Semantics Engineering with Concrete Syntax

Tijs van der Storm
Centrum Wiskunde & Informatica (CWI), Amsterdam, The Netherlands
University of Groningen, The Netherlands

Abstract

Semantics engineering tools like Redex can be used to define, explore, and debug formal definitions of programming language semantics. However, such tools are often based on abstract syntax, which makes the definition of rules and the exploration of execution traces rather unfriendly. In this paper we introduce Credex, a library in the Rascal meta-programming language for defining small-step evaluation-context semantics, where terms and matching patterns are what-you-see-is-what-you-get. Credex employs parsing for decomposing terms into context and redex. Since Rascal's grammar formalism is based on general parsing, a non-unique decomposition of a term literally corresponds to an ambiguous parse. We demonstrate the use of Credex, detail some aspects of its implementation, and discuss three case-studies.

2012 ACM Subject Classification Software and its engineering → Semantics; Software and its engineering → Integrated and visual development environments

Keywords and phrases Semantics engineering, syntax, parsing, reduction semantics

Digital Object Identifier 10.4230/OASIcs.EVCS.2023.29

1 Introduction

Operational semantics with evaluation contexts [10] is a popular framework to define the semantics of programming languages using rewriting rules. The use of contexts to separate traversal from the actual reduction rules leads to concise and modular definitions.

Specifications of formal semantics come to life by making them executable. It has been observed that semantic specifications could form central artifacts in language engineering [11]. Tools such as as Redex [9, 18] and K [23] allow language designers to explore, inspect, test, and debug their language designs. Redex, for instance, has been instrumental to uncover numerous bugs in published formal definitions [13]. Both K and Redex have been used to formalize realistic (subsets of) programming languages (e.g., [5, 17]).

Language workbenches [7, 8] such as Rascal [16] and Spoofax [12] offer integrated meta-formalisms and IDE services dedicated to the principled design and implementation of software languages. Nevertheless, most of the existing language workbenches lack support for semantics engineering. Although the Asf+Sdf Meta-Environment was based on algebraic specification and term rewriting, and both Spoofax and Rascal inherited those features, this means that the default style of defining a semantics is through definitional interpreters, which are limited to big-step evaluation, and do not facilitate the exploration and debugging offered by, e.g., Redex. The recent work on DynSem [26] aims to bridge the gap between interpreters and formal semantics, but still stays close to natural, big-step semantics.

In this paper we present Credex, a framework for defining executable semantics, in the small-step style of Redex. Credex's main selling points are: it is based on *concrete* syntax, rather than abstract syntax (e.g., prefix notation, or s-expressions), and, because Credex is a library in Rascal, it integrates very well with the other language workbench features of Rascal. Credex is novel in its (rather unconventional) use of parsing and (ambiguous) parse forests to drive the reduction process.

Eelco Visser has always been a syntax person. His PhD thesis centered around syntax definition using SDF. He published many papers about concrete syntax embedding, parsing, and syntax definition after that. Recently, he had ventured into dynamic semantics as well. CREDEX combines all of these strands of research.

2 Overview

Structural operational semantics in the style of Plotkin [21, 22] is based on rewriting (or reduction) of syntactic terms, where each rewrite step corresponds to a small-step execution step. The execution of a term is finished when it reduces to a value or no further rules apply (the term is "stuck"). A common slogan heard in this context is "everything is syntax", because both results (values) and auxiliar entities (stores, environments) are represented syntactically.

Ordinary operational semantics involves defining reduction rules that merely traverse the term until a redex is found "where something happens". Operational semantics with evaluation contexts (also known as context-sensitive reduction semantics) avoids this boilerplate by defining a context grammar, which captures the traversal through a term separately. For a syntactic sort e, a context grammar could look like this:

$$\begin{array}{lll} e & ::= & e * e \mid e - e \mid n \\ E & ::= & \square \mid E[\square * e] \mid E[e * \square] \mid E[\square - e] \mid E[n - \square] \end{array}$$

The nonterminal E defines patterns of terms where the box $\square$ represents a placeholder for a term to be plugged in. In this case the contexts define out-of-order evaluation for multiplication, and left-to-right evaluation for subtraction. Operational semantics with evaluation contexts works by first *splitting* a term into a context $E[\ ]$ and a *redex* (ocurring at the placeholder position). When a redex has been reduced, it is *plugged* back into the context and execution continues. The reduction rules only need to be defined for the redexes, thus avoiding a lot of boilerplate.

Tools like Redex implement this process of splitting/plugging on top of abstract syntax definitions. As a result, both the specifications and visualization tools employ s-expression notation to write and display terms. Systems like Rascal and Spoofax support meta-programming with *concrete syntax* (as pioneered by ASF+SDF [15]), which has some distinct benefits:

- Pattern matching with concrete syntax is WYSIWYG: object-level terms in the meta-program read like literal object language expressions.
- If terms are internally represented as concrete syntax trees, then rendering a term to object-language syntax is literally for free; no need for pretty printers.
- Concrete syntax trees can be transformed while maintaining layout information, thus making the display of intermediate and final results more human-friendly.
- If a concrete syntax grammar is already available to obtain a parser for a language, why not reuse it for the definition of a semantics?

CREDEX takes these considerations to heart: it starts with the actual concrete syntax grammar (as defined in Rascal), and then allows it to be modularly extended with a *concrete context grammar*. This second grammar drives the splitting process through the derived parser. Although the link between context-redex decomposition and context-free grammars has been made before [27], as far as the author is aware, CREDEX is the first tool to actually perform decomposition through parsing, and to leverage the potentially ambiguous parse forest for representing non-unique decompositions. Below we introduce CREDEX using an example.

```
syntax Expr
  = Num
  | bracket "(" Expr ")"
  | assoc Expr "*" Expr
  > left Expr "-" Expr
  ;

lexical Num
  = [\-]?[1-9][0-9]* | [0] ;
```

```
syntax E
  = "(" E ")"
  | E "*" Expr
  | Expr "*" E
  | E "-" Expr
  | Num "-" E
  | @redex "(" Num ")"
  | @redex Num "*" Num
  | @redex Num "-" Num;
```

Figure 1 Specifying concrete syntax (left) and concrete evaluation contexts (right).

3 Credex by example

3.1 Defining contexts

The left-hand side of Figure 1 shows a simple expression grammar in Rascal's built-in grammar formalism. The definition of whitespace is omitted, but it is implicitly included inbetween the symbols of a **`syntax`** declaration. An `Expr` is defined as a literal number, a bracketed expression, multiplication (which is **`assoc`**iative), and subtraction (which has lower precedence than multiplication due to the use of `>` instead of `|`). The last line of the grammar defines the lexical syntax of numbers using character classes and regular expressions.

The right-hand side of the figure shows a context grammar for expression contexts, which are conventionally named `E`. The way to read such a grammar is by viewing the occurrences of `E` as directions for tree traversal. So, in the first alternative, the only way to go, is inside the parentheses. For multiplication, the traversal can proceed either down the left-hand side or down the right-hand side, as indicated by two productions having `E` at the left-hand side, or at the right-hand side, respectively. Note that this is a language design decision: we are *defining* here that the evaluation order of multiplication is arbitrary. For subtraction, however, the two productions are asymmetric: only after having evaluated the left-hand side to a number (`Num`) can evaluation continue in the right-hand side. This enforces strict left-to-right evaluation for subtraction.

Finally, the grammar includes three productions annotated with `@redex`, to indicate sub-terms that are "interesting", and can be reduced are part of a step. In other words, these productions describe the sub-trees where the traversal via the `E` nonterminal should stop. Note that the context grammar (trivially) does not generate the original language, because there is no `E`-production to generate a single `Num`, which is a valid expression. This makes intuitive sense, however, since numbers are irreduceable.

3.2 Splitting Through Parsing

Splitting in Credex works as follows: the process starts with a (parsed, non-ambiguous) term over the object language (e.g., of type `Expr`). The term is then unparsed to text, and parsed *again*, but this time over the context grammar (e.g., `E`). This returns a (possibly ambiguous) parse forest, where certain sub-nodes are annotated with `@redex`. Split analyzes the parse forest, and produces a list of pairs corresponding to the contexts and redexes. The context is a parse-tree where the redex sub-node has been replaced with a designated place holder "□". Every ambiguity in the forest adds another context-redex pair.

Let's say we have the term `(1 - 2) * (2 - 3)`. Calling split on this term gives us a non-unique decomposition: 1) `(□) * (2 - 3)`, with redex `1 - 2`, or 2) `(1 - 2) * (□)`, with redex `2 - 3`. This corresponds to the definition of the contexts for multiplication: it's allowed to first evaluate the left-hand side, or the right-hand side.

```
CR red("par", E e, (E)`(<Num n>)`) = {<e, (Expr)`<Num n>`>};
CR red("mul", E e, (E)`<Num n1> * <Num n2>`) = {<e, [Expr]"<toInt(n1) * toInt(n2)>">};
CR red("sub", E e, (E)`<Num n1> - <Num n2>`) = {<e, [Expr]"<toInt(n1) - toInt(n2)>">};
default CR red(str _, E _, Tree _) = {}; // we're stuck or done.

RR applyExpr(Expr e) = apply(#E, #Expr, red, e, {"mul", "sub", "par"});
```

Figure 2 Reduction rules for the expression language.

Changing the example term to `(1 - 2) - (2 - 3)`, however, produces just a single decomposition: `(□) - (2 - 3)`, with redex `1 - 2`, because, according to the context grammar, evaluating subtraction needs to start at the left-hand side. Things are not thát simple, however, because of accidental ambiguities in the context grammar that we are unaware of, but we will address this problem in Section 3.4. Let's first look at how to specify reduction rules.

3.3 Specifying Reduction Rules

Reduction rules are specified as a case-based Rascal function, conventionally named `red` (short for "reduce"). The rules for the expression language are shown in Figure 2. Note that Rascal functions dispatch based on the patterns of their arguments, with the benefit that Credex specifications are modularly extensible by simpling adding additional cases for new combinations of syntax.

The first argument of `red` matches on is a literal string constant, acting as a rule label. The second argument is the context (in this case of type `E`). The third argument employs concrete syntax matching on `E`-contexts, where the part between backticks captures the redex. Note that this pattern is not a string, but tree pattern expressed in the concrete grammar of the object language, where fish-angle brackets are used to introduce (typed) pattern variables, such as `<Num n>`.

The result type of `red`, is `CR` a binary relation type associating a (possibly) modified context to a reduct[1]. Each case of `red` matches on a rule label (e.g., `mul`), the input context, and the redex. Observe that the rules employ concrete syntax matching (the parts between backticks) as defined by the context grammar, and note that the rules only match on syntax annotated with `@redex`. The `mul` and `sub` rules both use the `[Expr]` parsing-operator to create new expressions from the result of multiplication and subtraction, respectively.

The helper function `applyExpr` is used to have a term perform a single step, according to the set of rules identified by the set of labels provided to the generic function `apply`. The `apply` (Figure 3) function uses the reified types (`#E` and `#Expr`) to perform splitting[2]. It then iterates over the set of rule labels, tries invoking the `red` function, and unions the result(s). The result type, `RR`, is a relation from rule-label to terms, capturing which steps the input term `e` could have performed and with what result.

3.4 Eliminating Spurious Ambiguities

Let's return to splitting a term by parsing using the context grammars. In the above examples we got precisely the decompositions that we had wantend: two for the multiplication, and one for subtraction. If we consider the term `1 - 2 - 3`, however, a naïve splitting according to the grammar of Figure 1, will result in two decompositions: `□ - 3` with redex `1 - 2`, and `1 - □`, with redex `2 - 3`, which is incorrect.

[1] In a sense we use the relation type as an option type. We could have used Rascal's `Maybe` type, but the curly braces incur less syntactic noise than `just`-constructors.
[2] Reified types are Rascal's reflection system; think of `#E` as similar to `E.class` in Java.

```
rel[str,Tree] apply(type[Tree] C, type[Tree] T, CR(str,Tree,Tree) red, Tree t, set[str] rules) {
  result = {};
  for (<ctx, rx> ← split(C, T, t)) {
    for(l ← rules) {
      for (<ctx2, rt> ← red(l, ctx, rx)) {
        result += {<l, plug(ctx2, rt)>};
      }
    }
  }
  return result;
}
```

Figure 3 Pseudocode for `apply`.

```
  syntax E
    = "⟨" E "*" Expr "⟩"
    | "⟨" "(" E ")" "⟩"
    | "⟨" Expr "*" E "⟩"
    | "⟨" Num "-" E "⟩"
    | "⟨" E "-" Expr "⟩"
    | @redex "⟨" Num "-" Num "⟩"
    | @redex "⟨" Num "*" Num "⟩"
    | @redex "⟨" "(" Num ")" "⟩";
```

Figure 4 Parenthesization.

The reason is that the associativity (e.g., **`left`**) and priority annotations used to disambiguate the base grammar, do not transfer to the context grammars. The term `1 - 2 - 3` has two derivations, one through `E "-" Expr`, where `E` derives `Num "-" Num`, and one through `Num "-" E`, where `E` again derives `Num "-" Num`. Although the original term had not been ambiguous, it *became* ambiguous when it was unparsed to text, and reparsed over the context-grammar.

A solution lies in how humans disambiguate expressions: by adding parentheses. This time, however, the parentheses are needed everywhere, since we do not know about the meaning of the terms. The process is illustrated in Figure 5. The starting point is a base grammar L.g, which is used to parse an L-program Foo.L, resulting in a parse tree (indicated by triangles). The context grammar, L+Ctx.g extends the base grammar, and is automatically[3] transformed into a *parenthesized* grammar [19], ⟨L+Ctx⟩.g. The parse tree for Foo.L is unparsed with parentheses, *then* reparsed with ⟨L+Ctx⟩.g, leading to a parse forest Foo.⟨L+Ctx⟩; after removal of the parentheses we obtain a parse forest Foo.L+Ctx, as if it had been parsed over L+Ctx.g, but without the spurious ambiguities. The parenthesizing of the example expression context-grammar is shown in Figure 4, using the special parentheses ⟨ and ⟩.

When split decomposes such a parenthesized term using the parenthesized context grammar as the recipe, spurious ambiguities resulting from missing associativity and priority annotations (so-called *horizontal ambiguities* [4]) are avoided. This is illustrated in Table 1, where the result of splitting before and after parenthesization is shown. Splitting the first term, `(1 - 2) * (2 - 3)`, still produces the (desired) two decompositions. But the second term, `1 - 2 - 3`, now produces the one and only correct decomposition after parenthesizing. The specifications are not polluted by parenthesization, however, as the example specifications below will demonstrate.

[3] This is again an application of Rascal's type reflection capabilities.

Table 1 Difference between splitting before and after parenthesizing.

Input	Before	Parenthesized
`(1 - 2) * (2 - 3)`	(1 - 2) * (☐) [2 - 3] (☐) * (2 - 3) [1 - 2]	⟨⟨((⟨☐⟩))⟩ * ⟨((⟨2 - 3⟩))⟩⟩ [1 - 2] ⟨⟨((⟨1 - 2⟩))⟩ * ⟨((⟨☐⟩))⟩⟩ [2 - 3]
`1 - 2 - 3`	☐ - 3 [1 - 2] 1 - ☐ [2 - 3]	⟨⟨☐⟩ - 3⟩ [1 - 2]

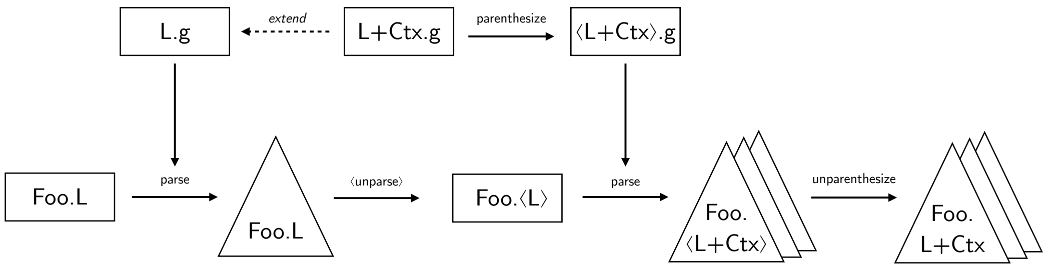

Figure 5 Avoiding spurious ambiguities by parenthesizing context grammars.

4 Example Credex Definitions

4.1 Lambda Calculus

Figure 6 shows the syntax (left) and semantics (right) of the call-by-value lambda calculus. The syntax makes a distinction between expressions (variables, values, and applications), and values (functions, numbers, and the built-in function `+`). The context grammar `E` declares a single context, indicating left-to-right evaluation of a sequence of expressions. Note how `E` "moves" through the sequence by having a prefix of `Value*` and a suffix of `Expr*`. There is only a single redex production: an application of a value to zero or more argument values.

The semantics of lambda calculus is defined using two reduction rules, one for addition, and one for function application. The first rule (`+`) simply performs the addition of two numbers and produces an equivalent expression as a result. The rule for β-reduction substitutes the argument for the parameter of the lambda abstraction using the `subst` function. In turn, this `subst` function (not shown) reuses the generic capture-avoiding substitution facilities of Credex (inspired by [6]).

Consider the term `((λ (x) (+ x 2)) ((λ (x) (+ x 2)) 1))`. Credex comes with a helper function to display the execution trace. The result of the example term is as follows:

```
((λ (x) (+ x 2)) ((λ (x) (+ x 2)) 1))
 └ ((λ (x) (+ x 2)) ((λ (x) (+ x 2)) 1)) —β→ ((λ (x) (+ x 2)) (+ 1 2))
    └ ((λ (x) (+ x 2)) (+ 1 2)) —+→ ((λ (x) (+ x 2)) 3)
       └ ((λ (x) (+ x 2)) 3) —β→ (+ 3 2)
          └ (+ 3 2) —+→ 5
```

Note that the rendered terms are rendered in the object syntax of the lambda calculus itself[4].

Here's term to illustrate capture-avoiding substitution: `((λ (x) (λ (y) x)) (λ (z) y))`. The variable `y` is free in the top-level argument, so a naïve syntactic substitution would cause capturing. But the result is as follows:

[4] The sequential trace is rendered in an indented fashion to allow for branching in the trace; see Section 4.2.

```
syntax Expr
  = var: Id
  | val: Value
  | app: "(" Expr+ ")";

syntax Value
  = lam: "(" "λ" "(" Id ")" Expr ")"
  | \num: Num
  | add: "+";
```

```
syntax E
  = "(" Value* E Expr* ")"
  | @redex "(" Value Value* ")";

CR red("+", E e, (E)‘(+ <Num n1> <Num n2>)‘)
  = {<e, [Expr]"<toInt(n1) + toInt(n2)>">};

CR red("β", E e, (E)‘((λ (<Id x>) <Expr b>) <Value v>)‘)
  = {<e, subst((Expr)‘<Id x>‘, (Expr)‘<Value v>‘, b)>};
```

Figure 6 Syntax (left) and semantics (right) of the call-by-value lambda calculus.

```
((λ (x) (λ (y) (+ y x))) (λ (z) y))
 └ ((λ (x) (λ (y) (+ y x))) (λ (z) y)) —β→ (λ (y_) (+ y_ (λ (z) y)))
```

Note how both occurrences of `y` are renamed to avoid capture.

The substitution facility offered by Credex is parameterized by name analysis of the object language using an embedded Rascal DSL to modularly and concisely express binding relations between declarations and variables. The resulting reference graph is used to detect capturing and rename variables accordingly, similar to the technique of Erdweg et al. [6].

Since Rascal's module system allows extension of both context-free grammars and functions specified in the pattern-based dispatch style (like the function `red`), semantic specifications using Credex can be modularly extended as well. As an example, consider the extension of the lambda calculus with `call/cc` . This would consist of a module extending the base-level semantics containing the following code:

```
syntax Value = "call/cc";

CR red("callcc", E e, (E)‘(call/cc <Value v>)‘)
  = {<e, (Expr)‘(<Value v> (λ (<Id x>) <Expr cc>))‘>}
  when
    Id x := fresh((Id)‘x‘, e),
    Expr cc := plug(#Expr, e, (Expr)‘<Id x>‘);
```

The first line extends the syntax of values with the primitive `call/cc` (it is not a redefinition of `Value`). The reduction rule labeled `callcc` extends the `red` function with the semantics of call-with-current-continuation. Additionally, this shows how contexts can be embedded into terms to model continuations. The result of calling `call/cc` with a function as argument is to call that function with a lambda modeling the current continuation, `cc`, which is constructed by plugging a fresh variable `x` into the `E` context `e`. The helper function `fresh` creates an unique new variable `x` relative to the context itself. Here's a trace showing `call/cc` in action:

```
(+ 1 (call/cc (λ (k) (k 2))))
 └ (+ 1 (call/cc (λ (k) (k 2)))) —callcc→ (+ 1 ((λ (k) (k 2)) (λ (x) (+ 1 x))))
    └ (+ 1 ((λ (k) (k 2)) (λ (x) (+ 1 x)))) —β→ (+ 1 ((λ (x) (+ 1 x)) 2))
       └ (+ 1 ((λ (x) (+ 1 x)) 2)) —β→ (+ 1 (+ 1 2))
          └ (+ 1 (+ 1 2)) —+→ (+ 1 3)
             └ (+ 1 3) —+→ 4
```

4.2 Imp: a Simple Imperative Language

Imp is a simple imperative language consisting of arithmetic expressions, boolean expressions, and statements (if-then-else, while, assignment, skip). Its semantics requires a mutable store, which is modeled syntactically as shown in Figure 7. `State` defines the store as a sequence of zero or more `VarInt` pairs (mapping identifiers to integers), separated by commas. A configuration `Conf` is defined as statement `Stmt` under a certain `State`. Configurations are the

```
syntax State = "[" {VarInt ","}* "]";
syntax VarInt = Id "↦" Int;
syntax Conf = State "⊢" Stmt;

syntax C = State "⊢" S;
```

```
syntax S
= Id ":=" A
| seq: S ";" Stmt
| "if" B "then" Stmt "else" Stmt "fi"
| @hole "skip" ";" Stmt
| @hole "if" Bool "then" Stmt "else" Stmt "fi"
| @hole "while" BExp "do" Stmt "od"
| @hole Id ":=" Int;
```

Figure 7 Stores and configurations (left) and statement contexts (right).

```
CR red("seq", C c, (S)`skip; <Stmt s2>`) = {<c, s2>};
CR red("if-true", C c, (S)`if true then <Stmt s1> else <Stmt s2> fi`) = {<c, s1>};
CR red("if-false", C c, (S)`if false then <Stmt s1> else <Stmt s2> fi`) = {<c, s2>};
CR red("while", C c, (S)`while <BExp b> do <Stmt s> od`)
 = {<c, (Stmt)`if <BExp b> then <Stmt s>; while <BExp b> do <Stmt s> od else skip fi`>};
CR red("assign", C c, (S)`<Id x> := <Int i>`) = {<c[state=s2], (Stmt)`skip`>}
 when isDefined(x, c.state), State s2 := update(x, i, c.state);
```

Figure 8 Statement reduction rules for the simple, imperative Imp language.

top-level terms that will be rewritten. The rule for context `C` simply declares that traversal should always go into the statement part, modeled by context `S`, which further defines the evaluation order of statements, where `A` and `B` represent contexts for arithmetic expressions and boolean expressions respectively.

The reduction rules for the imperative fragment of Imp are shown in Figure 8. The rule for sequencing states that the `skip` statement can be skipped, and execution proceeds with the right-hand side `s2`. The rules for if-then-else reduce to their respective branches based on value of their conditions. The semantics of while-loops is expressed in terms of if-then-else and another while-loop. Finally, assignment to a variable `x` updates the context *itself*, to record the updated value of `x` in the current state, and then reduces to skip. Here's the reduction trace of `x := 1; y := x + 2; if x <= y then x := x + y else y := 0 fi`:

```
[x↦0,y ↦0] ⊢ x:=1; y:=x+2; if x<=y then x:=x+y else y:=0 fi
 └ [x↦0,y ↦0] ⊢ x:=1; y:=x+2; if x<=y then x:=x+y else y:=0 fi —assign→ [x ↦ 1, y ↦0] ⊢ skip; y:=x+2; if x<=y then x:=x+y else y:=0 fi
   └ [x ↦ 1, y ↦0] ⊢ skip; y:=x+2; if x<=y then x:=x+y else y:=0 fi —seq→ [x ↦ 1, y ↦0] ⊢ y:=x+2; if x<=y then x:=x+y else y:=0 fi
     └ [x ↦ 1, y ↦0] ⊢ y:=x+2; if x<=y then x:=x+y else y:=0 fi —lookup→ [x ↦ 1, y ↦0] ⊢ y:=1+2; if x<=y then x:=x+y else y:=0 fi
       └ [x ↦ 1, y ↦0] ⊢ y:=1+2; if x<=y then x:=x+y else y:=0 fi —add→ [x ↦ 1, y ↦0] ⊢ y:=3; if x<=y then x:=x+y else y:=0 fi
         └ [x ↦ 1, y ↦0] ⊢ y:=3; if x<=y then x:=x+y else y:=0 fi —assign→ [x ↦ 1, y ↦ 3] ⊢ skip; if x<=y then x:=x+y else y:=0 fi
           └ [x ↦ 1, y ↦ 3] ⊢ skip; if x<=y then x:=x+y else y:=0 fi —seq→ [x ↦ 1, y ↦ 3] ⊢ if x<=y then x:=x+y else y:=0 fi
             └ [x ↦ 1, y ↦ 3] ⊢ if x<=y then x:=x+y else y:=0 fi —lookup→ [x ↦ 1, y ↦ 3] ⊢ if 1<=y then x:=x+y else y:=0 fi
               └ [x ↦ 1, y ↦ 3] ⊢ if 1<=y then x:=x+y else y:=0 fi —lookup→ [x ↦ 1, y ↦ 3] ⊢ if 1<=3 then x:=x+y else y:=0 fi
                 └ [x ↦ 1, y ↦ 3] ⊢ if 1<=3 then x:=x+y else y:=0 fi —leq→ [x ↦ 1, y ↦ 3] ⊢ if true then x:=x+y else y:=0 fi
                   └ [x ↦ 1, y ↦ 3] ⊢ if true then x:=x+y else y:=0 fi —if-true→ [x ↦ 1, y ↦ 3] ⊢ x:=x+y
                     ├ [x ↦ 1, y ↦ 3] ⊢ x:=x+y —lookup→ [x ↦ 1, y ↦ 3] ⊢ x:=1+y
                     │ └ [x ↦ 1, y ↦ 3] ⊢ x:=1+y —lookup→ [x ↦ 1, y ↦ 3] ⊢ x:=1+3
                     │   └ [x ↦ 1, y ↦ 3] ⊢ x:=1+3 —add→ [x ↦ 1, y ↦ 3] ⊢ x:=4
                     │     └ [x ↦ 1, y ↦ 3] ⊢ x:=4 —assign→ [x ↦ 4, y ↦ 3] ⊢ skip
                     └ [x ↦ 1, y ↦ 3] ⊢ x:=x+y —lookup→ [x ↦ 1, y ↦ 3] ⊢ x:=x+3
                       └ [x ↦ 1, y ↦ 3] ⊢ x:=x+3 —lookup→ [x ↦ 1, y ↦ 3] ⊢ x:=1+3
                         └ [x ↦ 1, y ↦ 3] ⊢ x:=1+3 —add→ [x ↦ 1, y ↦ 3] ⊢ x:=4
                           └ [x ↦ 1, y ↦ 3] ⊢ x:=4 —assign→ [x ↦ 4, y ↦ 3] ⊢ skip
```

Note how the concrete syntax for the store naturally transfers to rendering of terms. Note also the split in the execution trace, since Imp does not enforce an evaluation order for addition. Both branches evaluate to the same configuration, as expected.

4.3 QL: a Language for Questionnaires

The Questionnaire Language (QL) [7,8] is a DSL with a non-standard execution model. It interesting for three reasons: first, it models an event-based system, a questionnaire form where users interactively change inputs of values. Second, QL allows declare-after-use of questions: a computed question may refer to the value of another question occurring *later*

```
// expressions are immediately evaluated to values
CR red("eval", C c, (E)‘<Expr e>‘) = {<c, (Expr)‘<Value val>‘>}
when
  value v := eval(e, c.ui), Value val := [Value]"<v>";

// a user action updates the ui state and then expands to a block of statements
// to reconcile the UI with the consequences of the update
CR red("update", C c, (S)‘update(<Id x>, <Value v>)‘)
  = {<c[ui=updateVal(c.ui, x, v)], makeBlock(c.qs, c.ui, x)>};

// dealing with unit of statement sequencing
CR red("done", C c, (S)‘{ { } <Stmt* s2>}‘) = {<c, (Stmt)‘{ <Stmt* s2>}‘>};

// updating a question to a value that is the same as the old value is a no-op
CR red("val-same", C c, (S)‘val(<Id x>, <Value v>, <Value old>)‘) = {<c, (Stmt)‘{}‘>}
  when old == v;

// otherwise, updating is equivalent to a user action
CR red("val-diff", C c, (S)‘val(<Id x>, <Value v>, <Value old>)‘)
  = {<c, (Stmt)‘update(<Id x>, <Value v>)‘>} when old != v;

// updating visibility modifies the UI
CR red("vis", C c, (S)‘vis(<Id x>, <Bool b>)‘) = {<c[ui=updateVis(c.ui, x, b)], (Stmt)‘{}‘>};
```

Figure 9 Reduction rules for QL.

in the form. This requires a fixpoint computation in the style of spreadsheets. Finally, the consequences of user actions are not limited to the state, but also affect the UI; in other words, the semantic domain is complex.

The core reduction rules for QL are shown in Figure 9. First, QL features expressions for definining computed questions and conditional visibility of questions. Since such expressions are semantically rather uninteresting, they are evaluated in one step, using an existing interpreter (rule `eval`). This shows how CREDEX integrates well with other language engineering components within Rascal.

The semantics starts off with a user action `update(`x`, `v`)` modifying the value of a (non-computed) question. The reduction rule (`update`) for this redex expands to a sequence of derived statements (using `makeBlock`; not shown), representing the "update plan" according to the questionaire. Modifications to the state check whether a value did change, and if so, trigger new `update`-statements (rules lstlineval-same and `val-diff`). As result, execution continues till the state reaches a fixed point.

5 Instead of conclusion

In this short paper we have presented CREDEX, a library in Rascal for defining small-step semantics. Next to the basics detailed above, CREDEX comes with browser-based tools for visualizing execution graphs, interactive step-wise debugging, and functionality for randomized sentence generation. Further research directions include: a precise comparison to the matching algorithm of Redex [14], and investigating how CREDEX can be combined with Rascal's typechecking library, TYPEPAL.

There are many tools out there to define and execute formal semantics (e.g., [1–3,9,20,23–26], and others). CREDEX is unique in that it takes the basic execution model of Redex (but using concrete syntax), it is modular from the start, it employs parsing for context-redex decomposition, and it integrates well with the Rascal language workbench. As such, it brings semantics engineering a small step closer to language engineering.

References

1 Yves Bertot. A short presentation of coq. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 12–16, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

2 L. Thomas van Binsbergen, Peter D. Mosses, and Neil Sculthorpe. Executable component-based semantics. *Journal of Logical and Algebraic Methods in Programming*, 103:184–212, 2019. doi:10.1016/j.jlamp.2018.12.004.

3 L. Thomas van Binsbergen, Neil Sculthorpe, and Peter D. Mosses. Tool support for component-based semantics. In *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, pages 8–11, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2892664.2893464.

4 Claus Brabrand, Robert Giegerich, and Anders Møller. Analyzing ambiguity of context-free grammars. *Science of Computer Programming*, 75(3):176–191, 2010. doi:10.1016/j.scico.2009.11.002.

5 Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S. Adve, and Grigore Roşu. A Complete Formal Semantics of x86-64 User-Level Instruction Set Architecture. In *PLDI'19*, pages 1133–1148. ACM, June 2019. doi:10.1145/3314221.3314601.

6 Sebastian Erdweg, Tijs van der Storm, and Yi Dai. Capture-avoiding and hygienic program transformations. In Richard E. Jones, editor, *ECOOP'14*, volume 8586 of *Lecture Notes in Computer Science*, pages 489–514. Springer, 2014. doi:10.1007/978-3-662-44202-9_20.

7 Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido H. Wachsmuth, and Jimi van der Woning. The state of the art in language workbenches. In Martin Erwig, Richard F. Paige, and Eric Van Wyk, editors, *Software Language Engineering*, pages 197–217, Cham, 2013. Springer International Publishing.

8 Sebastian Erdweg, Tijs van der Storm, Markus Völter, Laurence Tratt, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. Evaluating and comparing language workbenches: Existing results and benchmarks for the future. *Computer Languages, Systems & Structures*, 44:24–47, 2015. Special issue on the 6th and 7th International Conference on Software Language Engineering (SLE 2013 and SLE 2014). doi:10.1016/j.cl.2015.08.007.

9 Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics engineering with PLT Redex*. MIT Press, 2009.

10 Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992. doi:10.1016/0304-3975(92)90014-7.

11 Jan Heering, Paul Klint, et al. Semantics of programming languages: A tool-oriented approach. *SIGPLAN Notices*, 35(3):39–48, 2000.

12 Lennart C.L. Kats and Eelco Visser. The spoofax language workbench: Rules for declarative specification of languages and ides. *SIGPLAN Not.*, 45(10):444–463, October 2010. doi:10.1145/1932682.1869497.

13 Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt, and Robert Bruce Findler. Run your research: on the effectiveness of lightweight mechanization. In John Field and Michael Hicks, editors, *POPL'12*, pages 285–296. ACM, 2012. doi:10.1145/2103656.2103691.

14 Casey Klein, Jay A. McCarthy, Steven Jaconette, and Robert Bruce Findler. A semantics for context-sensitive reduction semantics. In *APLAS'11*, volume 7078 of *Lecture Notes in Computer Science*, pages 369–383. Springer, 2011. doi:10.1007/978-3-642-25318-8_27.

15 Paul Klint. A meta-environment for generating programming environments. *ACM Trans. Softw. Eng. Methodol.*, 2(2):176–201, 1993. doi:10.1145/151257.151260.

16 Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 168–177. IEEE Computer Society, 2009. doi:10.1109/SCAM.2009.28.

17 Jacob Matthews and Robert Bruce Findler. An operational semantics for Scheme. *Journal of Functional Programming*, 18(01):47–86, 2008.

18 Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. *A Visual Environment for Developing Context-Sensitive Term Rewriting Systems*, pages 301–311. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004. doi:10.1007/978-3-540-25979-4_21.

19 Robert McNaughton. Parenthesis grammars. *J. ACM*, 14(3):490–500, July 1967. doi:10.1145/321406.321411.

20 Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable engineering of real-world semantics. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 175–188, New York, NY, USA, 2014. Association for Computing Machinery. doi:10.1145/2628136.2628143.

21 Gordon D. Plotkin. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60-61:3–15, 2004. Structural Operational Semantics. doi:10.1016/j.jlap.2004.03.009.

22 Gordon D. Plotkin. A structural approach to operational semantics. *The Journal of Logic and Algebraic Programming*, 60-61:17–139, 2004. Structural Operational Semantics. doi:10.1016/j.jlap.2004.05.001.

23 Grigore Rosu. K – a semantic framework for programming languages and formal analysis tools. In Doron Peled and Alexander Pretschner, editors, *Dependable Software Systems Engineering*, NATO Science for Peace and Security. IOS Press, 2017.

24 Carsten Schürmann. The twelf proof assistant. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, pages 79–83, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

25 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *SIGPLAN Not.*, 42(9):1–12, October 2007. doi:10.1145/1291220.1291155.

26 Vlad A. Vergu, Pierre Neron, and Eelco Visser. Dynsem: A DSL for dynamic semantics specification. In *RTA'15*, pages 365–378, 2015. doi:10.4230/LIPIcs.RTA.2015.365.

27 Yong Xiao, Amr Sabry, and Zena M. Ariola. From syntactic theories to interpreters: Automating the proof of unique decomposition. *Higher Order Symbol. Comput.*, 14(4):387–409, December 2001. doi:10.1023/A:1014408032446.