A Semantics for Context-Sensitive Reduction Semantics

Casey Klein[1], Jay McCarthy[2], Steven Jaconette[1], and Robert Bruce Findler[1]

[1] Northwestern University
[2] Brigham Young University

Abstract. This paper explores the semantics of the meta-notation used in the style of operational semantics introduced by Felleisen and Hieb. Specifically, it defines a formal system that gives precise meanings to the notions of contexts, decomposition, and plugging (recomposition) left implicit in most expositions. This semantics is not naturally algorithmic, so the paper also provides an algorithm and proves a correspondence with the declarative definition.

The motivation for this investigation is PLT Redex, a domain-specific programming language designed to support Felleisen-Hieb-style semantics. This style of semantics is the de-facto standard in operational semantics and, as such, is widely used. Accordingly, our goal is that Redex programs should, as much as possible, look and behave like those semantics. Since Redex's first public release more than seven years ago, its precise interpretation of contexts has changed several times, as we repeatedly encountered reduction systems that did not behave according to their authors' intent. This paper describes the culimation of that experience. To the best of our knowledge, the semantics given here accommodates even the most complex uses of contexts available.

1 Introduction

The dominant style of operational semantics in use today has at its heart the notion of a context that controls where evaluation occurs. These contexts allow the designer of a reduction semantics to factor the definition of a calculus into one part that specifies the atomic steps of computation and a second part that controls where these steps may occur. This factoring enables concise specification, e.g., that a language is call-by-value or call-by-name or call-by-need (Ariola and Felleisen 1997), that `if` expressions must evaluate the test position before the branches, and even that exceptions, continuations, and state (Felleisen and Hieb 1992) behave in the expected ways, all without cluttering the rules that describe the atomic steps of computation.

Unfortunately, the precise meaning of context decomposition has not been nailed down in a way that captures its diverse usage in the literature. Although an intuitive definition is easy to understand from a few examples, this intuition does not cover the full power of contexts. For example, which terms match the pattern $C[e]$ from this language, in which values and contexts are mutually referential?

$$\begin{aligned} C &::= [\,] \mid (v\ C) \mid (C\ e) \\ e &::= v \mid \mathsf{x} \mid (e\ e) \\ v &::= (\lambda\ (\mathsf{x})\ e) \mid (\mathsf{cont}\ C) \end{aligned}$$

And which terms match this bizarre, small language?

$$C ::= C[(\mathsf{f}\ [])] \mid []$$

To remedy this lack, we have developed a semantics for matching and reduction that not only supports these exotic languages but also captures the intuitive meanings of countless existing research papers. This semantics does not assume explicit language-specific definitions of plugging and decomposition, since most expositions leave these concepts implicit.

Our motivation for studying context-sensitive matching is its implementation in the domain-specific programming language Redex (Felleisen et al. 2010; Matthews et al. 2004). Redex is designed to support the semantics engineer with a lightweight toolset for operational semantics and related formalisms. Specifically, Redex supports rapid prototyping of context-sensitive operational semantics, random testing, automatic typesetting, and, via its embedding in Racket, access to a large palette of standard programming tools. Redex is widely used, having supported several dozen research papers as well as the latest Scheme standard (Sperber et al. 2007) and a number of larger models, including one of the Racket virtual machine (Klein et al. 2010).

In keeping with the spirit of Redex, we augment a standard proof-based validation of our work with testing. More concretely, in addition to proving a correspondence between a specification of context-sensitive matching and an algorithm for that specification, we have conducted extensive testing of the semantics, using a Redex model of Redex (there is little danger of meta-circularity causing problems, as the embedding uses a modest subset of Redex's functionality—notably, no contexts or reduction relations). This model allows us to test that our semantics gives the intended meanings to interesting calculi from the literature, something that would be difficult to prove.

The remainder of this paper builds up an intuitive understanding of what contexts are and how they are used via a series of examples, gives a semantics for Redex's rewriting system, and discusses an algorithm to implement the semantics.

2 Matching and Contexts

This section introduces the notion of contexts and explains through a series of examples how matching works in their presence. Each example comes with a lesson that informs the design of our context-sensitive reduction semantics semantics.

In its essence, a pattern of the form $C[e]$ matches an expression when the expression can be split into two parts, an outer part (the context) that matches C and an inner part that matches e. The outer part marks where the inner part appears with a hole, written $[]$. In other words, when thinking of an expression as a tree, matching against $C[e]$ finds some subtree of the expression that matches e, and then replaces that sub-term with a hole to build a new expression in such a way that the new expression matches C.

$$\begin{aligned} a &::= (+\ a\ a) \mid \mathit{number} \\ C &::= (+\ C\ a) \mid (+\ a\ C) \mid [] \end{aligned}$$

$$C[(+\ \mathit{number}_1\ \mathit{number}_2)] \longrightarrow C[\Sigma[[\mathit{number}_1, \mathit{number}_2]]]$$

Figure 1: Arithmetic Expressions

To get warmed up, consider figure 1. In this language a matches addition expressions and C matches contexts for addition expressions. More precisely, C matches an addition expression that has exactly one hole. For example, the expression (+ 1 2) matches $C[a]$ three ways, as shown in figure 2. Accordingly, the reduction relation given in figure 1 reduces addition expressions wherever they appear in an expression, e.g., reducing (+ (+ 1 2) (+ 3 4)) to two different expressions, (+ 3 (+ 3 4)) and (+ (+ 1 2) 7). This example tells us that our context matching semantics must support multiple decompositions for any given term.

A common use of contexts is to restrict the places where reduction may occur in order to model a realistic programming language's order of evaluation. Figure 3 gives a definition of E that enforces call-by-value left-to-right order of evaluation. For example, consider this nested set of function calls, $((f\ x)\ (g\ y))$, in which the result of $(g\ y)$ is passed to the result of $(f\ x)$. It decomposes into the context $([]\ (g\ y))$, allowing evaluation in the first position of the application. It does not, however, decompose into the context $((f\ x)\ [])$, since the grammar for E allows the hole to appear in the argument position of an application expression only when the function position is already a value. Accordingly, the reduction system insists that the call to f happens before the call to g. This example tells us that our semantics for decomposition must be able to support multiple different ways to decompose each expression form, depending on the subexpressions of that form (application expressions in this case).

C = []	a = (+ 1 2)
C = (+ [] 2)	a = 1
C = (+ 1 [])	a = 2

Figure 2: Example Decomposition

Contexts can also be used in clever ways to model the call-by-need λ-calculus. Like call-by-name, call-by-need evaluates the argument to a function only if the value is actually needed by the function's body. Unlike call-by-name, each function argument is evaluated at most once. A typical implementation of a language with call-by-need uses state to track if an argument has been evaluated, but it is also possible to give a direct explanation, exploiting contexts to control where evaluation occurs.

$$\begin{aligned} e &::= (e\ e) \mid x \mid v \\ v &::= (\lambda\ (x)\ e) \mid \texttt{+1} \mid \mathit{number} \\ E &::= (E\ e) \mid (v\ E) \mid [] \end{aligned}$$

Figure 3: λ-calculus

Figure 4 shows the contexts from Ariola and Felleisen (1997)'s model of call-by-need. The first three of E's alternatives are standard, allowing evaluation in the argument of the +1 primitive, as well as in the function position of an application (regardless of what appears in the argument position). The fourth alternative allows evaluation in the body of a λ-expression that is in the function position of an application. Intuitively, this case says that once we have determined the function to be applied, then we can begin to evaluate its body. Of course, the function may eventually need its argument, and at that point, the final alternative comes into play. It says that when an applied function needs its argument, then that argument may be evaluated.

$$\begin{aligned} E ::= {} & [] \mid (\texttt{+1}\ E) \mid (E\ e) \\ & \mid ((\lambda\ (x)\ E)\ e) \\ & \mid ((\lambda\ (x)\ E[x])\ E) \end{aligned}$$

Figure 4: Call-by-need Contexts

As an example, the expression ((λ (x) (+1 1)) (+1 2)) reduces by simplifying the body of the λ-expression to 2, without reducing the argument, because it decomposes into this context ((λ (x) []) (+1 2)) using the fourth alternative of E. In contrast, ((λ (x) (+1 x)) (+1 2)) reduces to ((λ (x) (+1 x)) 3) because the body of the λ-expression decomposes into the context (+1 []) with x in the hole, and thus the entire expression decomposes into the context ((λ (x) (+1 x)) []). This use of contexts tells us that our semantics must be able to support a sophisticated form of nesting, namely that sometimes a decomposition must occur in one part of a term in order for a decomposition to occur in another.

When building a model of first-class continuations, there is an easy connection to make, namely that an evaluation context is itself a natural representation for a continuation. That is, at the point that a continuation is grabbed, the context in which it is grabbed is the continuation. Figure 5 extends the left-to-right call-by-value model in figure 3 with support for continuations. It adds call/cc, the operator that grabs a continuation, and the new value form (cont E) that represents a continuation and can be applied to invoke the continuation.

$$
\begin{array}{l}
v ::= \text{....} \mid \mathsf{call/cc} \mid (\mathsf{cont}\ E) \\
E[(\mathsf{call/cc}\ v)] \longrightarrow E[(v\ (\mathsf{cont}\ E))] \\
E_1[((\mathsf{cont}\ E_2)\ v)] \longrightarrow E_2[v]
\end{array}
$$

Figure 5: Continuations

For example, the expression (+1 (call/cc (λ (k) (k 2)))) reduces by grabbing a continuation. In this model that continuation is represented as (cont (+1 [])), which is then applied to call/cc's argument in the original context, yielding the expression (+1 ((λ (k) (k 2)) (cont (+1 [])))). The next step is to substitute for k, which yields the expression (+1 ((cont (+1 [])) 2)). This expression has a continuation value in the function position of an application, making the next step invoke the continuation. So, we can simply replace the context of the continuation invocation with the context inside the continuation, plugging the argument passed to the continuation in the hole, yielding (+1 2). This reduction system tells us that our context decomposition semantics must be able to support contexts that appear in a term that play no part in any decomposition (and yet must still match a specified pattern, such as E).

Generalizing from ordinary continuations to delimited continuations is simply a matter of factoring the contexts into two parts, one that may contain prompts and one that may not. Figure 6 shows one way to do this, as an extension of the call-by-value lambda calculus from figure 3.

$$
\begin{array}{rl}
e & ::= \text{....} \mid (\#\ e) \\
v & ::= \text{....} \mid \mathsf{call/comp} \mid (\mathsf{comp}\ M) \\
M & ::= (M\ e) \mid (v\ M) \mid [] \\
E & ::= M \mid E[(\#\ M)]
\end{array}
$$

$$
\begin{array}{l}
E[(\#\ M[(\mathsf{call/comp}\ v)])] \longrightarrow \\
E[(\#\ M[(v\ (\mathsf{comp}\ M))])]
\end{array}
$$

Figure 6: Delimited Continuations

The non-terminal E matches an arbitrary evaluation context and M matches an evaluation context that does not contain any prompt expressions. Accordingly, the rule for grabbing a continuation exploits this factoring to record only the portion of the context between the call to call/comp and the nearest enclosing prompt.

The interesting aspect of this system is how E refers to M and how that makes it difficult to support an algorithm that matches E. For all of the example systems in this section so far, a matching algorithm can match a pattern of the form $C[e]$ by attempting to match C against the entire term and, once a match has been found, attempting to match what appeared at the hole against e. With E, however, this leads to an infinite loop because E expands to a decomposition that includes E in the first position.[3]

$$\begin{array}{l} C ::= C[(\mathsf{f}\ [])] \mid [] \\ C ::= (\mathsf{f}\ C) \mid [] \end{array}$$

Figure 7: Wacky Context

A simple fix that works for the delimited continuations example is to backtrack when encountering such cycles; that fix, however, does not work for the first definition of C given in figure 7. Specifically, C would match only $[]$ with an algorithm that treats that cycle as a failure to match, but the context $(f\ [])$ should match C, and more generally, the two definitions of C in figure 7 should be equivalent.

3 A Semantics for Matching

$$\begin{array}{rcl} C & ::= & \mathsf{hole} \\ & \mid & (\mathsf{left}\ C\ t) \\ & \mid & (\mathsf{right}\ t\ C) \\ t & ::= & a \mid \mathsf{hole} \\ & \mid & (\mathsf{cons}\ t\ t) \\ & \mid & (\mathsf{left}\ t\ t) \\ & \mid & (\mathsf{right}\ t\ t) \\ p & ::= & a \mid \mathsf{hole} \\ & \mid & (\mathsf{cons}\ p\ p) \\ & \mid & (\mathsf{name}\ x\ p) \\ & \mid & (\mathsf{nt}\ n) \\ & \mid & (\mathsf{in\text{-}hole}\ p\ p) \\ a & \in & \mathit{Literals} \\ x & \in & \mathit{Variables} \\ n & \in & \mathit{Non\text{-}Terminals} \end{array}$$

Figure 8: Patterns and Terms

This section formalizes the notion of matching used in the definitions of the example reduction systems in section 2. For ease of presentation, we stick to the core language of patterns and terms in figure 8. Redex supports a richer language of patterns (notably including a notion of Kleene star), but this core captures an essence suitable for explaining the semantics of matching.

Ignoring embedded contexts, a term t is simply a binary tree where leaf nodes are atoms a and interior nodes are constructed with cons. A context C is similarly a binary tree, but with a distinguished path (marked with left and right) from the root of the context to its hole.

Contexts are generated by decomposition and represent single-holed contexts. Although the hole can appear multiple times in a single term, such terms represent expressions that contain multiple, independently pluggable contexts.

Patterns p take one of six forms. Atomic patterns a and the hole pattern match only themselves. A pattern $(\mathsf{name}\ x\ p)$ binds the pattern variable x to the term matched by p. Repeated pattern variables force the corresponding sub-terms to be identical. A pattern $(\mathsf{nt}\ n)$ matches terms that match any of the alternatives of the non-terminal n (defined outside the pattern). We write decomposition patterns $p_1[p_2]$ using a separate keyword for clarity: $(\mathsf{in\text{-}hole}\ p_1\ p_2)$. Finally, interior nodes are matched by the pattern $(\mathsf{cons}\ p_1\ p_2)$, where p_1 and p_2 match the corresponding sub-terms.

[3] Some find the equivalent, non-problematic grammar $E ::= M \mid M[(\#\ E)]$ clearer. At least one author of the present paper (who has spent a considerable amount of time hacking on Redex's implementation, no less), however, does not and was surprised when Redex failed to terminate on a similar example. We have also received comments from Redex users who were surprised by similar examples, suggesting that Redex should support such definitions.

For example, the left-hand side of the reduction rule in figure 1 corresponds to the following pattern, where the literal empty is used for the empty sequence and the pattern number matches literal numbers:

```
(in-hole (name C (nt C))
         (cons +
               (cons (name number_1 number)
                     (cons (name number_2 number)
                           empty))))
```

Figure 9 gives a semantics for patterns via the judgment form $G \vdash t : p \mid b$, which defines when the pattern p matches the term t. The grammar G is a finite map from non-terminals to sets of patterns. The bindings b is a finite map from pattern variables to terms showing how the pattern variables of p can be instantiated to yield t. The $G \vdash t : p \mid b$ judgment relies on an auxiliary judgment $G \vdash t = C[t'] : p \mid b$ that performs decompositions. Specifically, it holds when t can be decomposed into a context C that matches p and contains the sub-term t' at its hole.

Many of the rules for these two judgment forms rely on the operator $\sqcup$. It combines two mappings into a single one by taking the union of their bindings, as long as the domains do not overlap. If the domains do overlap, then the corresponding ranges must be the same; otherwise $\sqcup$ is not defined. Accordingly, rules that use $\sqcup$ apply only when $\sqcup$ is well-defined.

The $G \vdash t : p \mid b$ rules are organized by the structure of p. The atom and hole rules produce an empty binding map because those pattern contain no pattern variables. The name rule matches p with t and produces a map extended with the binding (x, t). The nt rule applies if any of the non-terminal's alternatives match. The scope of an alternative's pattern variables is limited to that alternative, and consequently, the nt rule produces an empty binding map. The cons rule matches the sub-terms and combines the resulting sets of bindings. The in-hole rule uses the decomposition judgment form to find a decomposition and checks that the term in the hole matches p_2.

The rules for the $G \vdash t = C[t'] : p \mid b$ form are also organized around the pattern. The hole decomposition rule decomposes any term t into the empty context and t itself. The first of two cons decomposition rules applies when a decomposition's focus may be placed within a pair's left sub-term. This decomposition highlights the same sub-term t_1' as the decomposition of t_1 does, but places it within the larger context (left C t_2). The second of the cons decomposition rules does the same for the pair's right sub-term. The nt decomposition rule propagates decompositions but, as in the corresponding matching rule, ignores binding maps.

The in-hole decomposition rule performs a nested decomposition. Nested decomposition occurs, for example, when decomposing according to call-by-need evaluation contexts (see the last production in figure 4). The in-hole rule decomposes t into a composed context C_1 ++ C_2 and a sub-term t', where p_1 and p_2 match C_1 and C_2 respectively. The definition of context composition (figure 9, bottom-right) follows the path in C_1. The name decomposition rule is similar to the corresponding matching rule, but it introduces a binding to the context that is matched, not the entire term.

$$\boxed{G \vdash t : p \mid b} \quad \frac{}{G \vdash a : a \mid \varnothing} \qquad \frac{}{G \vdash \mathsf{hole} : \mathsf{hole} \mid \varnothing}$$

$$\frac{G \vdash t : p \mid b}{G \vdash t : (\mathsf{name}\ x\ p) \mid \{(x, t)\} \sqcup b}$$

$$\frac{p \in G(n) \qquad G \vdash t : p \mid b}{G \vdash t : (\mathsf{nt}\ n) \mid \varnothing}$$

$$\frac{G \vdash t_1 : p_1 \mid b_1 \qquad G \vdash t_2 : p_2 \mid b_2 \qquad k \in \{\mathsf{cons}, \mathsf{left}, \mathsf{right}\}}{G \vdash (k\ t_1\ t_2) : (\mathsf{cons}\ p_1\ p_2) \mid b_1 \sqcup b_2}$$

$$\frac{G \vdash t_1 = C[t_2] : p_1 \mid b_1 \qquad G \vdash t_2 : p_2 \mid b_2}{G \vdash t_1 : (\mathsf{in\text{-}hole}\ p_1\ p_2) \mid b_1 \sqcup b_2}$$

$$\boxed{G \vdash t = C[t'] : p \mid b}$$

$$\frac{}{G \vdash t = \mathsf{hole}[t] : \mathsf{hole} \mid \varnothing}$$

$$\frac{G \vdash t_1 = C[t_1'] : p_1 \mid b_1 \qquad G \vdash t_2 : p_2 \mid b_2 \qquad k \in \{\mathsf{cons}, \mathsf{left}, \mathsf{right}\}}{G \vdash (k\ t_1\ t_2) = (\mathsf{left}\ C\ t_2)[t_1'] : (\mathsf{cons}\ p_1\ p_2) \mid b_1 \sqcup b_2}$$

$$\frac{G \vdash t_1 : p_1 \mid b_1 \qquad G \vdash t_2 = C[t_2'] : p_2 \mid b_2 \qquad k \in \{\mathsf{cons}, \mathsf{left}, \mathsf{right}\}}{G \vdash (k\ t_1\ t_2) = (\mathsf{right}\ t_1\ C)[t_2'] : (\mathsf{cons}\ p_1\ p_2) \mid b_1 \sqcup b_2}$$

$$\frac{p \in G(n) \qquad G \vdash t_1 = C[t_2] : p \mid b}{G \vdash t_1 = C[t_2] : (\mathsf{nt}\ n) \mid \varnothing}$$

$$\frac{G \vdash t = C_1[t_1] : p_1 \mid b_1 \qquad G \vdash t_1 = C_2[t_2] : p_2 \mid b_2}{G \vdash t = (C_1 \mathbin{++} C_2)[t_2] : (\mathsf{in\text{-}hole}\ p_1\ p_2) \mid b_1 \sqcup b_2}$$

$$\frac{G \vdash t_1 = C[t_2] : p \mid b}{G \vdash t_1 = C[t_2] : (\mathsf{name}\ x\ p) \mid \{(x, C)\} \sqcup b}$$

$$\begin{aligned} G &\in \mathit{Non\text{-}Terminal} \rightarrow \wp(p) \\ b &\in \mathit{Variable} \rightarrow t \end{aligned} \qquad \begin{aligned} \mathsf{hole} \mathbin{++} C &= C \\ (\mathsf{left}\ C_1\ t) \mathbin{++} C_2 &= (\mathsf{left}\ (C_1 \mathbin{++} C_2)\ t) \\ (\mathsf{right}\ t\ C_1) \mathbin{++} C_2 &= (\mathsf{right}\ t\ (C_1 \mathbin{++} C_2)) \end{aligned}$$

Figure 9: Matching and Decomposition

4 An Algorithm for Matching

The rules in figure 9 provide a declarative definition of context-sensitive matching, but they do not lead directly to a tractable matching algorithm. There are two problems. First, as reflected in the two cons decomposition rules, an algorithm cannot know a priori whether to match on the left and decompose on the right or to decompose on the left and match on the right. An implementation that tries both possibilities scales exponentially in the number of nested cons patterns matched (counting indirect nesting through non-terminals). Second, the rules provide no answer to the question of whether to proceed in expanding a non-terminal if none of the input term has been consumed since last encountering that non-terminal. This question arises, for example, when decomposing by the non-terminal E from the grammar in figure 6, since E's second alternative causes the in-hole rule to decompose the same term by E. This second problem is the manifestation of left recursion in the form of grammars we consider.

The first problem can be solved by matching and decomposing simultaneously. Since these tasks differ only in their treatment of hole patterns, much work can be saved by sharing intermediate results between the tasks. Figure 10 demonstrates this approach with a function M that returns a set of pairs (d, b) representing possible ways to match or decompose the input term. In a pair representing a match, d is the marker $\bullet$; in a pair representing a decomposition, d is a pair (C, t) such that the input term can be decomposed into a context C and a sub-term t occurring in C's hole.

The first two M cases handle the pattern hole. If the term in question is also hole, then it may be considered either to match hole or to decompose into hole in the empty context. If the term is not hole, then only decomposition is possible. The third case handles atomic patterns by producing a match result only if the given term is identical to the atom.

The (meta) context in which a call to M appears may eventually discard some or all of the results it receives. For example, consider the fourth clause, which handles cons patterns. If the term is also a pair (constructed with any of cons, left, or right), then this case makes two recursive calls and examines the cross product of the results using the select helper function. For each result pair, the case merges their bindings and checks that the results are not both decompositions. If neither is a decomposition, select combines the pair into a match result; if exactly one is a decomposition, it extends the decomposition with the term matched by the non-decomposition. If both are decompositions, then the match fails.

The next case, for patterns (in-hole p_c p_h), recurs with p_c and the input term, expecting to receive decompositions. For each one, it makes another recursive call, this time with p_h and the sub-term in the decomposition's focus. Each of the latter call's results m is combined with the decomposition's context, yielding a match result if m is a match and a larger context if m is a decomposition.

The remaining three cases are straightforward. The name case recurs on the sub-pattern and extends the bindings of each of the results with either the matched term or the context carved out by the decomposition. The nt case tries each alternative, discarding the binding component of each result. The final case, a catch-all, applies when the pattern does not match or decompose the input term.

$$\begin{aligned}
&\mathsf{matches} : G\ p\ t \to \wp(b) && m ::= (d, b)\\
&\mathsf{matches}\llbracket G, p, t\rrbracket = \{b \mid (\bullet, b) \in \mathsf{M}\llbracket G, p, t\rrbracket\} && d ::= (C, t) \mid \bullet
\end{aligned}$$

$$\begin{aligned}
&\mathsf{M} : G\ p\ t \to \wp(m)\\
&\mathsf{M}\llbracket G, \mathsf{hole}, \mathsf{hole}\rrbracket &&= \{((\mathsf{hole}, \mathsf{hole}), \varnothing),\ (\bullet, \varnothing)\}\\
&\mathsf{M}\llbracket G, \mathsf{hole}, t\rrbracket &&= \{((\mathsf{hole}, t), \varnothing)\}\\
&\mathsf{M}\llbracket G, a, a\rrbracket &&= \{(\bullet, \varnothing)\}\\
&\mathsf{M}\llbracket G, (\mathsf{cons}\ p_l\ p_r), (k\ t_l\ t_r)\rrbracket &&= \{(d, b) \mid k \in \{\mathsf{cons},\ \mathsf{left},\ \mathsf{right}\},\\
&&&\qquad d \in \mathsf{select}\llbracket t_l, d_l, t_r, d_r\rrbracket,\\
&&&\qquad b = b_l \sqcup b_r,\\
&&&\qquad (d_r, b_r) \in \mathsf{M}\llbracket G, p_r, t_r\rrbracket,\\
&&&\qquad (d_l, b_l) \in \mathsf{M}\llbracket G, p_l, t_l\rrbracket\}\\
&\mathsf{M}\llbracket G, (\mathsf{in\text{-}hole}\ p_c\ p_h), t\rrbracket &&= \{(d, b) \mid d = \mathsf{combine}\llbracket C, d_h\rrbracket,\\
&&&\qquad b = b_c \sqcup b_h,\\
&&&\qquad (d_h, b_h) \in \mathsf{M}\llbracket G, p_h, t_c\rrbracket,\\
&&&\qquad ((C, t_c), b_c) \in \mathsf{M}\llbracket G, p_c, t\rrbracket\}\\
&\mathsf{M}\llbracket G, (\mathsf{name}\ x\ p), t\rrbracket &&= \{(d, b') \mid b' = \{(x, \mathsf{named}\llbracket d, t\rrbracket)\} \sqcup b,\\
&&&\qquad (d, b) \in \mathsf{M}\llbracket G, p, t\rrbracket\}\\
&\mathsf{M}\llbracket G, (\mathsf{nt}\ n), t\rrbracket &&= \{(d, \varnothing) \mid (d, b) \in \mathsf{M}\llbracket G, p, t\rrbracket, p \in G(n)\}\\
&\mathsf{M}\llbracket G, p, t\rrbracket &&= \{\}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{select} : t\ d\ t\ d \to \wp(d)\\
&\mathsf{select}\llbracket t_1, \bullet, t_2, \bullet\rrbracket &&= \{\bullet\}\\
&\mathsf{select}\llbracket t, (C, t_1'), t_2, \bullet\rrbracket &&= \{((\mathsf{left}\ C\ t_2), t_1')\}\\
&\mathsf{select}\llbracket t_1, \bullet, t_2, (C, t_2')\rrbracket &&= \{((\mathsf{right}\ t_1\ C), t_2')\}\\
&\mathsf{select}\llbracket t_1, (C, t_1'), t_2, (C', t_2')\rrbracket &&= \{\}
\end{aligned}$$

$$\begin{aligned}
&\mathsf{combine} : C\ d \to d\\
&\mathsf{combine}\llbracket C, \bullet\rrbracket &&= \bullet\\
&\mathsf{combine}\llbracket C_1, (C_2, t)\rrbracket &&= (C_1 \mathbin{++} C_2, t)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{named} : d\ t \to t\\
&\mathsf{named}\llbracket \bullet, t\rrbracket &&= t\\
&\mathsf{named}\llbracket (C, t_1), t_2\rrbracket &&= C
\end{aligned}$$

Figure 10: Core matching algorithm (cases apply in order)

Putting aside the problem of left recursion, the call $\mathsf{M}[\![G, p, t]\!]$ computes the set of b such that $G \vdash t : p \mid b$ or $G \vdash t = C[t'] : p \mid b$ for some C and t', and the top-level wrapper function $\mathsf{matches}$ restricts this set to the bindings associated with match derivations.

To make this precise, we first give a definition of left-recursion. Intuitively, a grammar is left-recursive if there is a way, in a straight-forward recursive parser, to get from some non-terminal back to that same non-terminal without consuming any input. So, our definition of left-recursion builds a graph from the grammar by connecting each pattern to the other patterns that might reached without consuming any input, and then checks for a cycle in the graph. The most interesting case is the last one, where an $\mathsf{in\text{-}hole}$ pattern is connected to its second argument when the first argument can generate hole.

Definition. A grammar G is *left recursive* if $p \rightarrow_G^* p$ for some p, where $\rightarrow_G^*$ is the transitive (but not reflexive) closure of $\rightarrow_G \subseteq p \times p$, the least relation satisfying the following conditions:

1. $(\textbf{nt}\ n) \rightarrow_G p$ if $p \in G(n)$,
2. $(\textbf{name}\ n\ p) \rightarrow_G p$.
3. $(\textbf{in-hole}\ p\ p') \rightarrow_G p$,
4. $(\textbf{in-hole}\ p\ p') \rightarrow_G p'$ if $G \vdash \textbf{hole} : p \mid b$, and

Theorem. For all G, p, and t, if G is not left recursive, then $b \in \textbf{matches}[\![G, p, t]\!] \Leftrightarrow G \vdash t : p \mid b$.

The complete proof is given in section 7.

Parsing algorithms that support left recursive context-free grammars go back nearly fifty years (Kuno 1965). We refer the reader to Frost et al. (2007, section 3) for a summary. We have implemented an extension of the packrat parsing algorithm (Warth et al. 2008) that dynamically detects left recursion and treats the choice leading to it as a failure. If the other choices for the same portion of the input make any progress at all, the algorithm repeats the parse attempt, in hopes that the entries added to the memo table during the failed attempt will cause a second attempt to succeed. This process continues as long as repeated attempts make additional progress. Extending the algorithm in figure 10 with a similar iterative phase allows matching of terms from left recursive grammars, such the ones in figure 6 and figure 7.

5 A Semantics for Reduction

We now put the notion of matching from section 3 to work in a formalization of the standard notation for context-sensitive reduction rules. As with patterns, we consider a core specification language that lacks many of the conveniences of a language like Redex but nevertheless highlights the principal ideas.

Figure 11 shows our definition. A user of Redex specifies a grammar and rules of the shape $p \longrightarrow r$, each consisting of a pattern p and a term template r. Redex uses the judgment form in the upper-left corner of the figure to determine if a particular term t

$$\frac{G \vdash t : p \mid b \qquad t' = \mathsf{inst}[\![r, b]\!]}{G \vdash t \,/\, p \longrightarrow t' \,/\, r}$$

$$\begin{array}{l}
r ::= a \\
\quad \mid \mathsf{hole} \\
\quad \mid (\mathsf{var}\ x) \\
\quad \mid (\mathsf{app}\ f\ r) \\
\quad \mid (\mathsf{in\text{-}hole}\ r\ r) \\
\quad \mid (\mathsf{cons}\ r\ r) \\
f \in\ t \rightarrow t
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{inst} : r\ b \rightarrow t} \\
\mathsf{inst}[\![a, b]\!] & = a \\
\mathsf{inst}[\![\mathsf{hole}, b]\!] & = \mathsf{hole} \\
\mathsf{inst}[\![(\mathsf{var}\ x), b]\!] & = b(x) \\
\mathsf{inst}[\![(\mathsf{in\text{-}hole}\ r_1\ r_2), b]\!] & = \mathsf{plug}[\![\mathsf{inst}[\![r_1, b]\!], \mathsf{inst}[\![r_2, b]\!]]\!] \\
\mathsf{inst}[\![(\mathsf{cons}\ r_1\ r_2), b]\!] & = \mathsf{join}[\![\mathsf{inst}[\![r_1, b]\!], \mathsf{inst}[\![r_2, b]\!]]\!] \\
\mathsf{inst}[\![(\mathsf{app}\ f\ r), b]\!] & = \delta(f, \mathsf{inst}[\![r, b]\!])
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{plug} : C\ t \rightarrow t} \\
\mathsf{plug}[\![\mathsf{hole}, t]\!] & = t \\
\mathsf{plug}[\![(\mathsf{left}\ C_l\ t_r), C]\!] & = (\mathsf{left}\ \mathsf{plug}[\![C_l, C]\!]\ t_r) \\
\mathsf{plug}[\![(\mathsf{left}\ C_l\ t_r), t]\!] & = (\mathsf{cons}\ \mathsf{plug}[\![C_l, t]\!]\ t_r) \\
\mathsf{plug}[\![(\mathsf{right}\ t_l\ C_r), C]\!] & = (\mathsf{right}\ t_l\ \mathsf{plug}[\![C_r, C]\!]) \\
\mathsf{plug}[\![(\mathsf{right}\ t_l\ C_r), t]\!] & = (\mathsf{cons}\ t_l\ \mathsf{plug}[\![C_r, t]\!])
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{join} : t\ t \rightarrow t} \\
\mathsf{join}[\![C, t]\!] & = (\mathsf{left}\ C\ t) \quad \text{where } \mathsf{no\text{-}ctxts}\ t \\
\mathsf{join}[\![t, C]\!] & = (\mathsf{right}\ t\ C) \quad \text{where } \mathsf{no\text{-}ctxts}\ t \\
\mathsf{join}[\![t_1, t_2]\!] & = (\mathsf{cons}\ t_1\ t_2)
\end{array}$$

$\delta : (t \rightarrow t)\ t \rightarrow t$

An unspecified function that applies metafunctions

$$\frac{}{\mathsf{no\text{-}ctxts}\ a} \qquad \frac{\mathsf{no\text{-}ctxts}\ t_1 \qquad \mathsf{no\text{-}ctxts}\ t_2}{\mathsf{no\text{-}ctxts}\ (\mathsf{cons}\ t_1\ t_2)}$$

Figure 11: A semantics for reduction (function cases apply in order)

reduces to t' by the given rule. The grammar in the figure's top-right gives the syntax for term templates, which include atoms, the context hole, references to variables bound by the left-hand side, applications of meta-level functions (e.g., substitution), hole-filling operations, and pairing operations.

The rest of the figure defines template instantiation. Atoms and hole instantiate to themselves, variables instantiate to their values, and meta-applications instantiate to the result of applying the meta-function to the instantiated argument template.

The instantiation of in-hole templates makes use of a generic plug function that accepts a context and a term and returns the result of plugging the context with the term. When plug's second argument is a context, it constructs a larger context by concatenating the two contexts, preserving the path to the hole. The path extension is necessary, for example, to support the following rule for an unusual control operator:

$$E[(\mathsf{call/cc2}\ v)] \longrightarrow E[(v\ (\mathsf{cont}\ E[E]))]$$

When plug's second argument is some non-context term, it replaces the left or right constructor with cons, producing a non-context term.

Insistence that plug's first argument be a context creates a potential problem for rules which extend contexts, like this one for another unusual control operator:

$$E[(\mathsf{call/cc+}\ v)] \longrightarrow E[(v\ (\mathsf{cont}\ (\mathsf{+1}\ E)))]$$

Although the rule does not explicitly define a path for the extended context $(\mathsf{1+}\ E)$, one can be safely inferred, since the term paired with E has no pluggable sub-terms.

The case of the inst function for cons templates performs this inference via the function join. When given a context and a term containing no contexts, join extends the context's path through the extra layer. When both arguments contain contexts, join combines the terms with cons, preventing possible ambiguity in a subsequent plugging operation.

Note, however, that the embedded contexts themselves remain pluggable by reduction rules and meta-functions that later pick apart the result term. For example, consider the rule for yet another unusual control operator:

$$E[(\mathsf{call/ccs}\ v)] \longrightarrow E[(v\ (\mathsf{tuple}\ (\mathsf{cont}\ E)\ (\mathsf{cont}\ E[E])))]$$

This rule calls v with a pair of continuation values. The term denoting this pair is not itself pluggable, but the embedded contexts can be plugged by subsequent reduction steps, after they are extracted by the reduction rules for projecting tuple components.

In addition to these contrived reduction rules, the semantics in figure 11 supports all of the systems in section 2, as well as the most sophisticated uses of contexts we have encountered in the literature, specifically:

- Ariola and Felleisen (1997)'s core call-by-need calculus. Their extension of this calculus to `letrec` uses decomposition in fundamentally the same way, but the particular formulation they choose makes use of pattern-matching constructs orthogonal to the ones we describe here, namely associative-commutative matching and a Kleene star-like construct that enforces dependencies between adjacent terms.

The examples directory distributed with Redex shows one way to define their `letrec` evaluation contexts without these constructs, which Redex does not currently support.

- Flatt et al. (2007)'s semantics for delimited control in the presence of dynamic binding, exception handling, and Scheme's `dynamic-wind` form.
- Chang and Felleisen (2011)'s call-by-need calculus, which defines evaluation contexts using a heavily left-recursive grammar.

6 Related Work

Barendregt (1984) makes frequent use of a notion of contexts specialized to λ-terms. Like ours, these contexts may contain multiple holes, but plug's behavior differs in that it fills all of the context's holes. Felleisen and Hieb (1992) exploit the power of a selective notion of context to give equational semantics for many aspects of programming languages, notably continuations and state. The meaning of multi-holed grammars does not arise in their work, since the grammar for contexts restricts them to exactly one hole.

Lucas (1995) later explored an alternative formulation of selective contexts. This formulation defines contexts not by grammars but by specifying, for each function symbol, which sub-term positions may be reduced. Because the specification depends only on the function symbol's identity (i.e., and not on its sub-terms), this formulation cannot express common evaluation strategies, such as left-to-right, call-by-value evaluation. Follow-up work on this form of context-sensitive rewriting focuses on tools for proving termination, generally a topic of limited interest when studying reduction systems designed to model a programming language since these systems are not expected to terminate.

As part of their work on SL, a meta-language similar to Redex, Xiao et al. (2001) define a semantics for Felleisen-Hieb contexts by translating grammars into their own formalism, an extension of finite tree automata. This indirect approach allows SL to prove decomposition lemmas automatically using existing automata algorithms, but it is considerably more complicated than our approach and does not allow for multi-holed grammars like the ones in figure 5 and figure 6.

Dubois (2000) develops the first formulation of a Felleisen-Hieb reduction semantics in a proof assistant, as part of a mechanized proof of the soundness of ML's type system. Her formulation encodes single-hole contexts as meta-level term-to-term functions (restricted to coincide with the usual grammar defining call-by-value evaluation) and therefore models plug as meta-application. The formulation does not use an explicit notion of decomposition; instead, the contextual closure reduction rule applies to terms that may be formed using the plug operation.

Berghofer's, Leroy's, and Xi's solutions to the POPLmark Challenge (Aydemir et al. 2005) use Dubois's encoding for the challenge's reduction semantics. Vouillon's solution uses a first-order encoding of contexts and therefore provides an explicit definition of plugging. The other submitted solutions use structural operational semantics, do not address dynamic semantics at all, or are no longer available online.

Danvy and Nielsen (2004) and Sieczkowski et al. (2010) provide an axiomatization of the various components of a Felleisen-Hieb reduction semantics, such as a decomposition relation, that together define the semantics. This axiomatization is not an appropriate basis for Redex for two reasons. First, it requires users to specify plugging and decomposition explicitly. Common practice leaves these definitions implicit, and one of our design goals for Redex is to support conventional definitions. Second, the axiomatization requires decomposition to be a (single-valued) function, ruling out the semantics in figure 1 and, more problematically, reduction semantics for multi-threaded programs and programs in languages like C and Scheme, which do not specify an order of evaluation for application expressions.

More broadly speaking, there are hundreds[4] of papers that use evaluation context semantics to model programming languages for just as many different purposes. Although we have not implemented anywhere near all of them in Redex, we have sought out interesting and non-standard ones over the years to try them out and to build our intuition about how a semantics should behave.

Acknowledgments Thanks to Stephen Chang for his many interesting examples of contexts that challenged our understanding of context-sensitive matching. Thanks also to Matthias Felleisen and Matthew Flatt for helpful discussions of the work.

A version of this paper can be found online at:

`http://www.eecs.northwestern.edu/~robby/plug/`

That web page contains a version of the paper with a complete proof of the theorem and the Redex models for all of the figures in this paper.

Bibliography

Zena M. Ariola and Matthias Felleisen. The Call-by-Need Lambda-Calculus. *J. Functional Programming* 7(3), pp. 265–301, 1997.

Brian E. Aydemir, Aaron Bohannon, Matthew Fairbairn, J. Nathan Foster, Benjamin C. Pierce, Peter Sewell, Dimitrios Vytiniotis, Geoffrey Washburn, Stephanie Weirich, and Steve Zdancewic. Mechanized Metatheory for the Masses: The POPLmark Challenge. In *Proc. Intl. Conf. Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science volume 3603, pp. 50–65, 2005.

H. P. Barendregt. The Lambda Calculus: Its Syntax and Semantics. North Holland, 1984.

[4] There are more than 400 citations to the original Felleisen-Hieb paper; while evaluation-context based semantics are still widely used, the paper is now rarely cited as it has become a standard part of the programming languages landscape.

Stephen Chang and Matthias Felleisen. The Call-by-need Lambda Calculus. Unpublished Manuscript, 2011.

Olivier Danvy and Lasse R. Nielsen. Refocusing in Reduction Semantics. Aarhus University, BRICS RS-04-26, 2004.

Catherine Dubois. Proving ML Type Soundness within Coq. In *Proc. Intl. Conf. Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science volume 1869, pp. 126–144, 2000.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2010.

Matthias Felleisen and Robert Hieb. The Revised Report on the Syntactic Theories of Squential Control and State. *Theoretical Computer Science* 103(2), pp. 235–271, 1992.

Matthew Flatt, Gang Yu, Robert Bruce Findler, and Matthias Felleisen. Adding Delimited and Composable Control to a Production Programming Environment. In *Proc. ACM Intl. Conf. Functional Programming*, pp. 165–176, 2007.

Richard A. Frost, Rahmatullah Hafiz, and Paul C. Callaghan. Modular and Efficient Top-Down Parsing for Ambiguous Left-Recursive Grammars. In *Proc. International Conference on Parsing Technology*, pp. 109–120, 2007.

Casey Klein, Matthew Flatt, and Robert Bruce Findler. The Racket Virtual Machine and Randomized Testing. 2010. `http://plt.eecs.northwestern.edu/racket-machine/`

Susumu Kuno. The Predictive Analyzer and a Path Elimination Technique. *Communications of the ACM* 8(7), pp. 453–462, 1965.

Salvador Lucas. Fundamentals of Context-Sensitive Rewriting. In *Proc. Seminar on Current Trends in Theory and Practice of Informatics*, Lecture Notes in Computer Science volume 1012, pp. 405–412, 1995.

Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A Visual Environment for Developing Context-Sensitive Term Rewriting Systems. In *Proc. Intl. Conf. Rewriting Techniques and Applications*, Lecture Notes in Computer Science volume 3091, pp. 301–311, 2004.

Filip Sieczkowski, Malgorzata Biernacka, and Dariusz Biernacki. Automating Derivations of Abstract Machines from Reduction Semantics: A Generic Formalization of Refocusing in Coq. In *Proc. Symp. Implementation and Application of Functional Languages*, To appear in Lecture Notes in Computer Science, 2010.

Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton van Straaten, Richard Kelsey, William Clinger, Jonathan Rees, Robert Bruce Findler, and Jacob Matthews. Revised [6] Report on the Algorithmic Language Scheme. Cambridge University Press, 2007.

Alessandro Warth, James R. Douglass, and Todd Millstein. Packrat Parsers Can Support Left Recursion. In *Proc. ACM SIGPLAN Wksp. Partial Evaluation and Program Manipulation*, pp. 103–110, 2008.

Yong Xiao, Amr Sabry, and Zena M. Ariola. From Syntactic Theories to Interpreters: Automating the Proof of Unique Decomposition. *Higher-Order and Symbolic Computation* 14(4), pp. 387–409, 2001.

7 Appendix

Theorem. For all G, p, and t, if G is not left recursive, then $b \in$ **matches**$[\![G, p, t]\!] \Leftrightarrow G \vdash t : p \mid b$.

Proof:

```
If b ∈ matches⟦G, p, t⟧, then (•, b) ∈ M⟦G, p, t⟧,
which by the soundness lemma below, implies G ⊢ t : p | b.
If G ⊢ t : p | b, then (•, b) ∈ M⟦G, p, t⟧ by the completeness lemmma below;
consequently b ∈ matches⟦G, p, t⟧.
```

Lemma (Soundness):

```
For all G, p, t, b, C, and t',
1. (•, b) ∈ M⟦G, p, t⟧ implies G ⊢ t : p | b, and
2. ((C, t'), b) ∈ M⟦G, p, t⟧ implies G ⊢ t = C[t'] : p | b.
```

Proof: By induction on the M derivation.

```
Case: hole (and t = hole)
  Given:
    M⟦G, hole, hole⟧ = {((hole, hole), ∅), (•, ∅)}
  By the hole decomposition rule,
    G ⊢ hole = hole[hole] : hole | ∅
  By the hole matching rule,
    G ⊢ hole : hole | ∅

Case: hole (and t ≠ hole)
  Given:
    M⟦G, hole, t⟧ = {((hole, t), ∅)}
  By the hole decomposition rule,
    G ⊢ t = hole[t] : hole | ∅

Case: atom
  Given:
    M⟦G, a, a⟧ = {(•, ∅)}, and
  By the atom matching rule,
    G ⊢ a : a | ∅

Case: cons
  Let (d, b) ∈ M⟦G, (cons p_1 p_2), (k t_1 t_2)⟧, i.e.,
    1. (d_1, b_1) ∈ M⟦G, p_1, t_1⟧,
    2. (d_2, b_2) ∈ M⟦G, p_2, t_2⟧,
    3. b_1 ⊔ b_2 = b, and
    4. d ∈ select⟦t_1, d_1, t_2, d_2⟧.

  Subcase: d_1 = d_2 = •
    Then d = • by the first 'select' clause, requiring
```

```
      G ⊢ (k t_1 t_2) : (cons p_1 p_2) : b,
    which holds by the cons matching rule, since:
      1. G ⊢ t_1 : p_1 | b_1 by induction,
      2. G ⊢ t_2 : p_2 | b_2 by induction, and
      3. b_1 ⊔ b_2 = b by hypothesis.

  Subcase: d_1 = • and d_2 = (C, t_2’)
    Then d = ((right t_1 C), t_2’) by the third ‘select’ clause, requiring
      G ⊢ (k t_1 t_2) = (right t_1 C)[t_2’] : (:cons p_1 p_2) : b,
    which holds by the cons-right decomposition rule, since:
      1. G ⊢ t_1 : p_1 | b_1 by induction,
      2. G ⊢ t_2 = C[t_2’] : p_2 | b_2 by induction, and
      3. b_1 ⊔ b_2 = b by hypothesis.

  Subcase: d_1 = (C, t_1’) and d_2 = •
    Similarly, using the second ‘select’ rule and the cons-left
    decomposition rule.

  Subcase: d_1 = (C_1, t_1’) and d_2 = (C_2, t_2’)
    Impossible since select⟦t_1, (C_1, t_1’), t_2, (C_2, t_2’)⟧ = ∅

Case: in-hole
  Let (d, b) ∈ M⟦G, (in-hole p_1 p_2), t⟧, i.e.,
    1. ((C_1, t_1), b_1) ∈ M⟦G, p_1, t⟧,
    2. (d_2, b_2) ∈ M⟦G, p_2, t_1⟧,
    3. b_1 ⊔ b_2 = b, and
    4. d = combine⟦C_1, d_2⟧.

  Subcase: d_2 = •
    combine⟦C_1, •⟧ = •, requiring:
      G ⊢ t : (in-hole p_1 p_2) | b,
    which holds by the in-hole matching rule, since:
      1. G ⊢ t = C_1[t_1] : p_1 | b_1 by induction,
      2. G ⊢ t_1 : p_2 | b_2 by induction, and
      3. b_1 ⊔ b_2 = b by hypothesis.

  Subcase: d_2 = (C_2, t_2)
    combine⟦C_1, (C_2, t_2)⟧ = (C_1 ++ C_2, t_2), requiring:
      G ⊢ t = (C_1 ++ C_2)[t_2] : (in-hole p_1 p_2) | b,
    which holds by the in-hole decomposition rule, since:
      1. G ⊢ t = C_1[t_1] : p_1 | b_1 by induction,
      2. G ⊢ t_1 = C_2[t_2] : p_2 | b_2 by induction, and
      3. b_1 ⊔ b_2 = b by hypothesis.

Case: name
```

Let $(d, b') \in M[\![G, (\text{name } x\ p), t]\!]$, i.e.,

1. $(d, b) \in M[\![G, p, t]\!]$,
2. $\{(x, \text{named}[\![d, t]\!])\} \sqcup b = b'$.

Subcase: $d = \bullet$

By the name matching rule, $G \vdash t : (\text{name } x\ p) \mid b'$ since:

1. $G \vdash t : p \mid b$, by induction, and
2. $\{(x, t)\} \sqcup b = b'$, since $\{(x, \text{named}[\![d, t]\!])\} \sqcup b = b'$ and $\text{named}[\![\bullet, t]\!] = t$ by the first ‘named’ clause.

Subcase: $d = (C, t')$

By the name decomposition rule, $G \vdash t = C[t'] : (\text{name } x\ p) \mid b'$ since:

1. $G \vdash t = C[t'] : p \mid b$, by induction, and
2. $\{(x, C)\} \sqcup b = b'$, since $\{(x, \text{named}[\![(C, t'), t]\!])\} \sqcup b = b'$ and $\text{named}[\![(C, t'), t]\!] = C$ by the second ‘named’ clause.

Case: nt

Let $(d, \emptyset) \in M[\![G, (\text{nt } n), t]\!]$, i.e.,

1. $p \in G(n)$, and
2. $(d, b) \in M[\![G, p, t]\!]$.

Subcase: $d = \bullet$

By the nt matching rule, $G \vdash t : (\text{nt } n) \mid \emptyset$ since:

1. $p \in G(n)$ by hypotheis, and
2. $G \vdash t : p \mid b$ by induction.

Subcase: $d = (C, t')$

Similarly by the nt decomposition rule.

Case: the "else" clause

Impossible since $M[\![G, p, t]\!] = \emptyset$

Lemma (Completeness):
For all G, p, t, b, C, and t', if G is not left recursive, then

1. $G \vdash t : p \mid b$ implies $(\bullet, b) \in M[\![G, p, t]\!]$, and
2. $G \vdash t = C[t'] : p \mid b$ implies $((C, t'), b) \in M[\![G, p, t]\!]$.

Proof: By (mutual) induction on the matching/decomposition derivation.

Case: atom matching

Given: $G \vdash a : a \mid \emptyset$

$(\bullet, \emptyset) \in M[\![G, a, a]\!]$ by M's atom case

Case: hole matching

Given: $G \vdash \text{hole} : \text{hole} \mid \emptyset$

$(\bullet, \emptyset) \in M[\![G, \text{hole}, \text{hole}]\!]$ by M's hole case for the term hole

```
Case: name matching
  Given:
    1. G ⊢ t : (name x p) | b',
    2. G ⊢ t : p | b, and
    3. b ⊔ {(x, t)} = b'
  By M's name clause, (•, b) ∈ M⟦G, (name x p), t⟧ since:
    1. (•, b) ∈ M⟦G, p, t⟧ by induction.
    2. named⟦•, t⟧ = t by definition.
    3. b ⊔ {(x, t)} = b' by hypothesis.

Case: nt matching
  Given:
    1. G ⊢ t : (:nt n) | ∅,
    2. p ∈ G(n), and
    3. G ⊢ t : p | b.
  By M's nt clause, (•, ∅) ∈ M⟦G, (nt n), t⟧ since:
    1. p ∈ G(n) by hypothesis.
    2. (•, b) ∈ M⟦G, p, t⟧ by induction.
    3. M⟦G, p', t⟧ terminates for n's other productions p' by the
    termination lemma below

Case: cons matching
  Given:
    1. G ⊢ (k t_1 t_2) : (cons p_1 p_2) | b,
    2. G ⊢ t_1 : p_1 | b_1,
    3. G ⊢ t_2 : p_2 | b_2, and
    4. b_1 ⊔ b_2 = b.
  By M's cons clause, (•, b) ∈ M⟦G, (cons p_1 p_2), (k t_1 t_2)⟧ since:
    1. (•, b_1) ∈ M⟦G, p_1, t_1⟧ by induction,
    2. (•, b_2) ∈ M⟦G, p_2, t_2⟧ by induction,
    3. b_1 ⊔ b_2 = b by hypothesis, and
    4. • ∈ select⟦t_1, •, t_2, •⟧ by its first clause

Case: in-hole matching
  Given:
    1. G ⊢ t : (in-hole p_1 p_2) | b,
    2. G ⊢ t = C[t'] : p_1 | b_1,
    3. G ⊢ t' : p_2 | b_2, and
    4. b_1 ⊔ b_2 = b.
  By M's in-hole clause, (•, b) ∈ M⟦G, (in_hole p_1 p_2), t⟧ since:
    1. ((C, t'), b_1) ∈ M⟦G, p_1, t⟧ by induction,
    2. (•, b_2) ∈ M⟦G, p_2, t'⟧ by induction,
    3. b_1 ⊔ b_2 = b by hypothesis, and
    4. • = combine⟦C, •⟧ by its first clause.
```

```
Case: hole decomposition
  Given: G ⊢ t = hole[t] : :hole | ∅
  Subcase: t = hole
    By the M clause that applies when the pattern and term are hole,
      ((hole, hole), ∅) ∈ M⟦G, hole, t⟧
  Subcase: t ≠ :hole
    By the M clause that applies when the pattern is hole but the term is not,
      ((hole, t), ∅) ∈ M⟦G, hole, t⟧

Case: cons-left decomposition
  Given:
    1. G ⊢ (k t_1 t_2) = (left C t_2)[t_1'] : (cons p_1 p_2) | b,
    2. G ⊢ t_1 = C[t_1'] : p_1 | b_1,
    3. G ⊢ t_2 : p_2 | b_2, and
    4. b_1 ⊔ b_2 = b'.
  By M's cons clause,
    (((left C t_2), t_1'), b) ∈ M⟦G, (cons p_1 p_2), (k t_1 t_2)⟧
  since:
    1. ((C, t_1'), b_1) ∈ M⟦G, p_1, t_1⟧ by induction,
    2. (●, b_2) ∈ M⟦G, p_2, t_2⟧ by induction,
    3. b_1 ⊔ b_2 = b' by hypothesis, and
    4. ((left C t_2), t_1') ∈ select⟦t_1, (C, t_1'), t_2, ●⟧
       by its second clause.

Case: cons-right decomposition
  Similar to cons-left composition

Case: nt decomposition
  Given:
    1. G ⊢ t = C[t'] : (nt n) | ∅,
    2. p ∈ G(n), and
    3. G ⊢ t = C[t'] : p | b.
  By M's nt clause, ((C, t'), ∅) ∈ M⟦G, (nt n), t⟧ since:
    1. p ∈ G(n) by hypothesis,
    2. ((C, t'), b) ∈ M⟦G, p, t⟧ by induction, and
    3. M⟦G, p', t⟧ terminates for n's other productions p' by the
    termination lemma below.

Case: in-hole decomposition
  Given:
    1. G ⊢ t = (C_1 ++ C_2)[t_2] : (in-hole p_1 p_2) | b,
    2. G ⊢ t = C_1[t_1] : p_1 | b_1,
    3. G ⊢ t_1 = C_2[t_2] : p_2 | b_2, and
    4. b_1 ⊔ b_2 = b'.
```

By M's in-hole clause,
((C_1 ++ C_2, t_2), b) ∈ M⟦G, (in-hole p_1 p_2), t⟧
since:

1. ((C_1, t_1), b_1) ∈ M⟦G, p_1, t⟧ by induction,
2. ((C_2, t_2), b_2) ∈ M⟦G, p_2, t_1⟧ by induction,
3. b_1 ⊔ b_2 = b' by hypothesis, and
4. combine⟦C_1, (C_2, t_2)⟧ = C_1 ++ C_2 by its second clause.

Case: name decomposition
Given:

1. G ⊢ t = C[t'] : (name x p) | b',
2. G ⊢ t = C[t'] : p | b, and
3. b ⊔ {(x, C)} = b'.

By M's name clause, ((C, t'), b') ∈ M⟦G, (name x p), t⟧ since:

1. ((C, t'), b) ∈ M⟦G, p, t⟧ by induction,
2. named⟦(C, t'), t⟧ = C by its second clause,
3. b ⊔ {(x, C)} = b' by hypothesis.

Lemma (Termination):
For all G, p, and t, M⟦G, p, t⟧ terminates if G is not left recursive.
Proof:
Patterns and terms can be assigned sizes such that each recursive call of M supplies smaller arguments than the ones it received. In particular, we interpret the size |t| of a term t as:

|a| = 1
|(cons t t')| = 1 + |t| + |t'|
|hole| = 1
|(left C t)| = 1 + |C| + |t|
|(right t C)| = 1 + |t| + |C|

and the size |p| of a pattern p as:

|a| = 0
|(name x p)| = 1 + |p|
|(nt n)| = 1 + max({|p| | p ∈ G(n)|})|
|(in-hole p p')| = 1 + max({|p|, |p'|})|
|(cons p p')| = 0
|hole| = 0

This assignment of pattern sizes is possible because the graph induced by $\to_G$ has no cycles. The assignment can be constructed bottom-up, by beginning with non-terminals n_0 for which n_0 $\not\to^*_G$ n for any non-terminal n, proceeding with the non-terminals n_1 for which n_1 $\to^*_G$ n_0 for only the previous n_0, and so on.

We interpret the overall size of the combined arguments lexicographically, treating the term argument as more significant than the pattern argument.

Each recursive call supplies either a smaller term or a term of the same size and a smaller pattern.

Case: within ‘cons’ clause
Each call supplies a strictly smaller term.

Case: within ‘in-hole’ clause
The first call supplies the same term and a pattern whose size is at most |(in-hole p_c p_h)| - 1.

The second call supplies a term that’s no larger than the input term (by the non-increasing subterms lemma below) and a pattern whose size is at most |(in-hole p_c p_h)| - 1.

Case: within ‘name’ clause
The call supplies the same term and strips the ‘name’ from the pattern, decreasing its size by 1.

Case: within ‘nt’ clause
Each call supplies the same term and a pattern whose size is at most |(nt n)| - 1.

Lemma (Non-increasing subterms)
If ((C, t’), b) ∈ M⟦G, p, t⟧, then |t’| ≤ |t|.
Proof: By induction on the M derivation.

Case: hole (and t = hole)
t = t’

Case: hole (and t ≠ hole)
t = t’

Case: atom
Impossible, since the result is not a decomposition.

Case: cons
By the definition of ‘select’, there are two possibilities that make the result a decomposition:
  Subcase: d_l = (C_l, t_l’) and d_r = •.

```
    Then select⟦t_l, (C_l, t_l'), t_r, ●⟧ = {((left C_l t_r), t_l')}
    |t_l'| ≤ |t_l| by induction
           < |(cons t_l t_r)|

  Subcase: d_l = ● and d_r = (C_r, t_r').

    By a similar argument.

Case: in-hole
By the definition of 'combine', there is one possibility that makes
the result a decomposition:

d_h = ((C_h, t_h), b_h) ∈ M⟦G, p_h, t_c⟧.
Then combine⟦C, (C_h, t_h)⟧ = (C ++ C_h, t_h).
|t_h| ≤ |t_c| and |t_c| ≤ |t|, both by induction.

Case: name
Suppose d = ((C, t'), b) ∈ M⟦G, p, t⟧.
By induction, |t'| ≤ |t|.

Case: M-nt
Similar to the 'name' case.
```