences of this results is that, if M is the original matching/decomposition function from RedexK, and M_no_ev is a function defined in terms of M_ev, that removes possible repeated results and every piece of type-dependency and soundness evidence embedded in the results of M_ev, then, we have the following correspondence:

$$M\ (g, p, t) = M\_no\_ev\ (g, p, t),$$

which is the expected correspondence with the original formalization of RedexK.

5 Related work

Coq libraries for reduction-semantics and related concepts. CoLoR (Blanqui and Koprowski, 2011) 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, in a dependently-typed fashion, 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. In doing so, the user that developed a model in Redex and tries to compile it into Coq using our tool, would be forced to work over a mechanization of the model, in Coq, that does not provide a direct explanation of its semantics (in terms of Redex's own semantics) but rather, through an encoding over CoLoR.

Sieczkowski et. al present in Sieczkowski et al (2010) 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, useful to specify logical systems and their properties, that has at its hearth a notion of patterns and pattern matching. In Bereczky et al (2022) it is presented a mechanization in Coq of a version of this meta-language, including its syntax, its notion of semantics, formal system and verified soundness results. In particular, its syntax defines a language of *patterns*, whose semantics is defined in terms of the set of elements (from a given model) that match against this pattern. In that sense, since the domain

of interpretation of the phrases of this language is the powerset of elements of the given model, matching logic is considered a multi-valued logic. 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 her/his mechanization in Redex.

6 Conclusion

We adapted RedexK (Casey Klein and Findler, 2011) 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 Casey Klein and Findler (2011), 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.

This first iteration enables a plethora of future opportunities for improvement, both, in pursuing a faithful and complete representation of Redex features into Coq, and in improving the possibilities offered by Redex for the development of semantics models.

Extending the capabilities of our mechanized model. 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 Redex's Kleene closure of patterns. This could be a reasonable next step in increasing the set of Redex's features captured by our mechanization.

Finally, another major update of the model would be the addition of typing annotations into the language of patterns, to automatically check well-formedness of the definitions of a given model. Naturally, this would come at the cost of limiting the possible Redex models that can be transpiled into Coq, with our tool. This is the usual tension between allowing expressivity of a language or enforcing well-formedness of the things that we can say with the language.

Further development of decidability results. While the main results shown in the present paper are related with the mechanization of RedexK in Coq, we already mentioned our interest in pursuing the development of a theory of

decidability about a model in Redex. This, to offer to the user a set of tactics and general tools to build decision procedures for the properties that she/he needs to study about a given model. While in the present mechanization we have already make explicit some decidability constraints, about the model being mechanized, this restrictions are actually completely expected for a model in Redex (*e.g.*, decidability of definitional equalities between atomic elements of the language).

Studying the *intersection of languages* problem, adapted to the kind of grammars that can be expressed in Redex, could be of importance to our development. The recognition of equivalence between regular expressions (RE) has been already studied, with several approaches proposed. The classical approach (Hopcroft, 2008) being representing each RE through their corresponding finite automatons, and studying the resulting product automaton. With regard to context-free grammars (CFG), the general problem of deciding equivalence between 2 CFG is known to be non-decidable Hopcroft (2008). However, there are simpler problems known to be decidable. In Nederhof and Satta (2004), Nederhof et. al study the problem of deciding intersection between a CFG and a non-recursive CFG. They show that the problem is decidable and PSPACE-complete. They also show that the problem remains decidable when generalized to several CFG, from which some of them are non-recursive. Recognizing restricted grammars, in Redex style, with good decidability properties could be reasonable next in pursuing these studies.

Solving efficiency issues when testing within Redex. As we already mentioned in the previous section, there is already work done in Coq implementing tools to extract an interpreter based on abstract machines, from a given deterministic reduction semantics model (Sieczkowski et al, 2010). This capability could solve the well-known performance problem in Redex, when trying to use a given mechanized semantics relation to reduce a term, from within the tool. Even more, this solution to the problem comes with the added benefit of a certified correspondence between both interpreters (the original, in terms of reduction semantics, and the extracted, in terms of a classical abstract machine), something that lacks other approaches, like re-writing by hand the whole model in another language.

References

Barendregt HP (1981) The Lambda Calculus: Its Syntax and Semantics. Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co., New York, N.Y.

Bereczky P, Chen X, Horpácsi D, Peña L, Tušil J (2022) Mechanizing matching logic in coq. In: Rusu V (ed) Proceedings of the Sixth Working Formal Methods Symposium, "Al. I. Cuza University", Iasi, Romania, 19-20 September, 2022, Open Publishing Association, Electronic Proceedings in Theoretical Computer Science, vol 369, pp 17–36, DOI 10.4204/EPTCS.369.2

Blanqui F, Koprowski A (2011) 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, DOI 10.1017/S0960129511000120

Casey Klein SJ Jay McCarthy, Findler RB (2011) A semantics for context-sensitive reduction semantics. In: APLAS'11

Chlipala A (2019) Certified Programming with Dependent Types. The MIT Press

Felleisen M, Hieb R (1992) The revised report on the syntactic theories of sequential control and state. TCS 103

Felleisen M, Finlder RB, Flatt M (2009) Semantics Engineering with PLT Redex. The MIT Press

Guha A, Saftoiu C, Krishnamurthi S (2010) The essence of JavaScript. In: ECOOP '10

Hopcroft J (2008) Introduction to Automata Theory, Languages, and Computation. Always Learning, Pearson Education, URL https://books.google.com.ar/books?id=tzttuN4gsVgC

Klein C, Clements J, Dimoulas C, Eastlund C, Felleisen M, Flatt M, McCarthy JA, Rafkind J, Tobin-Hochstadt S, Findler RB (2012) Run your research: On the effectiveness of lightweight mechanization. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, New York, NY, USA, POPL '12, pp 285–296, DOI 10.1145/2103656.2103691, URL http://doi.acm.org/10.1145/2103656.2103691

Landin PJ (1964) The mechanical evaluation of expressions. Comput J 6:308–320

Landin PJ (1965) Correspondence between algol 60 and church's lambda-notation: part i. Communications of the ACM 8:89–101

Lorenzen F, Erdweg S (2013) Modular and automated type-soundness verification for language extensions. SIGPLAN Not 48(9):331–342

Matthews J, Findler RB (2007) An operational semantics for scheme. Journal of Functional Programming

Nederhof MJ, Satta G (2004) The language intersection problem for non-recursive context-free grammars. Inf Comput 192(2):172–184, DOI 10.1016/j.ic.2004.03.004, URL https://doi.org/10.1016/j.ic.2004.03.004

Oeyen B, De Koster J, De Meuter W (2022) Reactive programming on the bare metal: A formal model for a low-level reactive virtual machine. In: Proceedings of the 9th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems, Association for Computing Machinery, New York, NY, USA, REBLS 2022, p 50–62, URL https://doi.org/10.1145/3563837.3568342

Paulson LC (1986) Constructing recursion operators in intuitionistic type theory. J Symb Comput 2(4):325–355

Politz JG, Carroll MJ, Lerner BS, Pombrio J, Krishnamurthi S (2012) A tested semantics for getters, setters, and eval in JavaScript. In: DLS '12

Politz JG, Martinez A, Milano M, Warren S, Patterson D, Li J, Chitipothu A, Krishnamurthi S (2013) Python: The full monty: A tested semantics for the Python programming language. In: OOPSLA '13

Sieczkowski F, Biernacka M, Biernacki D (2010) 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, Springer-Verlag, Berlin, Heidelberg, IFL'10, p 72–88

Soldevila M, Ziliani B, Silvestre B, Fridlender D, Mascarenhas F (2017) Decoding Lua: Formal semantics for the developer and the semanticist. In: Proceedings of the 13th ACM SIGPLAN Dynamic Languages Symposium, DLS 2017

Soldevila M, Ziliani B, Fridlender D (2020) 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

Soldevila M, Ziliani B, Silvestre B (2022) From specification to testing: Semantics engineering for lua 5.2. Journal of Automated Reasoning

Stoy JE (1977) Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. MIT Press, Cambridge, MA, USA

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