From Syntactic Theories to Interpreters: Automating the Proof of Unique Decomposition

Yong Xiao (`xiao@cs.uoregon.edu`) *
Department of Computer and Information Science
University of Oregon
Eugene, OR 97403

Amr Sabry (`sabry@cs.indiana.edu`) †
Computer Science Department
Indiana University
Bloomington, IN 47405

Zena M. Ariola (`ariola@cs.uoregon.edu`) ‡
Department of Computer and Information Science
University of Oregon
Eugene, OR 97403

Abstract. Developing syntactic theories for reasoning about programming languages usually involves proving a unique-decomposition lemma. The proof of such a lemma is tedious, error-prone, and is usually attempted many times during the design of a theory. We therefore investigate the automation of such proofs.

We map the unique-decomposition lemma to the problems of checking equivalence and ambiguity of syntactic definitions. Because checking these properties of context-free grammars is undecidable, we work with regular tree grammars and use algorithms on finite tree automata to proceed the checking. To make up for the insufficient expressiveness of regular tree grammars, we extend the basic framework with built-in types and constants, contexts, and polymorphic types.

Our implementation extends an earlier system by Xiao et al. that translates semantic specifications expressed as syntactic theories to interpreters. We have successfully used the combined system to generate interpreters and verify the unique-decomposition lemma for a number of examples.

Keywords: Syntactic theories; Interpreters; Proof automation; Regular tree grammars; Finite tree automata.

1. Introduction

Syntactic theories have been developed to reason about many aspects of modern programming languages. Having roots in the λ-calculus, these theories rely on transforming source programs to other programs: they

* Supported by National Science Foundation grant number CCR-9624711.
† Supported by National Science Foundation grant number CCR-9733088. Work started at the University of Oregon
‡ Supported by National Science Foundation grant number CCR-9624711.

require no understanding of mathematical domains, and do not rely on structures like environments, stores, or continuations (Felleisen, 1987). Only the *syntax* of the programming language is relevant.

In almost every semantic treatment of programming languages that is based on manipulating syntax, one is immediately required to prove a *decomposition lemma* which shows the equivalence of two alternative syntactic specifications of the language: one tailored for exposing the syntactic structure of the language, and the other reflecting the relevant semantic notions, *e.g.*, values, redexes, head normal forms, untypable programs, and evaluation contexts.

A simple example might make the idea more precise. The syntax of terms in the λ-calculus is usually given using the following inductive definition (Barendregt, 1984, p.22):

$$M ::= x \mid \lambda x.M \mid MM$$

where x is drawn from an infinite set of variables.

In order to reason about the semantic notion of *solvability* of terms syntactically, one instead needs to view the set of terms as consisting of head normal forms and terms containing a head redex (Barendregt, 1984, p.172):

$$\begin{array}{rcll} M & ::= & \lambda x_1 \ldots x_n.xM_1 \ldots M_m & n, m \geq 0 \\ & \mid & \lambda x_1 \ldots x_n.(\lambda x.M_0)M_1 \ldots M_m & n \geq 0, m \geq 1 \end{array}$$

Every term produced by the second definition is obviously also produced by the first. The converse, which is less evident, is the statement of the decomposition lemma.

In the above case, as in every natural case, the proof of the decomposition lemma is by induction on the structure of terms as given in the first definition. Such a proof is often straightforward: it would typically be omitted from books or published articles. However, as we discovered in several current projects (Ariola and Felleisen, 1997; Fiskio-Lasseter and Sabry, 1999; Launchbury and Sabry, 1997; Semmelroth and Sabry, 1999), such a proof is incredibly tedious in all but the most trivial programming languages. Moreover, in the process of designing the semantics one typically experiments with several definitions for the semantic notions. Every such experiment requires a laborious and error-prone proof attempt. A failed proof attempt typically reveals a missing case in one of the definitions. Even when one settles on the correct semantic notions, a successful proof might require several iterations where in each iteration one reformulates the syntactic definitions to strengthen the inductive hypothesis.

A further complication arises when the decomposition of a term must be *unique*. This is needed, for example, to derive a deterministic evaluation function (*i.e.*, an interpreter). The proof of unique-decomposition lemma is usually performed by using routine induction as before to find *some* decompositions, and then arguing that no further decompositions are possible. Published articles usually omit the complete proof of uniqueness as well. And again, as we discovered, arguing that no further decompositions are possible makes an already long and tedious proof even more intractable.

The obvious solution we seek is to automate the proof of unique decomposition. Given the two syntactic definitions of interest, we would like an algorithm that either confirms the equivalence of the two definitions or produces a counterexample. A counterexample is a term that has no decomposition, or in the uniqueness case, a term with more than one decomposition. Proving the decomposition lemma reduces to checking the equivalence of two grammars; proving uniqueness reduces to checking that a grammar is not ambiguous. However, both equivalence and ambiguity are undecidable in general. We thus restrict the syntactic definitions to regular tree grammars, for which the problems are decidable. Unfortunately, regular tree grammars are not quite expressive enough to accommodate contexts, polymorphic types, and other constructs usually needed in definitions of syntactic theories. We therefore engineer an extension of the framework that is expressive enough while retaining the decidability of the fundamental algorithms.

A prototype of the entire system is implemented and has been tested on a number of examples. The system is embedded in the SL system that generates interpreters from semantic specifications (Xiao et al., 2000).

The remainder of this paper is organized as follows: Section 2 shows the relationship between proofs of unique-decomposition lemmas and the problems of equivalence and ambiguity of grammars. Section 3 surveys the needed concepts about regular tree grammars and finite tree automata. Section 4 adapts these concepts to the development of algorithms for proving unique-decomposition lemmas. Section 5 describes extensions to the approach by considering built-in constants, tuples, contexts, and polymorphic types, and explains how to construct tree automata for them. Section 6 introduces the SL system and extends it to automatically prove unique-decomposition lemmas. Section 7 reviews related work and Section 8 concludes.

2. Equivalence and Ambiguity of Grammars

Abstractly speaking, the problem of unique decomposition consists of relating two syntactic definitions for the same language. Generally, these syntactic presentations would be given as context-free grammars G_1 and G_2. Unique decomposition (Barendregt, 1984, Corollary 8.3.8) can then be restated as problems about grammars G_1 and G_2:

- The existence of a decomposition can be reduced to checking that every term derived by G_1 can be derived by G_2 as well. Since the other implication is usually trivial, the existence of a decomposition is reduced to checking the equivalence of grammars G_1 and G_2, which is undecidable (Hopcroft and Ullman, 1979, Theorem 8.12).

- The uniqueness of a decomposition can be reduced to checking that every term in G_1 can only be derived by G_2 in a unique way. In other words, uniqueness requires checking whether grammar G_2 is unambiguous, which is also undecidable (Hopcroft and Ullman, 1979, Theorem 8.9).

Therefore, we cannot hope to solve the problem of unique decomposition automatically for syntactic definitions given as context-free grammars. Fortunately, syntactic definitions of programming languages are usually given using constructors, and the constructors impose structural restrictions on terms. These syntactic definitions can fit in a class of grammars, called *regular tree grammars*, for which checking equivalence and checking ambiguity are decidable.

3. Regular Tree Grammars and Finite Tree Automata

We review the necessary background on regular tree grammars and their computational models in this section.

3.1. Regular Tree Grammars

A regular tree grammar (Gécseg and Steinby, 1984) is a context-free grammar with one important restriction: the right-hand sides of the productions can only be generated using a fixed number of constructors. The formal definitions are as follows.

Definition 1. Let Σ be a finite set of constructors and let Φ be a finite set of variables. The set of *trees* over Σ and Φ, denoted as $\mathcal{T}(\Sigma, \Phi)$, is inductively defined as follows:

- α is a tree, where $\alpha \in \Phi$.
- If $t_1, \ldots, t_n$ are trees, then $c^n(t_1, \ldots, t_n)$ is a tree, where $n \geq 0$ and c^n is a constructor of arity n in Σ.

We sometimes blur the distinction between *terms* and *trees*, although terms are typically defined over infinite sets of constants and variables. Terms without variables are called *ground terms*.

Definition 2. A *regular tree grammar* is a quadruple of the form (Σ, N, S, P) where:

- Σ is a finite set of constructors.
- N is a finite set of non-terminals.
- S is the start symbol, $S \in N$.
- P is a finite set of rules (productions) of the form $L \rightarrow s$ where L is a non-terminal and $s \in \mathcal{T}(\Sigma, N)$.

If the same non-terminal appears as the left-hand side of several productions, we merge all the productions into one clause using alternatives. Non-terminals do not all necessarily derive ground terms. In the remainder of the paper, we assume that all non-terminals can be derived from the start symbol.

EXAMPLE 1. *A regular tree grammar for λ-calculus is:*

- Σ *is the set* $\{\mathsf{var}, \mathsf{lam}, \mathsf{app}\}$.
- N *is the set* $\{M\}$.
- S *is* M.
- P *is the set:*

$$M \ \rightarrow \ \mathsf{var} \mid \mathsf{lam}(\mathsf{var}, M) \mid \mathsf{app}(M, M)$$

The grammar has been modified from the original presentation in Section 1 by using terminal symbols such as lam and var. The infinite set of variables is abstracted into var. This abstraction is usually possible since no reference to individual variables is made in most semantic specifications. [1]

EXAMPLE 2. *A regular tree grammar expressing the decomposition of λ-calculus terms into head normal forms and terms with a head redex is:*

[1] The specification for the call-by-need calculus is an exception (Ariola and Felleisen, 1997).

– Σ *is the set* $\{\mathsf{var}, \mathsf{lam}, \mathsf{app}\}$.

– N *is the set* $\{M, H, I, U, R\}$.

– S *is* M.

– P *is the set:*

$$\begin{array}{rcl} M & \to & H \mid U \\ H & \to & I \mid \mathsf{lam}(\mathsf{var}, H) \\ I & \to & \mathsf{var} \mid \mathsf{app}(I, M) \\ U & \to & R \mid \mathsf{lam}(\mathsf{var}, U) \\ R & \to & \mathsf{app}(\mathsf{lam}(\mathsf{var}, M), M) \mid \mathsf{app}(R, M) \end{array}$$

Some non-terminals are introduced in the regular tree grammar for decompositions to avoid using the informal abbreviation "...".

3.2. Finite Tree Automata

Since the algorithms for checking equivalence and ambiguity of regular tree grammars are described using the notion of *finite tree automata*, we first define these automata.

Definition 3. (FTA) A *finite tree automaton* is a quadruple $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ where

– Σ is a finite set of constructors.

– Q is a finite set of states, ranged over by q, q_1, q_2, ...

– $Q_f \subseteq Q$ is a set of final states.

– Δ is a finite set of transition rules, each in one of the following forms:

- $q_1 \xrightarrow{\epsilon} q_2$,
- $c^n(q_1, \ldots, q_n) \to q$ where $n \geq 0$ and c^n is a constructor of arity n in Σ.

An automaton can be represented by its transition rules and final states when other information can be easily collected from the transition rules. If the final states are obvious to determine, they can be omitted as well.

EXAMPLE 3. *A possible tree automaton corresponding to the regular tree grammar of Example 1 is:*

$$\begin{array}{lcl}
q_{M_1} \xrightarrow{\epsilon} q_M & \mathsf{var} \rightarrow q_{M_1} & \\
q_{M_2} \xrightarrow{\epsilon} q_M & \mathsf{lam}(q_{M_4}, q_M) \rightarrow q_{M_2} & \mathsf{var} \rightarrow q_{M_4} \\
q_{M_3} \xrightarrow{\epsilon} q_M & \mathsf{app}(q_M, q_M) \rightarrow q_{M_3} &
\end{array}$$

where $q_{M_1}, q_{M_2}, q_{M_3}$ correspond to the three alternatives of M.

Definition 4. Given a finite tree automaton $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$, the *$\epsilon$-transition closure* of states $\{q_1, \ldots, q_n\}$, denoted by $cls_\epsilon\{q_1, \ldots, q_n\}$, is the set of states $\{q \mid q_i \xrightarrow{\epsilon}^* q, 1 \leq i \leq n\}$, where $\xrightarrow{\epsilon}^*$ is the reflexive, transitive closure of $\xrightarrow{\epsilon}$.

Definition 5. Given an automaton $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$, we have:

- a *run*[2] r of $\mathcal{A}$ on a ground term t is a total mapping from subterms of t to ϵ-transition closures of the singletons $\{q\}$, such that r is compatible with Δ, *i.e.*, for every subterm p, if $p = c^n(p_1, \ldots, p_n)$, where $c^n \in \Sigma$, $n \geq 0$, $r(p) = cls_\epsilon\{q\}$ and $r(p_i) = cls_\epsilon\{q_i\}$, for each $1 \leq i \leq n$, then there exist $q' \in cls_\epsilon\{q\}$ and $q'_i \in cls_\epsilon\{q_i\}$, for each $1 \leq i \leq n$, such that $c^n(q'_1, \ldots, q'_n) \rightarrow q' \in \Delta$.

- a run r of $\mathcal{A}$ on term t is *successful* if $r(t)$ contains a final state.

- a term t is *recognized* (or *accepted*) by an automata $\mathcal{A}$ if there exists a successful run of $\mathcal{A}$ on t.

- a term t can *reach* a state q if there exists a run r of $\mathcal{A}$ on t such that $q \in r(t)$.

Intuitively, a successful run on a ground term t tells us how t is constructed, *i.e.*, it corresponds to t's parse tree. For example, a successful run r of the automaton given in Example 3 on $\mathsf{app}(\mathsf{lam}(\mathsf{var}_1, \mathsf{var}_2), \mathsf{var}_3)$[3] could be defined as follows:

$$\begin{array}{rcl}
r(\mathsf{var}_1) & = & \{q_{M_4}\} \\
r(\mathsf{var}_2) & = & \{q_{M_1}, q_M\} \\
r(\mathsf{lam}(\mathsf{var}_1, \mathsf{var}_2)) & = & \{q_{M_2}, q_M\} \\
r(\mathsf{var}_3) & = & \{q_{M_1}, q_M\} \\
r(\mathsf{app}(\mathsf{lam}(\mathsf{var}_1, \mathsf{var}_2), \mathsf{var}_3)) & = & \{q_{M_3}, q_M\}
\end{array}$$

[2] Our definition of run is more general than the standard definition which is based on automata without ϵ-transitions. We keep ϵ-transitions because they will be used in proving uniqueness.

[3] We use var_i, $i = 1, 2, 3$ to distinguish the different positions of var.

As an example of r's compatibility with the productions Δ, note that the production for the constructor lam maps the states q_{M_4} and q_M to q_{M_2}, and that these states are included in the images of var_1, var_2, and $\mathsf{lam}(\mathsf{var}_1, \mathsf{var}_2)$, respectively.

Definition 6. The *tree language* recognized by an automaton $\mathcal{A}$, denoted by $L(\mathcal{A})$, is the set of all terms accepted by $\mathcal{A}$.

Two finite tree automata are *equivalent* if they recognize the same tree language. Like finite automata, finite tree automata are divided into deterministic finite tree automata (DFTAs) and non-deterministic ones (NFTAs). A finite tree automaton is *deterministic* if no two transition rules have the same left-hand side and there is no ϵ-transition rule. Otherwise, it is *non-deterministic*. For instance, the automaton for λ-terms in Example 3 is non-deterministic.

Many theorems in finite automata theory have counterparts in finite tree automata theory. Here are some theorems (Comon et al., 1999) we will use later.

THEOREM 1. *Given an NFTA, there exists an equivalent DFTA.*

The algorithm for constructing a DFTA from an NFTA is similar to the classical subset construction algorithm for finite state automata (Aho et al., 1985). It groups together the non-deterministic states that have common terms reaching them and treats the groups as deterministic states. The final deterministic states are the groups containing at least one non-deterministic final state.

The union, intersection, complement, and difference of tree languages are still tree languages. There is also a corresponding pumping lemma for tree languages, a useful corollary of which is that testing emptiness of a tree language is decidable.

THEOREM 2. *The class of tree languages is closed under union, intersection, complement, and difference.*

THEOREM 3. *Let $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ be a DFTA, then $L(\mathcal{A})$ is not empty if and only if there exists a term t in $L(\mathcal{A})$ with Height$(t) \leq |Q|$, where Height(t) is the height of t's tree representation and $|Q|$ is the number of states in Q.*

4. Algorithms for Proving Unique Decomposition

Our approach for automatically proving unique-decomposition lemmas consists of translating the regular tree grammars for both terms and decompositions into finite tree automata, and checking the existence and the uniqueness of decompositions using algorithms on finite tree automata algorithm.

4.1. Building Finite Tree Automata

Each regular tree grammar can be mapped to a finite tree automaton using the following translation. Our algorithms for checking decompositions are specialized to the output of this translation.

- If there are alternative production rules for the same non-terminal, introduce a non-terminal for each alternative, and add a production rule from the original non-terminal to the newly introduced ones. The reason alternative productions are distinguished in this way is to simplify the check for non-overlapping productions required for proving uniqueness. For example, the production rules of M in Example 1 are turned into:

$$\begin{array}{lll} M \rightarrow M_1 & \qquad & M_1 \rightarrow \mathsf{var} \\ M \rightarrow M_2 & & M_2 \rightarrow \mathsf{lam}(\mathsf{var}, M) \\ M \rightarrow M_3 & & M_3 \rightarrow \mathsf{app}(M, M) \end{array}$$

- Take apart the right-hand sides of the production rules in such a way that each right-hand side is a non-terminal, a constructor of arity zero or an application of a constructor to non-terminals. For example, the production rule $M_2 \rightarrow \mathsf{lam}(\mathsf{var}, M)$ is turned into $M_2 \rightarrow \mathsf{lam}(M_4, M)$ and $M_4 \rightarrow \mathsf{var}$.

- Reverse the arrows of the production rules, and introduce a state for each non-terminal. The state corresponding to the start symbol is the final state.

Example 3 shows the automaton translated from the regular tree grammar for λ-calculus from Example 1. The automaton for decompositions built from the tree grammar in Example 2 is given in Figure 1, where q_M is the final state.

The automaton produced by the above translation is usually non-deterministic since most grammars have alternatives and the alternatives create ϵ-transitions. This non-deterministic automaton is then converted to a deterministic automaton. It is during this translation

$$
\begin{array}{rclcrclcrcl}
q_{M_1} & \xrightarrow{\epsilon} & q_M & & q_H & \xrightarrow{\epsilon} & q_{M_1} & & & & \\
q_{M_2} & \xrightarrow{\epsilon} & q_M & & q_U & \xrightarrow{\epsilon} & q_{M_2} & & & & \\
& & & & & & & & & & \\
q_{H_1} & \xrightarrow{\epsilon} & q_H & & q_I & \rightarrow & q_{H_1} & & & & \\
q_{H_2} & \xrightarrow{\epsilon} & q_H & & \mathsf{lam}(q_{H_3}, q_H) & \rightarrow & q_{H_2} & & \mathsf{var} & \rightarrow & q_{H_3} \\
& & & & & & & & & & \\
q_{I_1} & \xrightarrow{\epsilon} & q_I & & \mathsf{var} & \rightarrow & q_{I_1} & & & & \\
q_{I_2} & \xrightarrow{\epsilon} & q_I & & \mathsf{app}(q_I, q_M) & \rightarrow & q_I & & & & \\
& & & & & & & & & & \\
q_{U_1} & \xrightarrow{\epsilon} & q_U & & q_R & \rightarrow & q_{U_1} & & & & \\
q_{U_2} & \xrightarrow{\epsilon} & q_U & & \mathsf{lam}(q_{U_3}, q_U) & \rightarrow & q_{U_2} & & \mathsf{var} & \rightarrow & q_{U_3} \\
& & & & & & & & & & \\
q_{R_1} & \xrightarrow{\epsilon} & q_R & & \mathsf{app}(q_{R_3}, q_M) & \rightarrow & q_{R_1} & & & & \\
q_{R_2} & \xrightarrow{\epsilon} & q_R & & \mathsf{app}(q_R, q_M) & \rightarrow & q_{R_2} & & & & \\
\mathsf{lam}(q_{R_4}, q_M) & \rightarrow & q_{R_3} & & \mathsf{var} & \rightarrow & q_{R_4} & & & &
\end{array}
$$

Figure 1. An automaton for the grammar of Example 2

that the checking of uniqueness of decomposition is made. The existence of a decomposition is instead carried out on the deterministic automaton.

4.2. Checking Existence of Decompositions

Given two deterministic automata: $\mathcal{A}_1$ for the terms and $\mathcal{A}_2$ for the decompositions, we can answer whether all terms have decompositions by checking whether $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$, which is equivalent to checking whether $L(\mathcal{A}_1) - L(\mathcal{A}_2)$ is empty. By Theorem 2, there exists an automaton $\mathcal{A}_1 - \mathcal{A}_2$ whose language is $L(\mathcal{A}_1) - L(\mathcal{A}_2)$. In order to check the emptiness of $L(\mathcal{A}_1) - L(\mathcal{A}_2)$, according to Theorem 3, it is sufficient to check whether the automaton $\mathcal{A}_1 - \mathcal{A}_2$ accepts a term with height less than or equal to its number of states. We thus generate all such terms and check whether they are accepted. If none is accepted, then $L(\mathcal{A}_1 - \mathcal{A}_2)$ is empty, *i.e.*, $L(\mathcal{A}_1) \subseteq L(\mathcal{A}_2)$, and all terms have decompositions. Otherwise, if a term is accepted by the automaton, then it is a counterexample without decomposition.

The number of terms being tested is $O(|\Sigma|^{m^k})$, where $|\Sigma|$ is the number of constructors, m is the maximal arity of the constructors in Σ, and k is the number of states in the automaton $\mathcal{A}_1 - \mathcal{A}_2$. The upper bound on k is obtained as follows: $\mathcal{A}_1 - \mathcal{A}_2$ is equivalent to $\mathcal{A}_1 \cap \overline{\mathcal{A}_2}$, where $\overline{\mathcal{A}_2}$ is the complement of $\mathcal{A}_2$. Suppose $\mathcal{A}_1$ has k_1 states and $\mathcal{A}_2$

has k_2 states, then by standard algorithms, $\overline{\mathcal{A}_2}$ has k_2+1 states, $\mathcal{A}_1 \cap \overline{\mathcal{A}_2}$ has at most $k_1 * (k_2+1)$ states. Usually k_1 is greater than $|\Sigma|$, and k_2 may be much greater. Therefore, k is greater than $|\Sigma|^2$ in most cases, and the algorithm is exponential in the number of constructors.

4.3. Checking Uniqueness of Decompositions

Proving uniqueness corresponds to checking the non-ambiguity of the grammar for the decomposition patterns. If the grammar is ambiguous, there exists a term t with more than one parse tree, which can only happen due to the presence of alternative productions for the same non-terminal. Suppose that, in the non-deterministic automaton translated from the grammar, states q_1 and q_2 correspond to those two alternatives. This means that there are two successful runs of the non-deterministic automaton on t that map a subterm of t to two different ϵ-transition closures containing q_1 and q_2, respectively. In other words, the sets of terms reaching states q_1 and q_2 have a non-empty intersection. Such states, reached by overlapping sets of terms, can be found during the translation of the non-deterministic automaton into a deterministic one, as the translation naturally groups non-deterministic states by the common terms that can reach them.

To formalize the above algorithm, we first define conflict sets.

Definition 7. Given an automaton $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$, a *conflict set* of $\mathcal{A}$ is a set C of at least two states, and a special state $q \in Q$ such that:

$$q_c \in C \text{ if and only if } q_c \xrightarrow{\epsilon} q \in \Delta$$

DFTAs do not have conflict sets since they do not have ϵ-transition rules. For general non-deterministic automata, conflict sets may exist for many reasons. But for the non-deterministic automata produced by our translation from regular tree grammars (as described in Section 4.1), conflict sets correspond to alternatives for non-terminals. For example, the only conflict set of the automaton for lambda-calculus given in Example 3 is $\{q_{M_1}, q_{M_2}, q_{M_3}\}$; the automaton for the decompositions in Figure 1 has $\{q_{M_1}, q_{M_2}\}$, $\{q_{H_1}, q_{H_2}\}$, $\{q_{I_1}, q_{I_2}\}$, $\{q_{U_1}, q_{U_2}\}$, and $\{q_{R_1}, q_{R_2}\}$ as conflict sets.

Because of the special constraints on automata imposed by our construction, we can check uniqueness of decompositions by keeping track of conflict sets while translating NFTAs to DFTAs. Our algorithm checks whether a deterministic state contains more than one non-deterministic state from the same conflict set. If there does not exist such a deterministic state, the grammar is not ambiguous, hence the decomposition for any term (if it has one) is unique. Otherwise, counterexamples violating uniqueness can be obtained by the following steps: Let $\mathcal{A}_N = (\Sigma, Q_N, Q_{Nf}, \Delta_N)$ be an NFTA and $\mathcal{A}_D = (\Sigma, Q_D, Q_{Df}, \Delta_D)$ be the DFTA translated from $\mathcal{A}_N$.

1. For each deterministic state, construct a term that reaches it. Such a term is also called *instance term* for the state. The instance terms are constructed bottom-up:

   a) If a deterministic state g has no instance term, and if there exists $c^0 \to g \in \Delta_D$, then c^0 is an instance term of g.

   b) If a deterministic state g has no instance term, states $g_1, \ldots, g_n$ have instance terms $t_1, \ldots, t_n$ respectively, and if there exists $c^n(g_1, \ldots, g_n) \to g \in \Delta_D$, then $c^n(t_1, \ldots, t_n)$ is an instance term of g.

   c) Repeat a) and b) above until all deterministic states have one instance term each.

2. Suppose a deterministic state g contains q_1 and q_2 which are from the same conflict set, and g has an instance term t. Let q be the state associated with the conflict set, *i.e.*, $q_1 \xrightarrow{\epsilon} q \in \Delta_N$ and $q_2 \xrightarrow{\epsilon} q \in \Delta_N$. Initiate a queue L with an element (q, t). Repeat the following steps until a counterexample is found:

   a) Get the first element out from L, let it be (q', t').

   b) If $q' \in Q_{Nf}$ then the term t' is the counterexample.

   c) Otherwise:

      – For each transition rule $c^n(p_1, \ldots, p_n) \to q'' \in \Delta_N$ where $q' = p_i$, let g_j be the deterministic state containing p_j for each $j \neq i$. Construct a term $t'' = c^n(t_1, \ldots, t_n)$, where t_j is the instance term of g_j for $j \neq i$ and $t_i = t'$. Append (q'', t'') to the queue L.

      – For each transition rule $q' \xrightarrow{\epsilon} q'' \in \Delta_N$, append (q'', t') into the queue L.

In the second step, we construct terms containing t as a subterm in a bottom-up fashion following the productions of the automaton $\mathcal{A}_N$. Runs of $\mathcal{A}_N$ on all the constructed terms map a subterm t to $cls_\epsilon\{q_1\}$ or $cls_\epsilon\{q_2\}$. The construction ends when we obtain an instance term for a deterministic final state. Termination is guaranteed since we assume that all non-terminals can be derived from the start symbol of the grammar and our traversing of the productions of $\mathcal{A}_N$ is in breadth-first order.

The grammars we have seen so far are non-ambiguous. We show in the following example how the algorithm works on an ambiguous grammar.

EXAMPLE 4. *Consider the following regular tree grammar:*

$$\begin{array}{lcl} S & \rightarrow & \mathsf{not}(\mathsf{false}) \mid \mathsf{not}(A) \\ A & \rightarrow & \mathsf{or}(\mathsf{true}, B) \mid \mathsf{or}(B, \mathsf{true}) \\ B & \rightarrow & \mathsf{true} \mid \mathsf{false} \end{array}$$

A NFTA for the grammar is:

$$\begin{array}{lcl@{\qquad}rcl@{\qquad}rcl} q_{S_1} & \xrightarrow{\epsilon} & q_S & \mathsf{not}(q_{S_3}) & \rightarrow & q_{S_1} & \mathsf{false} & \rightarrow & q_{S_3} \\ q_{S_2} & \xrightarrow{\epsilon} & q_S & \mathsf{not}(q_A) & \rightarrow & q_{S_2} & & & \\ \\ q_{A_1} & \xrightarrow{\epsilon} & q_A & \mathsf{or}(q_{A_3}, q_B) & \rightarrow & q_{A_1} & \mathsf{true} & \rightarrow & q_{A_3} \\ q_{A_2} & \xrightarrow{\epsilon} & q_A & \mathsf{or}(q_B, q_{A_4}) & \rightarrow & q_{A_2} & \mathsf{true} & \rightarrow & q_{A_4} \\ \\ q_{B_1} & \xrightarrow{\epsilon} & q_B & \mathsf{true} & \rightarrow & q_{B_1} & & & \\ q_{B_2} & \xrightarrow{\epsilon} & q_B & \mathsf{false} & \rightarrow & q_{B_2} & & & \end{array}$$

where the conflict sets are $\{q_{S_1}, q_{S_2}\}$, $\{q_{A_1}, q_{A_2}\}$ *and* $\{q_{B_1}, q_{B_2}\}$.

The above NFTA is translated into the following DFTA:

$$\begin{array}{rclcl} \mathsf{true} & \rightarrow & g_1 & = & \{q_{A_3}, q_{A_4}, q_{B_1}, q_B\} \\ \mathsf{false} & \rightarrow & g_2 & = & \{q_{S_3}, q_{B_2}, q_B\} \\ \mathsf{or}(g_1, g_1) & \rightarrow & g_3 & = & \{q_{A_1}, q_{A_2}, q_A\} \\ \mathsf{or}(g_1, g_2) & \rightarrow & g_4 & = & \{q_{A_1}, q_A\} \\ \mathsf{or}(g_2, g_1) & \rightarrow & g_5 & = & \{q_{A_2}, q_A\} \\ \mathsf{not}(g_2) & \rightarrow & g_6 & = & \{q_{S_1}, q_S\} \\ \mathsf{not}(g_3) & \rightarrow & g_7 & = & \{q_{S_2}, q_S\} \\ \mathsf{not}(g_4) & \rightarrow & g_7 & & \\ \mathsf{not}(g_5) & \rightarrow & g_7 & & \end{array}$$

The deterministic state g_3 *contains* q_{A_1} *and* q_{A_2} *which are from the same conflict set. The state* g_3 *has an instance term* $\mathsf{or}(\mathsf{true}, \mathsf{true})$. *We need to construct a term that is accepted by the automaton and that contains the instance term. The state* q_A *is the special state associated with the conflict set* $\{q_{A_1}, q_{A_2}\}$. *By following the non-deterministic transition rule* $\mathsf{not}(q_A) \rightarrow q_{S_2}$, *we obtain the term* $\mathsf{not}(\mathsf{or}(\mathsf{true}, \mathsf{true}))$ *that can reach* q_{S_2}. *Then by following the rule* $q_{S_2} \xrightarrow{\epsilon} q_S$, *the term can also reach the final state* q_S, *i.e., the term* $\mathsf{not}(\mathsf{or}(\mathsf{true}, \mathsf{true}))$ *is a counterexample to uniqueness.*

5. Extensions

Regular tree grammars are expressive enough for specifying terms and decompositions built only from constructors. However, in specifying unique-decomposition lemmas, we often use contexts and polymorphic types. These semantic notions cannot be directly defined by regular tree grammars. In order to handle definitions that use contexts and polymorphic types, we extend the definition of tree automata with variables that stand for unknown subterms, and identify necessary restrictions for maintaining decidability.

We first give the simple extensions by considering built-in constants and tuples, then we explain how to build automata for contexts and polymorphic types.

5.1. Built-in Constants

Some built-in types such as `int` and `string` are infinite. As we did previously, we can collapse an infinite type into a constant. For example, we collapse `int` into int and `string` into str. The transition rule $\mathsf{int} \to q$ means that all integer numbers can reach state q. It is a substitute for an infinite number of transition rules $\mathsf{0} \to q$, $\mathsf{1} \to q$, ..., $n \to q$. In this way, we can use finite constructors and transition rules to recognize elements of infinite types. This method works for specifications without specific built-in constants such as the examples considered so far, but the following definition requires distinguishing $\mathsf{0}$ from other constants.

EXAMPLE 5. *The following grammar defines integer lists with at most one occurrence of* $\mathsf{0}$*, and the occurrence can be only at the head.*

$$\begin{array}{rcl} G & \to & \mathsf{cons}(\mathsf{int}, I) \\ I & \to & \mathsf{nil} \mid \mathsf{cons}(\mathsf{n}, I) \;\; \textit{where } \mathsf{n} \neq \mathsf{0} \end{array}$$

Our solution is to divide the infinite set: the constants appearing in the specification are kept distinct; those that do not appear in the specification are collapsed into one special constant. The special constants are represented as $\mathsf{allexcept}(T, \{\mathsf{c}_1, \ldots, \mathsf{c}_\mathsf{n}\})$, where $\mathsf{allexcept}$ is a tag, T is an infinite built-in type, and $\mathsf{c}_1, \ldots, \mathsf{c}_\mathsf{n}$ are constants in the type T appearing in the specification. In other words, we add a production rule

$$T \to \mathsf{c}_1 \mid \cdots \mid \mathsf{c}_\mathsf{n} \mid \mathsf{allexcept}(T, \{\mathsf{c}_1, \ldots, \mathsf{c}_\mathsf{n}\})$$

in the grammars and move T from the set of terminals to the set of non-terminals. This change preserves the set of terms that can be derived from the grammar.

In this setting, an automaton for the grammar in Example 5 is:

$$
\begin{array}{rclcrcl}
\mathsf{cons}(q_{int}, q_I) & \rightarrow & q_G & & & & \\
0 & \rightarrow & q_{int} & & \mathsf{allexcept}(int, \{0\}) & \rightarrow & q_{int} \\
\mathsf{nil} & \rightarrow & q_I & & \mathsf{cons}(q_{I_1}, q_I) & \rightarrow & q_I \\
\mathsf{allexcept}(int, \{0\}) & \rightarrow & q_{I_1} & & & &
\end{array}
$$

5.2. Tuples

Tuples are frequently used in syntactic definitions. To build automata for definitions with tuples, we introduce a new constructor tuple. The arity of tuple is not fixed. The constructor tuple stands for any constructor tuple^2, tuple^3, ..., tuple^n, where tuple^n has arity n, $n \geq 2$.

EXAMPLE 6. *A pair of boolean values can be decomposed into four cases. The grammars are specified as follows:*

$$
\begin{array}{lcl}
G_1 & \rightarrow & (bool, bool) \\
G_2 & \rightarrow & (\mathsf{true}, \mathsf{true}) \mid (\mathsf{true}, \mathsf{false}) \mid (\mathsf{false}, \mathsf{true}) \mid (\mathsf{false}, \mathsf{false})
\end{array}
$$

The automaton for G_1 has the following transition rules, where q_{G_1} is the final state:

$$
\mathsf{tuple}(q_{bool}, q_{bool}) \rightarrow q_{G_1} \qquad \mathsf{true} \rightarrow q_{bool} \qquad \mathsf{false} \rightarrow q_{bool}
$$

5.3. Contexts

Contexts are usually used to identify the location of a subterm in a term. For instance, a call-by-value computation consists of successively applying the β_v-reduction rule to any subterm. In order to define a deterministic evaluation function (*i.e.*, an interpreter), we restrict the application of the reduction rule to a unique subterm of a program whose position is determined using an evaluation context.

EXAMPLE 7. *The set of call-by-value evaluation contexts has the following definition:*

$$
E ::= [\,] \mid E\ M \mid (\lambda x.M)\ E
$$

where M stands for the set of λ-terms defined in Example 1.

The $[\,]$ represents a "hole". If E is a context, then $E[t]$ denotes the term that results from placing t in the hole of E.

5.3.1. *Abstract Automata*

Our approach is to represent the holes as (meta-)variables and to represent context as terms with (meta-)variables. (For a more complete investigation of how to compute with contexts in general settings, we refer to the exposition in "Computing with Contexts" (Mason, 1999).) The finite tree automata introduced in Section 3 cannot handle terms with (meta-)variables. To that end, we introduce a new representation of a tree automaton, which is a quadruple $(\Sigma, (Q, \Phi_c), Q_f, \Delta)$, where Φ_c is a (finite) set of the states that do not appear in the right-hand side of any transition rule. The sets Q and Φ_c are disjoint, and their union is the set of all states in the automaton. We call the states in Φ_c *state variables*, and call automata with a non-empty set of state variables *abstract automata*. All the theorems in Section 3 still hold with the presence of Φ_c. The definition of run will then map a variable occurring in a term to the ϵ-transition closure of a state variable. Different variables should be mapped to different closures; different occurrences of the same variable should be mapped to the same closure.

EXAMPLE 8. *An abstract automaton for the evaluation contexts E of Example 7 is (the transition rules for q_M are omitted):*

$$\begin{array}{rclrcl}
q_{E_1} & \xrightarrow{\epsilon} & q_E & \alpha & \xrightarrow{\epsilon} & q_{E_1} \\
q_{E_2} & \xrightarrow{\epsilon} & q_E & \mathsf{app}(q_E, q_M) & \rightarrow & q_{E_2} \\
q_{E_3} & \xrightarrow{\epsilon} & q_E & \mathsf{app}(q_{E_4}, q_E) & \rightarrow & q_{E_3} \\
\mathsf{lam}(q_{E_5}, q_M) & \rightarrow & q_{E_4} & \mathsf{var} & \rightarrow & q_{E_5}
\end{array}$$

where α is the only state variable, $\{q_{E_1}, q_{E_2}, q_{E_3}\}$ is the conflict set. A context $(\lambda x.x)\ [\,]$ can be represented as the term $\mathsf{app}(\mathsf{lam}(\mathsf{var}_1, \mathsf{var}_2), \alpha)$. *A successful run r of the automaton in this term is:*

$$\begin{array}{rcl}
r(\mathsf{var}_1) & = & \{q_{E_5}\} \\
r(\mathsf{var}_2) & = & \{q_{M_1}, q_M\} \\
r(\mathsf{lam}(\mathsf{var}_1, \mathsf{var}_2)) & = & \{q_{E_4}\} \\
r(\alpha) & = & \{\alpha, q_{E_1}, q_E\} \\
r(\mathsf{app}(\mathsf{lam}(\mathsf{var}_1, \mathsf{var}_2), \alpha)) & = & \{q_{E_3}, q_E\}
\end{array}$$

The operation $E[t]$, that fills a context E with term t, corresponds to instantiating the state variables of the automaton for E as follows. Instantiations add ϵ-transitions from the final states in the automaton for the term t to the state variables which become normal states of the automaton for E. In order to avoid conflicts in state names, instantiations usually involve replications of the automata. Given an abstract automaton $\mathcal{A} = (\Sigma, (Q, \Phi_c), Q_f, \Delta)$, $\{\alpha_1, \ldots, \alpha_n\} \subseteq \Phi_c$, and the automata $\mathcal{A}_i = (\Sigma_i, (Q_i, \Phi_{ci}), Q_{fi}, \Delta_i)$ for each subterm t_i, $1 \leq i \leq n$, instantiating α_i with t_i, for all $1 \leq i \leq n$, involves the following steps:

1. Make a copy of $\mathcal{A}$, obtaining $\mathcal{A}' = (\Sigma', (Q', \Phi'_c), Q'_f, \Delta')$.
2. Rename states in $\mathcal{A}_i$ for $1 \le i \le n$ if necessary, so that there is no name conflict among states.
3. Combine $\mathcal{A}'$ and $\mathcal{A}_i$, for $1 \le i \le n$, but keep Q'_f as the set of final states.
4. Add transition rules $q_{fi} \xrightarrow{\epsilon} \alpha'_i$, where $q_{fi} \in Q_{fi}$. Move $\alpha'_1, \ldots, \alpha'_n$ from the set of state variables to the set of normal states.

The resulting automaton has the following components:

- constructors: $\Sigma' \cup \bigcup_{i=1}^{n} \Sigma_i$
- states: $(Q' \cup \{\alpha'_1, \ldots, \alpha'_n\} \cup \bigcup_{i=1}^{n} Q_i, (\Phi'_c - \{\alpha'_1, \ldots, \alpha'_n\}) \cup \bigcup_{i=1}^{n} \Phi_{ci})$
- final states: Q'_f
- transition rules: $\Delta' \cup \bigcup_{i=1}^{n} \Delta_i \cup \bigcup_{i=1}^{n} \{q_{fi} \xrightarrow{\epsilon} \alpha'_i \mid q_{fi} \in Q_{fi}\}$

EXAMPLE 9. *An automaton for $E[\mathsf{var}]$ is (the transition rules for q'_M are omitted):*

$$
\begin{array}{rclrcl}
q'_{E_1} & \xrightarrow{\epsilon} & q'_E & \alpha' & \xrightarrow{\epsilon} & q'_{E_1} \\
q'_{E_2} & \xrightarrow{\epsilon} & q'_E & \mathsf{app}(q'_E, q'_M) & \to & q'_{E_2} \\
q'_{E_3} & \xrightarrow{\epsilon} & q'_E & \mathsf{app}(q'_{E_4}, q'_E) & \to & q'_{E_3} \\
\mathsf{lam}(q'_{E_5}, q'_M) & \to & q'_{E_4} & \mathsf{var} & \to & q'_{E_5} \\
 & & & & & \\
\mathsf{var} & \to & q_{E_6} & q_{E_6} & \xrightarrow{\epsilon} & \alpha'
\end{array}
$$

where q'_E is the final state. $\{q'_{E_1}, q'_{E_2}, q'_{E_3}\}$ is the conflict set.

Notice that the conflict sets are also copied because alternatives in a context definition are still alternatives when the hole is filled with a term.

5.3.2. *From Context Definitions to Automata*

General context definitions do not belong to the class of regular tree grammars. For example, the context definition:

$$E ::= \mathsf{b}([\,]) \mid \mathsf{a}(E[\mathsf{c}([\,])])$$

where a,b, and c are unary constructors, defines contexts $\mathsf{a}^n(\mathsf{b}(\mathsf{c}^n([\,])))$. This language is not accepted by any regular tree grammar. It is however accepted by a *context-free tree grammar*. A context-free tree grammar is a tree grammar (Σ, N, S, P), where the rules have the form $L(\alpha_1, \ldots, \alpha_n) \to t$, where L is a non-terminal and $t \in \mathcal{T}(\Sigma \cup N \cup \{\alpha_1, \cdots, \alpha_n\})$.

EXAMPLE 10. *A context-free tree grammar for* $\mathsf{a}^n(\mathsf{b}(\mathsf{c}^n([\,])))$ *is:*

$$E(\alpha) \to \mathsf{b}(\alpha) \mid \mathsf{a}(E(\mathsf{c}(\alpha)))$$

As expected, the equivalence and ambiguity of context-free tree grammars is undecidable (Comon et al., 1999). Fortunately, context definitions are in a special subclass where the right-hand sides of production rules are restricted to $\mathcal{T}(\Sigma \cup N, \Phi)$ which, with some further restrictions, can be transformed to productions for regular tree grammars. For instance, the productions for the evaluation contexts E of Example 7 are:

$$E(\alpha) \to \alpha \mid \mathsf{app}(E(\alpha), M) \mid \mathsf{app}(\mathsf{lam}(\mathsf{var}, M), E(\alpha))$$

If we regard $E(\alpha)$ as a single non-terminal, the grammar becomes a regular tree grammar. We can then build an automaton for $E(\alpha)$, as shown in Example 8.

The following shows how we transform context definitions:

1. Introduce a variable for the hole of each context. In this way, each context is associated with a variable. Then, replace all occurrences of the holes by the corresponding variables, and replace all occurrences of unfilled contexts by filling the contexts with the corresponding variables. The production rules in the resulting context-free tree grammar G are grouped as follows:

$$\begin{array}{rcl} id_1(\alpha_{11}, \ldots, \alpha_{1k_1}) & \to & r_{11} \mid \cdots \mid r_{1m_1} \\ \vdots & & \vdots \\ id_n(\alpha_{n1}, \ldots, \alpha_{nk_n}) & \to & r_{n1} \mid \cdots \mid r_{nm_n} \end{array}$$

2. In each right-hand side r_{ij}, search for occurrences of $id_l(t_1, \ldots, t_{k_l})$ that do not match any left-hand side. For each such occurrence add the following production rules:

$$id_l(t_1, \ldots, t_{k_l}) \to r'_{l1} \mid \cdots \mid r'_{lm_l}$$

   where r'_{lh} is the substitution of $\alpha_{l1}, \ldots, \alpha_{lk_l}$ by $t_1, \ldots, t_{k_l}$ in r_{lh}.

3. Consider the left-hand side of each production rule as a single non-terminal, and do the appropriate renaming.

This process (when it terminates—see next paragraph) produces a grammar G' which differs from a regular tree grammar only in allowing the subterms $id_l(t_1, \ldots, t_{k_l})$ in the right-hand sides of the production rules. Fortunately, we can build automata for these subterms by instantiating the abstract automata for $id_l(\alpha_{l1}, \ldots, \alpha_{lk_l})$. Hence, we can build an automaton for the grammar G'.

Expansions in the second step may not terminate. To gain more intuition, consider the expansion of $E(\mathsf{c}(\alpha))$ in Example 10, which creates

$$E(\mathsf{c}(\alpha)) \rightarrow \mathsf{b}(\mathsf{c}(\alpha)) \mid \mathsf{a}(E(\mathsf{c}(\mathsf{c}(\alpha))))$$

where $E(\mathsf{c}(\mathsf{c}(\alpha)))$ is to be expanded.

Therefore, constraints are needed for transforming context-free tree grammars. Given a context-free tree grammar (Σ, N, S, P), for any recursive non-terminal application $id_l(t_1, \ldots, t_{k_l})$ in the right-hand side of a production rule $id_i(\alpha_{i1}, \ldots, \alpha_{ik_i}) \rightarrow r$, each t_h for $1 \leq h \leq k_l$ should be either a variable or a term in $\mathcal{T}(\Sigma \cup N_i)$ where N_i is the set of non-terminals not recursively defined with id_i.

These constraints guarantee termination of the expansion of context-free tree grammars described. First, the arguments of recursive non-terminal applications do not contain recursive non-terminal applications, so that we do not expand arguments. Second, the arguments do not contain variables. When a group of production rules is instantiated, the arguments of recursive non-terminal applications in the resulting right-hand sides are either in the original right-hand sides or replacements for variables. In other words, all arguments in the generated production rule are either variables or the arguments in the original context-free tree grammar. The combination of the arguments can be exhausted since both sets of variables and original arguments used in the grammar are finite. Therefore, there is no infinite expansion.

The constraints are not necessary conditions for being able to build automata for contexts, but they are sufficient. Moreover, context definitions under our constraints are expressive enough for production rules in most grammars of interest. Most context definitions in programming language semantics satisfy these constraints.

5.4. Polymorphism

A polymorphic type declaration defines a parametric type, where the parameters are the type variables. A monomorphic type has no parameter. We use state variables to represent type variables. The automata for polymorphic types are abstract automata as well, and type instantiation corresponds to linking the state variables. We add one more component Φ_t to the tuple representation of automata which is now $(\Sigma, (Q, \Phi_t, \Phi_c), Q_f, \Delta)$, where Φ_t, Φ_c are the sets of state variables for polymorphic types and contexts, respectively.

Polymorphic type definitions also belong to context-free tree grammars. Building automata for polymorphic types is similar to building automata for contexts and requires the same constraints.

For example, a polymorphic type $List$ is defined as follows:

$$List(\alpha) \;\rightarrow\; \mathsf{nil} \mid \mathsf{cons}(\alpha, List(\alpha))$$

The automaton for the type `List` above is:

$$\mathsf{nil} \rightarrow q_{List} \qquad \mathsf{cons}(\alpha, q_{List}) \rightarrow q_{List}$$

where α is the state variable for the type variable α.
The automaton for the type `int List` is:

$$\begin{array}{rclcrcl} \mathsf{nil} & \rightarrow & q'_{List} & \qquad & \mathsf{cons}(\alpha', q'_{List}) & \rightarrow & q'_{List}, \\ q_{int} & \rightarrow & \alpha' & & \mathsf{allexcept}(int, \{\}) & \rightarrow & q_{int} \end{array}$$

where the automaton for the type int is linked to the state α'.

6. The SL System

We developed a prototype system that proves unique-decomposition lemmas as we proposed in Section 4 and Section 5. The system is an extension of the SL system (Xiao et al., 2000) that generates interpreters from syntactic theories. We extend the SL language with phrases specifying unique-decomposition lemmas, and modify the SL compiler to check the validity of the lemmas before generating interpreters.

To explain the SL system and the extension, we use the call-by-value λ-calculus as our running example in this section.

6.1. The Call-by-value Lambda Calculus

The set of terms of the call-by-value λ-calculus is generated inductively over an infinite set of variables $Vars$ (ranged over by x, y, etc): it includes integer constants (ranged over by n), λ-abstractions, and applications:

$$\begin{array}{lrcl} (\mathit{Expressions}) & M & ::= & n \mid x \mid \lambda x.M \mid MM \\ & x & \in & \mathit{Vars} \\ & n & \in & \mathit{Int} \end{array}$$

The semantics is based on the β_v reduction rule which requires a syntactic definition of the notion of value:

$$\begin{array}{rrcl} (\mathit{Values}) & V & ::= & n \mid \lambda x.M \\ (\beta_v) & (\lambda x.M)V & \rightarrow & M[x := V] \end{array}$$

where $M[x := V]$ is the term resulting from the substitution of all free occurrences of variable x with V.

Call-by-value evaluation contexts have the following definition:

$$E ::= [\,] \mid EM \mid VE$$

They are used to define a stepping relation on programs, denoted by $\mapsto$, as follows:

$$\frac{M \to M'}{E[M] \mapsto E[M']}$$

At each step of evaluation of a program, we attempt to *decompose* the program into an evaluation context and a redex and rewrite the redex using the β_v rule. This idea can be used to automatically generate an interpreter.

6.2. Generating Interpreters from Syntactic Theories

The SL system is a semantic toolkit for manipulating syntactic theories: its most basic function is to produce interpreters based on specifications of syntactic theories. Currently the generated interpreters are in ML; the semantic specification is written in a domain-specific language that builds on ML.

The semantics of our running call-by-value example can be specified as in Figure 2.

The `SIGNATURE` part describes the abstract syntax of the language using ML type definitions. In general the type definitions may be polymorphic but are restricted to first-order type expressions (no function types). To account for cases in which the description needs more than one datatype, the start symbol of the grammar is explicitly given by the `startfrom` phrase.

The `SPECIFICATION` part describes the semantics of the language at essentially the same level of abstraction used in Section 6.1. A dynamic definition defines a subset of a datatype with semantic significance. The language has a primitive notion of contexts with `BOX` as the empty context. The definitions of axioms and inference rules use a richer notion of pattern-matching than the one used in most functional languages: they include dynamic constraints like in `t2:V`, and context constraints and context fillings like in `e:E` and `e t2`. Meta-operations of the semantics like substitution can be written directly in ML as auxiliary definitions.

As is apparent the SL system is *very* domain-specific, targeting exactly the kind of semantic specifications that are based on syntactic theories. It includes the right kind of primitive notions, *e.g.*, contexts, axioms, inference rules, and dynamic constraints. In addition, it performs basic checks to ensure the specifications are well-formed. Other

```
SIGNATURE:
type M = Num of int | Var of string
       | Lam of string*M | App of M*M;;

startfrom M;;

SPECIFICATION:
dynamic V = Num _ | Lam(_,_);;

let rec subst (t1,x,t2) = ...
  (* ML code implementing the substitution of all free
     occurrences of x in t1 with t2, omitted *);;

axiom betav:
App(Lam(x,t1), (t2:V)) ==> subst(t1, x, t2);;

context E =  BOX | App(E,_) |  App(V,E);;

inference eval:
t1 ==> t2
-------------------
(e:E) t1 |==> e t2
;;
```

Figure 2. SL Specification of a Simple CBV Language

than the basic syntactic checks, the SL system has a (meta-) type system that ensures that contexts are used appropriately, *e.g.*, every context has one hole, contexts are filled with expressions of the appropriate types, and both sides of each axiom have the same type. After performing these basic checks, the SL system compiles the specification into a non-deterministic automaton, which is then transliterated into ML code. The code uses success continuations for encoding the sequencing of states in the automaton, and exceptions with handlers for encoding the non-deterministic selection of a state in the automaton.

Feeding the code in Figure 2 to the SL system produces an interpreter. This interpreter can then be invoked on terms of the language to evaluate them by repeatedly decomposing terms into evaluation contexts and redexes, and contracting the redexes until it reaches an answer. For example, if the input file contains:

```
App(Lam("y",Var "y"), App( Lam("x",Var "x"), Num 3));;
```

the interpreter produces:

```
App (Lam ("y", Var "y"), App (Lam ("x", Var "x"), Num 3))
 ==>    by betav,eval,eval
App (Lam ("y", Var "y"), Num 3)
 ==>    by betav,eval
Num 3
```

6.3. Extending the SL system to Prove Unique Decomposition Lemmas

The interpreters generated by the SL system are not guaranteed to specify partial functions, and might get stuck on some programs as evaluation may reach terms for which no decomposition exists. For our running example, the terms x and $(n(\lambda x.x))$ cannot be decomposed into an evaluation context filled with a *redex* (a term matching the left-hand side of β_v rule). The existence of terms that have no decomposition indicates either a mistake in the definitions, or situations that need to be accounted for using a type system or run-time checks. In both cases, it is useful to have a precise statement that lists all the possible decompositions for a term: every term can be decomposed as a value (*i.e.*, a final answer that needs no further evaluation) or an evaluation context filled with a current instruction. The current instruction could be a valid redex or indicate an erroneous program that contains either a reference to an unbound variable or a faulty expression. The formal definitions are given below:

$$
\begin{array}{lrcl}
(\textit{Decomposed Terms}) & D & ::= & V \mid E[x] \mid E[R] \mid E[F] \\
(\textit{Values}) & V & ::= & n \mid \lambda x.M \\
(\textit{Redexes}) & R & ::= & (\lambda x.M)V \\
(\textit{Faulty Terms}) & F & ::= & nM \\
(\textit{Evaluation Contexts}) & E & ::= & [\,] \mid EM \mid VE
\end{array}
$$

To automatically check unique decomposition, we introduce a new phrase for SL specifications:

$$\texttt{decomp}\ T\ \texttt{of}\ p_1\ \texttt{|}\ \ldots\ \texttt{|}\ p_n\ \texttt{;;}$$

```
SIGNATURE:
type M = Num of int | Var of string
       | Lam of string*M | App of M*M;;

startfrom M;;

SPECIFICATION:

... (* other parts of the specification, omitted *)

dynamic V = Num _ | Lam(_,_);;
dynamic R = App(Lam(_,_),V);;
dynamic F = App(Num _,_);;

context E =  BOX | App(E, _) |  App(V, E);;

decomp M of (_:V) | (_:E)(Var _)
          | (_:E)(_:R) | (_:E)(_:F);;
```

Figure 3. SL Specification of Decomposing a Simple CBV Language

where T is a type describing the terms and $p_1, \ldots, p_n$ are the patterns describing the decompositions. We modify the SL compiler to check the unique-decomposition phrase before generating the interpreter. The implementation of automatic proofs follows the approach presented in Section 4 and Section 5.

The output of checking includes the information about the existence of decompositions for terms in the type specified in the `decomp` phrase. The information about uniqueness is shown only when the existence is satisfied. Counterexamples are given when either fails.

For the call-by-value λ-calculus example, we augment the SL specification in Figure 2 with two dynamic definitions for redexes and faulty terms, and a `decomp` phrase. The `decomp` phrase consists of the type `M` and alternative patterns for the four decompositions: a value, a demand of a variable in an evaluation context, a redex in an evaluation context, and a faulty term in an evaluation context. The resulting specification is in Figure 3.

The SL compiler checks the unique decomposition for the specification in Figure 3, and produces the following result:

```
All terms have decompositions.

Counterexamples to unique decomposition:

App((Num(_),  Var(_)))
App((App((Num(_),  Var(_))),  Num(_)))
... (* more counterexamples omitted *)
```

The counterexamples give us hints about how to modify the specification in order to obtain uniqueness. We notice that the problem of violating uniqueness originates from applying a number to another term. It can be fixed by restricting the evaluation in the right subterm of an application only when the left is a λ-abstraction. We change the evaluation context in the specification in Figure 3 as follows:

```
context E =  BOX | App(E, _) |  App(Lam(_,_), E);;
```

Unique decomposition holds in the new specification. The output is:

```
All terms have unique decompositions.
```

The SL system does not enumerate all counterexamples. We have tested our system on some small examples including the specification in Figure 3 and the specification for the example in the introduction. The SL system requires only few seconds for these examples. When we extend the datatype in the specification with more constructors, the execution time increases dramatically. For example, if we add an addition operation to the specification shown in Figure 3, proving the unique-decomposition lemma takes about two minutes; further extensions such as additional built-in types may require over an hour. Optimizing the algorithms and the implementation is one of our future goals.

7. Related Work

Much work has been done to develop systems for checking and building formal proofs. Influenced by the Automath project (de Bruijn, 1968), logical frameworks and theorem provers such as COQ (Dowek et al., 1993) and Twelf (Pfenning and Schürmann, 1999) are widely applied in specifying and proving properties of deductive systems. These frameworks can be used to prove existence of decompositions, which can be even done automatically in Twelf. However, contexts are not directly supported in these frameworks, and uniqueness is difficult to specify without the special built-in extensions provided by our system. The purposes of our small system are to provide a domain-specific language for specifying syntactic theories, to generate interpreters from specifications automatically, and to check certain properties of the specified syntactic theories automatically.

Tree automata are widely used in compilation, especially in pattern matching. In terms of pattern matching, equivalence of tree automata corresponds to exhaustiveness of patterns and non-ambiguity of regular tree grammars corresponds to non-overlapping of patterns. However, the patterns used in earlier research do not support dynamic values and contexts. Fähndrich and Boyland's work on abstract patterns is much closer to our work and we build on their results quite heavily (Fähndrich and Boyland, 1997).

8. Conclusion and Future Work

Checking existence and uniqueness of decomposition is important for specifying operational semantics using syntactic theories. Existence ensures that the evaluators consider all the cases for an input program; uniqueness guarantees that evaluators are deterministic at each step. We have introduced algorithms for automatically proving unique-decomposition lemmas. These algorithms are implemented in the SL system. A number of small examples have been tested in the system. The SL system is available from:

`http://www.cs.uoregon.edu/~ariola/SL/`.

We have made the first step towards integrating generating interpreters from syntactic theories and proving properties of the syntactic theories in the SL system. Currently, we are working on checking unique decomposition in the presence of side conditions in patterns (*e.g.*, equality and inequality constraints). In general, these may again lead to undecidability. We are also investigating other properties that could be automatically proved, as well as optimizing our implementation.

Acknowledgements

We would like to thank Michel Mauny, Xavier Leroy and Didier Rémy of the Cristal Group in INRIA for their contribution to the earlier version of the SL system. We are also deeply indebted to Miley Semmelroth, editors Carolyn Talcott and Olivier Danvy, and the anonymous reviewers, whose comments on this paper lead to substantial improvements in its presentation.

References

Aho, A., R. Sethi, and J. Ullman: 1985, *Compilers—Principles, Techniques, and Tools*. Reading, Mass.: Addison-Wesley.

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

Barendregt, H. P.: 1984, *The Lambda Calculus: Its Syntax and Semantics*, Vol. 103 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science Publishers B.V., Amsterdam, revised edition.

Comon, H., M. Dauchet, R. Gilleron, F. Jacquemart, D. Lugiez, S. Tison, and M. Tommasi: 1999, 'Tree Automata Techniques and Applications'. Book draft available at http://www.grappa.univ-lille3.fr/tata.

de Bruijn, N.: 1968, 'The Mathematical Language AUTOMATH, Its Usage, and Some of Its Extensions'. In: M. Laudet (ed.): *Proceedings of the Symposium on Automatic Demonstration*. Versailles, France, pp. 29–61, Springer-Verlag LNM 125.

Dowek, G., A. Felty, H. Herbelin, G. Huet, C. Murthy, C. Parent, C. Paulin-Mohring, and B. Werner: 1993, 'The Coq Proof Assistant User's Guide'. Technical Report Rapport Techniques 154, INRIA, Rocquencourt, France. Version 5.8.

Fähndrich, M. and J. Boyland: 1997, 'Statically Checkable Pattern Abstractions'. In: the *ACM SIGPLAN International Conference on Functional Programming*. pp. 75–84, ACM Press, New York.

Felleisen, M.: 1987, 'The Calculi of Lambda-v-CS-Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages'. Ph.D. thesis, Indiana University.

Fiskio-Lasseter, J. and A. Sabry: 1999, 'Putting Operational Techniques to the Test: A Syntactic Theory for Behavioral Verilog'. *Electronic Notes in Theoretical Computer Science* **26**, 32–49. Third International Workshop on Higher Order Operational Techniques in Semantics.

Gécseg, F. and M. Steinby: 1984, *Tree Automata*. Budapest: Akadémiai Kiadó.

Hopcroft, J. and J. Ullman: 1979, *Introduction to Automata Theory, Languages, and Computation*, Computer Science. Addison-Wesley.

Launchbury, J. and A. Sabry: 1997, 'Monadic State: Axiomatization and Type Safety'. In: the *ACM SIGPLAN International Conference on Functional Programming*. pp. 227–238, ACM Press, New York.

Mason, I. A.: 1999, 'Computing with Contexts'. *Higher-Order and Symbolic Computation* **12**(2), 171–201.

Pfenning, F. and C. Schürmann: 1999, 'System Description: Twelf - A Meta-Logical Framework for Deductive Systems'. In: H. Ganzinger (ed.): *Proceedings of the 16th International Conference on Automated Deduction*. Trento, Italy, pp. 202–206, Springer-Verlag, Berlin.

Semmelroth, M. and A. Sabry: 1999, 'Monadic Encapsulation in ML'. In: the *ACM SIGPLAN International Conference on Functional Programming*. pp. 8–17, ACM Press, New York.

Xiao, Y., Z. M. Ariola, and M. Mauny: 2000, 'From Syntactic Theories to Interpreters: A Specification Language and Its Compiler'. In: N. Derschowitz and C. Kirchner (eds.): *Proceedings of the First International Workshop on Rule-based Programming*. Montreal, Canada, pp. 1–16.